CNS Codes

1.Write a python program that contains a string (char pointer) with a value „Hello world‟. The program should XOR each character in this string with 0 and displays the result.

```
# Define the string
input_string = "Hello world"

# XOR each character in the string with 0
result = ''.join([chr(ord(char) ^ 0) for char in input_string])

# Display the result
print("Original string:", input_string)
print("Result after XOR with 0:", result)
```

2. Write a python program that contains a string (char pointer) with a value „Hello world‟. The program should AND or and XOR each character in this string with 127 and display the result.

```
# Define the string
input_string = "Hello world"

# Initialize empty strings to hold the results
and_result = ''
```

```
xor_result = ''

# Perform AND and XOR operations on each character
for char in input_string:
    and_char = chr(ord(char) & 127)  # AND with 127
    xor_char = chr(ord(char) ^ 127)  # XOR with 127
    and_result += and_char
    xor_result += xor_char

# Display the results
print("Original string: ", input_string)
print("Result after AND with 127: ", and_result)
print("Result after XOR with 127: ", xor_result)
```

3.(a) Caesar cipher:

```
def caesar_cipher(text, shift, mode='encrypt'):
    result = ''

    # Adjust the shift value for decryption
    if mode == 'decrypt':
        shift = -shift

    for char in text:
        if char.isalpha():  # Check if the character is a letter
            # Determine the ASCII offset based on uppercase or lowercase
```

```
            ascii_offset = ord('A') if char.isupper() else ord('a')
            # Perform the shift and wrap around the alphabet
            shifted_char = chr((ord(char) - ascii_offset + shift) % 26 + ascii_offset)
            result += shifted_char
        else:
            # Non-alphabetic characters remain unchanged
            result += char

    return result

# Example usage
if __name__ == "__main__":
    original_text = "Hello, World!"
    shift_value = 3

    # Encrypt the message
    encrypted_text = caesar_cipher(original_text, shift_value, mode='encrypt')
    print("Encrypted:", encrypted_text)

    # Decrypt the message
    decrypted_text = caesar_cipher(encrypted_text, shift_value, mode='decrypt')
    print("Decrypted:", decrypted_text)
```

3.(b) Substitution cipher

```
import string
```

```
def create_substitution_alphabet(key):
    # Create a substitution alphabet based on the provided key
    alphabet = string.ascii_lowercase
    key = key.lower()

    # Remove duplicates and keep only unique characters from the key
    key_unique = ''.join(sorted(set(key), key=key.index))

    # Create the substitution alphabet
    substitution_alphabet = key_unique + ''.join(sorted(set(alphabet) - set(key_unique)))

    return substitution_alphabet

def substitution_cipher(text, key, mode='encrypt'):
    result = ''
    substitution_alphabet = create_substitution_alphabet(key)
    alphabet = string.ascii_lowercase

    # Create a mapping for encryption and decryption
    if mode == 'encrypt':
        mapping = str.maketrans(alphabet, substitution_alphabet)
    else:  # decrypt mode
        mapping = str.maketrans(substitution_alphabet, alphabet)

    # Translate the text using the mapping
    result = text.translate(mapping)
```

```python
    return result

# Example usage
if __name__ == "__main__":
    original_text = "Hello, World!"
    key = "cipher"  # This is the substitution key

    # Encrypt the message
    encrypted_text = substitution_cipher(original_text, key, mode='encrypt')
    print("Encrypted:", encrypted_text)

    # Decrypt the message
    decrypted_text = substitution_cipher(encrypted_text, key, mode='decrypt')
    print("Decrypted:", decrypted_text)
```

3.(c) Hill Cipher

```python
import numpy as np

def matrix_mod_inverse(matrix, modulus):
    # Compute the determinant and its modular inverse
    det = int(np.round(np.linalg.det(matrix)))  # Determinant
    det_inv = pow(det, -1, modulus)  # Modular inverse of the determinant

    # Calculate the matrix of minors
```

```
    minors = np.linalg.inv(matrix).T * det
    cofactors = minors.round().astype(int) % modulus  # Cofactor matrix
    adjugate = cofactors.T  # Adjugate matrix

    # Inverse matrix
    inverse_matrix = (det_inv * adjugate) % modulus
    return inverse_matrix.astype(int)

def encrypt(plaintext, key_matrix):
    # Preprocess the plaintext
    plaintext = plaintext.replace(" ", "").lower()  # Remove spaces and convert to lowercase
    n = key_matrix.shape[0]  # Size of the key matrix
    # Pad plaintext if necessary
    while len(plaintext) % n != 0:
        plaintext += 'x'  # Padding with 'x'

    # Convert letters to numbers
    plaintext_numbers = [ord(char) - ord('a') for char in plaintext]
    plaintext_matrix = np.array(plaintext_numbers).reshape(-1, n)

    # Encrypt the plaintext
    ciphertext_matrix = (plaintext_matrix @ key_matrix) % 26
    ciphertext = ''.join(chr(num + ord('a')) for num in ciphertext_matrix.flatten())

    return ciphertext

def decrypt(ciphertext, key_matrix):
```

```
    # Calculate the inverse of the key matrix
    inverse_key_matrix = matrix_mod_inverse(key_matrix, 26)

    # Convert letters to numbers
    ciphertext_numbers = [ord(char) - ord('a') for char in ciphertext]
    n = key_matrix.shape[0]  # Size of the key matrix
    ciphertext_matrix = np.array(ciphertext_numbers).reshape(-1, n)

    # Decrypt the ciphertext
    plaintext_matrix = (ciphertext_matrix @ inverse_key_matrix) % 26
    plaintext = ''.join(chr(num + ord('a')) for num in plaintext_matrix.flatten())

    return plaintext

# Example usage
if __name__ == "__main__":
    key_matrix = np.array([[6, 24, 1],
                           [13, 16, 10],
                           [20, 17, 15]])  # Example 3x3 key matrix
    original_text = "hello world"

    # Encrypt the message
    encrypted_text = encrypt(original_text, key_matrix)
    print("Encrypted:", encrypted_text)

    # Decrypt the message
    decrypted_text = decrypt(encrypted_text, key_matrix)
```

```
    print("Decrypted:", decrypted_text)
```

3.(d) playfair cipher

```
def create_playfair_matrix(key):
    # Remove duplicates and handle 'J' as 'I'
    key = key.replace("j", "i").replace("J", "I").lower()
    key = ''.join(sorted(set(key), key=key.index))  # Unique characters
    alphabet = "abcdefghiklmnopqrstuvwxyz"  # 'j' is omitted
    matrix = key + ''.join(c for c in alphabet if c not in key)

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

def find_position(char, matrix):
    for row in range(5):
        for col in range(5):
            if matrix[row][col] == char:
                return row, col
    return None

def prepare_text(text):
    text = text.replace(" ", "").replace("j", "i").lower()  # Remove spaces and treat 'j' as 'i'
    prepared = []
    i = 0
    while i < len(text):
        if i + 1 < len(text) and text[i] == text[i + 1]:  # Check for double letters
            prepared.append(text[i] + 'x')  # Insert 'x' between duplicates
```

```
            i += 1
        else:
            prepared.append(text[i:i + 2])  # Take two characters at a time
            i += 2
    return prepared

def encrypt(plaintext, key):
    matrix = create_playfair_matrix(key)
    digraphs = prepare_text(plaintext)
    ciphertext = ""

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

        if row1 == row2:  # Same row
            ciphertext += matrix[row1][(col1 + 1) % 5] + matrix[row2][(col2 + 1) % 5]
        elif col1 == col2:  # Same column
            ciphertext += matrix[(row1 + 1) % 5][col1] + matrix[(row2 + 1) % 5][col2]
        else:  # Rectangle case
            ciphertext += matrix[row1][col2] + matrix[row2][col1]

    return ciphertext

def decrypt(ciphertext, key):
    matrix = create_playfair_matrix(key)
    digraphs = prepare_text(ciphertext)
```

```
    plaintext = ""

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

        if row1 == row2:  # Same row
            plaintext += matrix[row1][(col1 - 1) % 5] + matrix[row2][(col2 - 1) % 5]
        elif col1 == col2:  # Same column
            plaintext += matrix[(row1 - 1) % 5][col1] + matrix[(row2 - 1) % 5][col2]
        else:  # Rectangle case
            plaintext += matrix[row1][col2] + matrix[row2][col1]

    return plaintext.replace("x", "")  # Remove padding 'x'

# Example usage
if __name__ == "__main__":
    key = "playfair example"
    original_text = "hide the gold in the tree stump"

    # Encrypt the message
    encrypted_text = encrypt(original_text, key)
    print("Encrypted:", encrypted_text)

    # Decrypt the message
    decrypted_text = decrypt(encrypted_text, key)
    print("Decrypted:", decrypted_text)
```

4.Write a python program to implement the DES algorithm logic.

```
from Crypto.Cipher import DES

from secrets import token_bytes


# Function to pad the text to ensure it's a multiple of 8 bytes

def pad(text):

    while len(text) % 8 != 0:

        text += ' '

    return text


# Function to generate a random 8-byte key for DES

def generate_key():

    return token_bytes(8)


# Function to encrypt a plaintext message using DES

def encrypt(plaintext, key):

    des = DES.new(key, DES.MODE_ECB)  # Create a new DES cipher object

    padded_text = pad(plaintext)      # Pad the text if necessary

    ciphertext = des.encrypt(padded_text.encode('utf-8'))  # Encrypt the text

    return ciphertext


# Function to decrypt a ciphertext message using DES

def decrypt(ciphertext, key):

    des = DES.new(key, DES.MODE_ECB)  # Create a new DES cipher object

    decrypted_text = des.decrypt(ciphertext).decode('utf-8')  # Decrypt the text

    return decrypted_text.strip()  # Remove the padding (if any)
```

```python
# Example usage of DES algorithm
if __name__ == '__main__':
    key = generate_key()  # Generate a random 8-byte key
    print("Key:", key.hex())

    plaintext = "HELLO DES"
    print("Original text:", plaintext)

    # Encrypt the plaintext
    ciphertext = encrypt(plaintext, key)
    print("Encrypted text:", ciphertext.hex())

    # Decrypt the ciphertext
    decrypted_text = decrypt(ciphertext, key)
    print("Decrypted text:", decrypted_text)
```

5.Write a python program to implement the Blowfish algorithm logic.

```
from Crypto.Cipher import Blowfish

from Crypto.Util.Padding import pad, unpad

from secrets import token_bytes


# Function to generate a random key for Blowfish encryption

def generate_key():

    return token_bytes(16)  # Blowfish allows keys between 4 and 56 bytes (32 to 448 bits)


# Function to encrypt the plaintext using Blowfish

def encrypt(plaintext, key):

    cipher = Blowfish.new(key, Blowfish.MODE_ECB)  # ECB mode for Blowfish

    padded_text = pad(plaintext.encode('utf-8'), Blowfish.block_size)  # Pad plaintext to be a
multiple of the block size

    ciphertext = cipher.encrypt(padded_text)  # Encrypt the padded text

    return ciphertext


# Function to decrypt the ciphertext using Blowfish

def decrypt(ciphertext, key):

    cipher = Blowfish.new(key, Blowfish.MODE_ECB)

    decrypted_text = unpad(cipher.decrypt(ciphertext), Blowfish.block_size)  # Decrypt and
unpad the text

    return decrypted_text.decode('utf-8')


# Example usage of Blowfish algorithm

if __name__ == '__main__':

    key = generate_key()  # Generate a random key
```

```
    print("Key:", key.hex())

    plaintext = "HELLO BLOWFISH"
    print("Original text:", plaintext)

    # Encrypt the plaintext
    ciphertext = encrypt(plaintext, key)
    print("Encrypted text:", ciphertext.hex())

    # Decrypt the ciphertext
    decrypted_text = decrypt(ciphertext, key)
    print("Decrypted text:", decrypted_text)
```

6.Write a Python program to implement the Rijndael algorithm logic.

```
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad, unpad
from secrets import token_bytes

# Function to generate a random key for AES (Rijndael algorithm)
def generate_key(key_size=32):
    return token_bytes(key_size)  # AES supports 16 (128-bit), 24 (192-bit), or 32 bytes (256-bit)

# Function to encrypt plaintext using AES
def encrypt(plaintext, key):
```

```
    cipher = AES.new(key, AES.MODE_CBC)  # AES in CBC mode with a random IV
    iv = cipher.iv  # Get the initialization vector (IV)
    padded_text = pad(plaintext.encode('utf-8'), AES.block_size)  # Pad plaintext to match AES block size
    ciphertext = cipher.encrypt(padded_text)  # Encrypt the plaintext
    return iv + ciphertext  # Prepend the IV to the ciphertext for decryption

# Function to decrypt ciphertext using AES
def decrypt(ciphertext, key):
    iv = ciphertext[:16]  # Extract the IV from the ciphertext
    actual_ciphertext = ciphertext[16:]  # Get the actual ciphertext
    cipher = AES.new(key, AES.MODE_CBC, iv)  # Recreate the cipher object with the extracted IV
    decrypted_text = unpad(cipher.decrypt(actual_ciphertext), AES.block_size)  # Decrypt and unpad the text
    return decrypted_text.decode('utf-8')

# Example usage of the Rijndael (AES) algorithm
if __name__ == '__main__':
    key = generate_key(32)  # Generate a random 256-bit key
    print("Key:", key.hex())

    plaintext = "HELLO AES (RIJNDAEL)"
    print("Original text:", plaintext)

    # Encrypt the plaintext
    ciphertext = encrypt(plaintext, key)
    print("Encrypted text (hex):", ciphertext.hex())
```

```
    # Decrypt the ciphertext
    decrypted_text = decrypt(ciphertext, key)
    print("Decrypted text:", decrypted_text)
```

7.Write the RC4 logic in Java Using Java cryptography.

```
import java.util.*;
public class RC4 {
    static int n=3;
    static String plain_text="001010010010";
    static String key="101001000001";
    static List<Integer> S=new ArrayList<>();
    static List<Integer> key_list=new ArrayList<>();
    static List<Integer> pt=new ArrayList<>();
    static List<Integer> key_stream=new ArrayList<>();
    static List<Integer> cipher_text=new ArrayList<>();
    static List<Integer> original_text=new ArrayList<>();
    public static void main(String[] args) {
        encryption();
        System.out.println("---------------------------------------------------------");
        decryption();
    }
    // Function for encryption
    public static void encryption() {
        System.out.println("Plain text : "+plain_text);
        System.out.println("Key : "+key);
        System.out.println("n : "+n);
        // The initial state vector array
        for (int i=0;i<Math.pow(2,n);i++) {
            S.add(i);
```

```
    }
    System.out.println("S : "+S);
    key_list=convertToDecimal(key);
    pt=convertToDecimal(plain_text);
    System.out.println("Plain text ( in array form ): "+pt);
    // Making key_stream equal to length of state vector
    int diff=S.size()-key_list.size();
    if (diff!=0) {
        for (int i=0;i<diff;i++) {
            key_list.add(key_list.get(i));
        }
    }
    System.out.println("Key list : "+key_list);
    // Perform the KSA algorithm
    KSA();
    // Perform PGRA algorithm
    PGRA();
    // Performing XOR between generated key stream and plain text
    XOR();
}
// Function for decryption of data
public static void decryption() {
    S.clear();
    key_list.clear();
    pt.clear();
    key_stream.clear();
    // The initial state vector array
    for (int i=0;i<Math.pow(2,n);i++) {
        S.add(i);
    }
    key_list=convertToDecimal(key);
```

```
    pt=convertToDecimal(plain_text);
    // Making key_stream equal to length of state vector
    int diff=S.size()-key_list.size();
    if (diff!=0) {
        for (int i=0;i<diff;i++) {
            key_list.add(key_list.get(i));
        }
    }
    // KSA algorithm
    KSA();
    // Perform PRGA algorithm
    PGRA();
    // Perform XOR between generated key stream and cipher text
    do_XOR();
}
// KSA algorithm
public static void KSA() {
    int j=0;
    int N=S.size();
    // Iterate over the range [0, N]
    for (int i=0;i<N;i++) {
        j=(j+S.get(i)+key_list.get(i))%N;
        // Update S[i] and S[j]
        Collections.swap(S,i,j);
        System.out.println(i+" "+S);
    }
    System.out.println("The initial permutation array is : "+S);
}
// PGRA algorithm
public static void PGRA() {
    int N=S.size();
```

```
    int i=0,j=0;
    // Iterate over [0, length of pt]
    for (int k=0;k<pt.size();k++) {
        i=(i+1)%N;
        j=(j+S.get(i))%N;
        // Update S[i] and S[j]
        Collections.swap(S,i,j);
        System.out.println(k+" "+S);
        int t=(S.get(i)+S.get(j))%N;
        key_stream.add(S.get(t));
    }
    // Print the key stream
    System.out.println("Key stream : "+key_stream);
}
// Perform XOR between generated key stream and plain text
public static void XOR() {
    for (int i=0;i<pt.size();i++) {
        int c=key_stream.get(i)^pt.get(i);
        cipher_text.add(c);
    }
    // Convert the encrypted text to bits form
    String encrypted_to_bits="";
    for (int i:cipher_text) {
        encrypted_to_bits+=String.format("%0"+n+"d",Integer.parseInt(Integer.toBinaryString(i)));
    }
    System.out.println("Cipher text : "+encrypted_to_bits);
}
// Perform XOR between generated key stream and cipher text
public static void do_XOR() {
    for (int i=0;i<cipher_text.size();i++) {
        int p=key_stream.get(i)^cipher_text.get(i);
```

```java
            original_text.add(p);
        }
        // Convert the decrypted text to the bits form
        String decrypted_to_bits="";
        for (int i:original_text) {
            decrypted_to_bits+=String.format("%0"+n+"d",Integer.parseInt(Integer.toBinaryString(i)));
        }
        System.out.println("Decrypted text : "+decrypted_to_bits);
    }
    // Convert to decimal
    public static List<Integer> convertToDecimal(String input) {
        List<String> list=new ArrayList<>();
        List<Integer> decimalList=new ArrayList<>();
        for (int i=0;i<input.length();i+=n) {
            list.add(input.substring(i,Math.min(input.length(),i+n)));
        }
        for (String s:list) {
            decimalList.add(Integer.parseInt(s,2));
        }
        return decimalList;
    }
}
```

In python:

```python
class RC4:
    def __init__(self, n, plain_text, key):
        self.n = n
```

```
        self.plain_text = plain_text
        self.key = key
        self.S = []
        self.key_list = []
        self.pt = []
        self.key_stream = []
        self.cipher_text = []
        self.original_text = []

    def encryption(self):
        print("Plain text: " + self.plain_text)
        print("Key: " + self.key)
        print("n: " + str(self.n))

        # The initial state vector array
        self.S = list(range(2 ** self.n))
        print("S: " + str(self.S))

        self.key_list = self.convert_to_decimal(self.key)
        self.pt = self.convert_to_decimal(self.plain_text)
        print("Plain text (in array form): " + str(self.pt))

        # Making key_list equal to length of state vector
        diff = len(self.S) - len(self.key_list)
        if diff != 0:
            self.key_list.extend(self.key_list[:diff])
        print("Key list: " + str(self.key_list))
```

```
        # Perform the KSA algorithm
        self.KSA()

        # Perform PRGA algorithm
        self.PGRA()

        # Performing XOR between generated key stream and plain text
        self.XOR()

    def decryption(self):
        # Resetting lists for decryption
        self.S = []
        self.key_list = []
        self.pt = []
        self.key_stream = []

        # The initial state vector array
        self.S = list(range(2 ** self.n))
        self.key_list = self.convert_to_decimal(self.key)
        self.pt = self.convert_to_decimal(self.plain_text)

        # Making key_stream equal to length of state vector
        diff = len(self.S) - len(self.key_list)
        if diff != 0:
            self.key_list.extend(self.key_list[:diff])
```

```
        # KSA algorithm
        self.KSA()

        # Perform PRGA algorithm
        self.PGRA()

        # Perform XOR between generated key stream and cipher text
        self.do_XOR()

    def KSA(self):
        j = 0
        N = len(self.S)

        # Iterate over the range [0, N]
        for i in range(N):
            j = (j + self.S[i] + self.key_list[i]) % N
            # Update S[i] and S[j]
            self.S[i], self.S[j] = self.S[j], self.S[i]
            print(f"{i} {self.S}")

        print("The initial permutation array is: " + str(self.S))

    def PGRA(self):
        N = len(self.S)
        i = 0
        j = 0
```

```
    # Iterate over [0, length of pt]
    for k in range(len(self.pt)):
        i = (i + 1) % N
        j = (j + self.S[i]) % N
        # Update S[i] and S[j]
        self.S[i], self.S[j] = self.S[j], self.S[i]
        print(f"{k} {self.S}")
        t = (self.S[i] + self.S[j]) % N
        self.key_stream.append(self.S[t])

    # Print the key stream
    print("Key stream: " + str(self.key_stream))

def XOR(self):
    for i in range(len(self.pt)):
        c = self.key_stream[i] ^ self.pt[i]
        self.cipher_text.append(c)

    # Convert the encrypted text to bits form
    encrypted_to_bits = ''.join(format(i, f'0{self.n}b') for i in self.cipher_text)
    print("Cipher text: " + encrypted_to_bits)

def do_XOR(self):
    for i in range(len(self.cipher_text)):
        p = self.key_stream[i] ^ self.cipher_text[i]
        self.original_text.append(p)
```

```
        # Convert the decrypted text to bits form
        decrypted_to_bits = ''.join(format(i, f'0{self.n}b') for i in self.original_text)
        print("Decrypted text: " + decrypted_to_bits)

    def convert_to_decimal(self, input_str):
        decimal_list = []
        for i in range(0, len(input_str), self.n):
            decimal_list.append(int(input_str[i:i + self.n], 2))
        return decimal_list


# Example usage
if __name__ == "__main__":
    n = 3
    plain_text = "001010010010"
    key = "101001000001"

    rc4 = RC4(n, plain_text, key)
    rc4.encryption()
    print("--------------------------------------------------------")
    rc4.decryption()
```

8.Write a python program to implement RSA algorithm.

```
import random
from math import gcd
```

```
# Function to compute modular inverse
def mod_inverse(e, phi):
    d = 0
    x1, x2, x3 = 1, 0, phi
    y1, y2, y3 = 0, 1, e
    while y3 != 1:
        q = x3 // y3
        t1, t2, t3 = x1 - q * y1, x2 - q * y2, x3 - q * y3
        x1, x2, x3 = y1, y2, y3
        y1, y2, y3 = t1, t2, t3
    return y2 % phi

# Function to perform modular exponentiation
def mod_exp(base, exp, mod):
    result = 1
    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp //= 2
    return result

# RSA Key Generation
def generate_keypair(p, q):
    n = p * q
    phi = (p - 1) * (q - 1)
```

```
    # Choose e such that 1 < e < phi and gcd(e, phi) = 1
    e = random.randrange(2, phi)
    while gcd(e, phi) != 1:
        e = random.randrange(2, phi)

    # Compute d (modular inverse of e)
    d = mod_inverse(e, phi)

    return ((e, n), (d, n))

# Encryption function
def encrypt(public_key, plaintext):
    e, n = public_key
    encrypted_message = [mod_exp(ord(char), e, n) for char in plaintext]
    return encrypted_message

# Decryption function
def decrypt(private_key, ciphertext):
    d, n = private_key
    decrypted_message = ''.join([chr(mod_exp(char, d, n)) for char in ciphertext])
    return decrypted_message

# Example of RSA Algorithm
if __name__ == '__main__':
    p = 61  # Example prime number
    q = 53  # Example prime number
```

```
public_key, private_key = generate_keypair(p, q)

print("Public Key:", public_key)
print("Private Key:", private_key)

message = "HELLO"
print("Original message:", message)

encrypted_msg = encrypt(public_key, message)
print("Encrypted message:", encrypted_msg)

decrypted_msg = decrypt(private_key, encrypted_msg)
print("Decrypted message:", decrypted_msg)
```