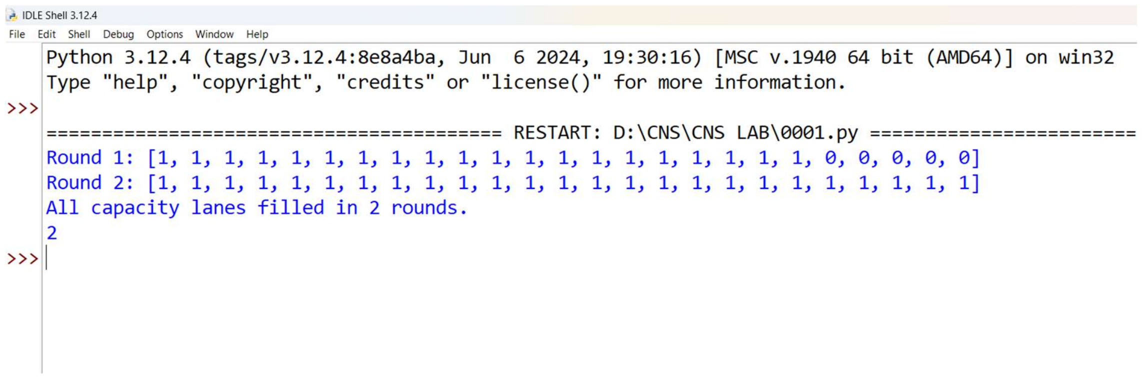
1. Write a program for SHA-3 option with a block size of 1024 bits and assume that each of the lanes in the first message block (P0) has at least one nonzero bit. To start, all of the lanes in the internal state matrix that correspond to the capacity portion of the initial state are all zeros. Show how long it will take before all of these lanes have at least one nonzero bit. Note: Ignore the permutation. That is, keep track of the original zero lanes even after they have changed position in the matrix

Code:

```
def capfill():
    s = [1] * 10 + [0] * 15
    r = 0


    while 0 in s[10:]:
        r += 1
        s = [1] * 10 + s[:15]
        print(f"Round {r}: {s}")


    print(f"All capacity lanes filled in {r} rounds.")
    return r


print(capfill())
```

Op:



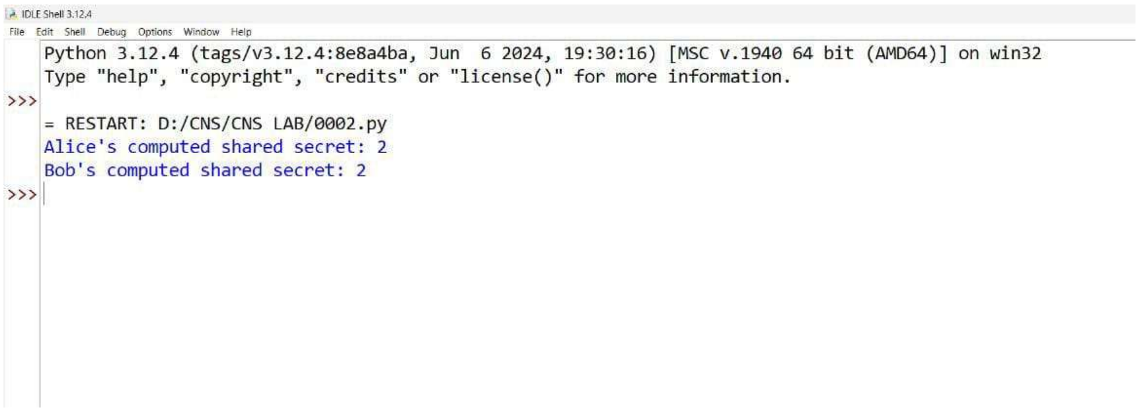
```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========================================= RESTART: D:\CNS\CNS LAB\0001.py ========================
Round 1: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
Round 2: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
All capacity lanes filled in 2 rounds.
2
>>>
```

2. Write a program for Diffie-Hellman protocol, each participant selects a secret number x and sends the other participant ax mod q for some public number a. What would happen if the participants sent each other xa for some public number a instead? Give at least one method Alice and Bob could use to agree on a key. Can Eve break your system without finding the secret numbers? Can Eve find the secret numbers?

Code:

```
def diffie_hellman(a, q, x_A, x_B):

    A = pow(a, x_A, q)

    B = pow(a, x_B, q)

    K_A = pow(B, x_A, q)

    K_B = pow(A, x_B, q)

    print("Alice's computed shared secret:", K_A)

    print("Bob's computed shared secret:", K_B)

    return K_A, K_B

a = 5

q = 23

x_A = 6

x_B = 15

K_A, K_B = diffie_hellman(a, q, x_A, x_B)
```

Op:



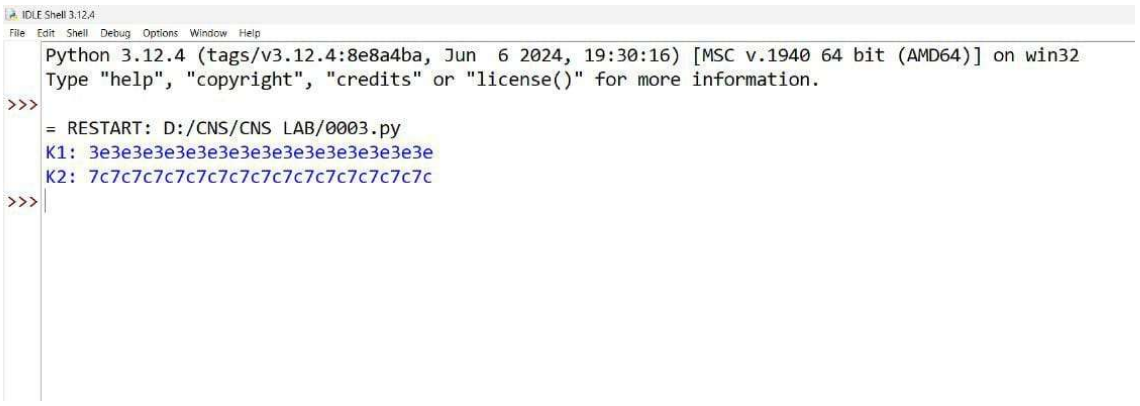
```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: D:/CNS/CNS LAB/0002.py
Alice's computed shared secret: 2
Bob's computed shared secret: 2
>>>
```

3. Write a program for subkey generation in CMAC, it states that the block cipher is applied to the block that consists entirely of 0 bits. The first subkey is derived from the resulting string by a left shift of one bit and, conditionally, by XORing a constant that depends on the block size. The second subkey is derived in the same manner from the first subkey. a. What constants are needed for block sizes of 64 and 128 bits? b. How the left shift and XOR accomplishes the desired result.

Code:

```
def aes(input_blk):
    return bytes([input_blk[i] ^ 0x1F for i in range(len(input_blk))])


def shift_xor(b, c, blk_size=128):
    s = (int.from_bytes(b, 'big') << 1) & ((1 << blk_size) - 1)
    if b[0] & 0x80:
        s ^= c
    return s.to_bytes(blk_size // 8, 'big')


def cmac_keys(k, blk_size=128):
    z_blk = bytes([0] * (blk_size // 8))
    r = aes(z_blk)
    R = 0x87 if blk_size == 128 else 0x1b
    k1 = shift_xor(r, R, blk_size)
    k2 = shift_xor(k1, R, blk_size)
    return k1, k2


key = bytes([0x2b, 0x7e, 0x15, 0x16, 0x28, 0xae, 0xd2, 0xa6, 0xab, 0xf7, 0x97, 0x75, 0x46,
0x0f, 0x92, 0x15])
k1, k2 = cmac_keys(key, 128)


print("K1:", k1.hex())
print("K2:", k2.hex())
```

op:



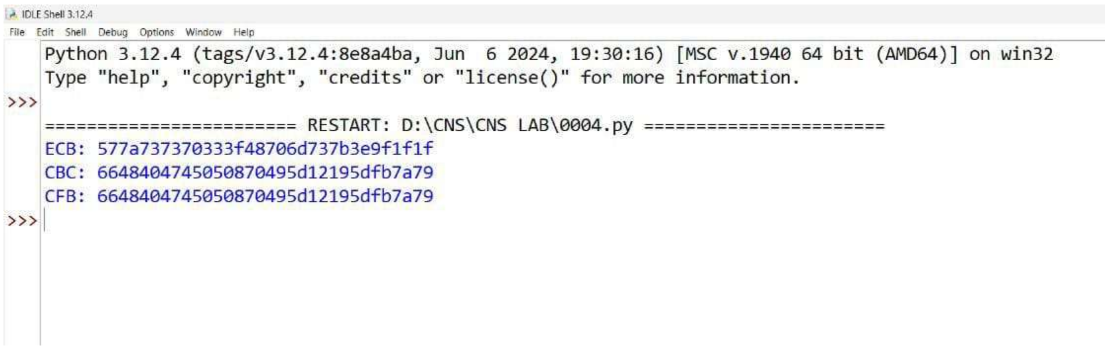
```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: D:/CNS/CNS LAB/0003.py
K1: 3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e3e
K2: 7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c7c
>>>
```

4. Write a program for ECB, CBC, and CFB modes, the plaintext must be a sequence of one or more complete data blocks (or, for CFB mode, data segments). In other words, for these three modes, the total number of bits in the plaintext must be a positive multiple of the block (or segment) size. One common method of padding, if needed, consists of a 1 bit followed by as few zero bits, possibly none, as are necessary to complete the final block. It is considered good practice for the sender to pad every message, including messages in which the final message block is already complete. What is the motivation for including a padding block when padding is not needed

Code:

```
def xorBlocks(b1, b2):
    return bytes([x ^ y for x, y in zip(b1, b2)])


def aes(b):
    return bytes([x ^ 0x1F for x in b])


def pad(p, size):
    return p + bytes([0x80] + [0] * (size - len(p) % size - 1))


def encryptEcb(p, k, size=16):
    return b''.join([aes(p[i:i + size]) for i in range(0, len(p), size)])


def decryptEcb(c, k, size=16):
    return b''.join([aes(c[i:i + size]) for i in range(0, len(c), size)])


def encryptCbc(p, k, iv, size=16):
    prev = iv
    return b''.join([aes(xorBlocks(p[i:i + size], prev)) if i + size <= len(p) else b'' for i in range(0,
len(p), size)])


def decryptCbc(c, k, iv, size=16):
    prev = iv
    return b''.join([xorBlocks(aes(c[i:i + size]), prev) for i in range(0, len(c), size)])


def encryptCfb(p, k, iv, size=16):
    prev = iv
    return b''.join([xorBlocks(p[i:i + size], aes(prev)) for i in range(0, len(p), size)])


def decryptCfb(c, k, iv, size=16):
    prev = iv
    return b''.join([xorBlocks(c[i:i + size], aes(prev)) for i in range(0, len(c), size)])


key = b"0123456789abcdef"
iv = b"1234567890abcdef"
plaintext = b"Hello, World!"


plaintext = pad(plaintext, 16)


print("ECB:", encryptEcb(plaintext, key).hex())
print("CBC:", encryptCbc(plaintext, key, iv).hex())
print("CFB:", encryptCfb(plaintext, key, iv).hex())
```

op:



```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
======================== RESTART: D:\CNS\CNS LAB\0004.py =======================
ECB: 577a737370333f48706d737b3e9f1f1f
CBC: 6648404745050870495d12195dfb7a79
CFB: 6648404745050870495d12195dfb7a79
>>>
```

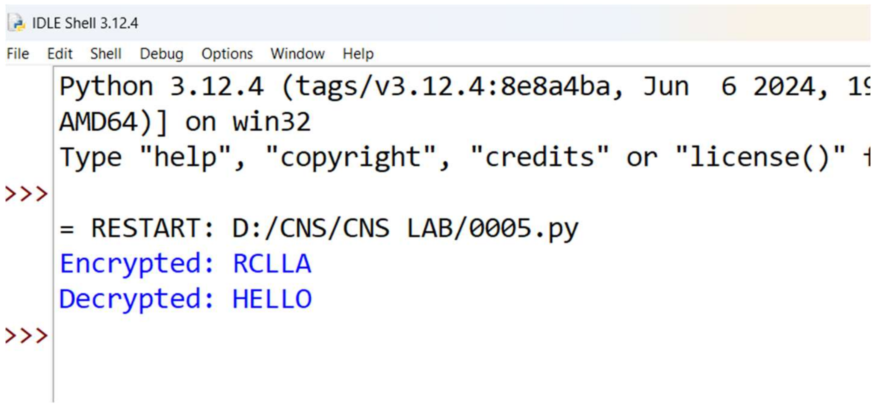
5. Write a program for Caesar cipher, known as the affine Caesar cipher, has the following form: For each plaintext letter p, substitute the ciphertext letter C: C = E([a, b], p) = (ap + b) mod 26 A basic requirement of any encryption algorithm is that it be one-to-one. That is, if p q, then E(k, p) E(k, q). Otherwise, decryption is impossible, because more than one plaintext character maps into the same ciphertext character. The affine Caesar cipher is not one-to-one for all values of a. For example, for a = 2 and b = 3, then E([a, b], 0) = E([a, b], 13) = 3.

Code:

```
import string


ALPHABET = string.ascii_uppercase

ALPHA = len(ALPHABET)


def gcd(a, b):

    while b:

        a, b = b, a % b

    return a


def modinverse(a, m):

    for i in range(1, m):

        if (a * i) % m == 1:

            return i

    return None
```

```
def affineencrypt(text, a, b):
    if gcd(a, ALPHA) != 1:
        raise ValueError("a must be coprime with 26.")
    text = text.upper()
    return "".join(ALPHABET[(a * ALPHABET.index(c) + b) % ALPHA] if c in ALPHABET else c for
c in text)


def affinedecrypt(text, a, b):
    ainv = modinverse(a, ALPHA)
    if ainv is None:
        raise ValueError("a has no modular inverse.")
    text = text.upper()
    return "".join(ALPHABET[(ainv * (ALPHABET.index(c) - b)) % ALPHA] if c in ALPHABET else c
for c in text)



a, b = 5, 8
msg = "HELLO"
enc = affineencrypt(msg, a, b)
dec = affinedecrypt(enc, a, b)

print("Encrypted:", enc)
print("Decrypted:", dec)
```

Op:



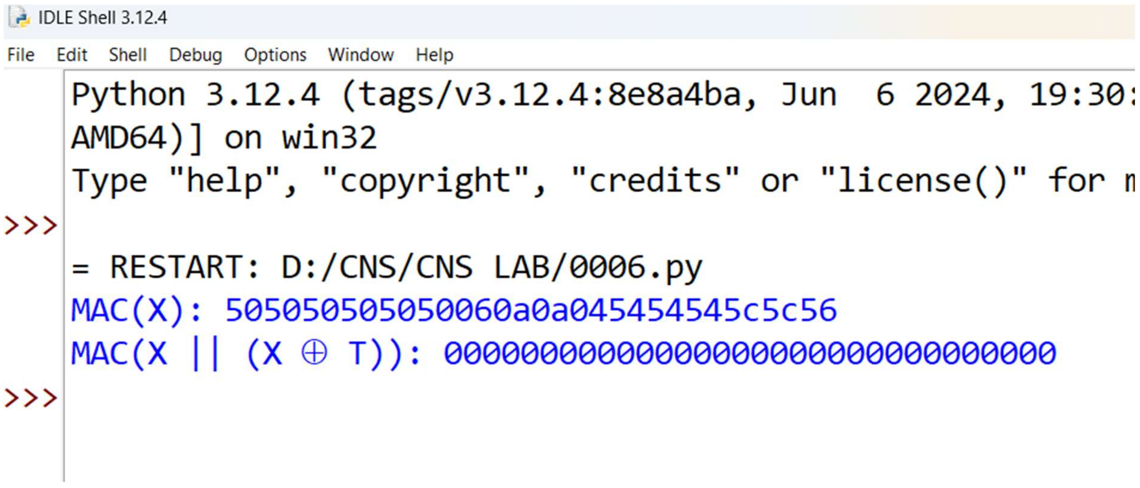
```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 1
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" 
>>>
= RESTART: D:/CNS/CNS LAB/0005.py
Encrypted: RCLLA
Decrypted: HELLO
>>>
```

6. Write a program for CBC MAC of a one block message X, say T = MAC(K, X), the adversary immediately knows the CBC MAC for the two-block message X || (X ⊕ T) since this is once again.

Code:

```
def xor(a, b):
    return bytes(x ^ y for x, y in zip(a, b))


def fake_encrypt(block, key):
    return xor(block, key)


key = b"1234567890abcdef"
X = b"abcdef1234567890"


T = fake_encrypt(X, key)
X2 = xor(X, T)
T2 = fake_encrypt(X2, key)


print("MAC(X):", T.hex())
print("MAC(X || (X ⊕ T)):", T2.hex())
```

Op:



```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for m
>>>
= RESTART: D:/CNS/CNS LAB/0006.py
MAC(X): 5050505050050060a0a045454545c5c56
MAC(X || (X ⊕ T)): 00000000000000000000000000000000
>>>
```