Caesar Cipher

Prashanth.S 19MID0020

In [ ]:

```
encryption --> (element + key) mod 26
decryption --> (element - key) mod 26
```

In [2]:

```python
def a2d(text):
    return [ord(i) for i in text]
```

In [3]:

```python
def encrpyt(text, key):
    dtext = a2d(text)
    result = []

    for i in dtext:

        ## capital letters
        if (i >= 65 and i <= 90):
            result.append((((i - 65) + key) % 26) + 65)

        ## small letters
        elif(i >= 97 and i <= 122):
            result.append((((i - 97) + key) % 26) + 97)

        ## no small and capital letters
        else:
            result.append(i)

    final = list(map(chr, result))
    return ''.join(final)
```

In [4]:

```python
def decrypt(text, key):
    dtext = a2d(text)
    result = []

    for i in dtext:

        ## capital letters
        if (i >= 65 and i <= 90):
            result.append((((i - 65) - key) % 26) + 65)

        ## small letters
        elif(i >= 97 and i <= 122):
            result.append((((i - 97) - key) % 26) + 97)

        else:
            result.append(i)

    final = list(map(chr, result))
    return ''.join(final)
```

In [5]:

```python
if __name__ == '__main__':
    text = input("Before encryption Plain text: ")
    key = int(input("Key: "))

    cipher = encrpyt(text, key)
    print("Cipher text : {}".format(cipher))

    plain = decrypt(cipher, key)
    print("After decryption Plain text : {}".format(plain))
```

```
Before encryption Plain text: Prashanth@123
Key: 5
Cipher text : Uwfxmfsym@123
After decryption Plain text : Prashanth@123
```

In [ ]:

Play-Fair Cipher

Prashanth.S 19MID0020

In [1]:
```
import string
from collections import OrderedDict
import numpy as np
from ordered_set import OrderedSet
```

In [2]:
```
def key_text_rule(key):
    for j in range(len(key)):
        for i in range(len(key)):
            if ((i%2==0) and (i+1!=len(key))):
                if ((key[i]) == (key[i+1])):
                    near = i+1
                    key = key[:near] + 'x' + key[near:]
                    break

    if (len(key)%2!=0):
        key = key[:len(key)+1] + 'z'
        return key
    else:
        return key
```

In [3]:
```
def matrix_fill(key):
    key = "".join(OrderedDict.fromkeys(key))   ## remove the repeated characters in the string
    str1 = string.ascii_lowercase
    for i in key:
        if i in str1:
            str1 = str1.replace(i,'')
    str1 = str1.replace('j','')
    matrix_elements = key + str1

    list1 = []
    ind = 0
    for i in range(5):
        temp = []
        for j in range(5):
            temp.append(matrix_elements[ind])
            ind+=1
        list1.append(temp)

    return list1
```

In [4]:
```
key = key_text_rule('monarchy')
plain_text = key_text_rule('instruments')

print(key)
print(plain_text)
```

```
monarchy
instrumentsz
```

In [5]:
```
matrix = matrix_fill(key)
matrix
```

Out[5]:
```
[['m', 'o', 'n', 'a', 'r'],
 ['c', 'h', 'y', 'b', 'd'],
 ['e', 'f', 'g', 'i', 'k'],
 ['l', 'p', 'q', 's', 't'],
 ['u', 'v', 'w', 'x', 'z']]
```

Encryption

In [6]:
```
def same_row_encrypt(ind1,ind2,ind3,ind4,matrix): ## Same 1st index(i.e i)

    ## loop
    if (ind2==4 or ind4==4):
        if (ind2==4):
            ind2 = 0
            print(matrix[ind1][ind2])
            print(matrix[ind3][ind4+1])

        if (ind4==4):
            ind4 = 0
            print(matrix[ind1][ind2+1])
            print(matrix[ind3][ind4])

    ## not a loop
    else:
        print(matrix[ind1][ind2+1])
        print(matrix[ind3][ind4+1])
```

In [7]:
```
def same_col_encrypt(ind1,ind2,ind3,ind4,matrix): ## Same 2nd index(i.e j)

    ## loop
    if (ind1==4 or ind3==4):
        if (ind1==4):
            ind1 = 0
            print(matrix[ind1][ind2])
            print(matrix[ind3+1][ind4])

        if (ind3==4):
            ind3 = 0
            print(matrix[ind1+1][ind2])
            print(matrix[ind3][ind4])

    ## not a loop
    else:
        print(matrix[ind1+1][ind2])
        print(matrix[ind3+1][ind4])
```

In [8]:
```
def diff(ind1,ind2,ind3,ind4,matrix):  ## Not in same row and same column
    print(matrix[ind1][ind4])
    print(matrix[ind3][ind2])
```

In [9]:
```
def check(i_index, j_index, matrix):
    for ind in range(len(i_index)):
        if ((ind%2==0) and ind!=len(i_index)):

            if (i_index[ind]==i_index[ind+1]): ## same i-value
                same_row_encrypt(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)

            elif (j_index[ind]==j_index[ind+1]): ## same j-value
                same_col_encrypt(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)

            else:
                diff(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)
```

In [10]:
```
i_index = []
j_index = []

for k in range(len(plain_text)):
    if (k%2==0) and (k+1!=len(plain_text)):
        word_1 = plain_text[k]
        word_2 = plain_text[k+1]

        ## fiding the letters in the matrix
        for i in range(5):
            for j in range(5):
                if ((word_1==matrix[i][j])):
                    i_index.append(i)
                    j_index.append(j)

        for i in range(5):
            for j in range(5):
                if ((word_2==matrix[i][j])):
                    i_index.append(i)
                    j_index.append(j)

print("Cipher text")
check(i_index, j_index,matrix)
```

```
Cipher text
g
a
t
l
m
z
c
l
r
q
t
x
```

In [11]:
```
def index_fill(plain_text, matrix):

    i_index = []
    j_index = []

    for k in range(len(plain_text)):
        if (k%2==0) and (k+1!=len(plain_text)):
            word_1 = plain_text[k]
            word_2 = plain_text[k+1]

            ## fiding the letters in the matrix
            for i in range(5):
                for j in range(5):
                    if ((word_1==matrix[i][j])):
                        i_index.append(i)
                        j_index.append(j)

            for i in range(5):
                for j in range(5):
                    if ((word_2==matrix[i][j])):
                        i_index.append(i)
                        j_index.append(j)

    print("Cipher text")
    check(i_index, j_index,matrix)
```

In [12]:
```
diff(2,3,0,2,matrix)      # (i,n) --> (a,g)
same_row_encrypt(3,3,3,4,matrix) # (s,t) --> (t,l)
diff(0,4,4,0,matrix)      # (r,u) --> (m,z)
same_col_encrypt(0,0,2,0,matrix) # (m,e) --> (c,l)
diff(0,2,3,4,matrix)      # (n,t) --> (r,q)
same_col_encrypt(3,3,4,3,matrix) # (s,x) --> (x,a)
```

```
g
a
t
l
m
z
c
l
r
q
x
a
```

Decryption

In [13]:
```
key = key_text_rule('monarchy')
plain_text = key_text_rule('gatlmzclrqtx')

print(key)
print(plain_text)
```

```
monarchy
gatlmzclrqtx
```

In [14]:
```
matrix = matrix_fill(key)
matrix
```

Out[14]:
```
[['m', 'o', 'n', 'a', 'r'],
 ['c', 'h', 'y', 'b', 'd'],
 ['e', 'f', 'g', 'i', 'k'],
 ['l', 'p', 'q', 's', 't'],
 ['u', 'v', 'w', 'x', 'z']]
```

In [15]:
```
def same_row_decrypt(ind1,ind2,ind3,ind4,matrix): ## Same 1st index(i.e i)

    print(end='')

    ## loop
    if (ind2==0 or ind4==0):
        if (ind2==0):
            ind2 = 4
            print(matrix[ind1][ind2])
            print(matrix[ind3][ind4-1])

        if (ind4==0):
            ind4 = 4
            print(matrix[ind1][ind2-1])
            print(matrix[ind3][ind4])

    ## not a loop
    else:
        print(matrix[ind1][ind2-1])
        print(matrix[ind3][ind4-1])
```

In [16]:
```
def same_col_decrypt(ind1,ind2,ind3,ind4,matrix): ## Same 2nd index(i.e j)
    print(end='')

    ## loop
    if (ind1==0 or ind3==0):
        if (ind1==0):
            ind1 = 4
            print(matrix[ind1][ind2])
            print(matrix[ind3-1][ind4])

        if (ind3==0):
            ind3 = 4
            print(matrix[ind1-1][ind2])
            print(matrix[ind3][ind4])

    ## not a loop
    else:
        print(matrix[ind1-1][ind2])
        print(matrix[ind3-1][ind4])
```

In [17]:
```
def diff_decrypt(ind1,ind2,ind3,ind4,matrix):  ## Not in same row and same column
    print(end='')
    print(matrix[ind1][ind4])
    print(matrix[ind3][ind2])
```

In [18]:
```
def check(i_index, j_index, matrix):
    for ind in range(len(i_index)):
        if ((ind%2==0) and ind!=len(i_index)):

            if (i_index[ind]==i_index[ind+1]): ## same i-value
                same_row_decrypt(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)

            elif (j_index[ind]==j_index[ind+1]): ## same j-value
                same_col_decrypt(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)

            else:
                diff_decrypt(i_index[ind],j_index[ind],i_index[ind+1],j_index[ind+1],matrix)
```

In [19]:
```
i_index = []
j_index = []

for k in range(len(plain_text)):
    if (k%2==0) and (k+1!=len(plain_text)):
        word_1 = plain_text[k]
        word_2 = plain_text[k+1]

        ## fiding the letters in the matrix
        for i in range(5):
            for j in range(5):
                if ((word_1==matrix[i][j])):
                    i_index.append(i)
                    j_index.append(j)

        for i in range(5):
            for j in range(5):
                if ((word_2==matrix[i][j])):
                    i_index.append(i)
                    j_index.append(j)

print("Plain text")
check(i_index, j_index,matrix)
```

```
Plain text
i
n
s
t
r
u
m
e
n
t
s
z
```

In [ ]:

In [ ]:

Hill Cipher

`Prashanth.S 19MID0020`

In [1]:
```
import string
from collections import OrderedDict
import numpy as np
from ordered_set import OrderedSet
import pymatrix
```

Encryption

In [2]:
```
def key_text_rule(key):
    for j in range(len(key)):
        for i in range(len(key)):
            if ((i%2==0) and (i+1!=len(key))):
                if ((key[i]) == (key[i+1])):
                    near = i+1
                    key = key[:near] + 'x' + key[near:]
                    break

    if (len(key)%2!=0):
        key = key[:len(key)+1] + 'z'
        return key
    else:
        return key
```

In [3]:
```
def text_to_matrix(dict1, text, n):
    list1 = []
    for i in text:
        list1.append(dict1[i])

    matrix = np.array(list1).reshape(n,n)
    return matrix
```

In [4]:
```
def encryption(small_dict, key, n):
    key_matrix = text_to_matrix(small_dict, key, n)
    key_matrix = np.matrix(key_matrix)

    main_encrypt_list = []

    for i in range(len(plain_text)):
        plain_list1 = []
        if ((i%2==0) and (i<=len(plain_text))):

            plain_list1.append(small_dict[plain_text[i]])
            plain_list1.append(small_dict[plain_text[i+1]])

            main_encrypt_list.append((np.dot(key_matrix, np.array(plain_list1).reshape(n,))) % 26)

    cipher_text = []
    dict_keys=list(small_dict.keys())

    for i in main_encrypt_list:

        val1 = i[0,0]
        val2 = i[0,1]

        cipher_text.append(dict_keys[val1])
        cipher_text.append(dict_keys[val2])

    cipher_text = ''.join(map(str,cipher_text))
    return (cipher_text, key_matrix)
```

Decryption

In [5]:
```
def gcd(a, b):

    if(b == 0):
        return a
    else:
        return gcd(b, a % b)
```

In [6]:
```
def modulo_multiplicative_inverse(key_matrix_det):

    if (gcd(key_matrix_det,26)==1):

        if (key_matrix_det>27):
            key_matrix_det = key_matrix_det%26

        num = 1
        while((key_matrix_det * num) % 26 !=1):
            num+=1

        return num

    else:
        return 0 # GCD(det,26)!=1, then modulo multiplicative inverse --> Not found
```

In [7]:
```
def decryption(cipher_text, key_matrix):

    key_matrix_det = int(np.linalg.det(key_matrix))
    one_by_det = modulo_multiplicative_inverse(key_matrix_det)

    if one_by_det:
        adj = (pymatrix.matrix(key_matrix.tolist())).adjoint()
        ## converting into numpy and int array

        key_matrix_adj = []
        for i in range(n):
            key_matrix_adj.append(adj[i])

        key_matrix_adj = np.array(key_matrix_adj).astype(int)
        key_matrix_adj

        for i in key_matrix_adj:
            if (i[0] < 0) : i[0] += 26
            if (i[1] < 0) : i[1] += 26

        key_inverse = key_matrix_adj * one_by_det

        main_decrypt_list = []
        for i in range(len(cipher_text)):

            plain_list1 = []
            if ((i%2==0) and (i<=len(cipher_text))):

                plain_list1.append(small_dict[cipher_text[i]])
                plain_list1.append(small_dict[cipher_text[i+1]])

                main_decrypt_list.append(np.round(np.dot(key_inverse, np.array(plain_list1).reshape(n,))) % 26)

        main_decrypt_list = np.int_(main_decrypt_list)

        original_text = []

        for i in main_decrypt_list:
            dict_keys=list(small_dict.keys())
            original_text.append(dict_keys[i[0]])
            original_text.append(dict_keys[i[1]])

        original_text = ''.join(map(str,original_text))

        return original_text

    else:
        return "GCD!=1, No Modulo Multiplicative Inverse"
```

In [8]:
```
if __name__ == '__main__':

    small_dict = dict()
    for index, letter in enumerate(string.ascii_lowercase):
          small_dict[letter] = index + 0

    plain_text = key_text_rule('prashanth')
    print(plain_text)
    n = 4
    key = 'test'

    cipher_text, key_matrix = encryption(small_dict, key, n)
    plain_text_dec = decryption(cipher_text,key_matrix)
```

```
prashanthz
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
/var/folders/gq/nsqxf83n1813yysq218vvtxc0000gn/T/ipykernel_6229/3833983723.py in <module>
     10     key = 'test'
     11 
---> 12     cipher_text, key_matrix = encryption(small_dict, key, n)
     13     plain_text_dec = decryption(cipher_text,key_matrix)

/var/folders/gq/nsqxf83n1813yysq218vvtxc0000gn/T/ipykernel_6229/992614898.py in encryption(small_dict, key, n)
      1 def encryption(small_dict, key, n):
----> 2     key_matrix = text_to_matrix(small_dict, key, n)
      3     key_matrix = np.matrix(key_matrix)
      4 
      5     main_encrypt_list = []

/var/folders/gq/nsqxf83n1813yysq218vvtxc0000gn/T/ipykernel_6229/4074971094.py in text_to_matrix(dict1, text, n)
      4         list1.append(dict1[i])
      5 
----> 6     matrix = np.array(list1).reshape(n,n)
      7     return matrix

ValueError: cannot reshape array of size 4 into shape (4,4)
```

In [ ]:
```
cipher_text
```

In [ ]:
```
plain_text_dec
```

In [ ]:
```
```

DES Algorithm

Prashanth.S 19MID0020

In [1]:
```python
def display_6(list1):
    for i in range(len(list1)):
        if (i%6==0 and i!=0):
            print("   ",end='')
        print(list1[i],end='')
```

In [2]:
```python
def display_7(list1):
    for i in range(len(list1)):
        if (i%7==0 and i!=0):
            print("   ",end='')
        print(list1[i],end='')
```

In [3]:
```python
def display_8(list1):
    for i in range(len(list1)):
        if (i%9==0):
            print("   ",end='')
        else:
            print(list1[i],end='')
```

In [4]:
```python
def left_right_split(matrix, cnt):
    left_str  = matrix[:cnt]
    right_str = matrix[cnt:]

    return (left_str, right_str)
```

Plain Text - part

Initial Permutation --> For 64 bits plain text

In [5]:
```python
def initial_permutation(elements):

    ## input  --> 64bits
    ## output --> 64bits

    ## initial_perm_matrix --> 1 to 64 bits
    ## 64bit plain text    --> 0 to 63 bits

    str_permutation_matrix = [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]

    permuted_matrix = [0 for i in range(64)]
    for i in range(0, len(str_permutation_matrix)):
        index              = str_permutation_matrix[i] - 1   ## so subtracting 1
        permuted_matrix[i] = elements[index]

    return permuted_matrix
```

Expansion Permutation --> For 32 bits-bit Right 64bit-plain text

In [6]:
```python
def expansion_permutation(elements):

    ## input(right_str) --> 32 bits
    ## output           --> 48 bits

    ## initial_perm_matrix --> 1 to 64 bits
    ## 64bit key           --> 0 to 63 bits

    expansion_matrix = [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 ]

    expanded_matrix = [0 for i in range(48)]

    for i in range(0, len(expanded_matrix)):
        index              = expansion_matrix[i] - 1
        expanded_matrix[i] = elements[index]

    return expanded_matrix
```

Key - part

Permuted Choice-1 --> For 64 bits key

In [7]:
```python
def permuted_choice_1(elements):

    ## input -->  64 bits
    ## output --> 56 bits

    ## initial_perm_matrix --> 1 to 64 bits
    ## 64bit key           --> 0 to 63 bits

    key_permutation_matrix = [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 ]

    permuted_matrix = [0 for i in range(56)]

    for i in range(0, len(key_permutation_matrix)):
        index              = key_permutation_matrix[i] - 1
        permuted_matrix[i] = elements[index]

    return permuted_matrix
```

Permuted Choice-2 --> For 56 bits key

In [8]:
```python
def permuted_choice_2(elements):

    ## left_str  --> 28 bits
    ## right_str --> 28 bits

    ## input -->  56 bits
    ## output --> 48 bits

    key_permutation_matrix = [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 ]

    permuted_matrix = [0 for i in range(48)]

    for i in range(0, len(key_permutation_matrix)):
        index              = key_permutation_matrix[i] - 1
        permuted_matrix[i] = elements[index]

    return permuted_matrix
```

XOR operation

In [9]:
```python
def xor_operation(i,j):
    if i!=j: return 1
    else: return 0
```

In [10]:
```python
def xor(expansion_permutation_matrix, key_permuted_matrix_2):

    ## input -->  48 bits
    ## output --> 48 bits

    list1 = []
    for i,j in zip(expansion_permutation_matrix,key_permuted_matrix_2):
        ans = xor_operation(i,j)
        list1.append(ans)
    return list1
```

In [11]:
```python
def binaryToDecimal(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
```

S-Box

In [12]:
```python
def s_box(xor_output):

    ## input -->  48 bits
    ## output --> 32 bits

    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]]

         ]

    list1 = []

    start_index = 0
    end_index = 6

    box_index = 0

    for i in range(8): ## every 6 bits to each S-box (i.e 8 S-box's)
        temp_list = []
        s_bits = xor_output[start_index:end_index]

        row_binary = str(s_bits[0]) + str(s_bits[-1])  ## 1st and 6th bits (0th and 5th)
        row_number = binaryToDecimal(int(row_binary))  ## binary to decimal

        col_binary = s_bits[1:5]                       ## 2nd, 3rd, 4th and 5th bits
        col_binary = [str(int) for int in col_binary]  ## converting the int-list to string-list
        col_binary = ''.join(col_binary)               ## join the string-list
        col_number = binaryToDecimal(int(col_binary))  ## binary to decimal

        temp_sbox            = sbox[box_index]                          ## choosing the required box
        decimal__table_value = temp_sbox[row_number][col_number]        ## with row, column searching box
        temp_list.append("{0:b}".format(decimal__table_value).zfill(4)) ## binary to decimal

        list1 = list1 + temp_list

        start_index += 6
        end_index   += 6
        box_index   += 1

    return list1
```

Final Permutation

In [13]:
```python
def final_permutation(s_box_output):

    ## input -->  32 bits
    ## output --> 32 bits

    final_perm = [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]
```

AES Encryption

Prashanth.S 19MID0020

Importing the Necessary Libraries

In [1]:
```
import numpy as np
import pandas as pd
import ast
import itertools
import collections
import struct
import sys
from sympy import poly, var
from collections import Counter
```

Getting inputs

In [2]:
```
col_names  = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f']
row_names = ['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', 'a', 'b', 'c', 'd', 'e', 'f']

enc_key_sbox = pd.read_excel('AES tables.xlsx', index_col=0)

enc_key_sbox.columns = col_names ## replacing the column names
enc_key_sbox.index = row_names    ## replacing the row names

enc_key_sbox.head()
```

Out[2]:

	0	1	2	3	4	5	6	7	8	9	a	b	c	d	e	f
0	63	7c	77	7b	f2	6b	6f	c5	30	1	67	2b	fe	d7	ab	76
1	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3	4	c7	23	c3	18	96	5	9a	7	12	80	e2	eb	27	b2	75
4	9	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84

Operational Functions

In [3]:
```
def key_sbox(element):
    row_index = element[0]
    col_index  = element[1]
    ans = enc_key_sbox.loc[[row_index],[col_index]][col_index][0]
    return str(ans)
```

In [4]:
```
def binaryToDecimal(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

    hexadecimal =  hex(decimal)[-1]
    return hexadecimal
```

In [5]:
```
def bcd_hexadecimal(bcd_list):
    w4 = []

    for i in range(0,len(bcd_list),2):
        temp_str = ' '
        temp_str = binaryToDecimal(int(bcd_list[i]))
        temp_str = temp_str + binaryToDecimal(int(bcd_list[i+1]))
        w4.append(temp_str)

    return w4
```

In [6]:
```
def HexaDecimaltoBCD(str):
    list1 = []
    for i in range(len(str)):
        decimal = int(str[i], 16)                                    ## hexadecimal -> decimal
        binary_num = bin(decimal).replace("0b", "")  # decimal -> binary
        list1.append(binary_num)

    ## binary in-terms of 4 bits
    for i in range(len(list1)):
        element = list1[i]
        if len(element)<4:
            diff = 4 - len(element)
            for j in range(diff):
                element = "0" + element
            list1[i] = element

    return list1
```

In [7]:
```
def Hexaword_BCD(list1):
    bcd = [ HexaDecimaltoBCD(i) for i in list1 ]
    bcd = list(np.concatenate(bcd).flat)    ## 2d list to 1d list
    return bcd
```

In [8]:
```
def XOR(list1, list2):

    def compare(element1, element2):
        ans = []
        for i,j in zip(element1,element2):
            if (i!=j):ans.append(1)
            else:ans.append(0)
        return ans

    main_ans = []
    for i in range(len(list1)):
        main_ans.append(compare(list1[i], list2[i]))

    main_ans = [ "".join(list(map(str, i))) for i in main_ans ]
    return main_ans
```

In [9]:
```
def binary_to_polynomial(a):
    str1 = ""
    nobits = len(a)
    for x in range (0,nobits-2):
        if (a[x] == '1'):
            if (len(str1)==0):str1 +="x^"+str(nobits-x-1)
            else: str1 +=" + x^"+str(nobits-x-1)
    if (a[nobits-2] == '1'):
        if (len(str1)==0):str1 +="x"
        else:str1 +=" + x"

    if (a[nobits-1] == '1'):str1 +="+1"
    print(str1)
```

In [10]:
```
def binary_division_modulo_2(val1, val2):

    def xor(a, b):
        result = []
        for i in range(1, len(b)):
            if a[i] == b[i]:result.append('0')
            else:result.append('1')
        return ''.join(result)

    def showpoly(a):
        str1 = ""
        nobits = len(a)
        for x in range (0,nobits-2):
            if (a[x] == '1'):
                if (len(str1)==0):str1 +="x**"+str(nobits-x-1)
                else: str1 +="+x**"+str(nobits-x-1)
        if (a[nobits-2] == '1'):
            if (len(str1)==0):str1 +="x"
            else:str1 +="+x"

        if (a[nobits-1] == '1'):str1 +="+1"
        print(str1)

    def divide(dividend, divisor):
        pick = len(divisor)
        tmp = dividend[0 : pick]

        while (pick < len(dividend)):

            if tmp[0] == '1':tmp = xor(divisor, tmp) + dividend[pick]
            else: tmp = xor('0'*pick, tmp) + dividend[pick]
            pick += 1

        if tmp[0] == '1':tmp = xor(divisor, tmp)
        else:tmp = xor('0'*pick, tmp)

        checkword = tmp
        return checkword

    val1= divide(val1,val2)
    return val1
```

In [11]:
```
def bcd_to_hexadecimal(list1):
        hexadecimal_ans = bcdToHexaDecimal(list1)
        hexadecimal_ans.reverse()   ## reversing the list
        hexadecimal_ans = [ i.lower() for i in hexadecimal_ans]  ## upper-case alphabets to lower-case
        hexadecimal_ans = ''.join(hexadecimal_ans)
        return hexadecimal_ans
```

In [12]:
```
# Function to convert BCD to hexadecimal
def bcdToHexaDecimal(s):
    len1 = len(s)
    check = 0
    num = 0
    sum = 0
    mul = 1
    ans = []

# Iterating through the bits backwards
    i = len1 - 1

    while(i >= 0):
        sum += (ord(s[i]) - ord('0')) * mul
        mul *= 2
        check += 1

        # Computing the hexadecimal number formed
        # so far and storing it in a vector.
        if (check == 4 or i == 0):

            if (sum <= 9):
                ans.append(chr(sum + ord('0')))
            else:
                ans.append(chr(sum + 55));

            # Reinitializing all variables for next group.
            check = 0
            sum = 0
            mul = 1
        i -= 1

    len1 = len(ans)

    # Printing the hexadecimal
    # number formed so far.
    i = len1 - 1

    while(i >= 0):
        return ans
```

In [13]:
```
def rotword(word):
    rot=word[1:]
    rot.append(word[0])
    return rot
```

In [14]:
```
def col1_generation(iteration_var, hex_key):

    ## round-constant table
    ## key-round and value-hexadecimal
    round_constant = {1:'1', 2:'2', 3:'4', 4:'8', 5:'10', 6:'20', 7:'40', 8:'80', 9:'1B', 10:'36'}

    ## taking the last column words
    last_col = hex_key[-1]
    left_shift = rotword(last_col)

    ## sub-word generation from S-Box
    subword = []
    for i in left_shift:
        val = key_sbox(i)
        if len(val)!=2:
            val = '0' + val
            subword.append(val)
        else:
            subword.append(val)

    ## subword --> hexadecimal(subword)
    y1 = Hexaword_BCD(subword)

    ## subword (XOR) Round Constant
    element = round_constant[iteration_var]
    initial = HexaDecimaltoBCD(element)
    initial_length = len(initial)

    if (initial_length == 1):
        between = initial
        temp_1 = [ [0 for j in range(4) for i in range(1)] ]
        last = [ [0 for j in range(4)] for i in range(6) ]
        temp_1.extend(between)
        temp_1.extend(last)
        final = temp_1
        if (element==1): ## if there is a single element(0,1,2, ... ,9) from the s-box --> length=1
            before1 = [ [0 for j in range(4) for i in range(1)] ]
            before1.extend(final)
            final = before1

    elif (initial_length == 2):  ## if there is a two element(11, 1a, b1, ...) from the s-box --> length=2
        last = [ [0 for j in range(4)] for i in range(6) ]
        first = initial
        first.extend(last)
        final = first

    final = [ "".join(list(map(str, i))) for i in final ]
    round_list = final

    round_ans = XOR(y1, round_list) ## --> g(col4_before)

    col1_before = [ HexaDecimaltoBCD(i) for i in hex_key[0] ]
    col1_before = list(np.concatenate(col1_before).flat)

    ## col1 --> col1_before (exor) g(col4_before)
    col1 = XOR(col1_before, round_ans)
    col1 = bcd_hexadecimal(col1)

    return col1
```

In [15]:
```
def key_generation(hex_key):
    complete_keys = []
    for i in range(1,11): ## 10 times running
        temp = []

        ## 1st col word = col_before[0] (exor) g(col_before[-1])
        col1_hex = col1_generation(i, hex_key)
        col1_bin  = Hexaword_BCD(col1_hex)

        for j in range(2,5): ## each loop, 3 times running ( remaining 3 words)
            if (j==2):  ## 2nd col words
                col1_before_bin = Hexaword_BCD(hex_key[j-1])
                col2_bin = XOR(col1_before_bin, col1_bin)
                col2_hex = bcd_hexadecimal(col2_bin)

            else:  ## 3rd and 4th column words
                col2_before_bin = Hexaword_BCD(hex_key[j-1])
                col3_bin = XOR(col2_before_bin, col2_bin)
                col3_hex = bcd_hexadecimal(col3_bin)
                col2_bin = Hexaword_BCD(col3_hex)
                temp.append(col3_hex)

        ## once again generating the hex_key
        hex_key = []

        hex_key.append(col1_hex)
        hex_key.append(col2_hex)
        hex_key.extend(temp)

        complete_keys.append(hex_key)
    return complete_keys
```

Round Operation

In [16]:
```
def add_round_key(hex_str, complete_keys,round_num, not_rest_rounds): ## exor with plain text and key

    if not_rest_rounds:
        temp_key = complete_keys[round_num]
    else:
        temp_key = pd.DataFrame(complete_keys[round_num]).T.values.tolist()

    add_round_key = []
    for i in range(4):
        bin_str    = Hexaword_BCD(hex_str[i])
        bin_keys = Hexaword_BCD(temp_key[i])
        bin_xor   = XOR(bin_str, bin_keys)
        bcd_round_key = bcd_hexadecimal(bin_xor)
        add_round_key.append(bcd_round_key)

    return add_round_key
```

Substitution Box

In [17]:
```
def subtitute_box(add_round_key):  ## output from add_round_key
    subword = []
    for i in range(4):
        temp_word = []
        for j in add_round_key:
            val = key_sbox(j[i])
            if len(val)!=2:
                val = '0' + val
                temp_word.append(val)
            else:
                temp_word.append(val)

        ## subword --> hexadecimal(subword)
        temp_word = Hexaword_BCD(temp_word)
        temp_word_hexa = bcd_hexadecimal(temp_word)
        subword.append(temp_word_hexa)
        temp_word_hexa = []

    return subword
```

Shift Rows

In [18]:
```
def shift_rows(subtitute_box_ans):
    ## shifting rows and columns
    shift_rows = []
    subword = collections.deque(subtitute_box_ans)
    for i in range(4):
        temp = collections.deque(subtitute_box_ans[i])
        temp.rotate(-i)
        shift_rows.append(list(temp))
    return shift_rows
```

Mix Columns

In [19]:
```
def mix_columns_each_round(shift_rows, rest_rounds):

    multiple = [['02', '03', '01', '01'],
                     ['01', '02', '03', '01'],
                     ['01', '01', '02', '03'],
                     ['03', '01', '01', '02']]

    if rest_rounds:
        shift_rows = pd.DataFrame(shift_rows).T.values.tolist()

    prod_ans1 = []
    row_ans = []
    final_ans  = []

    for i in range(4):
        for j in range(4):
            for k in range(4):

                ## hexadecimal to BCD
                num1 = Hexaword_BCD(multiple[i][k])
                num2 = Hexaword_BCD(shift_rows[j][k])

                ## Product operation
                num1_list = [int(item) for sublist in num1 for item in sublist]
                num2_list = [int(item) for sublist in num2 for item in sublist]
                ans = list(np.poly1d(num1_list) * np.poly1d(num2_list)) ## binary multiplication
                prod_ans1.append(ans)

            max_length = max([len(p) for p in prod_ans1])

           ## adding 0's
            temp_match_len = []
            for a in prod_ans1:
                for b in range(max_length - len(a)):
                    a.insert(0, 0)
                element = ''.join(map(str, a))
                #binary_to_polynomial(element)
                temp_match_len.append(a)
            prod_ans1 = temp_match_len

            ## summing up the polynomials
            temp_sum = []
            for c in range(max_length):
                temp_sum.append(sum([sub[c] for sub in prod_ans1]))
            for d in range(len(temp_sum)):
                if (temp_sum[d] %2!= 0): temp_sum[d] = 1  ## sum is odd number put 1
                else:  temp_sum[d] = 0                              ## sum is even number put 0
            prod_ans1 = temp_sum

            prod_ans1= ''.join(map(str, prod_ans1))

            if len(prod_ans1)>8:
                irreducible_polynomial = '100011011'
                ir_ans = binary_division_modulo_2(prod_ans1, irreducible_polynomial)
                prod_ans1= ir_ans
                hexadecimal_ans = bcd_to_hexadecimal(prod_ans1)
                row_ans.append(hexadecimal_ans)
            else:
                hexadecimal_ans = bcd_to_hexadecimal(prod_ans1)
                row_ans.append(hexadecimal_ans)

            prod_ans1 = []
        final_ans.append(row_ans)
        row_ans = []
    return final_ans
```

In [20]:
```
def main():
    round_num = 0
    add_round_key
    subtitute_box
    shift_rows
    mix_columns_each_round
```

Getting user inputs

In [21]:
```
def text_hexadecimal(text): ## all the blocks --> 16bytes
    hex_text   = []
    for i in text: ## character -> ascii (decimal) -> hexa-decimal
        hex_text.append(hex(ord(i))[2:])
    return hex_text ## 16-byte representation of the text
```

In [22]:
```
plain_text = "Two One Nine Two"
plain_key = 'Thats my Kung Fu'

hex_str = text_hexadecimal(plain_text)
hex_key = text_hexadecimal(plain_key)

print("Text Hexadecimal Representation : ",hex_str)
print("Key Hexadecimal Representation : ",hex_key)
```

```
Text Hexadecimal Representation :  ['54', '77', '6f', '20', '4f', '6e', '65', '20', '4e', '69', '6e', '65', '20', '54', '77', '6f']
Key Hexadecimal Representation :  ['54', '68', '61', '74', '73', '20', '6d', '79', '20', '4b', '75', '6e', '67', '20', '46', '75']
```

In [23]:
```
hex_str = [hex_str[i:i+4] for i in range(0, len(hex_str), 4)]
print("Hexadecimal Representation : ",hex_str)
```

```
Hexadecimal Representation :  [['54', '77', '6f', '20'], ['4f', '6e', '65', '20'], ['4e', '69', '6e', '65'], ['20', '54', '77', '6f']]
```

Key Generation

In [24]:
```
## splitting into 4*4 matrix
hex_key = [hex_key[i:i+4] for i in range(0, len(hex_key), 4)]

r0_key = [] ## to accomodate Round-0th key
r0_key.extend(hex_key)

r1_r10_keys = key_generation(hex_key)
r1_r10_keys.insert(0,(r0_key))
complete_keys = r1_r10_keys

for i in range(len(complete_keys)):
    print("Round - {} keys --> ".format(i),end=' ')
    print(*list(itertools.chain(*complete_keys[i])))
```

```
Round - 0 keys -->  54 68 61 74 73 20 6d 79 20 4b 75 6e 67 20 46 75
Round - 1 keys -->  e2 32 fc f1 91 12 91 88 b1 59 e4 e6 d6 79 a2 93
Round - 2 keys -->  56 08 20 07 c7 1a b1 8f 76 43 55 69 a0 3a f7 fa
Round - 3 keys -->  d2 60 0d e7 15 7a bc 68 63 39 e9 01 c3 03 1e fb
Round - 4 keys -->  a1 12 02 c9 b4 68 be a1 d7 51 57 a0 14 52 49 5b
Round - 5 keys -->  b1 29 3b 33 05 41 85 92 d2 10 d2 32 c6 42 9b 69
Round - 6 keys -->  bd 3d c2 87 b8 7c 47 15 6a 6c 95 27 ac 2e 0e 4e
Round - 7 keys -->  cc 96 ed 16 74 ea aa 03 1e 86 3f 24 b2 a8 31 6a
Round - 8 keys -->  8e 51 ef 21 fa bb 45 22 e4 3d 7a 06 56 95 4b 6c
Round - 9 keys -->  bf e2 bf 90 45 59 fa b2 a1 64 80 b4 f7 f1 cb d8
Round - 10 keys -->  28 fd de f8 6d a4 24 4a cc c0 a4 fe 3b 31 6f 26
```

Round-0 to Round-10

Round-0

In [25]:
```
rnd_cnt = 0
print("\n-------------------------------------------Round num - {} -------------------------------------------".format(rnd_cnt))
add_round_key_ans = add_round_key(hex_str, complete_keys,rnd_cnt, True)
print("\nAdd Round Key : ",add_round_key_ans)
```

```
-------------------------------------------Round num - 0 -------------------------------------------

Add Round Key :  [['00', '1f', '0e', '54'], ['3c', '4e', '08', '59'], ['6e', '22', '1b', '0b'], ['47', '74', '31', '1a']]
```

Round-1

In [26]:
```
rnd_cnt = 1
print("\n-------------------------------------------Round num - {} -------------------------------------------".format(rnd_cnt))
subtitute_box_ans = subtitute_box(add_round_key_ans)
shift_rows_ans = [['63', '2f', 'af', 'a2'],
                           ['eb', '93', 'c7', '20'],
                           ['9f', '92', 'ab', 'cb'],
                           ['a0', 'c0', '30', '2b']]
mix_columns_ans = mix_columns_each_round(shift_rows_ans,False)

print("Sibstitute Box : ",subtitute_box_ans)
print("Shift Rows      : ",shift_rows_ans)
print("Mix Columns   : ",mix_columns_ans)

print("\nEXOR OF ")
print("                 Mix Column ans : ", mix_columns_ans)
print("                 Key                    : ",pd.DataFrame(complete_keys[rnd_cnt]).T.values.tolist())

add_round_key_ans = add_round_key(mix_columns_ans, complete_keys,rnd_cnt, False)

print("\nAdd Round Key : ",add_round_key_ans)
```

```
-------------------------------------------Round num - 1 -------------------------------------------
Sibstitute Box :  [['63', 'eb', '9f', 'a0'], ['c0', '2f', '93', '92'], ['ab', '30', 'af', 'c7'], ['20', 'cb', '2b', 'a2']]
Shift Rows      :  [['63', '2f', 'af', 'a2'], ['eb', '93', 'c7', '20'], ['9f', '92', 'ab', 'cb'], ['a0', 'c0', '30', '2b']]
Mix Columns   :  [['ba', '84', 'e8', '1b'], ['75', 'a4', '8d', '40'], ['f4', '8d', '06', '7d'], ['7a', '32', '0e', '5d']]

EXOR OF
                Mix Column ans :  [['ba', '84', 'e8', '1b'], ['75', 'a4', '8d', '40'], ['f4', '8d', '06', '7d'], ['7a', '32', '0e', '5d']]
                Key                    :  [['e2', '91', 'b1', 'd6'], ['32', '12', '59', '79'], ['fc', '91', 'e4', 'a2'], ['f1', '88', 'e6', '93']]

Add Round Key :  [['58', '15', '59', 'cd'], ['47', 'b6', 'd4', '39'], ['08', '1c', 'e2', 'df'], ['8b', 'ba', 'e8', 'ce']]
```

Round-2 to Round-9

In [27]:
```
for i in range(2,10):  ## from Round-1 to Round-2
    rnd_cnt = i
    print("\n-------------------------------------------Round num - {} -------------------------------------------".format(rnd_cnt))

    subtitute_box_ans   = subtitute_box(add_round_key_ans)
    shift_rows_ans        = shift_rows(pd.DataFrame(subtitute_box_ans).T.values.tolist())
    mix_columns_ans    = mix_columns_each_round(shift_rows_ans, True)

    print("Sibstitute Box : ",subtitute_box_ans)
    print("Shift Rows      : ",shift_rows_ans)
    print("Mix Columns   : ",mix_columns_ans)

    print("\nEXOR OF ")
    print("                 Mix Column ans : ", mix_columns_ans)
    print("                 Key                    : ",pd.DataFrame(complete_keys[rnd_cnt]).T.values.tolist())

    add_round_key_ans = add_round_key(mix_columns_ans, complete_keys, rnd_cnt, False)
    print("\nAdd Round Key : ",add_round_key_ans)
```

```
-------------------------------------------Round num - 2 -------------------------------------------
Sibstitute Box :  [['6a', 'a0', '30', '3d'], ['59', '4e', '9c', 'f4'], ['cb', '48', '98', '9b'], ['bd', '12', '9e', '8b']]
Shift Rows      :  [['6a', '59', 'cb', 'bd'], ['4e', '48', '12', 'a0'], ['98', '9e', '30', '9c'], ['8b', '3d', 'f4', '9b']]
Mix Columns   :  [['15', 'c9', '7f', '9d'], ['ce', '4d', '4b', 'c2'], ['89', '71', 'be', '88'], ['65', '47', '97', 'cd']]

EXOR OF
                Mix Column ans :  [['15', 'c9', '7f', '9d'], ['ce', '4d', '4b', 'c2'], ['89', '71', 'be', '88'], ['65', '47', '97', 'cd']]
                Key                    :  [['56', 'c7', '76', 'a0'], ['08', '1a', '43', '3a'], ['20', 'b1', '55', 'f7'], ['07', '8f', '69', 'fa']]

Add Round Key :  [['43', '0e', '09', '3d'], ['c6', '57', '08', 'f8'], ['a9', 'c0', 'eb', '7f'], ['62', 'c8', 'fe', '37']]

-------------------------------------------Round num - 3 -------------------------------------------
Sibstitute Box :  [['1a', 'b4', 'd3', 'aa'], ['ab', '5b', 'ba', 'e8'], ['01', '30', 'e9', 'bb'], ['27', '41', 'd2', '9a']]
Shift Rows      :  [['1a', 'ab', '01', '27'], ['5b', '30', '41', 'b4'], ['e9', 'd2', 'd3', 'ba'], ['9a', 'aa', 'e8', 'bb']]
Mix Columns   :  [['aa', '65', 'fa', '88'], ['16', '0c', '05', '3a'], ['3d', 'c1', 'de', '2a'], ['b3', '4b', '5a', '0a']]

EXOR OF
                Mix Column ans :  [['aa', '65', 'fa', '88'], ['16', '0c', '05', '3a'], ['3d', 'c1', 'de', '2a'], ['b3', '4b', '5a', '0a']]
                Key                    :  [['d2', '15', '63', 'c3'], ['60', '7a', '39', '03'], ['0d', 'bc', 'e9', '1e'], ['e7', '68', '01', 'fb']]

Add Round Key :  [['78', '70', '99', '4b'], ['76', '76', '3c', '39'], ['30', '7d', '37', '34'], ['54', '23', '5b', 'f1']]

-------------------------------------------Round num - 4 -------------------------------------------
Sibstitute Box :  [['bc', '38', '04', '20'], ['51', '38', 'ff', '26'], ['ee', 'eb', '9a', '39'], ['b3', '12', '18', 'a1']]
Shift Rows      :  [['bc', '51', 'ee', 'b3'], ['38', 'eb', '12', '38'], ['9a', '18', '04', 'ff'], ['a1', '20', '26', '39']]
Mix Columns   :  [['10', 'bc', 'd3', 'f3'], ['d8', '94', 'e0', 'e0'], ['53', 'ea', '9e', '25'], ['24', '40', '73', '7b']]

EXOR OF
                Mix Column ans :  [['10', 'bc', 'd3', 'f3'], ['d8', '94', 'e0', 'e0'], ['53', 'ea', '9e', '25'], ['24', '40', '73', '7b']]
                Key                    :  [['a1', 'b4', 'd7', '14'], ['12', '68', '51', '52'], ['02', 'be', '57', '49'], ['c9', 'a1', 'a0', '5b']]

Add Round Key :  [['b1', '08', '04', 'e7'], ['ca', 'fc', 'b1', 'b2'], ['51', '54', 'c9', '6c'], ['ed', 'e1', 'd3', '20']]

-------------------------------------------Round num - 5 -------------------------------------------
Sibstitute Box :  [['c8', '74', 'd1', '55'], ['30', 'b0', '20', 'f8'], ['f2', 'c8', 'dd', '66'], ['94', '37', '50', 'b7']]
Shift Rows      :  [['c8', '30', 'f2', '94'], ['b0', 'c8', '37', '74'], ['dd', '50', 'd1', '20'], ['b7', '55', 'f8', '66']]
Mix Columns   :  [['2a', '26', '8f', 'e9'], ['78', '1e', '0c', '7a'], ['1b', 'a7', '6f', '0a'], ['5b', '62', '00', '3f']]

EXOR OF
                Mix Column ans :  [['2a', '26', '8f', 'e9'], ['78', '1e', '0c', '7a'], ['1b', 'a7', '6f', '0a'], ['5b', '62', '00', '3f']]
                Key                    :  [['b1', '05', 'd2', 'c6'], ['29', '41', '10', '42'], ['3b', '85', 'd2', '9b'], ['33', '92', '32', '69']]

Add Round Key :  [['9b', '23', '5d', '2f'], ['51', '5f', '1c', '38'], ['20', '22', 'bd', '91'], ['68', 'f0', '32', '56']]

-------------------------------------------Round num - 6 -------------------------------------------
Sibstitute Box :  [['14', 'd1', 'b7', '45'], ['26', 'cf', '93', '8c'], ['4c', '9c', '7a', '23'], ['15', '07', '81', 'b1']]
Shift Rows      :  [['14', '26', '4c', '15'], ['cf', '9c', '07', 'd1'], ['7a', '81', 'b7', '93'], ['b1', '45', '8c', '23']]
Mix Columns   :  [['a9', '37', 'aa', 'f2'], ['ae', 'd8', '0c', '21'], ['e7', '6c', 'b1', '9c'], ['f0', 'fd', '67', '3b']]

EXOR OF
                Mix Column ans :  [['a9', '37', 'aa', 'f2'], ['ae', 'd8', '0c', '21'], ['e7', '6c', 'b1', '9c'], ['f0', 'fd', '67', '3b']]
                Key                    :  [['bd', 'b8', '6a', 'ac'], ['3d', '7c', '6c', '2e'], ['c2', '47', '95', '0e'], ['87', '15', '27', '4e']]

Add Round Key :  [['14', '8f', 'c0', '5e'], ['93', 'a4', '60', '0f'], ['25', '2b', '24', '92'], ['77', 'e8', '40', '75']]

-------------------------------------------Round num - 7 -------------------------------------------
Sibstitute Box :  [['fa', 'dc', '3f', 'f5'], ['73', '49', 'f1', '9b'], ['ba', 'd0', '36', '09'], ['58', '76', '4f', '9d']]
Shift Rows      :  [['fa', '73', 'ba', '58'], ['49', 'd0', '76', 'dc'], ['36', '4f', '3f', 'f1'], ['9d', 'f5', '9b', '09']]
Mix Columns   :  [['9f', '37', '51', '37'], ['af', 'ec', '8c', 'fa'], ['63', '39', '04', '66'], ['4b', 'fb', 'b1', 'd7']]

EXOR OF
                Mix Column ans :  [['9f', '37', '51', '37'], ['af', 'ec', '8c', 'fa'], ['63', '39', '04', '66'], ['4b', 'fb', 'b1', 'd7']]
                Key                    :  [['cc', '74', '1e', 'b2'], ['96', 'ea', '86', 'a8'], ['ed', 'aa', '3f', '31'], ['16', '03', '24', '6a']]

Add Round Key :  [['53', '43', '4f', '85'], ['39', '06', '0a', '52'], ['8e', '93', '3b', '57'], ['5d', 'f8', '95', 'bd']]

-------------------------------------------Round num - 8 -------------------------------------------
Sibstitute Box :  [['ed', '12', '19', '4c'], ['1a', '6f', 'dc', '41'], ['84', '67', 'e2', '2a'], ['97', '00', '5b', '7a']]
Shift Rows      :  [['ed', '1a', '84', '97'], ['6f', '67', '00', '12'], ['e2', '5b', '19', 'dc'], ['7a', '4c', '41', '2a']]
Mix Columns   :  [['e8', '8a', '4b', 'f5'], ['74', '75', 'ee', 'e6'], ['d3', '1f', '75', '58'], ['55', '8a', '0c', '38']]

EXOR OF
                Mix Column ans :  [['e8', '8a', '4b', 'f5'], ['74', '75', 'ee', 'e6'], ['d3', '1f', '75', '58'], ['55', '8a', '0c', '38']]
                Key                    :  [['8e', 'fa', 'e4', '56'], ['51', 'bb', '3d', '95'], ['ef', '45', '7a', '4b'], ['21', '22', '06', '6c']]

Add Round Key :  [['66', '70', 'af', 'a3'], ['25', 'ce', 'd3', '73'], ['3c', '5a', '0f', '13'], ['74', 'a8', '0a', '54']]

-------------------------------------------Round num - 9 -------------------------------------------
Sibstitute Box :  [['33', '3f', 'eb', '92'], ['51', '8b', 'be', 'c2'], ['79', '66', '76', '67'], ['0a', '8f', '7d', '20']]
Shift Rows      :  [['33', '51', '79', '0a'], ['8b', '66', '8f', '3f'], ['76', '7d', 'eb', 'be'], ['20', '92', 'c2', '67']]
Mix Columns   :  [['b6', 'e7', '51', '8c'], ['84', '88', '98', 'ca'], ['34', '60', '66', 'fb'], ['e8', 'd7', '70', '51']]

EXOR OF
                Mix Column ans :  [['b6', 'e7', '51', '8c'], ['84', '88', '98', 'ca'], ['34', '60', '66', 'fb'], ['e8', 'd7', '70', '51']]
                Key                    :  [['bf', '45', 'a1', 'f7'], ['e2', '59', '64', 'f1'], ['bf', 'fa', '80', 'cb'], ['90', 'b2', 'b4', 'd8']]

Add Round Key :  [['09', 'a2', 'f0', '7b'], ['66', 'd1', 'fc', '3b'], ['8b', '9a', 'e6', '30'], ['78', '65', 'c4', '89']]
```

Round-10

In [28]:
```
rnd_cnt = 10
print("\n-------------------------------------------Round num - {} -------------------------------------------".format(rnd_cnt))
subtitute_box_ans = subtitute_box(add_round_key_ans)
shift_rows_ans     = shift_rows(pd.DataFrame(subtitute_box_ans).T.values.tolist())

print("Substitute Box : ",subtitute_box_ans)
print("Shift Rows      : ",shift_rows_ans)
print("\nEXOR OF ")
print("                 Mix Column ans : ", mix_columns_ans)
print("                 Key                    : ",pd.DataFrame(complete_keys[rnd_cnt]).T.values.tolist())

add_round_key_ans = add_round_key(shift_rows_ans, complete_keys,rnd_cnt,False)
print("\nAdd Round Key : ",add_round_key_ans)
```

```
-------------------------------------------Round num - 10 -------------------------------------------
Substitute Box :  [['01', '33', '3d', 'bc'], ['3a', '3e', 'b8', '4d'], ['8c', 'b0', '8e', '1c'], ['21', 'e2', '04', 'a7']]
Shift Rows      :  [['01', '3a', '8c', '21'], ['3e', 'b0', 'e2', '33'], ['8e', '04', '3d', 'b8'], ['a7', 'bc', '4d', '1c']]

EXOR OF
                Mix Column ans :  [['b6', 'e7', '51', '8c'], ['84', '88', '98', 'ca'], ['34', '60', '66', 'fb'], ['e8', 'd7', '70', '51']]
                Key                    :  [['28', '6d', 'cc', '3b'], ['fd', 'a4', 'c0', '31'], ['de', '24', 'a4', '6f'], ['f8', '4a', 'fe', '26']]

Add Round Key :  [['29', '57', '40', '1a'], ['c3', '14', '22', '02'], ['50', '20', '99', 'd7'], ['5f', 'f6', 'b3', '3a']]
```

Cipher Text

In [29]:
```
cipher_text = pd.DataFrame(add_round_key_ans).T.values.tolist()
print("Cipher Text : ")
cipher_text
```

```
Cipher Text :
```

Out[29]:
```
[['29', 'c3', '50', '5f'],
 ['57', '14', '20', 'f6'],
 ['40', '22', '99', 'b3'],
 ['1a', '02', 'd7', '3a']]
```

Diffie Helman Key Exchange

Prashanth.S 19MID0020

Importing the Necessary Libraries

In [1]:
```
import numpy as np
import random
```

In [2]:
```
'''
Xa = 3 ## private key of Agent-X
Ya = 7 ## private key of Agent-Y
A = ## public key of Agent-X (shared to Agent-Y)
B = ## public key of Agent-Y (shared to Agent-X)
S = ## shared key
'''
```

Out[2]:
```
'\nXa = 3 ## private key of Agent-X\nYa = 7 ## private key of Agent-Y\nA = ## public key of Agent-X (shared to Agent-Y)\nB = ## public key of Agent-Y (shared to Agent-X)\nS = ## shared key \n'
```

Primitive Roots Creation

In [3]:
```
def primitive_roots_table_creation(m):
    list1 = []
    b = 1
    for i in range(1,m-1):
        temp = []
        b+=1
        for j in range(1,m):
            temp.append(np.power(b,j) % (m))
        list1.append(temp)
    return list1
```

In [4]:
```
def primitive_roots_value(primitive_roots_table, m):
    # np.unique() --> returns the number bo unique values
    m = 13
    primitive_roots = []
    for i in primitive_roots_table:
        if (len(np.unique(i)) == m-1):
            primitive_roots.append(i[0])
    return primitive_roots
```

In [5]:
```
def public_key_generation(generator, private_key, premitive_root):
    return ((generator**private_key) % premitive_root)
def sharing(pub1, pub2):
    return (pub2, pub1)
def share_secret_key(shared_key, private_key, premitive_root):
    return ((shared_key**private_key) % premitive_root)
```

In [6]:
```
def isPrime(num):
    cnt = 0
    for i in range(2, np.int(np.sqrt(num))):
        if ((num%i) == 0):
            cnt = 1
            return False ## composite number
        if (cnt==0):
            return True ## prime number
```

In [7]:
```
def start():
    ## Alice portion
    Xa = 3
    Ya = public_key_generation(generator, Xa, premitive_root)
    print("Alice's public key : ",Ya)

    ## Bob portion
    Xb = 7
    Yb = public_key_generation(generator, Xb, premitive_root)

    Ya, Yb = sharing(Yb, Ya)
    print("\nAfter sharing")
    print("Alice's public key : ",Ya)
    print("Bob's public key : ",Yb)

    alice_K = share_secret_key(Yb, Xa, premitive_root)
    print("\nAlice's shared key : ",alice_K)

    bob_K = share_secret_key(Ya, Xb, premitive_root)
    print("Bob's shared key : ",bob_K)

    if (alice_K == bob_K):
        return alice_K
    else:
        return 0
```

In [8]:
```
def shift_characters(str1, n):
     return ''.join(chr((ord(char) - 97 - n) % 26 + 97) for char in str1)
```

In [9]:
```
primitive_roots_table = primitive_roots_table_creation(13)
primitive_roots = primitive_roots_value(primitive_roots_table,13)
primitive_roots
```

Out[9]:
```
[2, 6, 7, 11]
```

In [10]:
```
premitive_root = 13
if isPrime(premitive_root):
    generator = random.choice(primitive_roots)
if (generator < premitive_root):
    print("Continue")
    key_match = start()
else:
    print("Disontinue")
```

```
Continue
Alice's public key :  5

After sharing
Alice's public key :  5
Bob's public key :  2

Alice's shared key :  8
Bob's shared key :  8
```

```
/var/folders/gq/nsqxf83n1813yysq2l8vvtxc0000gn/T/ipykernel_3491/2137807101.py:3: DeprecationWarning: `np.int` is a deprecated alias for the builtin `int`. To silence this warning, use `int` by itself. Doing this will not modify any behavior and is safe. When replacing `np.int`, you may wish to use e.g. `np.int64` or `np.int32` to specify the precision. If you wish to review your current use, check the release note link for additional information.
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/devdocs/release/1.20.0-notes.html#deprecations
  for i in range(2, np.int(np.sqrt(num))):
```

Encryption

In [11]:
```
def encryption(plain_text):
    n = key_match
    return shift_characters(plain_text, n)
```

Decryption

In [12]:
```
def decryption(cipher_text):
    n = -key_match
    return shift_characters(cipher_text, n)
```

In [13]:
```
plain_text = "prashanth"
cipher_text = encryption(plain_text)
print(cipher_text)
```

```
hjskzsflz
```

In [14]:
```
decrypt_text = decryption(cipher_text)
print(decrypt_text)
```

```
prashanth
```

Rivest-Shamir-Adleman Encryption Algorithm

Prashanth.S 19MID0020

Importing the Necessary Libraries

In [1]:
```
import numpy as np
import random
```

Operational Functions

In [2]:
```
def si(n): return n-1
```

In [3]:
```
def num_check(num1, num2,condition):
    while condition:
        random_num = random.randint(2, (si(num1) * si(num2)) - 1)
        if (np.gcd(random_num, (si(num1) * si(num2))) == 1):
            break
        else:
            continue

    return random_num
```

In [4]:
```
def modulo_multiplicative_inverse(a, m):
    for x in range(1, m):
        if (((a%m) * (x%m)) % m == 1):return x
    return -1
```

In [5]:
```
## {e,n}
public_key = []

## {d,n}
private_key = []

prime_1 = 3
prime_2 = 11

public_key.append(num_check(prime_1, prime_2, True))
public_key.append(prime_1 * prime_2)


modulo_ans = modulo_multiplicative_inverse(public_key[0], (si(prime_1) * si(prime_2)))
private_key.append(modulo_ans)
private_key.append(public_key[1])
```

In [6]:
```
print(public_key)
print(private_key)
```

```
[3, 33]
[7, 33]
```

In [7]:
```
e = public_key[0]
#e = 7
d = private_key[0]
n = public_key[1]
```

Encryption

In [8]:
```
message=5
if (message < n):
    cipher_text = (message**e % n)
print(cipher_text)
```

```
26
```

Decryption

In [9]:
```
decrypt_text = (cipher_text**d % n)
decrypt_text
```

Out[9]:
```
5
```

In [10]:
```
if (message == decrypt_text):
    print("Successful Transmission")
else:
    print("Not Successful Transmission")
```

```
Successful Transmission
```

Elgamal Crypto System

Prashanth.S 19MID0020

In [1]:

```
import pandas as pd
import numpy as np
```

In [2]:

```
## Alice
Xa = 50 ## private_key
Ya = 14 ## public key

## Bob
Xb = 39 ## private_key
Yb = 53 ## public key

## general
prime_number = 61
generator = 6

message = 4
```

In [3]:

```
## Bob sends message to Alice
cipher_text = ( ( (Ya**Xb) * message )  % prime_number)
cipher_text
```

Out[3]:

```
57
```

In [4]:

```
Xa = -1 * Xa
if (Xa<=0):
    Xa = prime_number - 1 + Xa
Xa
```

Out[4]:

```
10
```

In [5]:

```
decrypt_text = ((cipher_text%prime_number) * (Yb**Xa)%prime_number)%prime_number
```

In [6]:

```
print("Plain Text : ", message)
print("Encrypted Plain Text    : ", cipher_text)
print("Decrypted Cipher Text : ",decrypt_text)
```

```
Plain Text :  4
Encrypted Plain Text    :  57
Decrypted Cipher Text :  4
```

Elgamal Based Digital Signature

Prashanth.S 19MID0020

Importing the Libraries

In [1]:
```
import numpy as np
import random
```

Getting inputs from the user

In [2]:
```
prime_number = 11
generator = 2
Message = 5
```

Checking the validity of Generator and Prime Number

In [3]:
```
if (generator < prime_number):
    if (np.gcd(prime_number, generator) == 1):
        pass
```

Private Key Generation

In [4]:
```
if (generator < prime_number):
    private_key = random.randint(2, prime_number-2)

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

```
Private Key :  2
```

Public Key Generation

In [5]:
```
public_key = (generator**private_key) % prime_number
print("Public Key : ",public_key)
```

```
Public Key :  4
```

Digital Signature Generation

In [6]:
```
hash_value = hash(Message)
print("Hash Value : ",hash_value)
```

```
Hash Value :  5
```

Select the secret-key (random number)

In [7]:
```
for i in range(2, prime_number):
    if np.gcd(i, prime_number - 1) == 1:
        secret_key = i
        break

print("Secret Key : ",secret_key)
```

```
Secret Key :  3
```

In [8]:
```
def inverse_value_generate(secret_key, prime_number):
    condition = True
    cnt = 0

    while condition:
        cnt+=1
        if ( (secret_key  * cnt) % prime_number == 1):
            condition = False

    return cnt
```

Computing the digital signature

In [9]:
```
 inv_secret_key = inverse_value_generate(secret_key, prime_number - 1)

y1 = (generator ** secret_key) % prime_number
y2 = ((inv_secret_key) * (hash_value - (private_key * y1))) % (prime_number - 1)

print("Digital Signature Y1 = {} and Y2 = {}".format(y1, y2))
```

```
Digital Signature Y1 = 8 and Y2 = 3
```

User-X to User-Y

In [10]:
```
print("User-X sending ---")
Message = 5
print("Message : {} and Digital Signature Y1 = {} and Y2 = {}".format(Message, y1, y2))
```

```
User-X sending ---
Message : 5 and Digital Signature Y1 = 8 and Y2 = 3
```

User-Y

In [11]:
```
hash_value = hash(Message)
print("Hash Value : ",hash_value)
```

```
Hash Value :  5
```

In [12]:
```
V1 = (generator ** hash_value) % prime_number
V2 = ( (public_key ** y1) * (y1 ** y2) ) % prime_number

print("V1 : ",V1)
print("V2 : ",V2)
```

```
V1 :  10
V2 :  10
```

In [13]:
```
if (V1 == V2):
    print("Successful")
```

```
Successful
```