Name: Maryum Shakeel

Sap ID: 48406

Subject: AI Lab

Batch: BSCS 6th Semester

Submitted by: Miss Ayesha Akram

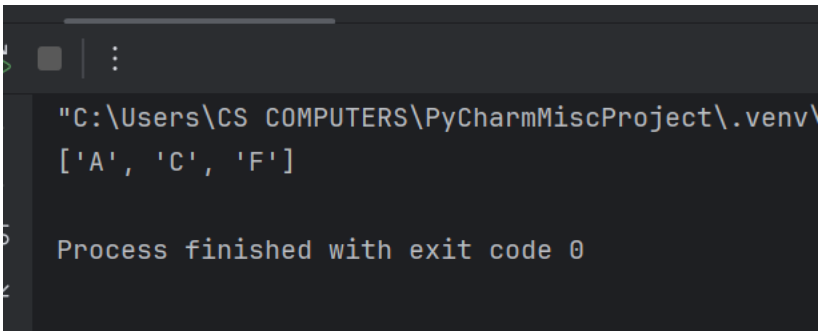
Task 1

Solution:

```
from collections import deque
import heapq
def bfs_shortest_path(graph, start, goal):
    queue = deque([(start, [start])])
    visited = set()
    while queue:
        node, path = queue.popleft()
        if node == goal:
            return path
        if node not in visited:
            visited.add(node)
            for neighbor in graph.get(node, []):
                queue.append((neighbor, path + [neighbor]))
    return None
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
```

```
    'E': ['F'],
    'F': []
}
print(bfs_shortest_path(graph, 'A', 'F'))
```

Output



Task 2

Solution:

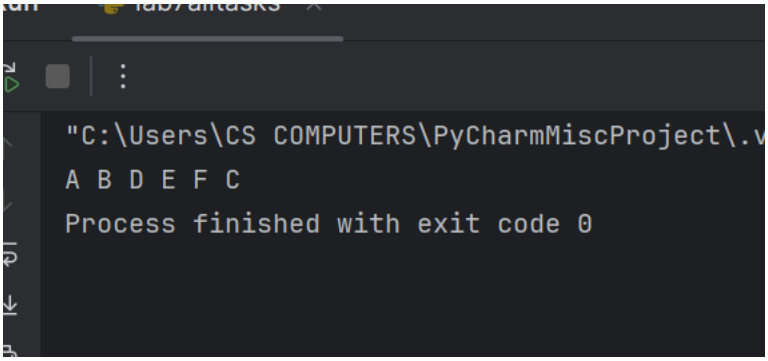
```
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}


# Depth First Search function
```

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
    for neighbor in graph.get(start, []):
        if neighbor not in visited:
            dfs(graph, neighbor, visited)


# Call DFS
dfs(graph, 'A')
```

Output



Task 3

Solution:

```
class Puzzle:
    def __init__(self, board, goal):
```

```
    self.board = board
    self.goal = goal

def bfs(self):
    queue = deque([(self.board, [])])
    visited = set()
    while queue:
        state, path = queue.popleft()
        if state == self.goal:
            return path
        visited.add(tuple(map(tuple, state)))
        for move, new_state in self.possible_moves(state):
            if tuple(map(tuple, new_state)) not in visited:
                queue.append((new_state, path + [move]))
    return None

def possible_moves(self, state):
    moves = []
    x, y = next((i, j) for i in range(3) for j in range(3) if state[i][j] == 0)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
```

```
            if 0 <= nx < 3 and 0 <= ny < 3:
                new_state = [row[:] for row in state]
                new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
                moves.append(((x, y), new_state))
        return moves


initial_state = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]
puzzle = Puzzle(initial_state, goal_state)
print(puzzle.bfs())
```

Output:

Task 4

Solution:

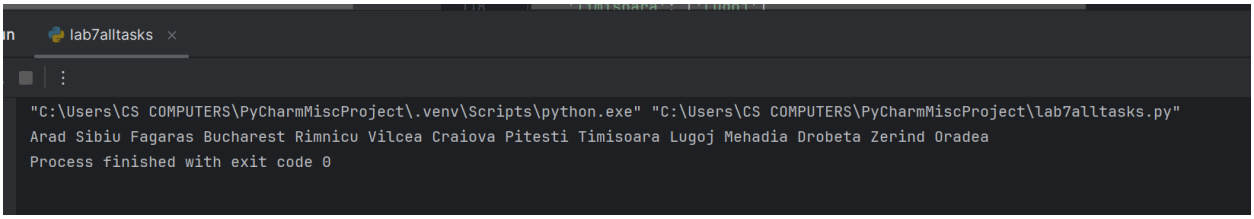
```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)
    print(start, end=' ')
```

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

# Make sure the Romania map is correctly defined
romania_map = {
    'Arad': ['Sibiu', 'Timisoara', 'Zerind'],
    'Sibiu': ['Fagaras', 'Rimnicu Vilcea'],
    'Fagaras': ['Bucharest'],
    'Rimnicu Vilcea': ['Craiova', 'Pitesti'],
    'Pitesti': ['Bucharest'],
    'Timisoara': ['Lugoj'],
    'Lugoj': ['Mehadia'],
    'Mehadia': ['Drobeta'],
    'Drobeta': ['Craiova'],
    'Craiova': ['Pitesti'],
    'Zerind': ['Oradea'],
    'Oradea': ['Sibiu']
}

# Call DFS properly
dfs(romania_map, 'Arad')  # Should print the traversal without errors
```

Output



Task 5

Solution:

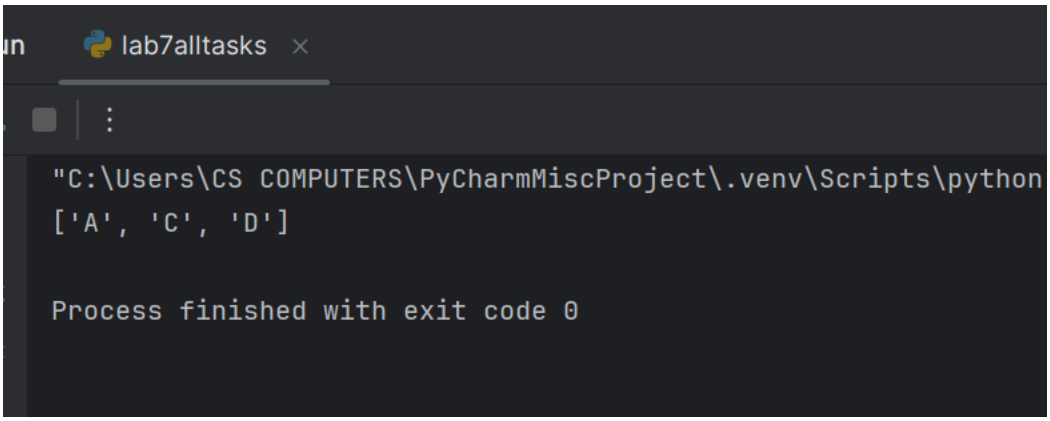
```
def astar(graph, start, goal, h):
    open_set = [(0, start)]
    g = {start: 0}
    came_from = {}

    while open_set:
        _, current = heapq.heappop(open_set)
        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[current] + cost
            if neighbor not in g or tentative_g < g[neighbor]:
                g[neighbor] = tentative_g
                heapq.heappush(open_set, (tentative_g + h[neighbor], neighbor))
                came_from[neighbor] = current
    return None


graph = {
    'A': {'B': 1, 'C': 4},
    'B': {'C': 2, 'D': 5},
    'C': {'D': 1},
    'D': {}
}
h = {'A': 7, 'B': 6, 'C': 2, 'D': 0}
print(astar(graph, 'A', 'D', h))
```

Output



Task 6

Solution:

```
def minimax(board, depth, is_max):
    scores = {'X': 1, 'O': -1, 'Tie': 0}
    winner = check_winner(board)
    if winner:
        return scores[winner]
    if is_max:
        best = -float('inf')
        for move in available_moves(board):
            board[move] = 'X'
            best = max(best, minimax(board, depth + 1, False))
            board[move] = ' '
        return best
    else:
        best = float('inf')
        for move in available_moves(board):
            board[move] = 'O'
            best = min(best, minimax(board, depth + 1, True))
            board[move] = ' '
        return best


def available_moves(board):
    return [i for i, x in enumerate(board) if x == ' ']


def check_winner(board):
    win_patterns = [(0, 1, 2), (3, 4, 5), (6, 7, 8), (0, 3, 6), (1, 4, 7),
(2, 5, 8), (0, 4, 8), (2, 4, 6)]
    for (x, y, z) in win_patterns:
        if board[x] == board[y] == board[z] != ' ':
            return board[x]
    return 'Tie' if ' ' not in board else None


def best_move(board):
```

```
    return max(available_moves(board), key=lambda move: minimax(board[:], 0,
False))


board = ['X', 'O', 'X', ' ', 'O', ' ', ' ', ' ', ' ']  # Example board
print(best_move(board))
```

Output