AMITY SCHOOL OF ENGINEERING & TECHNOLOGY

AMITY UNIVERSITY, MANESAR GURUGRAM

CRYPTOGRAPHY & NETWORK SECURITY LAB FILE

SUBMITTED TO:

Dr. Khusboo Tripathi

SUBMITTED BY:

Aayush Verma
A50105220004
BTECH CSE A, 7th SEM (20-24)

Index

S. No	Topic	Date	Remarks	Teacher's Signature
1.	Write a program to implement Caeser Cipher.	24/08/2023		
2.	Write a program to implement Monoalphabetic Cipher.			
3.	Write a program to implement Playfair Cipher.	21/09/2023		
4.	Write a program to implement Railfence Cipher.			
5.	Write a program to implement Columnar Transposition Cipher.	05/10/2023		
6.	Write a program to implement Polyalphabetic Cipher.			
7.	Write a program to implement Vigenère Cipher.	12/10/2023		
8.	Write a program to implement Vernam Cipher.			

9.	Write a program to implement DES Algorithm.	19/10/2023		
10.	Write a program to implement AES Algorithm.			
11.	Write a program to implement RSA Algorithm.	26/10/2023		
12.	Write a program to implement Digital Signature Algorithm.			

Practical-1

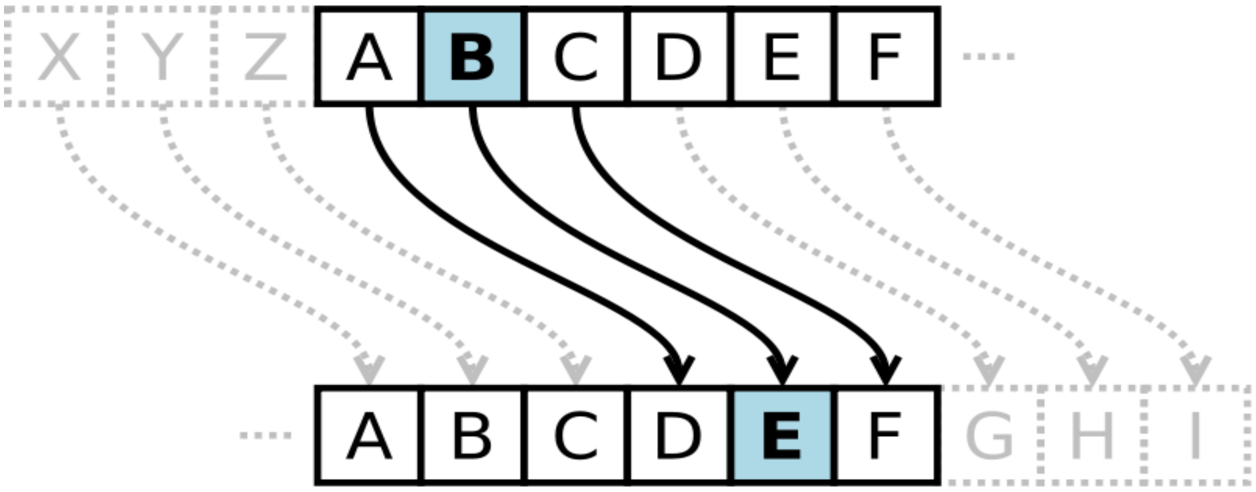
Aim: Write a program to implement Caeser Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Caesar cipher is a simple encryption technique that was used by Julius Caesar to send secret messages to his allies. It works by shifting the letters in the plaintext message by a certain number of positions, known as the "shift" or "key". The Caesar Cipher technique is one of the earliest and simplest methods of encryption technique. It's simply a type of substitution cipher, i.e., each letter of a given text is replaced by a letter with a fixed number of positions down the alphabet. The method is apparently named after Julius Caesar, who apparently used it to communicate with his officials. 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.

Caeser Cipher is easy to implement and use thus, making suitable for beginners to learn about encryption. It also requires only a small set of pre-shared information and can be modified easily to create a more secure variant.

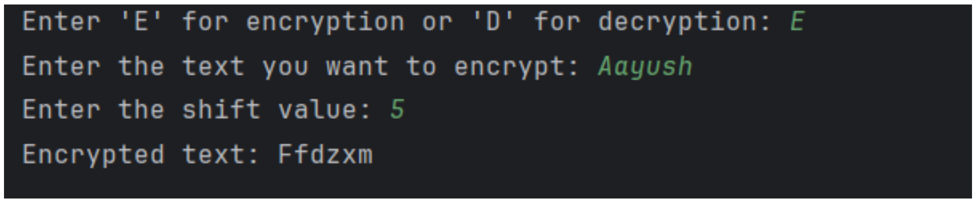
For example, if the shift is 3, then the letter A would be replaced by the letter D, B would become E, C would become F, and so on. The alphabet is wrapped around so that after Z, it starts back at A.



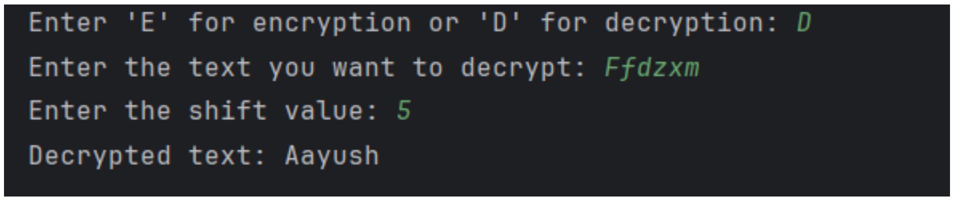
Source Code:

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

    for char in text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            shifted_char = chr(((ord(char) - 97 + shift) % 26) + 97)
            if is_upper:
                shifted_char = shifted_char.upper()
            encrypted_text += shifted_char
        else:
            encrypted_text += char

    return encrypted_text

def caesar_decipher(encrypted_text, shift):
    decrypted_text = ""

    for char in encrypted_text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            shifted_char = chr(((ord(char) - 97 - shift) % 26) + 97)
            if is_upper:
                shifted_char = shifted_char.upper()
            decrypted_text += shifted_char
        else:
            decrypted_text += char

    return decrypted_text

def main():
    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()

    if choice == 'E':
        text = input("Enter the text you want to encrypt: ")
        shift = int(input("Enter the shift value: "))
        encrypted_text = caesar_cipher(text, shift)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        encrypted_text = input("Enter the text you want to decrypt: ")
        shift = int(input("Enter the shift value: "))
        decrypted_text = caesar_decipher(encrypted_text, shift)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter 'E' for encryption or 'D' for decryption: E
Enter the text you want to encrypt: Aayush
Enter the shift value: 5
Encrypted text: Ffdzxm
```



```
Enter 'E' for encryption or 'D' for decryption: D
Enter the text you want to decrypt: Ffdzxm
Enter the shift value: 5
Decrypted text: Aayush
```

Practical-2

Aim: Write a program to implement Monoalphabetic Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: A monoalphabetic cipher is a simple form of substitution cipher in which each letter in the plaintext is consistently replaced by a corresponding letter in the ciphertext. In other words, it involves a one-to-one mapping of letters from the plaintext alphabet to the ciphertext alphabet. Here's how a monoalphabetic cipher works:

1. **Key:** The key for a monoalphabetic cipher is a mapping of each letter in the plaintext alphabet to a unique letter in the ciphertext alphabet. This mapping is typically defined by the person creating the cipher. For example, the letter 'A' in the plaintext alphabet may be consistently replaced by 'X' in the ciphertext alphabet, 'B' by 'Q,' 'C' by 'P,' and so on.
2. **Encryption:** To encrypt a message using a monoalphabetic cipher, you simply replace each letter in the plaintext with the corresponding letter in the ciphertext according to the key. Spaces and other non-alphabetic characters are usually left unchanged.
3. **Decryption:** To decrypt a message that has been encrypted using a monoalphabetic cipher, you reverse the process by replacing each letter in the ciphertext with its corresponding letter in the plaintext according to the same key.

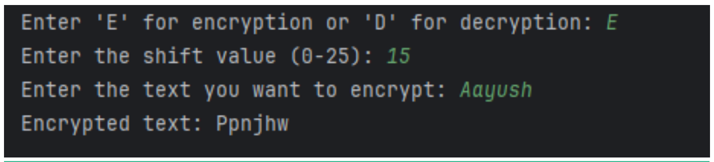
While monoalphabetic ciphers are easy to understand and use, they are not secure against modern cryptanalysis methods. They are highly susceptible to frequency analysis because the frequency of letters in the English language is well-known, and this information can be used to crack the cipher. Therefore, monoalphabetic ciphers are not considered secure for protecting sensitive information, and they are primarily used for educational purposes and as a historical example of early cryptographic techniques.

Source Code:

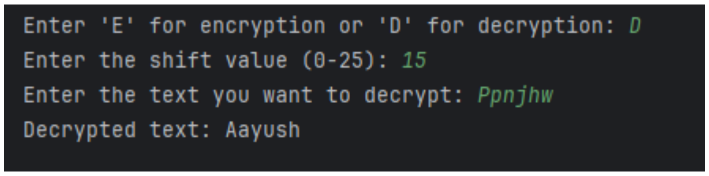
```
import string
def create_monoalphabetic_key(shift):
    alphabet = string.ascii_lowercase
    shifted_alphabet = alphabet[shift:] + alphabet[:shift]
    key = {}
    for letter, shifted_letter in zip(alphabet, shifted_alphabet):
        key[letter] = shifted_letter
    return key

def monoalphabetic_cipher(text, key):
    encrypted_text = ""
    for char in text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            shifted_char = key[char]
            if is_upper:
                shifted_char = shifted_char.upper()
            encrypted_text += shifted_char
        else:
            encrypted_text += char
    return encrypted_text
```

```
def monoalphabetic_decipher(encrypted_text, key):
    decrypted_text = ""
    reverse_key = {v: k for k, v in key.items()}
    for char in encrypted_text:
        if char.isalpha():
            is_upper = char.isupper()
            char = char.lower()
            shifted_char = reverse_key[char]
            if is_upper:
                shifted_char = shifted_char.upper()
            decrypted_text += shifted_char
        else:
            decrypted_text += char
    return decrypted_text

def main():
    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()

    shift = int(input("Enter the shift value (0-25): "))
    if shift < 0 or shift > 25:
        print("Shift value must be between 0 and 25.")
        return
    key = create_monoalphabetic_key(shift)

    if choice == 'E':
        text = input("Enter the text you want to encrypt: ")
        encrypted_text = monoalphabetic_cipher(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        encrypted_text = input("Enter the text you want to decrypt: ")
        decrypted_text = monoalphabetic_decipher(encrypted_text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")

if __name__ == "__main__":
    main()
```

Output:



```
Enter 'E' for encryption or 'D' for decryption: E
Enter the shift value (0-25): 15
Enter the text you want to encrypt: Aayush
Encrypted text: Ppnjhw
```



```
Enter 'E' for encryption or 'D' for decryption: D
Enter the shift value (0-25): 15
Enter the text you want to decrypt: Ppnjhw
Decrypted text: Aayush
```

Practical-3

Aim: Write a program to implement Playfair Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Playfair cipher is a classical symmetric encryption technique used to encrypt or encode text. It was invented by Charles Wheatstone in 1854 but was later popularized by Lyon Playfair. The Playfair cipher is a simple substitution cipher that operates on pairs of letters. It's relatively easy to understand and implement, making it a popular choice for educational purposes, but it's not considered secure for modern cryptographic purposes.

The Playfair cipher is relatively easy to understand and implement. It doesn't require complex mathematical operations or expensive equipment, making it accessible for educational purposes and early cryptographic exploration. Unlike simple substitution ciphers like the Caesar cipher, the Playfair cipher doesn't preserve the frequency of individual letters in the plaintext, making it somewhat resistant to basic frequency analysis attacks.

The Playfair cipher provides only limited security. While it may thwart casual eavesdroppers, it is susceptible to various attacks, especially the known-plaintext attack and the chosen-plaintext attack. These vulnerabilities make it unsuitable for modern cryptographic requirements. The creation and management of the key table can be challenging, as it requires choosing a suitable keyword and constructing the table correctly. Furthermore, the key table is relatively small (5x5), which means there are only 25 different substitutions possible, making it easier to break using more advanced cryptographic techniques.

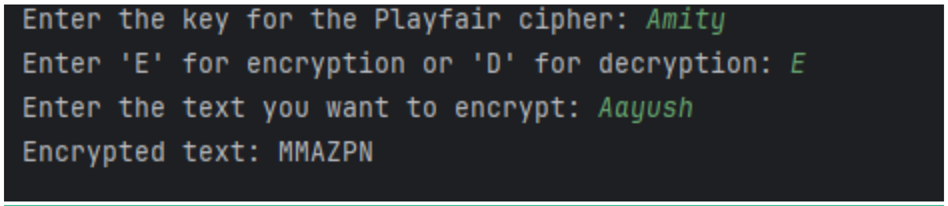
In summary, the Playfair cipher is a historical cipher with a simple design that offers some security advantages over basic ciphers, but it falls short of meeting the security requirements of modern encryption. Its vulnerabilities and key management challenges make it unsuitable for secure communication in contemporary settings, and it should primarily be seen as a piece of cryptographic history.

Source Code:

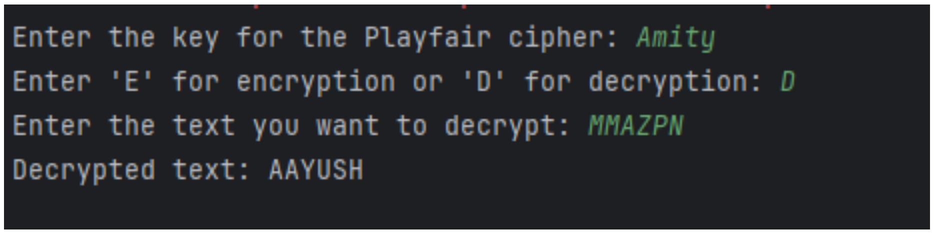
```
def prepare_text(text):

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

        text = text.replace("J", "I")

    return text


def create_playfair_matrix(key):

    key = prepare_text(key)

    alphabet = "ABCDEFGHIKLMNOPQRSTUVWXYZ"

    matrix = [[" for _ in range(5)] for _ in range(5)]

    key_set = set()


    row, col = 0, 0

    for char in key + alphabet:

        if char not in key_set:

            matrix[row][col] = char

            key_set.add(char)

            col += 1

            if col == 5:

                col = 0

                row += 1


    return matrix
```

```
def find_coordinates(matrix, char):
    for i in range(5):
        for j in range(5):
            if matrix[i][j] == char:
                return i, j


def playfair_encrypt(plaintext, matrix):
    ciphertext = ""
    plaintext = prepare_text(plaintext)

    for i in range(0, len(plaintext), 2):
        pair = plaintext[i:i+2]

        if len(pair) == 2:
            char1, char2 = pair[0], pair[1]
            row1, col1 = find_coordinates(matrix, char1)
            row2, col2 = find_coordinates(matrix, char2)

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

    return ciphertext


def playfair_decrypt(ciphertext, matrix):
    plaintext = ""
    ciphertext = prepare_text(ciphertext)

    for i in range(0, len(ciphertext), 2):
        pair = ciphertext[i:i+2]

        if len(pair) == 2:
            char1, char2 = pair[0], pair[1]
            row1, col1 = find_coordinates(matrix, char1)
            row2, col2 = find_coordinates(matrix, char2)

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

    return plaintext

def main():
    key = input("Enter the key for the Playfair cipher: ")
    matrix = create_playfair_matrix(key)

    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()

    if choice == 'E':
        plaintext = input("Enter the text you want to encrypt: ")
        ciphertext = playfair_encrypt(plaintext, matrix)
        print("Encrypted text:", ciphertext)
    elif choice == 'D':
        ciphertext = input("Enter the text you want to decrypt: ")
        plaintext = playfair_decrypt(ciphertext, matrix)
        print("Decrypted text:", plaintext)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter the key for the Playfair cipher: Amity
Enter 'E' for encryption or 'D' for decryption: E
Enter the text you want to encrypt: Aayush
Encrypted text: MMAZPN
```



```
Enter the key for the Playfair cipher: Amity
Enter 'E' for encryption or 'D' for decryption: D
Enter the text you want to decrypt: MMAZPN
Decrypted text: AAYUSH
```

Practical-4

Aim: Write a program to implement Railfence Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The rail fence technique is a simple form of transposition cipher. It's a type of cryptographic algorithm that rearranges the positions of the letters in a message to create a new, seemingly unrelated message. The technique gets its name from the way we write the message. Applying the rail fence technique in each text results in a zigzag pattern, with each letter in a row written out before moving to the next row.

To encrypt a message using the rail fence technique, first, we need to write the message in the first row of a table. Furthermore, we must write the second letter of the message in the second row. We need to continue this process until we've written all the letters in the message. Finally, to generate the encrypted message, we read the table row-wise.

To decrypt an encrypted message, the first step is to determine the number of rows in the table based on the length of the encrypted message. Furthermore, we need to write the first letter of the encrypted message in the first row, the second letter in the second row, and so on. We need to continue this process until we've written all the letters in the message.

The rail fence technique is relatively simple to understand and implement, making it a good choice for basic communication where a higher level of security is not required. We can manually perform the rail fence technique without requiring specialized equipment or software.

Overall, the rail fence technique is a relatively simple form of encryption. However, it doesn't provide strong security. Someone can easily break it with even a basic understanding of cryptography. Although, it can still be helpful for simple communication where a high level of security is not required.

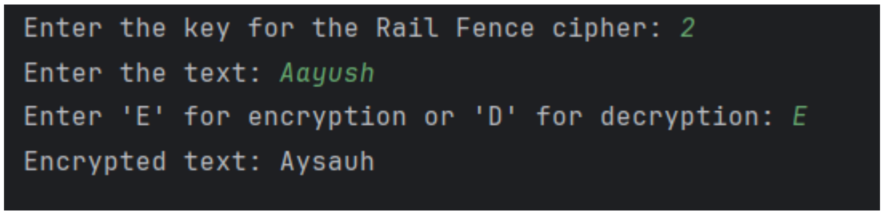
Source Code:

```
def railfence_encrypt(text, key):
    text = text.replace(" ", "")
    fence = ['' for _ in range(key)]
    direction = 1  # 1 for down, -1 for up
    row = 0

    for char in text:
        fence[row] += char

        if row == 0:
            direction = 1
        elif row == key - 1:
            direction = -1

        row += direction

    encrypted_text = ''.join(fence)
    return encrypted_text

def railfence_decrypt(text, key):
    text_length = len(text)
    fence = ['' for _ in range(key)]
```

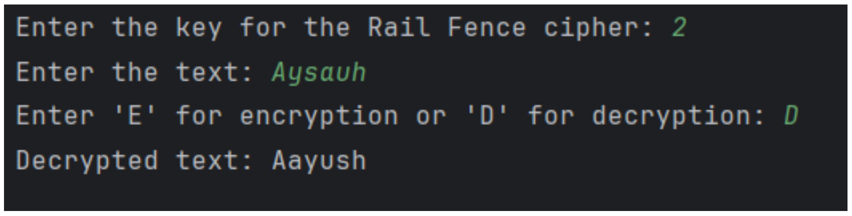
```
direction = 1
row = 0

for i in range(text_length):
    fence[row] += ' '


    if row == 0:
        direction = 1
    elif row == key - 1:
        direction = -1


    row += direction

index = 0


for i in range(key):
    for j in range(len(fence[i])):
        if fence[i][j] == ' ':
            fence[i] = fence[i][:j] + text[index] + fence[i][j + 1:]
            index += 1

direction = 1
row = 0
```

```
    decrypted_text = ''

    for _ in range(text_length):
        decrypted_text += fence[row][0]
        fence[row] = fence[row][1:]

        if row == 0:
            direction = 1
        elif row == key - 1:
            direction = -1

        row += direction

    return decrypted_text

def main():
    key = int(input("Enter the key for the Rail Fence cipher: "))
    text = input("Enter the text: ")

    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()

    if choice == 'E':
        encrypted_text = railfence_encrypt(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        decrypted_text = railfence_decrypt(text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter the key for the Rail Fence cipher: 2
Enter the text: Aayush
Enter 'E' for encryption or 'D' for decryption: E
Encrypted text: Aysauh
```



```
Enter the key for the Rail Fence cipher: 2
Enter the text: Aysauh
Enter 'E' for encryption or 'D' for decryption: D
Decrypted text: Aayush
```

Practical-5

Aim: Write a program to implement Columnar Transposition Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: A Columnar Transposition Cipher is a method of encryption that rearranges the characters in a message to create a ciphertext. To use it, you select a keyword or phrase, known as the "key." The key determines the number of rows used to write the plaintext message, with each character of the message written horizontally beneath the key in these rows. Afterward, the key is sorted alphabetically to determine the order in which the rows are read. The ciphertext is then created by reading the reordered rows vertically, column by column.

Decrypting a message encrypted using the Columnar Transposition Cipher involves reversing this process. You start by selecting the same key, sorting it, and then rearranging the columns in the original order to reveal the plaintext. However, it's important to understand that this method is relatively simple and not suitable for modern security needs, as it can be easily broken with advanced cryptographic techniques. In contemporary cryptography, more secure encryption methods are used to safeguard sensitive information.

It offers simplicity in its encryption process, making it easy to implement and understand for basic encoding needs. However, it is no longer considered a secure encryption method and is vulnerable to modern cryptographic attacks, so it's typically used for educational or recreational purposes rather than for secure communications.

The Columnar Transposition Cipher has several disadvantages. It offers relatively weak security as it is vulnerable to modern cryptanalysis techniques, such as frequency analysis and brute force attacks, making it inadequate for securing sensitive information in contemporary cryptography.

Despite the difference between transposition and substitution operations, they are often combined, as in historical ciphers like the ADFGVX cipher or complex high-quality encryption methods like the modern Advanced Encryption Standard (AES).

Source Code:

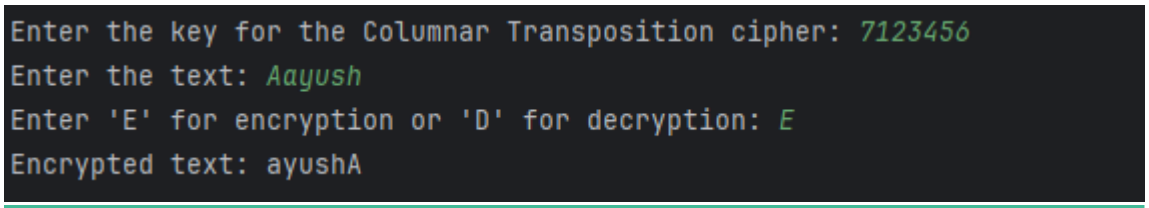
```
def columnar_transposition_encrypt(text, key):
    key_length = len(key)
    text_length = len(text)
    num_rows = (text_length + key_length - 1) // key_length

    grid = [['' for _ in range(key_length)] for _ in range(num_rows)]
    index = 0
    for col in range(key_length):
        for row in range(num_rows):
            if index < text_length:
                grid[row][col] = text[index]
                index += 1

    encrypted_text = ''
    for col in range(key_length):
        col_index = key.index(col + 1)
        for row in range(num_rows):
            encrypted_text += grid[row][col_index]

    return encrypted_text
```

```
def columnar_transposition_decrypt(text, key):
    key_length = len(key)
    text_length = len(text)
    num_rows = (text_length + key_length - 1) // key_length

    last_row_cols = key_length - (num_rows * key_length - text_length)
    grid = [['' for _ in range(key_length)] for _ in range(num_rows)]

    full_cols = num_rows - last_row_cols
    partial_cols = key_length - full_cols

    col_counts = [full_cols] * partial_cols + [last_row_cols] * full_cols

    segment_start = [sum(col_counts[:i]) for i in range(key_length)]
    segment_end = [segment_start[i] + col_counts[i] for i in range(key_length)]

    index = 0
    for col in range(key_length):
        for row in range(num_rows):
            start, end = segment_start[col], segment_end[col]
            if start <= row < end:
                grid[row][col] = text[index]
                index += 1
```

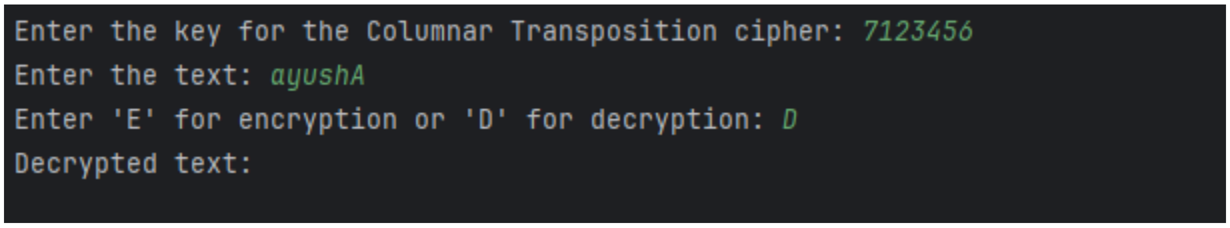
```
   decrypted_text = ''
    for row in range(num_rows):
        for col in range(key_length):
            decrypted_text += grid[row][key[col] - 1]

    return decrypted_text

def main():
    key = input("Enter the key for the Columnar Transposition cipher: ")
    key = [int(k) for k in key]

    text = input("Enter the text: ")

    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()

    if choice == 'E':
        encrypted_text = columnar_transposition_encrypt(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        decrypted_text = columnar_transposition_decrypt(text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")
```

```
if __name__ == "__main__":
    main()
```

Output:



```
Enter the key for the Columnar Transposition cipher: 7123456
Enter the text: Aayush
Enter 'E' for encryption or 'D' for decryption: E
Encrypted text: ayushA
```



```
Enter the key for the Columnar Transposition cipher: 7123456
Enter the text: ayushA
Enter 'E' for encryption or 'D' for decryption: D
Decrypted text:
```

Practical-6

Aim: Write a program to implement Polyalphabetic Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: A poly-alphabetic cipher is any cipher based on substitution, using several substitution alphabets. In polyalphabetic substitution ciphers, the plaintext letters are enciphered differently based upon their installation in the text. Rather than being a one-to-one correspondence, there is a one-to-many relationship between each letter and its substitutes.

For example, ‘a’ can be enciphered as ‘d’ in the starting of the text, but as ‘n’ at the middle. The polyalphabetic ciphers have the benefit of hiding the letter frequency of the basic language. Therefore, the attacker cannot use individual letter frequency static to divide the ciphertext.

The first Polyalphabetic cipher was the Alberti Cipher which was introduced by Leon Battista Alberti in the year 1467. It used a random alphabet to encrypt the plaintext, but at different points and it can change to a different mixed alphabet, denoting the change with an uppercase letter in the cipher text.

It can utilize this cipher, Alberti used a cipher disc to display how plaintext letters are associated to cipher text letters. In this cipher, each ciphertext character is based on both the corresponding plaintext character and the position of the plaintext character in the message.

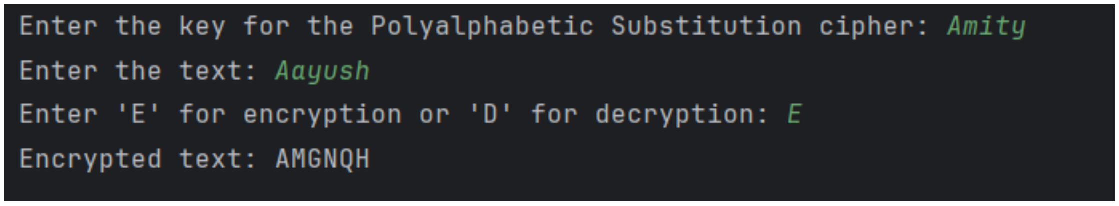
As the name polyalphabetic recommend this is achieved by using multiple keys rather than only one key. This implies that the key should be a stream of subkeys, in which each subkey depends somehow on the position of the plaintext character that needs subkey for encipherment.

Encryption is implemented by going to the row in the table correlating to the key and discover the column heading the corresponding letter of the plaintext character; the letter at the intersection of corresponding row and column of the Vigenère Square create the ciphertext character. The rest of the plaintext is encrypted in a similar method.

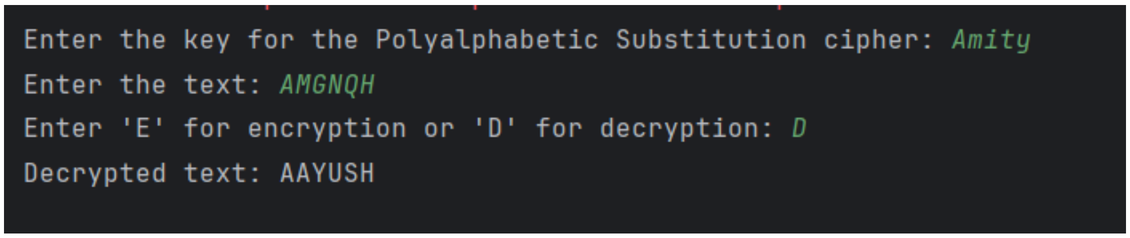
Source Code:

```
import string
def prepare_key(key, text_length):
    key = key.upper()
    key_length = len(key)

    if key_length >= text_length:
        return key[:text_length]
    else:
        repeats = text_length // key_length
        remainder = text_length % key_length
        prepared_key = (key * repeats) + key[:remainder]
        return prepared_key

def polyalphabetic_encrypt(text, key):
    text = text.upper()
    key = prepare_key(key, len(text))
    encrypted_text = ''

    for i in range(len(text)):
        if text[i] in string.ascii_uppercase:
            shift = ord(key[i]) - ord('A')
            encrypted_char = chr(((ord(text[i]) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text += encrypted_char
        else:
            encrypted_text += text[i]

    return encrypted_text

def polyalphabetic_decrypt(text, key):
    key = prepare_key(key, len(text))
    decrypted_text = ''

    for i in range(len(text)):
        if text[i] in string.ascii_uppercase:
            shift = ord(key[i]) - ord('A')
            decrypted_char = chr(((ord(text[i]) - ord('A') - shift) % 26) + ord('A'))
            decrypted_text += decrypted_char
        else:
            decrypted_text += text[i]

    return decrypted_text

def main():
    key = input("Enter the key for the Polyalphabetic Substitution cipher: ")
    text = input("Enter the text: ")
    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()


    if choice == 'E':
        encrypted_text = polyalphabetic_encrypt(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        decrypted_text = polyalphabetic_decrypt(text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter the key for the Polyalphabetic Substitution cipher: Amity
Enter the text: Aayush
Enter 'E' for encryption or 'D' for decryption: E
Encrypted text: AMGNQH
```



```
Enter the key for the Polyalphabetic Substitution cipher: Amity
Enter the text: AMGNQH
Enter 'E' for encryption or 'D' for decryption: D
Decrypted text: AAYUSH
```

Practical-7

Aim: Write a program to implement Vigenère Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Vigenère cipher is a method for encrypting text by using a keyword to determine the shifting pattern for each letter in plaintext. It's a type of polyalphabetic substitution cipher, meaning it uses multiple substitution alphabets to encode the text. This cipher was first described by Giovan Battista Bellaso in 1553 and gained notoriety for its resistance to decryption attempts until 1863, making it known as "the indecipherable cipher."

In the Vigenère cipher, you have a plaintext message and a key that consists of letters. Each letter in the plaintext is shifted by a corresponding letter from the key using Caesar ciphers. The Caesar cipher is a simple substitution cipher where each letter is shifted a fixed number of positions down the alphabet. In the Vigenère cipher, this shifting number varies based on the corresponding key letter. This makes it more complex and challenging to break compared to a standard Caesar cipher, where the shifting number is constant.

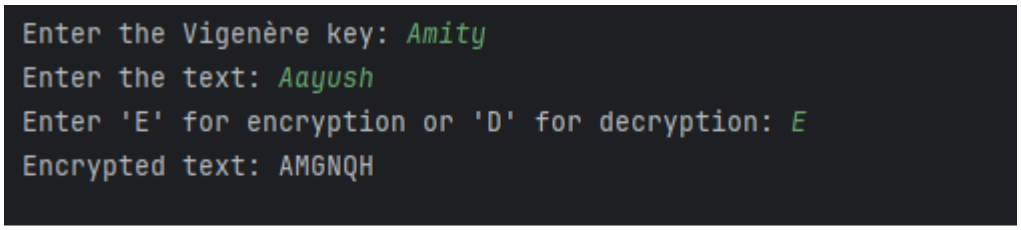
The strength of the Vigenère cipher lies in its ability to disguise the frequency of letters in the plaintext, which would otherwise be vulnerable to frequency analysis. Frequency analysis is a common technique in cryptanalysis that relies on the fact that certain letters occur more frequently than others in most languages. The Vigenère cipher's use of a changing key for each letter makes it difficult to detect these patterns.

However, the main weakness of the Vigenère cipher is that the key repeats. If a cryptanalyst correctly guesses the length of the key, they can treat the ciphertext as a series of interwoven Caesar ciphers, each with its own key. This significantly weakens security. Cryptanalysts can determine the key length through methods like brute force testing or the Kasiski examination. The Friedman test is also used to help figure out the key length, making it more susceptible to decryption once the key length is known.

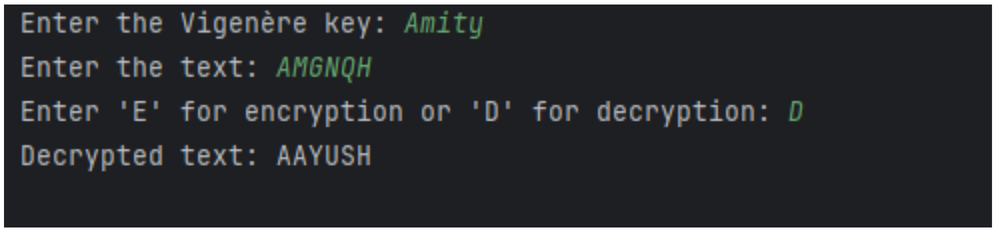
Source Code:

```
import string
def prepare_key(key, text_length):
    key = key.upper()
    key_length = len(key)

    if key_length >= text_length:
        return key[:text_length]
    else:
        repeats = text_length // key_length
        remainder = text_length % key_length
        prepared_key = (key * repeats) + key[:remainder]
        return prepared_key

def vigenere_encrypt(text, key):
    text = text.upper()
    key = prepare_key(key, len(text))
    encrypted_text = ''

    for i in range(len(text)):
        if text[i] in string.ascii_uppercase:
            shift = ord(key[i]) - ord('A')
            encrypted_char = chr(((ord(text[i]) - ord('A') + shift) % 26) + ord('A'))
            encrypted_text += encrypted_char
        else:
            encrypted_text += text[i]

    return encrypted_text

def vigenere_decrypt(text, key):
    key = prepare_key(key, len(text))
    decrypted_text = ''

    for i in range(len(text)):
        if text[i] in string.ascii_uppercase:
            shift = ord(key[i]) - ord('A')
            decrypted_char = chr(((ord(text[i]) - ord('A') - shift) % 26) + ord('A'))
            decrypted_text += decrypted_char
        else:
            decrypted_text += text[i]

    return decrypted_text

def main():
    key = input("Enter the Vigenère key: ")
    text = input("Enter the text: ")
    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()


    if choice == 'E':
        encrypted_text = vigenere_encrypt(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        decrypted_text = vigenere_decrypt(text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter the Vigenère key: Amity
Enter the text: Aayush
Enter 'E' for encryption or 'D' for decryption: E
Encrypted text: AMGNQH
```



```
Enter the Vigenère key: Amity
Enter the text: AMGNQH
Enter 'E' for encryption or 'D' for decryption: D
Decrypted text: AAYUSH
```

Practical-8

Aim: Write a program to implement Vernam Cipher.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Vernam cipher is an encryption technique that operates at the binary level, and while it is relatively simple and not widely used due to its susceptibility to external attacks, it remains historically significant as one of the earliest encryption methods, akin to the Caesar cipher. In the Vernam cipher, both the plaintext and a key string are considered in their binary form. The key must have a length less than or equal to that of the plaintext and follows the principles of symmetric-key cryptography. This cipher is categorized as a type of poly-alphabetic cipher, which is a part of substitution ciphers.

The Vernam cipher encryption process begins by converting both the plaintext and the key string into binary form. The key can be in either string or binary format. The key is repetitively extended to match the length of the plaintext. Then, an exclusive OR (XOR) operation is performed between the binary elements of the plaintext and the corresponding elements of the key string at the same positions. This XOR operation is the basis of the encryption process, transforming the plaintext into ciphertext.

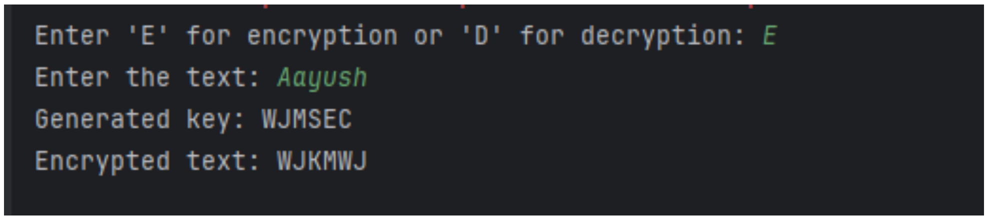
The decryption process in the Vernam cipher mirrors the encryption process, with the ciphertext being converted back into plaintext by applying the same XOR operation between the ciphertext and the key. This method of encryption and decryption, while conceptually straightforward, underscores the historical significance of the Vernam cipher in the development of cryptographic techniques.

In conclusion, the Vernam cipher, with its binary-level encryption and symmetric-key approach, represents a pivotal milestone in the history of cryptography. While its vulnerability to modern cryptographic attacks has limited its practical use, its pioneering role in encryption's evolution and its status as one of the early encryption techniques continue to make it a noteworthy chapter in the story of secure communication.

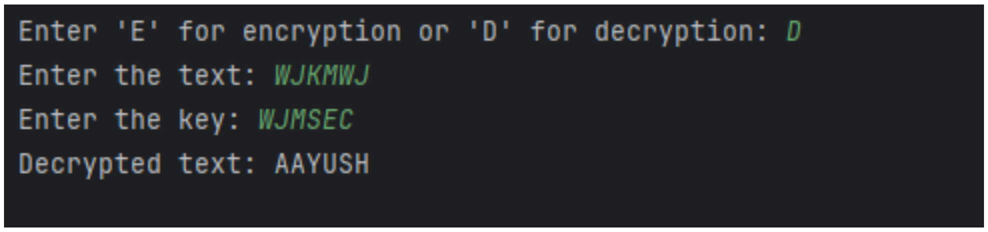
Source Code:

```
import random


def generate_random_key(length):
    key = ''
    for _ in range(length):
        key += random.choice('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    return key


def vernam_encrypt(text, key):
    text = text.upper()
    key = key.upper()
    if len(text) != len(key):
        return "Key length must match the text length for encryption."
    encrypted_text = ''

    for i in range(len(text)):
        if text[i] in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            shift = (ord(text[i]) + ord(key[i]) - 2 * ord('A')) % 26
            encrypted_char = chr(shift + ord('A'))
            encrypted_text += encrypted_char
        else:
            encrypted_text += text[i]
    return encrypted_text


def vernam_decrypt(text, key):
    text = text.upper()
    key = key.upper()
    if len(text) != len(key):
        return "Key length must match the text length for decryption."
    decrypted_text = ''
    for i in range(len(text)):
        if text[i] in 'ABCDEFGHIJKLMNOPQRSTUVWXYZ':
            shift = (ord(text[i]) - ord(key[i]) + 26) % 26
            decrypted_char = chr(shift + ord('A'))
            decrypted_text += decrypted_char
        else:
            decrypted_text += text[i]

    return decrypted_text


def main():
    choice = input("Enter 'E' for encryption or 'D' for decryption: ").upper()


if choice == 'E':
        text = input("Enter the text: ")
        key = generate_random_key(len(text))
        print("Generated key:", key)
        encrypted_text = vernam_encrypt(text, key)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        text = input("Enter the text: ")
        key = input("Enter the key: ")
        decrypted_text = vernam_decrypt(text, key)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")


if __name__ == "__main__":
    main()
```

Output:



```
Enter 'E' for encryption or 'D' for decryption: E
Enter the text: Aayush
Generated key: WJMSEC
Encrypted text: WJKMWJ
```



```
Enter 'E' for encryption or 'D' for decryption: D
Enter the text: WJKMWJ
Enter the key: WJMSEC
Decrypted text: AAYUSH
```

Practical-9

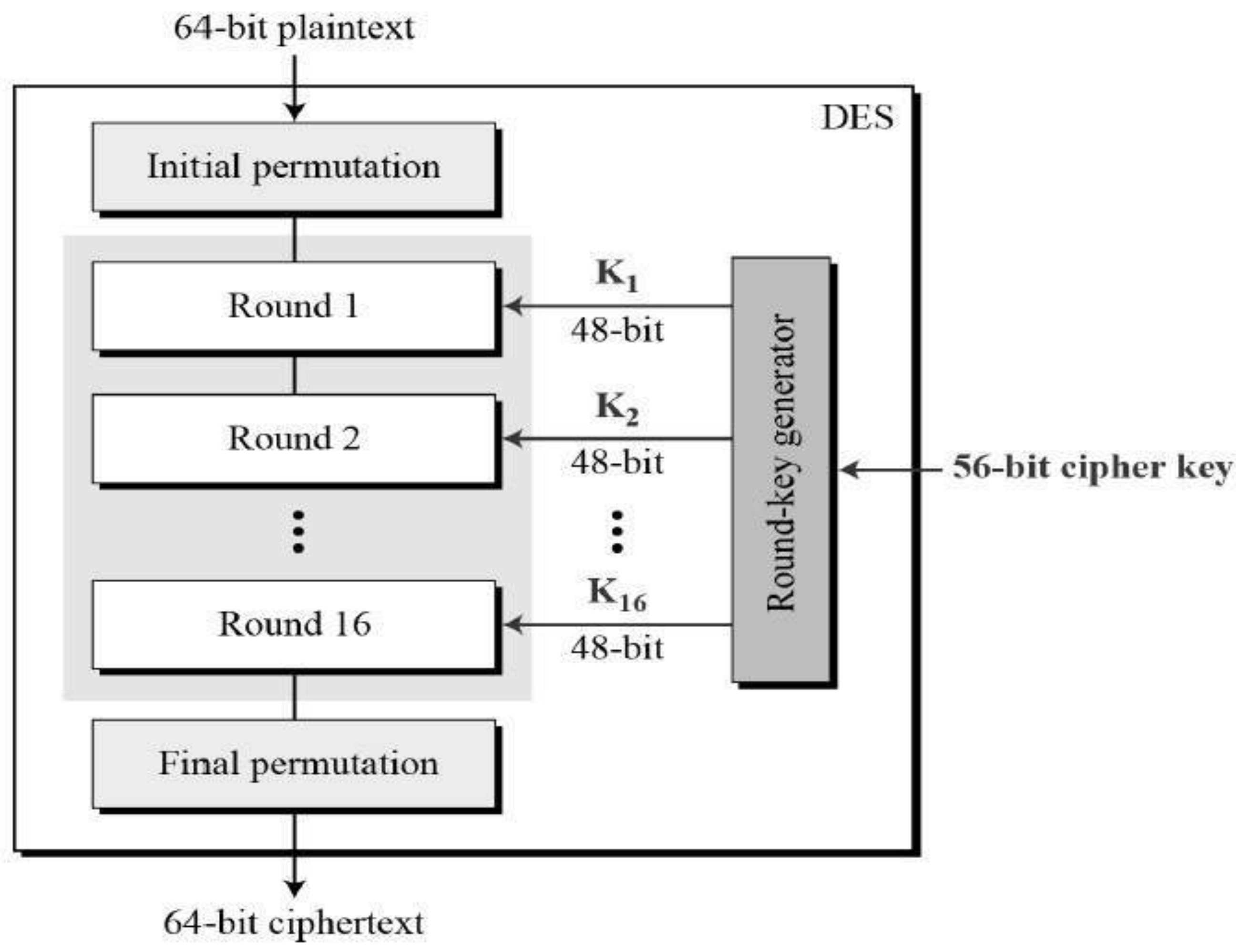
Aim: Write a program to implement DES Algorithm.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Data Encryption Standard (DES) is a symmetric-key block cipher published by the National Institute of Standards and Technology (NIST).

DES is an implementation of a Feistel Cipher. It uses a 16 round Feistel structure. The block size is 64-bit. Though, key length is 64-bit, DES has an effective key length of 56 bits, since 8 of the 64 bits of the key are not used by the encryption algorithm (function as check bits only).

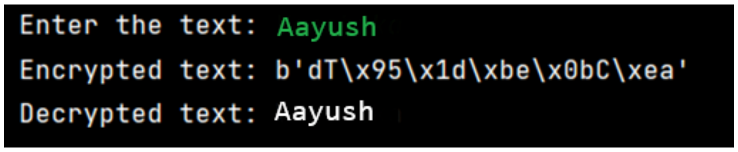
General structure of DES is depicted in the following illustration:



Source Code:

```
from Crypto.Cipher import DES
from Crypto.Random import get_random_bytes


def pad(text):
    while len(text) % 8 != 0:
        text += b' '
    return text


def des_encrypt(plaintext, key):
    cipher = DES.new(key, DES.MODE_ECB)
    padded_text = pad(plaintext)
    encrypted_text = cipher.encrypt(padded_text)
    return encrypted_text


def des_decrypt(encrypted_text, key):
    cipher = DES.new(key, DES.MODE_ECB)
    decrypted_text = cipher.decrypt(encrypted_text)
    return decrypted_text.rstrip()


def main():
    key = get_random_bytes(8)
    text = input("Enter the text: ").encode()
    encrypted_text = des_encrypt(text, key)
    print("Encrypted text:", encrypted_text)

    decrypted_text = des_decrypt(encrypted_text, key)
    print("Decrypted text:", decrypted_text.decode())

if __name__ == "__main__":
    main()
```

Output:



```
Enter the text: Aayush
Encrypted text: b'dT\x95\x1d\xbe\x0bC\xea'
Decrypted text: Aayush
```

Practical-10

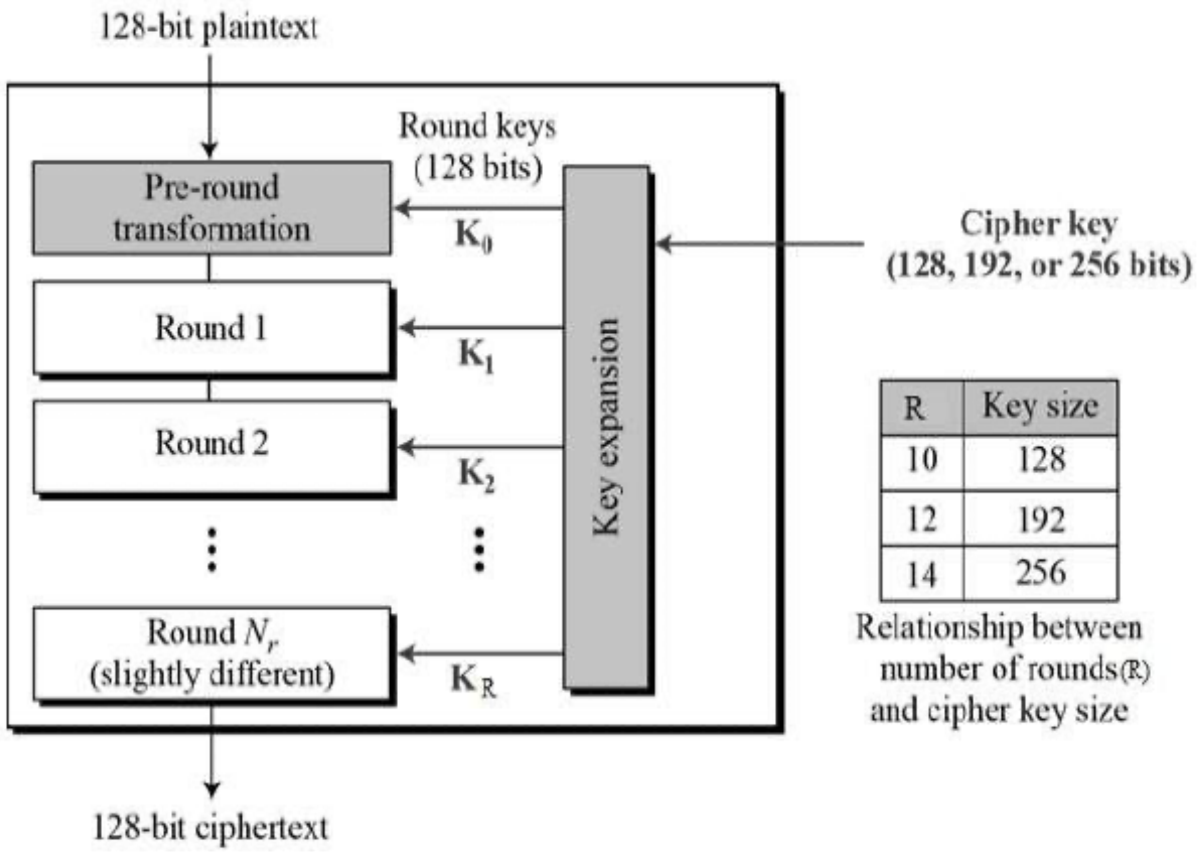
Aim: Write a program to implement AES Algorithm.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The more popular and widely adopted symmetric encryption algorithm likely to be encountered nowadays is the Advanced Encryption Standard (AES). It is found to be at least six times faster than triple DES. A replacement for DES was needed as its key size was too small. With increasing computing power, it was considered vulnerable against exhaustive key search attack.

Unlike DES, the number of rounds in AES is variable and depends on the length of the key. AES uses 10 rounds for 128-bit keys, 12 rounds for 192-bit keys and 14 rounds for 256-bit keys. Each of these rounds uses a different 128-bit round key, which is calculated from the original AES key.

General structure of DES is depicted in the following illustration:

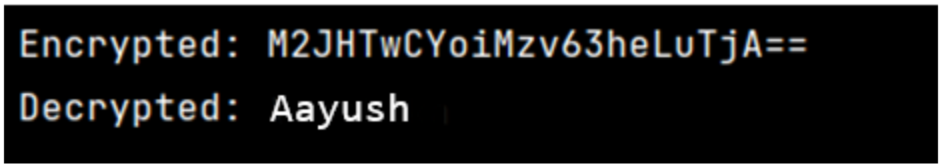


R	Key size
10	128
12	192
14	256

Source Code:

```
from Crypto.Cipher import AES

from Crypto.Random import get_random_bytes

from base64 import b64encode, b64decode

import os


def pad(text):

    block_size = 16

    return text + (block_size - len(text) % block_size) * chr(block_size - len(text) %
block_size)


def unpad(text):

    return text[:-ord(text[-1])]


def encrypt(key, plaintext):

    plaintext = pad(plaintext)

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

    ciphertext = cipher.encrypt(plaintext.encode())

    return b64encode(ciphertext).decode()


def decrypt(key, ciphertext):

    ciphertext = b64decode(ciphertext)

    cipher = AES.new(key, AES.MODE_ECB)
    plaintext = cipher.decrypt(ciphertext).decode()
    return unpad(plaintext)

def generate_key():
    return get_random_bytes(16)

if __name__ == '__main__':
    key = generate_key()

    text = "This is a secret message."
    encrypted_text = encrypt(key, text)
    print("Encrypted:", encrypted_text)

    decrypted_text = decrypt(key, encrypted_text)
    print("Decrypted:", decrypted_text)
```

Output:



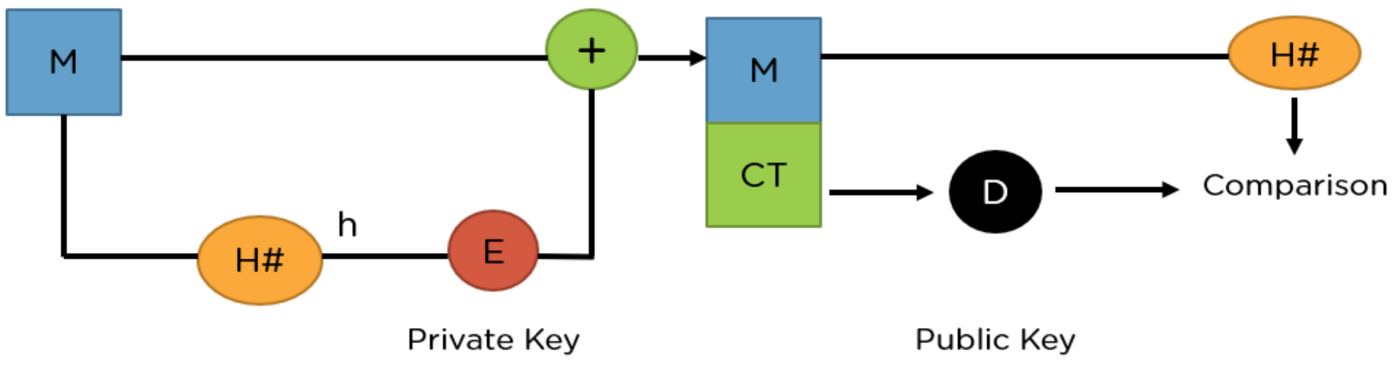
```
Encrypted: M2JHTwCYoiMzv63heLuTjA==
Decrypted: Aayush
```

Practical-11

Aim: Write a program to implement RSA Algorithm.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The RSA algorithm is a public-key signature algorithm developed by Ron Rivest, Adi Shamir, and Leonard Adleman. Their paper was first published in 1977, and the algorithm uses logarithmic functions to keep the working complex enough to withstand brute force and streamlined enough to be fast post-deployment. The image below shows it verifies the digital signatures using RSA methodology.

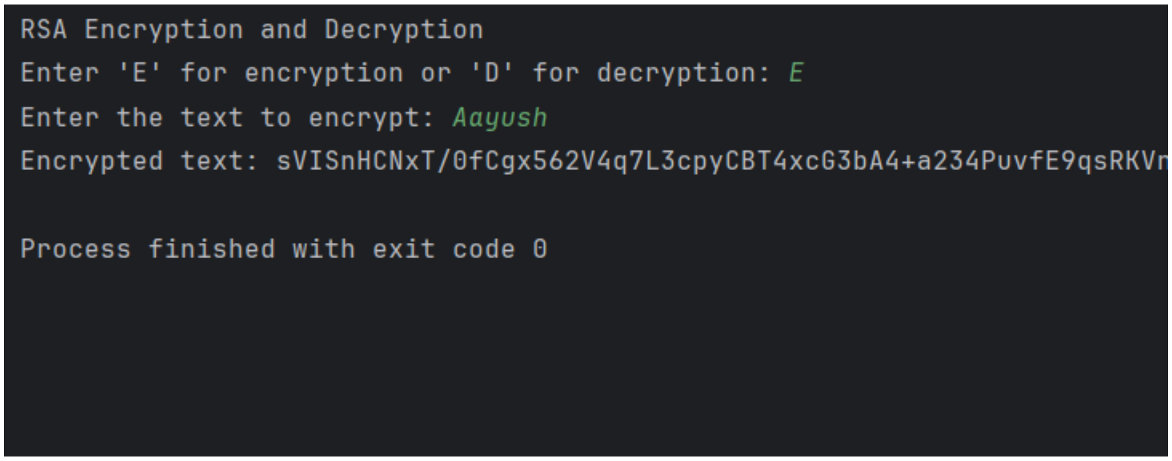


The RSA (Rivest–Shamir–Adleman) algorithm is a widely used public-key cryptosystem that provides secure data encryption and digital signatures. Unlike symmetric-key cryptography, which employs a single key for both encryption and decryption, RSA utilizes a pair of keys: a public key and a private key. The public key is used for encryption and can be openly shared, while the private key, kept secret by the recipient, is used for decryption. The security of RSA is based on the difficulty of factoring large composite numbers into their prime factors, a problem that is computationally infeasible for sufficiently large numbers. It has found applications in securing online communications, digital signatures, and authentication, making it a fundamental building block in modern cryptography and secure data transmission.

Source Code:

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes
import base64

def generate_key_pair():
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()

    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.PKCS8,
        encryption_algorithm=serialization.NoEncryption()
    )

    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    return private_pem, public_pem

def encrypt(public_key_pem, plaintext):
    public_key = serialization.load_pem_public_key(public_key_pem)

    ciphertext = public_key.encrypt(
        plaintext.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return base64.b64encode(ciphertext).decode()

def decrypt(private_key_pem, ciphertext):
    private_key = serialization.load_pem_private_key(private_key_pem,
password=None)

    ciphertext_bytes = base64.b64decode(ciphertext)
    plaintext = private_key.decrypt(
        ciphertext_bytes,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )

    return plaintext.decode()

if __name__ == '__main__':
    private_key_pem, public_key_pem = generate_key_pair()

    print("RSA Encryption and Decryption")
    choice = input("Enter 'E' for encryption or 'D' for decryption: ")

    if choice == 'E':
        text = input("Enter the text to encrypt: ")
        encrypted_text = encrypt(public_key_pem, text)
        print("Encrypted text:", encrypted_text)
    elif choice == 'D':
        ciphertext = input("Enter the encrypted text: ")
        decrypted_text = decrypt(private_key_pem, ciphertext)
        print("Decrypted text:", decrypted_text)
    else:
        print("Invalid choice. Please enter 'E' for encryption or 'D' for decryption.")
```

Output:



```
RSA Encryption and Decryption
Enter 'E' for encryption or 'D' for decryption: E
Enter the text to encrypt: Aayush
Encrypted text: sVISnHCNxT/0fCgx562V4q7L3cpyCBT4xcG3bA4+a234PuvfE9qsRKVr

Process finished with exit code 0
```

Practical-12

Aim: Write a program to implement Digital Signature Algorithm.

Platform Used: PyCharm Community Edition 2023.1.3

Brief Description: The Digital Signature Algorithm (DSA) is a public-key cryptosystem and Federal Information Processing Standard for digital signatures, based on the mathematical concept of modular exponentiation and the discrete logarithm problem. DSA is a variant of the Schnorr and ElGamal signature schemes. Specification FIPS 186-5 indicates DSA will no longer be approved for digital signature generation but may be used to verify signatures generated prior to the implementation date of that standard.

The DSA works in the framework of public-key cryptosystems and is based on the algebraic properties of modular exponentiation, together with the discrete logarithm problem, which is computationally intractable. The algorithm uses a key pair consisting of a public key and a private key. The private key is used to generate a digital signature for a message, and such a signature can be verified by using the signer's corresponding public key. The digital signature provides message authentication (the receiver can verify the origin of the message), integrity (the receiver can verify that the message has not been modified since it was signed) and non-repudiation (the sender cannot falsely claim that they have not signed the message).

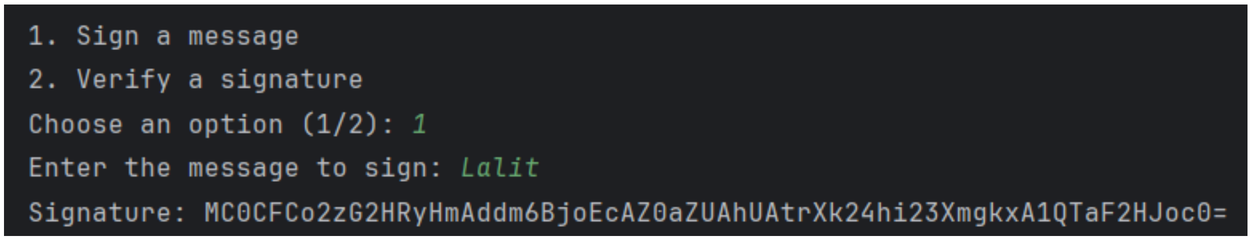
Digital signature is a cryptographic value that is calculated from the data and a secret key known only by the signer. In the real world, the receiver of message needs assurance that the message belongs to the sender, and he should not be able to repudiate the origination of that message. This requirement is crucial in business applications since the likelihood of a dispute over exchanged data is very high.

Digital signatures can also protect confidential information, ensuring that only authorized persons can access the data. Furthermore, digital signatures can also be used to track a digital transaction's source and ensure that it has not been modified.

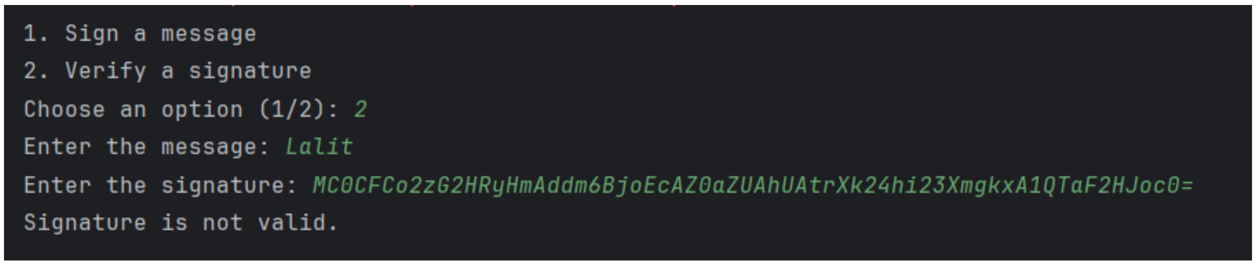
Source Code:

```
from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import dsa

from cryptography.hazmat.primitives.asymmetric import utils

import base64


def generate_dsa_key_pair():

    private_key = dsa.generate_private_key(key_size=1024)

    public_key = private_key.public_key()

    return private_key, public_key


def sign_message(private_key, message):

    signature = private_key.sign(

        message,

        hashes.SHA256()

    )

    return signature


def verify_signature(public_key, message, signature):

    try:

        public_key.verify(

            signature,

            message,
            hashes.SHA256()
        )
        return True
    except:
        return False

if __name__ == '__main__':
    private_key, public_key = generate_dsa_key_pair()

    print("1. Sign a message")
    print("2. Verify a signature")
    choice = input("Choose an option (1/2): ")

    if choice == '1':
        message = input("Enter the message to sign: ").encode()
        signature = sign_message(private_key, message)
        print("Signature:", base64.b64encode(signature).decode())
    elif choice == '2':
        message = input("Enter the message: ").encode()
        signature = base64.b64decode(input("Enter the signature: "))
        verified = verify_signature(public_key, message, signature)
        if verified:
            print("Signature is valid.")
        else:
            print("Signature is not valid.")
    else:
        print("Invalid choice. Please enter '1' to sign or '2' to verify.")
```

Output:



```
1. Sign a message
2. Verify a signature
Choose an option (1/2): 1
Enter the message to sign: Lalit
Signature: MC0CFCo2zG2HRyHmAddm6BjoEcAZ0aZUAhUAtrXk24hi23XmgkxA1QTaF2HJoc0=
```



```
1. Sign a message
2. Verify a signature
Choose an option (1/2): 2
Enter the message: Lalit
Enter the signature: MC0CFCo2zG2HRyHmAddm6BjoEcAZ0aZUAhUAtrXk24hi23XmgkxA1QTaF2HJoc0=
Signature is not valid.
```