VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shashank Ravindra karanam(1BM23CS312)

in partial fulfillment for the award of the degree of
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Aug 2025 to Dec 2025

B.M.S. College of Engineering,
Bull Temple Road, Bangalore 560019
(Affiliated To Visvesvaraya Technological University, Belgaum)
Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Shashank Ravindra karanam (1BM23CS312),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Dr.Seema Patil Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

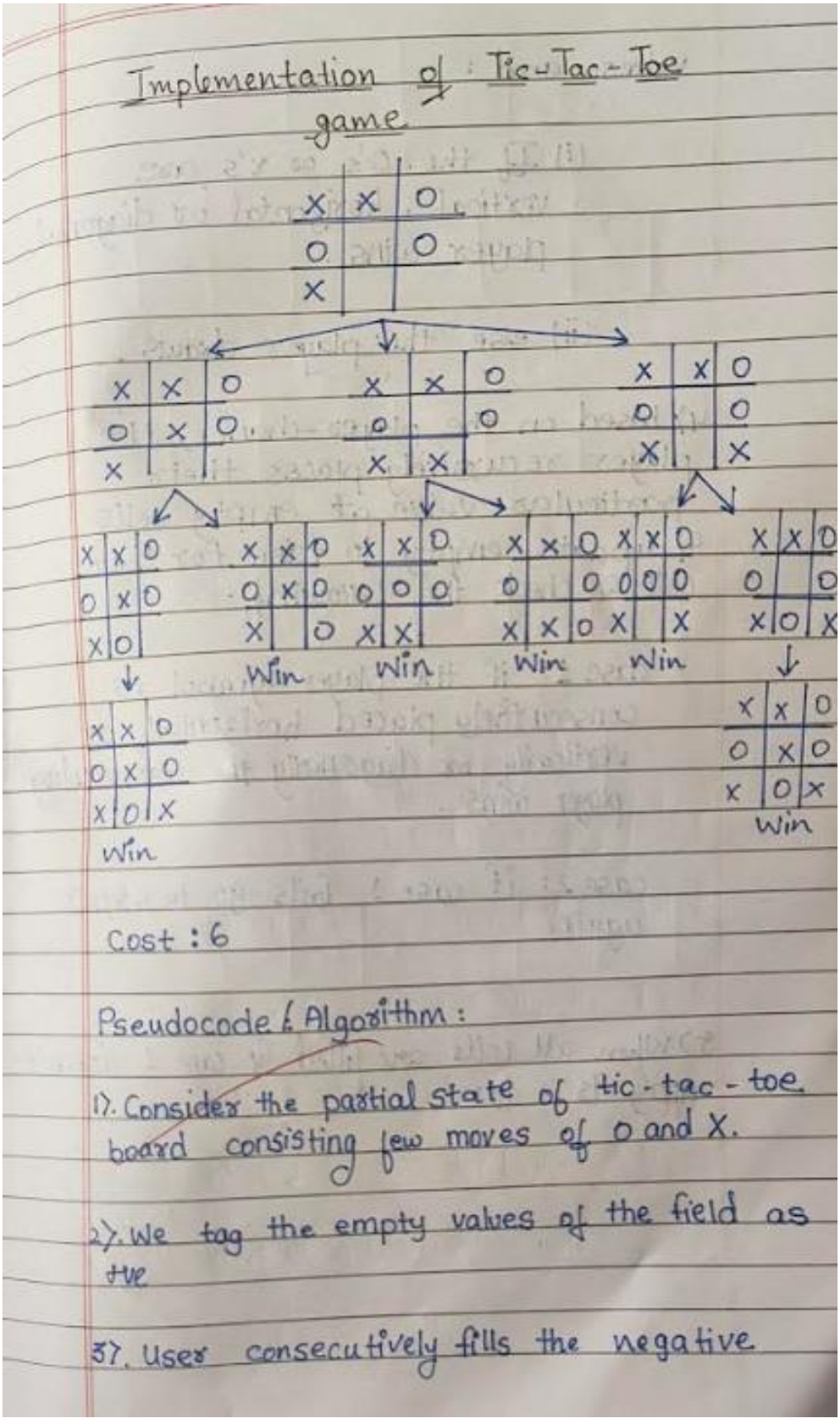
Index

Github Link: https://github.com/Shashank-Karanam/AI-LaB

Program 1

Implement Tic –Tac –Toe Game

Algorithm:



valves and :

(i) If the O's or X's are vertical, horizontal or diagonal, player wins.

(ii) else the player draws.

4) based on the player-chance, the player recursively places their particular value at empty cells & update empty to false for that cell & check the following.

case 1: if the player symbol is consecutively placed horizontally or vertically or diagonally the corresponding player wins.

case 2: if case 1 fails go to step 3 again

5) When all cells are filled & case 1 doesn't apply its a draw.

```
import math

# === YOUR DETAILS ===
NAME = "Shashank Ravindra Karanam"
USN = "1BM23CS3121"

# Function to print the board
def print_board(board):
    for row in board:
        print("|".join(row))
    print()

# Function to check if moves are left
def is_moves_left(board):
    for row in board:
        if " " in row:
            return True
    return False

# Function to evaluate board
def evaluate(board):
    # Check rows
    for row in board:
        if row.count(row[0]) == 3 and row[0] != " ":
            return 10 if row[0] == "O" else -10
    # Check cols
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] != " ":
            return 10 if board[0][col] == "O" else -10
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] != " ":
        return 10 if board[0][0] == "O" else -10
    if board[0][2] == board[1][1] == board[2][0] != " ":
        return 10 if board[0][2] == "O" else -10
    return 0

# Minimax function
def minimax(board, depth, is_max):
    score = evaluate(board)
    if score == 10: return score - depth
    if score == -10: return score + depth
    if not is_moves_left(board): return 0

    if is_max:
        best = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
                    best = max(best, minimax(board, depth + 1, False))
                    board[i][j] = " "
        return best
    else:
        best = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "X"
                    best = min(best, minimax(board, depth + 1, True))
                    board[i][j] = " "
        return best

# Find best move for computer
def find_best_move(board):
    best_val = -math.inf
    best_move = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                move_val = minimax(board, 0, False)
                board[i][j] = " "
                if move_val > best_val:
                    best_move = (i, j)
                    best_val = move_val
    return best_move

# Play game
def play_game():
    board = [[" "]*3 for _ in range(3)]
    print("Tic-Tac-Toe Game! You are X, Computer is O")
```

```
    print("Tic-Tac-Toe Game! You are X, Computer is O")
    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:  # Human move
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter col (0-2): "))
            if board[row][col] != " ":
                print("Invalid move, try again.")
                continue
            board[row][col] = "X"
        else:  # Computer move
            move = find_best_move(board)
            board[move[0]][move[1]] = "O"
            print(f"Computer places O at {move}")

        print_board(board)
        score = evaluate(board)
        if score == 10:
            print("Computer Wins!")
            print(f"\nSubmitted by: {NAME}, USN: {USN}")
            return
        elif score == -10:
            print("You Win!")
            print(f"\nSubmitted by: {NAME}, USN: {USN}")
            return

    print("It's a Draw!")
    print(f"\nSubmitted by: {NAME}, USN: {USN}")

play_game()
```

```
1
```

```
Tic-Tac-Toe Game! You are X, Computer is O
 | |
 | |
 | |

Enter row (0-2): 1
Enter col (0-2): 1
 | |
 |X|
 | |

Computer places O at (0, 0)
O| |
 |X|
 | |

Enter row (0-2): 0
Enter col (0-2): 2
O| |X
 |X|
 | |

Computer places O at (2, 0)
O| |X
 |X|
O| |

Enter row (0-2): 0
Enter col (0-2): 1
O|X|X
 |X|
O| |

Computer places O at (1, 0)
O|X|X
O|X|
O| |

Computer Wins!

Submitted by: Shashank Ravindra Karanam, USN: 1BM23CS312
```

```
1
```

Program 1

Implement vacuum cleaner agent

Algorithm:

Lab 2: Vacuum cleaner agent :

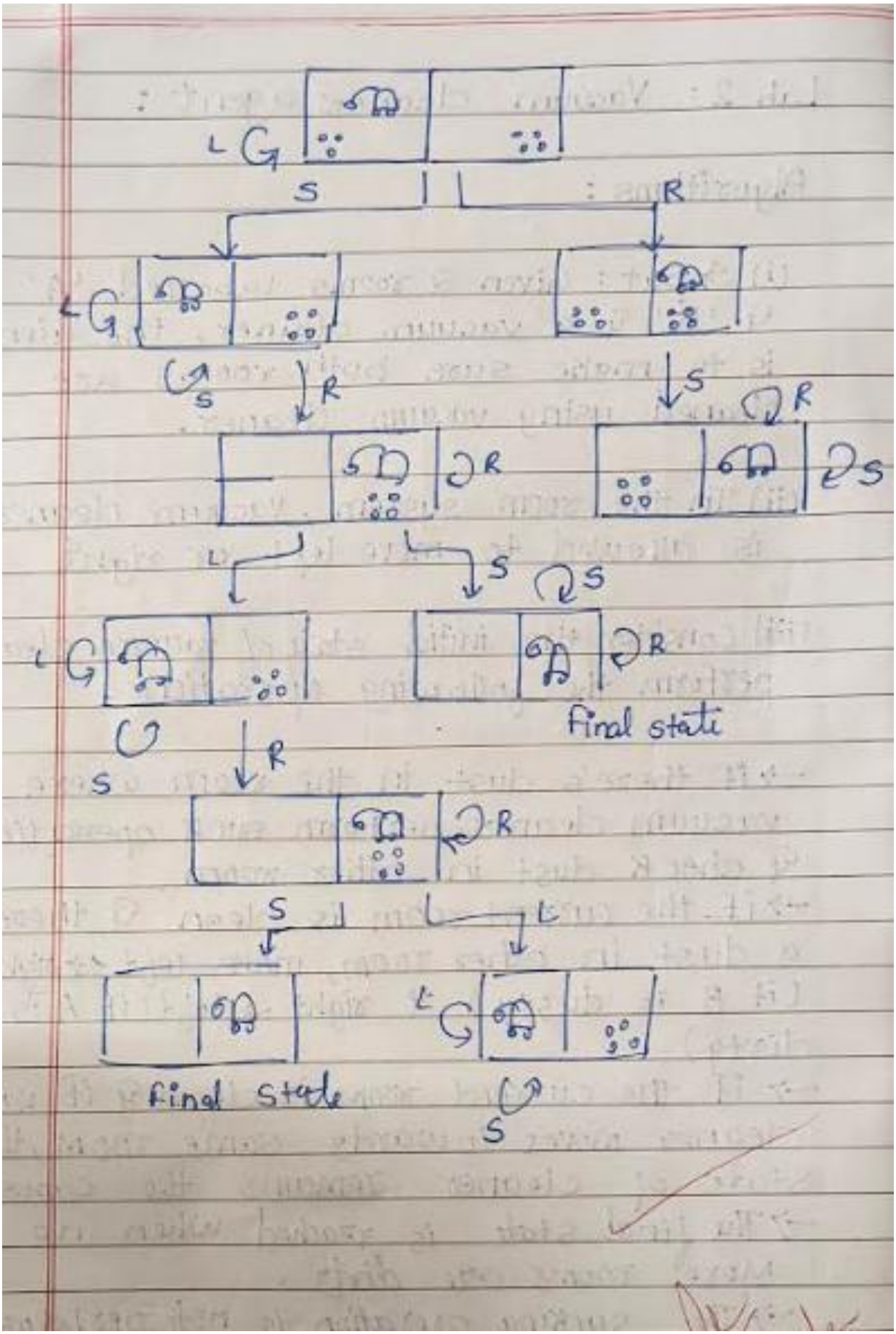
Algorithms :

(i) Input : Given 2 rooms labelled 'A' & 'B' & a vacuum cleaner, the aim is to make sure both rooms are cleaned using vacuum cleaner.

(ii) In the room system, vacuum cleaner is allowed to move left or right.

(iii) Consider the initial state of vacuum clear & perform the following operation.

→ if there's dust in the room where vacuum cleaner, perform suck operation & check dust in other room.
→ if the current room is clean & there's a dust in other room, move left → right (if B is dusty) & right → left (if A is dirty).
→ if the current room is clean & if vacuum cleaner moves towards same room, the state of cleaner remains the same.
→ The final state is reached when no more rooms are dirty.
→ The sucking operation is not performed when the dust is present in opposite room & vacuum cleaner is not in the room yet.

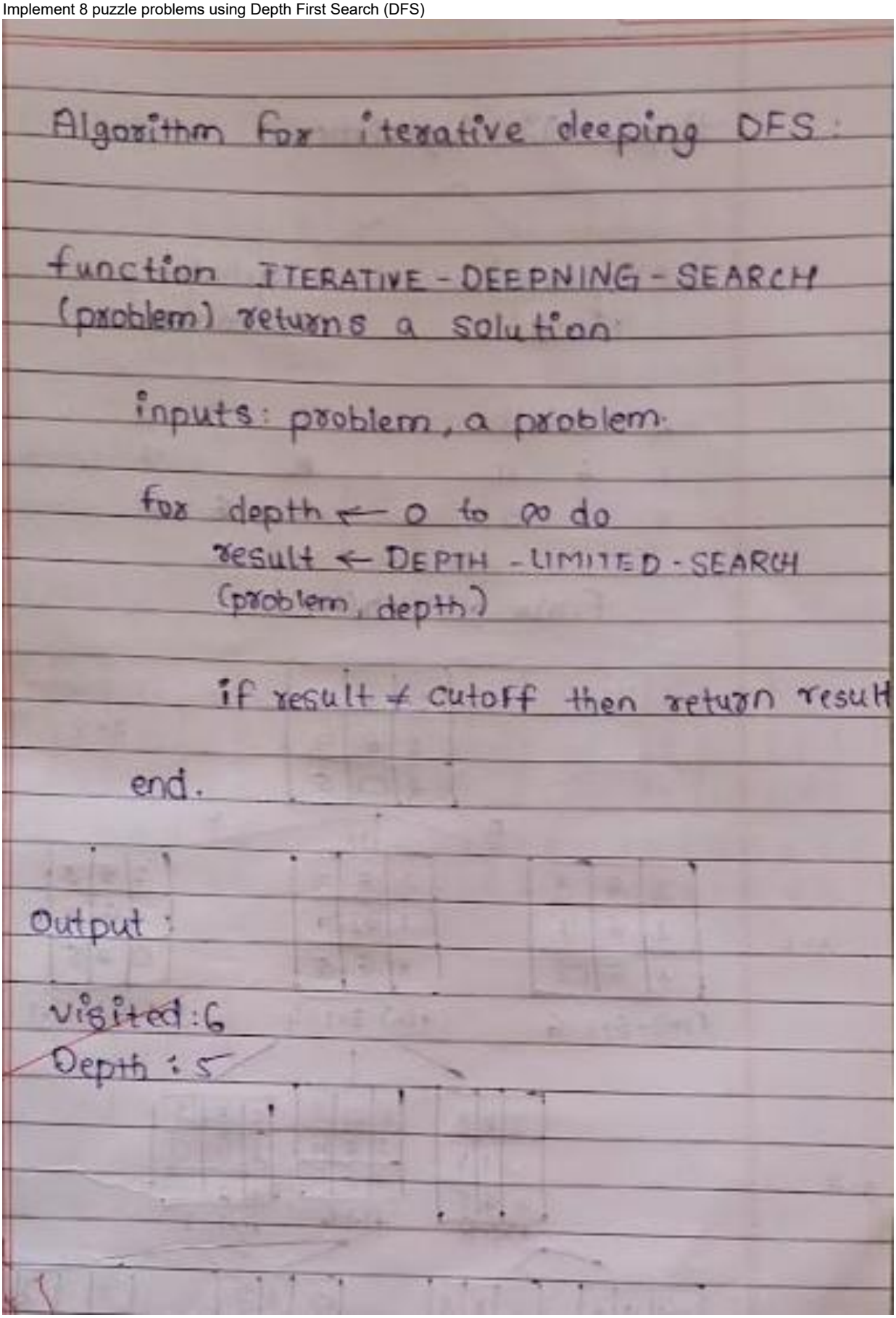

G
S
R
G
S
R
S
R
R
S
S
R
G
S
R
Final state
R
S
L
Final state
G
S

```
import random


NAME = "Shashank Ravindra Karanam"
USN = "1BM23CS312"


def reflex_vacuum_agent(location, status):
    if status == "Dirty":
        return "Suck"
    elif location == "A":
        return "Right"
    elif location == "B":
        return "Left"


def vacuum_world():
    locations = {"A": random.choice(["Clean", "Dirty"]),
                 "B": random.choice(["Clean", "Dirty"])}
    location = random.choice(["A", "B"])

    print("Initial State:", locations, "| Vacuum at:", location)

    steps = 5
    for _ in range(steps):
        status = locations[location]
        action = reflex_vacuum_agent(location, status)
        print(f"Vacuum at {location} | Status: {status} -> Action: {action}")

        if action == "Suck":
            locations[location] = "Clean"
        elif action == "Right":
            location = "B"
        elif action == "Left":
            location = "A"

        print("World State:", locations)

    print("\nFinal State:", locations)
    print(f"\nSubmitted by: {NAME}, USN: {USN}")

vacuum_world()
```

```
Initial State: {'A': 'Clean', 'B': 'Dirty'} | Vacuum at: A
Vacuum at A | Status: Clean -> Action: Right
World State: {'A': 'Clean', 'B': 'Dirty'}
Vacuum at B | Status: Dirty -> Action: Suck
World State: {'A': 'Clean', 'B': 'Clean'}
Vacuum at B | Status: Clean -> Action: Left
World State: {'A': 'Clean', 'B': 'Clean'}
Vacuum at A | Status: Clean -> Action: Right
World State: {'A': 'Clean', 'B': 'Clean'}
Vacuum at B | Status: Clean -> Action: Left
World State: {'A': 'Clean', 'B': 'Clean'}

Final State: {'A': 'Clean', 'B': 'Clean'}

Submitted by: Shashank Ravindra Karanam, USN: 1BM23CS312
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)



Algorithm for DFS approach :

1) Initialize stack for DFS
2) Based on available moves, create child nodes
3) Select left-node, explore the moves & create child nodes.
4) Select left-most node and continue exploring it.
5) If the result is not obtained go to the parent's next node, until result is obtained.
6) Terminate the program

Output :

Visited : 14154
Depth : 49

Goal:
1 2 3
8 □ 4
7 6 5

```
from collections import deque


GOAL_STATE = (1, 2, 3,
              8, 0, 4,
              7, 6, 5)  # 0 represents the blank tile


MOVES = {
    'Up':    (-1, 0),
    'Down':  (1, 0),
    'Left':  (0, -1),
    'Right': (0, 1)
}

def index_to_pos(index):
    return index // 3, index % 3

def pos_to_index(row, col):
    return row * 3 + col

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    zero_row, zero_col = index_to_pos(zero_index)

    for move, (dr, dc) in MOVES.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_zero_index = pos_to_index(new_row, new_col)
            new_state = list(state)

            new_state[zero_index], new_state[new_zero_index] = new_state[new_zero_index], new_state[zero_index]
            neighbors.append((tuple(new_state), move))
    return neighbors

def bfs(start_state, max_depth=50):
    queue = deque([(start_state, [], [start_state])])
    visited = set([start_state])

    while queue:
        state, path_moves, path_states = queue.popleft()

        if state == GOAL_STATE:
            return path_moves, path_states

        if len(path_moves) < max_depth:
            for neighbor, move in get_neighbors(state):
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append((neighbor, path_moves + [move], path_states + [neighbor]))
    return None, None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()




name = input("Enter your name: ")
usn = input("Enter your USN: ")


start_state = (2, 8, 3,
               1, 6, 4,
               7, 0, 5)

print(f"\nSolver started for {name} (USN: {usn})")
print("\nStart State:")
print_puzzle(start_state)

moves, states = bfs(start_state, max_depth=50)

if moves is not None:
    print(f"Solution found in {len(moves)} moves!\n")

    print("States passed through from start to goal:")
    for i, state in enumerate(states):
        print(f"Step {i}:")
```

```
            print(f"Step {i}: ")
            print_puzzle(state)

    print("Moves to solve the puzzle:")
    print(moves)
else:
    print("No solution found within the depth limit.")
```

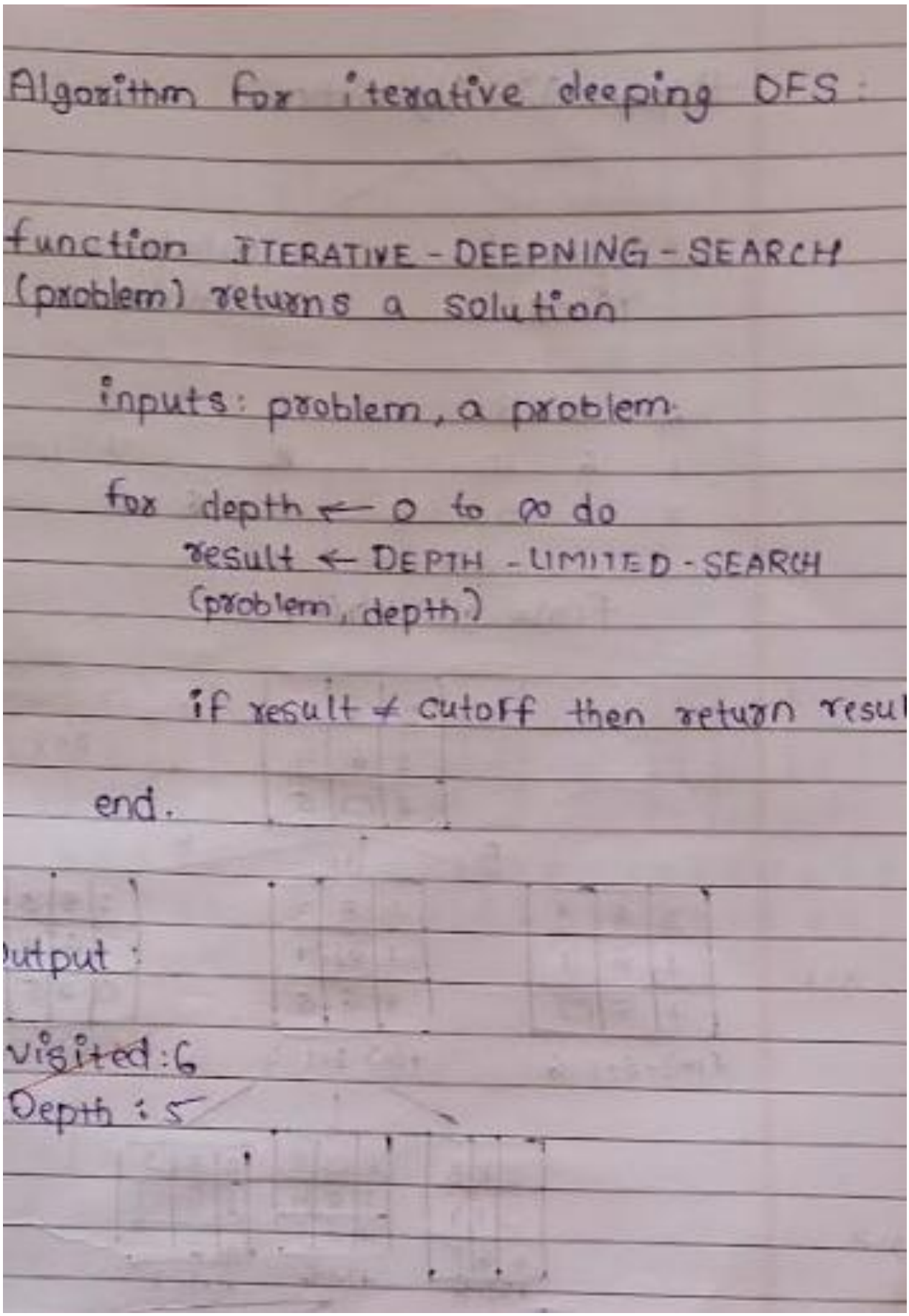
```
Enter your name: Shashank Ravindra
Enter your USN: 1BM23CS312

Solver started for Shashank Ravindra (USN: 1BM23CS312)

Start State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Solution found in 5 moves!

States passed through from start to goal:
Step 0:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Step 1:
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

Step 2:
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

Step 3:
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

Step 4:
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Step 5:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Moves to solve the puzzle:
['Up', 'Up', 'Left', 'Down', 'Right']
```

```
from collections import deque

GOAL_STATE = (1, 2, 3,
              8, 0, 4,
              7, 6, 5)  # 0 represents the blank tile


MOVES = {
    'Up':    (-1, 0),
    'Down':  (1, 0),
    'Left':  (0, -1),
    'Right': (0, 1)
}

def index_to_pos(index):
    return index // 3, index % 3

def pos_to_index(row, col):
    return row * 3 + col

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    zero_row, zero_col = index_to_pos(zero_index)

    for move, (dr, dc) in MOVES.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_zero_index = pos_to_index(new_row, new_col)
            new_state = list(state)

            new_state[zero_index], new_state[new_zero_index] = new_state[new_zero_index], new_state[zero_index]
            neighbors.append((tuple(new_state), move))
    return neighbors

def dfs(start_state, max_depth=50):
    stack = [(start_state, [], [start_state], 0)]
    visited = set()

    while stack:
        state, path_moves, path_states, depth = stack.pop()

        if state == GOAL_STATE:
            return path_moves, path_states

        if state in visited:
            continue
        visited.add(state)

        if depth < max_depth:
            for neighbor, move in get_neighbors(state):
                if neighbor not in visited:
                    stack.append((neighbor, path_moves + [move], path_states + [neighbor], depth + 1))
    return None, None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()


name = input("Enter your name: ")
usn = input("Enter your USN: ")


start_state = (2, 8, 3,
               1, 6, 4,
               7, 0, 5)

print(f"\nSolver started for {name} (USN: {usn})")
print("\nStart State:")
print_puzzle(start_state)

moves, states = dfs(start_state, max_depth=30)

if moves is not None:
    print(f"Solution found in {len(moves)} moves!\n")

    print("States passed through from start to goal:")
    for i, state in enumerate(states):
        print(f"Step {i}:")
```

```
            print(f"Step {i}: ")
            print_puzzle(state)

    print("Moves to solve the puzzle:")
    print(moves)
else:
    print("No solution found within the depth limit.")
```

```
Enter your name: Shashank Ravindra
Enter your USN: 1BM23CS312

Solver started for Shashank Ravindra (USN: 1BM23CS312)

Start State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Solution found in 29 moves!

States passed through from start to goal:
Step 0:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Step 1:
(2, 8, 3)
(1, 6, 4)
(7, 5, 0)

Step 2:
(2, 8, 3)
(1, 6, 0)
(7, 5, 4)

Step 3:
(2, 8, 3)
(1, 0, 6)
(7, 5, 4)

Step 4:
(2, 8, 3)
(0, 1, 6)
(7, 5, 4)

Step 5:
(0, 8, 3)
(2, 1, 6)
(7, 5, 4)

Step 6:
(8, 0, 3)
(2, 1, 6)
(7, 5, 4)

Step 7:
(8, 3, 0)
(2, 1, 6)
(7, 5, 4)

Step 8:
(8, 3, 6)
(2, 1, 0)
(7, 5, 4)
```

Program 2
Implement Iterative deepening search algorithm
Algorithm:



Code:

```
from collections import deque

# Define the goal state
GOAL_STATE = (1, 2, 3,
              8, 0, 4,
              7, 6, 5)  # 0 represents the blank tile

MOVES = {
    'Up':    (-1, 0),
    'Down':  (1, 0),
    'Left':  (0, -1),
    'Right': (0, 1)
}

def index_to_pos(index):
    return index // 3, index % 3

def pos_to_index(row, col):
    return row * 3 + col

def get_neighbors(state):
    neighbors = []
    zero_index = state.index(0)
    zero_row, zero_col = index_to_pos(zero_index)

    for move, (dr, dc) in MOVES.items():
        new_row, new_col = zero_row + dr, zero_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_zero_index = pos_to_index(new_row, new_col)
            new_state = list(state)
            new_state[zero_index], new_state[new_zero_index] = new_state[new_zero_index], new_state[zero_index]
            neighbors.append((tuple(new_state), move))
    return neighbors

def dls(state, goal_state, limit, path_moves, path_states, visited):
    """
    Depth-limited DFS:
    - state: current state
    - goal_state: goal state tuple
    - limit: current depth limit
    - path_moves: list of moves taken so far
    - path_states: list of states on the path so far
    - visited: set of visited states for current path (to avoid cycles)
    Returns (moves, states) if goal found else None
    """
    if state == goal_state:
        return path_moves, path_states
    if limit <= 0:
        return None
    visited.add(state)

    for neighbor, move in get_neighbors(state):
        if neighbor not in visited:
            result = dls(neighbor, goal_state, limit - 1,
                         path_moves + [move], path_states + [neighbor], visited)
            if result is not None:
                return result
    visited.remove(state)
    return None

def iddfs(start_state, goal_state, max_depth=50):
    for depth in range(max_depth + 1):
        visited = set()
        result = dls(start_state, goal_state, depth, [], [start_state], visited)
        if result is not None:
            return result
    return None, None

def print_puzzle(state):
    for i in range(0, 9, 3):
        print(state[i:i+3])
    print()

# --- Main ---

# Input user info
name = input("Enter your name: ")
usn = input("Enter your USN: ")

# Define a start state (make sure it is solvable)
start_state = (2, 8, 3
```

```
start_state = (2, 8, 3,
               1, 6, 4,


print(f"\nSolver started for {name} (USN: {usn})")
print("\nStart State:")
print_puzzle(start_state)

moves, states = iddfs(start_state, GOAL_STATE, max_depth=30)

if moves is not None:
    print(f"Solution found in {len(moves)} moves!\n")

    print("States passed through from start to goal:")
    for i, state in enumerate(states):
        print(f"Step {i}:")
        print_puzzle(state)

    print("Moves to solve the puzzle:")
    print(moves)
else:
    print("No solution found within the max depth limit.")
```

```
Enter your name: Shashank Ravindra
Enter your USN: 1BM23CS312

Solver started for Shashank Ravindra (USN: 1BM23CS312)

Start State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Solution found in 5 moves!

States passed through from start to goal:
Step 0:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)

Step 1:
(2, 8, 3)
(1, 0, 4)
(7, 6, 5)

Step 2:
(2, 0, 3)
(1, 8, 4)
(7, 6, 5)

Step 3:
(0, 2, 3)
(1, 8, 4)
(7, 6, 5)

Step 4:
(1, 2, 3)
(0, 8, 4)
(7, 6, 5)

Step 5:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)

Moves to solve the puzzle:
['Up', 'Up', 'Left', 'Down', 'Right']
```

7, 0, 5)

Program 1

Implement A* search algorithm

Algorithm:

A* search Algorithm:

- A* search evaluates nodes by adding g(n), depth & h(n), heuristic value.
- $f(n) = g(n) + h(n)$
- f(n) is the evaluation function which gives cheapest solution cost.
- g(n) is exact cost to reach goal.
- h(n) is assumed cost from current state to reach goal

Manhetten Distance:

- Find title's current position - in grid.
- Find goal position of title
- Calculated manhatten Distance:

$$f(n) = g(n) + h(n)$$

$$h(n) = |x_{curr} - x_{goal}| + |y_{curr} - y_{goal}|$$

- Sum distances of all results to get f(n)
- Return total distance for result.

Manhatten Distance:

1	5	8
3	2	
4	6	7

initial state

1	2	3
4	5	6
7	8	

final state

Solution:

1	5	8
3	2	
4	6	7

1

1	5	
3	2	8
4	6	7

1 2 3 4 5 6 7 8

0 1 3 1 1 2 2 2

= 12

1	5	8
3	2	7
4	6	

1 2 3 4 5 6 7 8

0 1 3 1 1 2 3 3

= 14

1	5	8
3		2
4	6	7

1 2 3 4 5 6 7 8

0 2 3 1 1 2 2 3

= 14

1

1		5
3	2	8
4	6	7

1 2 3 4 5 6 7 8

0 1 3 1 2 2 2 2

g = 13+1 = 14

	1	5
3	2	8
4	6	7

1 2 3 4 5 6 7 8

1 1 3 1 2 2 2 2

f = 14+2 = 16

1	2	5
3		8
4	6	7

1 2 3 4 5 6 7 8

0 0 3 1 2 2 2 2

f = 12+2 = 14

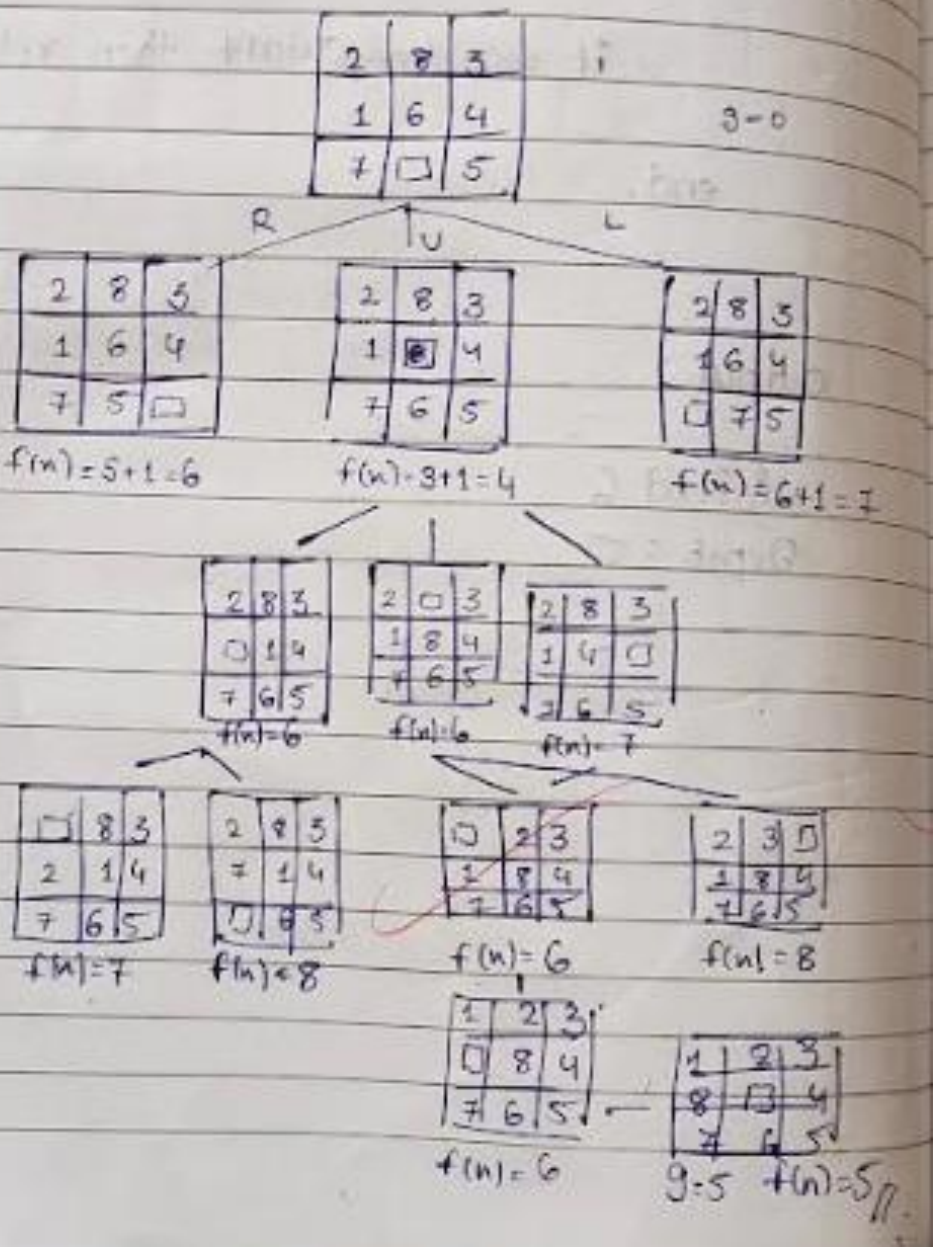
Apply A* Algorithm :

Misplaced Tiles

Manhatten Distance

2	8	3		1	2	3		
1	6	4		8		4		
7		5		7	6	5		

$f(n) = g(n) + h(n)$



```
import copy
import heapq

# Enter your details here
NAME = "Shashank Ravindra"
USN = "1BM23CS312"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")

# Goal state
goal_state = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]

def heuristic_manhattan(current_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = current_state[i][j]
            if value != 0:
                goal_x, goal_y = divmod(value - 1, 3)
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def is_goal(state):
    return state == goal_state

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    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 = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_manhattan(initial_state):
    open_list = []
    heapq.heappush(open_list, (heuristic_manhattan(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_manhattan(neighbor) + g + 1, g + 1, neighbor, path + [current_state]))
    return None

# Initial state example
initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [7, 0, 5]]

solution = a_star_manhattan(initial_state)

print("\nSolution Steps:\n")
for step in solution:
    for row in step:
        print(row)
    print()
```

Name: Shashank Ravindra
USN : 1BM23CS312

Solution Steps:

```
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

```python
import copy
import heapq


NAME = "Shashank Ravindra"
USN = "1BM23CS312"

print(f"Name: {NAME}")
print(f"USN : {USN}\n")


goal_state = [[1, 2, 3],
              [8, 0, 4],
              [7, 6, 5]]

def heuristic_misplaced(current_state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if current_state[i][j] != 0 and current_state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

def is_goal(state):
    return state == goal_state

def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    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 = copy.deepcopy(state)
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(new_state)
    return neighbors

def a_star_misplaced(initial_state):
    open_list = []
    heapq.heappush(open_list, (heuristic_misplaced(initial_state), 0, initial_state, []))
    visited = set()

    while open_list:
        heuristic, g, current_state, path = heapq.heappop(open_list)
        state_id = str(current_state)
        if state_id in visited:
            continue
        visited.add(state_id)

        if is_goal(current_state):
            return path + [current_state]

        for neighbor in get_neighbors(current_state):
            heapq.heappush(open_list, (heuristic_misplaced(neighbor) + g + 1, g + 1, neighbor, path + [current_state]))
    return None

initial_state = [[2, 8, 3],
                 [1, 6, 4],
                 [7, 0, 5]]

solution = a_star_misplaced(initial_state)

print("\nSolution Steps:\n")
for step in solution:
    for row in step:
        print(row)
    print()
```

```
Name: Shashank Ravindra
USN : 1BM23CS312


Solution Steps:
```

```
[2, 8, 3]
[1, 6, 4]
[7, 0, 5]

[2, 8, 3]
[1, 0, 4]
[7, 6, 5]

[2, 0, 3]
[1, 8, 4]
[7, 6, 5]

[0, 2, 3]
[1, 8, 4]
[7, 6, 5]

[1, 2, 3]
[0, 8, 4]
[7, 6, 5]

[1, 2, 3]
[8, 0, 4]
[7, 6, 5]
```

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:

Implement Hill climbing algorithm to solve N-Queens problem

Algorithm:

function HILL-CLIMBING (problem) returns a state that is a local maximum

current ← MAKE-NODE (problem.INITIAL-STATE)

loop do

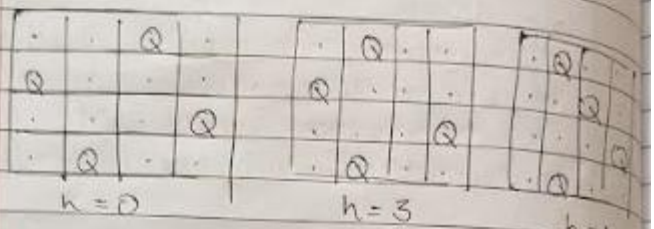
neighbor ← a highest-valued successor of current if neighbor.VALUE ≤ current.VALUE then return current.STATE.

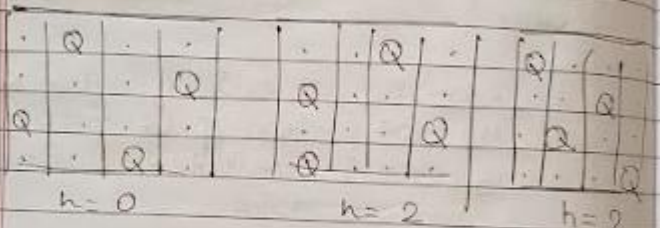
current ← neighbor.

- State: 4 queens on the board. No pair of queens are attacking each other.
- Variables: x_0, x_1, x_2, x_3, where x_j is the row position of the queen in column j. Assume that there is one queen per column.
- Initial state: random
- Goal: 4 queens, none attacking each other.
- neighbor rel^n: swap the row positions
- cost fun^s The no. of pairs of queens attacking each other.

Transition diagram

Initial = [3, 1, 2, 0]

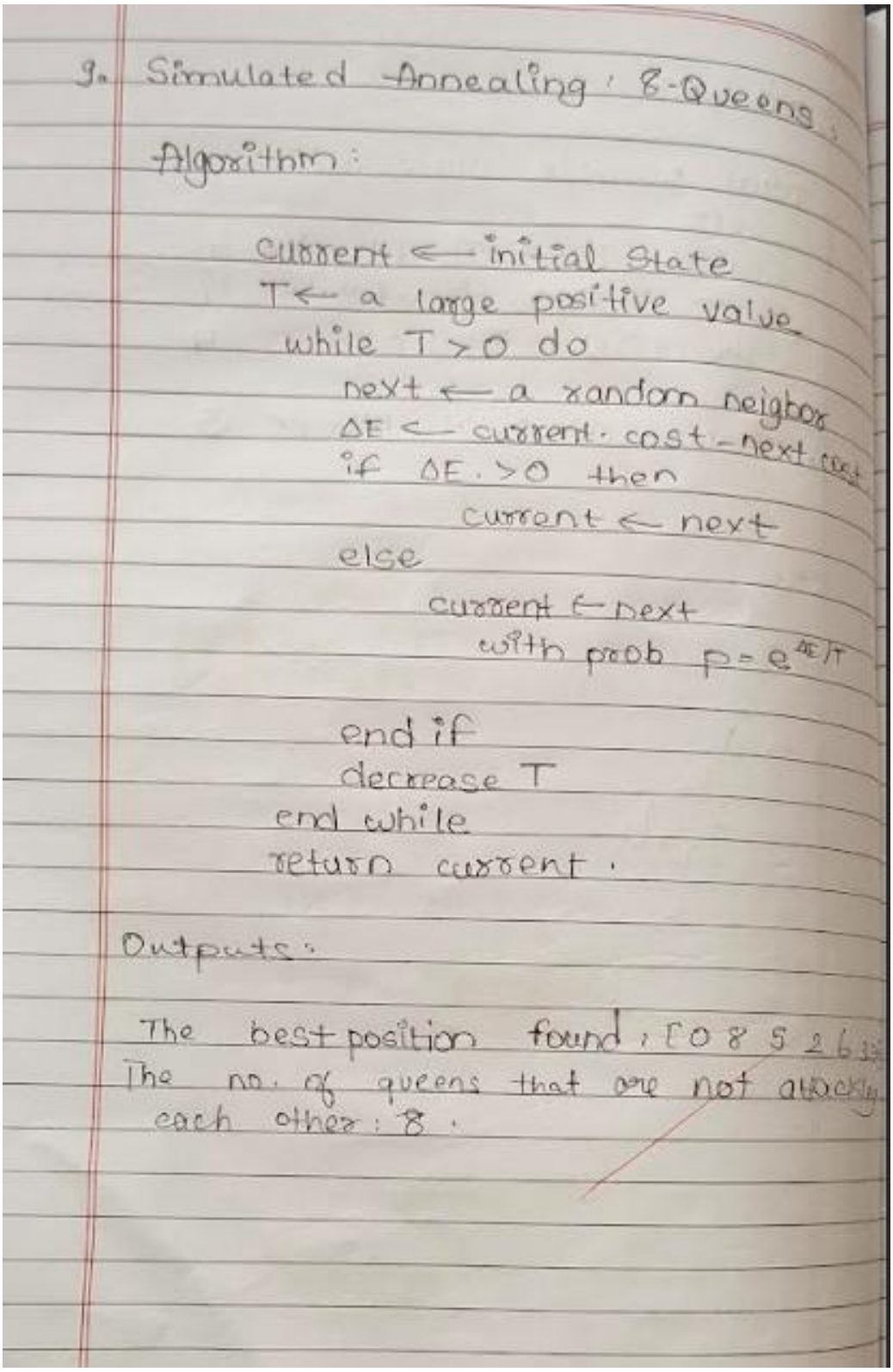




Output:

```python
import random


def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i+1, n):
            if state[i] == state[j]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost

def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


def get_best_neighbor(state):
    n = len(state)
    best_state = state[:]
    best_cost = calculate_cost(state)

    for col in range(n):
        for row in range(n):
            if state[col] != row:
                new_state = state[:]
                new_state[col] = row
                new_cost = calculate_cost(new_state)
                if new_cost < best_cost:
                    best_cost = new_cost
                    best_state = new_state
    return best_state, best_cost


def hill_climbing(n):
    current = random_state(n)
    current_cost = calculate_cost(current)

    while True:
        neighbor, neighbor_cost = get_best_neighbor(current)

        if neighbor_cost < current_cost:
            current, current_cost = neighbor, neighbor_cost
        else:
            break
```

Program 5
Simulated Annealing to Solve 8-Queens problem
Algorithm:



9. Simulated Annealing : 8-Queens :

Algorithm :

current ← initial state
T ← a large positive value
while T > 0 do
next ← a random neighbor
ΔE ← current. cost - next cost
if ΔE. > 0 then
current ← next
else
current ← next
with prob $p = e^{\Delta E/T}$
end if
decrease T
end while
return current.

Outputs :

The best position found : [0 8 5 2 6 3]
The no. of queens that are not attacking each other : 8.

```python
import random
import math


def calculate_cost(state):
    cost = 0
    n = len(state)
    for i in range(n):
        for j in range(i+1, n):
            if state[i] == state[j]:
                cost += 1
            if abs(state[i] - state[j]) == abs(i - j):
                cost += 1
    return cost


def random_state(n):
    return [random.randint(0, n-1) for _ in range(n)]


def get_neighbor(state):
    n = len(state)
    neighbor = state[:]
    col = random.randint(0, n-1)
    row = random.randint(0, n-1)
    neighbor[col] = row
    return neighbor


def simulated_annealing(n, max_steps=100000, start_temp=100, cooling_rate=0.99):
    current = random_state(n)
    current_cost = calculate_cost(current)
    T = start_temp

    for step in range(max_steps):
        if current_cost == 0:
            break
        neighbor = get_neighbor(current)
        neighbor_cost = calculate_cost(neighbor)
        delta = neighbor_cost - current_cost

        if delta < 0 or random.uniform(0, 1) < math.exp(-delta / T):
            current, current_cost = neighbor, neighbor_cost

        T = T * cooling_rate
        if T < 1e-5:
            break
```

```
    return current, current_cost


solution, cost = simulated_annealing (8)

print("Name: Shashank Ravindra" )
print("USN : 1BM23CS312" )
print("\nSimulated Annealing for 8-Queens Problem" )
print("Final State (row positions per column):" , solution)
print("Final Cost (0 means solved):" , cost)
if cost == 0:
    print("Solution Found: Queens are safe!" )
else:
    print("Stuck in Local Optimum (Not a solution)" )
```

```
Name: Shashank Ravindra
USN : 1BM23CS312

Simulated Annealing for 8-Queens Problem
Final State (row positions per column): [4, 6, 1, 5, 2, 0, 7, 3]
Final Cost (0 means solved): 0
Solution Found: Queens are safe!
```

Program 6

Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm:

Propositional Logic :

Implementation of truth table enumeration algorithm for deciding propositional entailment - i.e. create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

$\alpha = A \vee B$ $\quad KB = (A \vee C) \wedge (B \vee \neg C)$

checking if $KB \models \alpha$

A	B	C	A ∨ C	B ∨ ¬C	KB	α
false	false	false	false	true	false	false
false	false	true	true	false	false	false
false	true	false	false	true	false	true
false	true	true	true	true	true	true
true	false	false	true	true	true	true
true	false	true	true	false	false	true
true	true	false	true	true	true	true
true	true	true	true	true	true	true

Algorithm :

1. Input : → KB and α (Query).

2. List all symbols :
   - collect all propositional variables appearing in KB & the query.

3. Generate models :
   - A model is a truth assignment (True / False) to every symbol.
   - Create all possible combinations of truth values.

4. Check entailment :
   - For each model :
     - If KB is true, check if α is also true.
     - If you ever find a model where KB is true but α is false, then KB does not entail α.

5. Final Decision :
   - If in every model where KB is true, α is also true → return entailed.
   - Otherwise → return Not entailed

Output:

Truth Table evaluation

Model		KB	Query
P	Q		
True	True	True	True
True	False	False	False
False	True	False	True
False	False	False	False

KB: (P → Q) and (P)

Query: Q

Does KB entail Query?: YES.

```python
import itertools

def pl_true(expr, model):
    if isinstance(expr, str):
        return model[expr]
    elif isinstance(expr, tuple):
        op = expr[0]
        if op == "not":
            return not pl_true(expr[1], model)
        elif op == "and":
            return pl_true(expr[1], model) and pl_true(expr[2], model)
        elif op == "or":
            return pl_true(expr[1], model) or pl_true(expr[2], model)
        elif op == "implies":
            return (not pl_true(expr[1], model)) or pl_true(expr[2], model)
    return False

def get_symbols(expr):
    if isinstance(expr, str):
        return {expr}
    elif isinstance(expr, tuple):
        return get_symbols(expr[1]) | (get_symbols(expr[2]) if len(expr) > 2 else set())
    return set()

def tt_entails_print(KB, query):
    symbols = list(get_symbols(KB) | get_symbols(query))
    all_models = list(itertools.product([True, False], repeat=len(symbols)))

    entailment = True
    print("\nTruth Table Evaluation:")
    print("-" * 50)
    print("Model".ljust(20), "KB".ljust(10), "Query".ljust(10))
    print("-" * 50)

    for values in all_models:
        model = dict(zip(symbols, values))
        kb_val = pl_true(KB, model)
        q_val = pl_true(query, model)

        print(str(model).ljust(20), str(kb_val).ljust(10), str(q_val).ljust(10))

        if kb_val and not q_val:
            entailment = False

    print("-" * 50)
    return entailment


KB = ("and", ("implies", "P", "Q"), "P")
query = "Q"

result = tt_entails_print(KB, query)

print("\nName: Shashank Ravindra")
print("USN : 1BM23CS312")
print("\nKnowledge Base: (P → Q) and (P)")
print("Query: Q")
print("Does KB entail Query? :", "YES " if result else "NO ")
```

```
Truth Table Evaluation:
--------------------------------------------------
Model                KB         Query
--------------------------------------------------
{'P': True, 'Q': True} True       True
{'P': True, 'Q': False} False      False
{'P': False, 'Q': True} False      True
{'P': False, 'Q': False} False      False
--------------------------------------------------

Name: Shashank Ravindra
USN : 1BM23CS312
```

```
Knowledge Base: (P → Q) and (P)
Query: Q
Does KB entail Query? : YES
```

Program 7

Implement unification in first order logic

Algorithm:

Unification Algorithm :

Algorithm: Unify (Ψ_1, Ψ_2)

Step 1: If Ψ_1 or Ψ_2 is a variable or constant, then:

a) If Ψ_1 or Ψ_2 are identical, then return NIL.

b) Else if Ψ_1 is a variable,

a. then if Ψ_1 occurs in Ψ_2, then return FAILURE

b. Else return $\{(\Psi_2/\Psi_1)\}$.

c) Else if Ψ_2 is a variable,

a. If Ψ_2 occurs in Ψ_1 then return FAILURE,

b. Else return $\{(\Psi_1/\Psi_2)\}$.

d) Else return FAILURE.

Step 2: If the initial Predicate symbol in Ψ_1 and Ψ_2 are not same, then return FAILURE.

Step 3: If Ψ_1 and Ψ_2 have a different number of arguments, then return FAILURE.

Step 4: Set Substitution set(SUBST) to NIL.

Step 5: For i = 1 to the number of elements in ψ_1.

a) Call unify function with the ith element of ψ_1 and ith element of ψ_2, and put the result into S.

b) If S = failure then returns Failure.

c) If S ≠ NIL then do,

a. Apply S to the remainder of both L1 and L2.

b. SUBST = APPEND (S, SUBST).

Step 6: Return SUBST.

i) Find Most General Unifier (MGU) of
{ p(b, X, f(g(Z)) and p(Z, f(Y), f(Y)) }

S = { }

$Z \rightarrow b \rightarrow S = \{ Z \rightarrow b \}$

$X \rightarrow f(Y) \rightarrow S = \{ Z \rightarrow b, X \rightarrow f(Y) \}$

$Y \rightarrow g(Z) \rightarrow$ Substitute into $X \rightarrow X \rightarrow f(g(Z))$

Final MGU: $\{ Z \rightarrow b, X \rightarrow f(g(Z)), Y \rightarrow g(Z) \}$

ii) { Q(a, g(x, a), f(y)) and Q(a, g(f(b), a), x) }.

S = { }

$X \rightarrow f(b) \rightarrow S = \{ X \rightarrow f(b) \}$

$X \rightarrow f(Y) \therefore Y \rightarrow b \rightarrow S = \{ X \rightarrow f(b), Y \rightarrow b \}$

Final MGU: $\{ X \rightarrow f(b), Y \rightarrow b \}$.

iii) { p(f(a), g(y)), p(x, x) }

S = { }

From arg 1 : x → f(a) → S = { x → f(a) }

From arg 2 : x must = g(y) → req f(a) = g(y)

→ impossible (f ≠ g)

UNIFICATION FAILS.

iv) { prime (11) and prime (y) }

S = { }

y → 11

Final MGU : { y → 11 }.

v) { Knows (John, x), Knows (y, mother (y)) }

S = { }

John → y → S = { John → y }

x → mother (y) → S ≠

y → mother (John) → S = { y → John, x → mother (John) }

Final MGU: { y → John, x → mother (John) }

vi) { Knows (John, x), Knows (y, Bill) }

S = { }

y → John → S = { y → John }

x → Bill → S = { y → John, x → Bill }

Final MGU = { y → John, x → Bill }

```
# Unifier that allows you to declare which names are variables.
# Works for the example: p(b,X,f(g(Z)))  and  p(z,f(Y),f(Y))

import re
from collections import deque
print("Shashank Ravindra Karanam")
print("1BM23CS312")


# === Configuration: list the names you want treated as variables ===
VARIABLES = {"X", "Y", "Z", "z"}    # add or remove names as needed

# === Tokenizer & Parser ===
TOKEN_REGEX = r"\s*([A-Za-z0-9_]+|[,()])\s*"

def tokenize(s):
    return [t for t in re.findall(TOKEN_REGEX, s) if t.strip()]

class Term:
    def __init__(self, name, args=None):
        self.name = name
        self.args = args or []
    def is_variable(self):
        return (not self.args) and (self.name in VARIABLES)
    def is_constant(self):
        return (not self.args) and (self.name not in VARIABLES)
    def __repr__(self):
        if self.args:
            return f"{self.name}({', '.join(map(repr, self.args))})"
        return self.name

def parse_term(tokens):
    name = tokens.popleft()
    if tokens and tokens[0] == '(':
        tokens.popleft()
        args = []
        if tokens[0] != ')':
            while True:
                args.append(parse_term(tokens))
                if tokens[0] == ',':
                    tokens.popleft()
                    continue
                elif tokens[0] == ')':
                    break
        tokens.popleft()
        return Term(name, args)
    return Term(name, [])
```

```
    def parse(s):
        tks = deque(tokenize(s))
        term = parse_term(tks)
        if tks:
            raise ValueError("Extra tokens after parse: " + " ".join(tks))
        return term

    # === Substitution utilities ===
    def apply_subst(t, subst):
        if t.is_variable():
            if t.name in subst:
                return apply_subst(subst[t.name], subst)
            return t
        if t.args:
            return Term(t.name, [apply_subst(a, subst) for a in t.args])
        return t

    def occurs_check(var, term, subst):
        t = apply_subst(term, subst)
        if t.is_variable():
            return t.name == var
        if t.args:
            return any(occurs_check(var, a, subst) for a in t.args)
        return False

    # === Robinson Unification (iterative) ===
    def unify(a, b):
        subst = {}
        eqs = deque([(a, b)])
        trace = []
        step = 0
        while eqs:
            step += 1
            s, t = eqs.popleft()
            s = apply_subst(s, subst)
            t = apply_subst(t, subst)
            trace.append((step, s, t, dict(subst)))
            if repr(s) == repr(t):
                continue
            if s.is_variable():
                if occurs_check(s.name, t, subst):
                    return None, trace
                subst[s.name] = t
                continue
            if t.is_variable():
                if occurs_check(t.name, s, subst):
                    return None, trace
                subst[t.name] = s
                continue
            # both are function/constant (non-variable)
            if s.args and t.args:
```

```
            if s.name != t.name or len(s.args) != len(t.args):
                return None, trace
            for sa, ta in reversed(list(zip(s.args, t.args))):
                eqs.appendleft((sa, ta))
            continue
        # different constants -> fail
        return None, trace
    return subst, trace

# === Pretty print trace and result ===
def print_result(subst, trace):
    for item in trace:
        step, s, t, st = item
        print(f"Step {step}: unify {s}  with  {t}    S = {st}")
    if subst is None:
        print("\nResult: UNIFICATION FAILED")
    else:
        # simplify RHS by applying subst
        final = {k: apply_subst(v, subst) for k, v in subst.items()}
        print("\nFinal MGU:")
        for k, v in final.items():
            print(f" {k} -> {v}")

# === Example run ===
if __name__ == "__main__":
    s1 = "p(b,X,f(g(Z)))"
    s2 = "p(z,f(Y),f(Y))"
    t1 = parse(s1)
    t2 = parse(s2)
    subst, trace = unify(t1, t2)
    print("Input:")
    print(" ", s1)
    print(" ", s2)
    print("\nTrace:")
    print_result(subst, trace)
```

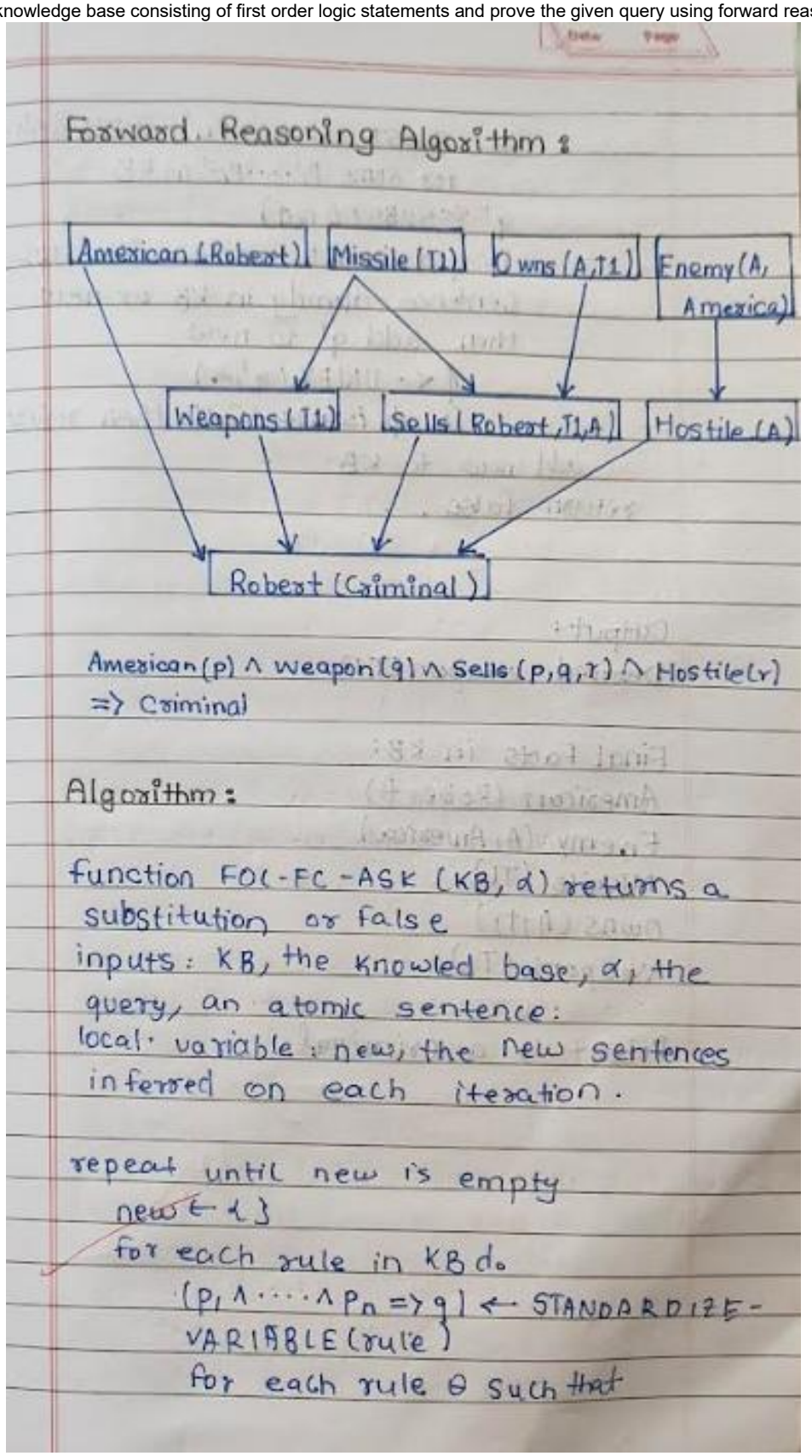
```
Shashank Ravindra Karanam
1BM23CS312
Input:
  p(b,X,f(g(Z)))
  p(z,f(Y),f(Y))

Trace:
Step 1: unify p(b, X, f(g(Z)))  with  p(z, f(Y), f(Y))    S = {}
Step 2: unify b  with  z    S = {}
Step 3: unify X  with  f(Y)    S = {'z': b}
Step 4: unify f(g(Z))  with  f(Y)    S = {'z': b, 'X': f(Y)}
Step 5: unify g(Z)  with  Y    S = {'z': b, 'X': f(Y)}

Final MGU:
 z -> b
```

```
    X -> f(g(Z))
    Y -> g(Z)
```

Program 8

Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



SUBST (θ, P1, ..., Pn) = SUBST(θ, P1', ..., Pn')
for some P1', ..., Pn' in KB
q' ← SUBST(θ, q)
if q does not unify with some
sentence already in KB or new
then add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return
add new to KB
return false.

Output:
Derived : Weapon(T1)

Final Facts in KB:
American (Robert)
Enemy (A, America)
Missile (T1)
Owns (A, T1)
Weapons (T1)

Robert is a criminal.

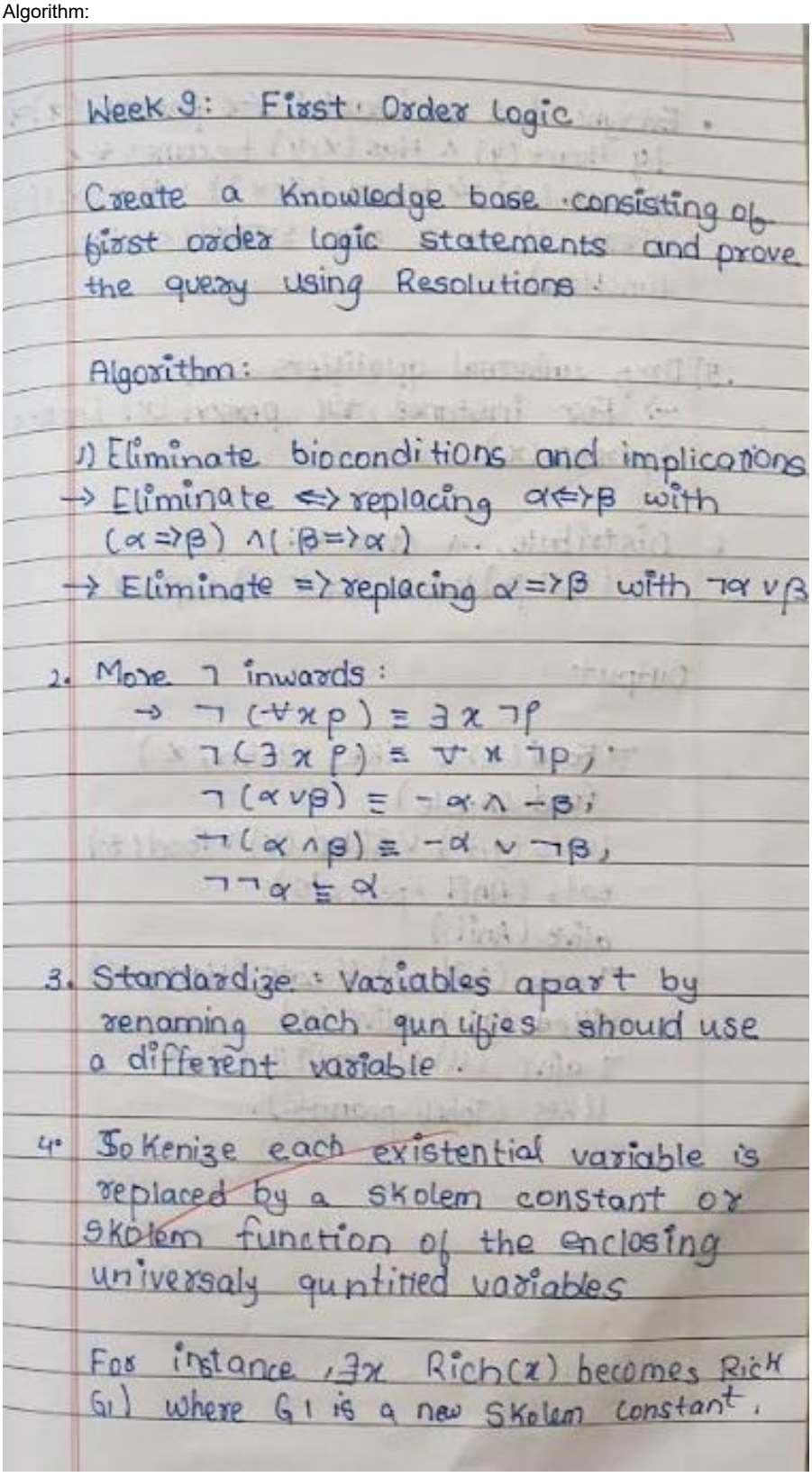
```
# Week 8 - AI Lab: Forward Chaining in FOL


# Define the Knowledge Base
knowledge_base = [
    "American(Robert)",
    "Enemy(A, America)",
    "Missile(T1)",
    "Owns(A, T1)",
    # Rules
    "Missile(x) => Weapon(x)",
    "Enemy(x, America) => Hostile(x)",
    "Missile(x) ∧ Owns(A, x) => Sells(Robert, x, A)",
    "American(p) ∧ Weapon(q) ∧ Sells(p, q, r) ∧ Hostile(r) => Criminal(p)"
]

# Facts derived initially
facts = set(["American(Robert)", "Enemy(A, America)", "Missile(T1)", "Owns(A, T1)"])

# Forward chaining simulation
def forward_chain(kb, facts):
    added = True
    while added:
        added = False
        for rule in kb:
            if "=>" in rule:
                premise, conclusion = rule.split("=>")
                premise = [p.strip() for p in premise.split("∧")]
                conclusion = conclusion.strip()

                # Check if all premises are satisfied
                match = True
                subs = {}
                for p in premise:
                    p = p.replace(" ", "")
                    if "(x)" in p:
                        base = p.replace("(x)", "(T1)")
                    elif "(p)" in p:
                        base = p.replace("(p)", "(Robert)")
                    elif "(q)" in p:
                        base = p.replace("(q)", "(T1)")
                    elif "(r)" in p:
                        base = p.replace("(r)", "(A)")
                    else:
                        base = p

                    if base not in facts:
                        match = False
                        break
```

```
                    if match:
                        # Substitute variables in conclusion
                        derived = (
                            conclusion.replace("(x)", "(T1)")
                            .replace("(p)", "(Robert)")
                            .replace("(q)", "(T1)")
                            .replace("(r)", "(A)")
                        )
                        if derived not in facts:
                            facts.add(derived)
                            added = True
                            print(f"Derived: {derived}")
    return facts

# Run forward chaining
final_facts = forward_chain(knowledge_base, facts)

print("\nFinal Facts in Knowledge Base:")
for f in sorted(final_facts):
    print(f)

if "Criminal(Robert)" in final_facts:
    print("\n Robert is proven to be a Criminal.")
else:
    print("\n Could not prove Robert is a Criminal.")

print("\nName: Shashank Ravindra")
print("USN: 1BM23CS312")
```

```
Derived: Weapon(T1)

Final Facts in Knowledge Base:
American(Robert)
Enemy(A, America)
Missile(T1)
Owns(A, T1)
Weapon(T1)

 Could not prove Robert is a Criminal.

Name: Shashank Ravindra
USN: 1BM23CS312
```

Program 9

Create a knowledge base consisting of first order logic statements and prove the given query using Resolution Algorithm:



Week 9: First Order Logic

Create a Knowledge base consisting of first order logic statements and prove the query using Resolutions.

Algorithm:

1) Eliminate biconditions and implications
→ Eliminate $\Leftrightarrow$ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
→ Eliminate $\Rightarrow$ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2. Move $\neg$ inwards:
→ $\neg(\forall x\, p) \equiv \exists x \neg p$
$\neg(\exists x\, p) \equiv \forall x \neg p$,
$\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$,
$\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$,
$\neg\neg\alpha \equiv \alpha$

3. Standardize: Variables apart by renaming each quantifies should use a different variable.

4. Skolemize each existential variable is replaced by a skolem constant or skolem function of the enclosing universaly quantified variables

For instance, $\exists x$ Rich(x) becomes Rich(G1) where G1 is a new Skolem constant.

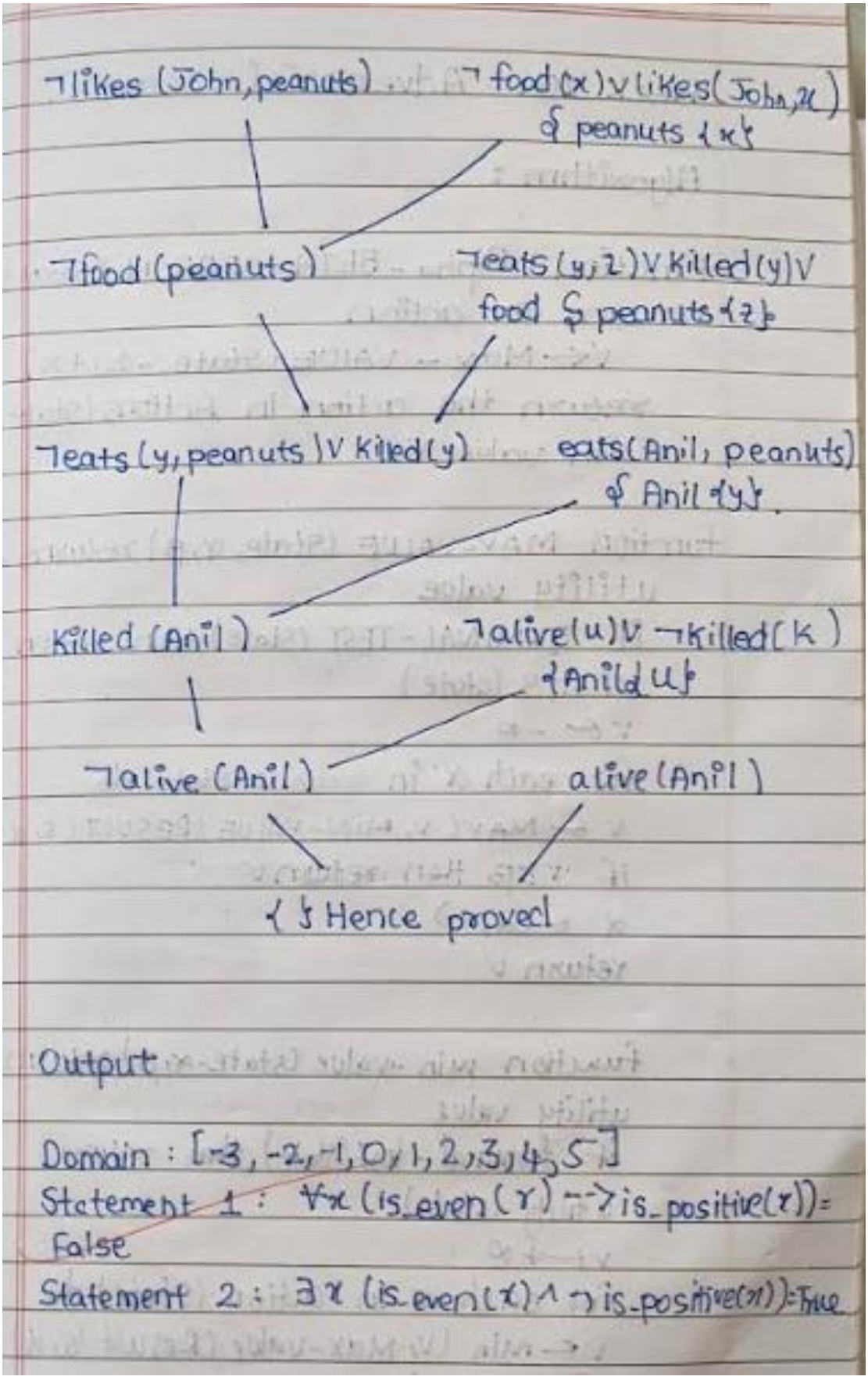
- Everyone has a heart "$\forall x$: person(x) $\exists y$ Heart(y) $\wedge$ Has(x,y) becomes $\forall x$ person(x) $\Rightarrow$ Heart(H(x)) $\wedge$ Has(x, where H is a new symbol (Skolem function)

5) Drop universal quatifiers :
-> For instance, $\forall x$ person(x) beco person(x)

6. Distribute $\wedge$ over $\vee$:
$(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$

Output

$\neg$ Food(x) $\vee$ likes(John, x)
food(Apple)
$\neg$ eats(y, z) $\vee$ killed(y) $\vee$ food(z)
eats(Anil, peanuts)
alive(Anil)
$\neg$ eats(Anil, w) $\vee$ eats(Harry, w)
Killed(g) $\vee$ alive(g)
$\neg$ alive(u) $\vee$ $\neg$ Killed(u)
likes(John, peanuts).



Output

Domain : [-3, -2, -1, 0, 1, 2, 3, 4, 5]

Statement 1 : $\forall x$ (is_even (x) --> is_positive(x)) = False

Statement 2 : $\exists x$ (is_even (x) $\wedge$ ¬ is_positive(x)) = True

```
name = "Shashank Ravindra Karanam"
usn  = "1BM23CS312"

print("Name:", name)
print("USN :", usn)
print("Topic: Prove 'John likes peanuts' using KB (forwa  rd chaining)")



# --------------------------
# Part 2: Knowledge base + rules (forward chaining  )
# We'll represent facts as tuples: ('predicate', a  rg1, arg2?, ...)
# e.g. ('food','apple'), ('eat','anil','peanuts'),  ('alive','anil')
# --------------------------

# Initial facts from the PPT
facts = set([
    ('food', 'apple'),
    ('food', 'vegetable'),
    ('eat', 'anil', 'peanuts'),
    ('alive', 'anil'),
    # the example says "Harry eats everything that Ani  l eats" -> handled as a rule
    # John likes all kinds of food -> handled as a rul  e
    # anything anyone eats and not killed is food -> r  ule
    # alive -> not killed and not killed -> alive (two  -way) -> rules
])

# Helper to add a fact and know if it was new
def add_fact(f):
    if f not in facts:
        facts.add(f)
        # print the derivation as we add it
        print("Derived:", f)
        return True
    return False

# Rules (as functions that check facts and possibl  y add new ones)
def rule_alive_implies_not_killed():
    # alive(x) -> not_killed(x)
    added = False
    for (pred, *args) in list(facts):
        if pred == 'alive':
            x = args[0]
            added |= add_fact(('not_killed', x))
    return added

def rule_not_killed_implies_alive():
    # not_killed(x) -> alive(x)
```

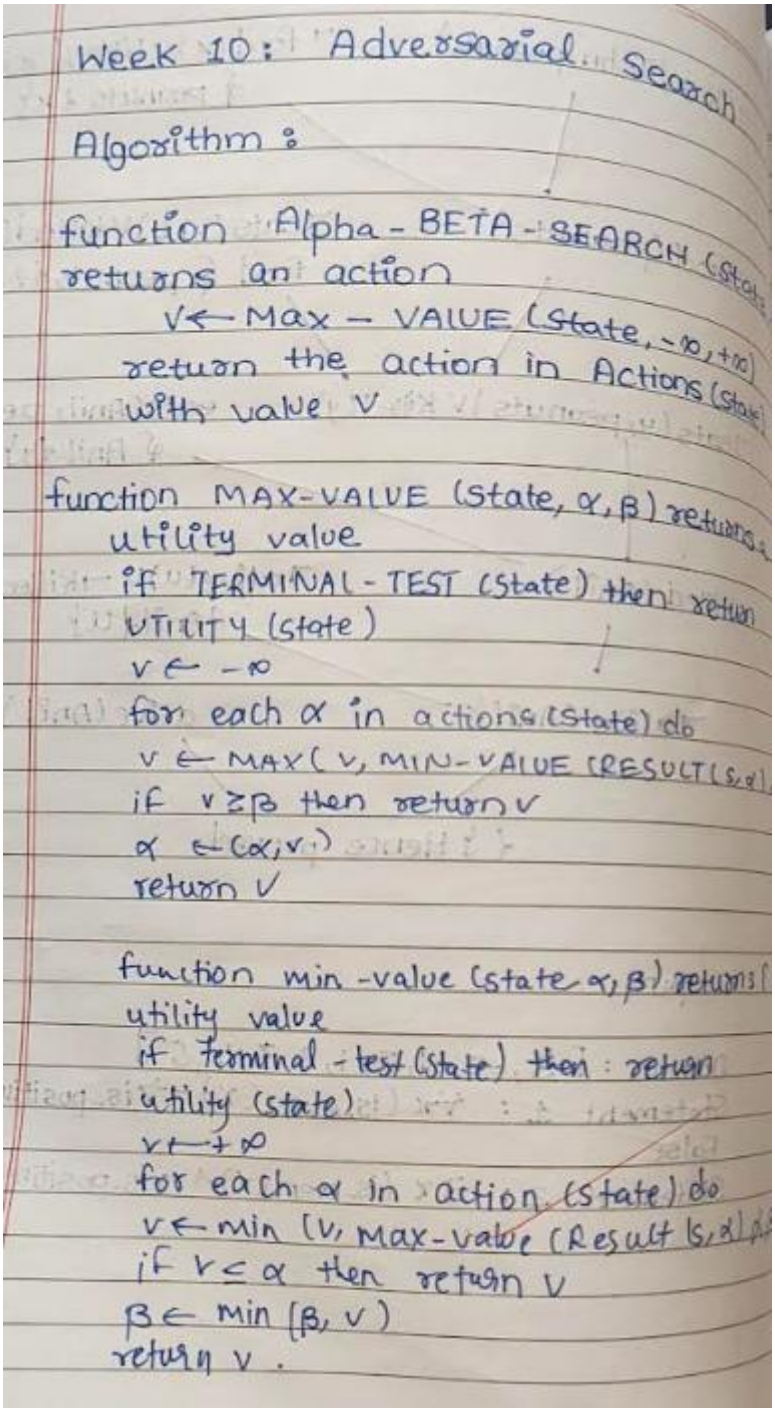
```
    added = False
    for (pred, *args) in list(facts):
        if pred == 'not_killed':
            x = args[0]
            added |= add_fact(('alive', x))
    return added

def rule_eat_and_not_killed_implies_food():
    # eat(person, item) ∧ not_killed(person) -> food(item)
    added = False
    # collect persons with not_killed
    nk = {args[0] for (pred,*args) in facts if pred == 'not_killed'}
    for (pred, *args) in list(facts):
        if pred == 'eat':
            person, item = args
            if person in nk:
                added |= add_fact(('food', item))
    return added

def rule_harry_eats_everything_anil_eats():
    # eat(anil, X) -> eat(harry, X)
    added = False
    for (pred, *args) in list(facts):
        if pred == 'eat' and args[0] == 'anil':
            _, item = args
            added |= add_fact(('eat', 'harry', item))
    return added

def rule_john_likes_all_food():
    # food(x) -> likes(john, x)
    added = False
    for (pred, *args) in list(facts):
        if pred == 'food':
            item = args[0]
            added |= add_fact(('likes', 'john', item))
    return added

# Collect rule functions in a list for the forward-chaining loop
rules = [
    rule_alive_implies_not_killed,
    rule_not_killed_implies_alive,
    rule_eat_and_not_killed_implies_food,
    rule_harry_eats_everything_anil_eats,
    rule_john_likes_all_food
]

# Run forward chaining until no new facts appear
changed = True
iteration = 0
print("Initial facts:", facts, "\n")
while changed:
```

```
    iteration += 1
    changed = False
    # apply each rule once per iteration
    for r in rules:
        if r():
            changed = True
    # safety: break if too many iterations (shouldn't  happen here)
    if iteration > 20:
        print("Stopping after 20 iterations (safety)." )
        break

print("\nFinal facts:" )
for f in sorted(facts):
    print(" ", f)

# ---------------------------
# Check the query: likes(john, peanuts)
# ---------------------------
query = ('likes', 'john', 'peanuts' )
print("\nQuery:", query)
print("Proved:" , query in facts)   # will print True if derived
```

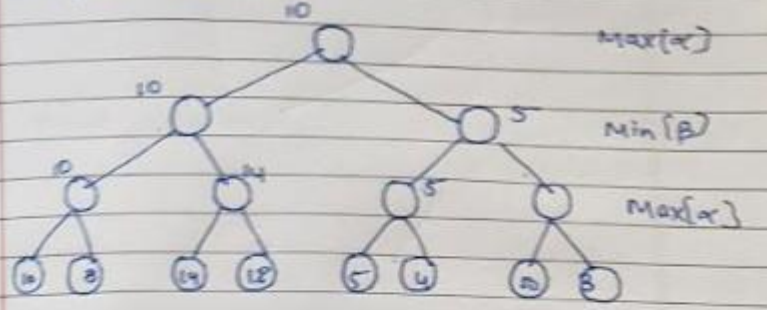
```
Name: Shashank Ravindra Karanam
USN : 1BM23CS312
Topic: Prove 'John likes peanuts' using KB (forward chaining)
Initial facts: {('food', 'vegetable'), ('eat', 'anil', 'peanuts'), ('food', 'apple'), ('alive', 'anil')}

Derived: ('not_killed', 'anil')
Derived: ('food', 'peanuts')
Derived: ('eat', 'harry', 'peanuts')
Derived: ('likes', 'john', 'apple')
Derived: ('likes', 'john', 'peanuts')
Derived: ('likes', 'john', 'vegetable')

Final facts:
  ('alive', 'anil')
  ('eat', 'anil', 'peanuts')
  ('eat', 'harry', 'peanuts')
  ('food', 'apple')
  ('food', 'peanuts')
  ('food', 'vegetable')
  ('likes', 'john', 'apple')
  ('likes', 'john', 'peanuts')
  ('likes', 'john', 'vegetable')
  ('not_killed', 'anil')

Query: ('likes', 'john', 'peanuts')
Proved: True
```

Program 10
Implement Alpha-Beta Pruning.

Algorithm:

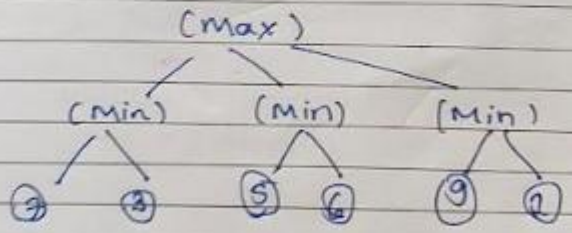


Output (examples)



Output:

Game tree:



Calculated value at root (max) = 5

```python
# ---- Part 1: Print your Name and USN ----
name = "Shashank Ravindra Karanam"
usn = "1BM23CS312"

print("====================================" )
print("Name:", name)
print("USN :", usn)
print("Topic: Alpha-Beta Pruning (Adversarial Search)" )
print("====================================\n" )


# ---- Part 2: Alpha-Beta Pruning Implementation -  ---

# Game Tree:
#             A
#         /       \
#        B         C
#      / | \     / | \
#     D  E  F   G  H  I
# Leaf values: D=3, E=5, F=6, G=9, H=1, I=2

# Each level alternates between MAX and MIN

explored_nodes =  []

def alphabeta(node, depth, alpha, beta, maximizingPlayer , tree):
    explored_nodes.append (node)

    # If it's a leaf node (its value is an int)
    if isinstance(tree[node], int):
        return tree[node]

    # If depth is 0, return 0 (base case safety)
    if depth ==  0:
        return 0

    if maximizingPlayer :
        value =  float('-inf')
        for child in tree[node]:
            # Recursively call for child
            value =  max(value, alphabeta(child, depth-1, alpha, beta, False, tree))
            alpha =  max(alpha, value)
            if alpha >= beta :
                print(f"Pruned remaining children of  {node} (alpha={alpha}, beta={beta})")
                break  # Beta cutoff
        return value
    else:
```

```
        value = float('inf')
        for child in tree[node]:
            value = min(value, alphabeta(child, depth-1, alpha, beta, True, tree))
            beta = min(beta, value)
            if beta <= alpha:
                print(f"Pruned remaining children of {node} (alpha={alpha}, beta={beta})")
                break  # Alpha cutoff
        return value


# Define the tree structure
tree = {
    'A': ['B', 'C'],
    'B': ['D', 'E', 'F'],
    'C': ['G', 'H', 'I'],
    'D': 3,
    'E': 5,
    'F': 6,
    'G': 9,
    'H': 1,
    'I': 2
}

# Run Alpha-Beta search
print("Running Alpha-Beta Pruning...\n")
best_value = alphabeta('A', 3, float('-inf'), float('inf'), True, tree)

print("\n====================================")
print("Best value for Root Node (A):", best_value)
print("Nodes explored (in order):", explored_nodes)
print("====================================\n")
```

```
====================================
Name: Shashank Ravindra Karanam
USN : 1BM23CS312
Topic: Alpha-Beta Pruning (Adversarial Search)
====================================

Running Alpha-Beta Pruning...

Pruned remaining children of C (alpha=3, beta=1)

====================================
Best value for Root Node (A): 3
Nodes explored (in order): ['A', 'B', 'D', 'E', 'F', 'C', 'G', 'H']
====================================
```