Ass 1 bfs

```
from collections import deque

def bfs(adjList, startNode, visited):
    # Create a queue for BFS
    q = deque()

    # Mark the current node as visited and enqueue it
    visited[startNode] = True
    q.append(startNode)

    # Iterate over the queue
    while q:
        # Dequeue a vertex from queue and print it
        currentNode = q.popleft()
        print(currentNode, end=" ")

        # Get all adjacent vertices of the dequeued vertex
        # If an adjacent has not been visited, then mark it visited and enqueue
it
        for neighbor in adjList[currentNode]:
            if not visited[neighbor]:
                visited[neighbor] = True
                q.append(neighbor)


def addEdge(adjList, u, v):
    adjList[u].append(v)


# Number of vertices in the graph
vertices = 5

# Adjacency list representation of the graph
adjList = [[] for _ in range(vertices)]

# Add edges to the graph
addEdge(adjList, 0, 1)
addEdge(adjList, 0, 2)
addEdge(adjList, 1, 3)
addEdge(adjList, 1, 4)
addEdge(adjList, 2, 4)

# Mark all the vertices as not visited
visited = [False] * vertices

# Perform BFS traversal starting from vertex 0
print("Breadth First Traversal starting from vertex 0:", end=" ")
bfs(adjList, 0, visited)
```

Ass1 DFS

```
def dfs(adjList, startNode, visited):
    # Create a stack for DFS
    stack = []

    # Mark the current node as visited and push it onto the stack
    visited[startNode] = True
    stack.append(startNode)

    # Iterate over the stack
    while stack:
        # Pop a vertex from stack and print it
        currentNode = stack.pop()
        print(currentNode, end=" ")

        # Get all adjacent vertices of the popped vertex
        # If an adjacent has not been visited, then mark it visited and push it
onto the stack
        for neighbor in adjList[currentNode]:
            if not visited[neighbor]:
                visited[neighbor] = True
                stack.append(neighbor)

# Add edge
def addEdge(adjList, u, v):
    adjList[u].append(v)



# Number of vertices in the graph
vertices = 5

# Adjacency list representation of the graph
adjList = [[] for _ in range(vertices)]

# Add edges to the graph
addEdge(adjList, 0, 1)
addEdge(adjList, 0, 2)
addEdge(adjList, 1, 3)
addEdge(adjList, 1, 4)
addEdge(adjList, 2, 4)

# Mark all the vertices as not visited
visited = [False] * vertices

# Perform DFS traversal starting from vertex 0
print("Depth First Traversal starting from vertex 0:", end=" ")
dfs(adjList, 0, visited)
```

Ass 2 A star

```
from collections import deque

class Graph:
    def __init__(self, adjac_list):
        self.adjac_list = adjac_list

    def get_neighbors(self, v):
        return self.adjac_list[v]

    def h(self, n):
        # Heuristic function should return heuristic value for node n
        # This can vary depending on the problem domain
        H = {'A': 1, 'B': 1, 'C': 1, 'D': 1}
        return H[n]

    def A_Star(self, start, stop):
        open_list = set([start])  # Set of nodes which have been visited but
whose neighbors haven't been evaluated
        closed_list = set()       # Set of nodes whose neighbors have been
evaluated
        distances = {}            # Dictionary to store distances from start node
to other nodes
        distances[start] = 0
        parent = {}               # Dictionary to store parent nodes in the path
        parent[start] = start

        while open_list:
            current_node = None

            # Find node with the lowest value of f(n) = g(n) + h(n)
            for v in open_list:
                if current_node is None or (distances[v] + self.h(v)) <
(distances[current_node] + self.h(current_node)):
                    current_node = v

            if current_node is None:
                print('Path does not exist!')
                return None

            # If the current node is the goal node, reconstruct and return the
path
            if current_node == stop:
                reconst_path = []
                while parent[current_node] != current_node:
                    reconst_path.append(current_node)
                    current_node = parent[current_node]
                reconst_path.append(start)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path

            open_list.remove(current_node)
            closed_list.add(current_node)
```

```
            # Explore neighbors of the current node
            for neighbor, weight in self.get_neighbors(current_node):
                if neighbor in closed_list:
                    continue

                tentative_distance = distances[current_node] + weight
                if neighbor not in open_list or tentative_distance <
distances[neighbor]:
                    parent[neighbor] = current_node
                    distances[neighbor] = tentative_distance
                    open_list.add(neighbor)

        print('Path does not exist!')
        return None

# Input
adjac_list = {'A': [('B', 1), ('C', 3), ('D', 7)], 'B': [('D', 5)], 'C': [('D',
12)]}
graph1 = Graph(adjac_list)
graph1.A_Star('A', 'D')
```

Ass 3 Dijkrstra

```python
import heapq

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

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

        for neighbor, weight in graph[current_node].items():
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(priority_queue, (distance, neighbor))

    return distances

# Example usage with user input:

graph = {}
n = int(input("Enter the number of nodes: "))
for _ in range(n):
    node = input("Enter node: ")
    graph[node] = {}
    m = int(input("Enter the number of neighbors of node {}: ".format(node)))
    for _ in range(m):
        neighbor, weight = input("Enter neighbor and weight (separated by space): ").split()
        graph[node][neighbor] = int(weight)

start_node = input("Enter the start node: ")

shortest_distances = dijkstra(graph, start_node)
print("Shortest distances from node", start_node, "to other nodes:", shortest_distances)
```

Ass 4 Queens

```
def is_safe(board, row, col, n):
    # Check if there is a queen in the same column
    for i in range(row):
        if board[i] == col:
            return False

        # Check diagonals
        if abs(board[i] - col) == row - i:
            return False

    return True

def solve_n_queens_backtracking(n):
    def backtrack(row):
        if row == n:
            return True

        for col in range(n):
            if is_safe(board, row, col, n):
                board[row] = col
                if backtrack(row + 1):
                    return True

        return False

    board = [-1] * n
    if backtrack(0):
        return board
    else:
        return None

def print_solution(board):
    if board is None:
        print("No solution exists.")
    else:
        for row in board:
            print(" ".join('Q' if i == row else '.' for i in range(len(board))))

# Example usage:
n = int(input("Enter the number of Queens :"))  # Change this to the desired
number of queens
solution = solve_n_queens_backtracking(n)
print("Backtracking Solution:")
print_solution(solution)
```

Ass 5

```
import random

class RestaurantChatbot:
    def __init__(self):
        self.menu_card_message = """
        Today's Menu:

        1. Appetizers:
            - Caesar Salad
            - Garlic Bread
            - Spring Rolls

        2. Main Course:
            - Grilled Salmon
            - Chicken Alfredo Pasta
            - Beef Steak

        3. Desserts:
            - Chocolate Lava Cake
            - Tiramisu
            - Fruit Tart

        4. Beverages:
            - Fresh Orange Juice
            - Iced Tea
            - Cappuccino

        Enjoy your meal!
                        """
        self.responses = {
            "greeting": ["Hello! Welcome to our restaurant.", "Hi there! How can I assist you today?"],
            "menu": [r"Sure! Here's our menu:", "Let me tell you about our delicious dishes:"],
            "reservation": ["Certainly! I can help you with that. When would you like to make a reservation?"],
            "thanks": ["You're welcome!", "No problem! Enjoy your meal!"]
        }

    def get_response(self, message):
        message = message.lower()  # Convert message to lowercase for case-insensitive matching
        if "hi" in message or "hello" in message:
            return random.choice(self.responses["greeting"])
        elif "menu" in message:
            return random.choice(self.responses["menu"]) + self.menu_card_message
        elif "reservation" in message:
            return random.choice(self.responses["reservation"])
        elif "thank" in message:
            return random.choice(self.responses["thanks"])
        else:
            return "I'm sorry, I didn't quite catch that. How can I assist you?"

# Example usage:
```

```python
chatbot = RestaurantChatbot()

# print(chatbot.get_response("Hi"))
# print(chatbot.get_response("Can I see the menu?"))
# print(chatbot.get_response("I'd like to make a reservation, please."))
# print(chatbot.get_response("Thanks for your help!"))

print('Hi I am a restaurant chatbot! \nFeel free to ask me any questions')
while True:
    ans = input("")
    print(chatbot.get_response(ans))
    if ans == "thank":
        break
```

Ass 6  Expert system

```python
class ExpertSystem:
    def __init__(self):
        self.rules = {
            ("runny nose", "sneezing", "cough"): "cold",
            ("fever", "muscle aches", "fatigue"): "flu",
            ("itchy eyes", "rash", "sneezing"): "allergy"
        }

    def diagnose(self, symptoms):
        for symptoms_list, disease in self.rules.items():
            if all(symptom in symptoms for symptom in symptoms_list):
                #
                return f"You may have {disease.capitalize()}."
        return "Your symptoms do not match any known diseases."

# Example usage:

expert_system = ExpertSystem()

patient_symptoms = input("Enter your symptoms (comma-separated): ").split(", ")
diagnosis = expert_system.diagnose(patient_symptoms)
print(diagnosis)
```