OBJECTIVES:

1. Vacuum-Cleaner Agent in Two Rooms (A And B)
2. To Implement Breadth First Search.
3. To Implement Depth First Search
4. To implement A* Algorithm
5. To implement Tic-Tac-Toe game

CODE AND OUTPUT:

```python
def vacuum_cleaner(rooms):
    print("Initial room states:", rooms)

    for room in rooms.keys():
        if rooms[room] == 1:
            print(f"Cleaning room {room}...")
            rooms[room] = 0
        else:
            print(f"Room {room} is already clean.")

    print("Final room states:", rooms)

# Initialize with both rooms dirty
rooms = {"A": 1, "B": 1}
vacuum_cleaner(rooms)
```

```
Initial room states: {'A': 1, 'B': 1}
Cleaning room A...
Cleaning room B...
Final room states: {'A': 0, 'B': 0}
```

MODIFICATION IN CODE:

```python
def vacuum_cleaner(rooms):
    print("Initial room states:", rooms)

    for room in rooms.keys():
        if rooms[room] == 1:
            print(f"Cleaning room {room}...")
            rooms[room] = 0
        else:
            print(f"Room {room} is already clean.")

    print("Final room states:", rooms)

# Initialize with four rooms, all dirty
rooms = {"A": 1, "B": 1, "C": 1, "D": 1}
vacuum_cleaner(rooms)
```

```
Initial room states: {'A': 1, 'B': 1, 'C': 1, 'D': 1}
Cleaning room A...
Cleaning room B...
Cleaning room C...
Cleaning room D...
Final room states: {'A': 0, 'B': 0, 'C': 0, 'D': 0}
```

CODE AND OUTPUT:

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F"],
    "D": [],
    "E": ["F"],
    "F": []
}
```

```
A B C D E F
[Done] exited with code=0 in 0.135 seconds
```

MODIFICATION IN CODE:

```python
from collections import deque

def bfs(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print(node, end=" ")
            visited.add(node)
            queue.extend(neighbor for neighbor in graph[node] if neighbor not in visited)

# Food hierarchy graph
graph = {
    "Main Course": ["Pasta", "Pizza", "Burger", "Fried Chicken", "Sandwich"],
    "Pasta": [],
    "Pizza": [],
    "Burger": [],
    "Fried Chicken": [],
    "Sandwich": []
}
```

CODE AND OUTPUT:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    "A": ["B", "C"],
    "B": ["D", "E"],
    "C": ["F"],
    "D": [],
    "E": ["F"],
    "F": []
}

dfs(graph, "A")
```

```
A B D E F C
[Done] exited with code=0 in 0.187 seconds
```

MODIFICATION:

```python
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

graph = {
    "Karachi": ["Hyderabad", "Thatta"],
    "Hyderabad": ["Mirpurkhas", "Jamshoro"],
    "Thatta": ["Badin"],
    "Mirpurkhas": [],
    "Jamshoro": ["Dadu"],
    "Badin": ["Dadu"],
    "Dadu": []
}
```

```
Karachi Hyderabad Mirpurkhas Jamshoro Dadu Thatta Badin
[Done] exited with code=0 in 0.128 seconds
```

CODE AND OUTPUT:

```
def a_star_algorithm(graph, start, goal, heuristic):
    open_list = PriorityQueue()
    open_list.put((0, start))
    g_score = {node: float('inf') for node in graph}
    g_score[start] = 0
    came_from = {}

    while not open_list.empty():
        _, current = open_list.get()
        if current == goal:
            path = []
            while current in came_from:
                path.append(current)
                current = came_from[current]
            return [start] + path[::-1]

        for neighbor, cost in graph[current].items():
            tentative_g_score = g_score[current] + cost
            if tentative_g_score < g_score[neighbor]:
                g_score[neighbor] = tentative_g_score
                open_list.put((g_score[neighbor] + heuristic[neighbor], neighbor))
                came_from[neighbor] = current

    return None

graph = {'A': {'B': 1, 'C': 3}, 'B': {'D': 1, 'E': 3}, 'C': {'F': 2}, 'D': {'G': 2}, 'E': {'G': 1}, 'F':
heuristic = {'A': 6, 'B': 4, 'C': 4, 'D': 2, 'E': 2, 'F': 2, 'G': 0}

print("Shortest Path:", a_star_algorithm(graph, 'A', 'G', heuristic))
```

```
Shortest Path: ['A', 'B', 'D', 'G']

[Done] exited with code=0 in 0.212 seconds
```

CODE:

```python
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 5)

def check_winner(board):
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != " ":
            return row[0]
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != " ":
            return board[0][col]
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return board[0][2]
    return None

def tic_tac_toe():
    board = [[" " for _ in range(3)] for _ in range(3)]
    turns = ["X", "O"]
    for i in range(9):
```

```python
for i in range(9):
    print_board(board)
    print(f"Player {turns[i % 2]}'s turn")
    try:
        row, col = map(int, input("Enter row and column (0, 1, or 2): ").split())
        if board[row][col] == " ":
            board[row][col] = turns[i % 2]
            winner = check_winner(board)
            if winner:
                print_board(board)
                print(f"Player {winner} wins!")
                return
        else:
            print("Invalid move! Try again.")
            continue
    except (ValueError, IndexError):
        print("Invalid input! Enter numbers 0, 1, or 2.")
        continue
print_board(board)
print("It's a tie!")
```

OUTPUT:

```
PS D:\python\__pycache__> & "C:/Users/Yousuf Traders/AppData/Local/Programs/Python/Python313/python.exe" "d:/python/mod1/import math.py"
  |  |  |
-----
  |  |  |
-----
  |  |  |
-----
  |  |  |
-----
Player X's turn
Enter row and column (0-3): 3 3
  |  |  |
-----
  |  |  |
-----
  |  |  |
-----
  |  |  | X
-----
Player O's turn
Enter row and column (0-3): 32 2
Invalid move! Row and column must be between 0 and 3. Try again.
Enter row and column (0-3): 2 2
  |  |  |
-----
  |  |  |
-----
  |  | O |
-----
  |  |  | X
-----
```

```
-----
Player X's turn
Enter row and column (0-3): 3 2
  |   |   |
-----
  |   |   |
-----
  |   | O |
-----
  |   | X | X
-----
Player O's turn
Enter row and column (0-3): 3 1
  |   |   |
-----
  |   |   |
-----
  |   | O |
-----
  | O | X | X
-----
Player X's turn
Enter row and column (0-3): 2 3
  |   |   |
-----
  |   |   |
-----
  |   | O | X
-----
  | O | X | X
-----
```