Practical No. 1

```
#BFS

graph = {
'A':['B','C'],
'B':['D','E'],
'C':['F'],
'D':[],
'E':['F'],
'F':[]
}
visited=[]
queue=[]

def bfs(visited,graph,node):
        visited.append(node)
        queue.append(node)
        while queue:
                s = queue.pop(0)
                print(s,end=" ")

                for neighbour in graph[s]:
                        if neighbour not in visited:
                                visited.append(neighbour)
                                queue.append(neighbour)

print("following path is Breadth-First Algorithm")
bfs(visited,graph,'A')



#DFS
graph = {
'A':['B','C'],
'B':['D','E'],
'C':['F'],
'D':[],
'E':['F'],
'F':[]
}
visited = set()

def dfs(visited,graph,node):
        if node  not in  visited:
                print(node,end=" \n")
                visited.add(node)

                for neighbour in graph[node]:
                        dfs (visited,graph,neighbour)
```

```
print("\nfollowing path is Depth-First Algorithm")

dfs(visited,graph,'A')
```

OUTPUT:

```
lab314@lab314-ThinkCentre-M70s:~$ python3 ass1.py
following path is Breadth-First Algorithm
A B C D E F


following path is Depth-First Algorithm
A
B
D
E
F
C
```

Practical No. 2

Implement A star Algorithm for any game search problem.

```
import heapq

# Directions (up, down, left, right)
directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class Node:
    def __init__(self, position, g_cost, h_cost, parent=None):
        self.position = position
        self.g_cost = g_cost  # Cost from start to node
        self.h_cost = h_cost  # Estimated cost from node to goal
        self.f_cost = g_cost + h_cost  # Total cost (f = g + h)
        self.parent = parent  # To track the path

    def __lt__(self, other):
        # Compare nodes based on their f_cost (for priority queue)
        return self.f_cost < other.f_cost

def heuristic(a, b):
    # Manhattan distance heuristic (use a different one if needed)
    return abs(a[0] - b[0]) + abs(a[1] - b[1])

def a_star(grid, start, goal):
    open_list = []
    closed_list = set()

    # Initialize start node and add it to the open list
    start_node = Node(start, 0, heuristic(start, goal))
    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)  # Get node with lowest f_cost
        current_position = current_node.position

        if current_position == goal:
            # Goal reached, reconstruct path
            path = []
            while current_node:
                path.append(current_node.position)
                current_node = current_node.parent
            return path[::-1]  # Return path from start to goal

        closed_list.add(current_position)  # Add to closed list

        # Explore neighbors
        for direction in directions:
            neighbor = (current_position[0] + direction[0], current_position[1] + direction[1])
```

```
            # Check if neighbor is within grid bounds
            if 0 <= neighbor[0] < len(grid) and 0 <= neighbor[1] < len(grid[0]):
                if grid[neighbor[0]][neighbor[1]] == 1:  # Obstacle (1 means obstacle)
                    continue  # Skip obstacles

                if neighbor in closed_list:
                    continue  # Skip already evaluated

                g_cost = current_node.g_cost + 1  # Movement cost (can be adjusted)
                h_cost = heuristic(neighbor, goal)
                neighbor_node = Node(neighbor, g_cost, h_cost, current_node)

                # Check if neighbor is in open list and has a higher f_cost
                if not any(node.position == neighbor and node.f_cost <= neighbor_node.f_cost for node
in open_list):
                    heapq.heappush(open_list, neighbor_node)

    return None  # Return None if no path found

def print_grid(grid):
    for row in grid:
        print(" ".join(str(cell) for cell in row))

# Example Usage:
# 0 = free space, 1 = obstacle
grid = [
    [0, 0, 0, 0, 0],
    [0, 1, 1, 0, 0],
    [0, 1, 0, 1, 0],
    [0, 0, 0, 1, 0],
    [0, 0, 0, 0, 0]
]

start = (0, 0)  # Starting point
goal = (4, 4)   # Goal point

path = a_star(grid, start, goal)

if path:
    print("Path found:", path)
else:
    print("No path found")
```

OUTPUT:

```
 python3 ass2.py
Path found: [(0, 0), (1, 0), (2, 0), (3, 0), (4, 0), (4, 1), (4, 2), (4, 3), (4, 4)]
```

Practical No.3

1.Selection Sort Algorithm:

CODE:

```
public class SelectionSort {

    // Method to perform selection sort
    public static void selectionSort(int[] array) {
        int size = array.length;

        for (int step = 0; step < size - 1; step++) {
            int minIndex = step;

            // Find the index of the smallest element in the remaining array
            for (int i = step + 1; i < size; i++) {
                if (array[i] < array[minIndex]) {
                    minIndex = i;
                }
            }

            // Swap the found minimum element with the first element
            int temp = array[step];
            array[step] = array[minIndex];
            array[minIndex] = temp;
        }
    }

    // Method to print the array
    public static void printArray(int[] array) {
        for (int value : array) {
            System.out.print(value + " ");
        }
        System.out.println();
    }

    // Main method
    public static void main(String[] args) {
        int[] data = {20, 12, 10, 15, 2};

        selectionSort(data);

        System.out.println("Sorted array in Ascending Order:");
        printArray(data);
    }
}
```

OUTPUT:

```
Sorted array in Ascending Order:
2 10 12 15 20
```

Practical No. 4

CODE:

```
def is_safe(board, row, col):
    n = len(board)

    # Check the row
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check the upper diagonal
    r, c = row, col
    while r >= 0 and c >= 0:
        if board[r][c] == 1:
            return False
        r -= 1
        c -= 1

    # Check the lower diagonal
    r, c = row, col
    while r < n and c >= 0:
        if board[r][c] == 1:
            return False
        r += 1
        c -= 1

    # If no conflicts found, it's safe to place a queen
    return True

def backtrack(board, col, solutions):
    n = len(board)

    # If all queens are placed, a valid solution is found
    if col == n:
        solutions.append([row[:] for row in board])
        return

    # Explore all possible positions in the current column
    for row in range(n):
        if is_safe(board, row, col):
            board[row][col] = 1  # Place a queen

            # Recursively move to the next column
            backtrack(board, col + 1, solutions)

            board[row][col] = 0  # Remove the queen (backtrack)

def solve_nqueens(n):
    board = [[0] * n for _ in range(n)]
```

```
    solutions = []
    backtrack(board, 0, solutions)
    return solutions

# Example usage
n = 4
solutions = solve_nqueens(n)

print(f"Total solutions for {n}-queens problem: {len(solutions)}")
for i, solution in enumerate(solutions):
    print(f"Solution {i+1}:")
    for row in solution:
        print(row)
    print()
```

OUTPUT:

```
Total solutions for 4-queens problem: 2
Solution 1:
[0, 0, 1, 0]
[1, 0, 0, 0]
[0, 0, 0, 1]
[0, 1, 0, 0]

Solution 2:
[0, 1, 0, 0]
[0, 0, 0, 1]
[1, 0, 0, 0]
[0, 0, 1, 0]


=== Code Execution Successful ===
```

```
class CustomerChatbot:
    def __init__(self):
        # Dictionary with customer queries and responses
        self.responses = {
            "hello": "Hi! Welcome to our customer support. How can I assist you today?",
            "hi": "Hello! How can I help you today?",
            "how are you": "I'm doing great as a bot, thanks for asking! How can I assist you?",
            "what is your name": "I'm Grok, your friendly customer support assistant!",
            "goodbye": "Thank you for chatting with me! Have a great day!",
            "help": "I can assist you with: services, hours, order status, payments, returns, shipping, or contact info. What would you like to know?",
            "services": "We provide product information, order tracking, technical support, and customer assistance.",
            "hours": "We're available Monday-Friday, 9 AM to 5 PM EST.",
            "order status": "Please provide your order ID (e.g., ORD123) to check your order status.",
            "payment methods": "We accept Visa, MasterCard, PayPal, and Apple Pay.",
            "return policy": "Items can be returned within 30 days with original receipt.",
            "shipping": "Free shipping on orders over $50. Standard delivery takes 3-5 business days.",
            "contact": "Email: support@company.com | Phone: 1-800-555-1234 | Live Chat: Available now!"
        }

        # Sample order database (for demonstration)
        self.orders = {
            "ord123": "Shipped - Estimated delivery: April 12, 2025",
            "ord456": "Processing - Expected to ship: April 10, 2025"
        }

    def get_response(self, user_input):
        """Process user input and return appropriate response"""
        user_input = user_input.lower().strip()

        # Check if input matches any predefined responses
        if user_input in self.responses:
            return self.responses[user_input]

        # Check if user is providing an order ID
        if "ord" in user_input and len(user_input) <= 6:
            return self.check_order_status(user_input)

        # Default response for unknown inputs
        return "I'm sorry, I didn't understand that. Type 'help' for available options!"

    def check_order_status(self, order_id):
        """Check order status from the sample database"""
        return self.orders.get(order_id, "Order not found. Please check your order ID and try again.")

    def run(self):
        """Main chatbot loop"""
        print("Welcome to Customer Support Chatbot! (Type 'goodbye' to exit)")
        print("How may I assist you today?")

        while True:
            try:
                user_input = input("You: ")
                if not user_input:  # Handle empty input
                    print("Bot: Please type something so I can assist you!")
                    continue

                response = self.get_response(user_input)
                print(f"Bot: {response}")

                if user_input.lower().strip() == "goodbye":
                    break

            except KeyboardInterrupt:
                print("\nBot: Goodbye! Thanks for chatting!")
                break
            except Exception as e:
                print(f"Bot: Oops! Something went wrong: {str(e)}. Please try again.")

def main():
    chatbot = CustomerChatbot()
    chatbot.run()

if __name__ == "__main__":
    main()
```

```
Welcome to Customer Support Chatbot! (Type 'goodbye' to exit)
How may I assist you today?
You: hello
Bot: Hi! Welcome to our customer support. How can I assist you today?
You: order status
Bot: Please provide your order ID (e.g., ORD123) to check your order status.
You: ord123
Bot: Shipped - Estimated delivery: April 12, 2025
You: goodbye
Bot: Thank you for chatting with me! Have a great day!
```

```
# Function to evaluate employee performance
def evaluate_performance(work_quality, teamwork, punctuality, leadership):
    # Rule 1: Excellent performance
    if work_quality == 5 and teamwork == 5 and punctuality == 5 and leadership == 5:
        print("Evaluation: Excellent - Highly recommend for promotion.")

    # Rule 2: Good performance
    elif work_quality >= 4 and teamwork >= 4 and punctuality >= 4 and leadership >= 4:
        print("Evaluation: Good - Satisfactory performance, may need improvement in leadership.")

    # Rule 3: Average performance
    elif work_quality >= 3 and teamwork >= 3 and punctuality >= 3 and leadership >= 3:
        print("Evaluation: Average - Needs improvement in multiple areas.")

    # Rule 4: Below average performance
    elif work_quality >= 2 and teamwork >= 2 and punctuality >= 2 and leadership >= 2:
        print("Evaluation: Below Average - Performance is lacking in all areas.")

    # Rule 5: Poor performance
    elif work_quality == 1 and teamwork == 1 and punctuality == 1 and leadership == 1:
        print("Evaluation: Poor - Immediate improvement is necessary.")

    # Default case: Custom evaluation based on specific areas
    else:
        print("Evaluation: Needs further review.")

def main():
    try:
        # Get employee ratings from the user
        print("Enter the ratings for the following criteria (1 to 5 scale):")
        work_quality = int(input("Work Quality: "))
        teamwork = int(input("Teamwork: "))
        punctuality = int(input("Punctuality: "))
        leadership = int(input("Leadership: "))

        # Validate ratings
        if (work_quality < 1 or work_quality > 5 or
            teamwork < 1 or teamwork > 5 or
            punctuality < 1 or punctuality > 5 or
            leadership < 1 or leadership > 5):
            print("Invalid input! Ratings should be between 1 and 5.")
        else:
            # Evaluate the employee performance based on the ratings
            evaluate_performance(work_quality, teamwork, punctuality, leadership)

    except ValueError:
        print("Invalid input! Please enter numeric values between 1 and 5.")

if __name__ == "__main__":
    main()
```

```
Enter the ratings for the following criteria (1 to 5 scale):
Work Quality: 2
Teamwork: 2
Punctuality: 3
Leadership: 5
Evaluation: Below Average - Performance is lacking in all areas.
```

8.

```
$ curl -o
https://dl.google.com/dl/cloudsdk/channels/rapid/downloads/google-cloud-sdk-380.0.0-linux
x86_64.tar.gz

$ ./google-cloud-sdk/install.sh
$ gcloud init
```

9.

```
public class Email {
public void sendMail(String [] addresses, String [] subjects, String [] messages) {
Messaging.SingleEmailMessage [] emails = new Messaging.SingleEmailMessage[] {};
Integer totalMails= addresses.size();
for(Integer i=0; i<totalMails; i++) {
Messaging.SingleEmailMessage email = new Messaging.SingleEmailMessage();
email.setSubject(subjects[i]);
email.setToAddresses(new List<String> { addresses[i] });
email.setPlainTextBody(messages[i]);
emails.add(email);
Messaging.sendEmail(emails); }
}
```