TASKS

1:

```
def xorStrings(str1, str2):
    if len(str1) != len(str2):
        print("ERROR: strings are not of equal length")
        return

    newStr = ""
    for i in range(len(str1)):
        newStr += chr(ord(str1[i]) ^ ord(str2[i]))

    return newStr
```

This task was pretty simple; check if len is same, iterate over each char and xor them together

2:

```
def generateRandomKey(length):
    return ''.join([chr(random.randint(0, 255)) for _ in range(length)])

def encryptFile(filepath):
    try:
        plainTextFile = open(filepath)
    except IOError:
        print("Could not find inputfile " + filepath)

    plainText = plainTextFile.read()
    randomKey = generateRandomKey(len(plainText))
    cipherText = xorStrings(plainText, randomKey)

    with open("key", "w+") as key_file:
        key_file.write(randomKey)

    with open("cipherTxt", "w+") as cipher_file:
        cipher_file.write(cipherText)
```

Read the file you want to encrypt
Generate a random key
xor the contents of the file with the random key
Write the result and key to new files

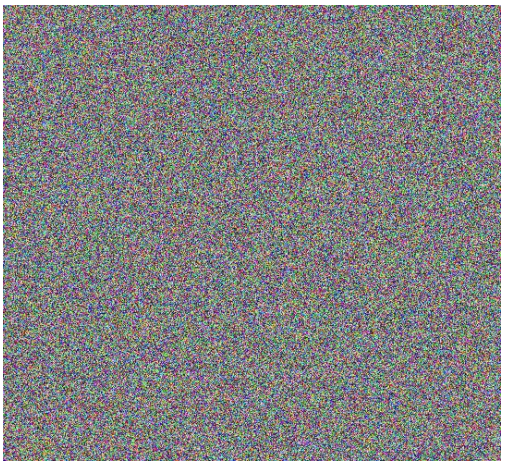
3:

```
def encryptBMP(filepath):
    with open(filepath, mode='r') as file:
        fileContent = file.read()

    # header is 34 bytes long, need to preserve them to see the encoded image
    bmpHeader = fileContent[0:34]
```

```
    randomKey = generateRandomKey(len(fileContent))
    cipherText = xorStrings(fileContent, randomKey)
    cipherText = bmpHeader + cipherText[34:]

    with open("bmpKey", "w+") as key_file:
        key_file.write(randomKey)

    with open("cipherImg.bmp", "w+") as cipher_file:
        cipher_file.write(cipherText)

    return (randomKey, cipherText, fileContent)

def decryptBMP(imgFilepath, keyFilepath):
    with open(imgFilepath, mode='r') as file:
        bmpImg = file.read()

    with open(keyFilepath, mode='r') as file:
        bmpKey = file.read()

    # header is 34 bytes long, need to preserve them to see the encoded image
    bmpHeader = bmpImg[0:34]
    originalFile = xorStrings(bmpImg, bmpKey)
    originalFile = bmpHeader + originalFile[34:]

    with open('original.bmp', 'w+') as file:
        file.write(originalFile)

    return originalFile
```

Encoder: basically same thing as task 2 except conserving the first 34 bytes of data to allow a computer to display the resulting image.
Decoder: use the key to decode a bmp image

Encoded Image:



Decoded image w/ key:

```
mustangEncrypted = encryptBMP('mustang.bmp')
mustangDecrypted = decryptBMP('cipherImg.bmp', 'bmpKey')

print(mustangEncrypted[2] == mustangDecrypted)
```

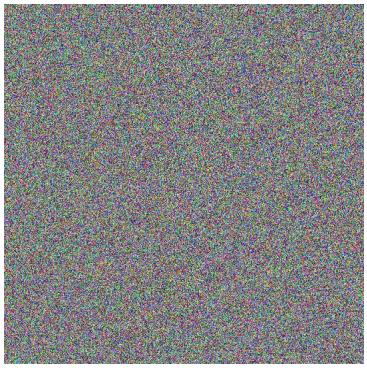
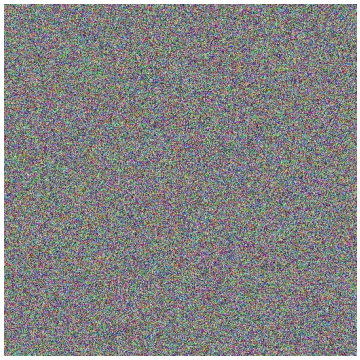
Verified working by comparing the output of decryptBMP with the input of encryptBMP

Task 4:

```
def twoTimeEncrypt(file1, file2):
    with open(file1, mode='r') as file:
        bmpImg1 = file.read()

    with open(file2, mode='r') as file:
        bmpImg2 = file.read()

    if len(bmpImg1) != len(bmpImg2):
        print("ERROR: images are not same size")
        return

    randomKey = generateRandomKey(len(bmpImg1))
    header1 = bmpImg1[0:34]
    header2 = bmpImg2[0:34]

    cipher1 = xorStrings(bmpImg1, randomKey)
    cipher2 = xorStrings(bmpImg2, randomKey)
    cipher3 = xorStrings(cipher1, cipher2)

    cipher1 = header1 + cipher1[34:]
    cipher2 = header2 + cipher2[34:]
    cipher3 = header1 + cipher3[34:]

    with open("bmpKey", "w+") as key_file:
        key_file.write(randomKey)

    with open("cipherImg1.bmp", "w+") as cipher_file:
        cipher_file.write(cipher1)

    with open("cipherImg2.bmp", "w+") as cipher_file:
        cipher_file.write(cipher2)

    with open("cipherImg3.bmp", "w+") as cipher_file:
        cipher_file.write(cipher3)
```

Almost the same as task3, except with 2 input files and outputting the xor of the 2 encrypted images in addition to the encrypted images

cipher1: cipher2: cipher1 xor cipher2


QUESTIONS

1. OTP is a form of encryption that, when used correctly, ensures perfect security, even with unlimited computing power. These are the conditions that must be met to be used correctly:
    - 1.a Key must be perfectly random
    - 1.b Key must be as long as the plaintext
    - 1.c Key must never be reused and destroyed upon use
    - 1.d Key must be transferred securely from the sender to the receiver separately from message
2. The images are indiscernible from random pixels. One interesting observation that I had was that in my first version of the code, I was choosing keys from only alphanumeric sections of the ascii table (34-190 ish). Interestingly enough, with those characters only, you could still see the outlines of the original image and relics of its original colors. This must be because with those values there are fewer 1's in each byte, thus resulting in more of the original values to bleed through when xor'd. This was a reminder that you need truly random keys
3. The xor of the two resulting images looks like an inverted combination of the two images. This is because they had the same key, so when you combine the two images, it essentially undoes the original encoding. (A ^ X) ^ (B ^ X) = (A ^ B)