# Ex No:01 RSA

# Date:

## AIM:

To execute RSA Algorithm Using python and encrypt and decrypt.

## ALGORITHM:

1. Start.
2. **Key Generation:**
   1. Choose two distinct prime numbers, p and q.
   2. Compute n = p\*q.
   3. Compute Φ (n)=(p-1)\*(q-1).where Φ is Eucler’s totient function.
   4. Choose an integer e such that 1 < e < Φ (n) and e is coprime to Φ (n).
   5. Compute the integer d such that d\*e ≡1(modΦ (n)), i.e., d is the modular multiplicative inverse of e modulo Φ (n).
   6. The public key is (n,e) and the private key is (n,d).
3. Encryption:
   1. Encrypt the message M using public key(n,e).
      1. Compute C ≡ Me (mad n).
4. Decryption:
   1. Decrypt the ciphertext C using the private key (n, d).
      1. Compute M ≡ cd (mod 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)

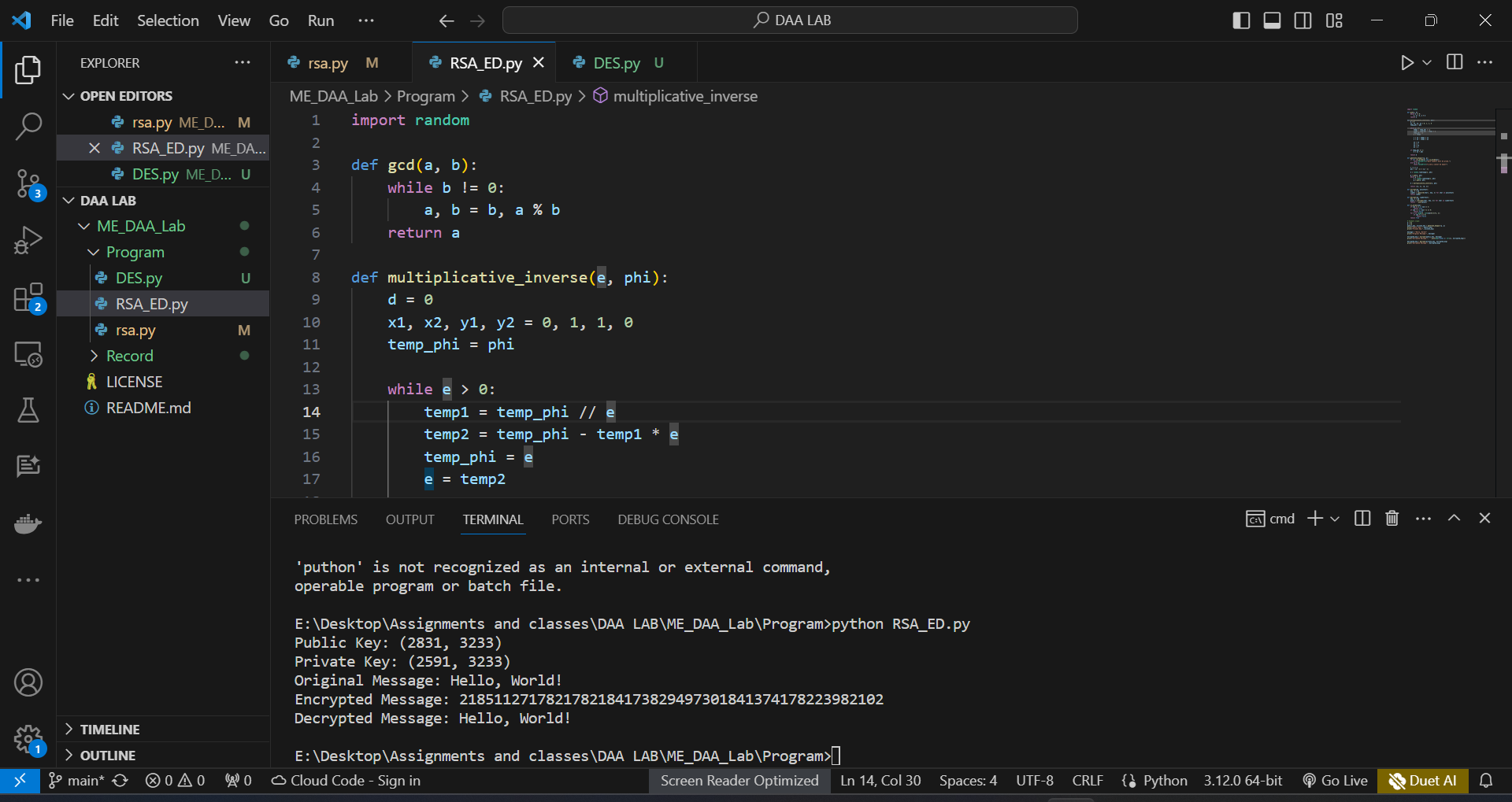
encrypted\_msg = encrypt(public\_key, message)

print("Encrypted Message:", ''.join(map(lambda x: str(x), encrypted\_msg)))

decrypted\_msg = decrypt(private\_key, encrypted\_msg)

print("Decrypted Message:", decrypted\_msg)

## OUTPUT:



## RESULT:

This RSA algorithm was executed successfully.

# Ex No:02 DES

# 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 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."""

# Inititalizing 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)]

# Apending 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))

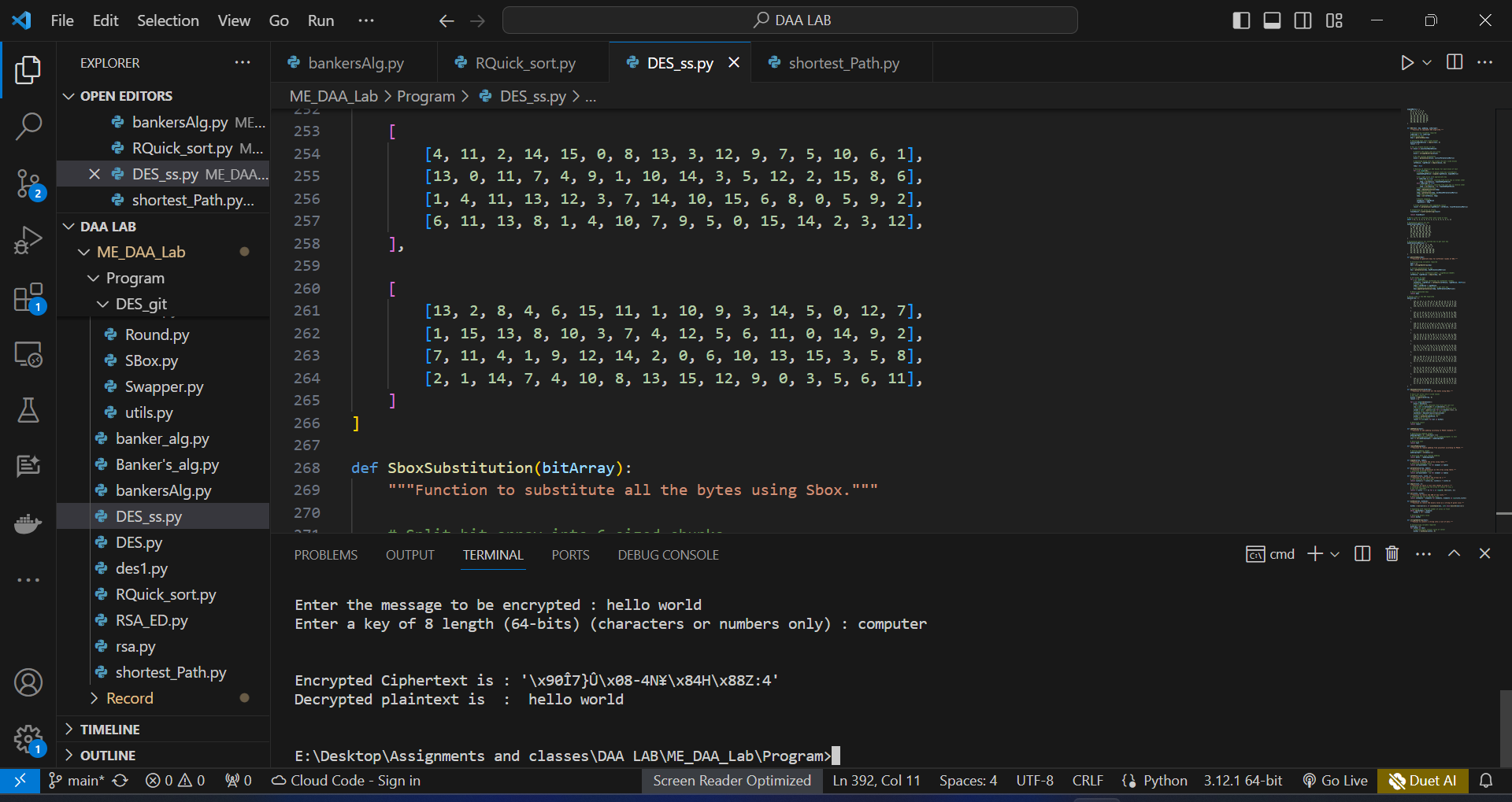
result = ''.join([chr(int(stringByte, 2)) for stringByte in stringBytesList])

return result

if \_\_name\_\_ == '\_\_main\_\_':

main()

## OUTPUT:



## RESULT:

Thus implementation of DES algorithm was completed successfully.

# Ex No:03 Banker’s Algorithm

# 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<=need, go to step 4, else it shows an error.
4. If request<=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="")

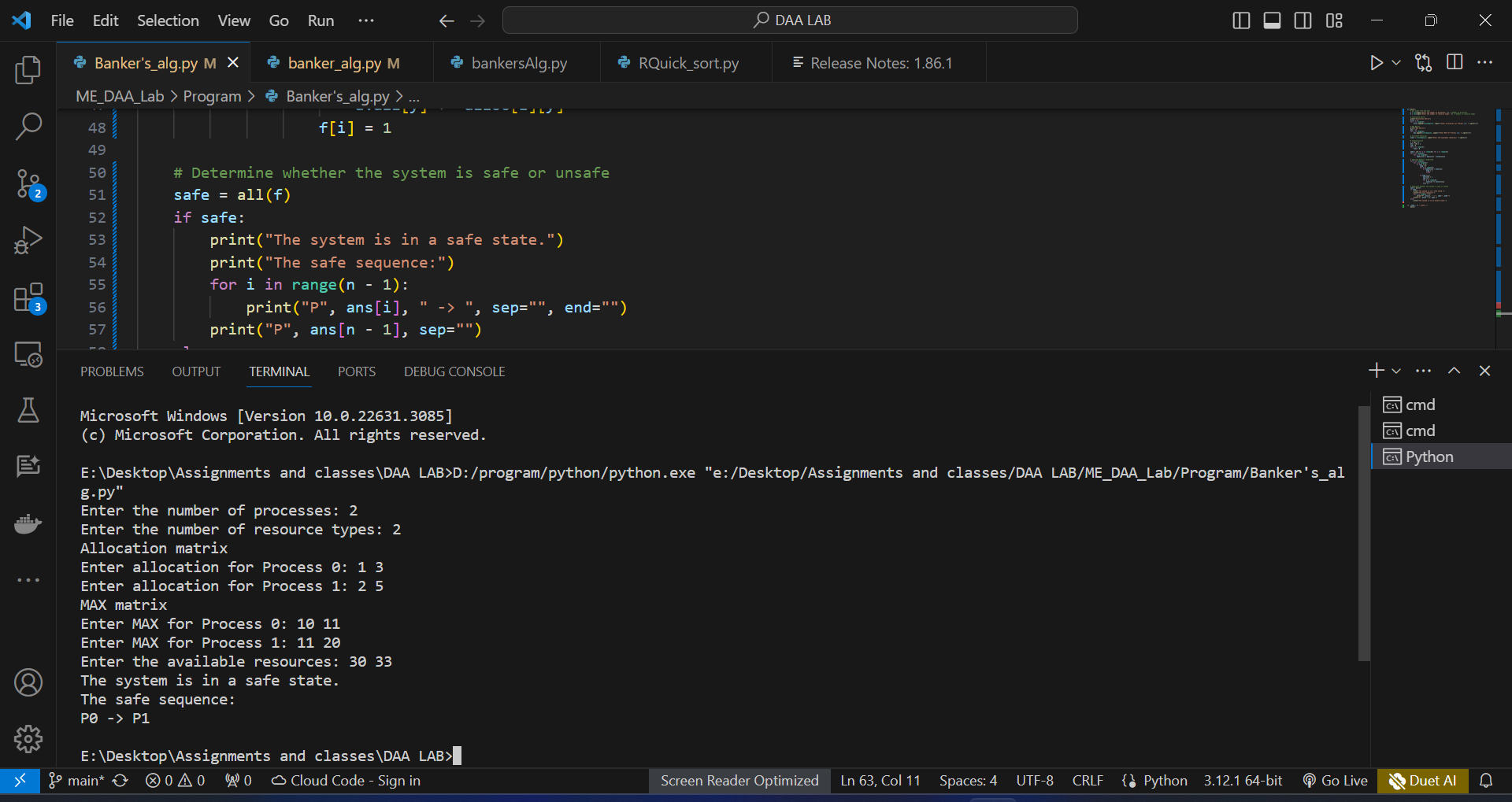
else:

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

if \_\_name\_\_ == "\_\_main\_\_":

main()

## OUTPUT:



## RESULT:

Thus implementation of bancker’s algorithm was completes successfully.

# Ex No:04 Randomized Quick Sort

# 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.

1. Repeat step 5 until it reaches the end of the array, then swap the pivot element with the second pointer.
2. Pivot elements are again chosen for the left and right sub arrays separately.
3. Repeat steps from 3 to 6.
4. Print the sorted array.
5. 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)

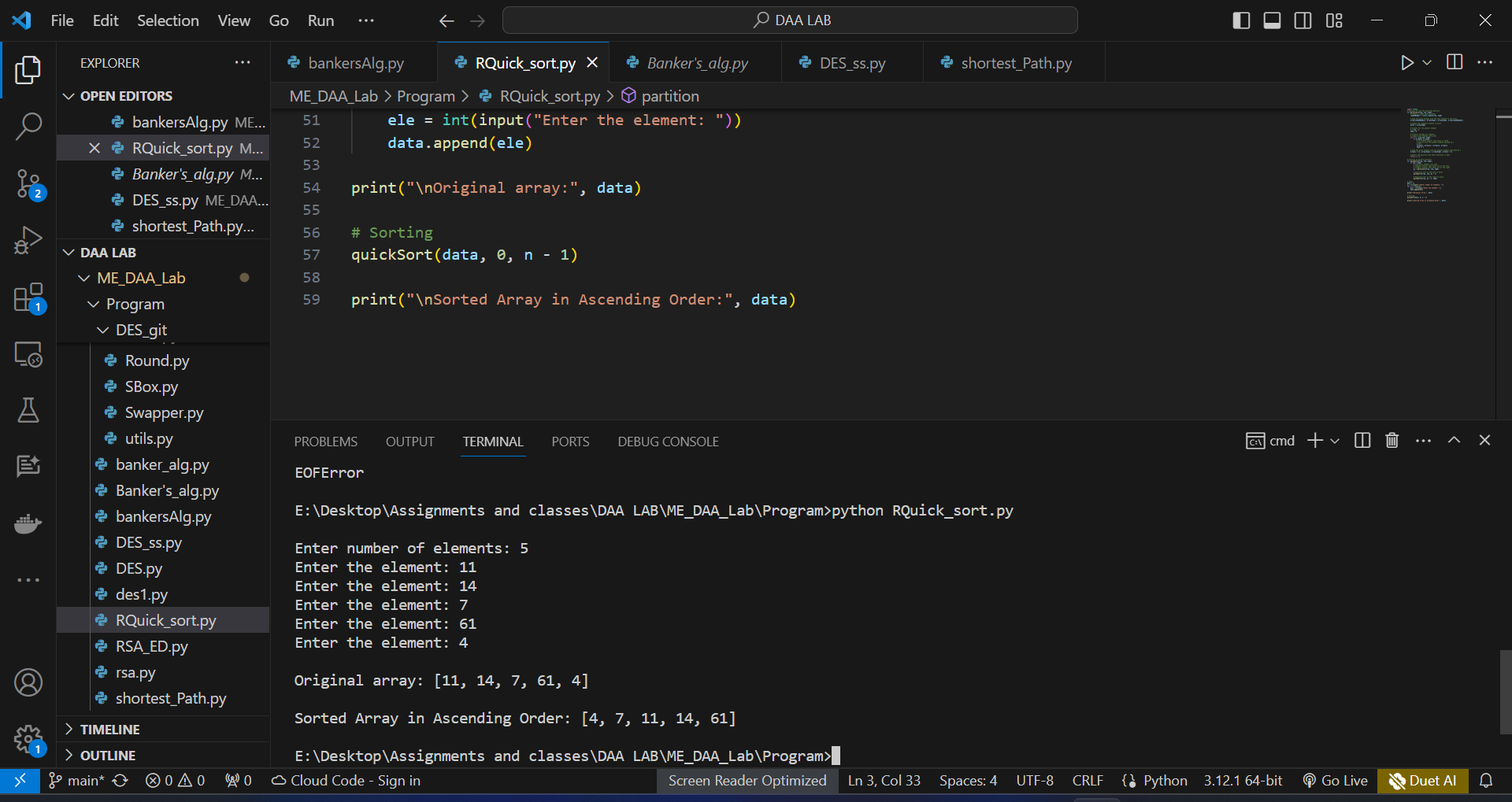
print("\nOriginal array:", data)

# Sorting

quickSort(data, 0, n - 1)

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

## OUTPUT:



## RESULT:

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