**1. (Encryption and Statistical Analysis) Though encryption is primarily designed to preserve confidentiality and integrity of data, the mechanism itself is vulnerable to brute force (statistical analysis). In other words, the more we see the encrypted data, the easier we can hack it. In this exercise, you are asked to crack the following cipher text. Please provide the decrypted result and explain your strategy in decrypting this text.**

a. Count the frequency of letters. List the top three most frequent characters.

```python
text_to_analyze = "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."

frequency_counter = dict()

def display_top_3_characters(counter):
    for _ in range(3):
        highest_freq_char = max(counter, key=counter.get)
        print(highest_freq_char, counter[highest_freq_char])
        del counter[highest_freq_char]

for char in text_to_analyze:
    if char in frequency_counter:
        frequency_counter[char] += 1
    else:
        frequency_counter[char] = 1

frequency_counter.pop(" ")

print("Three most common characters are:")
display_top_3_characters(frequency_counter)
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity main ● ?
Three most common characters are:
P 7
R 6
O 6
```

b. Knowing that this is English, what are commonly used three-letter words and two-letter words. Does the knowledge give you a hint on cracking the given text?

- **Yes, during my initial decryption attempt, presuming FP stands for 'is' and QDR for 'the' clarified the letter relationships, and make it easier to guess the remaining characters.**

c. Cracking the given text. Measure the time that you have taken to crack this message.

- **Time used ≈ 20 mins.**
  **Ans = “security is the first cause of misfortune. this is an old german proverb.”**

```python
text_to_analyze = (
    "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."
)

translation = {
    "F": "i",
    "P": "s",
    "Q": "t",
    "D": "h",
    "R": "e",
    "Z": "a",
    "K": "n",
    "C": "c",
    "S": "u",
    "O": "r",
    "X": "y",
    "A": "f",
    "L": "o",
    "J": "m",
    "B": "g",
    "I": "l",
    "U": "d",
    "E": "b",
    "M": "p",
    "T": "v",
}

deciphered_text = "".join(
    [translation[char] if char in translation else char for char in text_to_analyze]
)
print(deciphered_text)
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity main ● ?
security is the first cause of misfortune. this is an old german proverb.
```

d. Explain your process in hacking such messages.

- **Start from 2-letter word like “is” then 3-letter word like “the”, after that replace other with the known letter and use the context to guess others.**

e. If you know that the encryption scheme is based on Caesar (Monoalphabetic Substitution) that is commonly used by Caesar for sending messages to Cicero, does it allow you to crack it faster?

- **Yes, because we don’t have to guess the encryption scheme.**

f. Draw a cipher disc of the given text.

```python
text_to_analyze = (
    "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."
)

translation = {
    "F": "i",
    "P": "s",
    "Q": "t",
    "D": "h",
    "R": "e",
    "Z": "a",
    "K": "n",
    "C": "c",
    "S": "u",
    "O": "r",
    "X": "y",
    "A": "f",
    "L": "o",
    "J": "m",
    "B": "g",
    "I": "l",
    "U": "d",
    "E": "b",
    "M": "p",
    "T": "v",
}

sorted_items = list(translation.items())
sorted_items.sort(key=lambda pair: pair[1])
deciphered_chars = []
cipher_chars = []
for cipher, decipher in sorted_items:
    deciphered_chars.append(decipher)
    cipher_chars.append(cipher)
print("".join(deciphered_chars))
print("".join(cipher_chars))
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity main ● ?
abcdefghilmnoprstuvy
ZECURABDFIJKLMOPQSTX
```

g. Create a simple python program for cracking the Caesar cipher text using brute force attack. Explain the design and demonstrate your software. (You may use an English dictionary for validating results.)

```python
import requests
import itertools

english_dict = ["japanese", "hacker", "itsupport", "king"
                , "habit", "money", "security", "is", "the"
                , "first", "cause", "of", "misfortune", "this"
                , "an", "old", "german", "proverb"]

def decrypt(text_to_decrypt, decryption_cipher):
    decrypted_message = ""
    for char in text_to_decrypt:
        if char in " .":
            decrypted_message += char
        else:
            decrypted_message += chr(ord("A") + decryption_cipher.index(char))
    return decrypted_message

def is_message_valid(message):
    words = message.split(" ")
    for word in words:
        if word.strip(".").lower() not in english_dict:
            return False
    return True

alphabet = list("ABCDEFGHIJKLMNOPQRSTUVWXYZ")
is_decryption_found = False
cipher_text = "PRCSOFQX FP QDR AFOPQ CZSPR LA JFPALOQSKR. QDFP FP ZK LIU BROJZK MOLTROE."

for length in range(1, 6):
    print(f"Attempting decryption with cipher of length {length}")
    all_possible_permutations = list(itertools.permutations(alphabet, length))

    # Using permutations in reverse order for optimization
    for current_permutation in reversed(all_possible_permutations):
        current_key = ''.join([character for character in current_permutation])
        current_cipher = current_key + ''.join([char for char in alphabet if char not in current_key])

        decrypted_message = decrypt(cipher_text, current_cipher)
        if is_message_valid(decrypted_message):
            print(f'Decrypted message: {decrypted_message}')
            print(f'Used key: {current_key}')
            is_decryption_found = True
            break

    if is_decryption_found:
        break
```

```
Attempting decryption with cipher of length 1
Attempting decryption with cipher of length 2
Attempting decryption with cipher of length 3
Attempting decryption with cipher of length 4
Attempting decryption with cipher of length 5
Decrypted message: SECURITY IS THE FIRST CAUSE OF MISFORTUNE. THIS IS AN OLD GERMAN PROVERB.
Used key: ZECUR
```

## 2. (Symmetric Encryption) Vigenère is a complex version of the Caesar cipher. It is a polyalphabetic substitution.

a. Explain how it can be used to cipher data.

- **the encoding method is to specify one key known to both the sender and the receiver first. Suppose it is CAT. Next, create the Caesar Disc to identify the correspondence between letters. The procedure starts with the given key. Subsequently, the rest of the English alphabet is used, excluding the letters already used.**

b. If a key is the word “CAT”, please analyze the level of security provided by Vigenère compared to that of the Caesar cipher.

- **if we have the key as CAT, it's observed that we can easily guess the encrypted message. If one can guess the context and vocabulary, then it's possible to guess the key because it maps on a 1-1 basis. However, in the case of Vigenère, even if the key is CAT, when encrypting each letter, it maps to a different character. For example,**

  **HELLO -> JEENO**
  **EHOOL -> GHHQL**

  **This makes it much more challenging to guess compared to the actual raw data.**

c. Create a python program for ciphering data using Vigenère

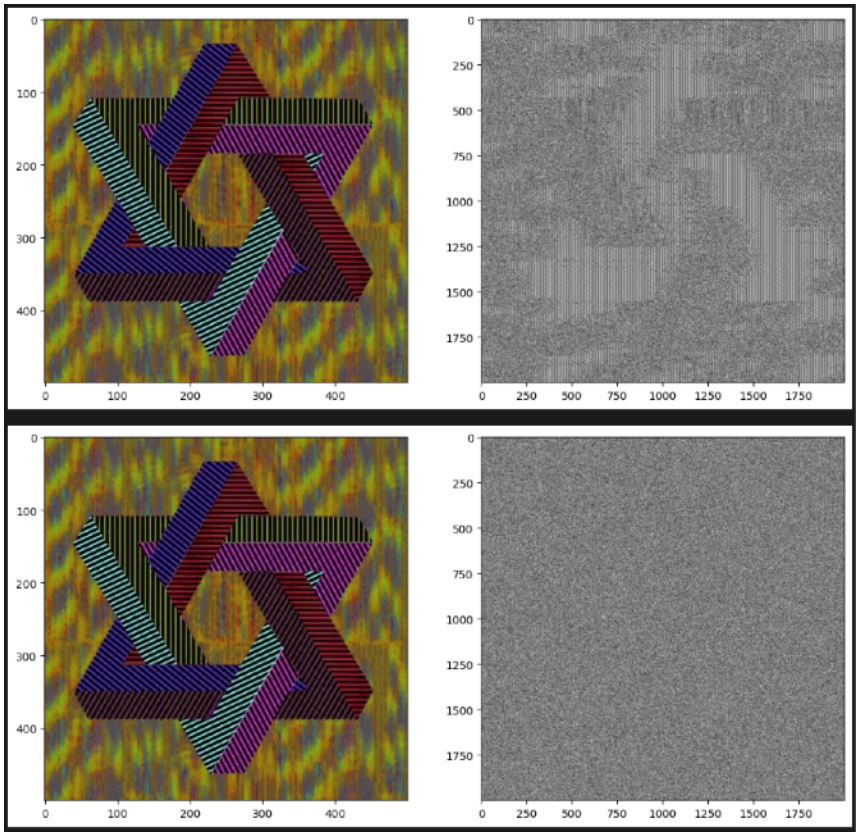
```python
def generate_vigenere_table() -> dict:
    vigenere_table = {}
    for i in range(26):
        vigenere_table[chr(i + 65)] = {}
        for j in range(26):
            vigenere_table[chr(i + 65)][chr(j + 65)] = chr((i + j) % 26 + 65)
    return vigenere_table


def vigenere_cipher(text: str, key: str) -> str:
    vigenere_table = generate_vigenere_table()
    encrypted_text = ""
    for i in range(len(text)):
        encrypted_text += vigenere_table[text[i].upper()][key[i % len(key)]]
    return encrypted_text


print(vigenere_cipher("HELLOWORLD", "CAT"))
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity main ● ?
JEENOPQREF
```

**3. (Mode in Block Cipher) Block Cipher is designed to have more randomness in a block. However, an individual block still utilizes the same key. Thus, it is recommended to use a cipher mode with an initial vector, chaining or feedback between blocks. This exercise will show you the weakness of Electronic Code Book mode which does not include any initial vector, chaining or feedback.**



AES-256-ECB

AES-256-CBC

- **From the images, it can be observed that on the AES-256-ECB side, the image on the right still retains some of its original structure, making it somewhat recognizable but still hard to discern. On the other hand, for the AES-256-CBC side, the image on the right retains very little of its original structure, making it much harder to identify than the AES-256-ECB counterpart. If we compare AES-256-CBC with AES-256-ECB, it is recommended to use AES-256-CBC over AES-256-ECB.**

## 4. (Encryption Protocol - Digital Signature)

a. Measure the performance of a hash function (sha1), RC4, Blowfish and DSA. Outline your experimental design.
(Please use OpenSSL for your measurement)

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity/activity4/3 main ● ? openssl speed sha1
                                7810 22:38:50
Doing sha1 for 3s on 16 size blocks: 14036233 sha1's in 2.65s
Doing sha1 for 3s on 64 size blocks: 9188416 sha1's in 2.59s
Doing sha1 for 3s on 256 size blocks: 4606979 sha1's in 2.78s
Doing sha1 for 3s on 1024 size blocks: 1527055 sha1's in 2.80s
Doing sha1 for 3s on 8192 size blocks: 202261 sha1's in 2.73s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
sha1             84650.12k   227158.33k   423963.30k   559266.98k   606882.49k
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity/activity4/3 main ● ? openssl speed rc4
Doing rc4 for 3s on 16 size blocks: 170382010 rc4's in 2.75s
Doing rc4 for 3s on 64 size blocks: 48228772 rc4's in 2.59s
Doing rc4 for 3s on 256 size blocks: 13252526 rc4's in 2.72s
Doing rc4 for 3s on 1024 size blocks: 3077813 rc4's in 2.53s
Doing rc4 for 3s on 8192 size blocks: 420508 rc4's in 2.71s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
rc4              989659.52k  1189912.06k  1245174.28k  1243640.30k  1270887.02k
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity/activity4/3 main ● ? openssl speed bf
Doing blowfish cbc for 3s on 16 size blocks: 21838785 blowfish cbc's in 2.78s
Doing blowfish cbc for 3s on 64 size blocks: 5313686 blowfish cbc's in 2.62s
Doing blowfish cbc for 3s on 256 size blocks: 1388090 blowfish cbc's in 2.70s
Doing blowfish cbc for 3s on 1024 size blocks: 368257 blowfish cbc's in 2.86s
Doing blowfish cbc for 3s on 8192 size blocks: 45270 blowfish cbc's in 2.81s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
The 'numbers' are in 1000s of bytes per second processed.
type             16 bytes     64 bytes    256 bytes   1024 bytes   8192 bytes
blowfish cbc     125671.05k   129910.83k   131526.30k   132036.49k   131800.84k
```

```
pawankanjeam@Pawans-MacBook-Pro ~/Desktop/class-lecture/2023S12110413-Computer-Security-Activity/activity4/3 main ● ? openssl speed dsa
Doing 512 bit sign dsa's for 10s: 106145 512 bit DSA signs in 8.35s
Doing 512 bit verify dsa's for 10s: 125039 512 bit DSA verify in 8.78s
Doing 1024 bit sign dsa's for 10s: 51710 1024 bit DSA signs in 9.40s
Doing 1024 bit verify dsa's for 10s: 48462 1024 bit DSA verify in 9.05s
Doing 2048 bit sign dsa's for 10s: 16603 2048 bit DSA signs in 9.09s
Doing 2048 bit verify dsa's for 10s: 15526 2048 bit DSA verify in 9.03s
LibreSSL 3.3.6
built on: date not available
options:bn(64,64) rc4(ptr,int) des(idx,cisc,16,int) aes(partial) blowfish(idx)
compiler: information not available
                  sign    verify    sign/s verify/s
dsa  512 bits 0.000079s 0.000070s  12708.3  14234.4
dsa 1024 bits 0.000182s 0.000187s   5501.2   5353.7
dsa 2048 bits 0.000548s 0.000582s   1825.7   1718.9
```

b. Comparing performance and security provided by each method.

- **From the result in a.) , If we measure performance based on throughput, we can see that sha1 and RC4 has the highest performance followed by, Blowfish, and DSA, respectively.**

  **SHA1 is a swift, one-way hash function.**
  **RC4 is a more complex stream cipher with potential predictability.**
  **Blowfish is a time-consuming block cipher that enhances security.**
  **DSA offers the highest security using a dual-key system but is the most complex.**

c. Explain the mechanism underlying Digital Signature. How does it combine the strength and weakness of each encryption scheme?

- **SHA1 is a fast method due to its simplicity but has low security with only a 160-bit size, making it vulnerable to attacks and potential hash collisions.**
- **RC4 is a stream cipher that uses XOR operations for encryption, but it's susceptible to various attacks.**
- **Blowfish, a block cipher, is more complex and secure but takes longer to encrypt data.**
- **DSA, the Digital Signature Algorithm, involves creating a pair of keys: public and private. It digests and signs data using these keys, achieving performance speeds similar to SHA1.**