Name:Maryam Mehreen
Roll Number:35889
Artificial Intelligence Lab

Q#1

Solution:

```python
graph = {
    'A': ['B', 'C'],
    'B': ['D'],
    'C': ['E'],
    'D': ['C', 'E'],
    'E': []
}

def find_path(graph, start, end, path=[]):
    path = path + [start]   # Add the start node to the current path
    print(f"Current path: {path}")  # Debug print to trace the path

    # If we reach the destination node, return the path
    if start == end:
        return path

    # If the starting node has no neighbors (or does not exist in the graph)
    if start not in graph:
        return None

    # Explore each neighbor of the current node
    for node in graph[start]:
        if node not in path:  # Avoid cycles by skipping nodes already in the path
            new_path = find_path(graph, node, end, path)  # Recursive call
            if new_path:
                return new_path  # Return the first valid path found
    return None

print(find_path(graph,'A','D'))
```

Output:

```
Current path: ['A']
Current path: ['A', 'B']
Current path: ['A', 'B', 'D']
['A', 'B', 'D']
```

Task#2

```
[ ] import heapq

    class Graph:
        def __init__(self):
            self.graph = {}
            self.weights = {}

        def add_edge(self, u, v, weight=1, directed=False):
            # Add the edge to the graph
            if u not in self.graph:
                self.graph[u] = []
            self.graph[u].append(v)

            if directed:
                # For directed graphs, add weight
                if (u, v) not in self.weights:
                    self.weights[(u, v)] = weight
            else:
                # For undirected graphs, add the edge in both directions
                if v not in self.graph:
                    self.graph[v] = []
                self.graph[v].append(u)
                if (u, v) not in self.weights:
                    self.weights[(u, v)] = weight
                if (v, u) not in self.weights:
                    self.weights[(v, u)] = weight

        def neighbors(self, node):
            return self.graph.get(node, [])
```

```python
def neighbors(self, node):
    return self.graph.get(node, [])

def edge_exists(self, u, v):
    return v in self.graph.get(u, [])

def dijkstra(self, start, end):
    # Implement Dijkstra's algorithm to find the shortest path
    priority_queue = [(0, start)]  # (distance, node)
    distances = {node: float('inf') for node in self.graph}
    distances[start] = 0
    previous_nodes = {node: None for node in self.graph}

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

        if current_distance > distances[current_node]:
            continue

        for neighbor in self.neighbors(current_node):
            weight = self.weights.get((current_node, neighbor), float('inf'))
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_nodes[neighbor] = current_node
                heapq.heappush(priority_queue, (distance, neighbor))

    # Reconstruct the shortest path
    path, current_node = [], end
    while current_node is not None:
```

```python
        # Reconstruct the shortest path
        path, current_node = [], end
        while current_node is not None:
            path.append(current_node)
            current_node = previous_nodes[current_node]
        path.reverse()

        return path if distances[end] != float('inf') else None


# Example Usage
if __name__ == "__main__":
    g = Graph()

    # Add edges (directed, undirected, and weighted)
    g.add_edge('A', 'B', weight=2, directed=True)
    g.add_edge('A', 'C', weight=3, directed=True)
    g.add_edge('B', 'D', weight=1, directed=True)
    g.add_edge('C', 'D', weight=4, directed=True)
    g.add_edge('D', 'E', weight=2, directed=False)  # Undirected

    # Find neighbors of node 'A'
    print(f"Neighbors of A: {g.neighbors('A')}")

    # Check if edge exists between 'A' and 'B'
    print(f"Edge exists between A and B: {g.edge_exists('A', 'B')}")
    print(f"Edge exists between B and A: {g.edge_exists('B', 'A')}")

    # Find shortest path from 'A' to 'E'
    shortest_path = g.dijkstra('A', 'E')
    if shortest_path:
```

```python
if shortest_path:
    print(f"Shortest path from A to E: {' -> '.join(shortest_path)}")
else:
    print("No path found from A to E.")
```

Output:

```
Neighbors of A: ['B', 'C']
Edge exists between A and B: True
Edge exists between B and A: False
Shortest path from A to E: A -> B -> D -> E
```