Program:

```
 #Ceaser
def encrypt(string, shift):
      cipher = ''
      for char in string:
            if char == ' ':
                  cipher = cipher + char
            elif char.isupper():
                  cipher = cipher + chr((ord(char) + shift - 65) % 26 + 65)
            else:
                  cipher = cipher + chr((ord(char) + shift - 97) % 26 + 97)
      return cipher
text = input("enter string: ")
s=int(input("Enter Shift Key: "))
print("original string: ", text)
print("after encryption: ", encrypt(text, s))
```

Output:

```
enter string: abcd
Enter Shift Key: 5
original string:  abcd
after encryption:  fghi
```

Program:

#Playfair

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

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

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

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']

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)
            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 = input("Enter Plain Text...: ")
text_Plain = removeSpaces(toLowerCase(text_Plain))
PlainTextList = Diagraph(FillerLetter(text_Plain))
if len(PlainTextList[-1]) != 2:
      PlainTextList[-1] = PlainTextList[-1]+'z'
key = input("Enter Key...: ")
key = toLowerCase(key)
Matrix = generateKeyTable(key, list1)
print("Plain Text:", text_Plain)
CipherList = encryptByPlayfairCipher(Matrix, PlainTextList)
CipherText = ""
for i in CipherList:
      CipherText += i
print("CipherText:", CipherText)
```

Output:

```
Enter Plain Text...: Periyar
Enter Key...: man
Plain Text: periyar
CipherText: khqkwbuw
```

Program:

#Hill Cipher.

```
keyMatrix = [[0] * 3 for i in range(3)]
messageVector = [[0] for i in range(3)]
cipherMatrix = [[0] for i in range(3)]
def getKeyMatrix(key):
    k = 0
    for i in range(3):
        for j in range(3):
            keyMatrix[i][j] = ord(key[k]) % 65
            k += 1
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):
    getKeyMatrix(key)
    for i in range(3):
        messageVector[i][0] = ord(message[i]) % 65
    encrypt(messageVector)
    CipherText = []
    for i in range(3):
        CipherText.append(chr(cipherMatrix[i][0] + 65))
    print("Ciphertext: ", "".join(CipherText))
def main():
    message=input("Enter Message in 3 character...: ")
    key = "ABCDEFGHI"
    HillCipher(message, key)
if __name__ == "__main__":
    main()
```

Output:

```
Enter Message in 3 character...: ABC
Ciphertext:  FOX
```

Program:

```python
#Vigenere Cipher
def generateKey(string, key):
	key = list(key)
	if len(string) == len(key):
		return(key)
	else:
		for i in range(len(string) -len(key)):
			key.append(key[i % len(key)])
	return("" . join(key))
def encryption(string, key):
	encrypt_text = []
	for i in range(len(string)):
		x = (ord(string[i]) +ord(key[i])) % 26
		x += ord('A')
		encrypt_text.append(chr(x))
	return("" . join(encrypt_text))
def decryption(encrypt_text, key):
	orig_text = []
	for i in range(len(encrypt_text)):
		x = (ord(encrypt_text[i]) -ord(key[i]) + 26) % 26
		x += ord('A')
		orig_text.append(chr(x))
	return("" . join(orig_text))
if __name__ == "__main__":
	string = input("Enter the message: ")
	keyword = input("Enter the keyword: ")
	key = generateKey(string, keyword)
	encrypt_text = encryption(string,key)
	print("Encrypted message:", encrypt_text)
	print("Decrypted message:", decryption(encrypt_text, key))
```

Output:

```
Enter the message: HELLO
Enter the keyword: ABC
Encrypted message: HFNLP
Decrypted message: HELLO
```

Program:

```
#railFence
def sequence(n):
	arr=[]
	i=0
	while(i<n-1):
		arr.append(i)
		i+=1
	while(i>0):
		arr.append(i)
		i-=1
	return(arr)

def railfence(s,n):
	s=s.lower()
	L=sequence(n)
	print("The raw sequence of indices: ",L)
	temp=L
	while(len(s)>len(L)):
		L=L+temp
	for i in range(len(L)-len(s)):
		L.pop()
	print("The row indices of the characters in the given string: ",L)
	print("Transformed message for encryption: ",s)
	num=0
	cipher_text=""
	while(num<n):
		for i in range(L.count(num)):
			cipher_text=cipher_text+s[L.index(num)]
			L[L.index(num)]=n
		num+=1
	print("The cipher text is: ",cipher_text)
plain_text=input("Enter the string to be encrypted: ")
n=int(input("Enter the number of rails: "))
railfence(plain_text,n)
def sequence(n):
	arr=[]
	i=0
	while(i<n-1):
		arr.append(i)
		i+=1
	while(i>0):
	arr.append(i)
	i-=1
return(arr)

def railfence(cipher_text,n):
	cipher_text=cipher_text.lower()
```

```python
    L=sequence(n)
    print("The raw sequence of indices: ",L)
    temp=L
    while(len(cipher_text)>len(L)):
        L=L+temp
    for i in range(len(L)-len(cipher_text)):
        L.pop()
    temp1=sorted(L)
    print("The row indices of the characters in the cipher string: ",L)
    print("The row indices of the characters in the plain string: ",temp1)
    print("Transformed message for decryption: ",cipher_text)
    plain_text=""
    for i in L:
        k=temp1.index(i)
        temp1[k]=n
        plain_text+=cipher_text[k]
    print("The cipher text is: ",plain_text)

cipher_text=input("Enter the string to be decrypted: ")
n=int(input("Enter the number of rails: "))
railfence(cipher_text,n)
```

Output:

```
Enter the string to be encrypted: i hate windows
Enter the number of rails: 5
The raw sequence of indices:  [0, 1, 2, 3, 4, 3, 2, 1]
The row indices of the characters in the given string:  [0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3]
Transformed message for encryption:  i hate windows
The cipher text is:  ii wnh daeostw
Enter the string to be decrypted: ii wnh daeostw
Enter the number of rails: 5
The raw sequence of indices:  [0, 1, 2, 3, 4, 3, 2, 1]
The row indices of the characters in the cipher string:  [0, 1, 2, 3, 4, 3, 2, 1, 0, 1, 2, 3, 4, 3]
The row indices of the characters in the plain string:  [0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4]
Transformed message for decryption:  ii wnh daeostw
The cipher text is:  i hate windows
```

Program:

```
#row & Column
import math

def row(s,key):
    temp=[]
    for i in key:
        if i not in temp:
            temp.append(i)
    k=""
    for i in temp:
        k+=i
    print("The key used for encryption is: ",k)
    b=math.ceil(len(s)/len(k))
    if(b<len(k)):
        b=b+(len(k)-b)

    arr=[['_' for i in range(len(k))]
         for j in range(b)]
    i=0
    j=0
    for h in range(len(s)):
        arr[i][j]=s[h]
        j+=1
        if(j>len(k)-1):
            j=0
            i+=1
    print("The message matrix is: ")
    for i in arr:
        print(i)
    cipher_text=""
    kk=sorted(k)
    for i in kk:
        h=k.index(i)
        for j in range(len(arr)):
            cipher_text+=arr[j][h]
    print("The cipher text is: ",cipher_text)

msg=input("Enter the message: ")
key=input("Enter the key in alphabets: ")
row(msg,key)



###Decryption

import math

def row(s,key):
    temp=[]
```

```
    for i in key:
        if i not in temp:
            temp.append(i)
    k=""
    for i in temp:
        k+=i
    print("The key used for encryption is: ",k)

    arr=[['' for i in range(len(k))]
         for j in range(int(len(s)/len(k)))]
    kk=sorted(k)
    d=0
    for i in kk:
        h=k.index(i)
        for j in range(len(k)):
            arr[j][h]=s[d]
            d+=1
    print("The message matrix is: ")
    for i in arr:
        print(i)
    plain_text=""
    for i in arr:
        for j in i:
            plain_text+=j
    print("The plain text is: ",plain_text)


msg=input("Enter the message to be decrypted: ")
key=input("Enter the key in alphabets: ")
row(msg,key)
```

Output:

```
Enter the message: ihatewindows
Enter the key in alphabets: love
The key used for encryption is:  love
The message matrix is:
['i', 'h', 'a', 't']
['e', 'w', 'i', 'n']
['d', 'o', 'w', 's']
['_', '_', '_', '_']
The cipher text is:  tns_ied_hwo_aiw_
Enter the message to be decrypted: tns_ied_hwo_aiw_
Enter the key in alphabets: love
The key used for encryption is:  love
The message matrix is:
['i', 'h', 'a', 't']
['e', 'w', 'i', 'n']
['d', 'o', 'w', 's']
['_', '_', '_', '_']
The plain text is:  ihatewindows____
```

Program:

```
# DES
# Hexadecimal to binary conversion


def hex2bin(s):
      mp = {'0': "0000",
            '1': "0001",
            '2': "0010",
            '3': "0011",
            '4': "0100",
            '5': "0101",
            '6': "0110",
            '7': "0111",
            '8': "1000",
            '9': "1001",
            'A': "1010",
            'B': "1011",
            'C': "1100",
            'D': "1101",
            'E': "1110",
            'F': "1111"}
      bin = ""
      for i in range(len(s)):
            bin = bin + mp[s[i]]
      return bin

# Binary to hexadecimal conversion


def bin2hex(s):
      mp = {"0000": '0',
            "0001": '1',
            "0010": '2',
            "0011": '3',
            "0100": '4',
            "0101": '5',
            "0110": '6',
            "0111": '7',
            "1000": '8',
            "1001": '9',
            "1010": 'A',
            "1011": 'B',
            "1100": 'C',
            "1101": 'D',
            "1110": 'E',
            "1111": 'F'}
      hex = ""
      for i in range(0, len(s), 4):
            ch = ""
```

```
            ch = ch + s[i]
            ch = ch + s[i + 1]
            ch = ch + s[i + 2]
            ch = ch + s[i + 3]
            hex = hex + mp[ch]

      return hex

# Binary to decimal conversion


def bin2dec(binary):

      binary1 = binary
      decimal, i, n = 0, 0, 0
      while(binary != 0):
            dec = binary % 10
            decimal = decimal + dec * pow(2, i)
            binary = binary//10
            i += 1
      return decimal

# Decimal to binary conversion


def dec2bin(num):
      res = bin(num).replace("0b", "")
      if(len(res) % 4 != 0):
            div = len(res) / 4
            div = int(div)
            counter = (4 * (div + 1)) - len(res)
            for i in range(0, counter):
                  res = '0' + res
      return res

# Permute function to rearrange the bits


def permute(k, arr, n):
      permutation = ""
      for i in range(0, n):
            permutation = permutation + k[arr[i] - 1]
      return permutation

# shifting the bits towards left by nth shifts


def shift_left(k, nth_shifts):
      s = ""
      for i in range(nth_shifts):
```

```
            for j in range(1, len(k)):
                s = s + k[j]
            s = s + k[0]
            k = s
            s = ""
        return k

# calculating xow of two strings of binary number a and b


def xor(a, b):
        ans = ""
        for i in range(len(a)):
            if a[i] == b[i]:
                ans = ans + "0"
            else:
                ans = ans + "1"
        return ans


# Table of Position of 64 bits at initial level: Initial Permutation Table
initial_perm = [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]

# Expansion D-box Table
exp_d = [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]

# Straight Permutation Table
per = [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]

# S-box Table
sbox = [[[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]]]

# Final Permutation Table
final_perm = [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 encrypt(pt, rkb, rk):
      pt = hex2bin(pt)

      # Initial Permutation
      pt = permute(pt, initial_perm, 64)
      print("After initial permutation", bin2hex(pt))

      # Splitting
      left = pt[0:32]
      right = pt[32:64]
      for i in range(0, 16):
            # Expansion D-box: Expanding the 32 bits data into 48 bits
            right_expanded = permute(right, exp_d, 48)

            # XOR RoundKey[i] and right_expanded
            xor_x = xor(right_expanded, rkb[i])

            # S-boxex: substituting the value from s-box table by calculating row
and column
            sbox_str = ""
            for j in range(0, 8):
                  row = bin2dec(int(xor_x[j * 6] + xor_x[j * 6 + 5]))
                  col = bin2dec(
                        int(xor_x[j * 6 + 1] + xor_x[j * 6 + 2] + xor_x[j * 6 + 3]
+ xor_x[j * 6 + 4]))
                  val = sbox[j][row][col]
                  sbox_str = sbox_str + dec2bin(val)

            # Straight D-box: After substituting rearranging the bits
            sbox_str = permute(sbox_str, per, 32)

            # XOR left and sbox_str
            result = xor(left, sbox_str)
            left = result

            # Swapper
            if(i != 15):
                  left, right = right, left
            print("Round ", i + 1, " ", bin2hex(left),
                  " ", bin2hex(right), " ", rk[i])

      # Combination
      combine = left + right

      # Final permutation: final rearranging of bits to get cipher text
      cipher_text = permute(combine, final_perm, 64)
      return cipher_text

# Padding function
def pad(msg):
```

```
  if(len(msg)%16!=0):
    print("Padding required")
    for i in range(abs(16-(len(msg)%16))):
      msg=msg+'0'
  else:
    print("No padding required")
  return(msg)

pt=input("Enter Plain Text ...: ")
pt=pad(pt)
print("Plain Text After Padding... : ",pt)
#pt = "123456ABCD132536"
key=input("Enter Key ...: ")
key=pad(key)
print("Key after Padding... : ",key)
#key = "AABB09182736CCDD"

# Key generation
# --hex to binary
key = hex2bin(key)

# --parity bit drop table
keyp = [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]

# getting 56 bit key from 64 bit using the parity bits
key = permute(key, keyp, 56)

# Number of bit shifts
shift_table = [1, 1, 2, 2,
                 2, 2, 2, 2,
                 1, 2, 2, 2,
                 2, 2, 2, 1]

# Key- Compression Table : Compression of key from 56 bits to 48 bits
key_comp = [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]
```

```
# Splitting
left = key[0:28] # rkb for RoundKeys in binary
right = key[28:56] # rk for RoundKeys in hexadecimal

rkb = []
rk = []
for i in range(0, 16):
      # Shifting the bits by nth shifts by checking from shift table
      left = shift_left(left, shift_table[i])
      right = shift_left(right, shift_table[i])

      # Combination of left and right string
      combine_str = left + right

      # Compression of key from 56 to 48 bits
      round_key = permute(combine_str, key_comp, 48)

      rkb.append(round_key)
      rk.append(bin2hex(round_key))

print("Encryption")
cipher_text = bin2hex(encrypt(pt, rkb, rk))
print("Cipher Text : ", cipher_text)

print("Decryption")
rkb_rev = rkb[::-1]
rk_rev = rk[::-1]
text = bin2hex(encrypt(cipher_text, rkb_rev, rk_rev))
print("Plain Text : ", text)
```

Output:

```
Enter Plain Text ...: ABCD
Padding required
Plain Text After Padding... :  ABCD000000000000
Enter Key ...: 1234
Padding required
Key after Padding... :  1234000000000000
Encryption
After initial permutation 0200020303010301
Round  1   03010301   FD29BBDB   000000040010
Round  2   FD29BBDB   86E8F28B   0020008000C0
Round  3   86E8F28B   E5860D75   000400408201
Round  4   E5860D75   52E8DD47   400000120408
Round  5   52E8DD47   C39792E1   008000081100
Round  6   C39792E1   B7D8A315   000002006020
Round  7   B7D8A315   3CB4B628   200000600800
Round  8   3CB4B628   899F2F78   00000080001A
Round  9   899F2F78   2036A888   000040810500
Round  10   2036A888   0034BAB6   004000080200
Round  11   0034BAB6   5BCE4658   000100504004
Round  12   5BCE4658   2C4AA14A   000001000088
Round  13   2C4AA14A   0CA8C46E   010000803001
Round  14   0CA8C46E   6F6BBC7F   000080220220
Round  15   6F6BBC7F   6DB47D8E   100000100902
Round  16   6BE66499   6DB47D8E   000800040104
Cipher Text :  C952BECB29FCDC33
Decryption
After initial permutation 6BE664996DB47D8E
Round  1   6DB47D8E   6F6BBC7F   000800040104
Round  2   6F6BBC7F   0CA8C46E   100000100902
Round  3   0CA8C46E   2C4AA14A   000080220220
Round  4   2C4AA14A   5BCE4658   010000803001
Round  5   5BCE4658   0034BAB6   000001000088
Round  6   0034BAB6   2036A888   000100504004
Round  7   2036A888   899F2F78   004000080200
Round  8   899F2F78   3CB4B628   000040810500
Round  9   3CB4B628   B7D8A315   00000080001A
Round  10   B7D8A315   C39792E1   200000600800
Round  11   C39792E1   52E8DD47   000002006020
Round  12   52E8DD47   E5860D75   008000081100
Round  13   E5860D75   86E8F28B   400000120408
Round  14   86E8F28B   FD29BBDB   000400408201
Round  15   FD29BBDB   03010301   0020008000C0
Round  16   02000203   03010301   000000040010
Plain Text :  ABCD000000000000
```

Program :

```
#!pip install pycrypto
#AES
import hashlib
from Crypto import Random
from Crypto.Cipher import AES
from base64 import b64encode, b64decode

class AESCipher(object):
    def __init__(self, key):
        self.block_size = AES.block_size
        self.key = hashlib.sha256(key.encode()).digest()

    def encrypt(self, plain_text):
        plain_text = self.__pad(plain_text)
        iv = Random.new().read(self.block_size)
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        encrypted_text = cipher.encrypt(plain_text.encode())
        return b64encode(iv + encrypted_text).decode("utf-8")

    def decrypt(self, encrypted_text):
        encrypted_text = b64decode(encrypted_text)
        iv = encrypted_text[:self.block_size]
        cipher = AES.new(self.key, AES.MODE_CBC, iv)
        plain_text = cipher.decrypt(encrypted_text[self.block_size:]).decode("utf-
8")
        return self.__unpad(plain_text)

    def __pad(self, plain_text):
        number_of_bytes_to_pad = self.block_size - len(plain_text) %
self.block_size
        ascii_string = chr(number_of_bytes_to_pad)
        padding_str = number_of_bytes_to_pad * ascii_string
        padded_plain_text = plain_text + padding_str
        print("The plain text after padding: ",padded_plain_text)
        return padded_plain_text

    @staticmethod
    def __unpad(plain_text):
        last_character = plain_text[len(plain_text) - 1:]
        return plain_text[:-ord(last_character)]

key=input("Enter the key: ")
c=AESCipher(key)
plain_text=input("Enter the message: ")
print("The message is: ", plain_text)

cipher=c.encrypt(plain_text)
print("Encrypted message is: ",cipher)
```

```
dec=c.decrypt(cipher)
print("Decrypted message is: ",dec)
```

Output :

```
Enter the key: Encrypt Me
Enter the message: Iam Secret Message
The message is:  Iam Secret Message
The plain text after padding:  Iam Secret Message
Encrypted message is:  UBzJRaz2yJZOBJlHTf6tl8evFXpVndEUnS50g8cY4vA5IldHZVl+hpNulIGl+n0z
Decrypted message is:  Iam Secret Message
```

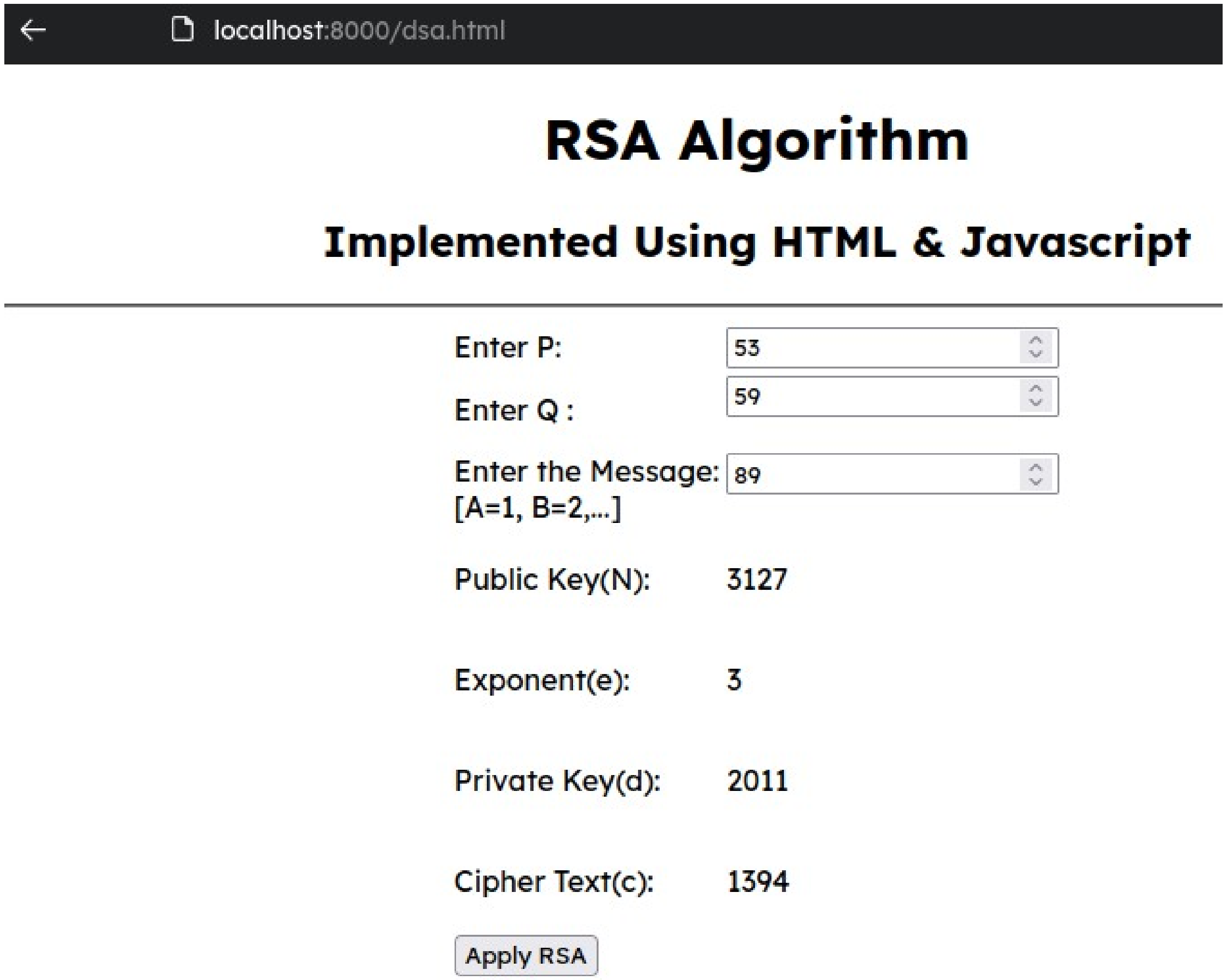
Program:

```
#RSA Algorithm using HTML and JavaScript
<!DOCTYPE html>
<html>
<head>
<title>RSA Encryption</title>
<meta name="viewport" content="width=device-width, initialscale=1.0">
</head>
<body>
<h1 style="text-align: center;">RSA Algorithm</h1>
<h2 style="text-align: center;">Implemented Using HTML & Javascript</h2>
<hr>
<table class="center">
<tr>
<td>Enter P:< /td>
<td><input type="number" value="53" id="p"></td>
</tr>
<tr>
<td>Enter Q :< /td>
<td><input type="number" value="59" id="q"></p>
</td>
</tr>
<tr>
<td>Enter the Message:< br>[A=1, B=2,...]</td>
<td><input type="number" value="89" id="msg"></p>
</td>
</tr>
<tr>
<td>Public Key(N):< /td>
<td>
<p id="publickey(N)"></p>
</td>
</tr>
<tr>
<td>Exponent(e):< /td>
<td>
<p id="exponent(e)"></p>
</td>
</tr>
<tr>
<td>Private Key(d):< /td>
<td>
<p id="privatekey(d)"></p>
</td>
</tr>
<tr>
<td>Cipher Text(c):< /td>
<td>
<p id="ciphertext(ct)"></p>
</td>
```

```
</tr>
<tr>
    <td><button onclick="RSA();">Apply RSA</button></td>
    </tr>
    </table>

    </body>
    <style>
        .center {
  margin-left: auto;
  margin-right: auto;
}
    </style>
    <script type="text/javascript">
    function RSA() {
    var gcd, p, q, no, n, t, e, i, x;
    gcd = function (a, b) { return (!b) ? a : gcd(b, a % b); };
    p = document.getElementById('p').value;
    q = document.getElementById('q').value;
    no = document.getElementById('msg').value;
    n = p * q;
    t = (p - 1) * (q - 1);
    for (e = 2; e < t; e++) {
    if (gcd(e, t) == 1) {
    break;
    }
    }
    for (i = 0; i < 10; i++) {
    x = 1 + i * t
    if (x % e == 0) {
    d = x / e;
    break;
    }
}
ctt = Math.pow(no, e).toFixed(0);
ct = ctt % n;
dtt = Math.pow(ct, d).toFixed(0);
dt = dtt % n;
document.getElementById('publickey(N)').innerHTML = n;
document.getElementById('exponent(e)').innerHTML = e;
document.getElementById('privatekey(d)').innerHTML = d;
document.getElementById('ciphertext(ct)').innerHTML = ct;
}
</script>
</html>
```

Output:


localhost:8000/dsa.html
RSA Algorithm
Implemented Using HTML & Javascript
Enter P:
53
Enter Q :
59
Enter the Message: [A=1, B=2,...]
89
Public Key(N):
3127
Exponent(e):
3
Private Key(d):
2011
Cipher Text(c):
1394
Apply RSA

Program:

```
#Diffie-Hellman Key Exchange
from random import randint
P = int(input("Enter a Prime Number..: "))
G = int(input("Enter a Primitive root..: "))
a = int(input("The Private Key a for Alice is.. : "))
x = int(pow(G,a,P))
a = int(input("The Private Key b for Bob is..: "))
y = int(pow(G,b,P))
ka = int(pow(y,a,P))
kb = int(pow(x,b,P))
print('Secret key for the Alice is : %d'%(ka))
print('Secret Key for the Bob is : %d'%(kb))
```

Output:

```
Enter a Prime Number..: 23
Enter a Primitive root..: 9
The Private Key a for Alice is.. : 4
The Private Key b for Bob is..: 3
Secret key for the Alice is : 2
Secret Key for the Bob is : 9
```

Program:

```
#SHA1
import hashlib
s=input("Enter the message to encrypt: ")
result=hashlib.sha1(s.encode())
print("The SHA1 for",'`',s,'`',"is..: ",result.hexdigest())
```

Output:

```
Enter the message to encrypt: I Love Linux
The SHA1 for ` I Love Linux ` is..:  5f0e9bfc2bc52a2ad8f50170ffe998b89ce9e937
```

Program:

```
#DSS
from Crypto.Util.number import *
from random import *
from hashlib import sha1
def hash_function(message):
    hashed=sha1(message.encode("UTF-8")).hexdigest()
    return hashed
def mod_inverse(a, m) :
    a=a%m;
    for x in range(1,m) :
        if((a*x)%m==1) :
            return(x)
    return(1)
def parameter_generation():
    q=getPrime(5)
    p=getPrime(10)
    while((p-1)%q!=0):
        p=getPrime(10)
        q=getPrime(5)
    print("Prime divisor (q): ",q)
    print("Prime modulus (p): ",p)
    flag=True
    while(flag):
        h=int(input("Enter integer between 1 and p-1(h): "))
        # h must be in between 1 and p-1
        if(1<h<(p-1)):
            g=1
            while(g==1):
                g=pow(h,int((p-1)/q))%p
            flag=False
        else:
            print("Wrong entry")
    print("Value of g is : ",g)
    return(p,q,g)
def per_user_key(p,q,g):
    x=randint(1,q-1)
    print("Randomly chosen x(Private key) is: ",x)
    y=pow(g,x)%p
    print("Randomly chosen y(Public key) is: ",y)
    return(x,y)
def signature(name,p,q,g,x):
    with open(name) as file:
        text=file.read()
        hash_component = hash_function(text)
        print("Hash of document sent is: ",hash_component)
    r=0
    s=0
    while(s==0 or r==0):
        k=randint(1,q-1)
```

```
        r=((pow(g,k))%p)%q
        i=mod_inverse(k,q)
        # converting hexa decimal to binary
        hashed=int(hash_component,16)
        s=(i*(hashed+(x*r)))%q
    return(r,s,k)
def verification(name,p,q,g,r,s,y):
    with open(name) as file:
        text=file.read()
        hash_component = hash_function(text)
        print("Hash of document received is: ",hash_component)
    w=mod_inverse(s,q)
    print("Value of w is : ",w)
    hashed=int(hash_component,16)
    u1=(hashed*w)%q
    u2=(r*w)%q
    v=((pow(g,u1)*pow(y,u2))%p)%q
    print("Value of u1 is: ",u1)
    print("Value of u2 is: ",u2)
    print("Value of v is : ",v)
    if(v==r):
        print("The signature is valid!")
    else:
        print("The signature is invalid!")
global_var=parameter_generation()
keys=per_user_key(global_var[0],global_var[1],global_var[2])
print()
file_name=input("Enter the name of document to sign: ")
components=signature(file_name,global_var[0],global_var[1],global_var[2],keys[0])
print("r(Component of signature) is: ",components[0])
print("k(Randomly chosen number) is: ",components[2])
print("s(Component of signature) is: ",components[1])
print()
file_name=input("Enter the name of document to verify: ")
verification(file_name,global_var[0],global_var[1],global_var[2],components[0],com
ponents[1],keys[1])
```

Output:

```
Prime divisor (q):  23
Prime modulus (p):  967
Enter integer between 1 and p-1(h): 949
Value of g is :  157
Randomly chosen x(Private key) is:  8
Randomly chosen y(Public key) is:  953

Enter the name of document to sign: document.txt
Hash of document sent is:  62c561457fa7b963c155dd3ecacd0a3c63a9ef96
r(Component of signature) is:  12
k(Randomly chosen number) is:  16
s(Component of signature) is:  19

Enter the name of document to verify: document.txt
Hash of document received is:  62c561457fa7b963c155dd3ecacd0a3c63a9ef96
Value of w is :  17
Value of u1 is:  17
Value of u2 is:  20
Value of v is :  12
The signature is valid!
```