#1. Distance Vector Routing Protocol Implementation

```
def get_user_input():
    num_nodes = int(input("Enter the number of nodes: "))
    nodes = []
    print("Enter node names:")
    for _ in range(num_nodes):
        nodes.append(input().strip())
    print("Enter the cost matrix (use -1 for no direct path):")
    cost_matrix = []
    for i in range(num_nodes):
        row = list(map(int, input().split()))
        cost_matrix.append(row)  # Fixed indentation issue
    return num_nodes, nodes, cost_matrix
def distance_vector_routing(num_nodes, nodes, cost_matrix):
    # Initialize distance table with infinite values
    distance_table = [[float('inf')] * num_nodes for _ in range(num_nodes)]
    # Copy cost matrix values into the distance table
    for i in range(num_nodes):
        for j in range(num_nodes):
            if cost_matrix[i][j] != -1:
                distance_table[i][j] = cost_matrix[i][j]
    # Set self-distance to zero
    for i in range(num_nodes):
        distance_table[i][i] = 0
    # Bellman-Ford Algorithm (Distance Vector Routing)
    updated = True
    while updated:
        updated = False
```

```
        for i in range(num_nodes):
            for j in range(num_nodes):
                for k in range(num_nodes):
                    if distance_table[i][j] > distance_table[i][k] + distance_table[k][j]:
                        distance_table[i][j] = distance_table[i][k] + distance_table[k][j]
                        updated = True  # Continue updating until no changes
    return distance_table
def print_routing_table(nodes, distance_table):
    print("\nFinal Distance Vector Table:")
    print("  ", "  ".join(nodes))  # Header row
    for i, node in enumerate(nodes):
        print(node, "  ", "  ".join(map(str, distance_table[i])))  # Each row of table
def main():
    num_nodes, nodes, cost_matrix = get_user_input()
    distance_table = distance_vector_routing(num_nodes, nodes, cost_matrix)
    print_routing_table(nodes, distance_table)
if __name__ == "__main__":
    main()
```

```
#2. Dijkstra's Algorithm Implementation for Network Routing
import heapq
def get_user_input():
    num_nodes = int(input("Enter the number of nodes: "))
    nodes = []
    print("Enter node names:")
    for _ in range(num_nodes):
        nodes.append(input().strip())
    print(f"Enter the {num_nodes}x{num_nodes} cost matrix (use -1 for no direct path):")
```

```
    cost_matrix = []
    for i in range(num_nodes):
        while True:
            row = list(map(int, input().split()))
            if len(row) == num_nodes:
                cost_matrix.append(row)
                break
            else:
                print(f"Invalid input! Enter exactly {num_nodes} values for row {i + 1}.")
    return num_nodes, nodes, cost_matrix
def dijkstra(num_nodes, nodes, cost_matrix, src):
    distances = {node: float('inf') for node in nodes}
    distances[nodes[src]] = 0
    priority_queue = [(0, nodes[src])]
    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)
        for i, neighbor in enumerate(nodes):
            if cost_matrix[nodes.index(current_node)][i] != -1:
                distance = current_distance + cost_matrix[nodes.index(current_node)][i]
                if distance < distances[neighbor]:
                    distances[neighbor] = distance
                    heapq.heappush(priority_queue, (distance, neighbor))
    return distances
def print_routing_table(nodes, distances):
    print("\nFinal Routing Table:")
    for node, distance in distances.items():
        print(f"Node {node}: {distance}")
def main():
```

```
    num_nodes, nodes, cost_matrix = get_user_input()
    src = input("Enter the source node: ").strip()
    while src not in nodes:
        print("Invalid node! Please enter a valid source node.")
        src = input("Enter the source node: ").strip()
    distances = dijkstra(num_nodes, nodes, cost_matrix, nodes.index(src))
    print_routing_table(nodes, distances)
if __name__ == "__main__":
    main()
```

```
#3. IP Address Decimal to Binary Converter
def decimal_to_binary_ip(ip_address):
    try:
        octets = ip_address.split(".")
        if len(octets) != 4:
            raise ValueError("Invalid IP address format")
        binary_octets = [format(int(octet), '08b') for octet in octets]
        binary_ip = ".".join(binary_octets)
        return binary_ip
    except ValueError:
        return "Invalid IP address"
# Get user input
ip_address = input("Enter an IP address (e.g., 192.168.1.1): ")
binary_ip = decimal_to_binary_ip(ip_address)
print("Binary representation:", binary_ip)
```

#4. IP Address Binary to Decimal Converter

```
def binary_to_decimal_ip(binary_ip):
    try:
        octets = binary_ip.split(".")
        if len(octets) != 4:
            raise ValueError("Invalid binary IP format")
        decimal_octets = [str(int(octet, 2)) for octet in octets]
        decimal_ip = ".".join(decimal_octets)
        return decimal_ip
    except ValueError:
        return "Invalid binary IP address"
# Get user input
binary_ip = input("Enter a binary IP address (e.g., 11000000.10101000.00000001.00000001): ")
decimal_ip = binary_to_decimal_ip(binary_ip)
print("Decimal representation:", decimal_ip)
```

#5. IP Subnet Calculator

```
def validate_ip(ip):
    """Validate if the given string is a valid IPv4 address."""
    octets = ip.split('.')
    if len(octets) != 4:
        return False
    try:
        for octet in octets:
            value = int(octet)
            if value < 0 or value > 255:
                return False
```

```
        return True
    except ValueError:
        return False
def get_subnet_info(ip_address, subnet_mask):
    """Calculate subnet information from an IP address and subnet mask."""
    # Convert IP address and subnet mask to binary
    ip_binary = "".join([format(int(octet), '08b') for octet in ip_address.split('.')])
    mask_binary = "".join([format(int(octet), '08b') for octet in subnet_mask.split('.')])
    # Calculate network bits and host bits
    network_bits = mask_binary.count('1')
    host_bits = 32 - network_bits
    # Calculate number of available hosts
    available_hosts = 2 ** host_bits - 2  # Subtract network and broadcast addresses
    if available_hosts < 0:
        available_hosts = 0  # In case of /31 or /32 masks
    # Calculate network address
    network_binary = ip_binary[:network_bits] + '0' * host_bits
    network_address = ".".join([str(int(network_binary[i:i+8], 2)) for i in range(0, 32, 8)])
    # Calculate broadcast address
    broadcast_binary = ip_binary[:network_bits] + '1' * host_bits
    broadcast_address = ".".join([str(int(broadcast_binary[i:i+8], 2)) for i in range(0, 32, 8)])
    # Calculate first and last usable host addresses
    if host_bits <= 1:  # For /31 and /32 masks
        first_host = network_address
        last_host = broadcast_address
    else:
        first_host_binary = ip_binary[:network_bits] + '0' * (host_bits - 1) + '1'
        first_host = ".".join([str(int(first_host_binary[i:i+8], 2)) for i in range(0, 32, 8)])
```

```
        last_host_binary = ip_binary[:network_bits] + '1' * (host_bits - 1) + '0'
        last_host = ".".join([str(int(last_host_binary[i:i+8], 2)) for i in range(0, 32, 8)])
    # CIDR notation
    cidr = f"{ip_address}/{network_bits}"
    return {
        "network_address": network_address,
        "broadcast_address": broadcast_address,
        "first_host": first_host,
        "last_host": last_host,
        "available_hosts": available_hosts,
        "subnet_mask": subnet_mask,
        "cidr": cidr,
        "network_bits": network_bits,
        "host_bits": host_bits
    }
def main():
    print("IP Subnet Calculator")
    print("--------------------")
    while True:
        ip_address = input("Enter IP address (e.g., 192.168.1.10): ")
        if not validate_ip(ip_address):
            print("Invalid IP address. Please try again.")
            continue
        # Offer both input methods
        choice = input("Enter subnet mask as: (1) Decimal (e.g., 255.255.255.0) or (2) CIDR
(e.g., 24): ")
```

```
if choice == "1":
    subnet_mask = input("Enter subnet mask (e.g., 255.255.255.0): ")
    if not validate_ip(subnet_mask):
        print("Invalid subnet mask. Please try again.")
        continue
elif choice == "2":
    try:
        cidr = int(input("Enter CIDR notation (e.g., 24): "))
        if cidr < 0 or cidr > 32:
            print("Invalid CIDR value. Please enter a number between 0 and 32.")
            continue
        # Convert CIDR to subnet mask
        subnet_binary = '1' * cidr + '0' * (32 - cidr)
        subnet_mask = ".".join([str(int(subnet_binary[i:i+8], 2)) for i in range(0, 32, 8)])
    except ValueError:
        print("Invalid CIDR value. Please enter a valid number.")
        continue
else:
    print("Invalid choice. Please try again.")
    continue
# Get subnet details
result = get_subnet_info(ip_address, subnet_mask)
# Print subnet information
print("\nSubnet Information:")
print(f"IP Address: {ip_address}")
print(f"Subnet Mask: {result['subnet_mask']} (/{result['network_bits']})")
print(f"Network Address: {result['network_address']}")
print(f"Broadcast Address: {result['broadcast_address']}")
```

```
        print(f"First Usable Host: {result['first_host']}")
        print(f"Last Usable Host: {result['last_host']}")
        print(f"Number of Usable Hosts: {result['available_hosts']}")
        print(f"CIDR Notation: {result['cidr']}")
        print(f"Network Bits: {result['network_bits']}")
        print(f"Host Bits: {result['host_bits']}")
        # Ask if user wants another calculation
        another = input("\nCalculate another subnet? (y/n): ")
        if another.lower() != 'y':
            break
if __name__ == "__main__":
    main()
```

#6. Simulate Internet model(TCP/IP)

Receiver.py

```
import socket
import base64


def physical_layer_reverse(message):
    """Reverse Physical Layer: Convert binary back to text."""
    binary_segments = message.split()
    text_message = ''.join(chr(int(b, 2)) for b in binary_segments)
    return text_message


def data_link_layer_reverse(message):
    """Reverse Data Link Layer: Remove start and end delimiters."""
    return message.replace("<START>", "").replace("<END>", "")
```

```
def network_layer_reverse(message):
    """Reverse Network Layer: Remove source and destination IP addresses."""
    return message.split(" | Data: ")[1]

def transport_layer_reverse(message):
    """Reverse Transport Layer: Remove checksum."""
    return message.split(" | Checksum: ")[0]

def session_layer_reverse(message):
    """Reverse Session Layer: Ensure only the encoded message remains."""
    parts = message.split(" | ", 1)  # Ensure split is only at the first "|"
    return parts[1] if len(parts) > 1 else message  # Avoid index error

def presentation_layer_reverse(message):
    """Reverse Presentation Layer: Decode Base64 to original text safely."""
    try:
        missing_padding = len(message) % 4
        if missing_padding:
            message += '=' * (4 - missing_padding)  # Fix padding
        return base64.b64decode(message).decode()
    except Exception as e:
        print(f"Base64 Decoding Error: {e}")
        return "ERROR"

def receiver():
    host = "127.0.0.1"
    port = 6000
```

```
with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
  s.bind((host, port))
  s.listen(1)
  print("Waiting for the process...")

  conn, addr = s.accept()
  with conn:
    print(f"Connected by {addr}")

    # Receive the full message
    received_message = b""
    while True:
      chunk = conn.recv(1024)
      if not chunk:
        break
      received_message += chunk

    message = received_message.decode()

    # Reverse OSI processing
    message = physical_layer_reverse(message)
    message = data_link_layer_reverse(message)
    message = network_layer_reverse(message)
    message = transport_layer_reverse(message)
    message = session_layer_reverse(message)
    message = presentation_layer_reverse(message)

    print(f"\nFinal Decoded Message: {message}")
```

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

Process.py

```
import socket
import time
import base64

def physical_layer(message):
    """Simulate Physical Layer: Convert the message to binary (bits)."""
    binary_message = ' '.join(format(ord(char), '08b') for char in message)
    print(f"Physical Layer (Bits): {binary_message}")
    time.sleep(1)
    return binary_message

def data_link_layer(message):
    """Simulate Data Link Layer: Add start and end delimiters to form a frame."""
    frame = f"<START>{message}<END>"
    print(f"Data Link Layer (Frames): {frame}")
    time.sleep(1)
    return frame

def network_layer(message):
    """Simulate Network Layer: Add source and destination IP addresses."""
    source_ip = "192.168.1.1"
    dest_ip = "192.168.1.2"
    packet = f"Src: {source_ip} | Dest: {dest_ip} | Data: {message}"
```

```python
    print(f"Network Layer (Packets): {packet}")
    time.sleep(1)
    return packet

def transport_layer(message):
    """Simulate Transport Layer: Add checksum."""
    checksum = sum(ord(char) for char in message) % 256
    segment = f"{message} | Checksum: {checksum}"
    print(f"Transport Layer (Segments): {segment}")
    time.sleep(1)
    return segment

def session_layer(message):
    """Simulate Session Layer: Add session ID."""
    session_id = "Session123"
    session_data = f"SessionID: {session_id} | {message}"
    print(f"Session Layer (Session Data): {session_data}")
    time.sleep(1)
    return session_data

def presentation_layer(message):
    """Simulate Presentation Layer: Encode the message into Base64."""
    encoded_message = base64.b64encode(message.encode()).decode()
    print(f"Presentation Layer (Formatted Data): {encoded_message}")
    time.sleep(1)
    return encoded_message

def application_layer(message):
```

```
    """Simulate Application Layer: Add user-related metadata."""
    user_data = f"User Message: {message}"
    print(f"Application Layer (User Data): {user_data}")
    time.sleep(1)
    return user_data

def simulate_osi_layers(message):
    """
    Simulates the manipulation of the message through the OSI layers.
    """
    print("\nSimulating OSI Layers:")
    message = application_layer(message)  # Application Layer
    message = presentation_layer(message)  # Presentation Layer
    message = session_layer(message)  # Session Layer
    message = transport_layer(message)  # Transport Layer
    message = network_layer(message)  # Network Layer
    message = data_link_layer(message)  # Data Link Layer
    message = physical_layer(message)  # Physical Layer
    return message

def process():
    host = "127.0.0.1"
    port = 5000
    forward_port = 6000

    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as user_socket:
        user_socket.bind((host, port))
        user_socket.listen(1)
```

```
    print("Waiting for the user...")

    conn, addr = user_socket.accept()
    with conn:
      print(f"\nConnected by {addr}")
      message = conn.recv(1024).decode()
      print(f"Received from user: {message}")

      # Simulate OSI model processing
      transformed_message = simulate_osi_layers(message)
      print(f"\nFinal Transformed Message: {transformed_message}")

      # Forward to receiver
      with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as receiver_socket:
        receiver_socket.connect((host, forward_port))
        receiver_socket.sendall(transformed_message.encode())  # Ensure full message
is sent
        print("\nFinal transformed message sent to the receiver.")

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

User.py

```
import socket

def user():
  host = "127.0.0.1"  # Localhost
  port = 5000  # Port to send to process
```

```
    with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
        s.connect((host, port))
        message = input("Enter the message to send: ")
        s.sendall(message.encode())
        print("Message sent to the process.")

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

#7. Client-Server communication using UDP

Client.py

```
import socket

# Create a UDP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Server details
server_address = ("localhost", 12345)

print("Type your messages below. Press Ctrl+C to terminate.")

try:
    while True:
        # Take input from the user
        message = input("You: ")
        if message.strip():  # Send only non-empty messages
            client_socket.sendto(message.encode(), server_address)
```

```
except KeyboardInterrupt:
    print("\nClient terminated.")
finally:
    # Close the socket
    client_socket.close()
```

Server.py

```
import socket

# Create a UDP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

# Bind the socket to an address and port
server_address = ("localhost", 12345)
server_socket.bind(server_address)

print("UDP Server is running on port 12345.")

while True:
    try:
        # Receive data from the client
        data, client_address = server_socket.recvfrom(1024)  # Buffer size is 1024 bytes
        print(f"Received: {data.decode()} from {client_address}")
    except Exception as e:
        print(f"Error occurred: {e}")
```

#9. CHAT application

Server.py

```
import socket
import threading

# Server Configuration
HOST = '127.0.0.1'
PORT = 12345

server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((HOST, PORT))
server.listen()

clients = []
nicknames = []

def broadcast(message):
    """Send message to all connected clients."""
    for client in clients:
        client.send(message.encode('utf-8'))

def handle_client(client):
    """Receive and broadcast messages."""
    while True:
        try:
            message = client.recv(1024).decode('utf-8')
            broadcast(message)  # Send to all clients
        except:
            index = clients.index(client)
            clients.remove(client)
```

```python
            client.close()
            nickname = nicknames.pop(index)
            broadcast(f"\n {nickname} has left the chat.\n")
            break

def receive_connections():
    """Accept new client connections."""
    while True:
        client, address = server.accept()
        print(f"Connected with {str(address)}")

        # Ask for a nickname
        client.send("NICK".encode('utf-8'))
        nickname = client.recv(1024).decode('utf-8')

        nicknames.append(nickname)
        clients.append(client)

        welcome_message = f"\n Hello {nickname}, welcome to the chat application!\n"
        broadcast(welcome_message)
        client.send("\n You are connected to the server!\n".encode('utf-8'))

        # Start a new thread to handle the client
        thread = threading.Thread(target=handle_client, args=(client,))
        thread.start()

print("Server is running on port 12345...")
receive_connections()
```

Client.py

```
import socket
import threading
import tkinter as tk
from tkinter import scrolledtext, messagebox


class ChatClient:
    def __init__(self, root, client):
        self.root = root
        self.client = client
        self.root.title("Chat Application")

        # Chat display area (with proper word wrapping)
        self.chat_area = scrolledtext.ScrolledText(self.root, width=60, height=20,
wrap="word")
        self.chat_area.pack(padx=10, pady=10)
        self.chat_area.config(state=tk.DISABLED)

        # Message input area
        self.entry = tk.Entry(self.root, width=50)
        self.entry.pack(padx=10, pady=5, side=tk.LEFT)

        # Send button
        self.send_button = tk.Button(self.root, text="Send", command=self.send_message)
        self.send_button.pack(padx=5, pady=5, side=tk.RIGHT)

        # Start receiving messages in a separate thread
        threading.Thread(target=self.receive_messages, daemon=True).start()
```

```
    def receive_messages(self):
        """Receive and display messages with proper formatting."""
        while True:
            try:
                message = self.client.recv(1024).decode('utf-8')
                self.display_message(message)
            except:
                messagebox.showerror("Error", "Connection lost!")
                self.client.close()
                break

    def display_message(self, message):
        """Format system messages and display them properly."""
        self.chat_area.config(state=tk.NORMAL)

        # Highlight system messages differently
        if message.startswith("\n") or message.startswith("Welcome") or
message.startswith("Server"):
            self.chat_area.insert(tk.END, message + "\n\n", "system")
        else:
            self.chat_area.insert(tk.END, message + "\n\n")

        self.chat_area.config(state=tk.DISABLED)
        self.chat_area.yview(tk.END)

    def send_message(self):
        """Send message to the server."""
```

```
    message = self.entry.get()
    if message:
        self.client.send(f"{nickname}: {message}".encode('utf-8'))
        self.entry.delete(0, tk.END)

# Connect to the server
HOST = '127.0.0.1'
PORT = 12345

client = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client.connect((HOST, PORT))

# Get user nickname
nickname = input("Enter your nickname: ")
client.send(nickname.encode('utf-8'))

# Start GUI
root = tk.Tk()
chat_app = ChatClient(root, client)

# Add styles for system messages
chat_app.chat_area.tag_config("system", foreground="blue", font=("Arial", 10, "bold"))

root.mainloop()
```

#10. Apply any cryptography algorithm for message transfer between client and Server

Server.py

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

# Generate RSA key pair
private_key = rsa.generate_private_key(
    public_exponent=65537,
    key_size=2048
)

# Extract the public key
public_key = private_key.public_key()

# Serialize public key to share with the client
pem_public_key = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

# Function to decrypt received message
def decrypt_message(encrypted_message):
    decrypted_message = private_key.decrypt(
        encrypted_message,
        padding.OAEP(
            mgf=padding.MGF1(algorithm=hashes.SHA256()),
            algorithm=hashes.SHA256(),
            label=None
```

```
        )
    )
    return decrypted_message.decode()

# Server setup
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 12345))  # Bind to localhost on port 12345
server_socket.listen(1)
print("Server is listening for a connection...")

conn, addr = server_socket.accept()
print(f"Connected to client: {addr}")

# Send the public key to the client
conn.send(pem_public_key)

# Receive encrypted message from client
encrypted_message = conn.recv(4096)
print("\nReceived Encrypted Message:", encrypted_message)

# Decrypt the message
received_message = decrypt_message(encrypted_message)
print("Decrypted Message:", received_message)

# Send plaintext acknowledgment (instead of encrypted)
ack_message = f"Acknowledgment: Received '{received_message}'"
conn.send(ack_message.encode())
```

```
# Close the connection
conn.close()
server_socket.close()
print("\nServer closed connection.")
```

Client.py

```
import socket
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import serialization, hashes

# Client setup
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
client_socket.connect(('localhost', 12345))

# Receive public key from server
pem_public_key = client_socket.recv(4096)
public_key = serialization.load_pem_public_key(pem_public_key)

# Get user message input
message = input("Enter message to send: ")

# Encrypt the message
encrypted_message = public_key.encrypt(
    message.encode(),
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
```

```
    )
)

# Send encrypted message to server
client_socket.send(encrypted_message)
print("\nMessage sent securely.")

# Receive and decode plaintext acknowledgment
ack_message = client_socket.recv(4096).decode()
print("\nServer Response:", ack_message)

# Close connection
client_socket.close()
```

```
#11. Go back ‘n’ ARQ
import random

def send_packets(window_size, total_packets, loss_probability):
    base = 0
    ack_received = [False] * total_packets  # Track acknowledgments

    while base < total_packets:
        print(f"Sending packets in window: {list(range(base, min(base + window_size, total_packets)))}")

        # Send packets in the window
        for i in range(base, min(base + window_size, total_packets)):
            if not ack_received[i]:  # Send only unacknowledged packets
```

```
            print(f"Sent packet {i}")
            if random.random() < loss_probability:
                print(f"Packet {i} lost!")
            else:
                print(f"Packet {i} received successfully!")
                ack_received[i] = True  # Mark as acknowledged

        # Move the window forward when acknowledgments are received
        while base < total_packets and ack_received[base]:
            print(f"Acknowledgement received for packet {base}")
            base += 1

        print("---")  # Separator for each transmission round

def main():
    total_packets = int(input("Enter total number of packets: "))
    window_size = int(input("Enter window size: "))
    loss_probability = float(input("Enter packet loss probability (0 to 1): "))

    send_packets(window_size, total_packets, loss_probability)

if __name__ == "__main__":
    main()

#12. Selective Repeat ARQ
import random

def send_packets_sr(window_size, total_packets, loss_probability):
```

```
    sent_packets = [False] * total_packets  # Track sent packets
    ack_received = [False] * total_packets  # Track acknowledgments
    base = 0

    while base < total_packets:
        print(f"Sending packets in window: {list(range(base, min(base + window_size,
total_packets)))}")

        # Send all packets in the window that have not been acknowledged
        for i in range(base, min(base + window_size, total_packets)):
            if not ack_received[i]:  # Only send if not acknowledged
                print(f"Sent packet {i}")
                sent_packets[i] = True

                # Simulate packet loss
                if random.random() < loss_probability:
                    print(f"Packet {i} lost!")
                else:
                    print(f"Packet {i} received successfully!")
                    ack_received[i] = True  # Mark packet as acknowledged

        # Check for contiguous acknowledgments and slide window
        while base < total_packets and ack_received[base]:
            print(f"Acknowledgement received for packet {base}")
            base += 1  # Slide window forward

        print("---")  # Separator for each transmission round
```

```
def main():
    total_packets = int(input("Enter total number of packets: "))
    window_size = int(input("Enter window size: "))
    loss_probability = float(input("Enter packet loss probability (0 to 1): "))

    send_packets_sr(window_size, total_packets, loss_probability)

if __name__ == "__main__":
    main()

#13. CRC
import random

def xor(a, b):
    """Perform XOR operation between two binary strings of equal length."""
    if len(a) != len(b):
        raise ValueError(f"Strings must be of equal length. Got lengths {len(a)} and {len(b)}")
    return ''.join('1' if a[i] != b[i] else '0' for i in range(len(a)))

def binary_division(dividend, divisor):
    """Perform binary division (modulo-2) and return the remainder."""
    if len(dividend) < len(divisor):
        raise ValueError("Dividend length must be greater than or equal to divisor length")

    # Initialize variables
    pick = len(divisor)
    tmp = list(dividend[:pick])  # Convert to list for easier manipulation
    dividend = list(dividend)  # Convert dividend to list
```

```
    while pick < len(dividend):
        if tmp[0] == '1':
            tmp = list(xor(''.join(tmp), divisor))  # XOR with divisor

        tmp.pop(0)  # Remove first bit
        tmp.append(dividend[pick])  # Add next bit from dividend
        pick += 1

    # Final XOR if necessary
    if tmp[0] == '1':
        tmp = list(xor(''.join(tmp), divisor))

    return ''.join(tmp[1:])  # Return remainder after removing first bit

def encode_crc(data, divisor):
    """Encode data using CRC and return codeword."""
    if not all(bit in '01' for bit in data):
        raise ValueError("Data must be a binary string (containing only 0s and 1s)")
    if not all(bit in '01' for bit in divisor):
        raise ValueError("Divisor must be a binary string (containing only 0s and 1s)")
    if not divisor.startswith('1'):
        raise ValueError("Generator polynomial (divisor) must start with 1")
    if len(divisor) < 2:
        raise ValueError("Generator polynomial (divisor) must be at least 2 bits long")
    if len(data) == 0:
        raise ValueError("Data string cannot be empty")
```

```
    appended_data = data + '0' * (len(divisor) - 1)
    remainder = binary_division(appended_data, divisor)
    codeword = data + remainder
    return codeword, remainder


def verify_crc(received_data, divisor):
    """Verify received data for errors."""
    if len(received_data) < len(divisor):
        raise ValueError("Received data length must be greater than divisor length")

    remainder = binary_division(received_data, divisor)
    return remainder == '0' * (len(divisor) - 1)


# Example usage
if __name__ == "__main__":
    try:
        # User Input
        data = input("Enter binary data: ").strip()
        divisor = input("Enter generator polynomial (binary): ").strip()

        # Encoding
        codeword, remainder = encode_crc(data, divisor)
        print(f"Encoded data (Codeword): {codeword}")
        print(f"CRC Remainder: {remainder}")

        # Verification
        received_data = input("Enter received binary data for verification: ").strip()
        if verify_crc(received_data, divisor):
```

```
        print("No error detected.")
    else:
        print("Error detected in received data.")

  except ValueError as e:
    print(f"Error: {e}")
  except Exception as e:
    print(f"An unexpected error occurred: {e}")
```

#14. Checksum

```
def calculate_checksum(data, block_size=8):
  """
  Calculate checksum for binary data.
  Args:
    data (str): Binary input data
    block_size (int): Size of each block in bits
  Returns:
    str: Binary checksum
  """
  # Validate input is binary
  if not all(bit in '01' for bit in data):
    raise ValueError("Input must be binary (0s and 1s only)")

  # Pad data if necessary to make it multiple of block_size
  if len(data) % block_size != 0:
    padding_length = block_size - (len(data) % block_size)
    data = data + '0' * padding_length  # Padding with zeros
```

```python
    # Split data into blocks
    blocks = [data[i:i + block_size] for i in range(0, len(data), block_size)]

    # Sum all blocks
    sum_val = 0
    for block in blocks:
        sum_val = (sum_val + int(block, 2)) % (2 ** block_size)

    # Take one's complement
    checksum = ((2 ** block_size) - 1) - sum_val

    # Convert to binary and ensure it's block_size bits
    checksum_binary = format(checksum, f'0{block_size}b')

    return checksum_binary


def verify_checksum(data, received_checksum, block_size=8):
    """
    Verify received data using checksum.
    Args:
        data (str): Received binary data
        received_checksum (str): Received checksum in binary
        block_size (int): Size of each block in bits
    Returns:
        bool: True if no error detected, False otherwise
    """
    # Validate inputs are binary
```

```
    if not all(bit in '01' for bit in data) or not all(bit in '01' for bit in received_checksum):
        raise ValueError("Both data and checksum must be binary (0s and 1s only)")

    # Calculate checksum for received data
    calculated_checksum = calculate_checksum(data, block_size)

    # If the calculated checksum + received checksum results in all 1s, it's valid
    checksum_sum = int(calculated_checksum, 2) + int(received_checksum, 2)
    valid_checksum = (checksum_sum % (2 ** block_size)) == (2 ** block_size) - 1

    return valid_checksum


def main():
    try:
        # Sender side
        print("Note: This implementation expects binary input (sequence of 0s and 1s)")

        original_data = input("Enter binary data to transmit: ").strip()

        # Validate binary input
        if not all(bit in '01' for bit in original_data):
            raise ValueError("Input must be binary (0s and 1s only)")

        block_size = int(input("Enter block size (in bits, default is 8): ") or "8")

        # Calculate checksum
        checksum = calculate_checksum(original_data, block_size)
```

```
        print(f"\nOriginal Data: {original_data}")
        print(f"Calculated Checksum: {checksum}")

        # Receiver side
        print("\n--- Receiver Side ---")
        received_data = input("Enter received binary data: ").strip()
        received_checksum = input("Enter received checksum (binary): ").strip()

        # Verify checksum
        if verify_checksum(received_data, received_checksum, block_size):
            print("\nNo error detected! Data is valid.")
        else:
            print("\nError detected! Data may be corrupted.")
            print(f"Expected checksum: {calculate_checksum(received_data, block_size)}")
            print(f"Received checksum: {received_checksum}")

    except ValueError as e:
        print(f"Error: {e}")
    except Exception as e:
        print(f"An unexpected error occurred: {e}")


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