Problem 1:

Encrypted output:

```
9ec6556c14335722a277790c33835db15291ecc2992c4b0660ae5b112b269a0ea37652e811150af4a5ab5
9d9bdd7a87a65e1ced00537ef376c4e162d1140a89a5ff0f779123ccf1995da7d748f542c0c150db0105435
22d3f70a13ad102085c94a79ca990c8d9ba197ed88211561c536dedb2d659ea874e6df68150c14ec19664
8628fab18a860cd826ee1e0891dfe43def08044354cf9aa9ee27cf90253180cd11e6a4235c24f9f6e296a54
5317051f1794502392c0d3459b228da89986be889d2eb379d50b5164451fc594fc7ea0bb2fec008abaf99f
a7313ba886a12a91caf4ca3f27b61494e095fb97cf128102cc64218488efc3eb162bfbbe474875502246fc6
d71473fa7b8d01fa6b186d7629f054f84376a57aa8e11d54882a3e3b563eac4fd45c7a1895576b4e6be92e
2dc6cdb3e6a85b171f9358268c07aa7a3749fce8cd40aef826d2979fed1364d6448d83a0f54bbe15db6011
8641ba556afd249acb69bdffb80b7291479f80dc50882423da9f5375cc038028bb5919b25769ad650c259b
3382359224223cbbc3598e20a8aadc581b7d0405bef8219fbf376633d9f046782190043a141af67f0f0e502
9e12e5ab49638e6a00ba488381b83e512b080ee3ed692ae355e2d7da47426af77779b694d4ad69db6093
d083fd05f2fb2dca7f3b4397adc4a69ffbe0bdecbfd1ee1a569d47b442fbadb9ca8169eaabb64d9831db524
6eaf454ae2f82dfecd24179db5b118fa310e4a8621911b5ec6eb9f58dd95a7778af6343ad0ab6180161c0b
14b42b234a99547994681d58354f8d42e43d93652d4d10002e1b4684d74287026ea9a2af48dfbbc34f8b2
4270bc5536acd9fa520222fd5ee741dfab3c1e22a752e1654202920e2a7e381a6a2cbfccbf646eb40b6b014
02ea4878d04f4cc7e0485605826dddc22b844679e51bc3752b2bb8673c19c492309a5c8fc587793fc3dad6
be20aff1d5f91d56b021f5bfe59aea90f5948978258191037d87857393b8fd39e325d2eb4d6671759c093b
a3d66d98d219cfc49ed6121e3e259eebe727e965bceed545ece66aa51558c2dd9ec9384d9bdc69e9b0297
217d4399d2a4130ed5c3a2ddab68723c2eb7a430efdd37c2a2eda8697737be90f77de373901858f2dd58f
7839a2bc155f0aa6aaf540a8fed0f37507120fbdc12c890e0a30846f5c629a22af4db65024d630dfa5a2fa42
04d818d0bd431d770c196d68ca43e3f9062ff897c4fea7b30fb7cfec18fd29a49539da17333758a3297c940
439dc07945825cca16e23cd4931f8775f1c0974d82fc58f33521b30e81ef40941201daabd35017999f5a301
70b1195911201daabd3501799721dad956ba4f4e71201daabd35017995c4ddaceb93364c51201daabd35
017994ec9b6469812c0701201daabd350179921192b134c115b4458bc5878345b6d34d0b7edf4b02d892
7fa36e8df1dd7ece01b4cf954573ba09a165e3fdee9128f3781ef30b99312a9d17ea1826f2e4d2f947a9e61
243f27806d7dc75ddb8c82a9abb5f6b6733e26461f4de559d114405776eb79382856f396f13130b87f478c
8794959a1496ed8f0696
```

Begin code:

```
'''

Homework Number: 2

Name: Alex Goebel

ECN Login: goebel2

Due Date: 2/4/2021

'''


#!/usr/bin/env python3
```

```
### hw2_starter.py

#All code besides main was copied from the lecture notes, some modifications were made but mostly it is the exact same as the notes


import sys

from BitVector import *



expansion_permutation = [31, 0, 1, 2, 3, 4, 3, 4, 5, 6, 7, 8, 7, 8, 9, 10, 11, 12, 11, 12, 13, 14, 15, 16, 15, 16, 17, 18, 19, 20, 19, 20, 21, 22, 23, 24, 23, 24, 25, 26, 27, 28, 27, 28, 29, 30, 31, 0]

pBoxPerm = [15,6,19,20,28,11,27,16,0,14,22,25,4,17,30,9,1,7,23,13,31,26,2,8,18,12,29,5,21,10,3,24]


def encrypt(inFile, encryptionKey):

    key = get_encryption_key(encryptionKey)

    round_key = generate_round_keys(key)

    bv = BitVector(filename = inFile)

    finalString = BitVector(size = 0)

    while (bv.more_to_read):

        bitvec = bv.read_bits_from_file(64)

        if len(bitvec) < 64:

            temp = BitVector(intVal = 0, size = 64-len(bitvec))

            bitvec = bitvec + temp

        if bitvec.size > 0:

            for i in range(0,16):

                [LE, RE] = bitvec.divide_into_two()

                newRE = RE.permute(expansion_permutation)

                out_xor = newRE ^ round_key[i]

                RE_modified = substitute(out_xor)
                rightHalf = RE_modified.permute(pBoxPerm)
                newRE = rightHalf ^ LE
                bitvec = RE + newRE
            finalString = finalString + RE + LE
    return finalString


def decrypt(inFile, encryptionKey):
    key = get_encryption_key(encryptionKey)
    round_key = generate_round_keys(key)[::-1]
    inputFile = open(inFile)
    bv = BitVector(hexstring = inputFile.read())
    finalString = BitVector(size = 0)
    count = 0
    while (count < bv.size):
        bitvec = bv[count:count+63]
        if len(bitvec) <= 64:
            temp = BitVector(intVal = 0, size = 64-len(bitvec))
            bitvec = bitvec + temp
        if bitvec.size <= count:
            for i in range(0,16):
                [LE, RE] = bitvec.divide_into_two()
                newRE = RE.permute(expansion_permutation)
                out_xor = newRE ^ round_key[i]
                RE_modified = substitute(out_xor)
                rightHalf = RE_modified.permute(pBoxPerm)
                newRE = rightHalf ^ LE
                bitvec = RE + newRE
            finalString = finalString + RE + LE
        count += 64
    final = finalString.get_bitvector_in_ascii()
    print(final)
    return finalString



key_permutation_1 = [56,48,40,32,24,16,8,0,57,49,41,33,25,17,
                      9,1,58,50,42,34,26,18,10,2,59,51,43,35,
                     62,54,46,38,30,22,14,6,61,53,45,37,29,21,
                     13,5,60,52,44,36,28,20,12,4,27,19,11,3]

def get_encryption_key(encryptionKey):
    key = encryptionKey
    key = BitVector(textstring = key)
    key = key.permute(key_permutation_1)
    return key




key_permutation_2 = [13,16,10,23,0,4,2,27,14,5,20,9,22,18,11,
                      3,25,7,15,6,26,19,12,1,40,51,30,36,46,
                     54,29,39,50,44,32,47,43,48,38,55,33,52,
                     45,41,49,35,28,31]

shifts_for_round_key_gen = [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]



def generate_round_keys(encryption_key):
    round_keys = []
    key = encryption_key.deep_copy()
    for round_count in range(16):
        [LKey, RKey] = key.divide_into_two()
        shift = shifts_for_round_key_gen[round_count]
        LKey << shift
        RKey << shift
        key = LKey + RKey
        round_key = key.permute(key_permutation_2)
        round_keys.append(round_key)
    return round_keys



s_boxes = {i:None for i in range(8)}

s_boxes[0] = [ [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] ]

s_boxes[1] = [ [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] ]

s_boxes[2] = [ [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] ]
```

```
s_boxes[3] = [ [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] ]

s_boxes[4] = [ [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] ]

s_boxes[5] = [ [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] ]

s_boxes[6] = [ [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] ]

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

def substitute( expanded_half_block ):
    '''
    This method implements the step "Substitution with 8 S-boxes" step you see inside
    Feistel Function dotted box in Figure 4 of Lecture 3 notes.
    '''
    output = BitVector (size = 32)
    segments = [expanded_half_block[x*6:x*6+6] for x in range(8)]
    for sindex in range(len(segments)):
        row = 2*segments[sindex][0] + segments[sindex][-1]
        column = int(segments[sindex][1:-1])
        output[sindex*4:sindex*4+4] = BitVector(intVal = s_boxes[sindex][row][column], size = 4)
    return output




if __name__ == '__main__':
    if len(sys.argv) != 5:
        sys.exit('Incorrect number of arguments, please try again')
    if sys.argv[1] == '-e':
        inFileName = sys.argv[2]
        encryptionKeyFile = sys.argv[3]
        encryptionKeyFile = open(encryptionKeyFile)
        encryptionKey = encryptionKeyFile.read()
        output = encrypt(inFileName, encryptionKey)
        outFile = sys.argv[4]
        outFile = open(outFile, 'w')
        outFile.write(output.get_hex_string_from_bitvector())
        outFile.close()
        encryptionKeyFile.close()
    if sys.argv[1] == '-d':
        inFileName = sys.argv[2]
        encryptionKeyFile = sys.argv[3]
        encryptionKeyFile = open(encryptionKeyFile)
        encryptionKey = encryptionKeyFile.read()

        output = decrypt(inFileName, encryptionKey)

        outFile = sys.argv[4]

        outFile = open(outFile, 'w')

        #output.write_to_file(outFile)

        output1 = str(output)

        outFile.write(output1)

        #outFile.write(output.get_ascii_from_bitvector())

        outFile.close()

        encryptionKeyFile.close()
```

End code

Much of my code was used from the lecture notes. I was unable to get decrypt working fully. The final string that is returned does not seem to be correct and I was unable to find the source of the error. However, encryption seemed to work fine. I started in main by checking to make sure there was the correct amount of arguments and then checked if the user was trying to encrypt or decrypt. For decryption, I started with getting the message to encrypt and the encryption key, both of which I sent into the encrypt function. The encrypt function then called the needed functions for the keys, checked to see if padding was needing, and carried out the substitutions and XORing required for the encryption. It then performed the final switch after all 16 rounds were completed, put the two halves back together for the final string and returned it to the main function. My main function then wrote the encrypted text into an output file in hexstring format.

Problem 2: I unfortunately was unable to get to this problem.