NAME:Mohammed Hashim Bin Yahya

ROLL NO. 20221429

COURSE: BSC(H) Computer Science

Q1

def encrypt(plaintext, shift):

encrypted\_text = ""

for char in plaintext:

if char.isalpha():

shift\_base = 65 if char.isupper() else 97

encrypted\_char = chr((ord(char) - shift\_base + shift) % 26 + shift\_base)

encrypted\_text += encrypted\_char

else:

encrypted\_text += char

return encrypted\_text

def decrypt(ciphertext, shift):

decrypted\_text = ""

for char in ciphertext:

if char.isalpha(): # Check if the character is a letter

shift\_base = 65 if char.isupper() else 97

decrypted\_char = chr((ord(char) - shift\_base - shift) % 26 + shift\_base)

decrypted\_text += decrypted\_char

else:

decrypted\_text += char # Non-alphabetic characters remain unchanged

return decrypted\_text

def main():

plaintext = input("Enter the text to encrypt: ")

shift = int(input("Enter the shift value: "))

encrypted\_text = encrypt(plaintext, shift)

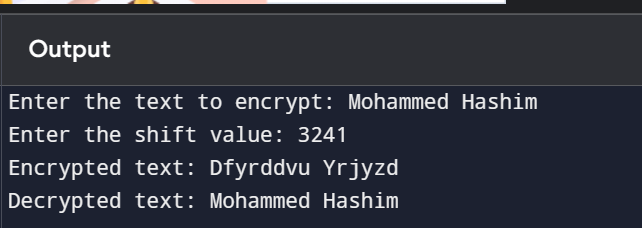
print(f"Encrypted text: {encrypted\_text}")

decrypted\_text = decrypt(encrypted\_text, shift)

print(f"Decrypted text: {decrypted\_text}")

if \_\_name\_\_ == "\_\_main\_\_":

main()



**Time Complexity**: The time complexity of both encryption and decryption is **O(n)**, where n is the length of the input string.

* This is because each function processes every character of the input text exactly once.

**Space Complexity**: The space complexity is **O(n)** as well, since new strings (with the same length as the input) are created to store the encrypted and decrypted texts

Q2

def rail\_fence\_encrypt(text, key):

"""

Encrypts the given text using Rail Fence Cipher.

Parameters:

text (str): The plaintext to encrypt.

key (int): Number of rails (rows) to use.

Returns:

str: Encrypted text (ciphertext).

"""

# Create a 2D list to represent the zigzag pattern

rail = [['' for \_ in range(len(text))] for \_ in range(key)]

direction\_down = False # Flag to track direction

row, col = 0, 0 # Start from the top rail

# Place each character in the zigzag pattern

for char in text:

rail[row][col] = char

col += 1

# Change direction if we reach the top or bottom rail

if row == 0 or row == key - 1:

direction\_down = not direction\_down

# Move to the next rail in the current direction

row += 1 if direction\_down else -1

# Read the characters row by row to form the ciphertext

encrypted\_text = ''.join(''.join(rail[row]) for row in range(key))

return encrypted\_text

def rail\_fence\_decrypt(ciphertext, key):

"""

Decrypts the given text using Rail Fence Cipher.

Parameters:

ciphertext (str): The ciphertext to decrypt.

key (int): Number of rails (rows) to use.

Returns:

str: Decrypted text (plaintext).

"""

# Create a 2D list to represent the zigzag pattern

rail = [['' for \_ in range(len(ciphertext))] for \_ in range(key)]

direction\_down = False

row, col = 0, 0

# Mark the zigzag positions with placeholders

for i in range(len(ciphertext)):

rail[row][col] = '\*'

col += 1

if row == 0 or row == key - 1:

direction\_down = not direction\_down

row += 1 if direction\_down else -1

# Fill the marked positions with the ciphertext characters

index = 0

for i in range(key):

for j in range(len(ciphertext)):

if rail[i][j] == '\*' and index < len(ciphertext):

rail[i][j] = ciphertext[index]

index += 1

# Read the zigzag pattern to reconstruct the plaintext

decrypted\_text = []

row, col = 0, 0

direction\_down = False

for i in range(len(ciphertext)):

decrypted\_text.append(rail[row][col])

col += 1

if row == 0 or row == key - 1:

direction\_down = not direction\_down

row += 1 if direction\_down else -1

return ''.join(decrypted\_text)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

plaintext = "HELLOTRANSPOSITION"

key = 3 # Number of rails

print("Plaintext:", plaintext)

# Encrypt the plaintext

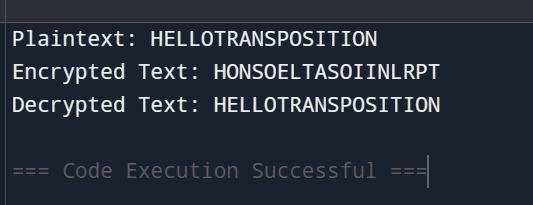
encrypted = rail\_fence\_encrypt(plaintext, key)

print("Encrypted Text:", encrypted)

# Decrypt the ciphertext

decrypted = rail\_fence\_decrypt(encrypted, key)

print("Decrypted Text:", decrypted)



### **Task Explanation:**

The Rail Fence Cipher is a transpositional cipher that involves writing the plaintext in a zigzag pattern across a given number of "rails" (rows), then reading off the ciphertext row by row. Decryption involves reversing this process.

### **Complexity:**

* **Encryption Time Complexity**: *O(n)O(n)*O(n), where *nn*n is the length of the text, as each character is processed once.
* **Decryption Time Complexity**: *O(n)O(n)*O(n), since we recreate the zigzag pattern and extract the original text.
* **Space Complexity**: *O(n)O(n)*O(n), for storing the zigzag pattern during encryption or decryption.

Q3

import hashlib

def hash\_password\_sha256(password):

"""

Hashes a password using the SHA-256 algorithm.

Parameters:

password (str): The input password to be hashed.

Returns:

str: The SHA-256 hashed representation as a hexadecimal string.

"""

# Encode the password to a bytes object

encoded\_password = password.encode('utf-8')

# Create a SHA-256 hash object

hash\_object = hashlib.sha256()

# Update the hash object with the encoded password

hash\_object.update(encoded\_password)

# Return the hexadecimal representation of the hash

return hash\_object.hexdigest()

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Input password

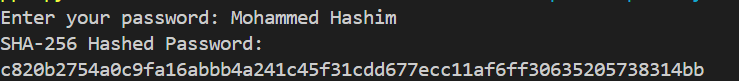
password = input("Enter your password: ")

# Compute its SHA-256 hash

hashed\_password = hash\_password\_sha256(password)

print("SHA-256 Hashed Password:")

print(hashed\_password)



### **Task Explanation:**

The **SHA-256** algorithm is a cryptographic hash function that produces a fixed-size 256-bit (32-byte) hash for a given input. This program will take a password string as input, hash it using SHA-256, and return its hexadecimal representation.

### **Complexity:**

* **Time Complexity**: *O(n)O(n)*O(n), where *nn*n is the length of the password string. Hashing processes every character once.
* **Space Complexity**: *O(1)O(1)*O(1), as the hash computation uses constant space.

Q5

import random

def load\_words\_from\_file(file\_path):

"""

Loads words from a dictionary file into a list.

Parameters:

file\_path (str): Path to the dictionary file.

Returns:

list: List of words from the file.

"""

try:

with open(file\_path, 'r') as file:

# Read lines, strip whitespace, and return as a list

return [line.strip() for line in file.readlines()]

except FileNotFoundError:

print(f"Error: File not found at {file\_path}.")

return []

def generate\_password(word\_list, num\_words=4, separator="-"):

"""

Generates a password using random words from a word list.

Parameters:

word\_list (list): List of words to choose from.

num\_words (int): Number of words in the password. Default is 4.

separator (str): Separator between words in the password. Default is "-".

Returns:

str: Generated password.

"""

if len(word\_list) < num\_words:

print("Error: Not enough words in the word list to generate a password.")

return ""

# Randomly select words from the list

selected\_words = random.sample(word\_list, num\_words)

# Join selected words with the separator

return separator.join(selected\_words)

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

# Path to a dictionary file

dictionary\_file = "/usr/share/dict/words" # Adjust path for your system or provide a custom file

words = load\_words\_from\_file(dictionary\_file)

if words:

# Generate a password with 4 random words and '-' as a separator

password = generate\_password(words, num\_words=4, separator="-")

print("Generated Password:", password)

OUTPUT

Assume the dictionary file contains words like apple, banana, cherry, date, elderberry, etc.

Generated Password: cherry-date-apple-elderberry

### **Task Explanation:**

This program generates a password by randomly selecting a combination of words from a dictionary file. The generated password can be customized to include a specific number of words and separators for better readability or security.

### **Complexity:**

* **Time Complexity**:
  + Loading words from the file: *O(n)O(n)*O(n), where *nn*n is the number of words in the file.
  + Random selection: *O(k)O(k)*O(k), where *kk*k is the number of words in the password.

Total: *O(n+k)O(n + k)*O(n+k).

* **Space Complexity**: *O(n)O(n)*O(n), to store the list of words from the dictionary.

Q6

import itertools

import string

def brute\_force\_attack(target\_password, max\_length=4):

"""

Simulates a brute-force attack to guess a target password.

Parameters:

target\_password (str): The password to guess.

max\_length (int): Maximum length of the password to attempt.

Returns:

str: The guessed password if found.

int: Total number of attempts.

"""

# Define the character set (adjust as needed)

character\_set = string.ascii\_letters + string.digits + string.punctuation

attempts = 0 # Counter for the number of attempts

print("Starting brute-force attack...")

# Iterate over all possible lengths up to max\_length

for length in range(1, max\_length + 1):

# Generate all possible combinations of the given length

for guess in itertools.product(character\_set, repeat=length):

attempts += 1

# Convert the tuple to a string

guess\_password = ''.join(guess)

# Check if the guessed password matches the target

if guess\_password == target\_password:

return guess\_password, attempts

return None, attempts # Return None if the password is not found

# Example usage

if \_\_name\_\_ == "\_\_main\_\_":

target = input("Enter the password to brute-force: ")

max\_len = int(input("Enter the maximum length to try: "))

guessed\_password, total\_attempts = brute\_force\_attack(target, max\_len)

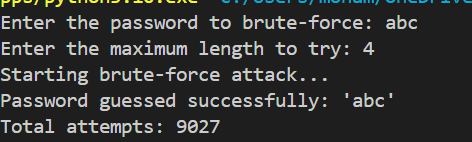
if guessed\_password:

print(f"Password guessed successfully: '{guessed\_password}'")

print(f"Total attempts: {total\_attempts}")

else:

print("Failed to guess the password within the given length.")



### **Task Explanation:**

A brute-force attack systematically attempts all possible combinations of characters to guess a password. This program simulates such an attack by generating combinations of characters up to a specified length and comparing them to a target password.

### **Complexity:**

* **Time Complexity**:
* *O(cl)O(c^l)*O(cl), where *cc*c is the size of the character set and *ll*l is the maximum length of the password.
* **Space Complexity**:

*O(l)O(l)*O(l), for storing the current guess.

Q7

**Generate a Key Pair (Private and Public Key)**

from cryptography.hazmat.primitives.asymmetric import rsa

from cryptography.hazmat.primitives import hashes

from cryptography.hazmat.primitives.asymmetric import padding

from cryptography.hazmat.primitives import serialization

def generate\_key\_pair():

private\_key = rsa.generate\_private\_key(

public\_exponent=65537,

key\_size=2048,

)

public\_key = private\_key.public\_key()

private\_pem = private\_key.private\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PrivateFormat.PKCS8,

encryption\_algorithm=serialization.NoEncryption()

)

public\_pem = public\_key.public\_bytes(

encoding=serialization.Encoding.PEM,

format=serialization.PublicFormat.SubjectPublicKeyInfo

)

return private\_pem, public\_pem

private\_key, public\_key = generate\_key\_pair()

print(f"Private Key: \n{private\_key.decode()}")

print(f"Public Key: \n{public\_key.decode()}")

**Digitally Sign a Document**

def sign\_document(document\_content, private\_key\_pem):

private\_key = serialization.load\_pem\_private\_key(private\_key\_pem, password=None)

document\_hash = hashes.Hash(hashes.SHA256())

document\_hash.update(document\_content.encode()) # Use the document content to generate hash

digest = document\_hash.finalize()

signature = private\_key.sign(

digest,

padding.PKCS1v15(),

hashes.SHA256()

)

return signature

document = "This is a confidential document that requires a digital signature."

signature = sign\_document(document, private\_key)

print(f"Document Signature: \n{signature.hex()}")

def verify\_signature(document\_content, signature, public\_key\_pem):

public\_key = serialization.load\_pem\_public\_key(public\_key\_pem)

document\_hash = hashes.Hash(hashes.SHA256())

document\_hash.update(document\_content.encode()) # Use the document content to generate hash

digest = document\_hash.finalize()

try:

public\_key.verify(

signature,

digest,

padding.PKCS1v15(),

hashes.SHA256()

)

print("Document is authentic and signature is valid.")

except Exception as e:

print(f"Signature verification failed: {e}")

print("\nVerifying the signature on the received document...\n")

verify\_signature(document, signature, public\_key)

**Sending the Signed Document**

print("\nSending the signed document and signature...")

print(f"Document: {document}")

print(f"Signature: {signature.hex()}")

print("\nVerification at the recipient's side...")

verify\_signature(document, signature, public\_key)

OUTPUT

Private Key:

-----BEGIN PRIVATE KEY-----

...

-----END PRIVATE KEY-----

Public Key:

-----BEGIN PUBLIC KEY-----

...

-----END PUBLIC KEY-----

Document Signature:

f61dcd0eb1e97f7bca7a57d8bb3...

Sending the signed document and signature...

Document: This is a confidential document that requires a digital signature.

Signature: f61dcd0eb1e97f7bca7a57d8bb3...

Verification at the recipient's side...

Document is authentic and signature is valid.

**Complexity:**

1. **Key Generation**:
   * **Time Complexity**: RSA key generation is **O(n^3)**, where n is the key size in bits (2048 bits for example).
2. **Signing**:
   * **Time Complexity**: Signing a document using RSA is **O(n^2)** due to the modular exponentiation involved.
3. **Verification**:
   * **Time Complexity**: Verification is also **O(n^2)**, since it involves similar modular exponentiation with the public key.
4. **Hashing**:
   * **Time Complexity**: Hashing the document is **O(m)**, where m is the length of the document.