

Ex No:01 IMPLEMENT RSA ASYMMETRIC (SECRET KEY ENCRYPTION) ALGORITHM

Date:

AIM:

To execute RSA Algorithm Using python and encrypt and decrypt.

ALGORITHM:

1. Start.
2. **Key Generation:**
 - a. Choose two distinct prime numbers, p and q.
 - b. Compute $n = p * q$.
 - c. Compute $\Phi(n) = (p-1) * (q-1)$. where Φ is Eucler's totient function.
 - d. Choose an integer e such that $1 < e < \Phi(n)$ and e is coprime to $\Phi(n)$.
 - e. Compute the integer d such that $d * e \equiv 1 \pmod{\Phi(n)}$, i.e., d is the modular multiplicative inverse of e modulo $\Phi(n)$.
 - f. The public key is (n,e) and the private key is (n,d).
3. Encryption:
 - a. Encrypt the message M using public key(n,e).
 - i. Compute $C \equiv M^e \pmod{n}$.
4. Decryption:
 - a. Decrypt the ciphertext C using the private key (n, d).
 - i. Compute $M \equiv c^d \pmod{N}$.
5. Stop

PROGRAM:

```
import random
```

```
def gcd(a, b):
```

```
    while b != 0:
```

```
        a, b = b, a % b
```

```
    return a
```

```
def multiplicative_inverse(e, phi):
```

```
    d = 0
```

```
    x1, x2, y1, y2 = 0, 1, 1, 0
```

```
    temp_phi = phi
```

```
    while e > 0:
```

```
        temp1 = temp_phi // e
```

```
        temp2 = temp_phi - temp1 * e
```

```
        temp_phi = e
```

```
e = temp2
```

```
x = x2 - temp1 * x1
```

```
y = y2 - temp1 * y1
```

```
x2 = x1
```

```
x1 = x
```

```
y2 = y1
```

```
y1 = y
```

```
if temp_phi == 1:
```

```
    d = y2 + phi
```

```
return d
```

```
def generate_keypair(p, q):
```

```
    if not (is_prime(p) and is_prime(q)):
```

```
        raise ValueError("Both numbers must be prime.")
```

```
    elif p == q:
```

```
        raise ValueError("p and q cannot be equal")
```

```
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 = multiplicative_inverse(e, phi)
```

```
    return ((e, n), (d, n))
```

```
def encrypt(pk, plaintext):
```

```
    key, n = pk
```

```
    cipher = [pow(ord(char), key, n) for char in plaintext]
```

```
    return cipher
```

```
def decrypt(pk, ciphertext):
```

```
    key, n = pk
```

```
    plain = [chr(pow(char, key, n)) for char in ciphertext]
```

```
    return ''.join(plain)
```

```
def is_prime(num):
```

```
    if num == 2 or num == 3:
```

```
        return True
```

```
    if num < 2 or num % 2 == 0:
```

```
        return False
```

```
    for n in range(3, int(num**0.5)+2, 2):
```

```
        if num % n == 0:
```

```
            return False
```

```
    return True
```

```
# Example usage:
```

```
p = 61
```

```
q = 53
```

```
public_key, private_key = generate_keypair(p, q)
```

```
print("Public Key:", public_key)
```

```
print("Private Key:", private_key)
```

```
message = "Hello, World!"
```

```
print("Original Message:", message)
```

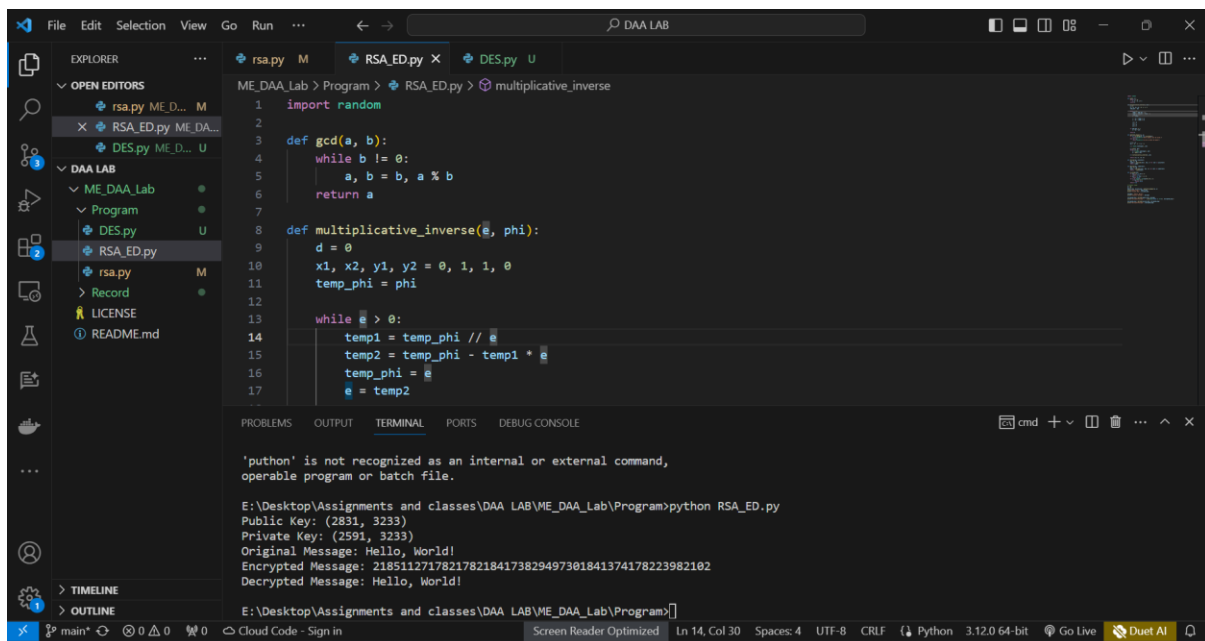
```
encrypted_msg = encrypt(public_key, message)

print("Encrypted Message:", ".join(map(lambda x: str(x), encrypted_msg)))
```

```
decrypted_msg = decrypt(private_key, encrypted_msg)

print("Decrypted Message:", decrypted_msg)
```

OUTPUT:



The screenshot shows a VS Code editor with a Python file named `RSA_ED.py` open. The code defines a `gcd` function and a `multiplicative_inverse` function. The terminal output shows the execution of the script, displaying the public and private keys, the original message, the encrypted message, and the decrypted message.

```
ME_DAA_Lab > Program > RSA_ED.py > multiplicative_inverse
1 import random
2
3 def gcd(a, b):
4     while b != 0:
5         a, b = b, a % b
6     return a
7
8 def multiplicative_inverse(e, phi):
9     d = 0
10    x1, x2, y1, y2 = 0, 1, 1, 0
11    temp_phi = phi
12
13    while e > 0:
14        temp1 = temp_phi // e
15        temp2 = temp_phi - temp1 * e
16        temp_phi = e
17        e = temp2
```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```
'python' is not recognized as an internal or external command,
operable program or batch file.

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python RSA_ED.py
Public Key: (2831, 3233)
Private Key: (2591, 3233)
Original Message: Hello, World!
Encrypted Message: 2185112717821782184173829497301841374178223982102
Decrypted Message: Hello, World!

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

RESULT:

This RSA algorithm was executed successfully.

Ex No:02 IMPLEMENT DATA ENCRYPTION STANDARD (DES) A SYMMETRIC ENCRYPTION ALGORITHM

Date:

AIM:

To implement DES to encode and decode a plain text using key in python.

ALGORITHM:

1. Start.
2. Get the values of Li, Ri, Ci, Di in hex format from the user.
3. Perform left circular shift of Ci Di with respective to the round with table values.
4. Using PC2 table arrange the elements in the same order.
5. Compute expansion permutation of ith round (Ri) and arrange elements in 6x8 matrix.
6. Convert PC2 matrix in 8x6 order.
7. Perform $A = PC2 \text{ EX-OR } E/P(Ri)$
8. Find the value of 'A' in S-Box Table.
9. Arrange the 'A' Matrix in permutation Function table order.
10. $Li+1 = Ri$ $Ri+1 = Li$ EX-OR P32.
11. To decrypt, apply the same algorithm with subkey used in reverse order.
12. Stop.

PROGRAM:

```
def main():
```

```
    print()
```

```
    # Taking inputs from the user
```

```
    plaintext = input("Enter the message to be encrypted : ")
```

```
    key = input("Enter a key of 8 length (64-bits) (characters or numbers only) : ")
```

```
    print()
```

```
    # Checking if key is valid or not
```

```
    if len(key) != 8:
```

```
        print("Invalid Key. Key should be of 8 length (8 bytes).")
```

```
        return
```

```
    # Determining if padding is required
```

```
    isPaddingRequired = (len(plaintext) % 8 != 0)
```

```
    # Encryption
```

```
    ciphertext = DESEncryption(key, plaintext, isPaddingRequired)
```

```
# Decryption
```

```
plaintext = DESDecryption(key, ciphertext, isPaddingRequired)
```

```
# Printing result
```

```
print()
```

```
print("Encrypted Ciphertext is : %r " % ciphertext)
```

```
print("Decrypted plaintext is : ", plaintext)
```

```
print()
```

```
# Permutation Matrix used after each SBox substitution for each round
```

```
eachRoundPermutationMatrix = [
```

```
    16, 7, 20, 21, 29, 12, 28, 17,
```

```
    1, 15, 23, 26, 5, 18, 31, 10,
```

```
    2, 8, 24, 14, 32, 27, 3, 9,
```

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

```
]
```

```
# Final Permutation Matrix for data after 16 rounds
```

```
finalPermutationMatrix = [
```

```
    40, 8, 48, 16, 56, 24, 64, 32,
```

```
    39, 7, 47, 15, 55, 23, 63, 31,
```

```
    38, 6, 46, 14, 54, 22, 62, 30,
```

```
    37, 5, 45, 13, 53, 21, 61, 29,
```

```
    36, 4, 44, 12, 52, 20, 60, 28,
```

```
    35, 3, 43, 11, 51, 19, 59, 27,
```

```
    34, 2, 42, 10, 50, 18, 58, 26,
```

```
    33, 1, 41, 9, 49, 17, 57, 25
```

```
]
```

```
def DESEncryption(key, text, padding):
```

```
    """Function for DES Encryption."""
```

```

# Adding padding if required
if padding == True:
    text = addPadding(text)

# Encryption
ciphertext = DES(text, key, padding, True)

# Returning ciphertext
return ciphertext

def DESDecryption(key, text, padding):
    """Function for DES Decryption."""

    # Decryption
    plaintext = DES(text, key, padding, False)

    # Removing padding if required
    if padding == True:
        # Removing padding and returning plaintext
        return removePadding(plaintext)

    # Returning plaintext
    return plaintext

# Initial Permutation Matrix for data
initialPermutationMatrix = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,

```

```

59, 51, 43, 35, 27, 19, 11, 3,
61, 53, 45, 37, 29, 21, 13, 5,
63, 55, 47, 39, 31, 23, 15, 7
]

#Expand matrix to get a 48bits matrix of datas to apply the xor with Ki
expandMatrix = [
    32, 1, 2, 3, 4, 5,
    4, 5, 6, 7, 8, 9,
    8, 9, 10, 11, 12, 13,
    12, 13, 14, 15, 16, 17,
    16, 17, 18, 19, 20, 21,
    20, 21, 22, 23, 24, 25,
    24, 25, 26, 27, 28, 29,
    28, 29, 30, 31, 32, 1
]

def DES(text, key, padding, isEncrypt):
    """Function to implement DES Algorithm."""

    # Initializing variables required
    isDecrypt = not isEncrypt

    # Generating keys
    keys = generateKeys(key)

    # Splitting text into 8 byte blocks
    plaintext8byteBlocks = nSplit(text, 8)
    result = []

    # For all 8-byte blocks of text
    for block in plaintext8byteBlocks:

```



```

# Convert the block into bit array
block = stringToBitArray(block)

# Do the initial permutation
block = permutation(block, initialPermutationMatrix)

# Splitting block into two 4 byte (32 bit) sized blocks
leftBlock, rightBlock = nSplit(block, 32)

temp = None

# Running 16 identical DES Rounds for each block of text
for i in range(16):
    # Expand rightBlock to match round key size(48-bit)
    expandedRightBlock = expand(rightBlock, expandMatrix)

    # Xor right block with appropriate key
    if isEncrypt == True:
        # For encryption, starting from first key in normal order
        temp = xor(keys[i], expandedRightBlock)
    elif isDecrypt == True:
        # For decryption, starting from last key in reverse order
        temp = xor(keys[15 - i], expandedRightBlock)

    # Sbox substitution Step
    temp = SboxSubstitution(temp)

    # Permutation Step
    temp = permutation(temp, eachRoundPermutationMatrix)

    # XOR Step with leftBlock
    temp = xor(leftBlock, temp)

# Blocks swapping

```

```

    leftBlock = rightBlock
    rightBlock = temp

    # Final permutation then appending result
    result += permutation(rightBlock + leftBlock, finalPermutationMatrix)

    # Converting bit array to string
    finalResult = bitArrayToString(result)

    return finalResult

# Matrix used for shifting after each round of keys
SHIFT = [1, 1, 2, 2, 2, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1]

# Permutation matrix for key
keyPermutationMatrix1 = [
    57, 49, 41, 33, 25, 17, 9,
    1, 58, 50, 42, 34, 26, 18,
    10, 2, 59, 51, 43, 35, 27,
    19, 11, 3, 60, 52, 44, 36,
    63, 55, 47, 39, 31, 23, 15,
    7, 62, 54, 46, 38, 30, 22,
    14, 6, 61, 53, 45, 37, 29,
    21, 13, 5, 28, 20, 12, 4
]

# Permutation matrix for shifted key to get next key
keyPermutationMatrix2 = [
    14, 17, 11, 24, 1, 5, 3, 28,
    15, 6, 21, 10, 23, 19, 12, 4,
    26, 8, 16, 7, 27, 20, 13, 2,
    41, 52, 31, 37, 47, 55, 30, 40,

```

```
51, 45, 33, 48, 44, 49, 39, 56,  
34, 53, 46, 42, 50, 36, 29, 32  
]
```

```
def generateKeys(key):  
    """Function to generate keys for different rounds of DES."""  
  
    # Initializing variables required  
    keys = []  
    key = stringToBitArray(key)  
  
    # Initial permutation on key  
    key = permutation(key, keyPermutationMatrix1)  
  
    # Split key in to (leftBlock->LEFT), (rightBlock->RIGHT)  
    leftBlock, rightBlock = nSplit(key, 28)  
  
    # 16 rounds of keys  
    for i in range(16):  
        # Do left shifting (different for different rounds)  
        leftBlock, rightBlock = leftShift(leftBlock, rightBlock, SHIFT[i])  
        # Merge them  
        temp = leftBlock + rightBlock  
        # Permutation on shifted key to get next key  
        keys.append(permutation(temp, keyPermutationMatrix2))  
  
    # Return generated keys  
    return keys  
  
# Sboxes used in the DES Algorithm  
SboxesArray = [  
    [  

```

[14, 4, 13, 1, 2, 15, 11, 8, 3, 10, 6, 12, 5, 9, 0, 7],
[0, 15, 7, 4, 14, 2, 13, 1, 10, 6, 12, 11, 9, 5, 3, 8],
[4, 1, 14, 8, 13, 6, 2, 11, 15, 12, 9, 7, 3, 10, 5, 0],
[15, 12, 8, 2, 4, 9, 1, 7, 5, 11, 3, 14, 10, 0, 6, 13],
],

[
[15, 1, 8, 14, 6, 11, 3, 4, 9, 7, 2, 13, 12, 0, 5, 10],
[3, 13, 4, 7, 15, 2, 8, 14, 12, 0, 1, 10, 6, 9, 11, 5],
[0, 14, 7, 11, 10, 4, 13, 1, 5, 8, 12, 6, 9, 3, 2, 15],
[13, 8, 10, 1, 3, 15, 4, 2, 11, 6, 7, 12, 0, 5, 14, 9],
],

[
[10, 0, 9, 14, 6, 3, 15, 5, 1, 13, 12, 7, 11, 4, 2, 8],
[13, 7, 0, 9, 3, 4, 6, 10, 2, 8, 5, 14, 12, 11, 15, 1],
[13, 6, 4, 9, 8, 15, 3, 0, 11, 1, 2, 12, 5, 10, 14, 7],
[1, 10, 13, 0, 6, 9, 8, 7, 4, 15, 14, 3, 11, 5, 2, 12],
],

[
[7, 13, 14, 3, 0, 6, 9, 10, 1, 2, 8, 5, 11, 12, 4, 15],
[13, 8, 11, 5, 6, 15, 0, 3, 4, 7, 2, 12, 1, 10, 14, 9],
[10, 6, 9, 0, 12, 11, 7, 13, 15, 1, 3, 14, 5, 2, 8, 4],
[3, 15, 0, 6, 10, 1, 13, 8, 9, 4, 5, 11, 12, 7, 2, 14],
],

[
[2, 12, 4, 1, 7, 10, 11, 6, 8, 5, 3, 15, 13, 0, 14, 9],
[14, 11, 2, 12, 4, 7, 13, 1, 5, 0, 15, 10, 3, 9, 8, 6],
[4, 2, 1, 11, 10, 13, 7, 8, 15, 9, 12, 5, 6, 3, 0, 14],
[11, 8, 12, 7, 1, 14, 2, 13, 6, 15, 0, 9, 10, 4, 5, 3],

],

[

[12, 1, 10, 15, 9, 2, 6, 8, 0, 13, 3, 4, 14, 7, 5, 11],

[10, 15, 4, 2, 7, 12, 9, 5, 6, 1, 13, 14, 0, 11, 3, 8],

[9, 14, 15, 5, 2, 8, 12, 3, 7, 0, 4, 10, 1, 13, 11, 6],

[4, 3, 2, 12, 9, 5, 15, 10, 11, 14, 1, 7, 6, 0, 8, 13],

],

[

[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],

[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],

[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],

[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],

],

[

[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],

[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],

[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],

[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],

]

]

```
def SboxSubstitution(bitArray):
```

```
    """Function to substitute all the bytes using Sbox."""
```

```
    # Split bit array into 6 sized chunks
```

```
    # For Sbox indexing
```

```
    blocks = nSplit(bitArray, 6)
```

```
    result = []
```

```

for i in range(len(blocks)):
    block = blocks[i]
    # Row number to be obtained from first and last bit
    row = int( str(block[0]) + str(block[5]), 2 )
    # Getting column number from the 2,3,4,5 position bits
    column = int("".join([str(x) for x in block[1:-1]]), 2)
    # Taking value from ith Sbox in ith round
    sboxValue = SboxesArray[i][row][column]
    # Convert the sbox value to binary
    binVal = binValue(sboxValue, 4)
    # Appending to result
    result += [int(bit) for bit in binVal]

# Returning result
return result

```

```

def addPadding(text):
    """Function to add padding according to PKCS5 standard."""

    # Determining padding length
    paddingLength = 8 - (len(text) % 8)
    # Adding paddingLength number of chr(paddingLength) to text
    text += chr(paddingLength) * paddingLength

    # Returning text
    return text

```

```

def removePadding(data):
    """Function to remove padding from plaintext according to PKCS5."""

    # Getting padding length

```

```
paddingLength = ord(data[-1])
```

```
# Returning data with removed padding
```

```
return data[ : -paddingLength]
```

```
def expand(array, table):
```

```
    """Function to expand the array using table."""
```

```
    # Returning expanded result
```

```
    return [array[element - 1] for element in table]
```

```
def permutation(array, table):
```

```
    """Function to do permutation on the array using table."""
```

```
    # Returning permuted result
```

```
    return [array[element - 1] for element in table]
```

```
def leftShift(list1, list2, n):
```

```
    """Function to left shift the arrays by n."""
```

```
    # Left shifting the two arrays
```

```
    return list1[n:] + list1[:n], list2[n:] + list2[:n]
```

```
def nSplit(list, n):
```

```
    """Function to split a list into chunks of size n."""
```

```
    # Chunking and returning the array of chunks of size n
```

```
    # and last remainder
```

```
    return [ list[i : i + n] for i in range(0, len(list), n)]
```

```
def xor(list1, list2):
```

```
    """Function to return the XOR of two lists."""
```

```
    # Returning the xor of the two lists
```

```
    return [element1 ^ element2 for element1, element2 in zip(list1,list2)]
```

```
def binValue(val, bitSize):
```

```
"""Function to return the binary value as a string of given size."""
```

```
binVal = bin(val)[2:] if isinstance(val, int) else bin(ord(val))[2:]
```

```
# Appending with required number of zeros in front
```

```
while len(binVal) < bitSize:
```

```
    binVal = "0" + binVal
```

```
# Returning binary value
```

```
return binVal
```

```
def stringToBitArray(text):
```

```
    """Funtion to convert a string into a list of bits."""
```

```
# Initializing variable required
```

```
bitArray = []
```

```
for letter in text:
```

```
    # Getting binary (8-bit) value of letter
```

```
    binVal = binValue(letter, 8)
```

```
    # Making list of the bits
```

```
    binValArr = [int(x) for x in list(binVal)]
```

```
    # Appending the bits to array
```

```
    bitArray += binValArr
```

```
# Returning answer
```

```
return bitArray
```

```
def bitArrayToString(array):
```

```
    """Function to convert a list of bits to string."""
```

```
# Chunking array of bits to 8 sized bytes
```

```
byteChunks = nSplit(array, 8)
```



```

# Initializing variables required

stringBytesList = []

stringResult = ""

# For each byte

for byte in byteChunks:

    bitsList = []

    for bit in byte:

        bitsList += str(bit)

    stringBytesList.append("".join(bitsList))

result = "".join([chr(int(stringByte, 2)) for stringByte in stringBytesList])

return result

if __name__ == '__main__':

    main()

```

OUTPUT:

```

[4, 11, 2, 14, 15, 0, 8, 13, 3, 12, 9, 7, 5, 10, 6, 1],
[13, 0, 11, 7, 4, 9, 1, 10, 14, 3, 5, 12, 2, 15, 8, 6],
[1, 4, 11, 13, 12, 3, 7, 14, 10, 15, 6, 8, 0, 5, 9, 2],
[6, 11, 13, 8, 1, 4, 10, 7, 9, 5, 0, 15, 14, 2, 3, 12],
],

[
[13, 2, 8, 4, 6, 15, 11, 1, 10, 9, 3, 14, 5, 0, 12, 7],
[1, 15, 13, 8, 10, 3, 7, 4, 12, 5, 6, 11, 0, 14, 9, 2],
[7, 11, 4, 1, 9, 12, 14, 2, 0, 6, 10, 13, 15, 3, 5, 8],
[2, 1, 14, 7, 4, 10, 8, 13, 15, 12, 9, 0, 3, 5, 6, 11],
]

def SboxSubstitution(bitArray):
    """Function to substitute all the bytes using Sbox."""
    # S-box substitution logic

```

Enter the message to be encrypted : hello world
Enter a key of 8 length (64-bits) (characters or numbers only) : computer
Encrypted Ciphertext is : '\x90i7j0\x08-4N\x\x84H\x88Z:4'
Decrypted plaintext is : hello world

RESULT:

Thus implementation of DES algorithm was completed successfully.

Ex No:03 IMPLEMENTATION OF BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

Date:

AIM:

To implement Banker's algorithm for deadlock avoidance using python.

ALGORITHM:

1. Start.
2. Get the input from the user.
3. If request \leq need, go to step 4, else it shows an error.
4. If request \leq available, go to step 5, else it must wait as the resource it requires is not
5. available.
6. If the resulting resource allocation is safe, the process is allocated to resources.
7. If it is unsafe, the old state process is restored.
8. Stop .

PROGRAM:

```
def main():

    # Get input from the user

    n = int(input("Enter the number of processes: ")) # Number of processes

    m = int(input("Enter the number of resource types: ")) # Number of resource types


    # Allocation Matrix

    print("Allocation matrix")

    alloc = []

    for i in range(n):

        alloc.append(list(map(int, input(f"Enter allocation for Process {i}: ").split()))))


    # MAX Matrix

    print("MAX matrix")

    max = []

    for i in range(n):

        max.append(list(map(int, input(f"Enter MAX for Process {i}: ").split()))))


    # Available Resources

    avail = list(map(int, input("Enter the available resources: ").split()))


    # Initialization

    f = [0] * n
```

```

ans = [0] * n
ind = 0
for k in range(n):
    f[k] = 0

need = [[0 for i in range(m)] for i in range(n)]
for i in range(n):
    for j in range(m):
        need[i][j] = max[i][j] - alloc[i][j]

# Applying Banker's Algorithm
for k in range(n):
    for i in range(n):
        if f[i] == 0:
            flag = 0
            for j in range(m):
                if need[i][j] > avail[j]:
                    flag = 1
                    break

            if flag == 0:
                ans[ind] = i
                ind += 1
                for y in range(m):
                    avail[y] += alloc[i][y]
                f[i] = 1

# Determine whether the system is safe or unsafe
safe = all(f)
if safe:
    print("The system is in a safe state.")
    print("The safe sequence:")

```

```

for i in range(n - 1):
    print("P", ans[i], " -> ", sep="", end="")

print("P", ans[n - 1], sep="")

else:

    print("The system is in an unsafe state.")

```

```

if __name__ == "__main__":
    main()

```

OUTPUT:

The screenshot shows a Python IDE with a file named 'Banker's_alg.py'. The code implements the Banker's algorithm, including functions to check for a safe state and print the safe sequence. The terminal output shows the program's execution, where it prompts the user for the number of processes (2), the number of resource types (2), the allocation matrix, the maximum matrix, and the available resources. It then determines that the system is in a safe state and prints the safe sequence: P0 -> P1.

```

48 f[i] = 1
49
50 # Determine whether the system is safe or unsafe
51 safe = all(f)
52 if safe:
53     print("The system is in a safe state.")
54     print("The safe sequence:")
55     for i in range(n - 1):
56         print("P", ans[i], " -> ", sep="", end="")
57     print("P", ans[n - 1], sep="")

```

```

Microsoft Windows [Version 10.0.22631.3885]
(c) Microsoft Corporation. All rights reserved.

E:\Desktop\Assignments and classes\DAA LAB>D:/program/python/python.exe "e:/Desktop/Assignments and classes/DAA LAB/ME_DAA_Lab/Program/Banker's_alg.py"
Enter the number of processes: 2
Enter the number of resource types: 2
Allocation matrix
Enter allocation for Process 0: 1 3
Enter allocation for Process 1: 2 5
MAX matrix
Enter MAX for Process 0: 10 11
Enter MAX for Process 1: 11 20
Enter the available resources: 30 33
The system is in a safe state.
The safe sequence:
P0 -> P1

```

RESULT:

Thus implementation of banker's algorithm was completed successfully.

Ex No:04 IMPLEMENTATION OF RANDOMIZED QUICK SORT USING DIVIDE AND CONQUER STRATEGY

Date:

AIM:

To implement randomized quick sort using divide and conquer strategy.

ALGORITHM:

1. Start the program.
2. Read an input array.
3. Select a pivot randomly, then swap it with the right most position of the array.
4. Fix two pointers i.e., pivot element and the left most element.
5. Pivot is now compared with all other elements, if any number smaller than pivot is found, swap that element with the greatest number found.
6. Repeat step 5 until it reaches the end of the array, then swap the pivot element with the second pointer.
7. Pivot elements are again chosen for the left and right sub arrays separately.
8. Repeat steps from 3 to 6.
9. Print the sorted array.
10. Stop the program.

PROGRAM:

```
import random

# Function to find the partition position
def partition(array, low, high):

    # Choose a random pivot position
    randomNumber = random.randint(low, high)

    # Swap the pivot element with the last element of the array
    array[randomNumber], array[high] = array[high], array[randomNumber]

    # Choose the rightmost element as pivot
    pivot = array[high]

    # Pointer for the greater element
    i = low - 1

    swap = 0

    # Traverse through all elements
    # Compare each element with pivot
    for j in range(low, high):
```

```
if array[j] <= pivot:
```

```
    # If element smaller than pivot is found
```

```
    # Swap it with the greater element pointed by i
```

```
    i += 1
```

```
    array[i], array[j] = array[j], array[i]
```

```
    swap += 1
```

```
# Swap the pivot element with the greater element specified by i
```

```
array[i + 1], array[high] = array[high], array[i + 1]
```

```
# Return the position from where partition is done
```

```
return i + 1
```

```
# Function to perform quicksort
```

```
def quickSort(array, low, high):
```

```
    if low < high:
```

```
        # Find pivot element such that
```

```
        # elements smaller than pivot are on the left
```

```
        # elements greater than pivot are on the right
```

```
        pi = partition(array, low, high)
```

```
        # Recursive call on the left of pivot
```

```
        quickSort(array, low, pi - 1)
```

```
        # Recursive call on the right of pivot
```

```
        quickSort(array, pi + 1, high)
```

```
# Input
```

```
data = []
```

```
n = int(input("\nEnter number of elements: "))
```

```
for i in range(n):
```

```
    ele = int(input("Enter the element: "))
```

```
data.append(ele)
```

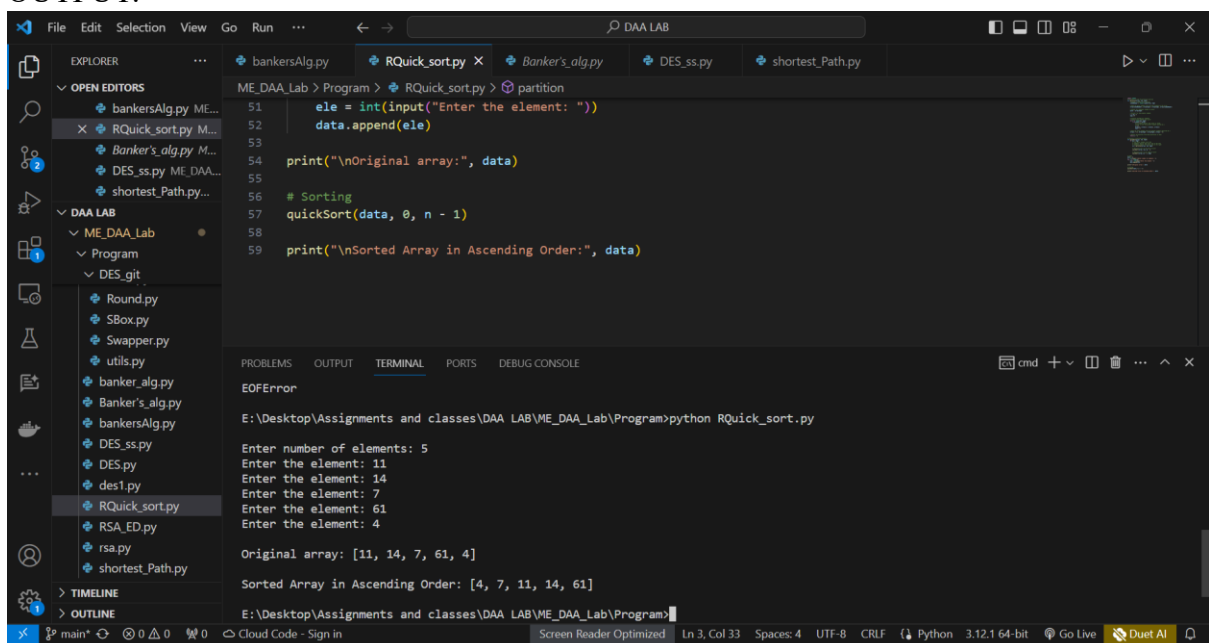
```
print("\nOriginal array:", data)
```

```
# Sorting
```

```
quickSort(data, 0, n - 1)
```

```
print("\nSorted Array in Ascending Order:", data)
```

OUTPUT:



The screenshot displays a Visual Studio Code editor window with a Python file named `RQuick_sort.py` open. The code implements a randomized quick sort algorithm. The terminal output shows the execution of the script, where the user enters 5 elements: 11, 14, 7, 61, and 4. The program then prints the original array and the sorted array in ascending order.

```
51     ele = int(input("Enter the element: "))
52     data.append(ele)
53
54     print("\nOriginal array:", data)
55
56     # Sorting
57     quickSort(data, 0, n - 1)
58
59     print("\nSorted Array in Ascending Order:", data)
```

Terminal Output:

```
EOFError
E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python RQuick_sort.py
Enter number of elements: 5
Enter the element: 11
Enter the element: 14
Enter the element: 7
Enter the element: 61
Enter the element: 4
Original array: [11, 14, 7, 61, 4]
Sorted Array in Ascending Order: [4, 7, 11, 14, 61]
```

RESULT:

Thus, the implementation of randomized quick sort using divide and conquer strategy in python was executed and verified successfully.

Ex No : **IMPLEMENT SHORTEST PATH USING DYNAMIC PROGRAMMING IN A MULTI-STAGED GRAPH**

DATE:

AIM:

To implement shortest path using dynamic programming in a multi staged graph using python.

ALGORITHM:

1. Start.
2. Create a cost table with dimensions (num stages+1)*min-nodes.
3. Initialize all entries to positive infinity except for the destination node in the last stage which is initialize to 0.
4. Iterate through each stage starting from the second-to last stage down to the first.
5. For each node in the current stage starting.
 - a. Calculate the minimum cost to reach each neighbor in the next stage.
 - b. Update the cost table with the minimum cost to reach each node.
6. The entry at position (0,0) in the cost table represents the shortest path cost from source node to the destination node..
7. Return the shortest path cost.
8. Stop.

PROGRAM:

Define a function to find the minimum cost and optimal path in a multistage graph

```
def min_cost_multistage_graph(graph, stages):  
    num_stages = len(stages)  
    num_nodes = len(graph)  
    # Initialize cost and parent arrays  
    cost = [float('inf')] * num_nodes  
    parent = [None] * num_nodes  
    # Set costs for nodes in the first stage to 0  
    for node in stages[0]:  
        cost[node] = 0  
    # Dynamic Programming approach to calculate minimum cost  
    for i in range(1, num_stages):  
        for node in stages[i]:  
            min_cost = float('inf')  
            for parent_node in stages[i - 1]:  
                edge_cost = graph[parent_node][node]  
                total_cost = cost[parent_node] + edge_cost
```



```

        if total_cost < min_cost:
            min_cost = total_cost
            parent[node] = parent_node
            cost[node] = min_cost

# Reconstruct the shortest path
path = [None] * num_stages
path[num_stages - 1] = stages[num_stages - 1][0]
for i in range(num_stages - 2, -1, -1):
    path[i] = parent[path[i + 1]]
return cost[stages[num_stages - 1][0]], path

# Get input from the user
num_nodes = int(input("Enter the number of nodes: "))
graph = [[0] * num_nodes for _ in range(num_nodes)]

# Input weights for edges
for i in range(num_nodes):
    edges = input(f"Enter weights for edges from node {i}: ").split()
    graph[i] = [int(weight) for weight in edges]

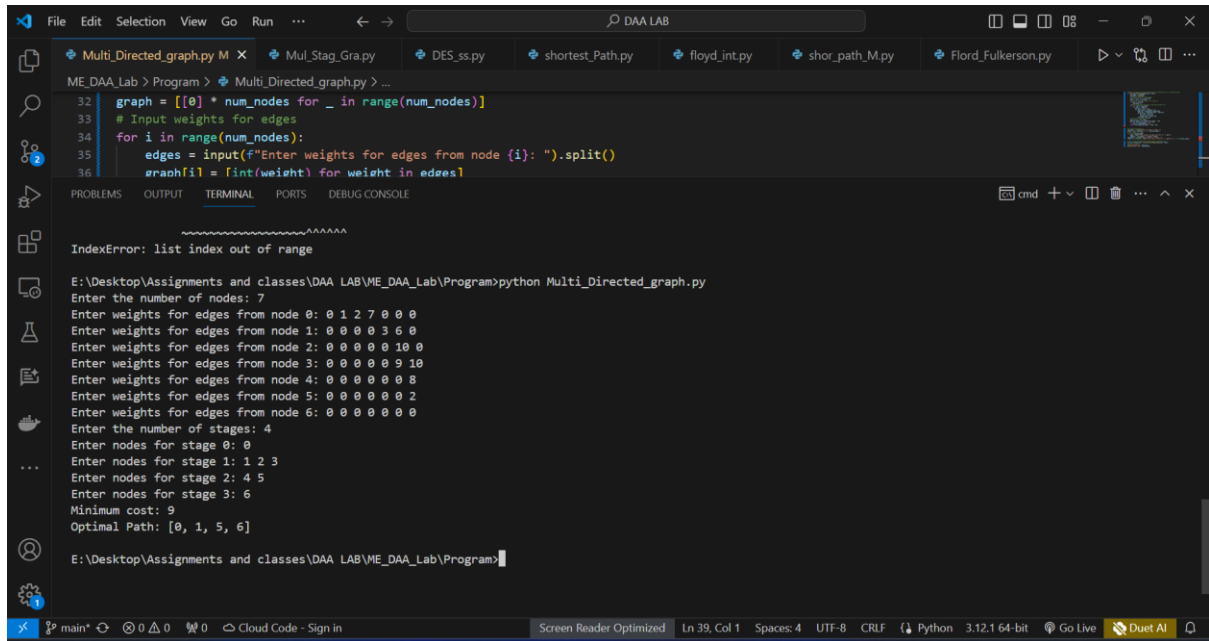
num_stages = int(input("Enter the number of stages: "))
stages = [list(map(int, input(f"Enter nodes for stage {i}: ").split()))) for i in range(num_stages)]

# Call the min_cost_multistage_graph function with user input
min_cost, optimal_path = min_cost_multistage_graph(graph, stages)

# Print the results
print(f"Minimum cost: {min_cost}")
print("Optimal Path:", optimal_path)

```

OUTPUT:



The screenshot shows a Visual Studio Code editor with a Python file named `Multi_Directed_graph.py` open. The code defines a graph with 7 nodes and takes input for edge weights. The terminal output shows the execution of the script, which prompts for the number of nodes (7), edge weights for each node, the number of stages (4), and nodes for each stage. It then calculates the minimum cost (9) and the optimal path ([0, 1, 5, 6]).

```
graph = [[0] * num_nodes for _ in range(num_nodes)]
# Input weights for edges
for i in range(num_nodes):
    edges = input(f"Enter weights for edges from node {i}: ").split()
    graph[i] = [int(weight) for weight in edges]
```

```
~~~~~
IndexError: list index out of range

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python Multi_Directed_graph.py
Enter the number of nodes: 7
Enter weights for edges from node 0: 0 1 2 7 0 0 0
Enter weights for edges from node 1: 0 0 0 0 3 6 0
Enter weights for edges from node 2: 0 0 0 0 0 10 0
Enter weights for edges from node 3: 0 0 0 0 0 9 10
Enter weights for edges from node 4: 0 0 0 0 0 0 8
Enter weights for edges from node 5: 0 0 0 0 0 0 2
Enter weights for edges from node 6: 0 0 0 0 0 0 0
Enter the number of stages: 4
Enter nodes for stage 0: 0
Enter nodes for stage 1: 1 2 3
Enter nodes for stage 2: 4 5
Enter nodes for stage 3: 6
Minimum cost: 9
Optimal Path: [0, 1, 5, 6]

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

RESULT:

Thus, the implementation of shortest path using dynamic programming in a multistage graph using python was executed and verified successfully.

Ex No :

**IMPLEMENTATION OF FORD FULKERSON
ALGORITHM TO COMPUTE THE MAXIMUM
FLOW IN A GRAPH**

DATE:

AIM:

To implement Ford Fulkerson algorithm to compute the maximum flow in a graph using python.

ALGORITHM:

1. Start.
2. Initialize the flow in all the edges to 0.
3. While there is an augmenting path between the source and the sink and add this path to the flow.
4. Repeat search for an s-t path p while it exists.
5. Find if there is a path from s to t using breadth first search. A path exists if $f(e) < c(e)$ for every edge e on the path.
6. If no path found, return max flow.
7. Else find minimum edge value for path p.
8. $\text{Flow} = \min(c(e) - f(e))$ for path p, $\text{max_flow} += \text{flow}$.
9. For all edge e of path incremented flow, $f(e) += \text{flow}$.
10. Update the residual graph.
11. Stop.

PROGRAM:

class Graph:

```
    def __init__(self, vertices):
```

```
        self.V = vertices
```

```
        self.graph = [[0] * vertices for _ in range(vertices)]
```

```
    def is_valid_edge(self, u, v, capacity):
```

```
        return 0 <= u < self.V and 0 <= v < self.V and capacity >= 0
```

```
    def add_edge(self, u, v, capacity):
```

```
        if self.is_valid_edge(u, v, capacity):
```

```
            self.graph[u][v] = capacity
```

```
    def print_solution(self, flow):
```

```
        print("Edge\tFlow")
```

```
        for i in range(self.V):
```

```
            for j in range(self.V):
```

```
                if flow[i][j] > 0:
```

```
                    print(f' {i} -> {j} \t {flow[i][j]}')
```

```
    def ford_fulkerson(self, source, sink):
```

```
        parent = [-1] * self.V
```

```

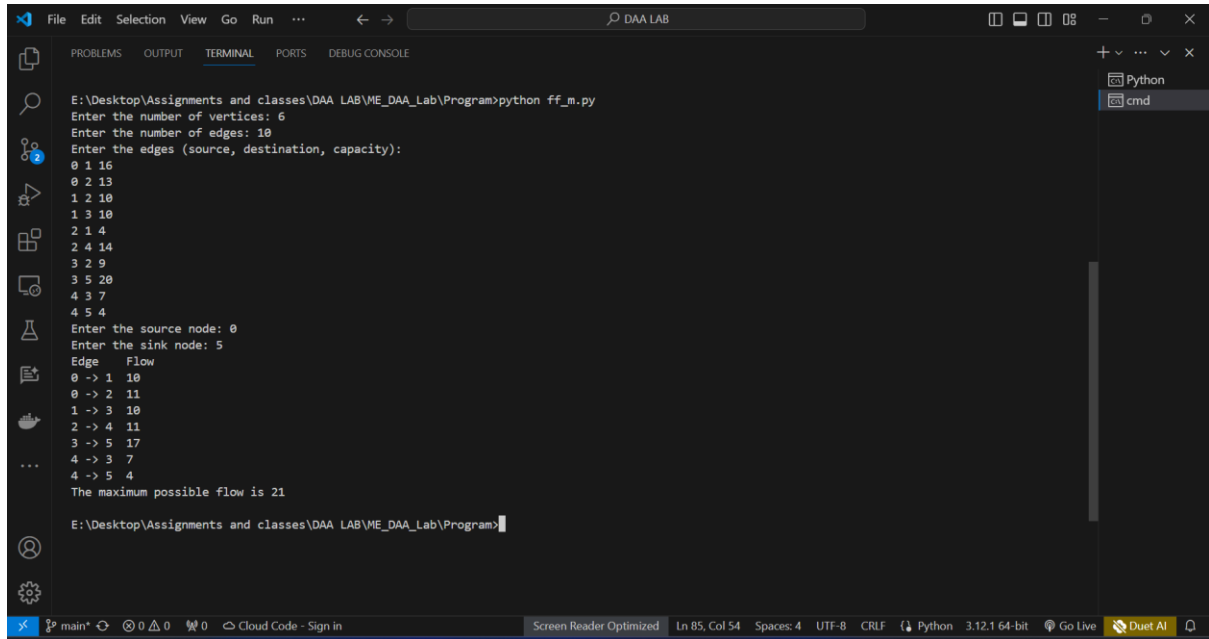
max_flow = 0
flow = [[0] * self.V for _ in range(self.V)]
while self.bfs(source, sink, parent):
    path_flow = float("inf")
    s = sink
    while s != source:
        path_flow = min(path_flow, self.graph[parent[s]][s])
        s = parent[s]
    max_flow += path_flow
    v = sink
    while v != source:
        u = parent[v]
        flow[u][v] += path_flow
        flow[v][u] -= path_flow
        v = parent[v]
    # Update the residual capacities of edges along the path
    v = sink
    while v != source:
        u = parent[v]
        self.graph[u][v] -= path_flow
        self.graph[v][u] += path_flow
        v = parent[v]
    # Reset parent array for next BFS
    parent = [-1] * self.V
    self.print_solution(flow)
    return max_flow
def bfs(self, source, sink, parent):
    visited = [False] * self.V
    queue = [source]
    visited[source] = True
    while queue:
        u = queue.pop(0)

```

```
        for ind, val in enumerate(self.graph[u]):
            if not visited[ind] and val > 0:
                queue.append(ind)
                visited[ind] = True
                parent[ind] = u
        return visited[sink]

if __name__ == "__main__":
    vertices = int(input("Enter the number of vertices: "))
    g = Graph(vertices)
    edges = int(input("Enter the number of edges: "))
    print("Enter the edges (source, destination, capacity): ")
    for _ in range(edges):
        u, v, capacity = map(int, input().split())
        g.add_edge(u, v, capacity)
    source = int(input("Enter the source node: "))
    sink = int(input("Enter the sink node: "))
    max_flow = g.ford_ulkerson(source, sink)
    print(f"The maximum flow is {max_flow}")
```

OUTPUT:



The screenshot shows a Visual Studio Code terminal window with the following content:

```
E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python ff_m.py
Enter the number of vertices: 6
Enter the number of edges: 10
Enter the edges (source, destination, capacity):
0 1 16
0 2 13
1 2 10
1 3 10
2 1 4
2 4 14
3 2 9
3 5 20
4 3 7
4 5 4
Enter the source node: 0
Enter the sink node: 5
Edge      Flow
0 -> 1    10
0 -> 2    11
1 -> 3    10
2 -> 4    11
3 -> 5    17
4 -> 3     7
4 -> 5     4
The maximum possible flow is 21

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

The terminal window has a menu bar (File, Edit, Selection, View, Go, Run, ...), a toolbar, and a sidebar with icons for Explorer, Search, Source Control, Run and Debug, and Extensions. The bottom status bar shows 'main*' and various settings like 'Screen Reader Optimized', 'Ln 85, Col 54', 'Spaces: 4', 'UTF-8', 'CRLF', 'Python 3.12.1 64-bit', 'Go Live', and 'Duet AI'.

RESULT:

Thus, the implementation of Ford Fulkerson algorithm to compute maximum flow in a graph using python was executed and verified successfully.

Ex No :

IMPLEMENTATION OF BOYER-MOORE ALGORITHM FOR PATTERN SEARCHING

DATE:

AIM:

Implementation of boyer-moore algorithm for pattern searching

ALGORITHM:

1. Start.
2. Create Bad Character Shift table for pattern characters.
3. Create Good Suffix Shift table.
4. Start matching from the right end of the pattern.
5. If a mismatch occurs:
 - a. Use Bad Character Shift to calculate the shift based on the mismatched character.
 - b. Use Good Suffix Shift for additional shifts.
 - c. Shift the pattern by the maximum of the calculated values.
6. Continue matching until a match is found or the end of the text is reached.
7. Return the starting index of the first occurrence if found, otherwise, return -1.
8. Stop.

PROGRAM:

```
def preprocess_pattern(pattern):  
    bad_char_shift = {}  
    pattern_length = len(pattern)  
    for i in range(pattern_length - 1):  
        bad_char_shift[pattern[i]] = pattern_length - i - 1  
    return bad_char_shift  
  
def boyer_moore_search(text, pattern):  
    bad_char_shift = preprocess_pattern(pattern)  
    text_length = len(text)  
    pattern_length = len(pattern)  
    occurrences = []  
    i = pattern_length - 1  
    while i < text_length:  
        j = pattern_length - 1  
        k = i  
        while j >= 0 and text[k] == pattern[j]:  
            k -= 1  
            j -= 1  
        if j == -1:  
            occurrences.append(i)
```

```
        occurrences.append(k + 1)

        bad_char_shift_value = bad_char_shift.get(text[i], pattern_length)
        i += max(1, pattern_length - j, bad_char_shift_value)
    else:
        bad_char_shift_value = bad_char_shift.get(text[i], pattern_length)
        i += max(1, bad_char_shift_value)

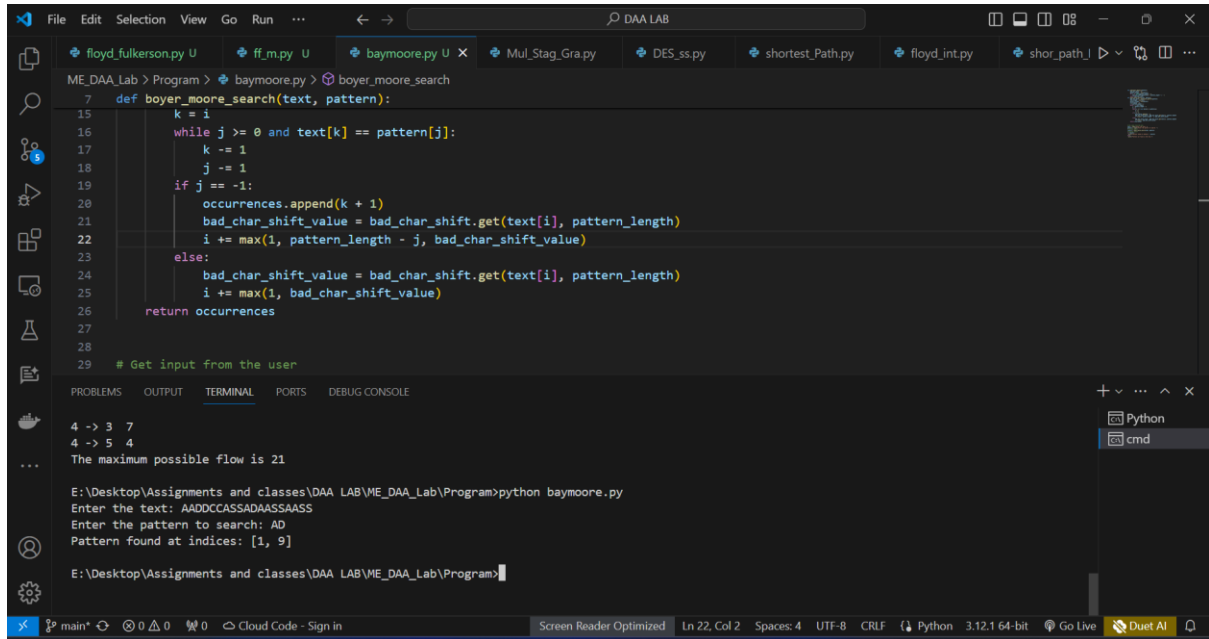
    return occurrences
```

```
# Get input from the user
text = input("Enter the text: ")
pattern = input("Enter the pattern to search: ")

# Perform search
results = boyer_moore_search(text, pattern)

# Display result
if results:
    print("Pattern found at indices:", results)
else:
    print("Pattern not found in the text.")
```


OUTPUT:



```
File Edit Selection View Go Run ... DAA LAB
floyd_fulkerson.py U ff_m.py U baymoore.py U X Mul_Stag_Gra.py DES_ss.py shortest_Path.py floyd_int.py shor_path_I
ME_DAA_Lab > Program > baymoore.py > boyer_moore_search
7 def boyer_moore_search(text, pattern):
15     k = i
16     while j >= 0 and text[k] == pattern[j]:
17         k -= 1
18         j -= 1
19     if j == -1:
20         occurrences.append(k + 1)
21         bad_char_shift_value = bad_char_shift.get(text[i], pattern_length)
22         i += max(1, pattern_length - j, bad_char_shift_value)
23     else:
24         bad_char_shift_value = bad_char_shift.get(text[i], pattern_length)
25         i += max(1, bad_char_shift_value)
26     return occurrences
27
28
29 # Get input from the user

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE
4 -> 3 7
4 -> 5 4
The maximum possible flow is 21

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python baymoore.py
Enter the text: AADDCCASSADAASSAASS
Enter the pattern to search: AD
Pattern found at indices: [1, 9]

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

RESULT:

Thus, the implementation of Boyer-Moore algorithm for pattern searching was executed and verified successfully.

Ex No :

**SOLVE THE GRAPH COLORING PROBLEM BY
BACKTRACKING AND CONSTRAINT
PROPAGATION (USING HEURISTICS)**

DATE:

AIM:

To solve the graph colouring problem by backtracking and constraint propagation using heuristics in python.

ALGORITHM:

1. Start.
2. Get the number of vertices and edges from the user.
3. If the nearby edge is adjacent, colour the edge with different colour.
4. Else nearby edge is not adjacent, colour the edge with same colour.
5. Create a recursive function that takes the graph, current index, number of vertices and output colour array.
6. If the current index is equal to the number of vertices, print the colour configuration in the output array.
7. Assign a colour to vertex (1 to m).
8. For every assigned colour, check if the configuration is safe.
9. If any recursive function returns the true, break the loop and return true.
10. If no recursive function returns true, then return false.
11. Stop.

PROGRAM:

```
import matplotlib.pyplot as plt
```

```
import networkx as nx
```

```
class Graph:
```

```
    def __init__(self, vertices):
```

```
        self.vertices = vertices
```

```
        self.graph = [[0 for _ in range(vertices)] for _ in range(vertices)]
```

```
    def add_edge(self, u, v):
```

```
        self.graph[u][v] = 1
```

```
        self.graph[v][u] = 1
```

```
    def is_safe(self, v, color, c):
```

```
        for i in range(self.vertices):
```

```
            if self.graph[v][i] == 1 and color[i] == c:
```

```
                return False
```

```
        return True
```

```

def graph_coloring_util(self, m, color, v):
    if v == self.vertices:
        return True

    for c in range(1, m + 1):
        if self.is_safe(v, color, c):
            color[v] = c
            if self.graph_coloring_util(m, color, v + 1):
                return True
            color[v] = 0

def graph_coloring(self, m):
    color = [0] * self.vertices
    if not self.graph_coloring_util(m, color, 0):
        print("No solution exists")
        return False

    print("Solution exists with the following coloring:")
    for c in color:
        print(c, end=" ")

    # Plotting the colored graph
    G = nx.Graph()
    for i in range(self.vertices):
        G.add_node(i)
    for i in range(self.vertices):
        for j in range(i + 1, self.vertices):
            if self.graph[i][j] == 1:
                G.add_edge(i, j)

    node_colors = [color[i] for i in range(self.vertices)]

```

```
        nx.draw(G, with_labels=True, node_color=node_colors, cmap=plt.cm.rainbow, node_size=1000)

        plt.show()

        return True

if __name__ == "__main__":
    # Get input from the user

    vertices = int(input("Enter the number of vertices: "))
    edges = int(input("Enter the number of edges: "))

    g = Graph(vertices)

    # Get edges from the user
    print("Enter the edges (vertex1 vertex2):")
    for _ in range(edges):
        u, v = map(int, input().split())
        g.add_edge(u, v)

    m = 3 # Number of colors
    g.graph_coloring(m)
```

OUTPUT:

The screenshot shows a VS Code editor with the following components:

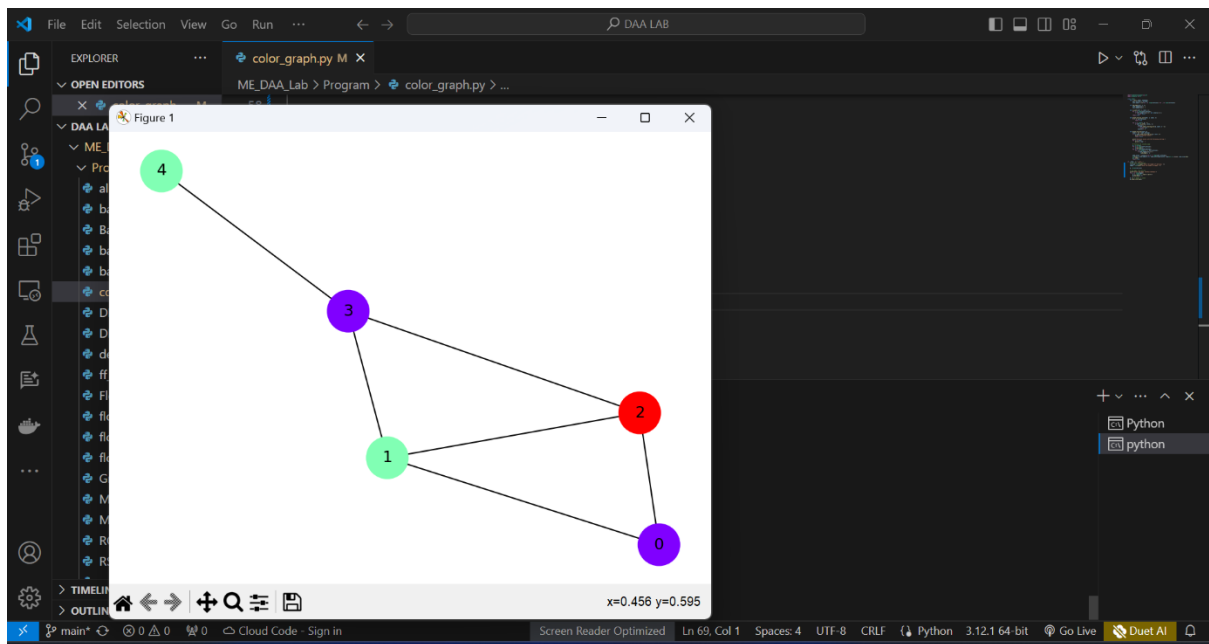
- Explorer Sidebar:**
 - Project structure: `color_graph.py` (selected), `ME_DAA_Lab` (expanded), `Program` (expanded), `all_pair_shortPath.py`, `banker_alg.py`, `Banker's_alg.py`, `bankersAlg.py`, `baymoore.py`.
- Main Editor:**
 - File: `color_graph.py` (selected).
 - Code content:


```
g = Graph(vertices)

# Get edges from the user
print("Enter the edges (vertex1 vertex2):")
for _ in range(edges):
    u, v = map(int, input().split())
    g.add_edge(u, v)

m = 3 # Number of colors
```
- Output Panel:**
 - Tab: `python` (selected).
 - Content:


```
E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\python color_graph.py
Enter the number of edges: 6
Enter the edges (vertex1 vertex2):
0 1
0 2
1 2
1 3
2 3
3 4
Solution exists with the following coloring:
```



RESULT:

Thus, the graph colouring problem by backtracking and constraint propagation using heuristics in python was executed and verified successfully.

Ex No : 9

**IMPLEMENTATION OF MAXIMUM CLIQUES
PROBLEM USING THE BRANCH AND CUT
ALGORITHM**

DATE:

AIM:

To implementation of maximum cliques problem using the branch and cut algorithm.

ALGORITHM:

1. Start.
2. Selection an undirected graph which consists of vertices and edges.
3. Use a heuristics or algorithm to find lower bound on the maximum clique size
4. Choose a vertex from the graph and branch in to subproblem.
 - a. In one subproblem, Include the chosen vertex in the clique and remove all vertices adjacent to it.
 - b. In other subproblem, exclude the chosen vertex form the cliques and remove it from the graph.
5. Solve each sub problem using relaxation techniques i.e. finding maximum cliques in the reduced graph.
6. Repeat the steps 4 and 5 for each subproblem until an optimal clique is found or the maximum cliques size equals the lower bound .
7. Stop.

PROGRAM:

MAX = 100

Stores the vertices

store = [0] * MAX

Graph

graph = [[0] * MAX for _ in range(MAX)]

Degree of the vertices

d = [0] * MAX

Function to check if the given set of vertices

in the store array is a clique or not

def is_clique(b):

 # Run a loop for all the set of edges

 # for the select vertex

 for i in range(1, b):

 for j in range(i + 1, b):

```

        # If any edge is missing
        if graph[store[i]][store[j]] == 0:
            return False
    return True

# Function to print the clique
def print_cli(n):
    for i in range(1, n):
        print(store[i], end=" ")
    print(", ", end=" ")

# Function to find all the cliques of size s
def findCliques(i, l, s):
    # Check if any vertices from i+1 can be inserted
    for j in range(i + 1, n - (s - l) + 1):
        # If the degree of the graph is sufficient
        if d[j] >= s - l:
            # Add the vertex to store
            store[l] = j
            # If the graph is not a clique of size k
            # then it cannot be a clique by adding another edge
            if is_clique(l + 1):
                # If the length of the clique is still less than the desired size
                if l < s:
                    # Recursion to add vertices
                    findCliques(j, l + 1, s)
                # Size is met
            else:
                print_cli(l + 1)

# Driver code
if __name__ == "__main__":

```

```

edges = int(input("Enter the number of edges: "))
print("Enter the edges (format: vertex1 vertex2):")
for _ in range(edges):
    edge = list(map(int, input().split()))
    graph[edge[0]][edge[1]] = 1
    graph[edge[1]][edge[0]] = 1
    d[edge[0]] += 1
    d[edge[1]] += 1

k = int(input("Enter the size of the clique (k): "))
n = int(input("Enter the number of vertices: "))

findCliques(0, 1, k)

```

OUTPUT:

```

Program > maxCliques.py > ...
31 def findCliques(i, l, s):
32     # Recursion to add vertices
33     findCliques(j, l + 1, s)
34     # Size is met
35     else:
36         print_cli(l + 1)
37
38 # Driver code
39 if __name__ == "__main__":
40     edges = int(input("Enter the number of edges: "))
41     print("Enter the edges (format: vertex1 vertex2):")
42     for _ in range(edges):
43
44
45
46
47
48
49 # Driver code
50 if __name__ == "__main__":
51     edges = int(input("Enter the number of edges: "))
52     print("Enter the edges (format: vertex1 vertex2):")
53     for _ in range(edges):
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

PROBLEMS OUTPUT TERMINAL PORTS DEBUG CONSOLE

```

1 2 3 ,
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python maxCliques.py
Enter the number of edges: 6
Enter the edges (format: vertex1 vertex2):
1 2
2 3
3 1
4 3
4 5
5 3
Enter the size of the clique (k): 3
Enter the number of vertices: 5
1 2 3 , 3 4 5 ,
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>

```

RESULT:

Thus an implementation of maximum clique problem using branch and cut method was executed and verified successfully.

Ex No : 10

IMPLEMENTATION OF CLOCK SYNCHRONISATION ALGORITHM

DATE:

AIM:

To implement clock synchronization algorithms using python.

ALGORITHM:

BERKLEY ALGORITHM:

1. Start.
2. Each node in the distributed system periodically sends its local time to a designated master node.
3. Choose one of the node to act as the master node responsible for coordinating the clock synchronization process.
4. The master node collects the local times sent by all the participating nodes in the system.
5. The master node calculates the average of the collected local times.
6. Calculate the difference between each node's local time and the average time calculated by the master node.
7. The master node send the time adjustment values back to each participating node.
8. Each participating node adjustment value recived from the master node.
9. Repeat the process to ensure continuous synchronization in the distributed system.
10. Stop.

CRISTIAN ALGORITHM:

1. The process on the client machine sends the request for fetching clock time to clock server at time t_0 .
2. The clock server listens to the request made by the client process and return the response in from of clock server time.
3. The client process fetches the response from the clock server at time T_1 and calculates the synchronized client clock time using the formula.

$$T_{client} = T_{server} + (T_1 - T_0)/2$$

PROGRAM:

BERKLEY ALGORITHM:

CLIENT:

```
import socket
```

```
import time
```

```
HOST = 'localhost'
```

```
PORT = 5000
```

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
client_socket.connect((HOST, PORT))
```

```
# Get client timestamp before sending
```

```
client_timestamp = time.time()

# Send client timestamp
client_socket.sendall(str(client_timestamp).encode())

# Receive server response (server timestamp and RTT)
data = client_socket.recv(1024).decode()
server_timestamp, rtt = data.split(',')

# Convert server timestamp to float
server_timestamp = float(server_timestamp)

# Convert RTT to float before subtraction
rtt = float(rtt)

# Calculate estimated server time (offset)
estimated_server_time = server_timestamp - rtt

# Print estimated server time
print('Estimated server time:', estimated_server_time)

client_socket.close()

SERVER:
import socket
import time

HOST = 'localhost'
PORT = 5000

server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind((HOST, PORT))
server_socket.listen()
```

```
print('Server started and listening on port:', PORT)
```

```
while True:
```

```
    conn, addr = server_socket.accept()
```

```
    print('Connected by', addr)
```

```
    # Receive timestamp from client
```

```
    data = conn.recv(1024).decode()
```

```
    client_timestamp = float(data)
```

```
    # Get server timestamp
```

```
    server_timestamp = time.time()
```

```
    # Calculate round trip time (RTT)
```

```
    rtt = (server_timestamp - client_timestamp) / 2
```

```
    # Send server timestamp and RTT
```

```
    response = str(server_timestamp) + ';' + str(rtt)
```

```
    conn.sendall(response.encode())
```

```
    conn.close()
```

```
server_socket.close()
```

CRISTIAN ALGORITHM:

CLIENT:

```
import socket
```

```
def client():
```

```
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

```
    s.connect(('localhost', 12345))
```

```
    while True:
```

```

        cmd = input("Enter TIME to get server time, or EXIT to quit: ").upper()
        s.send(cmd.encode())
        if cmd == 'EXIT':
            break
        elif cmd == 'TIME':
            server_time = float(s.recv(1024).decode())
            print(f"Server time: {server_time}")

    s.close()

if __name__ == "__main__":
    client()

SERVER
import socket
import time

def server():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind(('localhost', 12345))
    s.listen(1)
    print("Server is listening...")
    conn, addr = s.accept()
    print(f"Connected to {addr}")

    while True:
        data = conn.recv(1024).decode()
        if data == 'TIME':
            conn.send(str(time.time()).encode())
        elif data == 'EXIT':
            break

    conn.close()

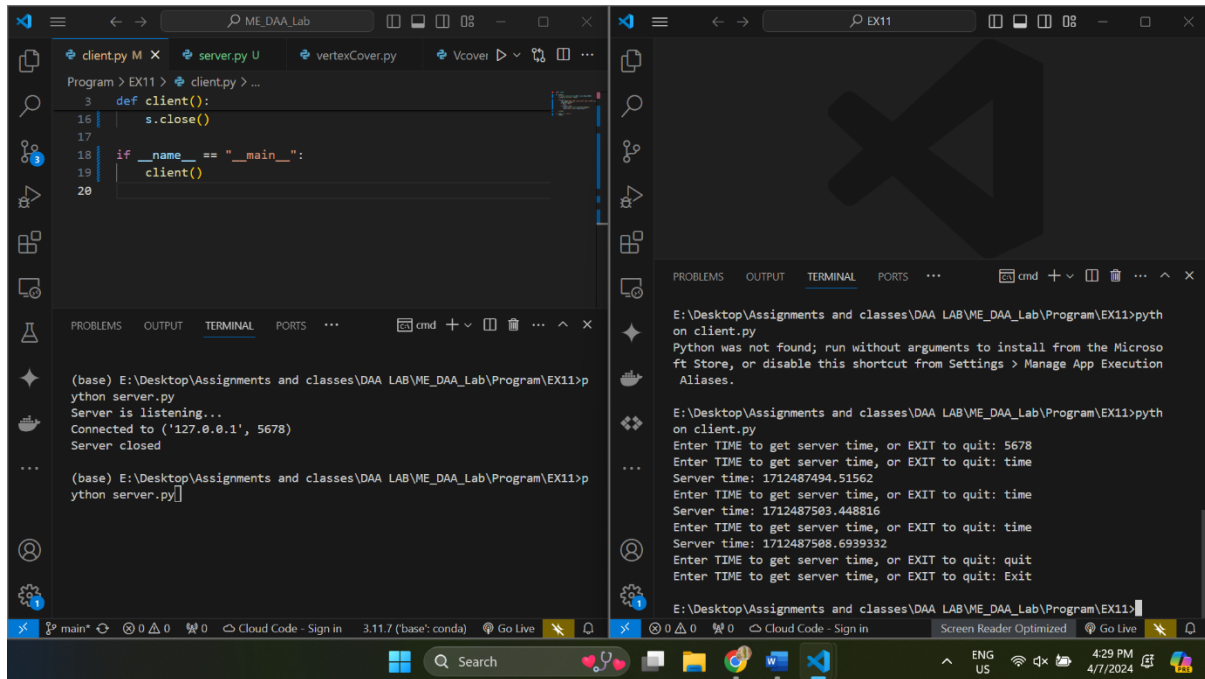
```

```
print("Server closed")
```

```
if __name__ == "__main__":
```

```
    server()
```

OUTPUT:



```
client.py M x server.py U vertexCover.py Vcover
Program > EX11 > client.py > ...
3 def client():
16     s.close()
17
18 if __name__ == "__main__":
19     client()
20

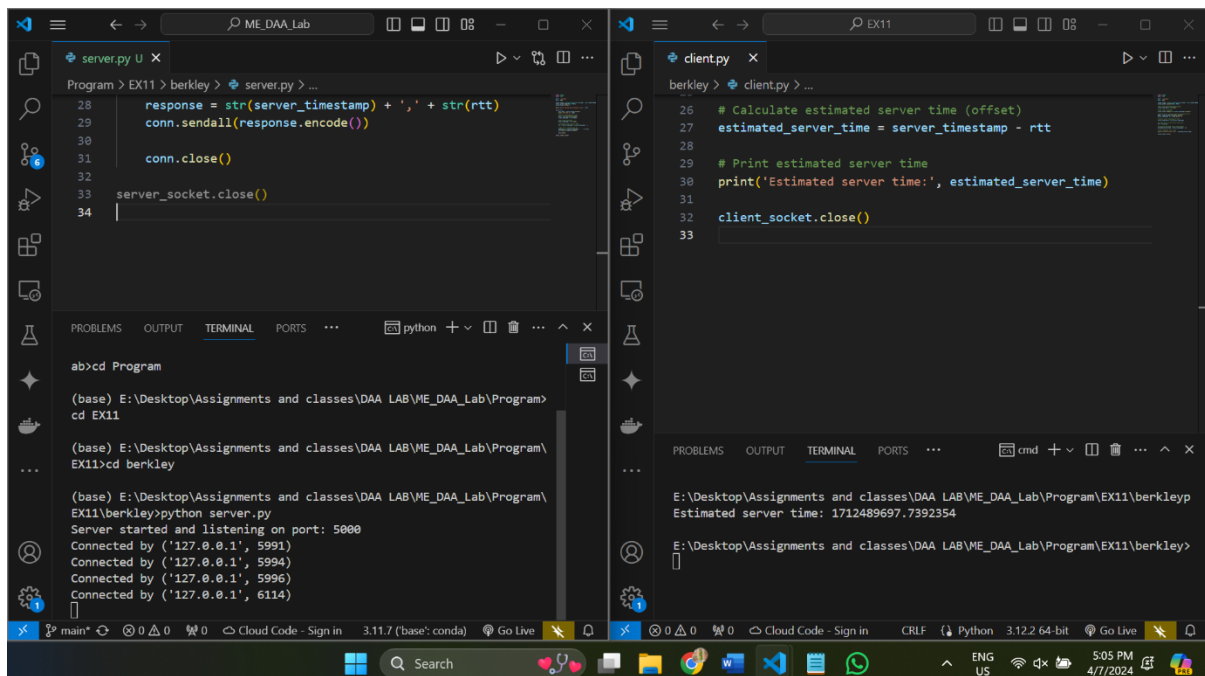
PROBLEMS OUTPUT TERMINAL PORTS ...
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>python server.py
Server is listening...
Connected to ('127.0.0.1', 5678)
Server closed

(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>python server.py

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>python client.py
Python was not found; run without arguments to install from the Microsoft Store, or disable this shortcut from Settings > Manage App Execution Aliases.

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>python client.py
Enter TIME to get server time, or EXIT to quit: 5678
Enter TIME to get server time, or EXIT to quit: time
Server time: 1712487494.51562
Enter TIME to get server time, or EXIT to quit: time
Server time: 1712487503.448816
Enter TIME to get server time, or EXIT to quit: time
Server time: 1712487508.6939332
Enter TIME to get server time, or EXIT to quit: quit
Enter TIME to get server time, or EXIT to quit: Exit

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>
```



```
server.py U
Program > EX11 > server.py > ...
28 response = str(server_timestamp) + ',' + str(rtt)
29 conn.sendall(response.encode())
30
31 conn.close()
32
33 server_socket.close()
34

PROBLEMS OUTPUT TERMINAL PORTS ...
ab>cd Program
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>cd EX11
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11>cd berkley
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11\berkley>python server.py
Server started and listening on port: 5000
Connected by ('127.0.0.1', 5991)
Connected by ('127.0.0.1', 5994)
Connected by ('127.0.0.1', 5996)
Connected by ('127.0.0.1', 6114)

client.py x
berkley > client.py > ...
26 # Calculate estimated server time (offset)
27 estimated_server_time = server_timestamp - rtt
28
29 # Print estimated server time
30 print('Estimated server time:', estimated_server_time)
31
32 client_socket.close()
33

PROBLEMS OUTPUT TERMINAL PORTS ...
E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11\berkley>
Estimated server time: 1712489697.7392354

E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program\EX11\berkley>
```

RESULT:

Thus the implementation of clock synchronization algorithm using python was executed successfully.

Ex No : 11

IMPLEMENTATION OF CROSSEWORD PUZZLES AS CONSTRAINT SATISFACTION PROBLEM

DATE:

AIM:

To implement crossword puzzle as constant satisfaction problem using python.

ALGORITHM:

1. Start.
2. Sort all the words by length in descending order.
3. Take the first word and place it on the board.
4. Take next word.
5. If there is a possible location for this word, loop through all the words that are on the board and check to see if the new interfaces.
6. If the word doesn't break the board then place it there and go to step 4 otherwise go to step 5.
7. Stop.

PROGRAM:

```
ways = 0
```

```
def printMatrix(matrix, n):
```

```
    for i in range(n):
```

```
        print(matrix[i])
```

```
def checkHorizontal(x, y, matrix, currentWord):
```

```
    n = len(currentWord)
```

```
    for i in range(n):
```

```
        if matrix[x][y + i] == '#' or matrix[x][y + i] == currentWord[i]:
```

```
            matrix[x] = matrix[x][:y + i] + currentWord[i] + matrix[x][y + i + 1:]
```

```
        else:
```

```
            matrix[0] = "@"
```

```
            return matrix
```

```
    return matrix
```

```
def checkVertical(x, y, matrix, currentWord):
```

```
    n = len(currentWord)
```

```
    for i in range(n):
```

```

    if matrix[x + i][y] == '#' or matrix[x + i][y] == currentWord[i]:
        matrix[x + i] = matrix[x + i][:y] + currentWord[i] + matrix[x + i][y + 1:]
    else:

        matrix[0] = "@"
        return matrix
    return matrix
def solvePuzzle(words, matrix, index, n):
    global ways
    if index < len(words):
        currentWord = words[index]
        maxLen = n - len(currentWord)

        for i in range(n):
            for j in range(maxLen + 1):
                temp = checkVertical(j, i, matrix.copy(), currentWord)
                if temp[0] != "@":
                    solvePuzzle(words, temp, index + 1, n)

            for i in range(n):
                for j in range(maxLen + 1):
                    temp = checkHorizontal(i, j, matrix.copy(), currentWord)
                    if temp[0] != "@":
                        solvePuzzle(words, temp, index + 1, n)
        else:

            # Calling of print function to
            # Print the crossword puzzle
            print(str(ways + 1) + " way to solve the puzzle ")

```

```
printMatrix(matrix, n)
```

```
print()
```

```
# Increase the ways
```

```
ways += 1
```

```
return
```

```
if __name__ == '__main__':
```

```
n1 = 10
```

```
matrix = []
```

```
matrix.append("#*****")
```

```
matrix.append("#*****")
```

```
matrix.append("#*****#****")
```

```
matrix.append("###***###")
```

```
matrix.append("#*****#****")
```

```
matrix.append("#*****#****")
```

```
matrix.append("#*****#****")
```

```
matrix.append("#*#####")
```

```
matrix.append("#*****")
```

```
matrix.append("***#####")
```

```
words = []
```

```
words.append("PUNJAB")
```

```
words.append("JHARKHAND")
```

```
words.append("MIZORAM")
```



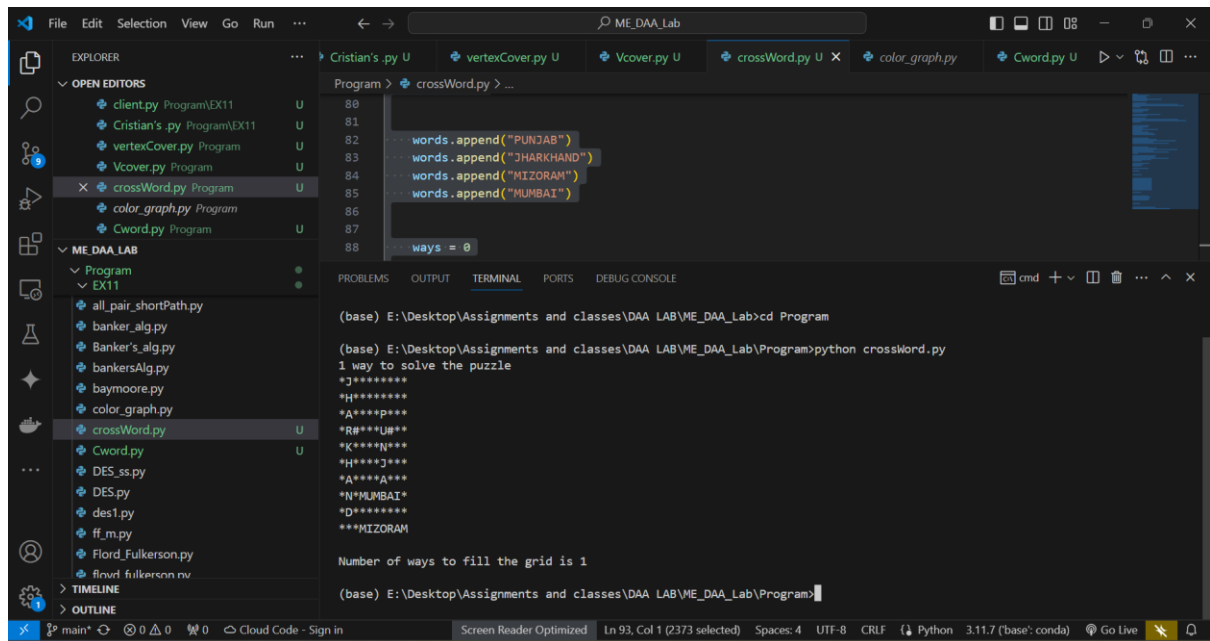
```
words.append("MUMBAI")
```

```
ways = 0
```

```
solvePuzzle(words, matrix, 0, n1)
```

```
print("Number of ways to fill the grid is " + str(ways))
```

OUTPUT:



The screenshot shows a Visual Studio Code editor window with the file explorer on the left and the code editor in the center. The file explorer shows a project named 'ME_DAA_Lab' with a folder 'Program' containing several Python files. The file 'crossWord.py' is selected. The code editor shows the following code:

```
80
81
82 words.append("PUNJAB")
83 words.append("JHARKHAND")
84 words.append("MIZORAM")
85 words.append("MUMBAI")
86
87
88 ways = 0
```

The terminal window at the bottom shows the execution of the script:

```
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab>cd Program
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python crossWord.py
1 way to solve the puzzle
*J*****
*H*****
*A****p***
*R****Ub**
*K****N***
*H****J***
*A****A***
*N*MUMBAI*
*D*****
***MIZORAM

Number of ways to fill the grid is 1
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

RESULT:

Thus the implementation of crossed puzzle as constraint satisfaction problem using python was executed and verified successfully.

Ex No : 12

IMPLEMENTATION OF VERTEX COVER PROBLEM

DATE:

AIM:

To implement the vertex cover problem using python.

ALGORITHM:

1. Start.
2. Initialize the result as {}.
3. Do the following while E is not empty.
4. Pick an arbitrary edge (u, v) from set E and add |u| and |v| to result.
5. Remove all edges from E which are either incident on u or v.
6. Return the result.
7. Stop.

PROGRAM:

class Graph:

```
def __init__(self, num_nodes):
```

```
    self.num_nodes = num_nodes
```

```
    self.adj_list = [[] for _ in range(num_nodes)]
```

```
def add_edge(self, u, v):
```

```
    self.adj_list[u].append(v)
```

```
    self.adj_list[v].append(u)
```

```
def print_graph(self):
```

```
    for node in range(self.num_nodes):
```

```
        print(f"Adjacent nodes of node {node}: {self.adj_list[node]}")
```

```
def print_vertex_cover(self):
```

```
    visited = [False] * self.num_nodes
```

```
    for u in range(self.num_nodes):
```

```
        if not visited[u]:
```

```
            for v in self.adj_list[u]:
```

```
                if not visited[v]:
```

```
                    visited[u] = True
```

```
        visited[v] = True
        break
    print("Vertex Cover:")
    for i in range(self.num_nodes):
        if visited[i]:
            print(i, end=" ")
    print()

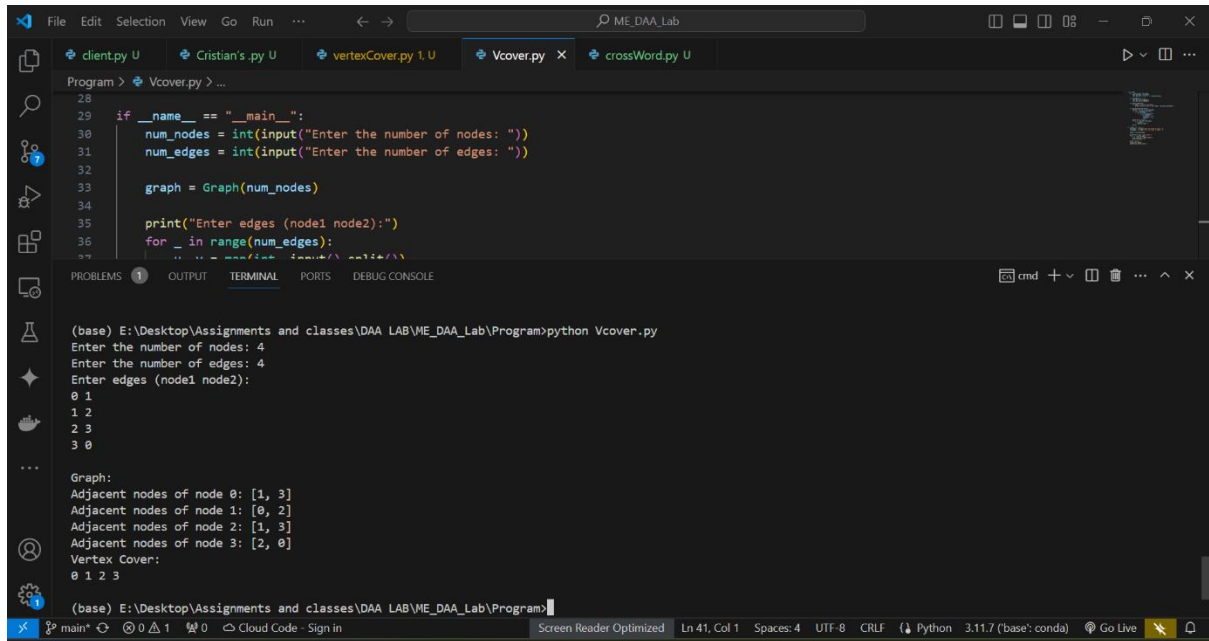
if __name__ == "__main__":
    num_nodes = int(input("Enter the number of nodes: "))
    num_edges = int(input("Enter the number of edges: "))

    graph = Graph(num_nodes)

    print("Enter edges (node1 node2):")
    for _ in range(num_edges):
        u, v = map(int, input().split())
        graph.add_edge(u, v)

    print("\nGraph:")
    graph.print_graph()
    graph.print_vertex_cover()
```

OUTPUT:



The screenshot shows a Visual Studio Code editor window with a Python file named `Vcover.py` open. The code defines a graph with 4 nodes and 4 edges, and prints the adjacent nodes for each node. The terminal output shows the execution of the script, which prompts the user to enter the number of nodes (4) and the number of edges (4), then prompts for the edges (0 1, 1 2, 2 3, 3 0). The output shows the graph structure and the vertex cover [0, 1, 2, 3].

```
28
29 if __name__ == "__main__":
30     num_nodes = int(input("Enter the number of nodes: "))
31     num_edges = int(input("Enter the number of edges: "))
32
33     graph = Graph(num_nodes)
34
35     print("Enter edges (node1 node2):")
36     for _ in range(num_edges):
37         node1, node2 = map(int, input().split())
38         graph.add_edge(node1, node2)
39
40     graph.print_adjacent_nodes()
41     graph.vertex_cover()
42
43 (base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>python Vcover.py
Enter the number of nodes: 4
Enter the number of edges: 4
Enter edges (node1 node2):
0 1
1 2
2 3
3 0
...
Graph:
Adjacent nodes of node 0: [1, 3]
Adjacent nodes of node 1: [0, 2]
Adjacent nodes of node 2: [1, 3]
Adjacent nodes of node 3: [2, 0]
Vertex Cover:
0 1 2 3
(base) E:\Desktop\Assignments and classes\DAA LAB\ME_DAA_Lab\Program>
```

RESULT:

Thus the implementation of the vertex cover problem using python was executed and verified successfully.