A)Caesar/Shift/Additive Cipher

```java
import java.util.Scanner;
public class CaesarCipher {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter text: ");
        String text = sc.nextLine();
        System.out.print("Enter shift key: ");
        int key = sc.nextInt();
        String encrypted = encrypt(text, key);
        String decrypted = decrypt(encrypted, key);
        System.out.println("Encoded Text: " + encrypted);
        System.out.println("Decoded Text: " + decrypted);
    }
    public static String encrypt(String text, int key) {
        StringBuilder result = new StringBuilder();
        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                char base = Character.isUpperCase(c) ? 'A' : 'a';
                result.append((char) ((c - base + key) % 26 + base));
            } else {
                result.append(c);
            }
        }
        return result.toString();
    }
    public static String decrypt(String text, int key) {
```

```
        return encrypt(text, 26 - key);
    }
}
```

B)Atbash cipher

```
import java.util.Scanner;


public class Atbash {
    public static String encrypt(String text) {
        StringBuilder encrypted = new StringBuilder();
        for (char c : text.toCharArray()) {
            if (Character.isLetter(c)) {
                char newChar = (char) ('A' + 'Z' - Character.toUpperCase(c));
                encrypted.append(Character.isLowerCase(c) ? Character.toLowerCase(newChar) :
newChar);
            } else {
                encrypted.append(c);
            }
        }
        return encrypted.toString();
    }
    public static String decrypt(String text) {
        return encrypt(text);
    }
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
```

```java
        System.out.print("Enter text to encrypt: ");
        String input = scanner.nextLine();
        String encryptedText = encrypt(input);
        System.out.println("Encrypted text: " + encryptedText);
        String decryptedText = decrypt(encryptedText);
        System.out.println("Decrypted text: " + decryptedText);
    }
}
```

C)Multiplicative Cipher

```java
import java.util.Scanner;
public class Multiplicative {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter text: ");
        String plainText = sc.nextLine();
        System.out.print("Enter key: ");
        int key = sc.nextInt();
        String encryptedText = encrypt(plainText, key);
        String decryptedText = decrypt(encryptedText, key);
        System.out.println("Encrypted Text: " + encryptedText);
        System.out.println("Decrypted Text: " + decryptedText);
    }
    public static String encrypt(String plainText, int key) {
        StringBuilder encryptedText = new StringBuilder();
```

```
    for (char c : plainText.toCharArray()) {
        if (Character.isLetter(c)) {
            char base = Character.isUpperCase(c) ? 'A' : 'a';
            encryptedText.append((char) ((c - base) * key % 26 + base));
        } else {
            encryptedText.append(c);
        }
    }
    return encryptedText.toString();
}
public static String decrypt(String encryptedText, int key) {
    int inverseKey = modInverse(key, 26);
    return encrypt(encryptedText, inverseKey);
}
public static int modInverse(int a, int m) {
    int m0 = m;
    int y = 0, x = 1;
    if (m == 1)
        return 0;
    while (a > 1) {
        int q = a / m;
        int t = m;
        m = a % m;
        a = t;
        t = y;
        y = x - q * y;
        x = t;
    }
```

```
        if (x < 0)
            x += m0;
        return x;
    }
}
```

D)Vigenère Cipher

```
import java.util.Scanner;

public class VigenereLab {

    static String generateKey(String text, String key) {
        StringBuilder newKey = new StringBuilder();
        key = key.toUpperCase();
        for (int i = 0, j = 0; i < text.length(); i++) {
            char ch = text.charAt(i);
            if (Character.isLetter(ch)) {
                newKey.append(key.charAt(j % key.length()));
                j++;
            } else {
                newKey.append(ch); // Keep spaces/punctuation unchanged
            }
        }
        return newKey.toString();
    }
```

```java
static String encrypt(String text, String key) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();
    key = generateKey(text, key);

    for (int i = 0; i < text.length(); i++) {
        char t = text.charAt(i);
        char k = key.charAt(i);
        if (Character.isLetter(t)) {
            char c = (char) (((t - 'A' + k - 'A') % 26) + 'A');
            result.append(c);
        } else {
            result.append(t);
        }
    }
    return result.toString();
}

static String decrypt(String text, String key) {
    StringBuilder result = new StringBuilder();
    text = text.toUpperCase();
    key = generateKey(text, key);

    for (int i = 0; i < text.length(); i++) {
        char t = text.charAt(i);
        char k = key.charAt(i);
        if (Character.isLetter(t)) {
            char c = (char) (((t - k + 26) % 26) + 'A');
```

```
                result.append(c);
            } else {
                result.append(t);
            }
        }
        return result.toString();
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter plaintext: ");
        String plaintext = sc.nextLine();
        System.out.print("Enter key: ");
        String key = sc.nextLine();

        String encrypted = encrypt(plaintext, key);
        String decrypted = decrypt(encrypted, key);

        System.out.println("Encrypted Text: " + encrypted);
        System.out.println("Decrypted Text: " + decrypted);
    }
}
```

E)One-Time Pad Cipher

```
import java.util.Scanner;
public class OneTimePadCipher {
    public static String encryptText(String plainText, String key) {
```

```
    String cipherText = "";
    int cipher[] = new int[key.length()];
    for (int i = 0; i < key.length(); i++) {
        cipher[i] = plainText.charAt(i) - 'A' + key.charAt(i) - 'A';
    }
    for (int i = 0; i < key.length(); i++) {
        if (cipher[i] > 25) {
            cipher[i] = cipher[i] - 26;
        }
    }
    for (int i = 0; i < key.length(); i++) {
        int x = cipher[i] + 'A';
        cipherText += (char) x;
    }
    return cipherText;
}

public static String decryptText(String cipherText, String key) {
    String plainText = "";
    int plain[] = new int[key.length()];
    for (int i = 0; i < key.length(); i++) {
        plain[i] = cipherText.charAt(i) - 'A' - (key.charAt(i) - 'A');
    }
    for (int i = 0; i < key.length(); i++) {
        if (plain[i] < 0) {
            plain[i] = plain[i] + 26;
        }
    }
```

```
    for (int i = 0; i < key.length(); i++) {
        int x = plain[i] + 'A';
        plainText += (char) x;
    }
    return plainText;
}

public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.print("Enter the plain text: ");
    String plainText = scanner.nextLine().toUpperCase();
    System.out.print("Enter the key: ");
    String key = scanner.nextLine().toUpperCase();
    if (plainText.length() != key.length()) {
        System.out.println("Error: The key length must match the plain text length.");
        scanner.close();
        return;
    }
    // Encrypt the plain text
    String encryptedText = encryptText(plainText, key);
    System.out.println("Encrypted Text: " + encryptedText);

    // Decrypt the cipher text
    String decryptedText = decryptText(encryptedText, key);
    System.out.println("Decrypted Text: " + decryptedText);

    scanner.close();
}
```

```
}
```

F)Implementation of Authentication, Authorization and Access Rights

```
import hashlib, jwt, datetime, random, string

SECRET_KEY = "secret"
users = {
    "admin_user": {"password": "admin123", "role": "admin"},
    "normal_user": {"password": "user123", "role": "user"}
}
user_token = None

def generate_token(username, role):
    payload = {"username": username, "role": role, "exp": datetime.datetime.utcnow() + datetime.timedelta(minutes=30)}
    return jwt.encode(payload, SECRET_KEY, algorithm="HS256")

def generate_common_passwords(n=100):
    common = ["123456", "password", "qwerty", "admin123", "user123", "welcome", "admin", "login", "passw0rd", "1234"]
    common.extend([u["password"] for u in users.values()])
    common = list(set(common))
    while len(common) < n:
        common.append(''.join(random.choices(string.ascii_letters + string.digits, k=random.randint(6, 10))))
    return common
```

```
def generate_rainbow_table(wordlist):
    return {hashlib.sha256(w.encode()).hexdigest(): w for w in wordlist}


def rainbow_attack():
    print("\n--- Rainbow Table Attack ---")
    rainbow_table = generate_rainbow_table(generate_common_passwords())
    cracked = {}
    for uname, data in users.items():
        h = hashlib.sha256(data["password"].encode()).hexdigest()
        if h in rainbow_table:
            cracked[uname] = {"password": rainbow_table[h], "role": data["role"]}
    if cracked:
        print("\nCracked Users:")
        for u, d in cracked.items():
            print(f"Username: {u}, Password: {d['password']}, Role: {d['role']}")
        choice = input("\nLogin using cracked username or 'n' to cancel: ")
        if choice in cracked:
            global user_token
            user_token = generate_token(choice, cracked[choice]["role"])
            print(f"\nLogged in as {choice} ({cracked[choice]['role']})")
            dashboard(cracked[choice]["role"])
    else:
        print("\nNo passwords cracked.")


def login():
    global user_token
    u = input("Username: "); p = input("Password: ")
```

```
    if u in users and hashlib.sha256(p.encode()).hexdigest() ==
hashlib.sha256(users[u]["password"].encode()).hexdigest():
        user_token = generate_token(u, users[u]["role"])
        print(f"\nLogin Successful as {users[u]['role'].capitalize()}!")
        dashboard(users[u]["role"])
    else:
        print("\nInvalid Credentials!")


def register_user():
    u = input("New username: "); p = input("New password: "); r = input("Role (user/admin):
").lower()
    if u in users: print("\nUser already exists!")
    else:
        users[u] = {"password": p, "role": r}
        print(f"\nUser '{u}' registered!")


def dashboard(role):
    while True:
        print("\n--- Dashboard ---\n1. View Users")
        if role == "admin": print("2. Register New User")
        print("3. Logout")
        c = input("Choice: ")
        if c == '1': print("Users:", list(users.keys()))
        elif c == '2' and role == "admin": register_user()
        elif c == '3': print("Logged out."); break
        else: print("Invalid Option.")


def main():
    while True:
```

```
        print("\n--- Menu ---\n1. Login\n2. Rainbow Table Attack\n3. Exit")
        c = input("Choice: ")
        if c == '1': login()
        elif c == '2': rainbow_attack()
        elif c == '3': print("Goodbye!"); break
        else: print("Invalid Option.")

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

G)Implementation of Key Exchange Diffie Hellman Algorithm

```
def power(a, b, p):
    return (a ** b) % p

def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True

def main():
    print("Diffie-Hellman Key Exchange")

    while True:
```

```
        P = int(input("Enter a prime number P: "))
        if is_prime(P):
            break
        else:
            print("P must be a prime number. Please try again.")

    G = int(input("Enter a primitive root G: "))

    print(f"Public keys: P = {P}, G = {G}")

    a = int(input("Enter A's private key a: "))
    b = int(input("Enter B's private key b: "))

    print(f"A's private key = {a}, B's private key = {b}")

    x = power(G, a, P)
    y = power(G, b, P)

    print(f"A's public key x = {x}, B's public key y = {y}")

    ka = power(y, a, P)
    kb = power(x, b, P)

    print(f"Shared secret key for A = {ka}, Shared secret key for B = {kb}")
    print(f"Final Shared Secret Key: {ka}")

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

H)Implementing Key Generation in AES

pip install pycryptodome

```
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
import base64

def pad(data):
    """PKCS7 Padding"""
    pad_length = 16 - (len(data) % 16)
    return data + bytes([pad_length] * pad_length)

def unpad(data):
    """Remove PKCS7 Padding"""
    pad_length = data[-1]
    if pad_length > 16:  # Invalid padding case
        raise ValueError("Invalid padding")
    return data[:-pad_length]

def encrypt_aes(key, plaintext):
    """Encrypts plaintext using AES (CBC mode)"""
    plaintext = pad(plaintext.encode())  # Ensure bytes input
    iv = get_random_bytes(16)  # 16-byte IV
    cipher = AES.new(key, AES.MODE_CBC, iv)
    ciphertext = cipher.encrypt(plaintext)
```

```
    return base64.b64encode(iv + ciphertext).decode()  # Encode as Base64

def decrypt_aes(key, encrypted_text):
    encrypted_data = base64.b64decode(encrypted_text)  # Decode from Base64
    iv, ciphertext = encrypted_data[:16], encrypted_data[16:]
    cipher = AES.new(key, AES.MODE_CBC, iv)
    plaintext = cipher.decrypt(ciphertext)
    return unpad(plaintext).decode()  # Decode after unpadding

key = get_random_bytes(16)
message = input("Enter message: ")
encrypted_message = encrypt_aes(key, message)
decrypted_message = decrypt_aes(key, encrypted_message)

print("Original Message:", message)
print("AES Key (Base64):", base64.b64encode(key).decode())
print("Encrypted Message:", encrypted_message)
print("Decrypted Message:", decrypted_message)
```

I)Implementation of RSA

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import serialization, hashes
import base64
import os

def generate_keys():
```

```
    """Generate RSA public and private keys."""
    private_key = rsa.generate_private_key(
        public_exponent=65537,
        key_size=2048
    )
    public_key = private_key.public_key()
    return private_key, public_key


def save_keys(private_key, public_key):
    """Save RSA keys to files."""
    private_pem = private_key.private_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PrivateFormat.TraditionalOpenSSL,
        encryption_algorithm=serialization.NoEncryption()
    )
    public_pem = public_key.public_bytes(
        encoding=serialization.Encoding.PEM,
        format=serialization.PublicFormat.SubjectPublicKeyInfo
    )

    with open("private_key.pem", "wb") as private_file:
        private_file.write(private_pem)

    with open("public_key.pem", "wb") as public_file:
        public_file.write(public_pem)
```

```
def load_keys():
    """Load RSA keys from files."""
    if not os.path.exists("private_key.pem") or not os.path.exists("public_key.pem"):
        print("Keys not found. Generating new keys...")
        private_key, public_key = generate_keys()
        save_keys(private_key, public_key)
    else:
        with open("private_key.pem", "rb") as private_file:
            private_key = serialization.load_pem_private_key(
                private_file.read(),
                password=None
            )

        with open("public_key.pem", "rb") as public_file:
            public_key = serialization.load_pem_public_key(
                public_file.read()
            )
    return private_key, public_key



def encrypt_message(public_key, message):
    """Encrypt a message using RSA public key."""
    ciphertext = public_key.encrypt(
        message.encode(),
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
```

```
        )
    )
    return base64.b64encode(ciphertext).decode()  # Convert to Base64 for readability


def decrypt_message(private_key, encrypted_text):
    """Decrypt a message using RSA private key."""
    ciphertext = base64.b64decode(encrypted_text)  # Decode from Base64
    plaintext = private_key.decrypt(
        ciphertext,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
        )
    )
    return plaintext.decode()


private_key, public_key = load_keys()
message = input("Enter the message to encrypt: ")
encrypted_message = encrypt_message(public_key, message)
print("\nEncrypted Message:", encrypted_message)

encrypted_input = input("\nEnter the encrypted message to decrypt: ")
try:
    decrypted_message = decrypt_message(private_key, encrypted_input)
    print("\nDecrypted Message:", decrypted_message)
```

```
except Exception as e:
    print("\nDecryption failed. Error:", e)
```

J)Web Security - SQL Injection

```
import streamlit as st
import sqlite3

def connect_db():
    conn = sqlite3.connect("users.db", check_same_thread=False)
    conn.execute("CREATE TABLE IF NOT EXISTS users (username TEXT PRIMARY KEY,
password TEXT)")
    return conn

def query_db(query, params=None):
    conn = connect_db()
    cur = conn.cursor()
    try:
        cur.execute(query, params or ())
        data = cur.fetchall()
        conn.commit()
    except Exception as e:
        st.error(f"Database Error: {e}")
        data = []
    conn.close()
    return data
```

```python
def authenticate(user, pwd, secure):
    q = "SELECT * FROM users WHERE username = ? AND password = ?" if secure else f"SELECT * FROM users WHERE username = '{user}' AND password = '{pwd}'"
    return query_db(q, (user, pwd) if secure else None)


def register(user, pwd):
    if query_db("SELECT * FROM users WHERE username = ?", (user,)): return False
    query_db("INSERT INTO users (username, password) VALUES (?, ?)", (user, pwd)); return True


def update_pwd(user, new_pwd, secure):
    q = "UPDATE users SET password = ? WHERE username = ?" if secure else f"UPDATE users SET password = '{new_pwd}' WHERE username = '{user}'"
    return query_db(q, (new_pwd, user) if secure else None)


def delete_user(user, secure):
    q = "DELETE FROM users WHERE username = ?" if secure else f"DELETE FROM users WHERE username = '{user}'"
    return query_db(q, (user,) if secure else None)


# UI
st.title(" Login System")
secure_mode = st.sidebar.checkbox("Secure Mode", True)


if st.sidebar.button("View Users"):
    st.sidebar.dataframe(query_db("SELECT * FROM users"))


if 'logged' not in st.session_state:
    st.session_state['logged'] = False
```

```python
if not st.session_state['logged']:
    u = st.text_input("Username")
    p = st.text_input("Password", type="password")
    if st.button("Login"):
        if authenticate(u, p, secure_mode):
            st.success("Login successful!")
            st.session_state['logged'] = True
            st.session_state['user'] = u
        else:
            st.error(" Invalid credentials.")
    if st.button("Register"):
        if register(u, p): st.success(" Registered successfully.")
        else: st.warning(" User already exists.")
else:
    new_p = st.text_input("New Password", type="password")
    if st.button("Update Password"): update_pwd(st.session_state['user'], new_p,
secure_mode); st.success("Password updated.")
    del_u = st.text_input("Delete Username")
    if st.button("Delete User"): delete_user(del_u, secure_mode); st.success("User deleted.")
    if st.button("Logout"): st.session_state['logged'] = False; st.experimental_rerun()
```

query:

```
-- Login
' OR '1'='1
```

-- Delete all users (enter in delete username field)

anything' OR '1'='1

-- Update:

x' OR '1'='1

hacked'; --

K)Implementation of Secure Hash Algorithm

```
import hashlib

def hash_message(message: str):
    encoded_msg = message.encode()
    hashes = {
        "SHA-1": hashlib.sha1(encoded_msg).hexdigest(),
        "SHA-224": hashlib.sha224(encoded_msg).hexdigest(),
        "SHA-256": hashlib.sha256(encoded_msg).hexdigest(),
        "SHA-384": hashlib.sha384(encoded_msg).hexdigest(),
        "SHA-512": hashlib.sha512(encoded_msg).hexdigest(),
        "SHA3-224": hashlib.sha3_224(encoded_msg).hexdigest(),
        "SHA3-256": hashlib.sha3_256(encoded_msg).hexdigest(),
        "SHA3-384": hashlib.sha3_384(encoded_msg).hexdigest(),
        "SHA3-512": hashlib.sha3_512(encoded_msg).hexdigest()
    }
    return hashes

if __name__ == "__main__":
```

```python
    message = input("Enter a message to hash: ")
    hash_results = hash_message(message)
    for algo, hash_value in hash_results.items():
        print(f"{algo}: {hash_value}")
```

L)Implementation of Digital Signature Standard

```python
# pip install python-docx
# pip install mammoth

import mammoth
import hashlib
import os

def extract_content(filepath):
    """Extract raw text content from a DOCX file."""
    with open(filepath, "rb") as docx_file:
        result = mammoth.extract_raw_text(docx_file)
        return result.value

def generate_sha512(filepath):
    """Generate SHA-512 hash for the content of the DOCX file."""
    content = extract_content(filepath)
    return hashlib.sha512(content.encode("utf-8")).hexdigest()

def sign_word(filepath):
    """Sign the DOCX file by generating its SHA-512 hash and saving the signature."""
    filehash = generate_sha512(filepath)
```

```
    sign_file = filepath.replace(".docx", ".sig")
    with open(sign_file, "w") as sig_file:
        sig_file.write(filehash)
    print(f"Both document and signature file saved! Signature file: {sign_file}")
    return filehash

def verify(filepath):
    """Verify the integrity of the DOCX file using the signature."""
    sig_file = filepath.replace(".docx", ".sig")
    if not os.path.exists(sig_file):
        print(f"Signature file not found: {sig_file}")
        return False

    file_hash = generate_sha512(filepath)
    with open(sig_file, "r") as f:
        saved_sig = f.read().strip()

    is_valid = file_hash == saved_sig
    if is_valid:
        print("Signature is valid. Document has not been modified.")
    else:
        print("Warning: File appears to have been tampered with!")

    return is_valid

# Use the path to your uploaded file
file_path = "sample.docx"
```

```
# Sign the document
sign_word(file_path)

# Verify the document
verify(file_path)

# Verify the document again for confirmation
verify(file_path)
```