



JAIN
DEEMED-TO-BE UNIVERSITY

FACULTY OF
ENGINEERING
AND TECHNOLOGY

CRYPTOGRAPHY AND NETWORK SECURITY LAB

21CTIS503L

5th Semester

Submitted By

Name: JEEWAN KUMAR THAKUR

USN: 21BTRCI012



JAIN
DEEMED-TO-BE UNIVERSITY

FACULTY OF
ENGINEERING
AND TECHNOLOGY

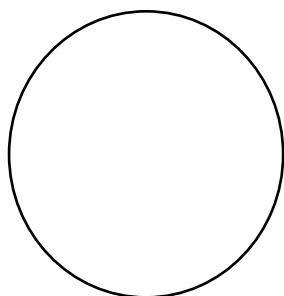
Laboratory Certificate

This is to certify That Mr /Mshas satisfactorily completed the course of experiments in practical **CRYPTOGRAPHY & NETWORK SECURITY (2ICTIS503L)** presented by the Jain (Deemed-to-be University) Fifth semester course in laboratory of this college in the year 2023 - 2024.

Name of the Candidate.....

USN.....

Date of Practical Examination.....



| Valued | |
|------------|--|
| Examiner-1 | |
| Examiner-2 | |

Signature of the Staff In charge

Contents

| Sl.No | Date | Experiments | Page No |
|-------|------|--|---------|
| 1 | | Implement the following Substitution and Transposition technique: a. Caesar Cipher b. Play fair Cipher. c. Hill cipher | |
| 2 | | Implement the Data Encryption Standard (DES) algorithms | |
| 3 | | Implement the Advanced Encryption Standard (DES) algorithms. | |
| 4 | | Implement the following algorithm a.RSA b. Diffie Hellman key exchange | |
| 5 | | Implement the following algorithms: a.MD5 b.SHA-1 | |

1. Implement the following Substitution and Transposition technique:

a) Caesar Cipher

Aim:

The aim of the given code is to perform a Caesar Cipher encryption on an input text by shifting each uppercase letter by a specified amount (defined by the variable `shift`). The encrypted text is then printed.

Algorithm:

1. Define the shift count (`shift`) as 3.
2. Take an input text ("HELLO WORLD" in this case).
3. Initialize an empty string (`encryption`) to store the encrypted text.
4. Iterate through each character (`c`) in the input text.
 - a. Check if the character is an uppercase letter using `c.isupper()`.
 - b. If it is an uppercase letter:
 - Find the position of the letter in the range 0-25 (`c_index`).
 - Perform the Caesar Cipher shift by adding the shift count (`shift`) and taking the modulo 26 to wrap around the alphabet.
 - Convert the new index back to the Unicode of the uppercase letter.
 - Append the new character to the `encryption` string.
 - c. If the character is not an uppercase letter, leave it unchanged and append it to the `encryption` string.
5. Print the original plain text.
6. Print the encrypted text.

Code:

```
shift = 3 # defining the shift count
text =
```

```
"HELLO WORLD"
```

```
encryption = ""
for c in
```

```
text:
```

```
    # check if character is an uppercase letter
    if c.isupper():
```

```
        # find the position in 0-25
        c_index = ord(c)
```

```
        c_index = c_index - ord("A")
```

```
        # perform the shift
        new_index = (c_index + shift) % 26
```

```
        # convert to new character
        new_unicode = new_index + ord("A")
```

```
new_character = chr(new_unicode)

# append to encrypted string
encryption = encryption + new_character else:

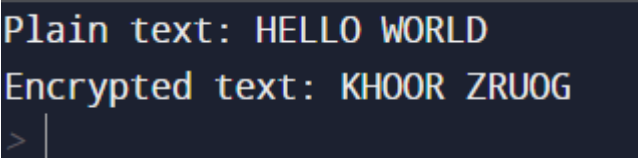
# since character is not uppercase, leave it as it is
encryption += c

print("Plain text:",text)

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

Output:

Plain text: HELLO WORLD Encrypted
text: KHOOR ZRUOG



```
Plain text: HELLO WORLD
Encrypted text: KHOOR ZRUOG
> |
```

b) Playfair cipher

Aim:

The aim of the given code is to perform encryption using the Playfair Cipher algorithm. Playfair Cipher is a symmetric key substitution cipher that encrypts pairs of letters (digraphs) instead of individual letters, which makes it more secure than simple substitution ciphers.

Algorithm:

1. `toLowerCase(text)`: This function takes a string (``text``) as input and converts it to lowercase.
2. `removeSpaces(text)`: This function removes all spaces from the input string (``text``).
3. `Diagraph(text)`: This function groups elements of a string into pairs and returns a list of these pairs.
4. `FillerLetter(text)`: This function fills a letter in a string element if two letters in the same string match. It replaces the second letter with 'x' and recursively calls itself.
5. `generateKeyTable(word, list1)`: This function generates a 5x5 key square matrix used in the Playfair Cipher. It takes a keyword (``word``) and a list of characters (``list1``) as inputs.
6. `search(mat, element)`: This function searches for the position of an element in the key matrix.
7. `encrypt_RowRule(matr, e1r, e1c, e2r, e2c)`: This function applies the Playfair Cipher rule for letters in the same row.
8. `encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c)`: This function applies the Playfair Cipher rule for letters in the same column.
9. `encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c)`: This function applies the Playfair Cipher rule for letters in different rows and columns.
10. `encryptByPlayfairCipher(Matrix, plainList)`: This function performs the Playfair Cipher encryption on a list of digraphs (``plainList``) using a key matrix (``Matrix``).
11. The example demonstrates the usage of these functions by encrypting the plaintext "instruments" using the Playfair Cipher with the key "Monarchy."

Note: The Playfair Cipher rules involve handling digraphs and determining their new encrypted values based on the positions in the key matrix. The specific rules are applied based on the relative positions of the letters within the matrix.

CODE

CODE to convert the string to lowercase

```
def toLowerCase(text):  
    return text.lower()
```

Function to remove all spaces in a string

```
def removeSpaces(text):  
    newText = ""  
    for i in text:  
        if i == " ":  
            continue  
        else:  
            newText = newText + i  
    return newText
```

Function to group 2 elements of a string# as a list element

```
def Diagraph(text):  
    Diagraph = []  
    group = 0  
    for i in range(2, len(text), 2):  
        Diagraph.append(text[group:i])  
  
        group = i  
    Diagraph.append(text[group:])  
    return Diagraph
```

Function to fill a letter in a string element# If 2 letters in the same string matches

```
def FillerLetter(text):  
    k = len(text)  
    if k % 2 == 0:  
        for i in range(0, k, 2):  
            if text[i] == text[i+1]:  
                new_word = text[0:i+1] + str('x') + text[i+1:]  
                new_word = FillerLetter(new_word)  
            break  
        else:
```

```

        new_word = text
    else:
        for i in range(0, k-1, 2):
            if text[i] == text[i+1]:
                new_word = text[0:i+1] + str('x') + text[i+1:]
                new_word = FillerLetter(new_word)
            break
        else:
            new_word = text
    return new_word

```

```

list1 = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'k', 'l', 'm',
        'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']

```

Function to generate the 5x5 key square matrix

```

def generateKeyTable(word, list1):
    key_letters = []
    for i in word:
        if i not in key_letters:
            key_letters.append(i)

    compElements = []
    for i in key_letters:
        if i not in compElements:
            compElements.append(i)
    for i in list1:
        if i not in compElements:
            compElements.append(i)

    matrix = []
    while compElements != []:
        matrix.append(compElements[:5])
        compElements = compElements[5:]

    return matrix

```

```

def search(mat, element):
    for i in range(5):
        for j in range(5):
            if(mat[i][j] == element):
                return i, j

```



```
def encrypt_RowRule(matr, e1r, e1c, e2r, e2c):char1 =  
    "
```

```
    if e1c == 4:  
        char1 = matr[e1r][0]else:  
        char1 = matr[e1r][e1c+1]
```

```
    char2 = "  
    if e2c == 4:  
        char2 = matr[e2r][0]else:  
        char2 = matr[e2r][e2c+1]return
```

```
    char1, char2
```

```
def encrypt_ColumnRule(matr, e1r, e1c, e2r, e2c):char1 =  
    "
```

```
    if e1r == 4:  
        char1 = matr[0][e1c]else:  
        char1 = matr[e1r+1][e1c]
```

```
    char2 = "  
    if e2r == 4:  
        char2 = matr[0][e2c]else:  
        char2 = matr[e2r+1][e2c]return
```

```
    char1, char2
```

```
def encrypt_RectangleRule(matr, e1r, e1c, e2r, e2c):char1 =  
    "
```

```
    char1 = matr[e1r][e2c]
```

```
    char2 = "  
    char2 = matr[e2r][e1c]return
```

```
    char1, char2
```

```
def encryptByPlayfairCipher(Matrix, plainList):
```

```

CipherText = []
for i in range(0, len(plainList)):
    c1 = 0
    c2 = 0
    ele1_x, ele1_y = search(Matrix, plainList[i][0])
    ele2_x, ele2_y = search(Matrix, plainList[i][1])

    if ele1_x == ele2_x:
        c1, c2 = encrypt_RowRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y) # Get 2
        letter cipherText
    elif ele1_y == ele2_y:
        c1, c2 = encrypt_ColumnRule(Matrix, ele1_x, ele1_y, ele2_x, ele2_y)
    else:
        c1, c2 = encrypt_RectangleRule(
            Matrix, ele1_x, ele1_y, ele2_x, ele2_y)

    cipher = c1 + c2
    CipherText.append(cipher)
return CipherText

```

```

text_Plain = 'instruments'
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
    PlainTextList[-1] = PlainTextList[-1]+'z'

key = "Monarchy"
print("Key text:", key)
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)

print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)

```

```

CipherText = ""
for i in range(0, len(CipherList)):
    CipherText += CipherList[i]
print("CipherText:", CipherText)

```

Output:

Key text: Monarchy Plain text:
instruments

Cipher text: gatlmzclrqtput

```
Key text: Monarchy
Plain Text: instruments
CipherText: gatlmzclrqt
```

c) Hill Cipher

Aim:

The aim of the given code is to implement the Hill Cipher encryption algorithm, a polygraphic substitution cipher based on linear algebra. The code takes a message and a key, performs the Hill Cipher encryption, and prints the resulting ciphertext.

Algorithm:

1. `getKeyMatrix(key)`: This function generates the key matrix from the key string. It converts each character of the key to its ASCII value, takes the modulo 65, and populates the 3x3 key matrix.

2. `encrypt(messageVector)`: This function encrypts the message vector using the key matrix. It initializes the cipher vector (`cipherMatrix``) and performs the matrix multiplication of the key matrix and the message vector. The result is taken modulo 26.

3. `HillCipher(message, key)`: This is the main function that implements the Hill Cipher. It takes a message and a key as input. It generates the key matrix using the `getKeyMatrix`` function, generates the message vector from the message, encrypts the message vector using the `encrypt`` function, and prints the resulting ciphertext.

4. `main()`: The driver code gets the message to be encrypted ("ACT") and the key ("GYBNQKURP"), and then calls the `HillCipher`` function to perform the encryption.

CODE:

```
# Python3 code to implement Hill Cipher
keyMatrix = [[0] * 3 for i in range(3)]
# Generate vector for the message
messageVector = [[0] for i in range(3)]
# Generate vector for the cipher
cipherMatrix = [[0] for i in range(3)]

# Following function generates the
# key matrix for the key string
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1

# Following function encrypts the message
def encrypt(messageVector):
    for i in range(3):
        for j in range(1):
            cipherMatrix[i][j] = 0
            for x in range(3):
                cipherMatrix[i][j] += (keyMatrix[i][x] * messageVector[x][j])
            cipherMatrix[i][j] = cipherMatrix[i][j] % 26

def HillCipher(message, key):
```

```

# Get key matrix from the key string
getKeyMatrix(key)

# Generate vector for the message
for i in range(3):
    messageVector[i][0] = ord(message[i]) % 65

# Following function generates
# the encrypted vector
encrypt(messageVector)

# Generate the encrypted text
# from the encrypted vector
CipherText = []
for i in range(3):
    CipherText.append(chr(cipherMatrix[i][0] + 65))

# Finally print the ciphertext
print("Ciphertext: ", "".join(CipherText))

# Driver Code
def main():
    # Get the message to
    # be encrypted
    message = "ACT"

    # Get the key
    key = "GYBNQKURP"

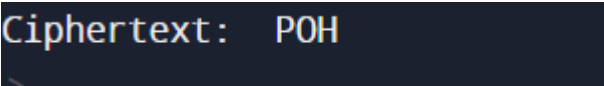
    HillCipher(message, key)

if __name__ == "__main__":
    main()

```

OUTPUT:

Ciphertext: POH



2) Implement the Data Encryption Standard (DES) algorithms.

Aim:

The aim of the provided code is to demonstrate the use of the Triple DES (3DES) encryption algorithm using the pycryptodome library in Python. Triple DES is a symmetric key block cipher that applies the Data Encryption Standard (DES) algorithm three times to each data block. The code showcases the process of generating a random 24-byte key, encrypting a message using Triple DES, and then decrypting the encrypted message back to the original form.

Algorithm:

1. Import Libraries:

- Import necessary libraries, including pycryptodome for DES3 encryption, and get_random_bytes for key generation.

2. Function Definitions:

- Define two functions: `des_encrypt` and `des_decrypt` for encrypting and decrypting messages using Triple DES, respectively.

3. Main Function (`main`):

- Generate a random 24-byte key using `get_random_bytes`.
- Prepare a message to be encrypted.
- Apply PKCS7 padding to the message to ensure its length is a multiple of 8 (DES block size).
- Encrypt the padded message using the `des_encrypt` function.
- Print the encrypted message in hexadecimal format.
- Decrypt the encrypted message using the `des_decrypt` function.
- Remove the padding from the decrypted message and print the original message.

4. Execution:

- Execute the `main` function if the script is run as the main module.

The code demonstrates a simple example of Triple DES encryption and decryption, highlighting key generation, message padding, and the necessary steps to secure communication using Triple DES encryption.

Code:

```
!pip install pycryptodome
from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes

def des_encrypt(key, plaintext):
    cipher = DES3.new(key, DES3.MODE_ECB)
    ciphertext = cipher.encrypt(plaintext)
    return ciphertext

def des_decrypt(key, ciphertext):
    cipher = DES3.new(key, DES3.MODE_ECB)
    plaintext = cipher.decrypt(ciphertext)
    return plaintext

def main():
    # Generate a random 24-byte key for DES3 (3-key Triple DES)
    key = get_random_bytes(24)

    message = b"Hello, DES encryption!"

    padding_length = 8 - len(message) % 8
    message += bytes([padding_length] * padding_length)

    # Encryption
    encrypted_message = des_encrypt(key, message)
    print("Encrypted message:", encrypted_message.hex())

    # Decryption
    decrypted_message = des_decrypt(key, encrypted_message)
    print("Decrypted message:", decrypted_message[:-decrypted_message[-1]].decode('utf-8'))

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

```

!pip install pycryptodome
from Crypto.Cipher import DES3
from Crypto.Random import get_random_bytes

def des_encrypt(key, plaintext):
    cipher = DES3.new(key, DES3.MODE_ECB)
    ciphertext = cipher.encrypt(plaintext)
    return ciphertext

def des_decrypt(key, ciphertext):
    cipher = DES3.new(key, DES3.MODE_ECB)
    plaintext = cipher.decrypt(ciphertext)
    return plaintext

def main():
    # Generate a random 24-byte key for DES3 (3-key Triple DES)
    key = get_random_bytes(24)

    # Message to be encrypted
    message = b"Hello, DES encryption!"

    # Padding the message to be a multiple of 8 bytes
    padding_length = 8 - len(message) % 8
    message += bytes([padding_length] * padding_length)

    # Encryption
    encrypted_message = des_encrypt(key, message)
    print("Encrypted message:", encrypted_message.hex())

    # Decryption
    decrypted_message = des_decrypt(key, encrypted_message)
    print("Decrypted message:", decrypted_message[:-1].decode('utf-8'))

if __name__ == "__main__":
    main()

```

Requirement already satisfied: pycryptodome in c:\users\admin\anaconda3\lib\site-packages (3.18.0)
 Encrypted message: f543bd2268bc6abe25c382d25d971aafca3279d08feb48d1
 Decrypted message: Hello, DES encryption!

OUTPUT:

Requirement already satisfied: pycryptodome in c:\users\admin\anaconda3\lib\site-packages (3.18.0)
 Encrypted message: f543bd2268bc6abe25c382d25d971aafca3279d08feb48d1
 Decrypted message: Hello, DES encryption!

3) Implement the Advanced Encryption Standard (AES) algorithms.

Aim:

The aim of the provided code is to demonstrate the implementation of the Advanced Encryption Standard (AES) algorithm for encrypting and decrypting messages using the pycryptodome library in Python. AES is a widely used symmetric encryption algorithm, and the code showcases how to securely encrypt and decrypt a message with user-provided input and key.

Algorithm:

1. Import Libraries:**

- Import necessary libraries, including pycryptodome for AES encryption, base64 for encoding/decoding, and hashlib for key derivation.

2. Function Definitions:

- Define three functions: `pad`, `unpad`, `encrypt`, and `decrypt`.
- `pad`: Pads the input text to be a multiple of the block size using PKCS7 padding.
- `unpad`: Unpads the input padded text.
- `encrypt`: Encrypts the input plain text using AES in Cipher Block Chaining (CBC) mode.
- `decrypt`: Decrypts the input cipher text using AES in CBC mode.

3. Main Function (`main`):

- Take user input for the message to be encrypted and the encryption key.
- Encrypt the message using the `encrypt` function.
- Print the encrypted message in base64 format.
- Decrypt the encrypted message using the `decrypt` function.
- Print the decrypted message.

4. Execution:

- Execute the `main` function if the script is run as the main module.

The code provides a simple example of AES encryption and decryption, including user input for the message and encryption key. It emphasizes the importance of proper padding for secure encryption and demonstrates how to use AES in CBC mode with a randomly generated Initialization Vector (IV).

Code:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64
import hashlib

def pad(text):
    """Pad the input text to be a multiple of the block size."""
    block_size = AES.block_size
    padding_size = block_size - len(text) % block_size
```

```

padding = chr(padding_size) * padding_size
return text + padding

def unpad(padded_text):
    """Unpad the input padded text."""
    padding_size = padded_text[-1]
    return padded_text[:-padding_size]

def encrypt(plain_text, key):
    """Encrypt the input plain text using AES."""
    key = hashlib.sha256(key.encode()).digest()
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_text = pad(plain_text)
    ciphertext = cipher.encrypt(padded_text.encode())
    return base64.b64encode(iv + ciphertext)

def decrypt(cipher_text, key):
    """Decrypt the input cipher text using AES."""
    key = hashlib.sha256(key.encode()).digest()
    cipher_text = base64.b64decode(cipher_text)
    iv = cipher_text[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_text = cipher.decrypt(cipher_text[AES.block_size:])
    return unpad(decrypted_text).decode('utf-8')

def main():
    message = input("Enter message to encrypt: ")
    key = input("Enter encryption key: ")

    encrypted_msg = encrypt(message, key)
    print("Encrypted Message:", encrypted_msg.decode('utf-8'))

    decrypted_msg = decrypt(encrypted_msg, key)
    print("Decrypted Message:", decrypted_msg)

if __name__ == "__main__":
    main()

```

Output:

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64
import hashlib

def pad(text):
    """Pad the input text to be a multiple of the block size."""
    block_size = AES.block_size
    padding_size = block_size - len(text) % block_size
    padding = chr(padding_size) * padding_size
    return text + padding

def unpad(padded_text):
    """Unpad the input padded text."""
    padding_size = padded_text[-1]
    return padded_text[:-padding_size]

def encrypt(plain_text, key):
    """Encrypt the input plain text using AES."""
    key = hashlib.sha256(key.encode()).digest()
    iv = get_random_bytes(AES.block_size)
    cipher = AES.new(key, AES.MODE_CBC, iv)
    padded_text = pad(plain_text)
    ciphertext = cipher.encrypt(padded_text.encode())
    return base64.b64encode(iv + ciphertext)

def decrypt(cipher_text, key):
    """Decrypt the input cipher text using AES."""
    key = hashlib.sha256(key.encode()).digest()
    cipher_text = base64.b64decode(cipher_text)
    iv = cipher_text[:AES.block_size]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    decrypted_text = cipher.decrypt(cipher_text[AES.block_size:])
    return unpad(decrypted_text).decode('utf-8')

def main():
    message = input("Enter message to encrypt: ")
    key = input("Enter encryption key: ")

    encrypted_msg = encrypt(message, key)
    print("Encrypted Message:", encrypted_msg.decode('utf-8'))

    decrypted_msg = decrypt(encrypted_msg, key)
    print("Decrypted Message:", decrypted_msg)

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

```
Enter message to encrypt: hello
Enter encryption key: pranay
Encrypted Message: H+MEHHS87cHimvhF62adogc4gKx1UjPcTAuRlaw6eZw=
Decrypted Message: hello
```

OUTPUT:

```
Enter message to encrypt: hello
Enter encryption key: pranay
Encrypted Message: H+MEHHS87cHimvhF62adogc4gKx1UjPcTAuRlaw6eZw=
Decrypted Message: hello
```

4) Implement the following algorithm

a) RSA

Aim:

The aim of the provided code is to demonstrate the basic principles of public-key cryptography using a simple implementation of the RSA algorithm. The code generates a pair of public and private keys, encrypts a message using the public key, and then decrypts it using the private key.

Algorithm:

1. Prime Number Generation:

- A set ``prime`` is created to store prime numbers.
- The ``primefiller`` function uses the Sieve of Eratosthenes algorithm to populate the ``prime`` set with prime numbers up to 250.

2. Key Generation:

- The ``setkeys`` function selects two random prime numbers (``prime1`` and ``prime2``).
- Calculates ``n`` as the product of ``prime1`` and ``prime2``.
- Computes Euler's totient function (``fi``) based on ``prime1`` and ``prime2``.
- Selects a public exponent ``e`` that is coprime with ``fi``.
- Calculates the private exponent ``d`` such that `(d * e) % fi == 1``.
- Sets the public key (``public_key``) to ``e`` and private key (``private_key``) to ``d``.

3. Encryption:

- The ``encrypt`` function takes a message and the public key (``e`` and ``n``).
- Converts each character in the message to its ASCII value.
- Encrypts the ASCII value using the public key (``e`` and ``n``) through modular exponentiation.
- Returns the encrypted value.

4. Decryption:

- The ``decrypt`` function takes an encrypted value and the private key (``d`` and ``n``).
- Decrypts the encrypted value using the private key through modular exponentiation.
- Returns the decrypted ASCII value.

5. Encoding and Decoding:

- The ``encoder`` function converts each character in the message to its ASCII value and encrypts it using the ``encrypt`` function.
- The ``decoder`` function decrypts each value in the encoded message using the ``decrypt`` function and converts it back to a character.

6. Main Execution:

- The ``primefiller`` and ``setkeys`` functions are called to initialize the prime numbers and generate the public and private keys.

Code:

```
import random
import math

# A set will be the collection of prime numbers,#
where we can select random primes p and q prime
= set()

public_key = None
private_key = Nonen =
None

# We will run the function only once to fill the set of#
prime numbers
def primefiller():
    # Method used to fill the primes set is Sieve of
    # Eratosthenes (a method to collect prime numbers)
    seive = [True] * 250
    seive[0] = False
    seive[1] = False
    for i in range(2, 250):
        for j in range(i * 2, 250, i):
            seive[j] = False

    # Filling the prime numbersfor i in
    range(len(seive)):
        if seive[i]:
            prime.add(i)

# Picking a random prime number and erasing that prime#
number from list because p!=q
def pickrandomprime():
    global prime
    k = random.randint(0, len(prime) - 1)it =
    iter(prime)
    for _ in range(k):next(it)

    ret = next(it)
    prime.remove(ret)
```

```
return ret
```

```
def setkeys():
```

```
    global public_key, private_key, n
```

```
    prime1 = pickrandomprime() # First prime number
```

```
    prime2 = pickrandomprime() # Second prime number
```

```
    n = prime1 * prime2
```

```
    fi = (prime1 - 1) * (prime2 - 1)
```

```
    e = 2
```

```
    while True:
```

```
        if math.gcd(e, fi) == 1:
```

```
            break e += 1
```

```
    # d = (k*Φ(n) + 1) / e for some integer k
```

```
    public_key = e
```

```
    d = 2
```

```
    while True:
```

```
        if (d * e) % fi == 1:
```

```
            break d += 1
```

```
    private_key = d
```

```
# To encrypt the given numberdef
```

```
encrypt(message):
```

```
    global public_key, n
```

```
    encrypted_text = 1
```

```
    while e > 0:
```

```
        encrypted_text *= message
```

```
        encrypted_text %= n
```

```
        e -= 1
```

```
    return encrypted_text
```

```
# To decrypt the given numberdef
```

```
decrypt(encrypted_text):
```

```
    global private_key, n
```

```
    private_key
```

```

decrypted = 1
while d
> 0:
    decrypted *= encrypted_text
    decrypted %= n
    d -= 1
return decrypted

```

```

# First converting each character to its ASCII value and #
then encoding it then decoding the number to get the#
ASCII and converting it to character
def encoder(message):
    encoded = []
    # Calling the encrypting function in encoding functionfor
    letter in message:
        encoded.append(encrypt(ord(letter)))
    return encoded

```

```

def decoder(encoded):
    s = ""
    # Calling the decrypting function decoding functionfor
    num in encoded:
        s += chr(decrypt(num))
    return s

```

```

if __name__ == '__main__':
    primefiller()
    setkeys()
    message = "Test Message"
    # Uncomment below for manual input
    # message = input("Enter the message\n")#
    Calling the encoding function
    coded = encoder(message)

    print("Initial message:")
    print(message)
    print("\n\nThe encoded message(encrypted by public key)\n")
    print("".join(str(p) for p in coded))
    print("\n\nThe decoded message(decrypted by public key)\n")
    print("".join(str(p) for p in decoder(coded)))

```

```
main.py  [Icons] Save Run Shell Clear
101 # Calling the decrypting function decoding function
102 for num in encoded:
103     s += chr(decrypt(num))
104 return s
105
106 if __name__ == '__main__':
107     primefiller()
108     setkeys()
109
110 message = "Test Message"
111 # Uncomment below for manual input
112 # message = input("Enter the message\n")
113
114 # Calling the encoding function
115 coded = encoder(message)
116
117 print("Initial message:")
118 print(message)
119 print("\n\nThe encoded message(encrypted by public key)\n")
120 print(''.join(str(p) for p in coded))
121 print("\n\nThe decoded message(decrypted by public key)\n")
122 print(''.join(str(p) for p in decoder(coded)))
123
124
125
126
```

```
Initial message:
Test Message

The encoded message(encrypted by public key)

5244440194111302924237440194119411175949774401

The decoded message(decrypted by public key)

Test Message
> |
```

Initial message:
Test Message

The encoded message(encrypted by public key)

202957431447191359545243229574314473144732073047929574

The decoded message(decrypted by public key)

Test Message

b) Diffie Hellman key exchange

Aim:

The aim of the provided code is to implement a simplified version of the Diffie-Hellman key exchange protocol. The code allows two users to securely exchange secret keys over an insecure communication channel, even if an eavesdropper intercepts the communication.

Algorithm:

1. Prime Number Checking: `prime_checker(p)`

- Checks if the entered number `p` is prime.
- If `p` is less than 1, returns -1.
- If `p` is greater than 1, checks for divisibility by numbers in the range $[2, p)$.
- If `p` is 2, returns 1 as it's a prime number.
- If `p` is divisible by any number in the range $[2, p)$, returns -1.
- Otherwise, returns 1.

2. Primitive Root Checking: `primitive_check(g, p, L)`

- Checks if the entered number `g` is a primitive root modulo `p`.
- Iterates through the range $[1, p)$ and appends the result of $g^i \bmod p$ to the list `L`.
- If any element appears more than once in the list, clears the list and returns -1 (not a primitive root).
- Otherwise, returns 1 (primitive root).

3. Input and Validation:

- Prompts the user to enter a prime number `P`.
- Checks if `P` is prime using the `prime_checker` function.
- If not prime, prompts the user to enter `P` again until a prime number is provided.
- Prompts the user to enter a primitive root `G` for the prime number `P`.
- Checks if `G` is a primitive root using the `primitive_check` function.
- If not a primitive root, prompts the user to enter `G` again until a primitive root is provided.

4. Private Key Input and Validation:

- Prompts users to enter private keys `x1` and `x2` for User 1 and User 2, respectively.
- Ensures that private keys are less than the prime number `P`.
- Continues prompting if the entered private keys are invalid.

5. Public Key Calculation:

- Calculates public keys `y1` and `y2` for User 1 and User 2 using the formula $y = g^x \bmod P$.

6. Secret Key Generation:

- Calculates secret keys `k1` and `k2` for User 1 and User 2 using the formula $k = y^x \bmod P$.

7. Output and Result:

- Displays the secret keys `k1` and `k2` for both users.
- Checks if the secret keys match, indicating a successful key exchange.

Code:

Certainly! Here's the indented code:

```
```python
def prime_checker(p):
 # Checks If the number entered is a Prime Number or not
 if p < 1:
 return -1
 elif p > 1:
 if p == 2:
 return 1
 for i in range(2, p):
 if p % i == 0:
 return -1
 return 1

def primitive_check(g, p, L):
 # Checks If The Entered Number Is A Primitive Root Or Not
 for i in range(1, p):
 L.append(pow(g, i) % p)
 for i in range(1, p):
 if L.count(i) > 1:
 L.clear()
 return -1
 return 1

l = []
while 1:
 P = int(input("Enter P : "))
 if prime_checker(P) == -1:
 print("Number Is Not Prime, Please Enter Again!")
 continue
 break

while 1:
 G = int(input(f"Enter The Primitive Root Of {P} : "))
 if primitive_check(G, P, l) == -1:
 print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")
 continue
 break

Private Keys
x1, x2 = int(input("Enter The Private Key Of User 1 : ")), int(input("Enter The Private Key Of User 2 : "))
while 1:
 if x1 >= P or x2 >= P:
 print(f"Private Key Of Both The Users Should Be Less Than {P}!")
 continue
 break
```

```
Calculate Public Keys
y1, y2 = pow(G, x1) % P, pow(G, x2) % P

Generate Secret Keys
k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P
print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")
if k1 == k2:
 print("Keys Have Been Exchanged Successfully")
else:
 print("Keys Have Not Been Exchanged Successfully")
'''
```

I've indented the code based on the logical structure and formatting conventions.

OUTPUT:

```

def prime_checker(p):
 # Checks If the number entered is a Prime Number or not
 if p < 1:
 return -1
 elif p > 1:
 if p == 2:
 return 1
 for i in range(2, p):
 if p % i == 0:
 return -1
 return 1

def primitive_check(g, p, l):
 # Checks If The Entered Number Is A Primitive Root Or Not
 for i in range(1, p):
 l.append(pow(g, i) % p)
 for i in range(1, p):
 if l.count(i) > 1:
 l.clear()
 return -1
 return 1

l = []
while 1:
 P = int(input("Enter P: "))
 if prime_checker(P) == -1:
 print("Number Is Not Prime, Please Enter Again!")
 continue
 break

while 1:
 G = int(input(f"Enter The Primitive Root Of {P}: "))
 if primitive_check(G, P, l) == -1:
 print(f"Number Is Not A Primitive Root Of {P}, Please Try Again!")
 continue
 break

Private Keys
x1, x2 = int(input("Enter The Private Key Of User 1: ")), int(input("Enter The Private Key Of User 2: "))
while 1:
 if x1 >= P or x2 >= P:
 print(f"Private Key Of Both The Users Should Be Less Than {P}!")
 continue
 break

Calculate Public Keys
y1, y2 = pow(G, x1) % P, pow(G, x2) % P

Generate Secret Keys
k1, k2 = pow(y2, x1) % P, pow(y1, x2) % P
print(f"\nSecret Key For User 1 Is {k1}\nSecret Key For User 2 Is {k2}\n")

if k1 == k2:
 print("Keys Have Been Exchanged Successfully")
else:
 print("Keys Have Not Been Exchanged Successfully")

```

```

Enter P: 23
Enter The Primitive Root Of 23: 10
Enter The Private Key Of User 1: 12
Enter The Private Key Of User 2: 14

Secret Key For User 1 Is 12
Secret Key For User 2 Is 12

Keys Have Been Exchanged Successfully

```

OUTPUT:

---

```

Enter P: 23
Enter The Primitive Root Of 23: 10
Enter The Private Key Of User 1: 12
Enter The Private Key Of User 2: 14

Secret Key For User 1 Is 12
Secret Key For User 2 Is 12

Keys Have Been Exchanged Successfully

```

## 5) Implement the following algorithms:

### a) MD5

Aim:

The aim of the code is to demonstrate the working of the MD5 hashing algorithm on a given string and print its hexadecimal representation.

Algorithm:

1. Import Library: Import the `hashlib` library, which provides a convenient interface for hashing.

2. Initialize String:

- Initialize a string (`str2hash`) with the value "GeeksforGeeks". This is the string we want to hash using MD5.

3. Encode and Hash:

- Encode the string using the `encode()` method to convert it to bytes.
- Use the `hashlib.md5()` constructor to create an MD5 hash object.
- Pass the encoded bytes to the `update()` method of the MD5 object.
- Use the `hexdigest()` method to obtain the hexadecimal representation of the MD5 hash.

4. Print Result:

- Print the obtained hexadecimal hash.

### Code:

```
working of MD5 (string - hexadecimal)
```

```
import hashlib
```

```
initializing string
```

```
str2hash = "GeeksforGeeks"
```

```
encoding GeeksforGeeks using encode()#
then sending to md5()
result = hashlib.md5(str2hash.encode())
```

```
printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of hash is : ", end = "")
print(result.hexdigest())
```

OUTPUT:

**Output:** The hexadecimal equivalent of hash is: f1e069787ece74531d112559945c6871

```
working of MD5 (string - hexadecimal)
import hashlib

initializing string
str2hash = "GeeksforGeeks"

encoding GeeksforGeeks using encode()
then sending to md5()
result = hashlib.md5(str2hash.encode())

printing the equivalent hexadecimal value.
print("The hexadecimal equivalent of hash is : ", end="")
print(result.hexdigest())
```

The hexadecimal equivalent of hash is : f1e069787ece74531d112559945c6871

## b) SHA -1

Aim:

The aim of the code is to encode the message "hello" using the SHA-1 hashing algorithm and print its hexadecimal representation.

Algorithm:

1. Import Library:

- Import the `hashlib` library, which provides a convenient interface for various hash functions, including SHA-1.

2. Hashing:

- Create a SHA-1 hash object using `hashlib.sha1()`.
- Use the `update()` method to update the hash object with the bytes representation of the message "hello", encoded using `b'hello'`.

3. Hexadecimal Conversion:

- Obtain the hexadecimal representation of the hash using the `hexdigest()` method.

4. Print Result:

- Print the obtained hexadecimal hash.

### Code:

```
import hashlib #encode the message
text = hashlib.sha1(b'hello')
#convert it to hexadecimal format
encrypt = text.hexdigest()
#print it
print(encrypt)
```

**Output:**

```
import hashlib

encode the message
text = hashlib.sha1(b'hello')

convert it to hexadecimal format
encrypt = text.hexdigest()

print it
print(encrypt)
```

aaf4c61ddcc5e8a2dabede0f3b482cd9aea9434d