# TASK 1:

graph = [('A', 'B', 'E', 'C'),  
 ('B', 'A', 'E', 'D'),  
 ('C', 'F', 'G'),  
 ('D', 'B', 'E'),  
 ('E', 'B', 'D'),  
 ('F', 'C'),  
 ('G', 'C')]  
  
  
nodes = sorted(set(node for edge in graph for node in edge))  
  
# Create an empty adjacency matrix filled with zeros  
adjacency\_matrix = [[0] \* len(nodes) for \_ in range(len(nodes))]  
  
# Populate the adjacency matrix based on the edges in the graph  
for edge in graph:  
 from\_node = edge[0]  
 for to\_node in edge[1:]:  
 from\_index = nodes.index(from\_node)  
 to\_index = nodes.index(to\_node)  
 adjacency\_matrix[from\_index][to\_index] = 1  
  
# Print the adjacency matrix  
for row in adjacency\_matrix:  
 print(' '.join(map(str, row)))

# OUTPUT:

# 

# TASK 3:

from queue import PriorityQueue  
from queue import deque  
  
graph = {  
 'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},  
 'Oradea': {'Zerind': 71, 'Sibiu': 151},  
 'Zerind': {'Oradea': 71, 'Arad': 75},  
 'Timisoara': {'Arad': 118, 'Lugoj': 111},  
 'Lugoj': {'Mehadia': 70, 'Timisoara': 111},  
 'Mehadia': {'Drobeta': 75, 'Lugoj': 70},  
 'Drobeta': {'Mehadia': 75, 'Craiova': 120},  
 'Sibiu': {'Oradea': 151, 'Arad': 140, 'Fagaras': 99, 'Rimnicu Vilcea': 80},  
 'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},  
 'Craiova': {'Drobeta': 120, 'Pitesti': 138, 'Rimnicu Vilcea': 146},  
 'Fagaras': {'Sibiu': 99, 'Bucharest': 211},  
 'Pitesti': {'Craiova': 138, 'Rimnicu Vilcea': 97, 'Bucharest': 101},  
 'Neamt': {'Iasi': 87},  
 'Iasi': {'Neamt': 87, 'Vaslui': 92},  
 'Vaslui': {'Urziceni': 142, 'Iasi': 92},  
 'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},  
 'Bucharest': {'Fagaras': 211, 'Giurgiu': 90, 'Pitesti': 101, 'Urziceni': 85},  
 'Giurgiu': {'Bucharest': 90},  
 'Hirsova': {'Eforie': 86, 'Urziceni': 98},  
 'Eforie': {'Hirsova': 86}  
}  
  
  
def ucs(graph, start, goal):  
 queue = PriorityQueue()  
 queue.put((0, [start]))  
  
 while not queue.empty():  
 cost, path = queue.get()  
 current\_node = path[-1]  
  
 if current\_node == goal:  
 return path, cost  
  
 for neighbor, edge\_cost in graph[current\_node].items():  
 new\_path = path + [neighbor]  
 new\_cost = cost + edge\_cost  
 queue.put((new\_cost, new\_path))  
  
  
start\_node = 'Arad'  
goal\_node = 'Bucharest'  
shortest\_path, total\_cost = ucs(graph, start\_node, goal\_node)  
  
  
print(f"Shortest path from {start\_node} to {goal\_node}:", shortest\_path)  
print("Total cost:", total\_cost)  
  
  
def bfs(graph, start, goal):  
 queue = deque([(start, [start])])  
 visited = set()  
  
 while queue:  
 current\_node, path = queue.popleft()  
  
 if current\_node == goal:  
 return path, len(visited) + 1, len(queue)  
  
 if current\_node not in visited:  
 visited.add(current\_node)  
 queue.extend((neighbor, path + [neighbor]) for neighbor in graph[current\_node])  
  
 return None, len(visited) + 1, len(queue)  
  
  
bfs\_path, bfs\_time\_complexity, bfs\_space\_complexity = bfs(graph, start\_node, goal\_node)  
  
print("\nBreadth-First Search:")  
print("Path:", bfs\_path)  
print("Time complexity:", bfs\_time\_complexity)  
print("Space complexity:", bfs\_space\_complexity)  
  
  
def dfs(graph, start, goal):  
 queue = [(start, [start])]  
 visited = set()  
  
 while queue:  
 current\_node, path = queue.pop()  
  
 if current\_node == goal:  
 return path, len(visited) + 1, len(queue)  
  
 if current\_node not in visited:  
 visited.add(current\_node)  
 queue.extend((neighbor, path + [neighbor]) for neighbor in graph[current\_node])  
  
 return None, len(visited) + 1, len(queue)  
  
  
dfs\_path, dfs\_time\_complexity, dfs\_space\_complexity = dfs(graph, start\_node, goal\_node)  
  
print("\nDepth-First Search:")  
print("Path:", dfs\_path)  
print("Time complexity:", dfs\_time\_complexity)  
print("Space complexity:", dfs\_space\_complexity)

# OUTPUT: