CN&S LAB (21CDL61) - LAB Manual

List of Experiments

1. Socket Programming using TCP – Convert Lowercase to Uppercase.
2. Simulating TCP 3-Way Handshake and Connection Termination Using Python.
3. Socket Programming using UDP – Convert Lowercase to Uppercase.
4. Ping Simulation (ICMP Echo Request/Reply) Using Python.
5. Packet Fragmentation Simulation Using Python.
6. Simulation of Dijkstra's Algorithm Using Python.
7. Simulation of Bellman-Ford Algorithm Using Python.
8. Simulation of RSA Algorithm Using Python.
9. Diffie–Hellman Key Exchange Implementation Using Python.

EXPERIMENT NO: 1

Title: Socket Programming using TCP – Convert Lowercase to Uppercase

Aim

To implement a TCP-based client-server socket program using Python, where the client sends a lowercase string to the server and the server converts it into uppercase and sends it back.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	`socket` (built-in)
OS	Windows / Linux / macOS

Theory

A socket is one endpoint of a two-way communication link between two programs running on a network. Socket programming enables communication between processes using IP and port numbers.

TCP (Transmission Control Protocol)

- Connection-oriented protocol
- Reliable data transfer
- Uses `SOCK_STREAM` type socket

TCP Socket Communication Steps:

1. The server creates a socket and binds it to a port.
2. The Server listens for connections.
3. Client creates a socket and connects to the server.
4. Server accepts connection.
5. Data is exchanged.
6. Both sockets are closed.

Algorithm

Server (`tcp_server.py`):

1. Import `socket` module.
2. Create a TCP socket using `socket.socket()`.
3. Bind the socket to host (`localhost`) and port (`12345`).
4. Use `listen()` to wait for incoming connections.
5. Use `accept()` to accept the connection from the client.
6. Receive the string from the client.
7. Convert the string to uppercase using `.upper()`.
8. Send the modified string back to the client.
9. Close the connection.

Client (`tcp_client.py`):

1. Import `socket` module.
2. Create a TCP socket.
3. Connect to the server using `connect()`.
4. Read a lowercase string from the user.
5. Send the string to the server.
6. Receive the modified string from the server.
7. Print the uppercase string.
8. Close the connection.

Program

Server Program: `tcp_server.py`

```
import socket

# Create a TCP/IP socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Bind the socket to a specific address and port
server_socket.bind(('localhost', 12345))
server_socket.listen(1)

print("Server is waiting for client connection...")

# Accept a connection
conn, addr = server_socket.accept()
print(f"Connected by {addr}")

# Receive data from client
data = conn.recv(1024).decode()
print(f"Received from client: {data}")

# Convert lowercase to uppercase
upper_data = data.upper()

# Send back the converted data
conn.send(upper_data.encode())

# Close the connection
conn.close()
server_socket.close()
```

Client Program: `tcp_client.py`

```
import socket

# Create a TCP/IP socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# Connect to the server
client_socket.connect(('localhost', 12345))

# Input lowercase string from user
message = input("Enter a lowercase string: ")

# Send data to server
client_socket.send(message.encode())

# Receive response from server
```

```
data = client_socket.recv(1024).decode()
print(f"Received from server: {data}")

# Close the socket
client_socket.close()
```

Result

The TCP client-server socket program was successfully implemented and executed. The server correctly converted the lowercase string to uppercase and returned it to the client.

Sample Output

Client Terminal

```
Enter a lowercase string: hello world
Received from server: HELLO WORLD
```

Server Terminal

```
Server is waiting for client connection...
Connected by ('127.0.0.1', 56789)
Received from client: hello world
```

EXPERIMENT NO: 2

Title: Simulating TCP 3-Way Handshake and Connection Termination Using Socket Programming in Python

Aim

To simulate the TCP 3-way handshake (connection establishment) and connection termination steps using Python, with logging messages to illustrate the actual flow.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	`socket`, `time` (both built-in)
OS	Windows / Linux / macOS

Theory

TCP 3-Way Handshake:

1. **Client → Server**: SYN (synchronize)
2. **Server → Client**: SYN-ACK (synchronize-acknowledge)
3. **Client → Server**: ACK (acknowledge)

This establishes a reliable connection between client and server.

TCP Connection Termination:

1. **Client → Server**: FIN
2. **Server → Client**: ACK
3. **Server → Client**: FIN
4. **Client → Server**: ACK

Algorithm

Server:

1. Create a socket, bind, and listen.
2. Accept the client connection.
3. Simulate and print messages corresponding to SYN-ACK, ACK, etc.
4. After message exchange, simulate connection termination steps.

Client:

1. Create and connect a socket to the server.
2. Simulate sending SYN, receiving SYN-ACK, sending ACK.
3. Send a dummy message.
4. Simulate FIN and ACK steps for connection termination.

Program

Server Program – `tcp_handshake_server.py`

```
import socket
import time

# Create socket
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_socket.bind(('localhost', 12346))
server_socket.listen(2)

print("Server: Listening for connections...")

conn, addr = server_socket.accept()
print(f"Server: Received SYN from {addr}")
time.sleep(2)

print("Server: Sending SYN-ACK")
time.sleep(2)

data = conn.recv(1024).decode()
print(f"Server: Received ACK → Handshake complete")
time.sleep(2)

# Receive message
message = conn.recv(1024).decode()
print(f"Server: Received data → {message}")

# Simulate connection termination
print("Server: Sending ACK for FIN")
time.sleep(2)
print("Server: Sending FIN")
time.sleep(2)

conn.send("FIN".encode())  # Send FIN

final_ack = conn.recv(1024).decode()
print(f"Server: Received final ACK → Connection closed.")

conn.close()
server_socket.close()
```

Client Program – `tcp_handshake_client.py`

```
import socket
import time

# Create socket
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

print("Client: Sending SYN")
time.sleep(2)

client_socket.connect(('localhost', 12346))

print("Client: Received SYN-ACK")
time.sleep(2)

print("Client: Sending ACK")
client_socket.send("ACK".encode())
time.sleep(2)

# Send actual message
```

```
client_socket.send("Hello from Client!".encode())
time.sleep(2)

# Simulate connection termination
print("Client: Sending FIN")
time.sleep(2)

# Receive server's FIN
fin = client_socket.recv(1024).decode()
if fin == "FIN":
    print("Client: Received FIN")
    print("Client: Sending final ACK")
    client_socket.send("ACK".encode())

client_socket.close()
```

Result

The 3-way TCP handshake and 4-step connection termination were successfully simulated and printed. This gave a clear idea of how TCP connections are established and closed.

Sample Output

Client Terminal

```
Client: Sending SYN
Client: Received SYN-ACK
Client: Sending ACK
Client: Sending FIN
Client: Received FIN
Client: Sending final ACK
```

Server Terminal

```
Server: Listening for connections...
Server: Received SYN from ('127.0.0.1', 54321)
Server: Sending SYN-ACK
Server: Received ACK → Handshake complete
Server: Received data → Hello from Client!
Server: Sending ACK for FIN
Server: Sending FIN
Server: Received final ACK → Connection closed.
```

EXPERIMENT NO: 3

Title: Socket Programming using UDP – Convert Lowercase to Uppercase

Aim

To implement a UDP-based client-server socket program in Python where the client sends a lowercase string, and the server converts it into uppercase and sends it back.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	`socket`
OS	Windows / Linux / macOS

Theory

A socket is a software endpoint for sending or receiving data between two machines.

UDP (User Datagram Protocol):

- Connectionless: No handshake before sending data.
- Unreliable: No guarantee that packets will arrive or arrive in order.
- Fast and lightweight compared to TCP.
- Used in applications like video streaming, VoIP, DNS.

Key UDP Functions in Python:

- `socket.socket(socket.AF_INET, socket.SOCK_DGRAM)`: Creates a UDP socket.
- `sendto(data, address)`: Sends data to a specific address.
- `recvfrom(buffer_size)`: Receives data and also gives the sender's address.

Algorithm

Server (udp_server.py):

1. Import the `socket` module.
2. Create a UDP socket.
3. Bind the socket to a host and port.
4. Wait to receive a message from the client using `recvfrom()`.
5. Convert the received lowercase string to uppercase.
6. Send the uppercase string back to the client using `sendto()`.
7. Close the socket (optional, since UDP is connectionless).

Client (udp_client.py):

1. Import the `socket` module.
2. Create a UDP socket.
3. Input a lowercase string from the user.

4. Send the string to the server using `sendto()`.
5. Receive the converted string from the server using `recvfrom()`.
6. Print the result.
7. Close the socket.

Program

Server Program: `udp_server.py`

```
import socket

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

# Bind to localhost on port 12347
server_socket.bind(('localhost', 12347))

print("UDP Server is running and waiting for data...")

# Receive data from client
data, client_addr = server_socket.recvfrom(1024)
print(f"Received from client: {data.decode()}")

# Convert to uppercase
upper_data = data.decode().upper()

# Send it back to the client
server_socket.sendto(upper_data.encode(), client_addr)
print(f"Sent to client: {upper_data}")
```

Client Program: `udp_client.py`

```
import socket

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

# Server address
server_addr = ('localhost', 12347)

# Input message
message = input("Enter a lowercase string: ")

# Send to server
client_socket.sendto(message.encode(), server_addr)

# Receive response
data, _ = client_socket.recvfrom(1024)
print(f"Received from server: {data.decode()}")

# Close socket
client_socket.close()
```

Result

The UDP client-server program was successfully implemented and executed. The server received the lowercase string, converted it to uppercase, and sent it back to the client.

Sample Output

Client Terminal

```
Enter a lowercase string: good morning
Received from server: GOOD MORNING
```

Server Terminal

```
UDP Server is running and waiting for data...
Received from client: good morning
Sent to client: GOOD MORNING
```

EXPERIMENT NO: 4

Title: Ping Simulation (ICMP Echo Request/Reply) Using Python

Aim

To simulate the behaviour of the Ping command using Python by sending ICMP Echo Request packets and receiving ICMP Echo Reply, measuring response time.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	`socket`, `time`, `os`, `struct`
OS	Windows / Linux / macOS
Access	Administrator or Root privileges (for raw sockets)

Theory

Ping uses the Internet Control Message Protocol (ICMP) to:

- Send Echo Request to a host
- Wait for Echo Reply
- Measure round-trip time

Fields in ICMP Echo Packet:

- Type: 8 (Request), 0 (Reply)
- Code: 0
- Checksum
- Identifier
- Sequence Number
- Payload (data)

Algorithm

1. Create a raw ICMP socket.
2. Construct an ICMP Echo Request packet:
    - Set type, code, checksum, ID, sequence, data.
3. Send packet to the target IP.
4. Start timer.
5. Wait for ICMP Echo Reply.
6. Stop timer and calculate Round Trip Time.
7. Repeat 4–5 times.

Python Program – `ping_simulation.py`

```
import socket
import os
import struct
```

```
import time

ICMP_ECHO_REQUEST = 8

def checksum(source_string):
    """Calculate the checksum of a packet"""
    sum = 0
    max_count = (len(source_string) // 2) * 2

    count = 0
    while count < max_count:
        val = source_string[count + 1] * 256 + source_string[count]
        sum += val
        sum = sum & 0xffffffff
        count += 2

    if max_count < len(source_string):
        sum += source_string[len(source_string) - 1]
        sum = sum & 0xffffffff

    sum = (sum >> 16) + (sum & 0xffff)
    sum += (sum >> 16)
    answer = ~sum
    answer = answer & 0xffff
    return answer >> 8 | (answer << 8 & 0xff00)

def create_packet(id):
    """Create ICMP Echo Request packet"""
    header = struct.pack('bbHHh', ICMP_ECHO_REQUEST, 0, 0, id, 1)
    data = struct.pack('d', time.time())
    chksum = checksum(header + data)
    header = struct.pack('bbHHh', ICMP_ECHO_REQUEST, 0, chksum, id, 1)
    return header + data

def ping(dest_addr, timeout=1):
    try:
        # Create raw socket
        sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_ICMP)
    except PermissionError:
        print("error: Run this script as administrator/root to access raw
sockets.")
        return

    pid = os.getpid() & 0xFFFF
    packet = create_packet(pid)
    try:
        sock.sendto(packet, (dest_addr, 1))
        start_time = time.time()
        sock.settimeout(timeout)
        recv_packet, _ = sock.recvfrom(1024)
        end_time = time.time()
        rtt = (end_time - start_time) * 1000
        print(f"Reply from {dest_addr}: time={round(rtt, 2)}ms")
    except socket.timeout:
        print("Request timed out.")
    finally:
        sock.close()

# Run ping
target = input("Enter the IP address or hostname to ping: ")
print(f"\nPinging {target}...\n")
for i in range(4):
    ping(target)
    time.sleep(1)
```

Result

The simulation successfully sent ICMP Echo Requests and received Echo Replies, demonstrating basic Ping functionality and calculating Round-Trip Time (RTT) for each packet.

Sample Output

```
Enter the IP address or hostname to ping: google.com

Pinging google.com...

Reply from 142.250.182.142: time=18.24ms
Reply from 142.250.182.142: time=19.02ms
Reply from 142.250.182.142: time=18.61ms
Reply from 142.250.182.142: time=17.99ms
```

EXPERIMENT NO: 5

Title: Packet Fragmentation Simulation Using Python

Aim

To simulate IP packet fragmentation using Python, demonstrating how packets larger than the MTU (Maximum Transmission Unit) are split into smaller fragments for transmission at the network layer.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	None
OS	Windows / Linux / macOS

Theory

When an IP packet is larger than the allowed MTU (typically 1500 bytes for Ethernet), it is fragmented into smaller packets.

Important Fields in IP Fragmentation:

- MTU: Max size of a packet that can be transmitted.
- Header Size: Size of IP header (usually 20 bytes).
- Payload Size: MTU - Header size.
- Offset: Position of the fragment in the original data.
- MF (More Fragments): Set to 1 for all fragments except the last one.

Algorithm

1. Accept the following inputs:
    - Total packet size
    - MTU
    - Header size (default = 20 bytes)
2. Calculate maximum payload per fragment = MTU - header size.
3. Determine how many fragments are needed.
4. For each fragment:
    - Compute the size
    - Calculate offset (in 8-byte units)
    - Set the MF (More Fragments) flag accordingly
5. Display the fragmentation table.

Python Program – `packet_fragmentation_simulator.py`

```
import math

def fragment_packet(packet_size, mtu, header_size=20):
    payload_size = mtu - header_size

    if payload_size <= 0:
```

```
        print("Error: MTU must be greater than header size.")
        return

    total_data = packet_size - header_size
    num_fragments = math.ceil(total_data / payload_size)

    print("\nFragmentation Result:")
    print(f"{'Fragment':<10}{'Start Byte':<15}{'End
Byte':<15}{'MF':<5}{'Offset'}")

    offset = 0
    for i in range(1, num_fragments + 1):
        start_byte = offset
        end_byte = start_byte + payload_size - 1

        # Last fragment may be smaller
        if end_byte >= total_data:
            end_byte = total_data - 1
            mf = 0
        else:
            mf = 1

        print(f"{i:<10}{start_byte:<15}{end_byte:<15}{mf:<5}{offset // 8}")

        offset += payload_size

# Input
packet_size = int(input("Enter total packet size (in bytes): "))
mtu = int(input("Enter MTU size (in bytes): "))
header_size = int(input("Enter header size (in bytes) [default=20]: ") or 20)

# Run simulation
fragment_packet(packet_size, mtu, header_size)
```

Result

The simulation successfully fragmented a large packet into smaller fragments based on the given MTU and header size. The output displays offset and MF values as used in real IP fragmentation.

Sample Output

```
Enter total packet size (in bytes): 4000
Enter MTU size (in bytes): 1500
Enter header size (in bytes) [default=20]: 20

Fragmentation Result:
Fragment Start Byte     End Byte       MF   Offset
1         0              1479           1    0
2         1480           2959           1    185
3         2960           3979           0    370
```

EXPERIMENT NO: 6

Title: Simulation of Dijkstra's Algorithm Using Python

Aim

To simulate Dijkstra's Algorithm for finding the shortest path from a source node to all other nodes in a graph, representing a link-state routing protocol.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	None (uses built-in Python structures)
OS	Windows / Linux / macOS

Theory

Dijkstra's Algorithm is used to find the shortest path between nodes in a graph, which may represent, for example, road networks or routers in a computer network.

Used in:

- **Link State Routing Protocols** (e.g., OSPF)

Key Concepts:

- The graph is made up of nodes (routers) and edges (links with weights).
- The shortest path tree is calculated from the source node.
- At each step, the node with the minimum tentative distance is selected.

Algorithm Steps

1. Create a graph with all nodes and edge weights.
2. Initialize distances of all nodes as infinity, except the source node (0).
3. Maintain a set of visited nodes.
4. For the current node:
   - Update distances of its neighbors if a shorter path is found.
5. Mark the node as visited.
6. Repeat until all nodes are visited.

Python Program – `dijkstra_simulator.py`

```
import heapq

def dijkstra(graph, start):
    # Initialize distances with infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Use a priority queue to store (distance, node)
```

```
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_node = heapq.heappop(priority_queue)

        # Skip if this node was already processed with a shorter distance
        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight

            # If a shorter path is found
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example graph as an adjacency list
graph = {
    'A': {'B': 2, 'C': 4},
    'B': {'A': 2, 'C': 1, 'D': 7},
    'C': {'A': 4, 'B': 1, 'E': 3},
    'D': {'B': 7, 'E': 1, 'F': 5},
    'E': {'C': 3, 'D': 1, 'F': 7},
    'F': {'D': 5, 'E': 7}
}

# Input source
source = input("Enter the source node: ").upper()

# Run Dijkstra
if source in graph:
    shortest_paths = dijkstra(graph, source)
    print(f"\nShortest paths from node {source}:")
    for node, distance in shortest_paths.items():
        print(f"{source} -> {node} = {distance}")
else:
    print("Invalid source node.")
```

Result

The program successfully simulates Dijkstra's Algorithm, calculating the shortest path from the source router to all other routers in the given network graph.

Sample Output

```
Enter the source node: A

Shortest paths from node A:
A -> A = 0
A -> B = 2
A -> C = 3
A -> D = 10
A -> E = 6
A -> F = 15
```

EXPERIMENT NO: 7

Title: Simulation of Bellman-Ford Algorithm Using Python

Aim

To simulate the Bellman-Ford Algorithm in Python to find the shortest paths from a source node to all other nodes in a network graph, demonstrating Distance Vector Routing.

Requirements

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	None (uses built-in Python structures)
OS	Windows / Linux / macOS

Theory

The Bellman-Ford Algorithm is used for finding the shortest path in graphs with positive or negative edge weights (but no negative cycles).

Used in:

- Distance Vector Routing Protocols (e.g., RIP)

Key Characteristics:

- Works by relaxing edges repeatedly (|V| - 1 times)
- Can detect negative weight cycles
- Slower than Dijkstra, but more flexible

Algorithm Steps

1. Initialize distance to all nodes as ∞ (except source = 0)
2. For (number of vertices - 1) times:
    - For each edge (u, v):
        - If `distance[u] + weight(u, v) < distance[v]`, update distance[v]
3. Repeat once more to check for negative cycles.
4. If further relaxation is possible, a negative cycle exists.

Python Program – `bellman_ford_simulator.py`

```
def bellman_ford(graph, vertices, source):
    # Step 1: Initialize distances
    distance = {v: float('inf') for v in vertices}
    distance[source] = 0

    # Step 2: Relax edges |V| - 1 times
    for _ in range(len(vertices) - 1):
        for u, v, w in graph:
            if distance[u] != float('inf') and distance[u] + w < distance[v]:
```

```
                distance[v] = distance[u] + w

    # Step 3: Check for negative weight cycles
    for u, v, w in graph:
        if distance[u] != float('inf') and distance[u] + w < distance[v]:
            print("Graph contains a negative weight cycle.")
            return None

    return distance

# Define graph as a list of edges: (u, v, weight)
graph = [
    ('A', 'B', 4),
    ('A', 'C', 2),
    ('B', 'C', 3),
    ('B', 'D', 2),
    ('B', 'E', 3),
    ('C', 'B', 1),
    ('C', 'D', 4),
    ('C', 'E', 5),
    ('E', 'D', -5)
]

# Unique vertices
vertices = {'A', 'B', 'C', 'D', 'E'}

# Input source
source = input("Enter the source node: ").upper()

if source not in vertices:
    print("Invalid source node.")
else:
    result = bellman_ford(graph, vertices, source)
    if result:
        print(f"\nShortest distances from node {source}:")
        for node in sorted(result):
            print(f"{source} -> {node} = {result[node]}")
```

Result

The simulation of the Bellman-Ford algorithm was successfully implemented. The shortest path from the source node to all other nodes was computed, and negative weight cycles were detected if present.

Sample Output

```
Enter the source node: A

Shortest distances from node A:
A -> A = 0
A -> B = 3
A -> C = 2
A -> D = 0
A -> E = 6
```

EXPERIMENT 8:

Title: Simulation of RSA Algorithm Using Python

Aim:

To implement and simulate the RSA algorithm for encryption and decryption using Python.

- Understand the RSA public-key cryptosystem.
- Generate keys and perform encryption and decryption.
- Validate prime inputs and handle ASCII characters in messages.

Requirements:

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	None (uses built-in Python structures)
OS	Windows / Linux / macOS

Theory:

RSA (Rivest–Shamir–Adleman) is a public-key cryptosystem that is widely used for secure data transmission.
It uses:

- Two large prime numbers `p` and `q`.
- Computes `n = p × q` and $\varphi(n) = (p-1)(q-1)$
- Selects a public key exponent `e` such that `gcd(e, φ(n)) = 1`
- Computes the private key exponent `d` such that `(e × d) % φ(n) = 1`

Encryption:

$$C = (M^e) \bmod n$$

Decryption:

$$M = (C^d) \bmod n$$

Key Concepts:

1. Public Key Cryptography: RSA is an asymmetric cryptographic technique using a pair of keys:
   - Public Key (e, n) – used for encryption.
   - Private Key (d, n) – used for decryption.
2. Prime Numbers: Two distinct prime numbers `p` and `q` are selected to generate keys. They ensure the security of the RSA algorithm.
3. Modulus `n`: Computed as `n = p × q`. It is used in both encryption and decryption.
4. Euler's Totient Function `φ (n)`: Calculated as `φ (n) = (p-1) (q-1)`. It determines the number of integers less than `n` that are coprime to `n`.
5. Encryption Exponent `e`: A number chosen such that `1 < e < φ (n)` and `gcd (e, φ (n)) = 1`.

6. Decryption Exponent `d`: The modular multiplicative inverse of `e` modulo `φ (n)`. That is, `(e × d) mod φ (n) = 1`.
7. Encryption & Decryption:
   - Encryption: $C=M^e \bmod n$
   - Decryption: $M=C^d \bmod n$
   - Where `M` is the plaintext message, and `C` is the cipher text.

Algorithm Steps:

Key Generation:

1. Choose two distinct prime numbers `p` and `q`.
2. Compute `n = p × q`.
3. Compute φ (n) = (p − 1) (q − 1).
4. Choose an encryption key `e` such that `1 < e < φ (n)` and `gcd (e, φ (n)) = 1`.
5. Compute the decryption key `d` such that `(e × d) mod φ (n) = 1`.

Result:

- Public Key: (e, n)
- Private Key: (d, n)

Encryption:

1. Convert each character of the plaintext message into its ASCII value.
2. For each character (M), compute: $C=M^e \bmod n$
3. Collect the cipher text values as the encrypted message.

Decryption:

1. For each encrypted character (C), compute: $M=C^d \bmod n$
2. Convert the numeric values back to characters to obtain the original message.

Python Program - `rsa_simulator.py`:

```
# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):  # Efficient primality test
        if n % i == 0:
            return False
    return True

# Function to compute GCD (used to check if e and φ (n) are coprime)
def gcd(a, b):
    while b:
        a, b = b, a % b
    return a

# Function to compute modular inverse (to find private key d)
def mod_inverse(e, phi):
    for d in range(2, phi):
        if (e * d) % phi == 1:
            return d
    return None  # If no modular inverse exists
```

```
# Function to encrypt the message
def encrypt(text, e, n):
    return [pow(ord(char), e, n) for char in text]

# Function to decrypt the cipher
def decrypt(cipher, d, n):
    return ''.join([chr(pow(c, d, n)) for c in cipher])

# Input: Get valid prime number for p
while True:
    p = int(input("Enter prime number p: "))
    if is_prime(p):
        break
    print("p is not a prime. Please enter a valid prime.")

# Input: Get valid prime number for q
while True:
    q = int(input("Enter prime number q: "))
    if is_prime(q):
        break
    print("q is not a prime. Please enter a valid prime.")

# Calculate n and Euler's totient function φ(n)
n = p * q
phi = (p - 1) * (q - 1)

# Select smallest odd e such that gcd(e, phi) = 1
e = next(i for i in range(3, phi, 2) if gcd(i, phi) == 1)

# Find modular inverse of e to get private key d
d = mod_inverse(e, phi)

# Display public and private keys
print(f"\nPublic Key (e, n): ({e}, {n})")
print(f"Private Key (d, n): ({d}, {n})")

# Input message to encrypt
message = input("Enter the message to encrypt: ")

# Encrypt and decrypt
cipher = encrypt(message, e, n)
decrypted = decrypt(cipher, d, n)

# Output results
print("Encrypted message (numeric):", cipher)
print("Decrypted message:", decrypted)
```

Result:

The RSA algorithm was successfully implemented and verified with encryption and decryption of text using manually entered prime numbers.

Sample Output:

```
Enter prime number p: 17
Enter prime number q: 11

Public Key (e, n): (3, 187)
Private Key (d, n): (123, 187)
```

```
Enter the message to encrypt: hi
Encrypted message (numeric): [23, 162]
Decrypted message: hi
```

EXPERIMENT NO: 9

Title: Diffie–Hellman Key Exchange Implementation Using Python

Aim:

To implement the Diffie–Hellman Key Exchange algorithm using Python, demonstrating how two parties can securely establish a shared secret over an insecure channel.

Requirements:

Component	Specification
Hardware	Any system with Python installed
Software	Python 3.x
Libraries	None (uses built-in Python structures)
OS	Windows / Linux / macOS

Theory:

The Diffie–Hellman algorithm is used to securely exchange cryptographic keys over a public channel. It allows two users to generate a shared secret key that can be used for further encryption, without actually transmitting the secret itself.

The algorithm works using modular arithmetic and exponentiation. Both users agree on a large prime number and a base (both public). Each user then selects a private key, computes a corresponding public key, and exchanges it. Using the received public key and their own private key, both users compute the same shared secret key. This key can now be used for secure communication.

Diffie–Hellman is a foundational technique in cryptography and is widely used in secure protocols like HTTPS, SSH, and VPNs.

Key Concepts:

- The Diffie–Hellman Key Exchange algorithm allows two users to securely generate a shared secret key over a public communication channel.
- It is based on the discrete logarithm problem, which is computationally hard to solve.
- The shared secret key can be used later in symmetric encryption algorithms.

Algorithm Steps:

1. Select a large prime number `p` and a primitive root `g` (both public).
2. Each user chooses a private key:
    - User A selects private key `x`
    - User B selects private key `y`
3. Each user computes their public key:
    - `A = g^x mod p`
    - `B = g^y mod p`
4. The public keys are exchanged between the users.
5. Each user computes the shared secret:
    - `SharedKey_A = B^x mod p`
    - `SharedKey_B = A^y mod p`

6. Both users will now have the same shared key, which can be used for secure communication.

Python Program - `diffie_hellman.py`:

```
# Function to compute (base^exponent) % modulus efficiently
def power(base, exponent, modulus):
    return pow(base, exponent, modulus)

# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):  # Check divisibility from 2 to sqrt(n)
        if n % i == 0:
            return False
    return True

# Prompt user to enter a prime number p
p = int(input("Enter a prime number p: "))
while not is_prime(p):
    p = int(input("Invalid input. Enter a prime number p: "))

# Prompt user to enter a primitive root g modulo p
g = int(input("Enter a primitive root g: "))

# Private keys for two users (should be kept secret)
x = int(input("Enter private key for user 1: "))  # Private key x
y = int(input("Enter private key for user 2: "))  # Private key y

# Calculate public keys using: A = g^x mod p, B = g^y mod p
A = power(g, x, p)  # Public key of user 1
B = power(g, y, p)  # Public key of user 2

# Display public keys
print(f"\nUser 1's Public Key (A): {A}")
print(f"User 2's Public Key (B): {B}")

# Each user computes the shared key using other's public key and own private key
# Shared key: (B^x mod p) and (A^y mod p)
shared_key_1 = power(B, x, p)  # Computed by user 1
shared_key_2 = power(A, y, p)  # Computed by user 2

# Display the shared keys
print(f"\nShared Key computed by user 1: {shared_key_1}")
print(f"Shared Key computed by user 2: {shared_key_2}")

# Verify if both shared keys are equal
if shared_key_1 == shared_key_2:
    print("\nKey exchange successful. Shared key established.")
else:
    print("\nKey exchange failed. Shared keys do not match.")
```

Result:

The Diffie–Hellman Key Exchange algorithm was successfully implemented. A common shared key was derived independently by both users over a public channel.

Sample Output:

```
Enter a prime number p: 23
```

```
Enter a primitive root g: 5
Enter private key for user 1: 6
Enter private key for user 2: 15

User 1's Public Key: 8
User 2's Public Key: 2

Shared Key computed by user 1: 2
Shared Key computed by user 2: 2

Key exchange successful. Shared key established.
```