VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Dyuthi (1BM23CS097)

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 **Dyuthi (1BM23CS097),** 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.

Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

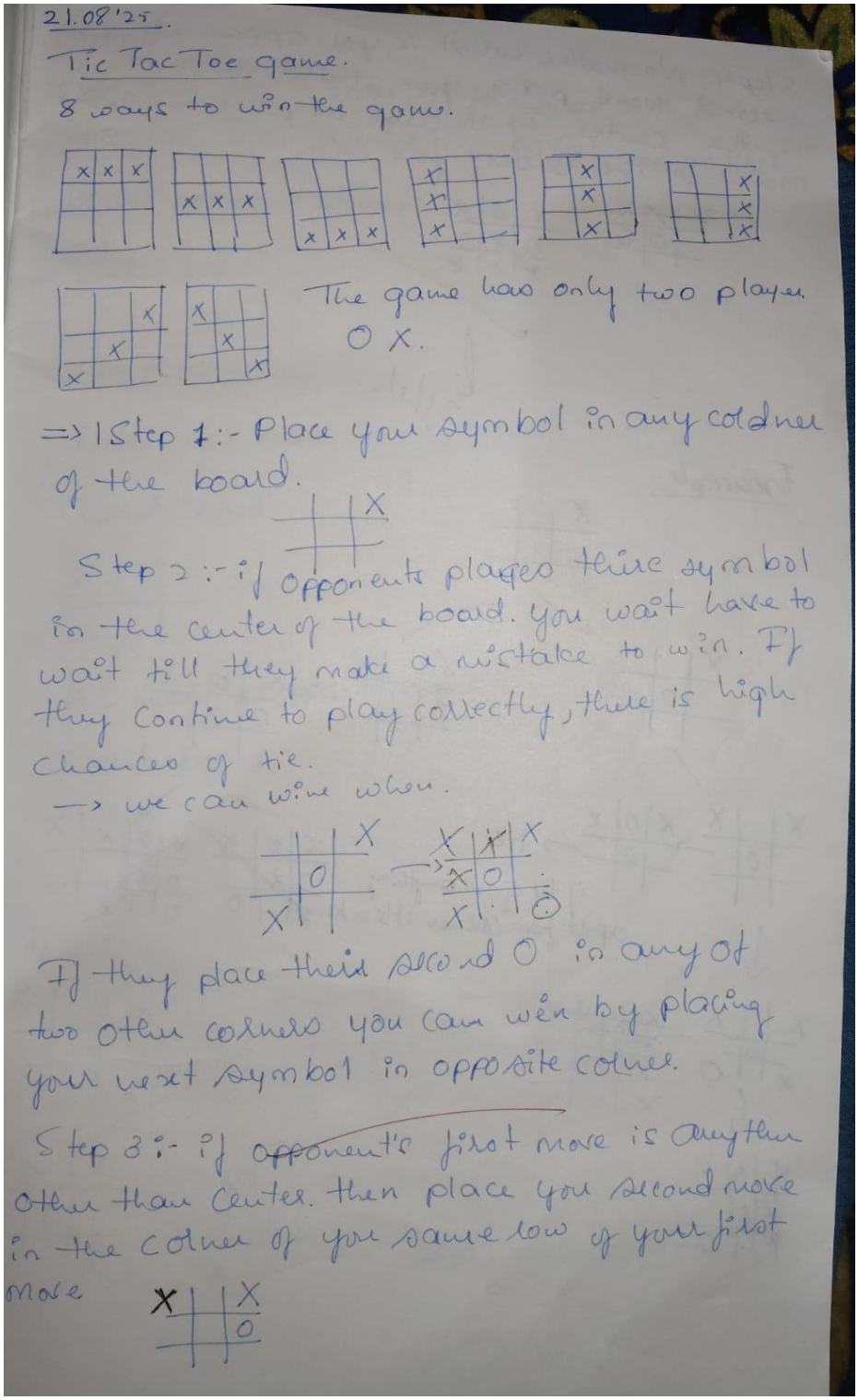
Index

Github Link:
https://github.com/GayathriS-CSE/AI-LAB

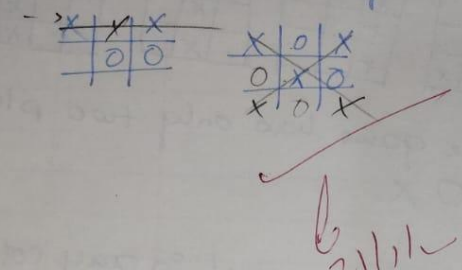
Program 1

Implement Tic –Tac –Toe Game Implement
vacuum cleaner agent

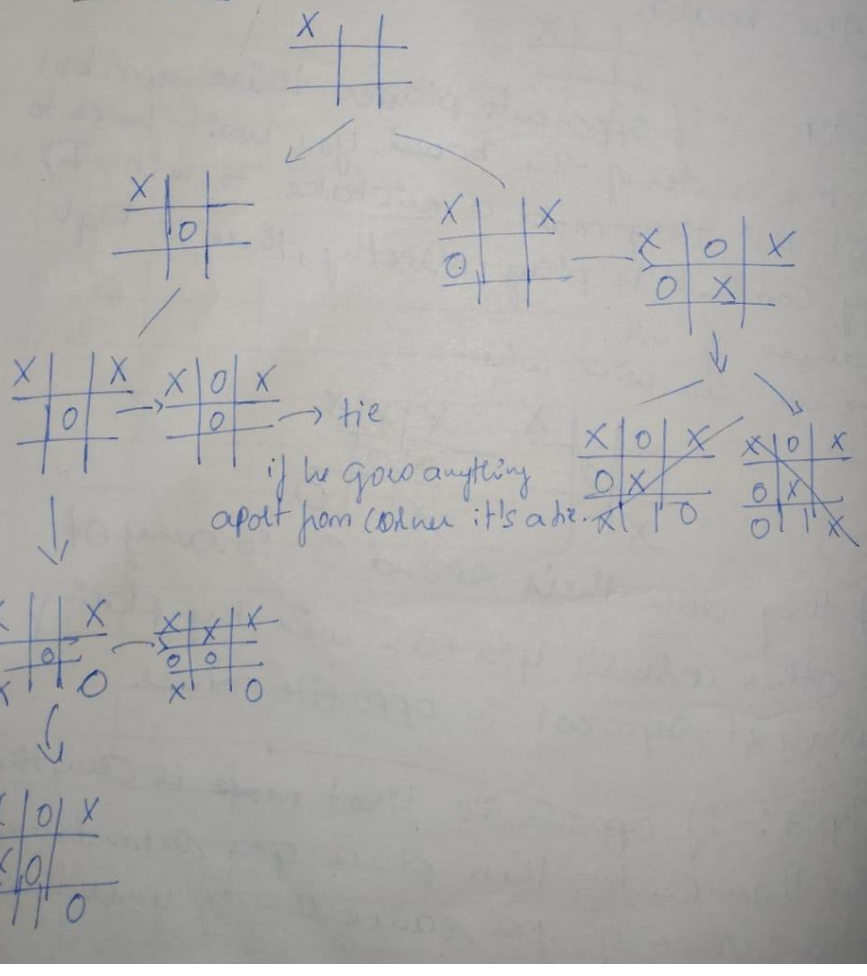
Algorithm:



Step 4:- No matter what is your opponents second move place your third move in the center. If in case he place his second move in the center. you win.



Example



Output:-

who goes first ? Enter 'H' for human or 'A' for AI: h

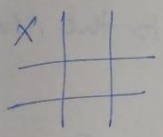
Tic Tac Toe: You (X) vs AI (O). You go first.

Board positions reference:

1	2	3
4	5	6
7	8	9

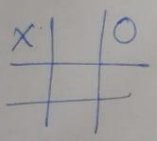
Enter you move (1-9): 1

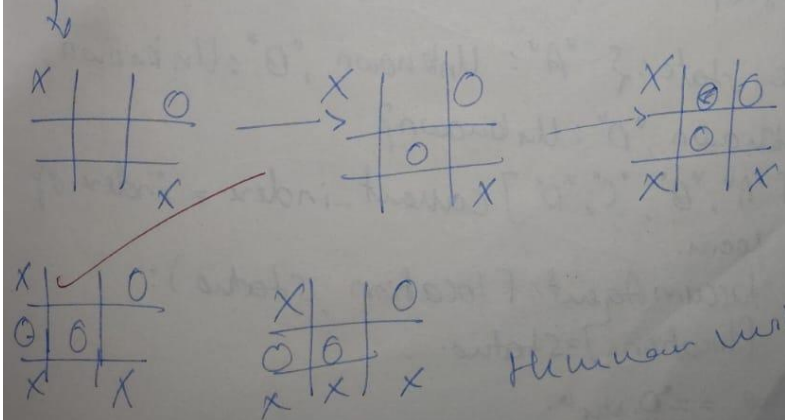
Current board.



AI moves at position 3

Current board.





Human wins

Vaccum cleaner Agent.

Problems statement.

Design an Algorithm where there are 4 Rooms and the vaccum cleaner should clean the room if it's Dirty $2^{*k} = n \times n^{loc} = 4 \times 2^4$.

Step 1: start the agent in any one of the 4 rooms (A, B or C, D)

Step 2: check if the current room is dirty or clean.

Step 3: If the room is Dirty then perform action such (clean the room)

Step 4: If the ROOM is clean move to the next room (or) right.

If in A → B or B → C, C → D, D → A.

Step 5: Repeat 2-4 steps until all rooms are clean.

Step 6: stop the process.

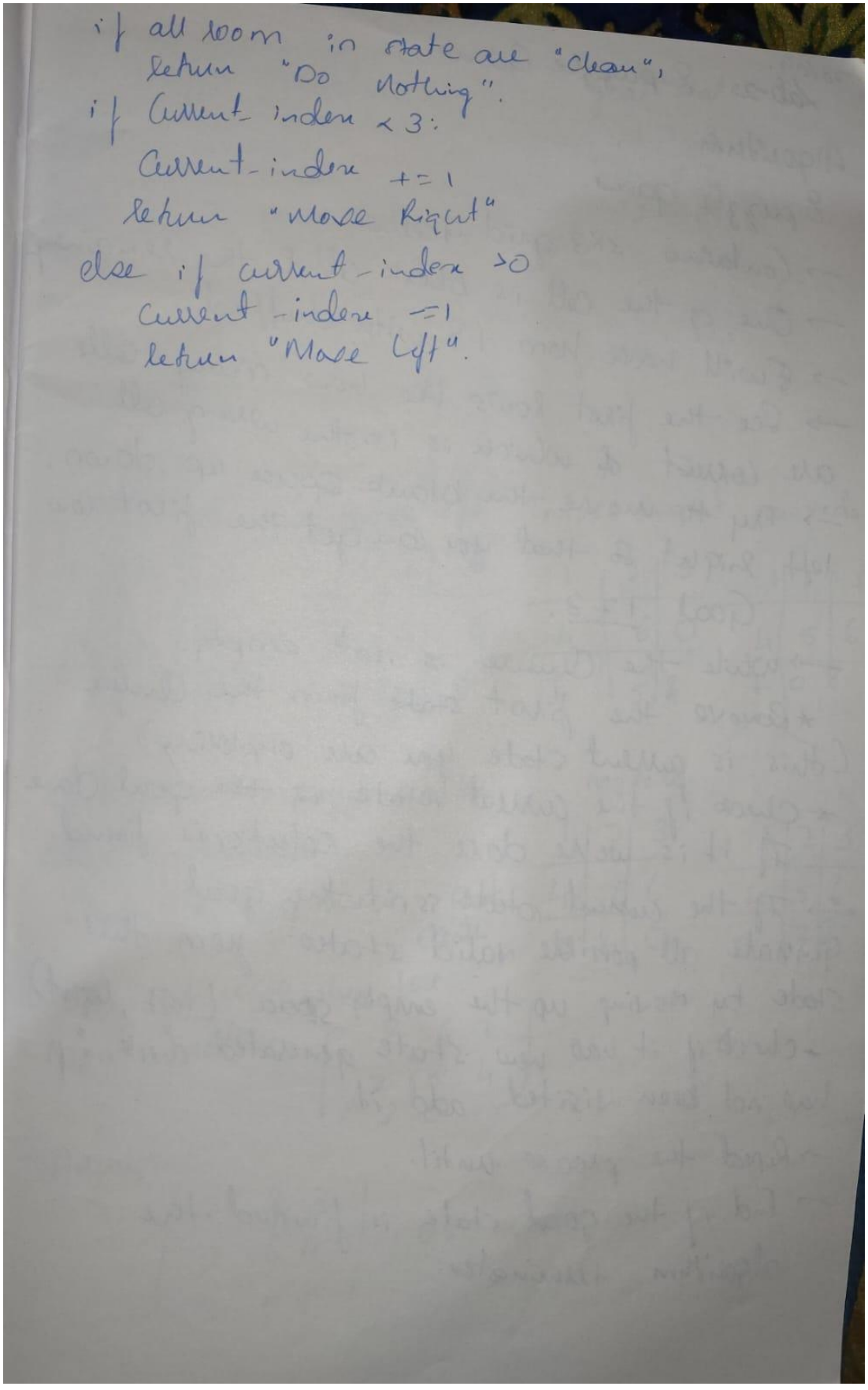
Initialize state: {"A": Unknown, "B": Unknown, "C": Unknown, "D": Unknown}

path = ["A", "B", "C", "D"] current_index = index of starting room.

function VaccumAgent [location, status):

State [location] = status.

if status == "Dirty":



```
if all room in state are "clean",
    return "Do Nothing".
if current_index < 3:
    current_index += 1
    return "Move Right"
else if current_index > 0
    current_index -= 1
    return "Move Left".
```

Code:

```
Tic –Tac –Toe Game
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)


def check_winner(board, player):
    # Check rows, columns and diagonals
    for i in range(3):
        if all([cell == player for cell in board[i]]) or \
           all([board[j][i] == player for j in range(3)]):
            return True

    if all([board[i][i] == player for i in range(3)]) or \
       all([board[i][2 - i] == player for i in range(3)]):
        return True

    return False


def is_full(board):
    return all(cell in ['X', 'O'] for row in board for cell in row)


def get_move(player):
    while True:
        try:
            move = input(f"Player {player}, enter your move (row and column: 1 1): ")
            row, col = map(int, move.split())
            if row in [1, 2, 3] and col in [1, 2, 3]:
                return row - 1, col - 1
            else:
                print("Invalid input. Enter numbers between 1 and 3.")
        except ValueError:
            print("Invalid input. Enter two numbers separated by space.")


def play_game():
    board = [[" " for _ in range(3)] for _ in range(3)]
    current_player = "X"

    while True:
        print_board(board)
        row, col = get_move(current_player)
```

```
        if board[row][col] != " ":
            print("That spot is taken. Try again.")
            continue

        board[row][col] = current_player

        if check_winner(board, current_player):
            print_board(board)
            print(f"Player {current_player} wins!")
            break

        if is_full(board):
            print_board(board)
            print("It's a draw!")
            break

        current_player = "O" if current_player == "X" else "X"


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

Output:

```
  |   |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 1
X |   |
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 1 2
X | O |
---------
  |   |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 1 3
X | O | X
---------
  |   |
---------
  |   |
---------
Player O, enter your move (row and column: 1 1): 2 2
X | O | X
---------
  | O |
---------
  |   |
---------
Player X, enter your move (row and column: 1 1): 3 3
X | O | X
---------
  | O |
---------
  |   | X
---------
Player O, enter your move (row and column: 1 1): 3 1
X | O | X
---------
  | O |
---------
O |   | X
---------
Player X, enter your move (row and column: 1 1): 2 3
X | O | X
---------
  | O | X
---------
O |   | X
---------
Player X wins!
```

Vacuum Cleaner

```
def vacuum_simulation():
    cost = 0

    # Get initial states and location
    state_A = int(input("Enter state of A (0 for clean, 1 for dirty): "))
    state_B = int(input("Enter state of B (0 for clean, 1 for dirty): "))
    location = input("Enter location (A or B): ").upper()

    # Vacuum operation loop
    while True:
        if location == 'A':
            if state_A == 1:
                print("Cleaning A.")
                state_A = 0
                cost += 1
            elif state_B == 1:
                print("Moving vacuum right")
                location = 'B'
                cost += 1
            else:
                print("Turning vacuum off")
                break

        elif location == 'B':
            if state_B == 1:
                print("Cleaning B.")
                state_B = 0
                cost += 1
```

```
        elif state_A == 1:
            print("Moving vacuum left")
            location = 'A'
            cost += 1
        else:
            print("Turning vacuum off")
            break
    print(f"Cost: {cost}")
    print(f"{{'A': {state_A}, 'B': {state_B}}}")


vacuum_simulation()
```

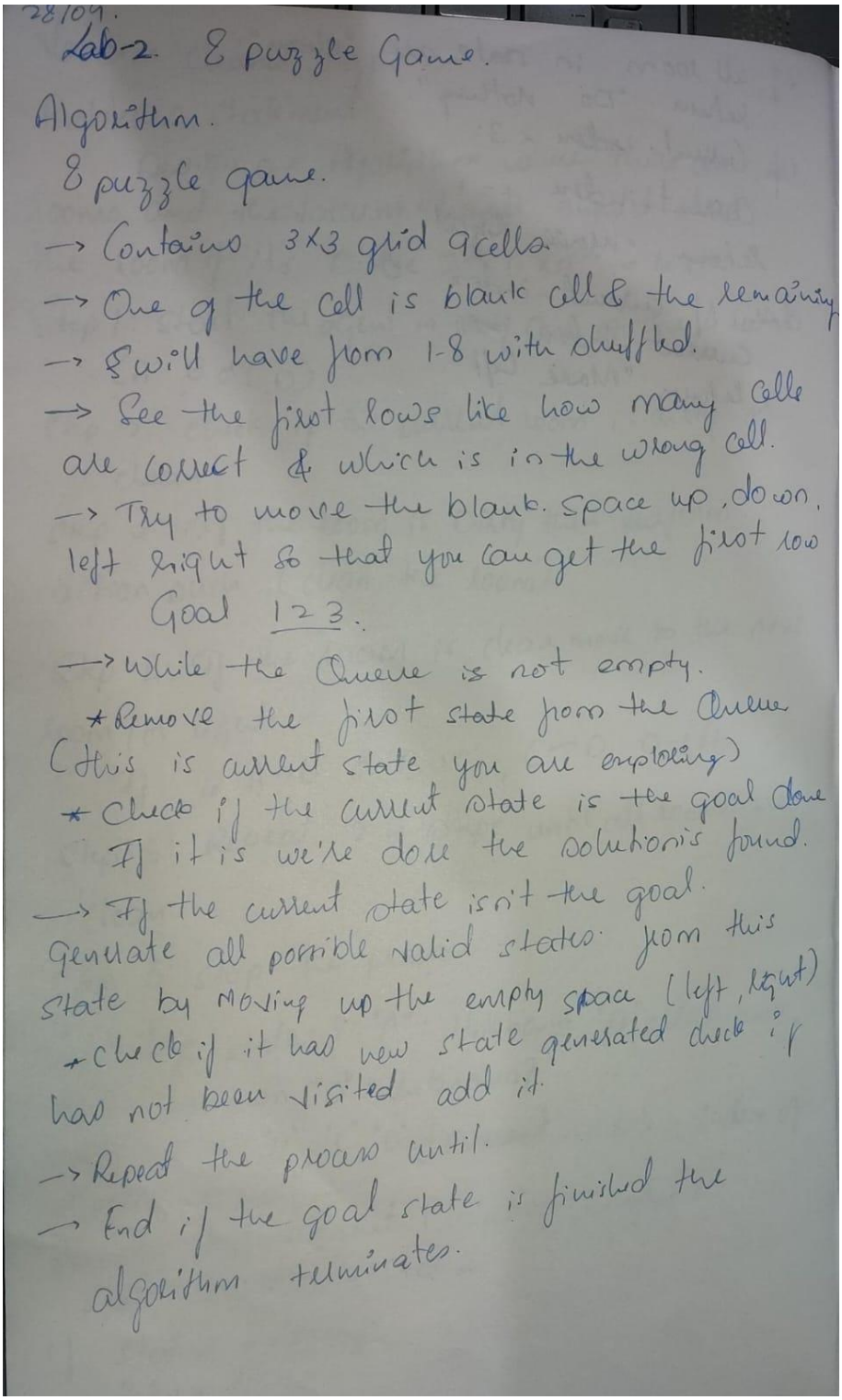
OUTPUT

```
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A
Cleaning A.
Turning vacuum off
Cost: 1
{'A': 0, 'B': 0}
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



28/09.

Lab-2. 8 puzzle Game.

Algorithm.

8 puzzle game.

→ Contains 3×3 grid 9cells.
→ One of the cell is blank cell & the remaining
→ 8 will have from 1-8 with shuffled.
→ See the first rows like how many cells are correct & which is in the wrong cell.
→ Try to move the blank space up, down, left right so that you can get the first row

Goal 1 2 3.

→ While the Queue is not empty.
* Remove the first state from the Queue (this is current state you are exploring)
* Check if the current state is the goal state If it is we're done the solution is found.
→ If the current state isn't the goal. Generate all possible valid states from this state by moving up the empty space (left, right)
* check if it has new state generated check if has not been visited add it.
→ Repeat the process until.
→ End if the goal state is finished the algorithm terminates.

Initial state

1	2	3
4	0	6
7	5	8

Goal state.

1	2	3
4	5	6
7	8	0

1	2	3
4	0	6
7	5	8

move up

1	0	3
4	2	6
7	5	8

move left

1	2	3
0	4	6
7	5	8

move right

1	2	3
4	6	0
7	5	8

move down

1	2	3
4	5	6
7	0	8

move up

1	2	3
4	5	6
7	8	0

1	2	3
4	5	6
0	7	8

Best: When the puzzle can be solved in minimal steps -> 2 no of moves -> Best case

Worst: 150 move when the input matrix. $\begin{bmatrix} 8 & 6 & 7 \\ 2 & 5 & 4 \\ 3 & 0 & 1 \end{bmatrix}$

Average:

$\begin{pmatrix} 2 & 8 & 3 \\ 1 & 6 & 4 \\ 7 & 0 & 5 \end{pmatrix}$ -> 65 moves

11/0

Lab-3.

IDDFS Algorithm.

Iterative Deepening Depth First Search.

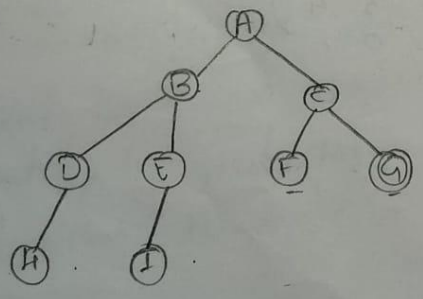
IDDFS combines the benefits of BFS (finds the shallowest/shortest solution) and DFS (low memory usage).

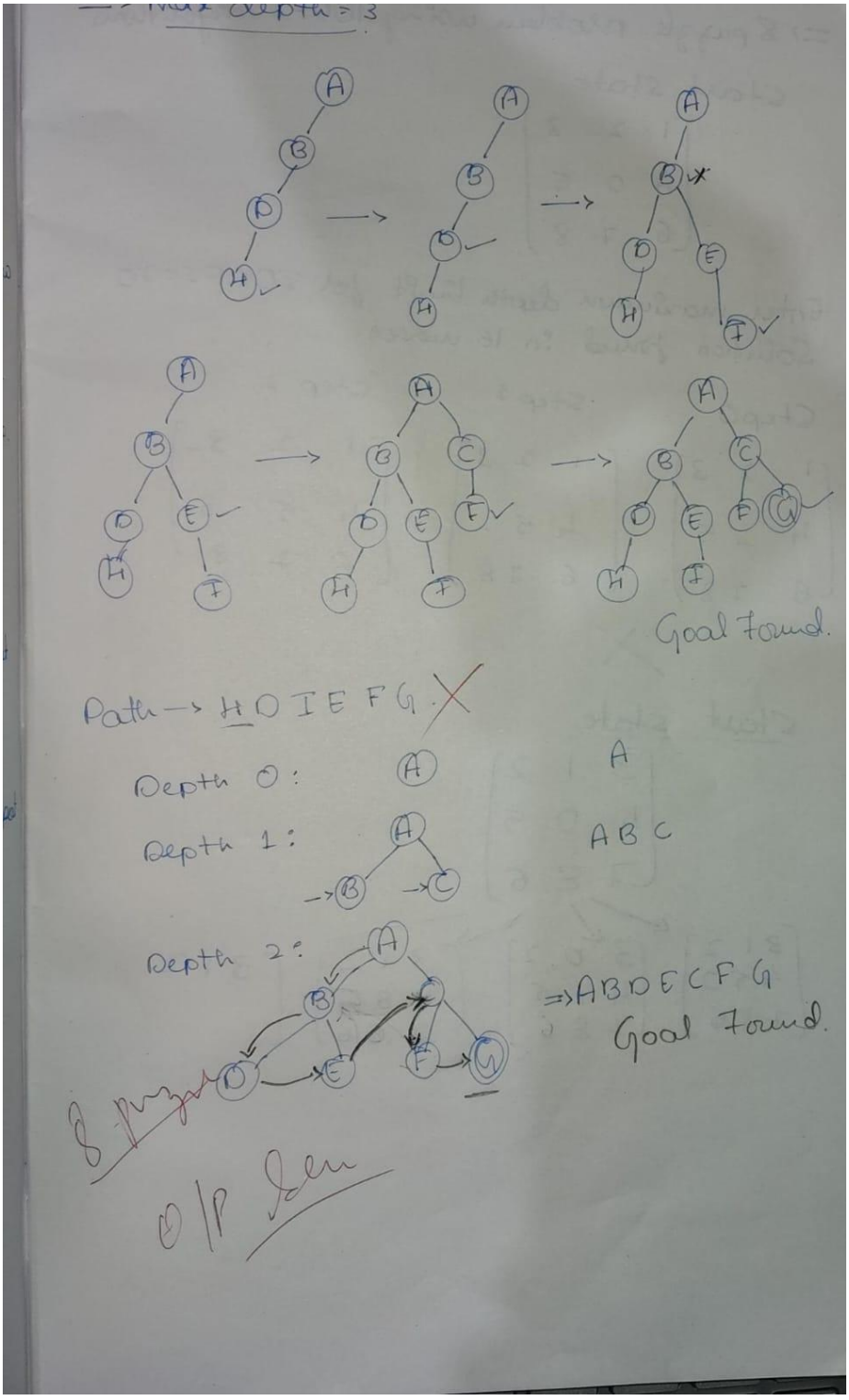
It does this by running DFS up to a depth limit, increasing the depth limit step by step.

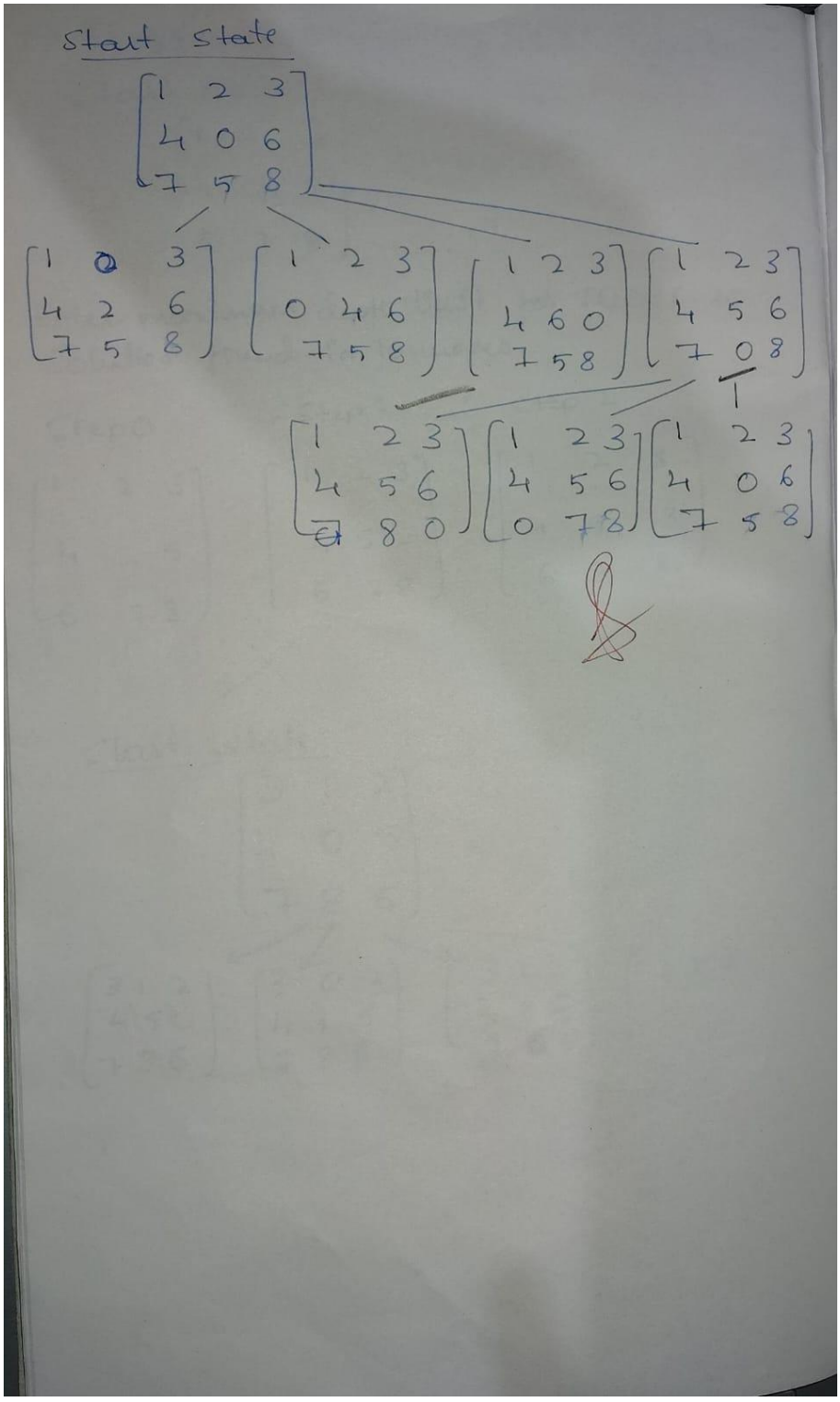
Algorithm Step

1. Start with depth limit = 0.
2. Perform Depth-Limited Search (DLS) up to that depth.
3. If the goal is found → stop.
4. Otherwise, increase the depth limit and repeat.
5. Continue until the goal is found (or all nodes are explored).

Example.




Max depth = 3
Goal found.
Path → H D O I E F G
Depth 0: A
A
Depth 1:
A B C
Depth 2:
⇒ A B D E C F G
Goal found.
8 puzzle
O/P seq


Start State
1 2 3
4 0 6
7 5 8
1 2 3
4 5 6
7 8 0

Code:
Usig DFS 8 puzzel without heuristic

```
# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))

# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def get_neighbors(state):
    # Find the empty tile (0)
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
                break

    neighbors = []
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            # Swap empty tile with adjacent tile
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def dfs_limited(start, depth_limit):
    stack = [(start, [start])]
    visited = set([start])

    while stack:
        current, path = stack.pop()

        if current == goal:
            return path

        if len(path) - 1 >= depth_limit:  # already reached depth limit
            continue

        for neighbor in get_neighbors(current):
            if neighbor not in visited:
                visited.add(neighbor)
                stack.append((neighbor, path + [neighbor]))

    return None
```

```
# Example start state
start = ((2, 8, 3),
         (1, 6, 4),
         (7, 0, 5))

solution_path = dfs_limited(start, depth_limit=5)

if solution_path:
    print(f"Solution found in {len(solution_path) - 1} moves:")
    for state in solution_path:
        for row in state:
            print(row)
        print()
else:
    print("No solution found within 5 moves.")
```

OUTPUT

```
Solution found in 5 moves:
(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)
```

Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS)

```
from copy import deepcopy
GOAL_STATE = [
    [1, 2, 3],
    [4, 5, 0],
    [6, 7, 8]
]

# Possible moves of the blank (0) tile: up, down, left, right
MOVES = [(-1, 0), (1, 0), (0, -1), (0, 1)]

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

def is_goal(state):
    return state == GOAL_STATE

def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)

    for dx, dy in MOVES:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = deepcopy(state)
            # Swap blank with neighbor
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
```

```
            neighbors.append(new_state)
    return neighbors

def dfs(state, depth, limit, path, visited):
    if is_goal(state):
        return path + [state]
    if depth == limit:
        return None

    for neighbor in get_neighbors(state):
        # To avoid cycles, do not revisit states in current path
        if neighbor not in visited:
            result = dfs(neighbor, depth + 1, limit, path + [state], visited + [neighbor])
            if result is not None:
                return result
    return None

def iterative_deepening_search(initial_state, max_depth=50):
    for depth_limit in range(max_depth):
        print(f"Searching with depth limit = {depth_limit}")
        result = dfs(initial_state, 0, depth_limit, [], [initial_state])
        if result is not None:
            return result
    return None

def print_state(state):
    for row in state:
        print(' '.join(str(x) for x in row))
    print()
```

```
if __name__ == "__main__":
    initial_state = [
        [1, 2, 3],
        [4, 0, 5],
        [6, 7, 8]
    ]

    solution = iterative_deepening_search(initial_state)

    if solution:
        print(f"Solution found in {len(solution)-1} moves!")
        for step, state in enumerate(solution):
            print(f"Step {step}:")
            print_state(state)
    else:
        print("No solution found.")
```

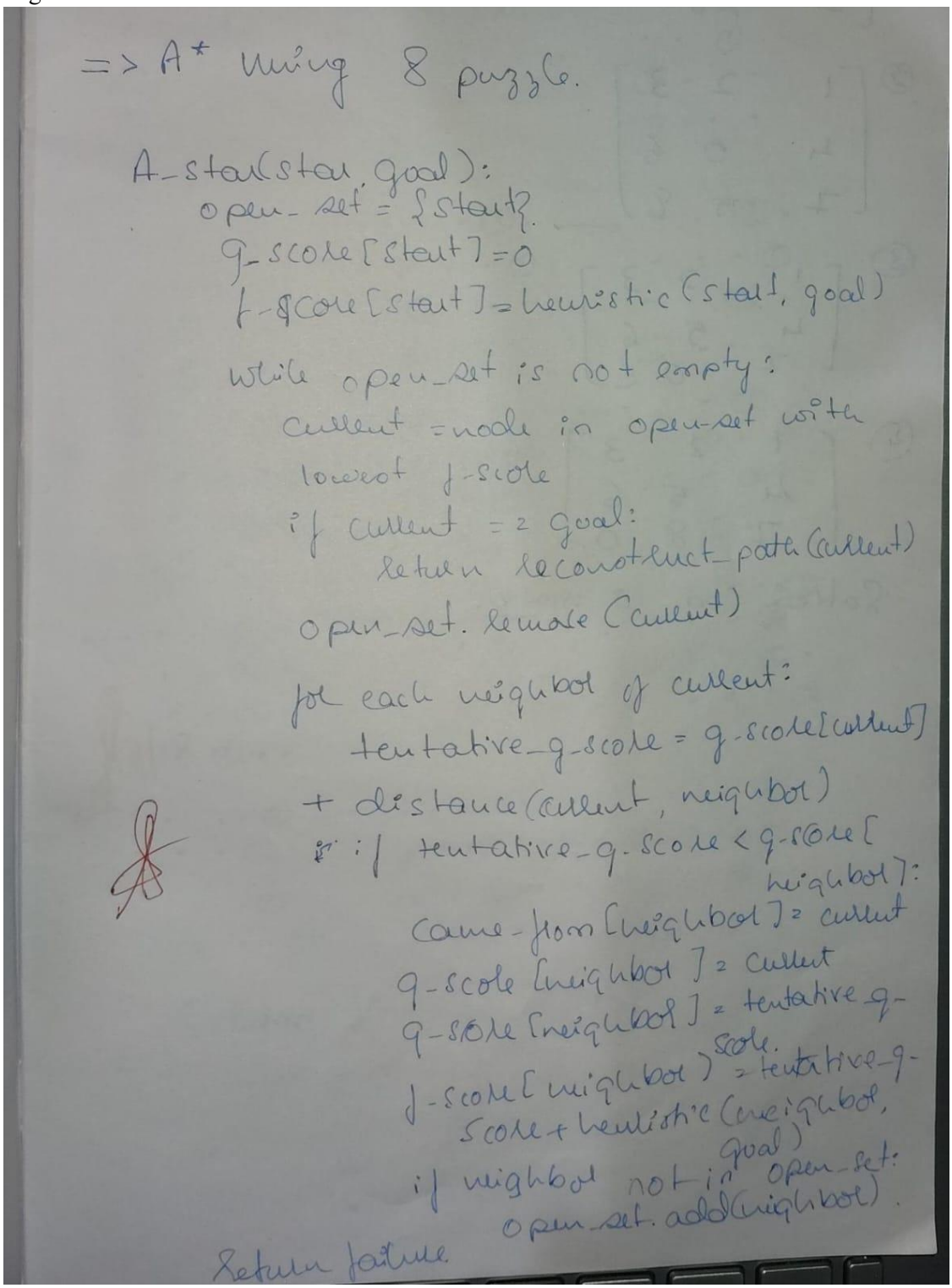
OUTPUT

```
Searching with depth limit = 0
Searching with depth limit = 1
Solution found in 1 moves!
Step 0:
1 2 3
4 0 5
6 7 8

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

Program 3

Implement A* search algorithm

Algorithm:



```
=> A* using 8 puzzle.

A_star(start, goal):
    open_set = {start}
    g_score[start] = 0
    f_score[start] = heuristic(start, goal)
    while open_set is not empty:
        current = node in open_set with
        lowest f_score
        if current = = goal:
            return reconstruct_path(current)
        open_set.remove(current)
        for each neighbor of current:
            tentative_g_score = g_score[current]
            + distance(current, neighbor)
            if tentative_g_score < g_score[
                                neighbor]:
                came_from[neighbor] = current
                g_score[neighbor] = current
                g_score[neighbor] = tentative_g_
                f_score[neighbor) = tentative_g_
                    score + heuristic(neighbor,
                    goal)
                if neighbor not in open_set:
                    open_set.add(neighbor).
Return failure
```

Output:-

Solution path.

Initial $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 7 & 5 & 8 \end{bmatrix}$

(1) $\begin{bmatrix} 1 & 2 & 3 \\ 0 & 4 & 6 \\ 7 & 5 & 8 \end{bmatrix}$

(2) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 0 & 6 \\ 7 & 5 & 8 \end{bmatrix}$

(3) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 0 & 8 \end{bmatrix}$

(4) $\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$

Solved in 3 moves.

Code:
Misplace Tiles

```
import heapq


# Goal state
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))


# Moves: Up, Down, Left, Right
moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]


# Heuristic: Misplaced tiles
def misplaced_tiles(state):
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal[i][j]:
                count += 1
    return count


# Find blank position (0)
def find_blank(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
# Generate neighbors
def get_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
```

```
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state)
    return neighbors
# A* Search
def astar(start):
    pq = []
    heapq.heappush(pq, (misplaced_tiles(start), 0, start, []))
    visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + misplaced_tiles(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None

# Example usage
```

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

solution = astar(start_state)

# Print solution path
for step in solution:
    for row in step:
        print(row)
    print("------")
```

OUTPUT

```
(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)
------
```

Manhatten:

```
import heapq
goal = ((1, 2, 3),
        (8, 0, 4),
        (7, 6, 5))


moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right
def manhattan_distance(state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                # goal position of this tile
                goal_x = (value - 1) // 3
                goal_y = (value - 1) % 3
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance
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)
    for dx, dy in moves:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
```

```
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors
def astar(start):
    pq = []
    heapq.heappush(pq, (manhattan_distance(start), 0, start, []))  # (f, g, state, path)
    visited = set()
    while pq:
        f, g, state, path = heapq.heappop(pq)
        if state == goal:
            return path + [state]
        if state in visited:
            continue
        visited.add(state)
        for neighbor in get_neighbors(state):
            if neighbor not in visited:
                new_g = g + 1
                new_f = new_g + manhattan_distance(neighbor)
                heapq.heappush(pq, (new_f, new_g, neighbor, path + [state]))

    return None
start_state = ((2, 8, 3),
               (1, 6, 4),
               (7, 0, 5))
solution = astar(start_state)

if solution is None:
    print("No solution found.")
else:
    print("Solution path:")
```

```
    for step in solution:
        for row in step:
            print(row)
        print("------")
```

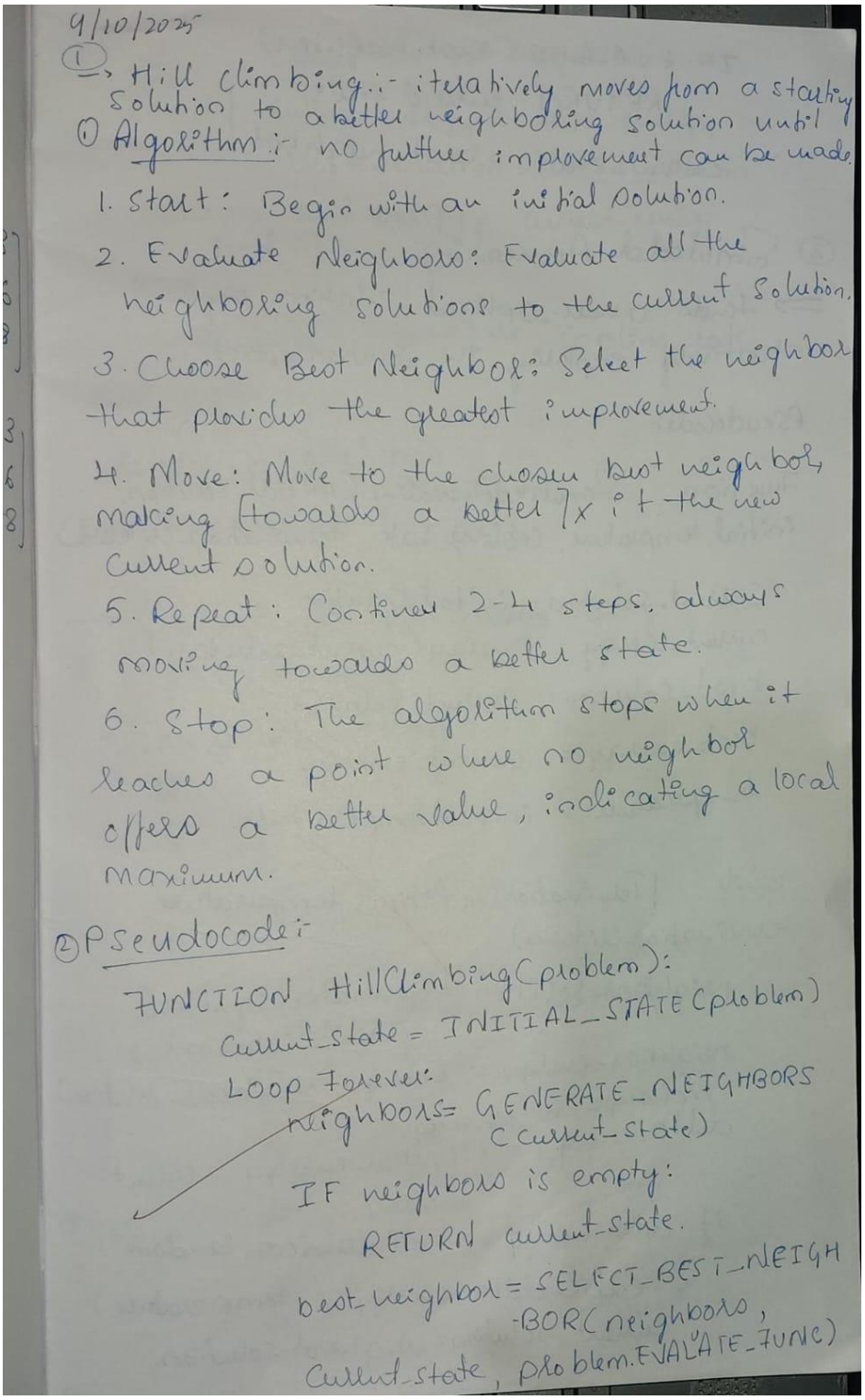
OUTPUT:

```
Solution path:
(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:



9/10/2025

(1) → Hill climbing :- iteratively moves from a starting solution to a better neighboring solution until

① Algorithm :- no further improvement can be made.

1. Start : Begin with an initial solution.
2. Evaluate Neighbors : Evaluate all the neighboring solutions to the current solution.
3. Choose Best Neighbor : Select the neighbor that provides the greatest improvement.
4. Move : Move to the chosen best neighbor, making towards a better / x ; t the new current solution.
5. Repeat : Continue 2-4 steps, always moving towards a better state.
6. Stop : The algorithm stops when it reaches a point where no neighbor offers a better value, indicating a local maximum.

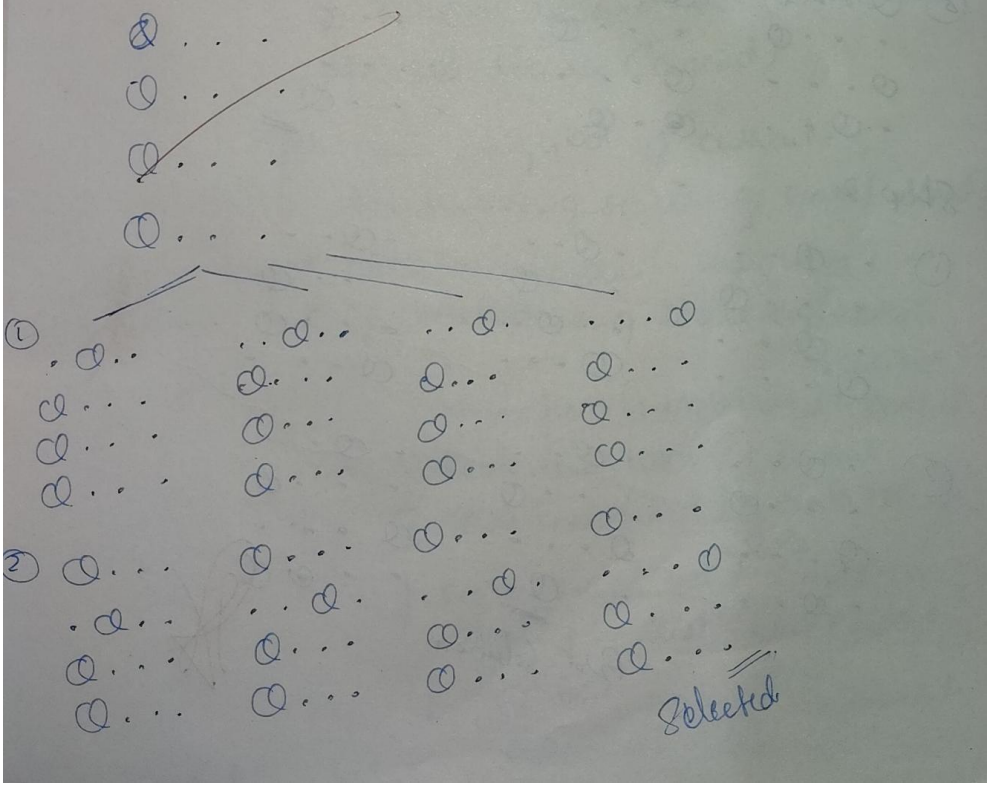
② Pseudocode :-

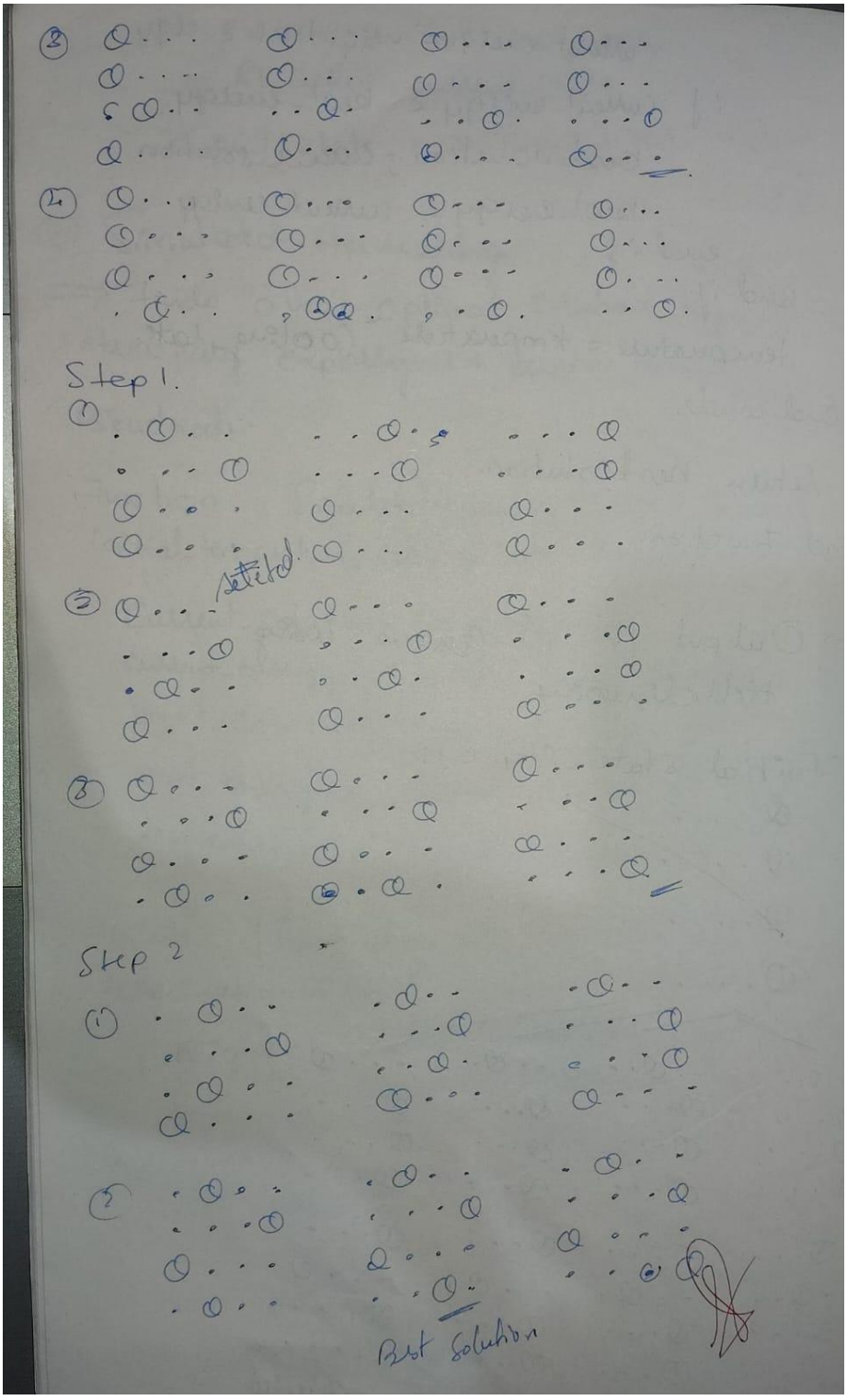
```
FUNCTION HillClimbing(problem):
    current_state = INITIAL_STATE(problem)
    LOOP forever:
        neighbors = GENERATE_NEIGHBORS
                    (current_state)
        IF neighbors is empty:
            RETURN current_state.
        best_neighbor = SELECT_BEST_NEIGH
                        -BOR(neighbors,
        current_state, problem.EVALUATE_FUNC)
```

IF EVALAUTE (best_neighbor)
RETURN current_state
current_state = best_neighbor.

→ Output for n-Queens using Hill climbing

Initial state (step 0):




Step 1.
selected.
Step 2
Best solution

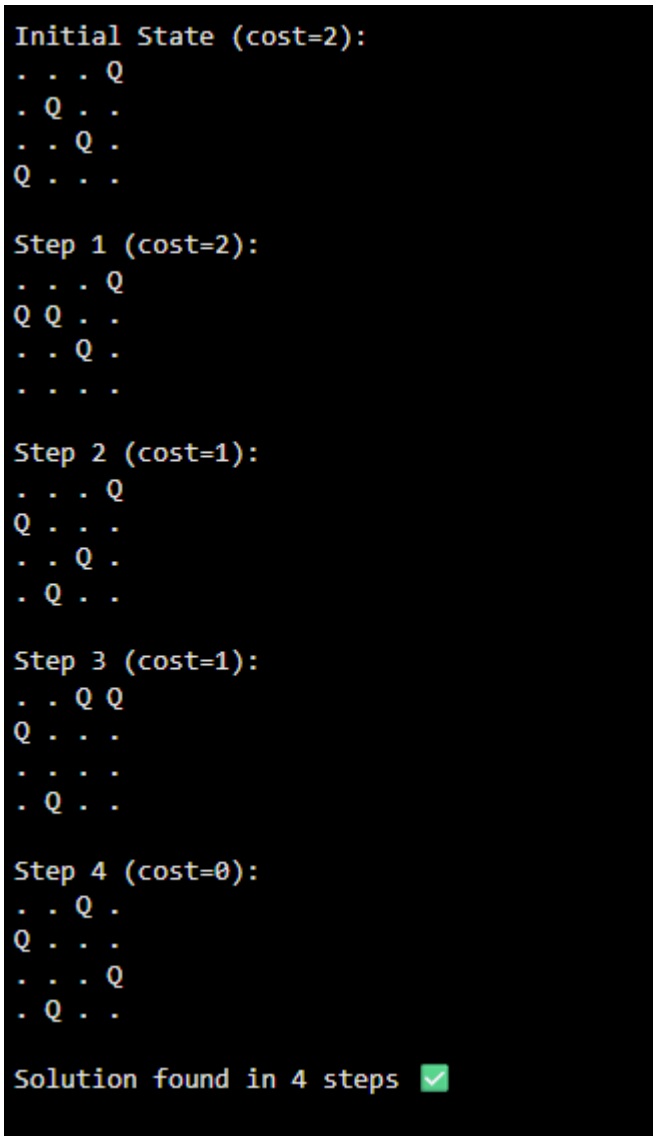
Code:

```
import random


def compute_cost(state):
    n = len(state)
    cost = 0
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j]:              # same row
                cost += 1
            elif abs(state[i] - state[j]) == abs(i - j):  # same diagonal
                cost += 1
    return cost



def get_neighbors(state):
    neighbors = []
    n = len(state)
    for col in range(n):        # pick a column
        for row in range(n):    # try moving queen in this column to another row
            if row != state[col]:
                new_state = state.copy()
                new_state[col] = row
                neighbors.append(new_state)
    return neighbors




def print_board(state):
    n = len(state)
```

```
    for r in range(n):
        line = ""
        for c in range(n):
            line += "Q " if state[c] == r else ". "
        print(line)
    print("")


def hill_climb(initial_state, max_sideways=50):
    current = initial_state
    current_cost = compute_cost(current)
    steps = 0
    sideways_moves = 0

    print("Initial State (cost={}):".format(current_cost))
    print_board(current)

    while True:
        neighbors = get_neighbors(current)
        costs = [compute_cost(n) for n in neighbors]
        min_cost = min(costs)

        if min_cost > current_cost:
            # no better neighbor -> stop
            break

        # pick one of the best neighbors randomly
        best_neighbors = [n for n, c in zip(neighbors, costs) if c == min_cost]
        next_state = random.choice(best_neighbors)
        next_cost = compute_cost(next_state)
```

```
        # handle sideways moves
        if next_cost == current_cost:
            if sideways_moves >= max_sideways:
                break
            else:
                sideways_moves += 1
        else:
            sideways_moves = 0

        current = next_state
        current_cost = next_cost
        steps += 1

        print("Step {} (cost={}):".format(steps, current_cost))
        print_board(current)

        if current_cost == 0:
            print("Solution found in {} steps ☑".format(steps))
            return current

    print("Local minimum reached (cost={}) ✖".format(current_cost))
    return current

initial_state = [3, 1, 2, 0]

final = hill_climb(initial_state, max_sideways=10)
```

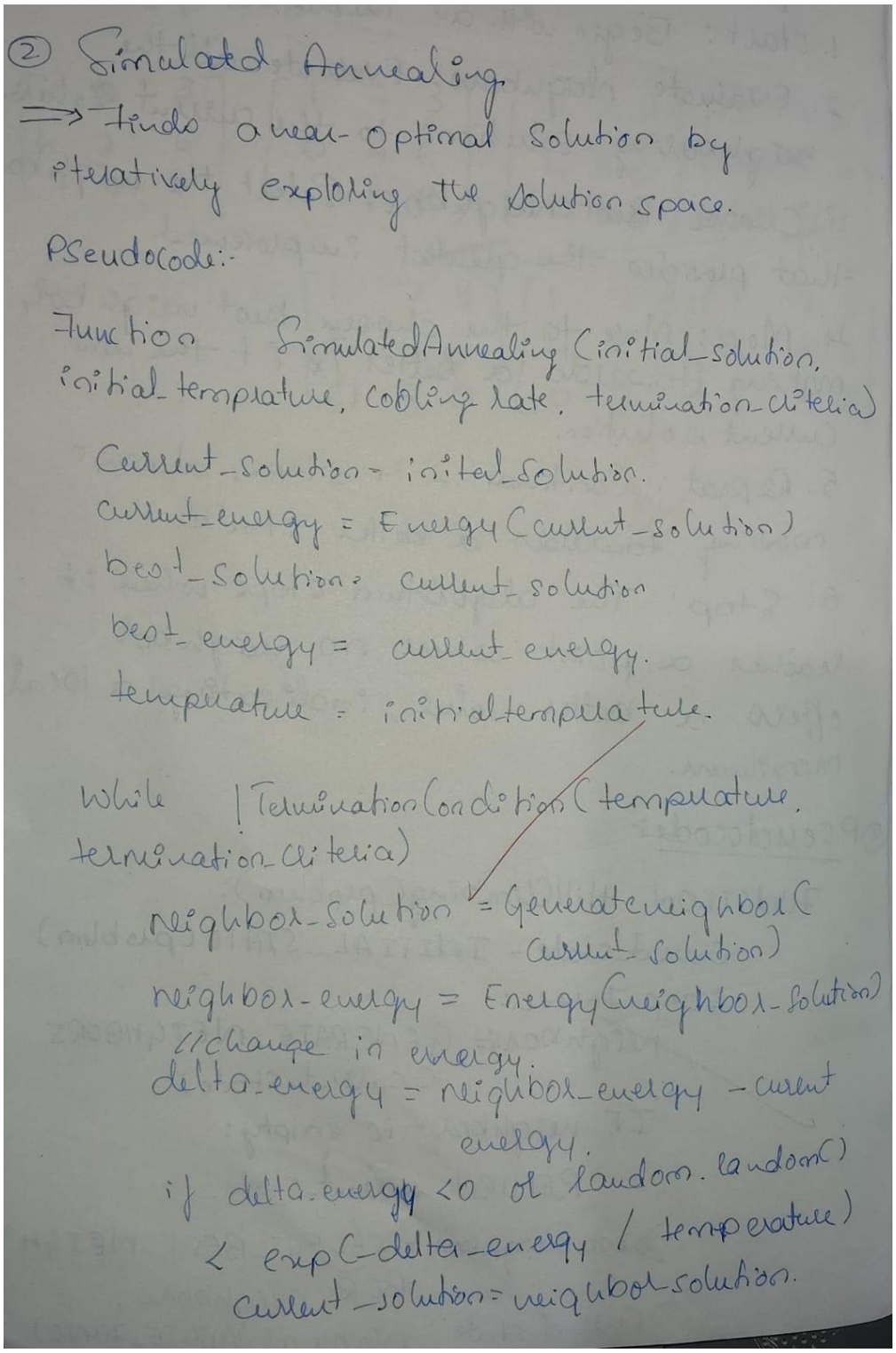
OUTPUT:


Initial State (cost=2):
. . . Q
. Q . .
. . Q .
Q . . .
Step 1 (cost=2):
. . . Q
Q Q . .
. . Q .
. . . .
Step 2 (cost=1):
. . . Q
Q . . .
. . Q .
. Q . .
Step 3 (cost=1):
. . Q Q
Q . . .
. . . .
. Q . .
Step 4 (cost=0):
. . Q .
Q . . .
. . . Q
. Q . .
Solution found in 4 steps ✅

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:

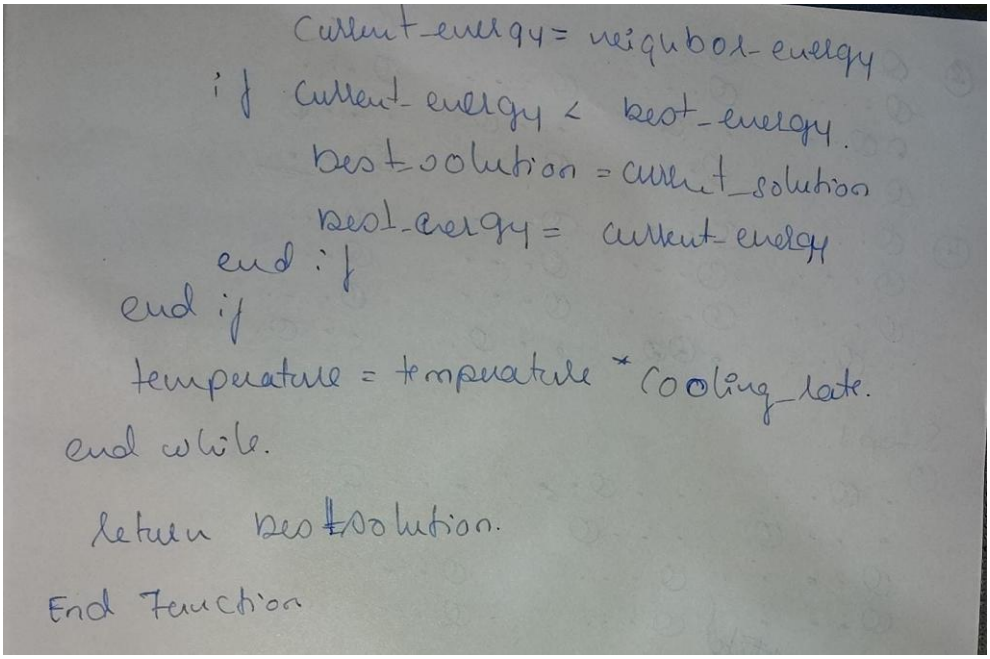


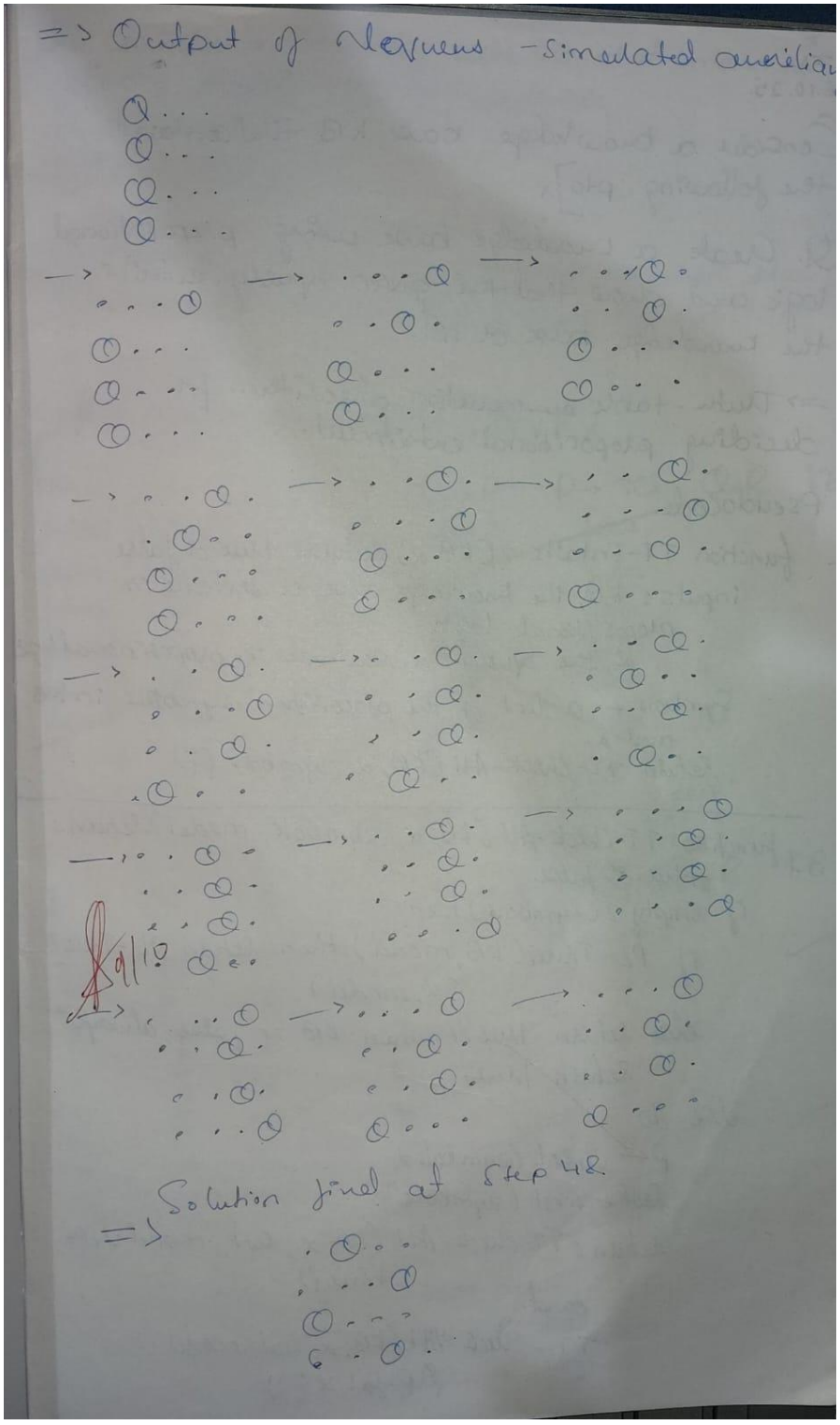
② Simulated Annealing

⟹ Finds a near-optimal solution by iteratively exploring the solution space.

Pseudocode:-

```
Function SimulatedAnnealing (initial_solution,
initial_temperature, cooling_rate, termination_criteria)
    Current_solution = initial_solution.
    current_energy = Energy(current_solution)
    best_solution = current_solution
    best_energy = current_energy.
    temperature = initial_temperature.

While !TerminationCondition(temperature,
termination_criteria)
    neighbor_solution = Generateneighbor(
                        current_solution)
    neighbor_energy = Energy(neighbor_solution)
    //change in energy.
    delta_energy = neighbor_energy - current
                   energy.
    if delta_energy < 0 or random.random()
       < exp(-delta_energy / temperature)
       current_solution = neighbor_solution.
```


current_energy = neighbor_energy
if current_energy < best_energy.
best_solution = current_solution
best_energy = current_energy
end if
end if
temperature = temperature * cooling_rate.
end while.
return best_solution.
End Function


=> Output of Nqueens - Simulated annealing
9/10
=> Solution found at step 48.

Code:

```
from scipy.optimize import dual_annealing
import numpy as np

def queens_max(x):
    cols = np.round(x).astype(int)
    n = len(cols)

    if len(set(cols)) < n:
        return 1e6
    attacks = 0
    for i in range(n):
        for j in range(i + 1, n):
            if abs(i - j) == abs(cols[i] - cols[j]):
                attacks += 1
    return attacks
n = 8
bounds = [(0, n - 1)] * n
result = dual_annealing(queens_max, bounds)

best_cols = np.round(result.x).astype(int).tolist()
not_attacking = n

print(f"The best position found is: {best_cols}")
print(f"The number of queens that are not attacking each other is: {not_attacking}")
```

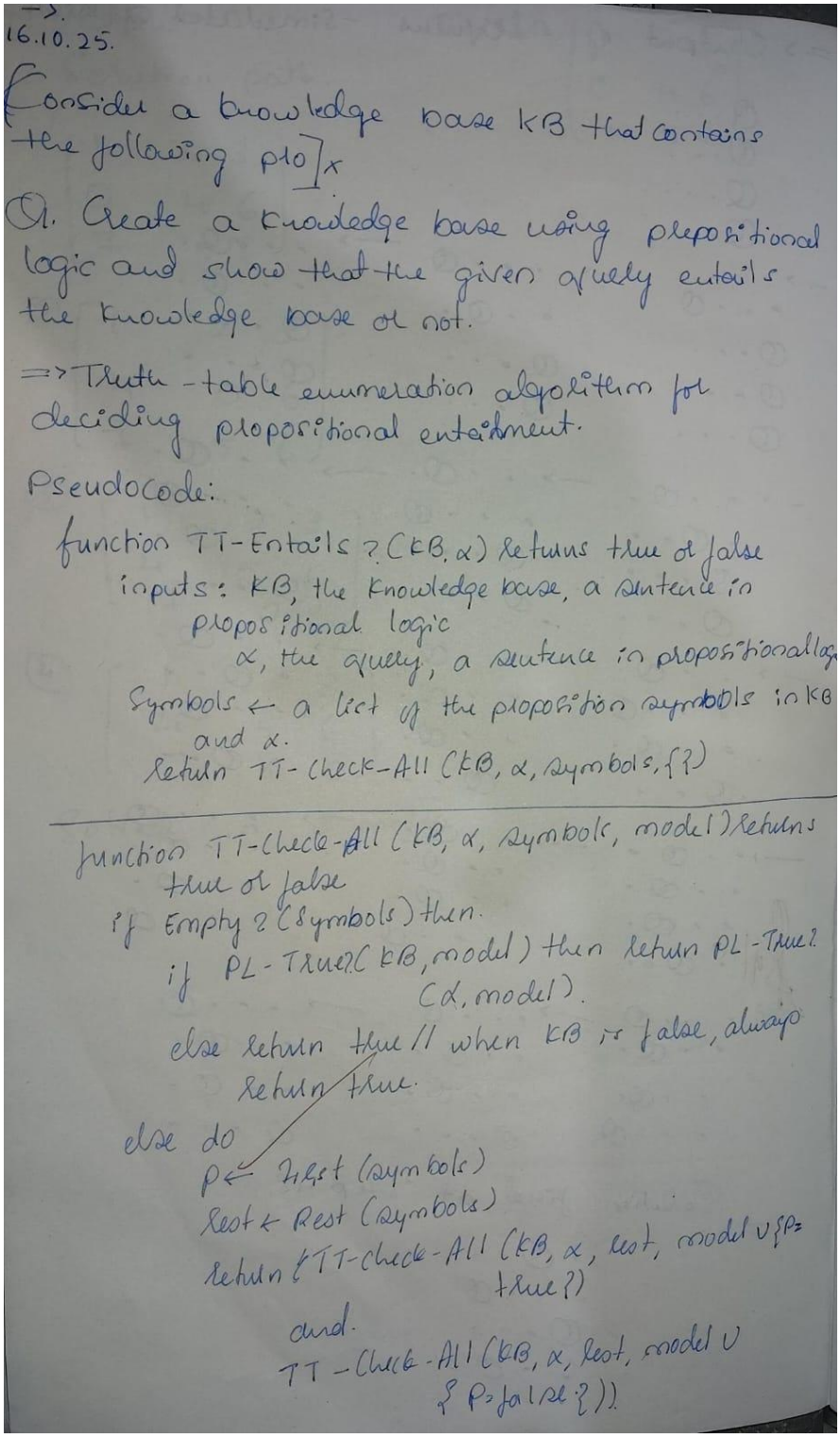
OUTPUT:

```
The best position found is: [7, 4, 6, 1, 3, 5, 0, 2]
The number of queens that are not attacking each other is: 8
```

Program 6

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

Algorithm:



16.10.25.

Consider a knowledge base KB that contains the following p10]x

Q1. Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

=> Truth-table enumeration algorithm for deciding propositional entailment.

Pseudocode:

function TT-Entails ? (KB, α) returns true or false
inputs: KB, the knowledge base, a sentence in propositional logic
α, the query, a sentence in propositionallog
Symbols ← a list of the proposition symbols in KB and α.
return TT-check-All (KB, α, symbols, {?)

---

function TT-Check-All (KB, α, symbols, model) returns true or false
if Empty ? (Symbols) then.
if PL-True?(KB, model) then return PL-True? (α, model).
else return true // when KB is false, always return true.
else do
P ← First (symbols)
Rest ← Rest (symbols)
return (TT-check-All (KB, α, rest, model ∪ {P= true?)
and.
TT-Check-All (KB, α, rest, model ∪ {P= false?}))

Q2. Consider a knowledge base KB that contains the following propositional logic sentences:

$\{Q \to P,\ P \to \neg Q,\ Q \vee R\}$

i) Construct a truth table that shows the truth value of each sentence in KB and indicate the models in which the KB is true.

$KB = (Q \to P,\ P \to \neg Q,\ Q \vee R)$.

P	Q	R	$Q \to P$	$P \to \neg Q$	$Q \vee R$	KB
false	false	false	true	false		
false	false	true	true	false		
false	true	false	false	true		
false	true	true	false	true		
true	false	false	false	true		
true	false	true	false	true		
true	true	false	true	false		
true	true	true	true	false		

P	Q	R	$Q \to P$	$P \to \neg Q$	$Q \vee R$	KB
F	F	F	T	T	F	
F	F	T	T	T	T	✓
F	T	F	F	T	T	
F	T	T	F	T	T	
T	F	F	T	T	F	
T	F	T	T	T	T	✓
T	T	F	T	F	T	
T	T	T	T	F	T	

=> KB is true when the conditions are

P = false Q = false R = True.

P = True Q = false R = true.

(ii) Does KB entail R?

$KB = \{Q \to P,\ P \to \neg Q,\ Q \vee R\}$.

$\alpha = R$.

KB is true when $P=F,\ Q=F,\ R=T$.

and $P=T,\ Q=F,\ R=T$.

in both case R = True.

∴ KB entails R

(iii) Does KB entail $R \to P$?

$KB = \{Q \to P,\ P \to \neg Q,\ Q \vee R\}$.

$\alpha = R \to P$. ?

KB is true when

	$R \to P$	KB
$P=T,\ Q=F,\ R=T$.	T	✓
$P=F,\ Q=F,\ R=T$.	F.	

∴ KB does not entail $R \to P$.

(iv) Does KB entail $Q \to R$?

$KB = \{Q \to P,\ P \to \neg Q,\ Q \vee R\}$.

$\alpha = Q \to R$.

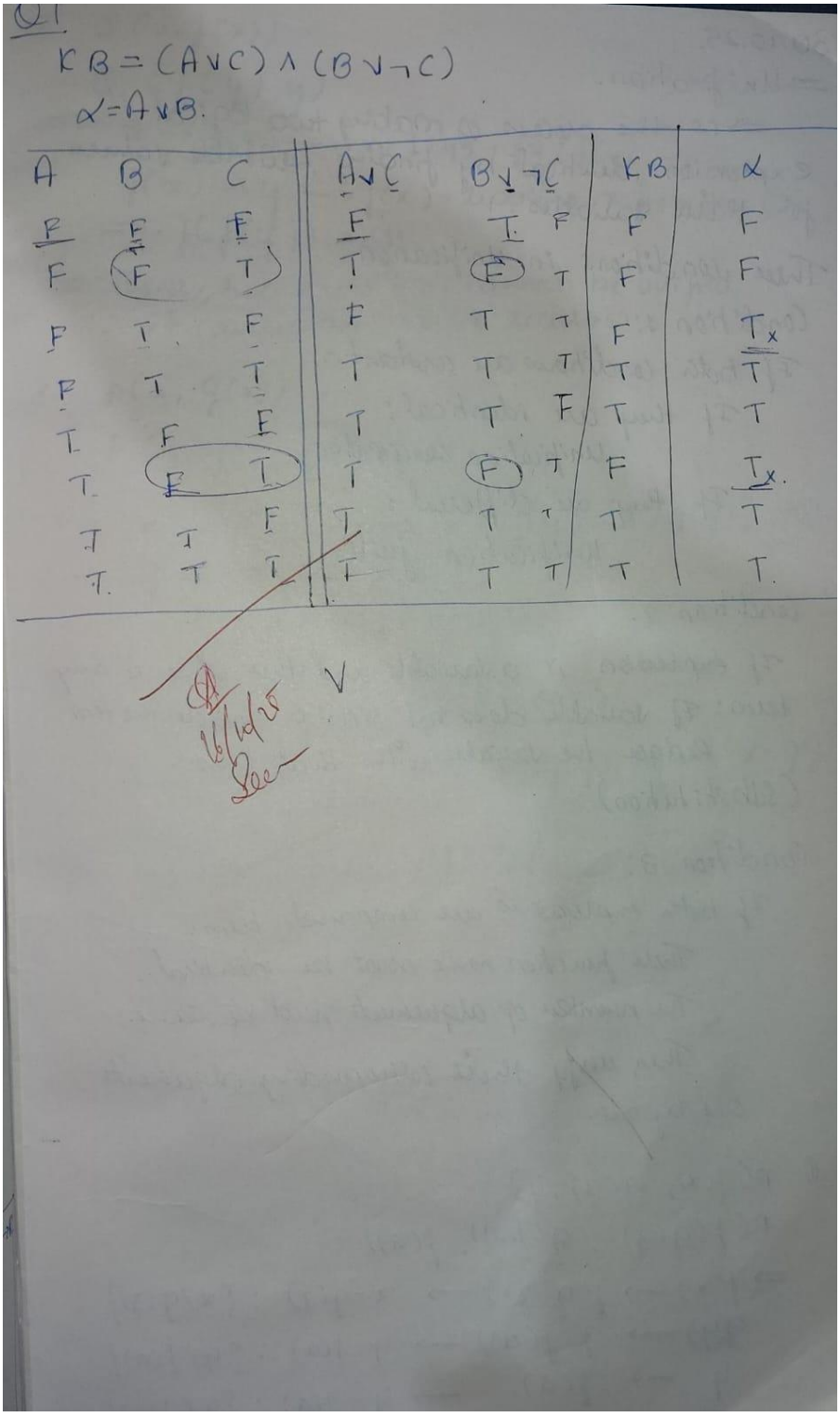
KB is true when

	$Q \to R$
$P=T\ Q=F.R=T$.	T
$P=F\ Q=F\ R=T$.	T.

In both condition $Q \to R$ is T.

∴ KB entails $Q \to R$

P	Q	R	KB	KB ⇒ R	KB ⇒ R → P	KB ⇒ Q → R
F	F	F	F			
F	F	T	T	True	False.	True
F	T	F	F			
F	T	T	F			
T	F	F	F	True	True.	True
T	F	T	T			
T	T	F	F			
T	T	T	R			


Q1
KB = (A ∨ C) ∧ (B ∨ ¬C)
α = A ∨ B.
A B C A∨C B∨¬C KB α
F F F F T F F F
F F T T F F F
F T F F T F T x
F T T T T T T
T F F T T T T
T F T T F F T x
T T F T T T T
T T T T T T T
16/4/25
Lec
V

Code:

```
import itertools
from sympy import symbols, sympify

A, B, C = symbols('A B C')

alpha_input = input("Enter alpha (example: A | B): ")
kb_input = input("Enter KB (example: (A | C) & (B | ~C)): ")

alpha = sympify(alpha_input, evaluate=False)
kb = sympify(kb_input, evaluate=False)

GREEN = "\033[92m"
RESET = "\033[0m"

print(f"\nTruth Table for α = {alpha_input}, KB = {kb_input}\n")
print(f"{'A':<6}{'B':<6}{'C':<6}{'α':<10}{'KB':<10}")

entailed = True

for values in itertools.product([False, True], repeat=3):
    subs = {A: values[0], B: values[1], C: values[2]}
    alpha_val = alpha.subs(subs)
    kb_val = kb.subs(subs)

    alpha_str = f"{GREEN}{alpha_val}{RESET}" if kb_val else str(alpha_val)
    kb_str = f"{GREEN}{kb_val}{RESET}" if kb_val else str(kb_val)

    print(f"{str(values[0]):<6}{str(values[1]):<6}{str(values[2]):<6}"
          f"{alpha_str:<10}{kb_str:<10}")
```

```
    if kb_val and not alpha_val:
        entailed = False

if entailed:
    print(f"\n KB |= α holds (KB entails α)\n")
else:
    print(f"\n KB does NOT entail α\n")
```

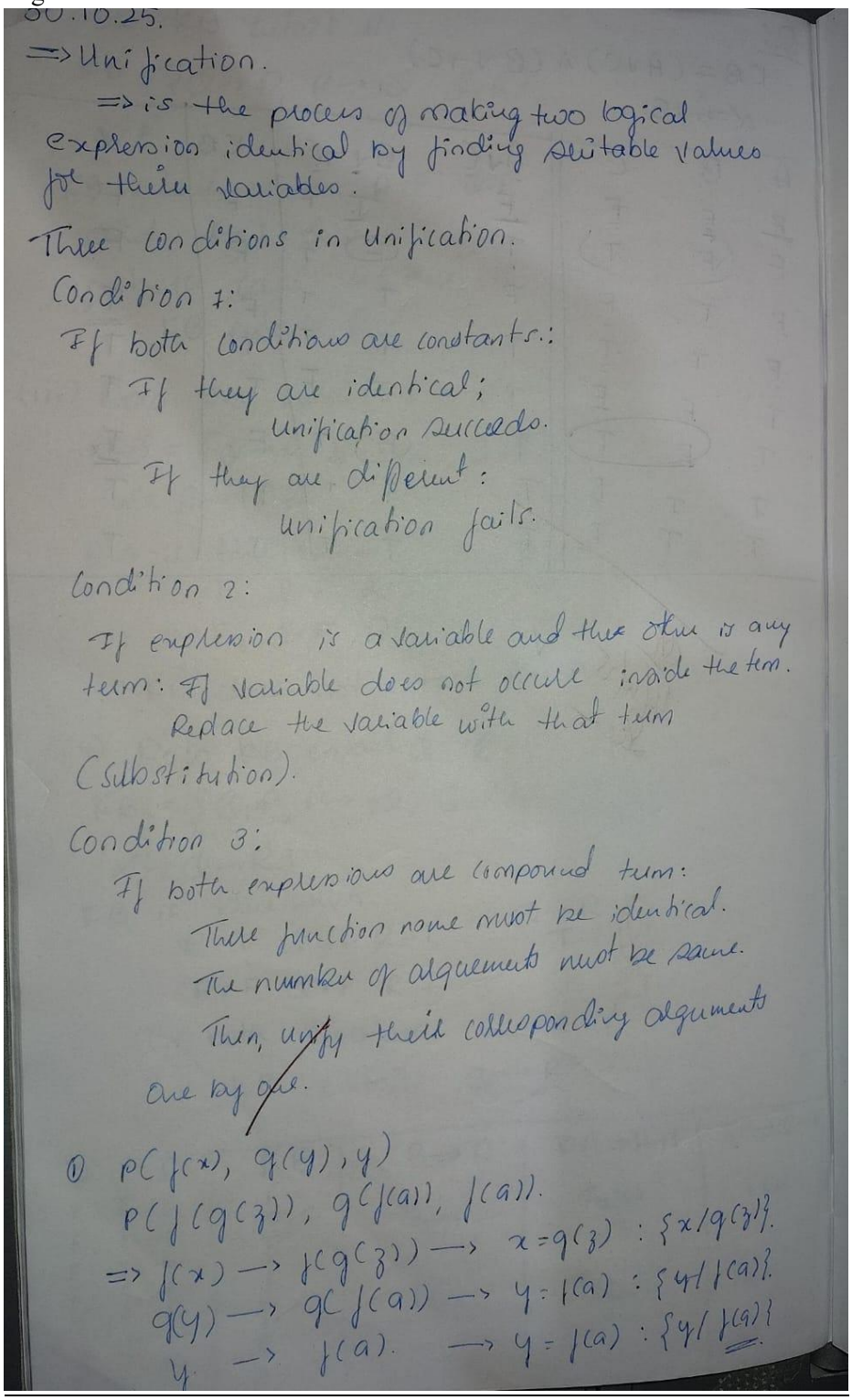
OUTPUT:

```
Enter alpha (example: A | B): A|B
Enter KB (example: (A | C) & (B | ~C)): (A | C) & (B | ~C)

Truth Table for α = A|B, KB = (A | C) & (B | ~C)

A     B     C     α         KB
False False False False     False
False False True  False     False
False True  False True      False
False True  True  TrueTrue
True  False False TrueTrue
True  False True  True      False
True  True  False TrueTrue
True  True  True  TrueTrue

 KB |= α holds (KB entails α)
```

Program 7

Implement unification in first order logic

Algorithm:



30.10.25.

=> Unification.

=> is the process of making two logical expression identical by finding suitable values for their variables.

Three conditions in Unification.

Condition 1:

If both conditions are constants.:

If they are identical;

Unification succeeds.

If they are different:

unification fails.

Condition 2:

If expression is a variable and the other is any term: If variable does not occure inside the term.

Replace the variable with that term

(Substitution).

Condition 3:

If both expressions are compound term:

The function name must be identical.

The number of arguments must be same.

Then, unify their corresponding arguments one by one.

① $P(f(x), g(y), y)$

$P(f(g(z)), g(f(a)), f(a))$.

=> $f(x) \rightarrow f(g(z)) \rightarrow x = g(z) : \{x/g(z)\}$.

$g(y) \rightarrow g(f(a)) \rightarrow y = f(a) : \{y/f(a)\}$.

$y \rightarrow f(a) \rightarrow y = f(a) : \{y/f(a)\}$

2. $Q(x, f(x))$

$Q(f(y), y)$

$\Rightarrow x \to f(y) \longrightarrow x = f(y) : \{x / f(y)\}$. — ①

$f(x) \to y. \to f(x) =$ replacing x from the ①

$\Rightarrow f(f(y)) = y$

= These two expressions cannot be unified.

∵ it causes an infinite recursion.

3. $P(x, g(x))$

$P(g(y), g(g(z)))$

$\Rightarrow x \to g(y) \longrightarrow x = g(y) : \{x / g(y)\}$. — ①

$g(x) \to g(g(z))$

$\hookrightarrow x = g(z)$. — ②.

From 1 and 2:

$x = g(y) = g(z) \; ; y = z$.

Substitute $y = z$ into $x = g(y)$ to get $x = g(z)$*

MGU

$\Rightarrow \{x \to g(z), y \mapsto z\}$.

30/10/24

Code:

```
def occurs_check(var, expr):
    if var == expr:
        return True
    if isinstance(expr, tuple):
        return any(occurs_check(var, sub) for sub in expr[1:])  # Skip function symbol
    return False
def substitute(expr, subst):
    if isinstance(expr, str):
        # Follow substitution chain until fully resolved
        while expr in subst:
            expr = subst[expr]
        return expr
    # If it's a function term: (f, arg1, arg2, ...)
    return (expr[0],) + tuple(substitute(sub, subst) for sub in expr[1:])
def unify(Y1, Y2, subst=None):
    if subst is None:
        subst = {}
    Y1 = substitute(Y1, subst)
    Y2 = substitute(Y2, subst)

    # Case 1: identical
    if Y1 == Y2:
        return subst

    # Case 2: Y1 is variable
    if isinstance(Y1, str):
        if occurs_check(Y1, Y2):
            return "FAILURE"
        subst[Y1] = Y2
        return subst
```

```
    # Case 3: Y2 is variable
    if isinstance(Y2, str):
        if occurs_check(Y2, Y1):
            return "FAILURE"
        subst[Y2] = Y1
        return subst

    # Case 4: function mismatch
    if Y1[0] != Y2[0] or len(Y1) != len(Y2):
        return "FAILURE"

    # Case 5: unify arguments
    for a, b in zip(Y1[1:], Y2[1:]):
        subst = unify(a, b, subst)
        if subst == "FAILURE":
            return "FAILURE"

    return subst
expr1 = ("p", "X", ("f", "Y"))
expr2 = ("p", "a", ("f", "b"))

output = unify(expr1, expr2)
print(output)
```

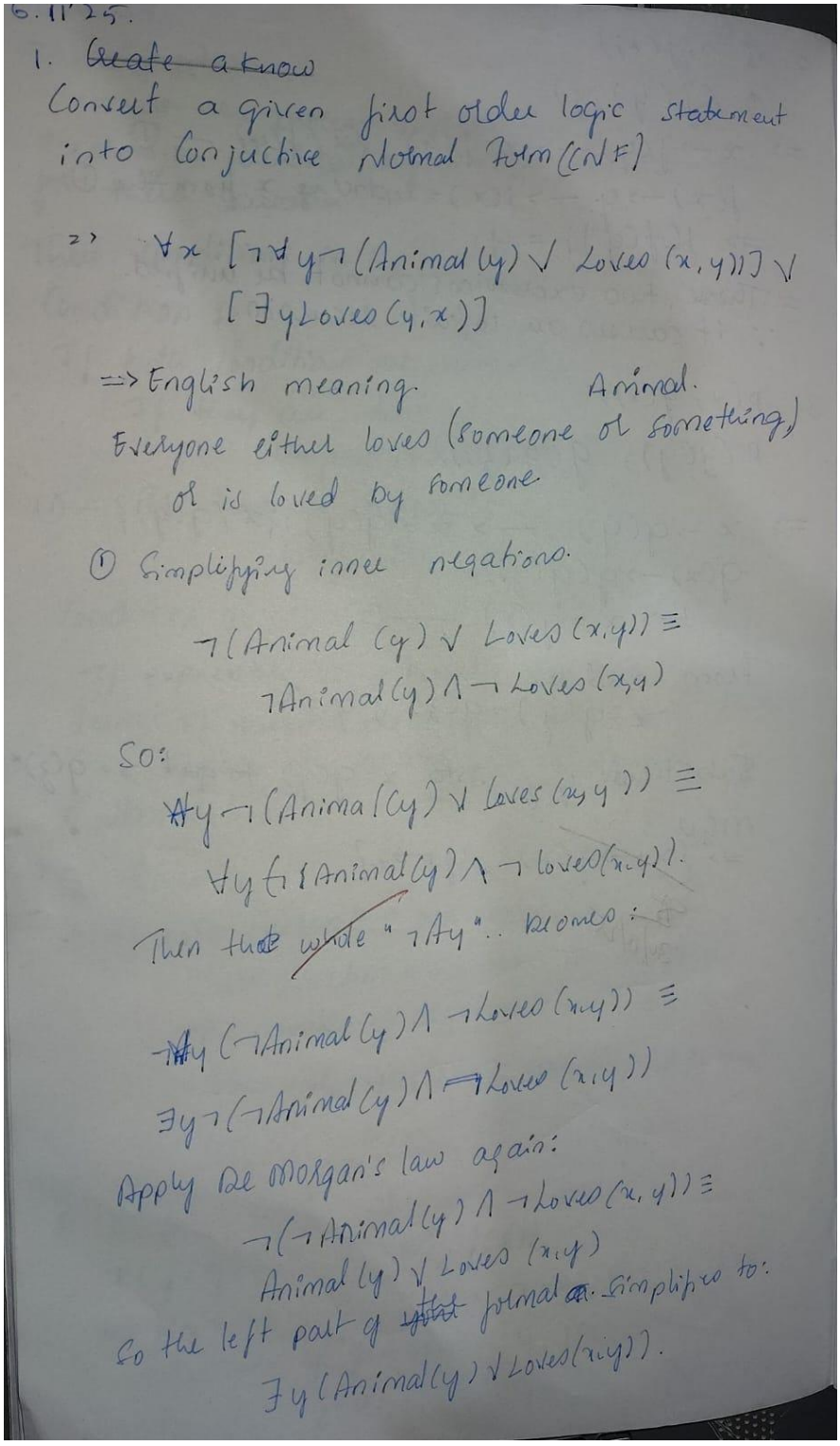
OUTPUT:

```
{'X': 'a', 'Y': 'b'}
```

Program 8

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

Algorithm:



② Substitute back into the full formula.

$\forall x [\exists y (Animal(y) \vee Loves(x,y))] \vee [\exists y\, Loves(y,x)]$

③ Simplify Structure

For every x: either there exists some y such that y is an animal of x loves y, of there exists some y who loves x.

$\forall x (\exists y (Animal(y) \vee Loves(x,y)) \vee \exists y\, Loves(y,x))$

of equivalently (prenex form):

$\forall x\, \exists y\, \exists z\, [Animal(y) \vee Loves(x,y) \vee Loves(z,x)]$.

1. Bring to prenex form.

$\forall x\, \exists y\, \exists z\, (Animal(y) \vee Loves(x,y) \vee Loves(z,x))$

2. Skolemize (remove existential quantifiers)

~~$y \to f(x)$~~

$z \to g(x)$

$\forall x (Animal(f(x)) \vee Loved(x, f(x)) \vee Loves(g(x), x))$

3. Drop universal quantifiers.

$Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(g(x), x)$.

4. Convert to CNF (clauses).

$\{Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(g(x), x)\}$.

1.) $Animal(f(x)) \vee Loves(x, f(x)) \vee Loves(g(x), x)$.

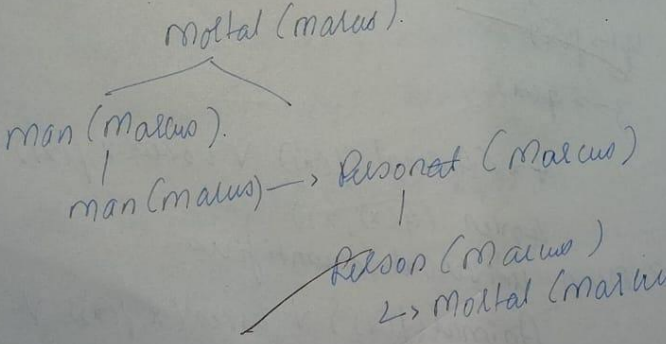
1. Create a Knowledge base consisting of first order logic statements and prove the given query using forward reasoning:

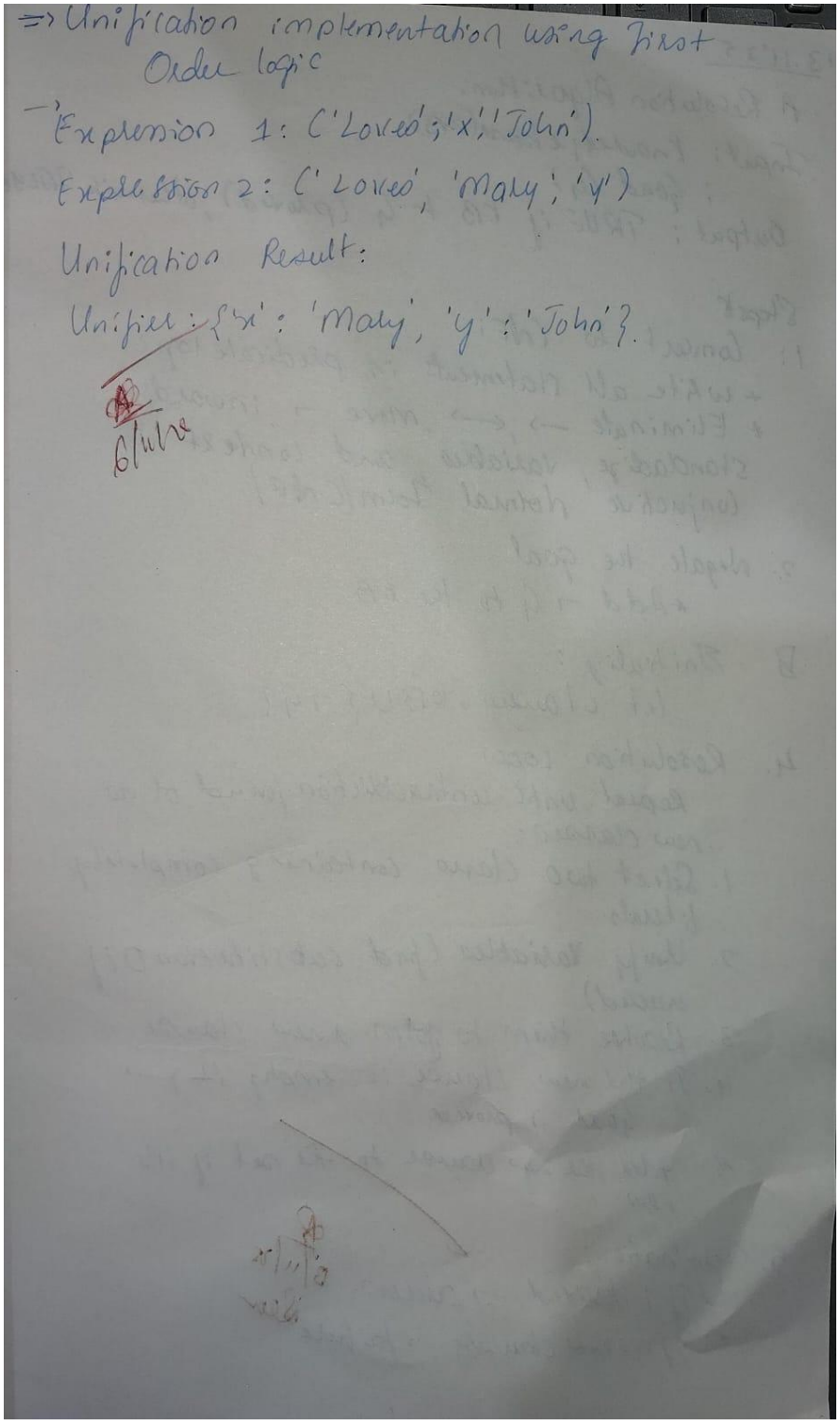
Knowledge Base (First-Order logic statements):

* Man(Marcus): Marcus is a man.
* Pompeian(Marcus): Marcus is a pompeian.
* All(x) (Pompeian(x) → Roman(x)): All Pompeians are Romans.
* All(x) (Roman(x) → Loyal(x)): All Romans are loyal.
* All(x) (Man(x) → Person(x)): All men are persons.
* All(x) (Person(x) → Mortal(x)): All persons are mortal.

Query:

Mortal(Marcus): Is Marcus mortal?




⇒ Unification implementation using First Order logic
→ Expression 1: ('Loved','x','John').
Expression 2: ('Loved', 'Mary','y')
Unification Result:
Unifier: {'x': 'Mary', 'y': 'John'}.

Code:

```
from collections import deque
class KnowledgeBase:
    def __init__(self):
        self.facts = set()
        self.rules = []
        self.inferred = set()

    def add_fact(self, fact):
        if fact not in self.facts:
            print(f"Adding fact: {fact}")
            self.facts.add(fact)
            return True
        return False

    def add_rule(self, premises, conclusion):
        self.rules.append((premises, conclusion))

    def forward_chain(self):
        agenda = deque(self.facts)

        while agenda:
            fact = agenda.popleft()
            if fact in self.inferred:
                continue
            self.inferred.add(fact)

            for (premises, conclusion) in self.rules:
                if all(p in self.inferred for p in premises):
                    if conclusion not in self.facts:
```

```
                print(f"Inferred new fact: {conclusion} from {premises} => {conclusion}")
                self.facts.add(conclusion)
                agenda.append(conclusion)

                if conclusion == 'Criminal(West)':
                    print("\n☑ Goal Reached: West is Criminal")
                    return True
        return False
kb = KnowledgeBase()

kb.add_fact('American(West)')
kb.add_fact('Enemy(Nono, America)')
kb.add_fact('Missile(M1)')
kb.add_fact('Owns(Nono, M1)')

kb.add_rule(premises=['Missile(M1)'], conclusion='Weapon(M1)')

kb.add_rule(premises=['Missile(M1)', 'Owns(Nono, M1)'], conclusion='Sells(West, M1, Nono)')

kb.add_rule(premises=['Enemy(Nono, America)'], conclusion='Hostile(Nono)')
kb.add_rule(premises=['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'],
conclusion='Criminal(West)')
kb.forward_chain()
```

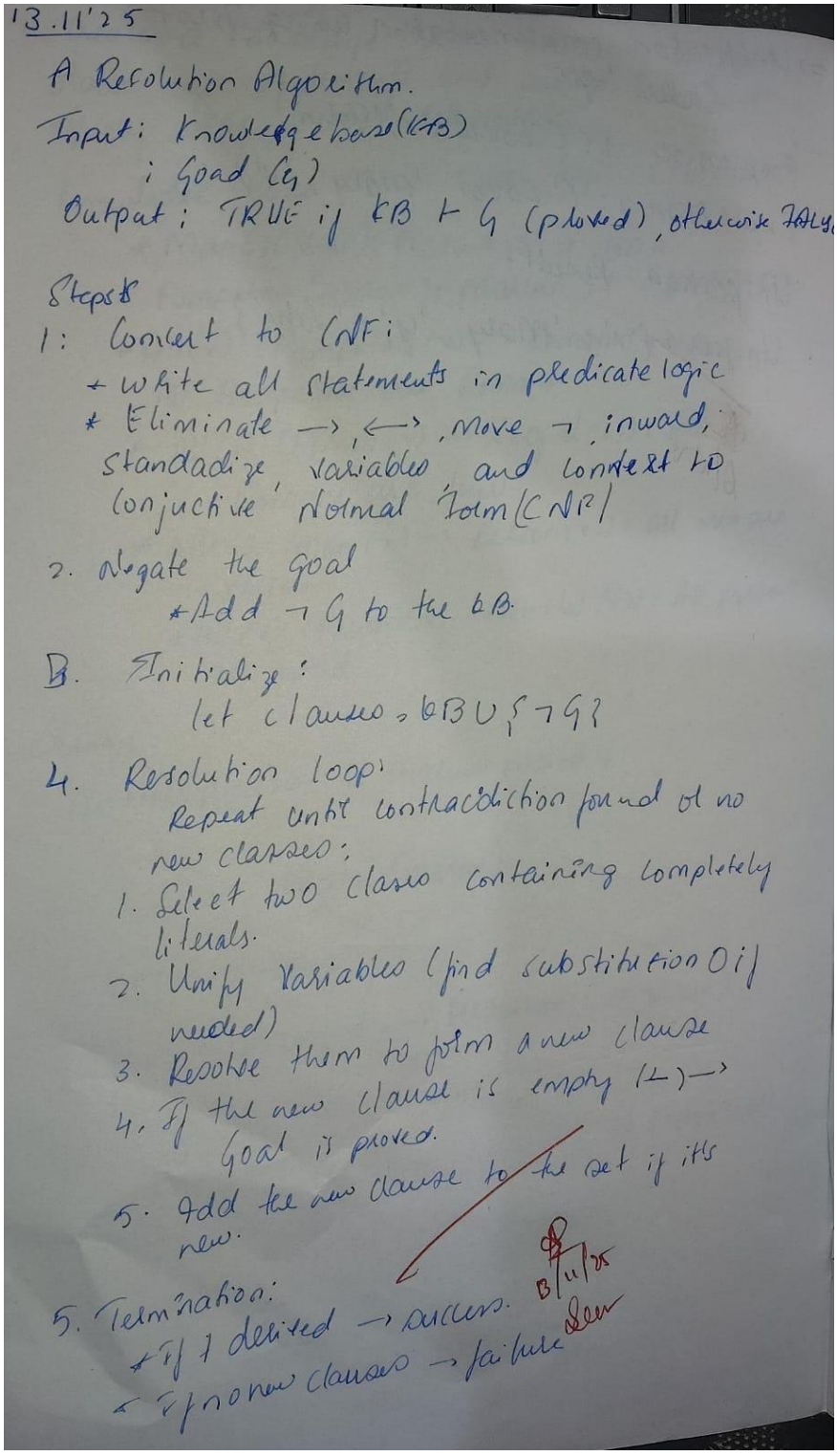
OUTPUT:

```
Adding fact: American(West)
Adding fact: Enemy(Nono, America)
Adding fact: Missile(M1)
Adding fact: Owns(Nono, M1)
Inferred new fact: Weapon(M1) from ['Missile(M1)'] => Weapon(M1)
Inferred new fact: Hostile(Nono) from ['Enemy(Nono, America)'] => Hostile(Nono)
Inferred new fact: Sells(West, M1, Nono) from ['Missile(M1)', 'Owns(Nono, M1)'] => Sells(West, M1, Nono)
Inferred new fact: Criminal(West) from ['American(West)', 'Weapon(M1)', 'Sells(West, M1, Nono)', 'Hostile(Nono)'] => Criminal(West)

✅ Goal Reached: West is Criminal
True
```

Program 9

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

Algorithm:



Code:

```
from itertools import combinations

def get_clauses():
    n = int(input("Enter number of clauses in Knowledge Base: "))
    clauses = []
    for i in range(n):
        clause = input(f"Enter clause {i+1}: ")
        clause_set = set(clause.replace(" ", "").split("v"))
        clauses.append(clause_set)
    return clauses

def resolve(ci, cj):
    resolvents = []
    for di in ci:
        for dj in cj:
            if di == ('~' + dj) or dj == ('~' + di):
                new_clause = (ci - {di}) | (cj - {dj})
                resolvents.append(new_clause)
    return resolvents

def resolution_algorithm(kb, query):
    kb.append(set(['~' + query]))
    derived = []
    clause_id = {frozenset(c): f"C{i+1}" for i, c in enumerate(kb)}

    step = 1
    while True:
        new = []
        for (ci, cj) in combinations(kb, 2):
```

```
            resolvents = resolve(ci, cj)
            for res in resolvents:
                if res not in kb and res not in new:
                    cid_i, cid_j = clause_id[frozenset(ci)], clause_id[frozenset(cj)]
                    clause_name = f"R{step}"
                    derived.append((clause_name, res, cid_i, cid_j))
                    clause_id[frozenset(res)] = clause_name
                    new.append(res)
                    print(f"[Step {step}] {clause_name} = Resolve({cid_i}, {cid_j}) → {res or '{}'}")
                    step += 1

                    # If empty clause found → proof complete
                    if res == set():
                        print("\n☑ Query is proved by resolution (empty clause found).")
                        print("\n--- Proof Tree ---")
                        print_tree(derived, clause_name)
                        return True
        if not new:
            print("\n❌ Query cannot be proved by resolution.")
            return False
        kb.extend(new)


def print_tree(derived, goal):
    tree = {name: (parents, clause) for name, clause, *parents in [(r[0], r[1], r[2:][0], r[2:][1]) for r in derived]}

    def show(node, indent=0):
        if node not in tree:
            print(" " * indent + node)
            return
```

```
        parents, clause = tree[node]
        print("  " * indent + f"{node}: {set(clause) or '{}'}")
        for p in parents:
            show(p, indent + 4)

    show(goal)
```

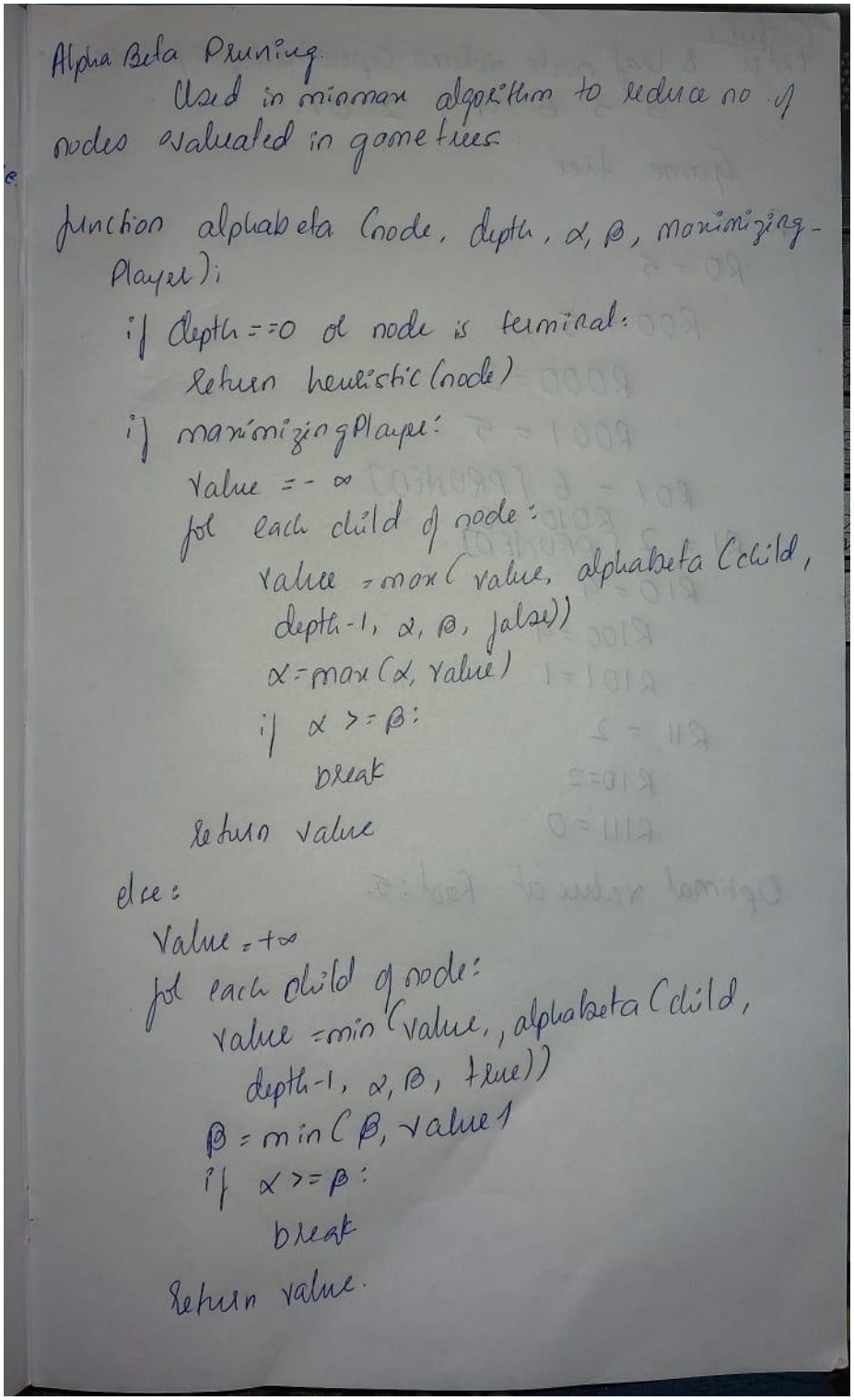
OUPUT:

```
=== FOL Resolution Demo with Proof Tree ===
Enter number of clauses in Knowledge Base: 3
Enter clause 1: P
Enter clause 2: ~P v Q
Enter clause 3: ~Q
Enter query to prove: Q
[Step 1] R1 = Resolve(C1, C2) → {'Q'}
[Step 2] R2 = Resolve(C2, C4) → {'~P'}
[Step 3] R3 = Resolve(C1, R2) → {}

✅ Query is proved by resolution (empty clause found).

--- Proof Tree ---
R3: {}
    C1
    R2: {'~P'}
        C2
        C4
True
```

Program 10

Implement Alpha-Beta Pruning.

Algorithm:



Alpha Beta Pruning.
Used in minmax algorithm to reduce no of nodes evaluated in game trees.

function alphabeta (node, depth, α, β, maximizing-Player);
if depth == 0 or node is terminal:
return heuristic (node)
if maximizingPlayer:
value = - ∞
for each child of node:
value = max (value, alphabeta (child, depth-1, α, β, false))
α = max (α, value)
if α >= β:
break
return value
else:
value = +∞
for each child of node:
value = min (value, alphabeta (child, depth-1, α, β, true))
β = min (β, value)
if α >= β:
break
Return value.

Output:
Enter 8 leaf node values seperated by spaces:
3 5 6 9 1 2 0 7.

Game tree

R = 5
- R0 = 5
  - R00 = 5
    - R000 = 3
    - R001 = 5
  - R01 = 6 [PRUNED]
    - R010 = 6
- R1 = 2 [PRUNED]
  - R10 = 9
    - R100 = 9
    - R101 = 1
  - R11 = 2
    - R110 = 2
    - R111 = 0

Optimal value at Root: 5.

Code:

```
class Node:
    def __init__(self, name):
        self.name = name
        self.children = []
        self.value = None
        self.pruned = False

def alpha_beta(node, depth, maximizing, values, alpha, beta, index):
    # Terminal node
    if depth == 3:
        node.value = values[index[0]]
        index[0] += 1
        return node.value

    if maximizing:
        best = float('-inf')
        for i in range(2): # 2 children
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, False, values, alpha, beta, index)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best
    else:
        best = float('inf')
```

```
        for i in range(2):
            child = Node(f"{node.name}{i}")
            node.children.append(child)
            val = alpha_beta(child, depth + 1, True, values, alpha, beta, index)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                node.pruned = True
                break
        node.value = best
        return best

def print_tree(node, indent=0):
    prune_mark = " [PRUNED]" if node.pruned else ""
    val = f" = {node.value}" if node.value is not None else ""
    print(" " * indent + f"{node.name}{val}{prune_mark}")
    for child in node.children:
        print_tree(child, indent + 4)

# --- main ---
print("=== Alpha-Beta Pruning with Tree ===")
values = list(map(int, input("Enter 8 leaf node values separated by spaces: ").split()))

root = Node("R")
alpha_beta(root, 0, True, values, float('-inf'), float('inf'), [0])

print("\n--- Game Tree ---")
print_tree(root)

print("\nOptimal Value at Root:", root.value)
```

OUTPUT:

```
=== Alpha-Beta Pruning with Tree ===
Enter 8 leaf node values separated by spaces: 3 5 6 9 1 2 0 7

--- Game Tree ---
R = 5
    R0 = 5
        R00 = 5
            R000 = 3
            R001 = 5
        R01 = 6 [PRUNED]
            R010 = 6
    R1 = 2 [PRUNED]
        R10 = 9
            R100 = 9
            R101 = 1
        R11 = 2
            R110 = 2
            R111 = 0

Optimal Value at Root: 5
```