Cryptography and Information Security Lab

CSEL-4110

Assignment on Cryptographic Techniques and Algorithms

Submitted By

NISHAT MAHMUD
ID: B190305003

Submitted To

MOHAMMED NASIR UDDIN, PHD
Professor
Dept. of CSE
Jagannath University, Dhaka - 1100

Caesar Cipher

CaesarCipher.py

```python
# Caesar Cipher Encryption and Decryption

# Function to encrypt using Caesar Cipher
def caesar_encrypt(plaintext, key):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():  # Encrypt only alphabetic characters
            char = char.upper()  # Convert all characters to uppercase
            shift = ord('A')  # Use uppercase letters for shifting
            encrypted_char = chr((ord(char) - shift + key) % 26 + shift)  # Shift character
            ciphertext += encrypted_char
        else:
            ciphertext += char  # Non-alphabet characters remain unchanged
    return ciphertext

# Function to decrypt using Caesar Cipher
def caesar_decrypt(ciphertext, key):
    plaintext = ""
    for char in ciphertext:
        if char.isalpha():  # Decrypt only alphabetic characters
            shift = ord('A')  # Use uppercase letters for shifting
            decrypted_char = chr((ord(char) - shift - key) % 26 + shift)  # Reverse shift
            plaintext += decrypted_char
        else:
            plaintext += char  # Non-alphabet characters remain unchanged
    return plaintext

# Example Usage
key = 3  # Shift key
message = "nishatmahmud"

print("Original Message:", message)
ciphertext = caesar_encrypt(message, key)
print("Encrypted Message:", ciphertext)
decrypted_message = caesar_decrypt(ciphertext, key)
print("Decrypted Message:", decrypted_message)
```

Output

Output

```
Original Message: nishatmahmud
Encrypted Message: QLVKDWPDKPXG
Decrypted Message: NISHATMAHMUD
```

Multiplicative Cipher

MultiplicativeCipher.py

```python
# Multiplicative Cipher Encryption and Decryption

# Function to find the modular inverse of 'a' under modulo 'm'
def mod_inverse(a, m):
    for i in range(1, m):
        if (a * i) % m == 1:
            return i
    return None

# Function to encrypt using Multiplicative Cipher
def multiplicative_encrypt(plaintext, key):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():  # Encrypt only alphabetic characters
            char = char.upper()  # Convert to uppercase
            x = ord(char) - ord('A')  # Convert letter to number (A=0, B=1, ..., Z=25)
            encrypted_char = (key * x) % 26  # Apply multiplicative cipher formula
            ciphertext += chr(encrypted_char + ord('A'))  # Convert number back to letter
        else:
            ciphertext += char  # Non-alphabet characters remain unchanged
    return ciphertext

# Function to decrypt using Multiplicative Cipher
def multiplicative_decrypt(ciphertext, key):
    plaintext = ""
    a_inv = mod_inverse(key, 26)  # Find the modular inverse of 'a'
    if a_inv is None:
        return "Decryption not possible, 'a' has no modular inverse!"

    for char in ciphertext:
        if char.isalpha():  # Decrypt only alphabetic characters
            x = ord(char) - ord('A')  # Convert letter to number (A=0, B=1, ..., Z=25)
            decrypted_char = (a_inv * x) % 26  # Apply decryption formula
            plaintext += chr(decrypted_char + ord('A'))  # Convert number back to letter
        else:
            plaintext += char  # Non-alphabet characters remain unchanged
    return plaintext

# Example Usage
key = 5  # Key 'a' (must be coprime with 26)
message = "nishatmahmud"

print("Original Message:", message)
ciphertext = multiplicative_encrypt(message, key)
print("Encrypted Message:", ciphertext)
decrypted_message = multiplicative_decrypt(ciphertext, key)
print("Decrypted Message:", decrypted_message)
```

Output

Output

```
Original Message: nishatmahmud
Encrypted Message: NOMJARIAJIWP
Decrypted Message: NISHATMAHMUD
```

Affine Cipher

AffineCipher.py

```python
# Affine Cipher Encryption and Decryption

# Function to find the modular inverse of 'a' under modulo 'm'
def mod_inverse(a, m):
    for i in range(1, m):
        if (a * i) % m == 1:
            return i
    return None

# Function to encrypt using Affine Cipher
def affine_encrypt(plaintext, a, b):
    ciphertext = ""
    for char in plaintext:
        if char.isalpha():  # Encrypt only alphabetic characters
            x = ord(char.upper()) - ord('A')  # Convert letter to number (A=0, B=1, ..., Z=25)
            encrypted_char = (a * x + b) % 26  # Apply Affine formula
            ciphertext += chr(encrypted_char + ord('A'))  # Convert number back to letter
        else:
            ciphertext += char  # Non-alphabet characters remain unchanged
    return ciphertext

# Function to decrypt using Affine Cipher
def affine_decrypt(ciphertext, a, b):
    plaintext = ""
    a_inv = mod_inverse(a, 26)  # Find the modular inverse of 'a'
    if a_inv is None:
        return "Decryption not possible, 'a' has no modular inverse!"

    for char in ciphertext:
        if char.isalpha():  # Decrypt only alphabetic characters
            x = ord(char.upper()) - ord('A')  # Convert letter to number (A=0, B=1, ..., Z=25)
            decrypted_char = (a_inv * (x - b)) % 26  # Apply Affine decryption formula
            plaintext += chr(decrypted_char + ord('A'))  # Convert number back to letter
        else:
            plaintext += char  # Non-alphabet characters remain unchanged
    return plaintext
```

AffineCipher.py

```
# Example Usage
key_a = 5  # Key 'a' (must be coprime with 26)
key_b = 8  # Key 'b'
message = "nishatmahmud"

print("Original Message:", message)
ciphertext = affine_encrypt(message, key_a, key_b)
print("Encrypted Message:", ciphertext)
decrypted_message = affine_decrypt(ciphertext, key_a, key_b)
print("Decrypted Message:", decrypted_message)
```

Output

Output

```
Original Message: nishatmahmud
Encrypted Message: VWURIZQIRQEX
Decrypted Message: NISHATMAHMUD
```

Vigenère Cipher

VigenèreCipher.py

```
# Vigenère Cipher Encryption and Decryption

# Function to generate the key in a repeated manner to match the length of the message
def generate_key(plaintext, key):
    key = key.upper()
    key = list(key)
    if len(plaintext) == len(key):
        return key
    else:
        for i in range(len(plaintext) - len(key)):
            key.append(key[i % len(key)])
    return "".join(key)

# Function to encrypt using Vigenère Cipher
def vigenere_encrypt(plaintext, key):
    # Remove spaces and convert to uppercase
    plaintext = plaintext.replace(" ", "").upper()
    key = generate_key(plaintext, key)

    ciphertext = ""
    for i in range(len(plaintext)):
```

VigenèreCipher.py

```
        # Shift the character based on the key
        p = ord(plaintext[i]) - ord('A')
        k = ord(key[i]) - ord('A')
        encrypted_char = chr((p + k) % 26 + ord('A'))
        ciphertext += encrypted_char

    return ciphertext

# Function to decrypt using Vigenère Cipher
def vigenere_decrypt(ciphertext, key):
    key = generate_key(ciphertext, key)

    plaintext = ""
    for i in range(len(ciphertext)):
        # Reverse the shift based on the key
        c = ord(ciphertext[i]) - ord('A')
        k = ord(key[i]) - ord('A')
        decrypted_char = chr((c - k + 26) % 26 + ord('A'))  # +26 ensures we avoid negative
mod
        plaintext += decrypted_char

    return plaintext

# Example Usage
message = "nishatmahmud"
key = "KEYWORD"

print("Original Message:", message)
ciphertext = vigenere_encrypt(message, key)
print("Encrypted Message:", ciphertext)
decrypted_message = vigenere_decrypt(ciphertext, key)
print("Decrypted Message:", decrypted_message)
```

Output

Output

```
Original Message: nishatmahmud
Encrypted Message: XMQDOKPKLKQR
Decrypted Message: NISHATMAHMUD
```

Playfair Cipher

PlayfairCipher.py

```python
# Playfair Cipher Encryption and Decryption

# Function to generate 5x5 Playfair cipher key matrix
def generate_key_matrix(key):
    key = key.upper().replace('J', 'I')  # Replace J with I (common in Playfair Cipher)
    key_matrix = []
    used_letters = set()

    for char in key:
        if char not in used_letters and char.isalpha():
            key_matrix.append(char)
            used_letters.add(char)

    # Add remaining letters to the matrix
    for char in "ABCDEFGHIKLMNOPQRSTUVWXYZ":  # 'J' is omitted
        if char not in used_letters:
            key_matrix.append(char)
            used_letters.add(char)

    # Return as a 5x5 matrix
    return [key_matrix[i:i+5] for i in range(0, 25, 5)]

# Function to split message into digraphs (pairs of letters)
def prepare_message(message):
    message = message.upper().replace('J', 'I')  # Replace J with I
    digraphs = []
    i = 0
    while i < len(message):
        char1 = message[i]
        if i + 1 < len(message):
            char2 = message[i + 1]
            if char1 != char2:
                digraphs.append(char1 + char2)
                i += 2
            else:
                digraphs.append(char1 + 'X')  # Insert X between repeated letters
                i += 1
        else:
            digraphs.append(char1 + 'X')  # Add X if the last letter is single
            i += 1
    return digraphs

# Function to find the position of a letter in the key matrix
def find_position(char, key_matrix):
    for row in range(5):
        for col in range(5):
            if key_matrix[row][col] == char:
                return row, col
    return None
```

PlayfairCipher.py

```python
# Encryption function
def encrypt(plaintext, key):
    key_matrix = generate_key_matrix(key)
    digraphs = prepare_message(plaintext)
    ciphertext = ""

    for digraph in digraphs:
        row1, col1 = find_position(digraph[0], key_matrix)
        row2, col2 = find_position(digraph[1], key_matrix)

        # Same row: Shift right
        if row1 == row2:
            ciphertext += key_matrix[row1][(col1 + 1) % 5]
            ciphertext += key_matrix[row2][(col2 + 1) % 5]
        # Same column: Shift down
        elif col1 == col2:
            ciphertext += key_matrix[(row1 + 1) % 5][col1]
            ciphertext += key_matrix[(row2 + 1) % 5][col2]
        # Rectangle swap
        else:
            ciphertext += key_matrix[row1][col2]
            ciphertext += key_matrix[row2][col1]

    return ciphertext

# Decryption function
def decrypt(ciphertext, key):
    key_matrix = generate_key_matrix(key)
    digraphs = [ciphertext[i:i+2] for i in range(0, len(ciphertext), 2)]
    plaintext = ""

    for digraph in digraphs:
        row1, col1 = find_position(digraph[0], key_matrix)
        row2, col2 = find_position(digraph[1], key_matrix)

        # Same row: Shift left
        if row1 == row2:
            plaintext += key_matrix[row1][(col1 - 1) % 5]
            plaintext += key_matrix[row2][(col2 - 1) % 5]
        # Same column: Shift up
        elif col1 == col2:
            plaintext += key_matrix[(row1 - 1) % 5][col1]
            plaintext += key_matrix[(row2 - 1) % 5][col2]
        # Rectangle swap
        else:
            plaintext += key_matrix[row1][col2]
            plaintext += key_matrix[row2][col1]

    return plaintext
```

PlayfairCipher.py

```
# Example Usage
key = "MONARCHY"
message = "nishatmahmud"
print("Original Message:", message)
ciphertext = encrypt(message, key)
print("Encrypted Message:", ciphertext)
decrypted_message = decrypt(ciphertext, key)
print("Decrypted Message:", decrypted_message)
```

Output

Output

```
Original Message: nishatmahmud
Encrypted Message: AGPBRSORCOZC
Decrypted Message: NISHATMAHMUD
```

RSA Encryption

RSAencryption.py

```
# Helper function: Extended Euclidean Algorithm to find the modular inverse
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    else:
        return x % m

# Helper function: Modular Exponentiation
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
```

RSAencryption.py

```python
        while exp > 0:
        if (exp % 2) == 1:  # If exp is odd, multiply base with result
            result = (result * base) % mod
        exp = exp >> 1  # Divide the exponent by 2
        base = (base * base) % mod  # Square the base
    return result

# Step 1: Key Generation
def generate_keys():
    # Choose two prime numbers (small values for simplicity, use larger primes in real use)
    p = 61
    q = 53
    n = p * q
    phi_n = (p - 1) * (q - 1)

    # Public exponent (commonly used value is 65537)
    e = 65537

    # Compute the private key 'd' (modular inverse of e modulo phi_n)
    d = mod_inverse(e, phi_n)

    # Public key is (e, n) and private key is (d, n)
    return e, d, n

# Step 2: Encryption (Public key is (e, n), plaintext is m)
def encrypt(m, e, n):
    return mod_exp(m, e, n)

# Step 3: Decryption (Private key is (d, n), ciphertext is c)
def decrypt(c, d, n):
    return mod_exp(c, d, n)

# Example Usage
if __name__ == "__main__":
    # Generate keys
    e, d, n = generate_keys()
    print(f"Public key (e, n): ({e}, {n}), Private key (d, n): ({d}, {n})")

    # Sample message (must be smaller than n)
    m = 42
    print(f"Original message: {m}")

    # Encrypt the message
    c = encrypt(m, e, n)
    print(f"Encrypted message (ciphertext): {c}")

    # Decrypt the ciphertext
    decrypted_message = decrypt(c, d, n)
    print(f"Decrypted message: {decrypted_message}")
```

Output

Output

```
Public key (e, n): (65537, 3233)
Private key (d, n): (2753, 3233)
Original message: 42
Encrypted message (ciphertext): 2557
Decrypted message: 42
```

Rabin Cryptosystem

RabinCryptosystem.py

```python
# Helper function: Extended Euclidean Algorithm to find the modular inverse
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    else:
        return x % m

# Helper function: Modular Exponentiation
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if (exp % 2) == 1:  # If exp is odd, multiply base with result
            result = (result * base) % mod
        exp = exp >> 1  # Divide the exponent by 2
        base = (base * base) % mod  # Square the base
    return result

# Step 1: Key Generation
def generate_keys():
    # Prime numbers p and q (small values for simplicity, use larger primes in real-world use)
    p = 7
    q = 11
    n = p * q
    return p, q, n

# Step 2: Encryption (Public key is n, plaintext is m)
def encrypt(m, n):
    return mod_exp(m, 2, n)
```

RabinCryptosystem.py

```python
# Step 3: Decryption (Private keys are p, q)
def decrypt(c, p, q, n):
    # Compute square roots modulo p and q
    mp = mod_exp(c, (p + 1) // 4, p)
    mq = mod_exp(c, (q + 1) // 4, q)

    # Use Chinese Remainder Theorem to get four possible plaintexts
    inv_q = mod_inverse(q, p)
    inv_p = mod_inverse(p, q)

    x1 = (mp * q * inv_q + mq * p * inv_p) % n
    x2 = (mp * q * inv_q - mq * p * inv_p) % n

    # Return the four possible solutions
    return x1, n - x1, x2, n - x2

# Example Usage
if __name__ == "__main__":
    # Generate keys
    p, q, n = generate_keys()
    print(f"Public key (n): {n}, Private keys (p, q): ({p}, {q})")

    # Sample message (must be smaller than n)
    m = 5
    print(f"Original message: {m}")

    # Encrypt the message
    c = encrypt(m, n)
    print(f"Encrypted message (ciphertext): {c}")

    # Decrypt the ciphertext
    possible_messages = decrypt(c, p, q, n)
    print(f"Decrypted possible messages: {possible_messages}")
```

Output

Output

```
Public key (n): 77
Private keys (p, q): (7, 11)
Original message: 5
Encrypted message (ciphertext): 25
Decrypted possible messages: (16, 61, 72, 5)
```

ElGamal Algorithm

ElGamalAlgorithm.py

```python
# Helper function: Modular Exponentiation
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if (exp % 2) == 1:  # If exp is odd, multiply base with result
            result = (result * base) % mod
        exp = exp >> 1  # Divide the exponent by 2
        base = (base * base) % mod  # Square the base
    return result

# Helper function: Extended Euclidean Algorithm to find the modular inverse
def extended_gcd(a, b):
    if b == 0:
        return a, 1, 0
    gcd, x1, y1 = extended_gcd(b, a % b)
    x = y1
    y = x1 - (a // b) * y1
    return gcd, x, y

def mod_inverse(a, m):
    gcd, x, _ = extended_gcd(a, m)
    if gcd != 1:
        raise ValueError("Modular inverse does not exist")
    else:
        return x % m

# Step 1: Key Generation
def generate_keys():
    # Choose a large prime number (small prime for simplicity, use large in real scenarios)
    p = 23
    g = 5   # Primitive root modulo p

    # Private key (x) is a random number between 1 and p-2
    x = 6  # Alice's private key (this should be kept secret)

    # Public key (y = g^x mod p)
    y = mod_exp(g, x, p)

    # Public key is (p, g, y) and private key is (x)
    return p, g, y, x

# Step 2: Encryption (Public key is (p, g, y), plaintext is m)
def encrypt(m, p, g, y):
    k = 15  # Random ephemeral key chosen by the sender (should be random each time)

    # Compute c1 = g^k mod p
    c1 = mod_exp(g, k, p)

    # Compute c2 = m * y^k mod p
    c2 = (m * mod_exp(y, k, p)) % p

    return c1, c2
```

ElGamalAlgorithm.py

```python
# Step 3: Decryption (Private key is x, ciphertext is (c1, c2))
def decrypt(c1, c2, p, x):
    # Compute the shared secret c1^x mod p
    s = mod_exp(c1, x, p)

    # Compute the modular inverse of s mod p
    s_inv = mod_inverse(s, p)

    # Decrypt the message m = c2 * s_inv mod p
    m = (c2 * s_inv) % p

    return m

# Example Usage
if __name__ == "__main__":
    # Generate keys
    p, g, y, x = generate_keys()
    print(f"Public key (p, g, y): ({p}, {g}, {y}), Private key (x): {x}")

    # Sample message (must be smaller than p)
    m = 13
    print(f"Original message: {m}")

    # Encrypt the message
    c1, c2 = encrypt(m, p, g, y)
    print(f"Encrypted message (c1, c2): ({c1}, {c2})")

    # Decrypt the ciphertext
    decrypted_message = decrypt(c1, c2, p, x)
    print(f"Decrypted message: {decrypted_message}")
```

Output

Output

```
Public key (p, g, y): (23, 5, 8)
Private key (x): 6
Original message: 13
Encrypted message (c1, c2): (19, 3)
Decrypted message: 13
```

Diffie-Hellman Key Exchange

DiffieHellmanKeyExchange.py

```python
# Helper function: Modular Exponentiation
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod
    while exp > 0:
        if (exp % 2) == 1:  # If exp is odd, multiply base with result
            result = (result * base) % mod
        exp = exp >> 1  # Divide the exponent by 2
        base = (base * base) % mod  # Square the base
    return result

# Diffie-Hellman Key Exchange
def diffie_hellman():
    # Step 1: Agree on a large prime number (p) and a base (g)
    p = 23  # A small prime number for simplicity, use a large prime in real scenarios
    g = 5   # A primitive root modulo p

    print(f"Publicly shared values: p = {p}, g = {g}")

    # Step 2: Alice chooses a private key
    a_private = 6  # Alice's private key (this should be kept secret)
    print(f"Alice's private key: {a_private}")

    # Alice computes her public key and sends it to Bob
    a_public = mod_exp(g, a_private, p)
    print(f"Alice's public key: {a_public}")

    # Step 3: Bob chooses a private key
    b_private = 15  # Bob's private key (this should be kept secret)
    print(f"Bob's private key: {b_private}")

    # Bob computes his public key and sends it to Alice
    b_public = mod_exp(g, b_private, p)
    print(f"Bob's public key: {b_public}")

    # Step 4: Alice and Bob compute the shared secret
    # Alice computes the shared secret using Bob's public key and her private key
    shared_secret_alice = mod_exp(b_public, a_private, p)
    print(f"Alice's computed shared secret: {shared_secret_alice}")

    # Bob computes the shared secret using Alice's public key and his private key
    shared_secret_bob = mod_exp(a_public, b_private, p)
    print(f"Bob's computed shared secret: {shared_secret_bob}")

    # The shared secrets should be the same for both Alice and Bob
    if shared_secret_alice == shared_secret_bob:
        print(f"Shared secret successfully computed: {shared_secret_alice}")
    else:
        print("Error: Shared secrets do not match!")

# Run the Diffie-Hellman key exchange example
if __name__ == "__main__":
    diffie_hellman()
```

Output

Output

```
Publicly shared values: p = 23, g = 5
Alice's private key: 6
Alice's public key: 8
Bob's private key: 15
Bob's public key: 19
Alice's computed shared secret: 2
Bob's computed shared secret: 2
Shared secret successfully computed: 2
```

Data Encryption Standard

DataEncryptionStandard.py

```python
# Initial Permutation (IP) table
IP = [
    58, 50, 42, 34, 26, 18, 10, 2,
    60, 52, 44, 36, 28, 20, 12, 4,
    62, 54, 46, 38, 30, 22, 14, 6,
    64, 56, 48, 40, 32, 24, 16, 8,
    57, 49, 41, 33, 25, 17, 9, 1,
    59, 51, 43, 35, 27, 19, 11, 3,
    61, 53, 45, 37, 29, 21, 13, 5,
    63, 55, 47, 39, 31, 23, 15, 7
]

# Final Permutation (FP) table (Inverse of IP)
FP = [
    40, 8, 48, 16, 56, 24, 64, 32,
    39, 7, 47, 15, 55, 23, 63, 31,
    38, 6, 46, 14, 54, 22, 62, 30,
    37, 5, 45, 13, 53, 21, 61, 29,
    36, 4, 44, 12, 52, 20, 60, 28,
    35, 3, 43, 11, 51, 19, 59, 27,
    34, 2, 42, 10, 50, 18, 58, 26,
    33, 1, 41, 9, 49, 17, 57, 25
]

# A simple key for illustration (64-bit binary string)
key = "1010101010111011000010010001100000100111001101101100110011011101"

# Function to apply permutation
def apply_permutation(input_text, table):
    return ''.join([input_text[i - 1] for i in table])

# Simple XOR function for two binary strings
def xor(a, b):
    return ''.join([str(int(a[i]) ^ int(b[i])) for i in range(len(a))])
```

DataEncryptionStandard.py

```
# Simplified encryption function
def encrypt(plain_text):
    # Step 1: Initial Permutation
    permuted_text = apply_permutation(plain_text, IP)

    # Step 2: Split into two halves
    left = permuted_text[:32]
    right = permuted_text[32:]

    # Step 3: Simplified round function (normally 16 rounds, simplified here)
    for _ in range(16):
        temp = right
        right = xor(left, key[:32])  # XOR left half with the key (simplified)
        left = temp

    # Step 4: Combine left and right halves
    combined_text = left + right

    # Step 5: Final Permutation
    cipher_text = apply_permutation(combined_text, FP)

    return cipher_text

# Main execution
if __name__ == "__main__":
    # Example plain text (64-bit binary string)
    plain_text = "0110000101100010011000110110010001100101011001100110011101101000"  #
"abcdefgh" in binary

    print("Plain Text: ", plain_text)

    # Encrypt the plain text
    cipher_text = encrypt(plain_text)

    print("Cipher Text: ", cipher_text)
```

Output

Output

```
Plain Text:  0110000101100010011000110110010001100101011001100110011101101000
Cipher Text:  0110000101100010011000110110010001100101011001100110011101101000
```

Advanced Encryption Standard

AdvancedEncryptionStandard.py

```
# Example S-Box (substitution box used in AES)
S_BOX = [
    [0x63, 0x7C, 0x77, 0x7B, 0xF2, 0x6B, 0x6F, 0xC5, 0x30, 0x01, 0x67, 0x2B, 0xFE, 0xD7, 0xAB,
0x76],
    [0xCA, 0x82, 0xC9, 0x7D, 0xFA, 0x59, 0x47, 0xF0, 0xAD, 0xD4, 0xA2, 0xAF, 0x9C, 0xA4, 0x72,
0xC0],
    [0xB7, 0xFD, 0x93, 0x26, 0x36, 0x3F, 0xF7, 0xCC, 0x34, 0xA5, 0xE5, 0xF1, 0x71, 0xD8, 0x31,
0x15],
    [0x04, 0xC7, 0x23, 0xC3, 0x18, 0x96, 0x05, 0x9A, 0x07, 0x12, 0x80, 0xE2, 0xEB, 0x27, 0xB2,
0x75],
    [0x09, 0x83, 0x2C, 0x1A, 0x1B, 0x6E, 0x5A, 0xA0, 0x52, 0x3B, 0xD6, 0xB3, 0x29, 0xE3, 0x2F,
0x84],
    [0x53, 0xD1, 0x00, 0xED, 0x20, 0xFC, 0xB1, 0x5B, 0x6A, 0xCB, 0xBE, 0x39, 0x4A, 0x4C, 0x58,
0xCF],
    [0xD0, 0xEF, 0xAA, 0xFB, 0x43, 0x4D, 0x33, 0x85, 0x45, 0xF9, 0x02, 0x7F, 0x50, 0x3C, 0x9F,
0xA8],
    [0x51, 0xA3, 0x40, 0x8F, 0x92, 0x9D, 0x38, 0xF5, 0xBC, 0xB6, 0xDA, 0x21, 0x10, 0xFF, 0xF3,
0xD2],
    [0xCD, 0x0C, 0x13, 0xEC, 0x5F, 0x97, 0x44, 0x17, 0xC4, 0xA7, 0x7E, 0x3D, 0x64, 0x5D, 0x19,
0x73],
    [0x60, 0x81, 0x4F, 0xDC, 0x22, 0x2A, 0x90, 0x88, 0x46, 0xEE, 0xB8, 0x14, 0xDE, 0x5E, 0x0B,
0xDB],
    [0xE0, 0x32, 0x3A, 0x0A, 0x49, 0x06, 0x24, 0x5C, 0xC2, 0xD3, 0xAC, 0x62, 0x91, 0x95, 0xE4,
0x79],
    [0xE7, 0xC8, 0x37, 0x6D, 0x8D, 0xD5, 0x4E, 0xA9, 0x6C, 0x56, 0xF4, 0xEA, 0x65, 0x7A, 0xAE,
0x08],
    [0xBA, 0x78, 0x25, 0x2E, 0x1C, 0xA6, 0xB4, 0xC6, 0xE8, 0xDD, 0x74, 0x1F, 0x4B, 0xBD, 0x8B,
0x8A],
    [0x70, 0x3E, 0xB5, 0x66, 0x48, 0x03, 0xF6, 0x0E, 0x61, 0x35, 0x57, 0xB9, 0x86, 0xC1, 0x1D,
0x9E],
    [0xE1, 0xF8, 0x98, 0x11, 0x69, 0xD9, 0x8E, 0x94, 0x9B, 0x1E, 0x87, 0xE9, 0xCE, 0x55, 0x28,
0xDF],
    [0x8C, 0xA1, 0x89, 0x0D, 0xBF, 0xE6, 0x42, 0x68, 0x41, 0x99, 0x2D, 0x0F, 0xB0, 0x54, 0xBB,
0x16]
]

# A simplified key for illustration (16 bytes or 128 bits)
key = "10101010101110110000100100011000"

# Apply S-box substitution (similar to SubBytes in AES)
def substitute_bytes(state):
    substituted = []
    for byte in state:
        row = (byte >> 4) & 0x0F  # High nibble
        col = byte & 0x0F  # Low nibble
        substituted.append(S_BOX[row][col])
    return substituted

# Shift rows (similar to ShiftRows in AES)
def shift_rows(state):
    state[1], state[5], state[9], state[13] = state[5], state[9], state[13], state[1]  # 2nd
row shift
    state[2], state[6], state[10], state[14] = state[10], state[14], state[2], state[6]  # 3rd
row shift
    state[3], state[7], state[11], state[15] = state[15], state[3], state[7], state[11]  # 4th
row shift
    return state
```

AdvancedEncryptionStandard.py

```python
# Add round key (XOR with the key)
def add_round_key(state, round_key):
    return [state[i] ^ round_key[i] for i in range(16)]

# Simplified AES encryption function (1 round for simplicity)
def aes_encrypt(plain_text):
    # Step 1: Convert plain_text to a state array (16 bytes)
    state = [ord(char) for char in plain_text]

    # Step 2: Add round key (initial key addition)
    state = add_round_key(state, [ord(k) for k in key])

    # Step 3: Substitute bytes using S-box
    state = substitute_bytes(state)

    # Step 4: Shift rows
    state = shift_rows(state)

    # Step 5: Add round key again (for simplification, using the same key)
    state = add_round_key(state, [ord(k) for k in key])

    # Return the encrypted state as the final cipher text (in hex form for readability)
    cipher_text = ''.join(format(x, '02x') for x in state)
    return cipher_text

# Main execution
if __name__ == "__main__":
    # Example plain text (16 characters)
    plain_text = "abcdefghijklmnoq"  # Must be 16 characters (128-bit block size)

    print("Plain Text: ", plain_text)

    # Encrypt the plain text
    cipher_text = aes_encrypt(plain_text)

    print("Cipher Text: ", cipher_text)
```

Output

Output

```
Plain Text:  abcdefghijklmnoq
Cipher Text:  62818f39118e69105b68315b7b30807d
```