CN&S LAB (21CDL61): PYTHON CODE WALKTHROUGH AND IMPLEMENTATION NOTES

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

Program Files

- `tcp_server.py` – Server-side Python script.
- `tcp_client.py` – Client-side Python script.

Detailed Explanation

Server Program: `tcp_server.py`

```
import socket
```

- This line imports Python's built-in `socket` module, which allows access to the low-level network interface.
- The `socket` module is necessary for creating server and client programs that communicate over TCP/IP or UDP.

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

- `socket.socket()` creates a new socket object.
- `AF_INET` specifies the address family: IPv4.
- `SOCK_STREAM` specifies the socket type: TCP (Transmission Control Protocol), which is connection-oriented and reliable.
- `server_socket` is now a socket that can listen for and accept TCP connections.

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

- This line associates the socket with an IP address and port number.
- `'localhost'` refers to the local machine (127.0.0.1), meaning the server will only accept connections from the same machine.
- `12345` is the port number chosen for the server to listen on (can be any unused port above 1024).

```
server_socket.listen(1)
```

- This puts the server into listening mode, allowing it to accept incoming connections.
- The argument `1` defines the backlog queue—i.e., how many unaccepted connections the system will allow before refusing new ones.

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

- Prints a message indicating the server is ready and waiting for a client to connect.

```
# Accept a connection
conn, addr = server_socket.accept()
```

- Blocks the execution and waits until a client attempts to connect.
- Once a client connects:
  - `conn` becomes a new socket object specific to that client.
  - `addr` contains the address of the connected client (as a tuple: IP address and port).

```
print(f"Connected by {addr}")
```

- Displays the address (IP and port) of the client that just connected.

```
# Receive data from client
data = conn.recv(1024).decode()
```

- `conn.recv(1024)` reads up to 1024 bytes from the client connection.
- `.decode()` converts the received bytes into a string using UTF-8 encoding.
- This is a blocking call—it will wait until data is received.

```
print(f"Received from client: {data}")
```

- Prints the original message received from the client.

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

- Calls the built-in string method `.upper()` to convert all lowercase letters in the string to uppercase.
- The result is stored in `upper_data`.

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

- Converts the uppercase string back into bytes using `.encode()` and sends it to the client.
- This completes the echo transformation (lowercase to uppercase).

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

- `conn.close()` closes the connection to the specific client.
- `server_socket.close()` shuts down the server socket completely—no new connections will be accepted.

Client Program: `tcp_client.py`

```
import socket
```

- Imports the `socket` module to enable networking functionalities in the client script.

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

- Creates a new socket object using IPv4 and TCP protocol.
- This socket will be used to initiate a connection to the server.

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

- Initiates a connection to the server running on IP `'localhost'` (i.e., the same machine) and port `12345`.
- If the server is not running or the port is wrong, this will raise an error.

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

- Prompts the user to input a string that is expected to be in lowercase.
- This is the data to be sent to the server for processing.

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

- Encodes the string to bytes using `.encode()` and sends it through the socket to the server.
- TCP ensures reliable and ordered delivery.

```
# Receive response from server
data = client_socket.recv(1024).decode()
```

- Waits to receive up to 1024 bytes of data from the server.
- Decodes the received byte stream into a human-readable string.

```
print(f"Received from server: {data}")
```

- Displays the uppercase string received from the server, completing the data round trip.

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

- Closes the connection to the server.
- This is essential to free up resources and gracefully terminate the client program.

EXPERIMENT NO: 2

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

Program Files and Detailed Explanation

`tcp_handshake_server.py`

```
import socket
import time
```

- `socket`: Enables network communication using the TCP/IP protocol.
- `time`: Used to insert delays (simulating real-time behavior like waiting for responses).

```
server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- Creates a TCP socket using IPv4 addressing (`AF_INET`) and stream type (`SOCK_STREAM` for TCP).

```
server_socket.bind(('localhost', 12346))
```

- Binds the server to the localhost interface (`127.0.0.1`) on port `12346` for incoming connections.

```
server_socket.listen(2)
```

- Listens for up to 2 client connections (backlog).

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

- Outputs that the server is ready and waiting for incoming client connections.

```
conn, addr = server_socket.accept()
```

- Accepts the incoming connection from a client. It blocks execution until a client connects.
- `conn`: New socket object for communication with the client.
- `addr`: Address of the connecting client.

```
print(f"Server: Received SYN from {addr}")
time.sleep(2)
```

- Simulates receiving a SYN (synchronize) signal from the client (as part of TCP handshake).
- `sleep(2)`: Delay added for simulation purposes.

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

- Simulates sending SYN-ACK to client.

```
data = conn.recv(1024).decode()
```

- Receives the client's ACK message. This completes the 3-way handshake.
- `recv(1024)`: Reads up to 1024 bytes of data.
- `.decode()`: Converts byte data to string.

```
print(f"Server: Received ACK → Handshake complete")
time.sleep(2)
```

- Acknowledges that the TCP connection is now established.

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

- Server receives the actual data/message sent by the client.
- Displays the received message.

```
print("Server: Sending ACK for FIN")
time.sleep(2)
print("Server: Sending FIN")
time.sleep(2)
```

- Simulates TCP connection termination: acknowledging client's FIN and initiating its own FIN.

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

- Sends a `FIN` signal (converted to bytes) to initiate server-side connection closure.

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

- Receives the final ACK from the client confirming termination.

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

- Closes the connection and the main server socket.

`tcp_handshake_client.py`

```
import socket
import time
```

- Imports socket module for network operations and time module for adding delays.

```
client_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

- Creates a TCP socket using IPv4 addressing.

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

- Simulates sending a SYN signal (client initiates connection).
- Adds delay to mimic network response time.

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

- Connects to the server on localhost and port `12346`.
- This sends SYN and receives SYN-ACK implicitly as part of the `connect()` process.

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

- Indicates that SYN-ACK is received from the server (in simulation).

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

- Sends ACK to the server to complete the TCP 3-way handshake.
- Adds a delay before next step.

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

- Sends actual message/data to server after the handshake.
- Adds delay to simulate processing.

```
print("Client: Sending FIN")
time.sleep(2)
```

- Simulates connection termination initiated by the client by sending a FIN (not explicitly done in code but assumed here).

```
fin = client_socket.recv(1024).decode()
```

- Receives the server's FIN message (part of 4-way termination).

```
if fin == "FIN":
    print("Client: Received FIN")
    print("Client: Sending final ACK")
    client_socket.send("ACK".encode())
```

- If FIN is received from the server, sends final ACK to complete the connection termination.

```
client_socket.close()
```

- Closes the client-side socket after communication ends.

EXPERIMENT NO: 3

Title: Socket Programming using UDP – Convert Lowercase to Uppercase

Program Files and Detailed Explanation

Server Program: `udp_server.py`

```
import socket
```

- Imports the `socket` module**,** which provides low-level networking interface for implementing sockets (UDP/TCP).

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

- Creates a UDP socket using `AF_INET` for IPv4 and `SOCK_DGRAM` for datagram communication (i.e., UDP).

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

- Binds the socket to a local IP address and port number.
- `'localhost'` restricts it to only accept connections from the same machine.
- Port `12347` is the communication endpoint.

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

- Displays a message to indicate the server is ready to receive data.

```
# Receive data from client
data, client_addr = server_socket.recvfrom(1024)
```

- Waits for incoming data from any client.
- `recvfrom(1024)` receives up to 1024 bytes.
- Returns the data and the address of the client.

```
print(f"Received from client: {data.decode()}")
```

- Decodes and prints the received data (usually sent as bytes).

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

- Converts the received lowercase message to uppercase.

```
# Send it back to the client
server_socket.sendto(upper_data.encode(), client_addr)
```

- Sends the modified data back to the same client using its address.

```
print(f"Sent to client: {upper_data}")
```

- Displays what was sent back to the client.

Client Program: `udp_client.py`

```
import socket
```

- Imports the socket module for client-side communication.

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

- Creates a UDP client socket.

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

- Specifies the server's IP address and port to which the client will send data.

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

- Takes user input from the keyboard.

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

- Encodes the message into bytes and sends it to the server using `sendto()`.

```
# Receive response
data, _ = client_socket.recvfrom(1024)
```

- Waits for and receives response data from the server.

```
print(f"Received from server: {data.decode()}")
```

- Prints the response after decoding it.

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

- Closes the UDP socket after communication ends.

Implementation Notes

- UDP is connectionless, so no handshake is needed.
- The server listens indefinitely until it receives a message.
- Data is sent/received using `sendto()` and `recvfrom()`.
- Unlike TCP, UDP does not guarantee delivery or order of packets.
- This experiment shows a simple echo server behavior with case transformation.

EXPERIMENT NO: 4

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

Program Files and Detailed Explanation

Program: `ping_simulation.py`

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

- `import socket`: Imports the `socket` module to use low-level networking interfaces.
- `import os`: Used to get the process ID (PID) for creating a unique packet identifier.
- `import struct`: Provides functions to convert between Python values and C structs represented as Python bytes objects.
- `import time`: Used to timestamp the packet and calculate the round-trip time (RTT).

```
ICMP_ECHO_REQUEST = 8
```

- ICMP_ECHO_REQUEST: This constant represents the type field value (8) for an ICMP Echo Request message, which is what a `ping` sends.

Function: `checksum(source_string)`

```
def checksum(source_string):
    """Calculate the checksum of a packet"""
```

- Purpose: To compute the Internet Checksum required by the ICMP packet header.

```
    sum = 0
    max_count = (len(source_string) // 2) * 2
```

- Initializes the `sum` variable.
- `max_count`: Calculates the largest even number ≤ length of the source string, since the checksum is computed 2 bytes at a time.

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

- Iterates over the string two bytes at a time.
- Computes a 16-bit value from two adjacent bytes (in little-endian order) and adds it to `sum`.
- Ensures `sum` remains within 32 bits using a bitwise AND.

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

- If there is a remaining byte (odd-length string), it's added as-is.

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

- Folds 32-bit sum to 16 bits by adding carry bits.
- Applies bitwise NOT to get the final checksum.
- Converts it to network byte order (big-endian) before returning.

Function: `create_packet(id)`

```
def create_packet(id):
    """Create ICMP Echo Request packet"""
```

- Builds a raw ICMP Echo Request packet.

```
    header = struct.pack('bbHHh', ICMP_ECHO_REQUEST, 0, 0, id, 1)
```

- Packs the ICMP header:
  - Type (8), Code (0), Checksum (0 as placeholder), ID, Sequence number.

```
    data = struct.pack('d', time.time())
```

- Packs the current time as payload using double-precision float.

```
    chksum = checksum(header + data)
```

- Calculates checksum on the combined header and data.

```
    header = struct.pack('bbHHh', ICMP_ECHO_REQUEST, 0, chksum, id, 1)
```

- Recreates the header with the correct checksum inserted.

```
    return header + data
```

- Returns the full ICMP packet.

Function: `ping(dest_addr, timeout=1)`

```
def ping(dest_addr, timeout=1):
```

- Sends one ICMP Echo Request to `dest_addr` and waits for a reply with a timeout.

```
    try:
        sock = socket.socket(socket.AF_INET, socket.SOCK_RAW,
socket.IPPROTO_ICMP)
```

- Creates a raw socket with IPv4 addressing (`AF_INET`) and ICMP protocol.
- Note: Requires administrator/root privileges to create raw sockets.

```
    except PermissionError:
        print("error: Run this script as administrator/root to access raw sockets.")
        return
```

- Handles the exception if the script is not run with elevated privileges.

```
    pid = os.getpid() & 0xFFFF
```

- Gets the current process ID and masks it to 16 bits (as ICMP IDs are 16-bit).

```
    packet = create_packet(pid)
```

- Calls the earlier function to create the Echo Request packet.

```
    try:
        sock.sendto(packet, (dest_addr, 1))
```

- Sends the packet to the destination.
- The `1` is a dummy port number since ICMP does not use ports.

```
        start_time = time.time()
        sock.settimeout(timeout)
```

- Records the send time.
- Sets the timeout for receiving a response.

```
        recv_packet, _ = sock.recvfrom(1024)
        end_time = time.time()
```

- Receives a response packet of up to 1024 bytes.
- Records the time when the response is received.

```
        rtt = (end_time - start_time) * 1000
        print(f"Reply from {dest_addr}: time={round(rtt, 2)}ms")
```

- Calculates round-trip time in milliseconds.
- Displays the reply and RTT.

```
    except socket.timeout:
        print("Request timed out.")
```

- Informs the user if no response is received in the allotted time.

```
    finally:
        sock.close()
```

- Closes the socket in all cases to release system resources.

Main Script Execution

```
target = input("Enter the IP address or hostname to ping: ")
print(f"\nPinging {target}...\n")
```

- Prompts the user for a target IP/hostname.
- Displays the target to the user.

```
for i in range(4):
    ping(target)
    time.sleep(1)
```

- Sends 4 ping requests at 1-second intervals, similar to the standard `ping` command behavior.

Implementation Notes

- Permissions: Must be run with administrative privileges due to use of raw sockets.
- ICMP Format: The packet manually constructs an ICMP Echo Request (type=8, code=0).
- Checksum: Follows the standard Internet checksum algorithm used in ICMP.
- Timing: Round-trip time is measured by comparing send and receive timestamps.
- Socket Options: Uses raw sockets specifically for `IPPROTO_ICMP`.
- Port Number: Although a port number is included in `sendto`, it is ignored by ICMP.

EXPERIMENT NO: 5

Title: Packet Fragmentation Simulation Using Python

Program Files and Detailed Explanation

`packet_fragmentation_simulator.py`:

```
import math
```

- Imports the `math` module to use `math.ceil()` for rounding up during fragmentation calculation.

```
def fragment_packet(packet_size, mtu, header_size=20):
```

- Defines a function named `fragment_packet` that takes three parameters:
  - `packet_size`: Total size of the IP packet (including header),
  - `mtu`: Maximum Transmission Unit size,
  - `header_size`: Size of the IP header (default is 20 bytes).

```
    payload_size = mtu - header_size
```

- Calculates the payload that can be carried in each fragment after excluding the header.

```
    if payload_size <= 0:
        print("Error: MTU must be greater than header size.")
        return
```

- Checks if the payload size is valid. If the MTU is less than or equal to the header size, fragmentation is not possible, and the function exits with an error message.

```
    total_data = packet_size - header_size
```

- Extracts only the data portion from the full packet size (excluding the original packet's header).

```
    num_fragments = math.ceil(total_data / payload_size)
```

- Calculates the number of fragments needed by dividing the total data by the payload size and rounding it up.

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

- Prints the table headers for the fragmentation output.

```
    offset = 0
```

- Initializes `offset`, which keeps track of the position of data in the original packet.

```
    for i in range(1, num_fragments + 1):
```

- Loops over each fragment using a 1-based index.

```
        start_byte = offset
        end_byte = start_byte + payload_size - 1
```

- Calculates the starting and ending byte positions of the current fragment.

```
        if end_byte >= total_data:
            end_byte = total_data - 1
            mf = 0
```

- Checks if this is the last fragment. If so:
  - Adjusts `end_byte` to avoid going past total data,
  - Sets `mf` (More Fragments flag) to 0.

```
        else:
            mf = 1
```

- If it's not the last fragment, sets the More Fragments (`mf`) flag to 1.

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

- Prints the fragment number, start and end bytes, More Fragments flag, and the offset (divided by 8 as required by IP header format).

```
        offset += payload_size
```

- Moves the offset forward by the payload size for the next fragment.

```
# 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)
```

- Takes user input for packet size, MTU, and optionally header size. Uses default value 20 if header input is blank.

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

- Calls the `fragment_packet` function with the user-provided values to simulate the fragmentation process.

EXPERIMENT NO: 6

Title: Simulation of Dijkstra's Algorithm Using Python

Program Files and Detailed Explanation

`dijkstra_simulator.py`

```
import heapq
```

- Imports the `heapq` module, which provides an implementation of the min-heap queue.
- It is used here to always retrieve the node with the smallest tentative distance efficiently.

```
def dijkstra(graph, start):
```

- Defines the `dijkstra()` function which takes:
  - `graph`: A dictionary representing the graph in adjacency list form.
  - `start`: The starting node for calculating shortest paths.

```
    # Initialize distances with infinity
    distances = {node: float('inf') for node in graph}
```

- Initializes the `distances` dictionary to keep track of the shortest known distance from `start` to every other node.
- Initially, all distances are set to infinity (`float('inf')`).

```
    distances[start] = 0
```

- The distance to the starting node is zero (0), since there's no cost to reach itself.

```
    # Use a priority queue to store (distance, node)
    priority_queue = [(0, start)]
```

- A priority queue (min-heap) is initialized with a tuple: (distance, node) = (0, start).
- This ensures the node with the smallest distance is processed first.

```
    while priority_queue:
```

- Loop continues as long as there are nodes to process in the priority queue.

```
        current_distance, current_node = heapq.heappop(priority_queue)
```

- Pops the node with the smallest distance from the heap.
- `current_node`: the node to process next.
- `current_distance`: its tentative shortest distance from `start`.

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

- Optimization: Skip processing if a shorter path to this node was already found and recorded in `distances`.

```
for neighbor, weight in graph[current_node].items():
```

- For every neighboring node and its corresponding edge weight from the current node.

```
distance = current_distance + weight
```

- Calculates the new potential distance to reach the neighbor via the current node.

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

- If this newly calculated distance is less than the previously known shortest distance:
  - Update the `distances` dictionary.
  - Add the neighbor to the priority queue with its updated distance, so it can be processed in the future.

```
return distances
```

- After all nodes have been processed, return the `distances` dictionary containing the shortest paths from `start`.

Graph Representation

```
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}
}
```

- This is the adjacency list of the graph.
- Each key is a node, and the value is a dictionary of neighboring nodes and the edge weights to them.

User Input & Execution

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

- Prompts the user to enter the starting node for Dijkstra's algorithm.
- Converts the input to uppercase to match the keys in the graph.

```
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.")
```

- If the user input matches a node in the graph:
  - Calls the `dijkstra()` function with the selected source.
  - Prints all shortest distances from the source to each reachable node.
- If the source is not valid, an error message is displayed.

EXPERIMENT NO: 7

Title: Simulation of Bellman-Ford Algorithm Using Python

Program Files and Detailed Explanation

`Bellman_ford.py`

```
def bellman_ford(graph, vertices, source):
```

- Defines a function named `bellman_ford` which takes three arguments:
- `graph`: a list of all edges in the format `(u, v, w)`
- `vertices`: a set of all nodes in the graph
- `source`: the node from which shortest distances will be calculated

```
    distance = {v: float('inf') for v in vertices}
```

- Initializes a dictionary `distance`
- Each key is a vertex from the graph
- Each value is initially set to `float('inf')` meaning "infinite distance"
- Represents that initially, all vertices are unreachable from the source

```
    distance[source] = 0
```

- Sets the distance to the source vertex as `0`
- Because the distance from the source to itself is always zero

```
    for _ in range(len(vertices) - 1):
```

- Runs the outer loop exactly `V - 1` times where `V` is the number of vertices
- This is required in Bellman-Ford to ensure all shortest paths are computed correctly
- Each iteration is called a relaxation pass

```
        for u, v, w in graph:
```

- Begins a loop through each edge in the graph
  Each edge is represented by a tuple `(u, v, w)`
- `u`: starting vertex of the edge
- `v`: ending vertex of the edge
- `w`: weight of the edge from `u` to `v`

```
            if distance[u] != float('inf') and distance[u] + w < distance[v]:
```

- Two conditions are checked:
- `distance[u] != float('inf')`: confirms vertex `u` is already reachable
- `distance[u] + w < distance[v]`: checks whether a new shorter distance to `v` through `u` is found

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

- Updates the distance to vertex `v` with the new shorter distance
- The new value is the sum of the distance to `u` and the weight of the edge `(u, v)`

```
    for u, v, w in graph:
```

- Starts a second loop through all edges
- This loop is used to detect negative weight cycles in the graph

```
        if distance[u] != float('inf') and distance[u] + w < distance[v]:
```

- Checks if a shorter path to `v` can still be found after `V - 1` iterations
- If true, then a negative weight cycle exists
- Because Bellman-Ford guarantees no further relaxation should be possible after `V - 1` passes

```
            print("Graph contains a negative weight cycle.")
```

- If a negative cycle is detected, prints a message indicating this
- Negative cycles make shortest path computation unreliable

```
            return None
```

- Exits the function early, returning `None`
- No valid shortest path distances can be returned in the presence of negative cycles

```
    return distance
```

- If no negative cycles are detected, the dictionary `distance` is returned
- This dictionary holds the shortest distances from the `source` to all other vertices

Main Program (Caller)

```
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)
]
```

- Defines the graph as a list of edges in the form `(u, v, w)`
- Each tuple represents a directed edge with a specific weight
- Edge from `E` to `D` has a negative weight `-5`, used to test detection of negative cycles

```
vertices = {'A', 'B', 'C', 'D', 'E'}
```

- Defines the set of all vertices (nodes) in the graph
- Used to initialize distance table and control the loop iterations

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

- Takes user input to specify the source vertex
- `.upper()` ensures the input is converted to uppercase to match graph node names

```
if source not in vertices:
```

- Checks whether the user-input source node exists in the graph

```
    print("Invalid source node.")
```

- Prints an error message if the entered source node is not in the graph

```
else:
```

- If the entered source node is valid, continue execution

```
    result = bellman_ford(graph, vertices, source)
```

- Calls the `bellman_ford` function
- Passes the defined graph, set of vertices, and user-specified source node
- Stores the result (shortest path distances) in the variable `result`

```
    if result:
```

- Checks if `result` is not `None`
- If `None`, it means a negative weight cycle was detected
- If not `None`, valid distances are available

```
        print(f"\nShortest distances from node {source}:")
```

- Prints a heading for displaying the computed shortest path distances

```
        for node in sorted(result):
```

- Iterates through all nodes in the `result` dictionary
- Nodes are sorted alphabetically for consistent output order

```
            print(f"{source} -> {node} = {result[node]}")
```

- Prints the shortest distance from the source node to the current node
- Format: `source -> destination = distance`

EXPERIMENT 8:

Title: Simulation of RSA Algorithm Using Python

Program Files and Detailed Explanation

`rsa_simulator.py`

```
# Function to check if a number is prime
def is_prime(n):
    if n <= 1:
        return False
```

- This defines a function `is_prime(n)` to check whether a number is prime.
- If `n` is 1 or less, it returns `False` immediately, because prime numbers are greater than 1.

```
    for i in range(2, int(n**0.5) + 1):  # Efficient primality test
        if n % i == 0:
            return False
```

- It loops from 2 to the square root of `n` (inclusive).
- If any number `i` divides `n` without remainder, then `n` is not prime.
- This method improves efficiency compared to checking all numbers up to `n`.

```
    return True
```

- If no divisor is found, `n` is a prime number.

```
# 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
```

- This is the Euclidean Algorithm for computing the Greatest Common Divisor.
- It continues swapping `a` and `b` with `b` and `a % b` until `b` becomes 0.
- The final value of `a` is the GCD.

```
# 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
```

- This function finds the modular inverse of `e` modulo `phi`.
- It loops through all possible values `d` from 2 to `phi - 1`.
- If `(e * d) % phi == 1`, then `d` is the modular inverse.
- If no such `d` exists, it returns `None`.

```
# Function to encrypt the message
def encrypt(text, e, n):
```

```
    return [pow(ord(char), e, n) for char in text]
```

- This function encrypts a string message `text`.
- `ord(char)` converts each character to its ASCII value.
- `pow(ord(char), e, n)` performs modular exponentiation: `(ASCII value)^e mod n`.
- Returns a list of encrypted numbers (cipher text).

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

- Decrypts the list of cipher integers using the private key `d` and modulus `n`.
- `pow(c, d, n)` reverses the encryption using modular exponentiation.
- `chr()` converts the resulting number back to a character.
- `join()` combines all characters into the original string.

```
# 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.")
```

- Asks the user to enter a prime number `p`.
- Keeps asking until a valid prime is given using the `is_prime` function.

```
# 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.")
```

- Repeats the same process for the second prime number `q`.

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

- `n` is the modulus for the public and private keys.
- `phi` is Euler's Totient Function, used in key generation.
- It is `(p-1)(q-1)` for prime `p` and `q`.

```
# 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)
```

- Finds the smallest odd number `e` (starting from 3) that is coprime with `phi`.
- Uses `gcd(i, phi)` to check coprimality.
- `next(...)` returns the first valid `e`.

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

- Calls `mod_inverse` to find `d`, the modular inverse of `e` modulo `phi`.
- This `d` becomes the private key exponent.

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

- Prints the RSA public and private key pairs.

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

- Prompts the user for a message (string) to encrypt.

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

- Calls `encrypt()` with the public key to get the cipher.
- Calls `decrypt()` with the private key to get the original message.

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

- Displays the encrypted form as a list of numbers.
- Displays the decrypted message to confirm correctness.

EXPERIMENT NO: 9

Title: Diffie–Hellman Key Exchange Implementation Using Python

Program Files and Detailed Explanation

`diffie_hellman.py`

Function: Modular Exponentiation

```
def power(base, exponent, modulus):
    return pow(base, exponent, modulus)
```

- Purpose: Computes `(base^exponent) % modulus` efficiently using Python's built-in `pow()` function.
- Why: This is essential in cryptographic calculations to handle very large exponents without overflow.

Function: Prime Number Check

```
def is_prime(n):
    if n <= 1:
        return False
    for i in range(2, int(n**0.5) + 1):
        if n % i == 0:
            return False
    return True
```

- Purpose: Checks if a number `n` is a prime.
- Logic:
  - Any number ≤ 1 is not prime.
  - Loop checks for divisibility from 2 to √n.
  - If any number divides `n`, it's not prime.

Input: 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: "))
```

- Asks user to enter a valid prime number `p`.
- Repeats input until a valid prime is entered.

Input: Primitive Root `g`

```
g = int(input("Enter a primitive root g: "))
```

- The base `g` (generator) is used in the exponential calculation.
- Should ideally be a primitive root modulo p, though the program doesn't validate this.
- In real cryptography, `g` must be carefully selected.

Input: Private Keys

```
x = int(input("Enter private key for user 1: "))
y = int(input("Enter private key for user 2: "))
```

- Users 1 and 2 enter their private keys `x` and `y` respectively.
- These values are kept secret and never shared.

Compute Public Keys

```
A = power(g, x, p)  # Public key of user 1
B = power(g, y, p)  # Public key of user 2
```

- User 1 computes public key `A = g^x mod p`.
- User 2 computes public key `B = g^y mod p`.
- These public keys are shared openly over insecure channels.

Display Public Keys

```
print(f"\nUser 1's Public Key (A): {A}")
print(f"User 2's Public Key (B): {B}")
```

- Shows the computed public keys `A` and `B`.

Compute Shared Secret Keys

```
shared_key_1 = power(B, x, p)
shared_key_2 = power(A, y, p)
```

- User 1 computes `shared_key_1 = B^x mod p`
- User 2 computes `shared_key_2 = A^y mod p`
- Mathematically, both are equal to: `g^(xy) mod p`

Display Shared Secret Keys

```
print(f"\nShared Key computed by user 1: {shared_key_1}")
print(f"Shared Key computed by user 2: {shared_key_2}")
```

- Each user prints their version of the shared secret.

Verify the Shared Key

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

- Verifies if the computed shared secrets are identical.
- If yes → Secure key exchange is successful.
- If not → There's an error or mismatch in inputs.