```
from sys import exit
from time import time

KeyLength = 10
SubKeyLength = 8
DataLength = 8
FLength = 4

# Tables for initial and final permutations (b1, b2, b3, etc.. )
IPtable = (2, 6, 3, 1, 4, 8, 5, 7)
FPtable = (4, 1, 3, 5, 7, 2, 8, 6)

# Tables for subkey generation (k1, k2, k3, etc ...)
P10table = (3, 5, 2, 7, 4, 10, 1, 9, 8, 6)
P8table = (6, 3, 7, 4, 8, 5, 10, 9)

# Tables for the fk function
EPtable = (4, 1, 2, 3, 2, 3, 4, 1)
S0table = (1, 0, 3, 2, 3, 2, 1, 0, 0, 2, 1, 3, 3, 1, 3, 2)
S1table = (0, 1, 2, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
S1_mod =  (2, 1, 0, 3, 2, 0, 1, 3, 3, 0, 1, 0, 2, 1, 0, 3)
P4table = (2, 4, 3, 1)

def perm(inputByte, permTable):
#This is the function that permutates an input byte and permutation table
(Variables defined globally)
    """Permute input byte according to permutation table"""
#This will then exectue the task of rearranging the bites of the inputByte and
mapping the first value and second values to their coressponding locations within
the permutation table
    outputByte = 0
    for index, elem in enumerate(permTable):
        if index >= elem:
            outputByte |= (inputByte & (128 >> (elem - 1))) >> (index - (elem - 1))
#This for loop loops through the elements of the permTable selected and if the
index of the element is greate than or equal to the elemtn value and if true the
corresponding bit in the output has been set and it will then mask the input byte
with a bit mask that has a 1 in the correct position and a 0 else
        else:
            outputByte |= (inputByte & (128 >> (elem - 1))) << ((elem - 1) - index)
#The elem - 1 is what is used to shift the input to the right if the element is
less than the element value, allowing to get the bit in the correct position.
Following the result is masked with bit mask before, then OR bitwise operator is
used with the output byte
    return outputByte

def ip(inputByte):
    """Perform the initial permutation on data"""
    return perm(inputByte, IPtable)
#This takes the input byte of the data and then permutates it with the intial
permutation or the IP defined above

def fp(inputByte):
    """Perform the final permutation on data"""
#This function will swap the inputbye by breaking it down by the low bits 0-3 and
the high bits 4-7, intern this function shiftes inputbyte left by 4bits moving high
to low and then masked with the hex code 0xff (11111111)
    return perm(inputByte, FPtable)
```

```
def swap(inputByte):
    """Swap the two nibbles of data"""
    return (inputByte << 4 | inputByte >> 4) & 0xff

def keyGen(key):
    """Generate the two required subkeys"""
    def leftShift(keyBitList):

#This performs a circular left shift on the list of bits from keybitlist variable
this will then perform a shift on the first and second five bits returing that
result as shiftedkey
        shiftedKey = [None] * KeyLength
#Creates the shiftedkey list with same length as the keybit list
        shiftedKey[0:9] = keyBitList[1:10]
#The first 9 bits of the shiftedkey are assigned to values 2 to 10 of the key bit
list
        shiftedKey[4] = keyBitList[0]
#The the first bit of shifted key is assigned to the value of bit 1 of keybit list
and then the 5th bit of shiftedkey is assigned the value of bit 6 of keybitlist
        shiftedKey[9] = keyBitList[5]
        return shiftedKey



#This will convert the input 10 digit key from a integer to a list of binary
numbers
    keyList = [(key & 1 << i) >> i for i in reversed(range(KeyLength))]
    permKeyList = [None] * KeyLength
#Takes the inpuit key which is the integer value represetngin an 8 bit binary key
generates the keylist containg the individual bits of the key
    for index, elem in enumerate(P10table):
#Begin enumertaing through the permutation table for the 10 bit key to generate the
two sub keys
        permKeyList[index] = keyList[elem - 1]

    shiftedOnce = leftShift(permKeyList)
    shiftedTwice = leftShift(leftShift(shiftedOnce))
    subKey1 = subKey2 = 0
    for index, elem in enumerate(P8table):
#Following the generation of the the shift once and twice keys the P8 permutation
are applied to each of the shifted keys to generate two 8 bit subkeys subkey1 & 2
        subKey1 += (128 >> index) * shiftedOnce[elem - 1]

        subKey2 += (128 >> index) * shiftedTwice[elem - 1]
#The following keys subKey1 and subKey2 are being stored as a tuple
    return (subKey1, subKey2)

def fk(subK, inputData):
    """Apply Feistel function on data with given subkey"""
    def F(sKey, rightside):
#Utalize the complex function in order to encrpyt data through left and right sides
of the broken down keys that will x or it with and swap
        auxillary = sKey ^ perm(swap(rightside), EPtable)
#This function creates the two indicies for the 4 bit breakdown of the two 8 bit
generated keys using the >> shift operator and the hexidecimal reference to the
positions
        index1 = ((auxillary & 0x80) >> 4) + ((auxillary & 0x40) >> 5) + \
                 ((auxillary & 0x20) >> 5) + ((auxillary & 0x10) >> 2)
# Utalizes four specific bits from the 'msk' variable using bitwise AND with the
```

```
bitmasks of 0x80 0x20 0x40 and 0x10 allows the perogram to identify the bit 1-4 and
combines it in order to form the index from the s table
        index2 = ((auxillary & 0x08) >> 0) + ((auxillary & 0x04) >> 1) + \
                 ((auxillary & 0x02) >> 1) + ((auxillary & 0x01) << 2)
#The following values are also predefined bitmasks with these values to be look up
in the S0 and S1 substitution boxes
        sbox_Outputs = swap((S0table[index1] << 2) + S1table[index2])
#These results are then combined and permutated with the 4 bit static permutation
map table

        return perm(sbox_Outputs, P4table)
#This is now the output of F to be used in the fk function

    leftside, rightside = inputData & 0xf0, inputData & 0x0f
    return (leftside ^ F(subK, rightside)) | rightside

def encrypt(key, plaintext):
    """Encrypt plaintext with given key"""
    data = fk(keyGen(key)[0], ip(plaintext))
    print("SW Value after fk operation: ", swap(data))
    return fp(fk(keyGen(key)[1], swap(data)))

def decrypt(key, ciphertext):
    """Decrypt ciphertext with given key"""
    data = fk(keyGen(key)[1], ip(ciphertext))
    print("SW Value after fk operation: ",swap(data))
    return fp(fk(keyGen(key)[0], swap(data)))


print("|--------------|Example Problem 1|-----------------|","\n")

cipher = encrypt(0b1010000010, 0b10111101)
print("Ciphertext: ", cipher)
decrpyted = decrypt(0b1110001110, cipher)
print("Plaintext: ", decrpyted, "\n")


print("|--------------|Example Problem 2|-----------------|","\n")
cipher = encrypt(0b1011110110, 0b11110110)
print("Ciphertext: ", cipher)
decrpyted = decrypt(0b1011110110, cipher)
print("Plaintext: ", decrpyted, "\n")


def spec_fk(subK, inputData):
    """Apply Feistel function on data with given subkey"""
    def F_spec(sKey, rightside):
#Utalize the complex function in order to encrpyt data through left and right sides
of the broken down keys that will x or it with and swap
        auxillary = sKey ^ perm(swap(rightside), EPtable)
#This function creates the two indicies for the 4 bit breakdown of the two 8 bit
generated keys using the >> shift operator and the hexidecimal reference to the
positions
        index1 = ((auxillary & 0x80) >> 4) + ((auxillary & 0x40) >> 5) + \
                 ((auxillary & 0x20) >> 5) + ((auxillary & 0x10) >> 2)
#Utalizes four specific bits from the 'msk' variable using bitwise AND with the
bitmasks of 0x80 0x20 0x40 and 0x10 allows the perogram to identify the bit 1-4 and
combines it in order to form the index from the s table
        index2 = ((auxillary & 0x08) >> 0) + ((auxillary & 0x04) >> 1) + \
```

```
                 ((auxillary & 0x02) >> 1) + ((auxillary & 0x01) << 2)
#The following values are also predefined bitmasks with these values to be look up
in the S0 and S1 substitution boxes
        sbox__mod_Outputs = swap((S0table[index1] << 2) + S1_mod[index2])
#These results are then combined and permutated with the 4 bit static permutation
map table

        return perm(sbox__mod_Outputs, P4table)
#This is now the output of F to be used in the fk function

    leftside, rightside = inputData & 0xf0, inputData & 0x0f
    return (leftside ^ F_spec(subK, rightside)) | rightside


def encrypt_spec(key, plaintext):
    """Encrypt plaintext with given key"""
    data = spec_fk(keyGen(key)[0], ip(plaintext))
    print("SW Value after fk operation: ", swap(data))
    return fp(spec_fk(keyGen(key)[1], swap(data)))

def decrypt_spec(key, ciphertext):
    """Decrypt ciphertext with given key"""
    data = spec_fk(keyGen(key)[1], ip(ciphertext))
    print("SW Value after fk operation: ",swap(data))
    return fp(spec_fk(keyGen(key)[0], swap(data)))

print("|--------------|Example Problem 3|-----------------|","\n")
cipher = encrypt_spec(0b1100100101, 0b00100101)
print("Ciphertext: ", cipher)
decrpyted = decrypt_spec(0b1100100101, cipher)
print("Plaintext: ", decrpyted, "\n")
```