✅ **Graph Traversal using DFS (Recursive) and BFS with User Input**

**INPUT :-**

```
from collections import defaultdict, deque

class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # because it's an undirected graph

    def dfs_recursive(self, v, visited=None):
        if visited is None:
            visited = set()
        visited.add(v)
        print(v, end=' ')
        for neighbor in self.graph[v]:
            if neighbor not in visited:
                self.dfs_recursive(neighbor, visited)

    def bfs(self, start):
        visited = set()
        queue = deque([start])
        visited.add(start)

        while queue:
            vertex = queue.popleft()
            print(vertex, end=' ')
            for neighbor in self.graph[vertex]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)

# Main execution starts here
g = Graph()

# Input from user
num_edges = int(input("Enter number of edges: "))
print("Enter each edge as two space-separated vertices (e.g., 0 1):")
for _ in range(num_edges):
    u, v = map(int, input().split())
    g.add_edge(u, v)
```

```
start_node = int(input("Enter the starting node for DFS and BFS: "))

print("\nDFS Traversal:")
g.dfs_recursive(start_node)

print("\n\nBFS Traversal:")
g.bfs(start_node)
```

**OUTPUT :-**

```
Enter number of edges: 5
Enter each edge as two space-separated vertices (e.g., 0 1):
0 1
0 2
1 3
2 4
3 4
Enter the starting node for DFS and BFS: 0

DFS Traversal:
0 1 3 4 2

BFS Traversal:
0 1 2 3 4
```

✅ ***A* Algorithm with User Input***

**INPUT :-**

```
from queue import PriorityQueue

def a_star(graph, start, goal, h):
    open_set = PriorityQueue()
    open_set.put((0, start))
    came_from = {}
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0

    while not open_set.empty():
        _, current = open_set.get()

        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            path.append(start)
            return path[::-1]

        for neighbor, cost in graph[current].items():
            tentative_g = g_score[current] + cost
            if tentative_g < g_score[neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + h[neighbor]
                open_set.put((f_score, neighbor))

    return []

# User input
graph = {}
nodes = input("Enter all nodes (space-separated): ").split()

for node in nodes:
    graph[node] = {}
    neighbors = input(f"Enter neighbors of {node} with cost (format: B:2 C:5), or press
Enter if none: ")
    if neighbors:
        for entry in neighbors.split():
```

```
            neighbor, cost = entry.split(":")
            graph[node][neighbor] = int(cost)

heuristic = {}
print("\nEnter heuristic values for each node:")
for node in nodes:
    h = int(input(f"Heuristic value for {node}: "))
    heuristic[node] = h

start = input("\nEnter start node: ")
goal = input("Enter goal node: ")

path = a_star(graph, start, goal, heuristic)
print("\nPath found:", path if path else "No path found.")
```

**OUTPUT :-**

```
Enter all nodes (space-separated): A B C D E
Enter neighbors of A with cost (format: B:2 C:5), or press Enter if none: B:1 C:4
Enter neighbors of B with cost (format: B:2 C:5), or press Enter if none: D:2
Enter neighbors of C with cost (format: B:2 C:5), or press Enter if none: D:5
Enter neighbors of D with cost (format: B:2 C:5), or press Enter if none: E:1
Enter neighbors of E with cost (format: B:2 C:5), or press Enter if none:

Enter heuristic values for each node:
Heuristic value for A: 7
Heuristic value for B: 6
Heuristic value for C: 5
Heuristic value for D: 2
Heuristic value for E: 0

Enter start node: A
Enter goal node: E

Path found: ['A', 'B', 'D', 'E']
```

✅ **Kruskal's MST Algorithm with User Input**

**INPUT :-**

```
class Graph:
    def __init__(self, vertices):
        self.V = vertices
        self.edges = []

    def add_edge(self, u, v, w):
        self.edges.append((w, u, v))

    def find(self, parent, i):
        if parent[i] != i:
            parent[i] = self.find(parent, parent[i])
        return parent[i]

    def union(self, parent, rank, x, y):
        if rank[x] < rank[y]:
            parent[x] = y
        elif rank[x] > rank[y]:
            parent[y] = x
        else:
            parent[y] = x
            rank[x] += 1

    def kruskal(self):
        self.edges.sort()
        parent = list(range(self.V))
        rank = [0] * self.V
        mst = []

        for weight, u, v in self.edges:
            root_u = self.find(parent, u)
            root_v = self.find(parent, v)
            if root_u != root_v:
                mst.append((u, v, weight))
                self.union(parent, rank, root_u, root_v)

        print("\nEdges in the Minimum Spanning Tree:")
        for u, v, w in mst:
```

```
            print(f"{u} -- {v} : {w}")


# ----- User Input -----
V = int(input("Enter number of vertices: "))
E = int(input("Enter number of edges: "))

g = Graph(V)

print("Enter edges in the format: u v weight")
for _ in range(E):
    u, v, w = map(int, input("Edge: ").split())
    g.add_edge(u, v, w)

g.kruskal()
```

**OUTPUT :-**

```
Enter number of vertices: 4
Enter number of edges: 5
Enter edges in the format: u v weight
Edge: 0 1 10
Edge: 0 2 6
Edge: 0 3 5
Edge: 1 3 15
Edge: 2 3 4

Edges in the Minimum Spanning Tree:
2 -- 3 : 4
0 -- 3 : 5
0 -- 1 : 10
```

✅ **N-Queens with User Input and Backtracking**

**INPUT :-**

```
def is_safe(board, row, col, n):
    for i in range(col):
        if board[row][i] == 1:
            return False

    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    for i, j in zip(range(row, n), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def solve_n_queens(board, col, n):
    if col >= n:
        return True

    for i in range(n):
        if is_safe(board, i, col, n):
            board[i][col] = 1
            if solve_n_queens(board, col + 1, n):
                return True
            board[i][col] = 0  # Backtrack

    return False

def print_board(board):
    for row in board:
        print(" ".join(['Q' if cell else '.' for cell in row]))

# ---- User Input ----
n = int(input("Enter the number of queens (n): "))
board = [[0] * n for _ in range(n)]

if solve_n_queens(board, 0, n):
    print("\nSolution:")
    print_board(board)
else:
    print("No solution found.")
```

**OUTPUT :-**

**Enter the number of queens (n): 4**

**Solution:**
**. Q . .**
**. . . Q**
**Q . . .**
**. . Q .**

**Solution for the n-queens problem using Backtracking and Branch and Bound:**

```
class NQueens:
    def __init__(self, n):
        self.n = n
        self.board = [-1] * n
        self.solutions = []

    def is_safe(self, row, col):
        for i in range(row):
            if self.board[i] == col or abs(self.board[i] - col) == abs(i - row):
                return False
        return True

    def solve_backtracking(self, row=0):
        if row == self.n:
            self.solutions.append(self.board.copy())
            return
        for col in range(self.n):
            if self.is_safe(row, col):
                self.board[row] = col
                self.solve_backtracking(row + 1)
                self.board[row] = -1

    def solve_branch_bound(self, row=0):
        if row == self.n:
            self.solutions.append(self.board.copy())
            return
        for col in range(self.n):
            if self.is_safe(row, col):
                self.board[row] = col
                self.solve_branch_bound(row + 1)
                self.board[row] = -1

    def print_solutions(self):
        if not self.solutions:
            print("No solution found!")
            return
        print(f"Found {len(self.solutions)} solutions:")
        for solution in self.solutions:
            for row in range(self.n):
                board_row = ['Q' if col == solution[row] else '.' for col in range(self.n)]
                print(" ".join(board_row))
            print()

if __name__ == "__main__":
    n = int(input("Enter the value of n for the n-queens problem: "))
```

```
queens_problem = NQueens(n)

queens_problem.solve_backtracking()
queens_problem.print_solutions()

queens_problem.solutions = []
queens_problem.solve_branch_bound()
queens_problem.print_solutions()
```

## Example Input/Output:

**Input:**

```
Enter the value of n for the n-queens problem: 4
```

**Output (Backtracking):**

```
Found 2 solutions:
Q . . .
. . Q .
. Q . .
. . . Q

. . Q .
Q . . .
. . . Q
. Q . .
```

**Output (Branch and Bound):**

```
Found 2 solutions:
Q . . .
. . Q .
. Q . .
. . . Q

. . Q .
Q . . .
. . . Q
. Q . .
```

✅ **Elementary Chatbot for Customer Interaction**

**INPUT :-**

```
def chatbot():
    print("Welcome to ShopBot! How can I assist you today?")
    while True:
        user = input("You: ").lower()
        if "price" in user:
            print("ChatBot: Prices are mentioned on the product page.")
        elif "return" in user:
            print("ChatBot: You can return the product within 7 days.")
        elif "bye" in user or "exit" in user:
            print("ChatBot: Thank you for visiting. Goodbye!")
            break
        else:
            print("ChatBot: I'm sorry, can you rephrase that?")

chatbot()
```

**OUTPUT :-**

```
Welcome to ShopBot! How can I assist you today?
You: what is the price of the phone
ChatBot: Prices are mentioned on the product page.
You: how can I return my order
ChatBot: You can return the product within 7 days.
You: okay bye
ChatBot: Thank you for visiting. Goodbye!
```

**EXPERT SYSTEM (PR-6)**

## 🏥 Hospitals and Medical Facilities Expert System

```
# Advanced Medicine Prescription Expert System with Symptom Suggestions
# Knowledge base: symptoms, medicine, and advice
knowledge_base = {
    "fever": {
        "medicine": "Paracetamol",
        "dosage": {
            "low": "250mg once a day",
            "medium": "500mg twice a day",
            "high": "650mg every 6 hours"
        },
        "advice": "Stay hydrated and rest well."
    },
    "headache": {
        "medicine": "Aspirin",
        "dosage": {
            "low": "75mg once a day",
            "medium": "150mg twice a day",
            "high": "300mg every 8 hours"
        },
        "advice": "Avoid screen time and get proper sleep."
    },
    "cold": {
        "medicine": "Cetirizine",
        "dosage": {
            "low": "5mg at night",
            "medium": "10mg once a day",
            "high": "10mg twice a day"
        },
        "advice": "Avoid cold drinks and rest."
    },
    "cough": {
        "medicine": "Cough Syrup",
        "dosage": {
            "low": "5ml twice a day",
```

```
            "medium": "10ml three times a day",
            "high": "10ml every 4 hours"
        },
        "advice": "Drink warm water and avoid fried food."
    }
}

def medicine_expert_system():
    print("💊 Welcome to Advanced Medicine Prescription Expert System")

    while True:
        # Show available symptom suggestions
        print("\n🩺 Available symptoms:", ", ".join(knowledge_base.keys()))

        user_input = input("Enter your symptoms (comma separated) or type 'exit'
to quit: ").lower()

        if user_input == "exit":
            print("👋 Thank you! Stay healthy!")
            break

        symptoms = [s.strip() for s in user_input.split(",")]

        for symptom in symptoms:
            if symptom in knowledge_base:
                severity = input(f"Enter severity for {symptom} (low / medium / high):
").lower()

                if severity in knowledge_base[symptom]["dosage"]:
                    med_info = knowledge_base[symptom]
                    print(f"\n🔎 Symptom: {symptom.capitalize()}")
                    print(f"💊 Medicine: {med_info['medicine']}")
                    print(f"💉 Dosage: {med_info['dosage'][severity]}")
                    print(f"📝 Advice: {med_info['advice']}")
                else:
                    print("❗ Invalid severity level. Please enter low, medium, or
high.")
            else:
                print(f"⚠️ Sorry, we don't have information for: {symptom}")

# Run the system
medicine_expert_system()
```

**OUTPUT:-**

**💊 Welcome to Advanced Medicine Prescription Expert System**

**🩺 Available symptoms: fever, headache, cold, cough**
**Enter your symptoms (comma separated) or type 'exit' to quit: fever, cough**
**Enter severity for fever (low / medium / high): high**

**🔎 Symptom: Fever**
**💊 Medicine: Paracetamol**
**💉 Dosage: 650mg every 6 hours**
**📝 Advice: Stay hydrated and rest well.**

**Enter severity for cough (low / medium / high): medium**

**🔎 Symptom: Cough**
**💊 Medicine: Cough Syrup**
**💉 Dosage: 10ml three times a day**
**📝 Advice: Drink warm water and avoid fried food.**

**🩺 Available symptoms: fever, headache, cold, cough**
**Enter your symptoms (comma separated) or type 'exit' to quit: headache**
**Enter severity for headache (low / medium / high): low**

**🔎 Symptom: Headache**
**💊 Medicine: Aspirin**
**💉 Dosage: 75mg once a day**
**📝 Advice: Avoid screen time and get proper sleep.**

**🩺 Available symptoms: fever, headache, cold, cough**
**Enter your symptoms (comma separated) or type 'exit' to quit: stomachache**
**⚠️ Sorry, we don't have information for: stomachache**

**🩺 Available symptoms: fever, headache, cold, cough**
**Enter your symptoms (comma separated) or type 'exit' to quit: exit**
**👋 Thank you! Stay healthy!**