Implementing Substitution and Transposition Ciphers

1 Code: Python code for implementing Caesar Cipher

```
# Function to encrypt the text using Caesar Cipher
def encrypt(text, s):
    result = ""  # Initialize result as an empty string

    # Traverse through the text
    for i in range(len(text)):
        char = text[i]

        # Encrypt uppercase characters
        if char.isupper():
            result += chr((ord(char) + s - 65) % 26 + 65)

        # Encrypt lowercase characters
        elif char.islower():
            result += chr((ord(char) + s - 97) % 26 + 97)

        # For non-alphabetical characters, add them unchanged
        else:
            result += char

    return result

# Input the text to encrypt
text = input("Enter the text to encrypt: ")
s = int(input("Enter the shift value: "))

# Output the original text and the encrypted text
print("Text: " + text)
print("Cipher: " + encrypt(text, s))
```

Code: Python code for implementing Railfence Cipher

```
# Function to implement the Rail Fence Cipher
def RailFence(txt):
    result = ""  # Initialize result as an empty string

    # First loop for characters at even indices
    for i in range(len(txt)):
        if i % 2 == 0:  # Even index
            result += txt[i]

    # Second loop for characters at odd indices
    for i in range(len(txt)):
        if i % 2 != 0:  # Odd index
            result += txt[i]

    return result

# Input the string to encrypt
txt = input("Enter a string: ")

# Output the encrypted text
print("Encrypted text using Rail Fence Cipher: " + RailFence(txt))
```

RSA Encryption and Decryption

Code: Python code for implementing RSA Algorithm

pip install pycryptodome

```
from Crypto.PublicKey import RSA
from Crypto.Cipher import PKCS1_OAEP
import binascii

# Generate RSA key pair (1024 bits)
keyPair = RSA.generate(1024)

# Get the public key
pubKey = keyPair.publickey()

# Print the public key (n and e) in hexadecimal format
print(f"Public key: (n={hex(pubKey.n)}, e={hex(pubKey.e)})")

# Export the public key in PEM format
pubKeyPEM = pubKey.exportKey()
print(pubKeyPEM.decode('ascii'))

# Print the private key (n and d) in hexadecimal format
print(f"Private key: (n={hex(pubKey.n)}, d={hex(keyPair.d)})")

# Export the private key in PEM format
privKeyPEM = keyPair.exportKey()
print(privKeyPEM.decode('ascii'))

# Encryption
msg = b'Ismile Academy'  # Message to encrypt in bytes

# Initialize the encryptor with public key
encryptor = PKCS1_OAEP.new(pubKey)
encrypted = encryptor.encrypt(msg)

# Print the encrypted message in hexadecimal format
print("Encrypted:", binascii.hexlify(encrypted))
```

Message Authentication Codes (MAC)

Code: Python code for implementing MD5 Algorithm

```
import hashlib

# Generating MD5 hash for 'Ismile'
result1 = hashlib.md5(b'Ismile')

# Generating MD5 hash for 'Esmile'
result2 = hashlib.md5(b'Esmile')

# Printing the byte equivalent of each hash
print("The byte equivalent of 'Ismile' hash is: ", end="")
print(result1.digest())

print("The byte equivalent of 'Esmile' hash is: ", end="")
print(result2.digest())
```

Code: Python code for implementing SHA Algorithm

import hashlib

```
# Function to generate SHA hash
def generate_sha(text, algorithm="sha256"):
    # Select the hashing algorithm
    if algorithm == "sha1":
        hash_object = hashlib.sha1()
    elif algorithm == "sha256":
        hash_object = hashlib.sha256()
    elif algorithm == "sha512":
        hash_object = hashlib.sha512()
    else:
        raise ValueError("Unsupported algorithm. Use 'sha1', 'sha256', or 'sha512'.")

    # Update the hash object with the bytes of the input text
    hash_object.update(text.encode('utf-8'))

    # Return the hexadecimal representation of the hash
    return hash_object.hexdigest()

# Input the text and choose SHA algorithm (sha1, sha256, sha512)
text = input("Enter the text to hash: ")
algorithm = input("Enter the SHA algorithm (sha1, sha256, sha512): ").lower()

# Output the SHA hash
try:
    print(f"{algorithm.upper()} Hash:", generate_sha(text, algorithm))
except ValueError as e:
    print(e)
```

Digital Signatures
Code: Python code for implementing SHA Algorithm
pip install pycryptodome

```
from Crypto.Signature import PKCS1_v1_5
from Crypto.Hash import SHA256
from Crypto.PublicKey import RSA
from Crypto import Random

# Function to generate a signature using the private key
def generate_signature(private_key, message):
    key = RSA.importKey(private_key)
    hashed_message = SHA256.new(message.encode('utf-8'))  # Hash the message using
SHA-256
    signer = PKCS1_v1_5.new(key)  # Create a signer using the private key
    signature = signer.sign(hashed_message)  # Generate the signature
    return signature

# Function to verify the signature using the public key
def verify_signature(public_key, message, signature):
    key = RSA.importKey(public_key)
    hashed_message = SHA256.new(message.encode('utf-8'))  # Hash the message
    verifier = PKCS1_v1_5.new(key)  # Create a verifier using the public key
    return verifier.verify(hashed_message, signature)  # Verify the signature

# Generate RSA key pair
random_generator = Random.new().read
key_pair = RSA.generate(2048, random_generator)

# Export public and private keys
private_key = key_pair.export_key()
public_key = key_pair.publickey().export_key()

# Message to sign
message = "Hello, World!"

# Generate the signature using the private key
signature = generate_signature(private_key, message)
print("Generated Signature:", signature)

# Verify the signature using the public key
is_valid = verify_signature(public_key, message, signature)
print("Signature Verification Result:", is_valid)
```

Key Exchange using Diffe-Hellman
Code: Python code for implementing Diffie-Hellman

```
from random import randint

def diffie_hellman_key_exchange():
    # Define the prime number P and the generator G
    P = 23
    G = 9

    print('The Value of P is:', P)
    print('The Value of G is:', G)

    # Alice's secret number (private key)
    a = 4  # Alice's private key
    print('Secret Number for Alice is:', a)

    # Alice calculates x
    x = pow(G, a, P)
    print('Alice sends x to Bob:', x)

    # Bob's secret number (private key)
    b = 6  # Bob's private key
    print('Secret Number for Bob is:', b)

    # Bob calculates y
    y = pow(G, b, P)
    print('Bob sends y to Alice:', y)

    # Alice calculates the secret key
    ka = pow(y, a, P)
    print('Secret key for Alice is:', ka)

    # Bob calculates the secret key
    kb = pow(x, b, P)
    print('Secret key for Bob is:', kb)
```

```python
    # Check if both secret keys are the same
    if ka == kb:
        print('Key exchange successful! Both keys are equal.')
    else:
        print('Key exchange failed! Keys are not equal.')

# Execute the Diffie-Hellman key exchange
diffie_hellman_key_exchange()
```