

[100 Points] Modify the Caesar Shift Cipher Python Implementation Listed Such That Takes the Script Takes as Input a Binary File and Encrypts/Decrypts the Output to Another Binary File

```
1 message = 'This is my secret message'
2 key = 11
3 mode = 'encrypt'
4 SYMBOLS='ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnopqrstuvwxyz1234567890!?. '
5 translated=''
6 for symbol in message:
7     if symbol in SYMBOLS:
8         symbolIndex=SYMBOLS.find(symbol)
9         if mode == 'encrypt':
10             translatedIndex=(symbolIndex+key)%len(SYMBOLS)
11         elif mode == 'decrypt':
12             translatedIndex=(symbolIndex-key)%len(SYMBOLS)
13         translated+=SYMBOLS[translatedIndex]
14     else:
15         translated+=symbol
16 print(translated)
```

Sweigart, A.(2018). *Cracking Odes with Python*. San Francisco, CA: No Starch Press

Below is the Python script I wrote for the caesar cipher, I utilized "Affine_Cipher.py" as the test input to verify functionality.

```
1 print("Homework 1: byte-wise caesar cypher")
2
3 def Caesar_Bytewise(shift, inFile, outFile):
4     with open(inFile, "rb") as f:
5         with open(outFile, "wb") as output:
6             byte = f.read(1)
7             while byte != b"":
8                 # Convert byte into an integer value
9                 value = int.from_bytes(byte, byteorder='big')
10                # Calculate the output value given the shift
11                out_value = (value + shift) % 256
12                # Write the output value to the output file, as a byte
13                output.write(out_value.to_bytes(1, byteorder='big', signed=False))
14                byte = f.read(1)
15            output.close()
16        f.close()
17
18 def main():
19     Caesar_Bytewise(100, "Affine_Cipher.py", "HW1_output.txt")
20
21     # Used for testing:
22     #Caesar_Bytewise(156, "HW1_output.txt", "HW1_outputVerification.txt")
23
24 if __name__ == "__main__":
25     main()
```

Modified function for byte-wise caesar cipher, written to be portable to larger python script

[5 Points] Represent in binary using Two's complement (signed) - Calculators, Internet allowed

Converting to Two's complement:

If positive: $2's\ Complement(x) = x$; if negative: $2's\ Complement(x) = x \oplus 0xFF + 1$

a) 127:

$$127 = \boxed{0111\ 1111}$$

b) -128:

$$128 = 10000000 \rightarrow 01111111 + 1 = \boxed{10000000}$$

c) -1:

$$1 = 00000001 \rightarrow 11111110 + 1 = \boxed{11111111}$$

[5 Points] Represent the following decimal numbers in hexadecimal (unsigned) - Calculators, Internet allowed

a) 10:

$$10 = \boxed{0xA}$$

b) 33:

$$33 = 16 \cdot 2 + 1 = \boxed{0x21}$$

c) 120:

$$120 = 16 \cdot 7 + 8 = \boxed{0x78}$$

[5 Points] Find two different words ($plainText_1$, $plainText_2$) that are at least 4 characters long such that the cipher-texts produced using the Caesar Cipher algorithm are identical ($cipherText_1 = cipherText_2$) provided that they use two different keys (key_1 , key_2). List the two plain-text words ($plainText_1$, $plainText_2$), and their corresponding keys (key_1 , and key_2).

The below pairing was found by using a brute force approach with two online Caesar Cipher algorithms. For the first, I chose a four letter word at random, and shifted it by some key x . This produced $plainText_1 \rightarrow cipherText_1$ with $key = x$. I then used $cipherText_1$ as the input to the second Caesar Cipher, and checked each key to see if it produced a corresponding English word, let this key be x' . This would indicate that $key_2 = 26 - x'$.

The words, cipher texts, and keys are as follows:

Plain Text	Cipher Text	Key
cows	htbx	5
semi	htbx	15

[10 Points] Fill in the blank (all from the book):

- a) Symmetric Encryption Participants share a secret key to encrypt and decrypt and decryption is the inverse operation of encryption.
- b) Asymmetric/Public Key Encryption Participants share a public portion of the key, but a private portion is retained by a single participant for secure encryption and decryption and digital signatures.
- c) Cryptography The science of secret writing.
- d) Cryptanalysis The science and art of breaking the secure properties of crypto systems.
- e) Moore's Law The concept that computational power doubles every eighteen months so that brute force attacks against cryptosystems will be faster in the future.
- f) Kerckhoffs' Principle The concept that a cryptosystem should be secure even if the attacker knows all the details (but not the keys) of the targeted cryptosystem.

1.9 [25 Points] Compute x as far as possible without a calculator. Where appropriate, make use of a smart decomposition of the exponent as shown in the example in Sect. 1.4.1:

a) $x = 3^2 \mod 13$

$x = 9$

b) $x = 7^2 \mod 13$

$$x = 49 = 13 \cdot 3 + 10 \rightarrow \boxed{x = 10}$$

c) $x = 7^{100} \mod 13$

$$x = 7^{100} \mod 13 = (7^2)^{50} \mod 13 = 10^{50} \mod 13$$

$$10^2 \mod 13 = 9$$

$$10^3 \mod 13 = 10^2 \cdot 10 \mod 13$$

$$10^3 \mod 13 = 90 \mod 13 = 12$$

$$x = 10^{50} \mod 13 = (10^2 \cdot 10^3)^{10} \mod 13$$

$$x = (9 \cdot 12)^{10} \mod 13$$

$$9 \cdot 12 \mod 13 = 108 \mod 13 = 4$$

$$x = 4^{10} \mod 13 = (4^2)^5 \mod 13$$

$$4^2 \mod 13 = 3$$

$$x = 3^5 \mod 13 = 3^2 \cdot 3^3 \mod 13$$

$$3^3 \mod 13 = 27 \mod 13 = 1$$

$$x = 9 \cdot 1 \mod 13$$

$$x = 7^{100} \mod 13 = \boxed{9 = x}$$

- d) This last problem is called a *discrete logarithm* and points to a hard problem which we discuss in Chap. 8. The security of many public-key schemes is based on the hardness of solving the discrete logarithm for large numbers, e.g., with more than 1000 bits.

$$7 \times x = 11 \pmod{13}$$

$$7 \cdot x = 11 \pmod{13}$$

$$7 \cdot 9 = 63$$

$$63 \pmod{13} = 11$$

$x = 9$

1.11 [25 Points] This problem deals with the affine cipher with the key parameters $\alpha = 7$, $\beta = 22$.

I utilized a python script to solve this problem, and the code I wrote is included in Appendix A.

- a) Decrypt the following text: falszztysyzyjkywjrztjztyynaryjkyswarztyegyyj

The decrypted text is as follows: **firstthesentenceandthentheevidencesaidthequeen**

- b) Who wrote the line?

Lewis Carroll in *Alice's Adventures Under Ground*

1.13 [25 Points] In an attack scenario, we assume that the attacker Oscar manages somehow to provide Alice with a few pieces of plaintext that she encrypts. Show how Oscar can break the affine cipher by using two pairs of plaintext-ciphertext, (x_1, y_1) and (x_2, y_2) . What is the condition for choosing x_1 and x_2 ?

Remark: In practice, such an assumption turns out to be valid for certain settings, e.g., encryption by Web servers, etc. This attack scenario is, thus, very important and is denoted as a *chosen plaintext* attack.

One must select x_1 and x_2 such that $y_1 \neq y_2$. This would then give you the following equations:

$$\alpha \cdot x_1 + \beta \pmod{26} = y_1 \tag{1}$$

$$\alpha \cdot x_2 + \beta \pmod{26} = y_2 \tag{2}$$

With (1) and (2), Oscar has two equations with two unknowns: α and β . Thus, this system of equations can be solved, giving Oscar the keys α and β .

Appendix A: Python Script

```
import string
import math

def ModularInverse(alpha):
    running = True
    inv_alpha = 1
    while running:
        if ( (alpha * inv_alpha) % 26 == 1):
            print( "Modular inverse of {0} is: {1}".format(alpha, inv_alpha) )
            running = False
            return inv_alpha
        else:
            inv_alpha += 1

def affineDecrypt(inv_alpha, beta, input):
    output = ""
    for letter in input:
        in_index = string.ascii_lowercase.index(letter)
        out_index = ( inv_alpha * (in_index - beta) ) % 26
        output += string.ascii_lowercase[out_index]
    return output

def affineEncrypt(alpha, beta, input):
    output = ""
    for letter in input:
        in_index = string.ascii_lowercase.index(letter)
        out_index = ( in_index * alpha + beta ) % 26
        output += string.ascii_lowercase[out_index]
    return output

def main():
    encrypt = True
    alpha = 7
    beta = 22
    if encrypt:
        print("Starting encrypt with alpha = {0} and beta = {1}: \n".format(alpha, beta))
        input = "firstthesentenceandthentheevidencesaidthequeen"
        output = affineEncrypt(alpha, beta, input)
    else:
        print("Starting decrypt with alpha = {0} and beta = {1}: \n".format(alpha, beta))
        input = "falszztysyjzyjkywjrztyjztyynaryjkyswarztyegyij"
        output = affineDecrypt(ModularInverse(alpha), beta, input)

    print( "The output is: \n" + output )

if __name__ == "__main__":
    main()
```