VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Ritesh Mohan Nayak (1BM23CS350)

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
Sep-2024 to Jan-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

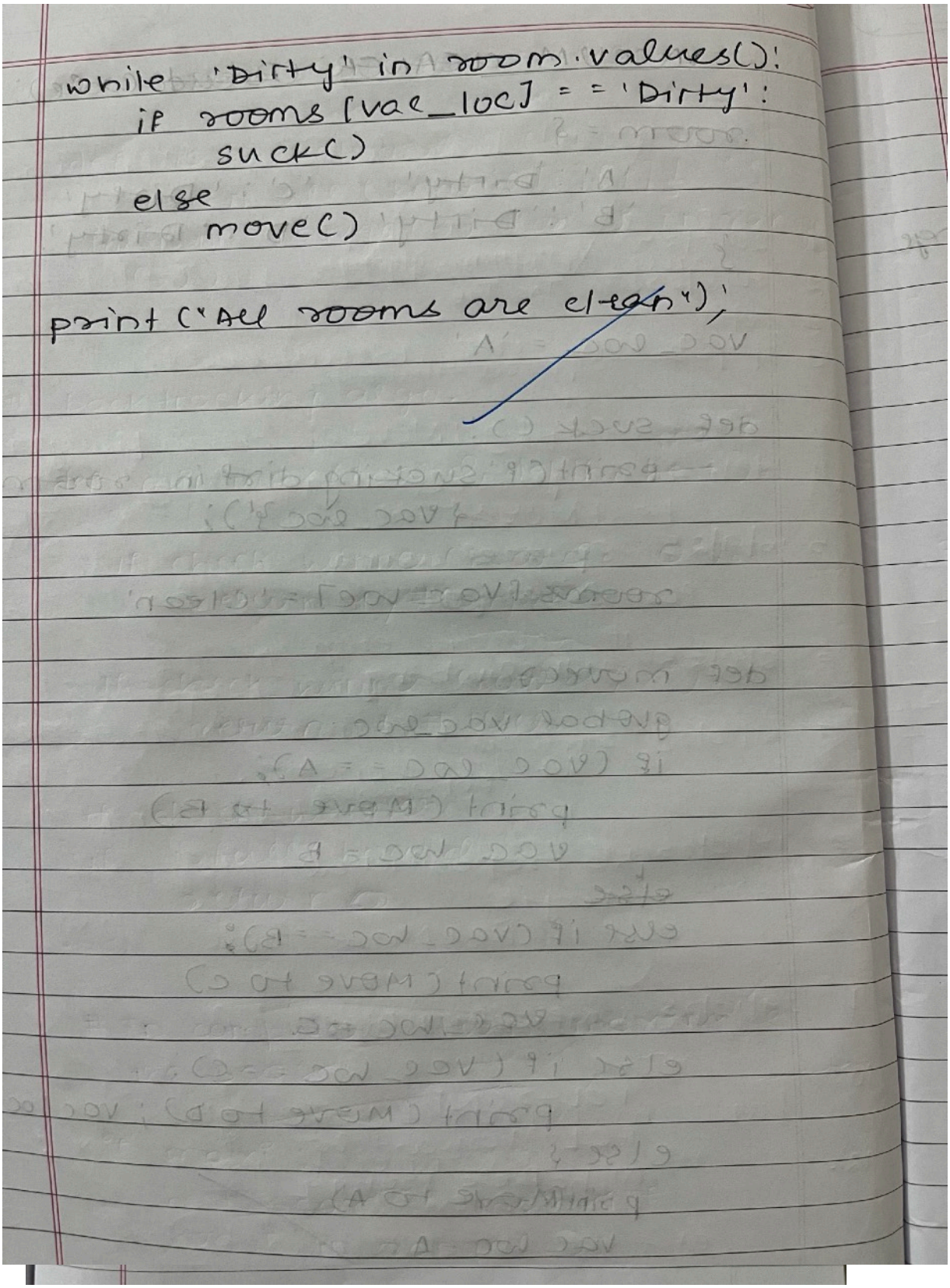
This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Ritesh Mohan Nayak (1BM23CS350),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** 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.

Prof. Sandhya A Kulkarni Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/riteshmohancs23-beep/AI_LAB

Program 1



Program 1.1: Implement Vacuum Cleaner

Algorithm:

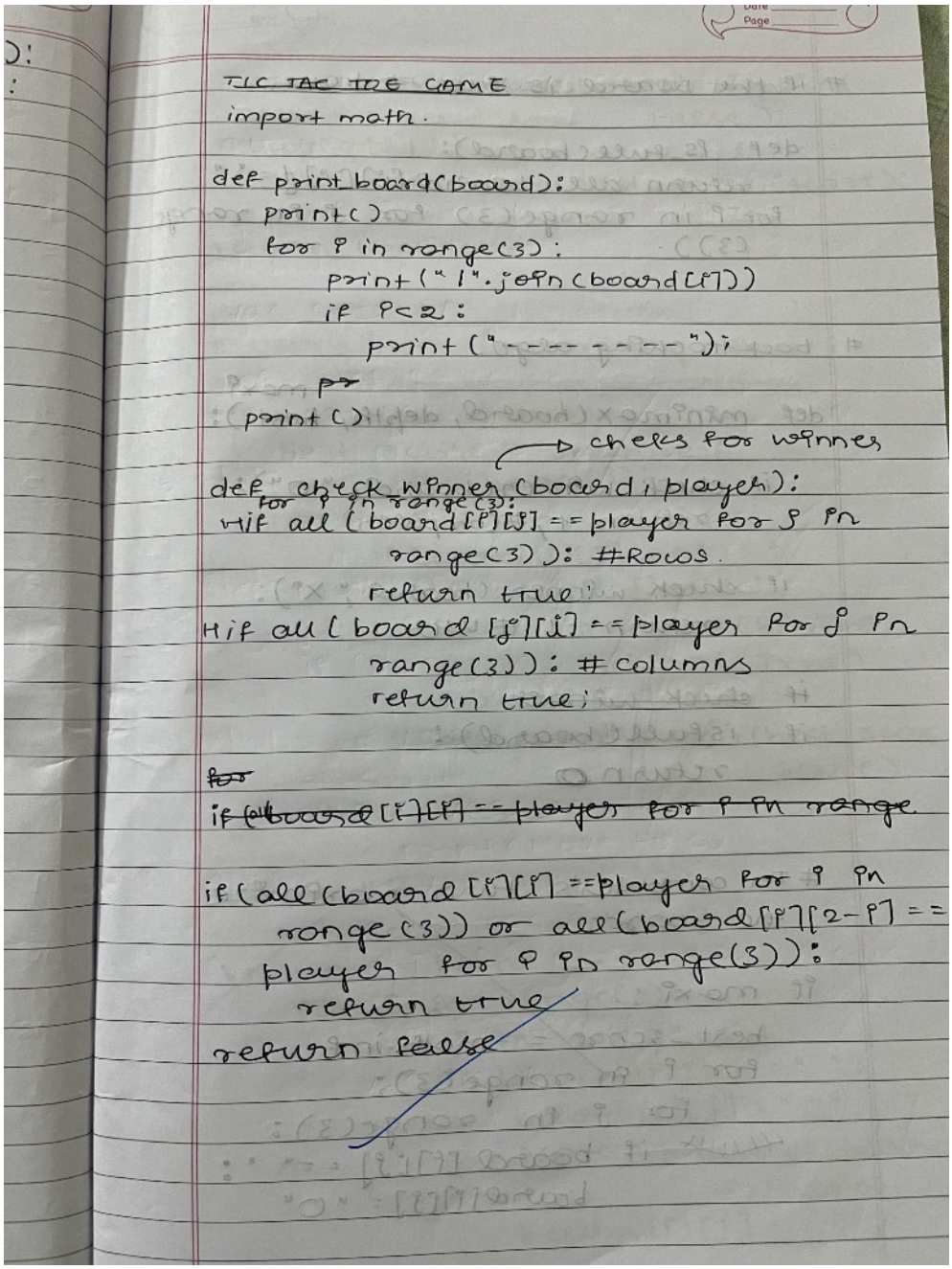
Code:

```
#VACCUM CLEANER
import random
import time

# Define 4 rooms
rooms = {
    "Room A": random.choice(["Clean", "Dirty"]),
    "Room B": random.choice(["Clean", "Dirty"]),
    "Room C": random.choice(["Clean", "Dirty"]),
    "Room D": random.choice(["Clean", "Dirty"])
}

def print_status():
    print("\nCurrent Room Status:")
    for room, status in rooms.items():
        print(f"{room}: {status}")

def vacuum_cleaner():
    print("Vacuum Cleaner Agent Started...\n")
    time.sleep(1)

    for room in rooms.keys():
        print(f"\nVacuum enters {room}...")
        if rooms[room] == "Dirty":
            print(f"{room} is Dirty. Cleaning...")
            rooms[room] = "Clean"
            time.sleep(1)
            print(f"{room} is now Clean ")
        else:
            print(f"{room} is already Clean ")
        time.sleep(1)

    print("\nAll rooms have been checked and cleaned!\n")

# Initial status
```

```
print_status()

# Run the cleaner
vacuum_cleaner()

# Final status
print_status()



#Output
# Current Room Status:
# Room A: Clean
# Room B: Dirty
# Room C: Dirty
# Room D: Clean
# Vacuum Cleaner Agent Started...


# Vacuum enters Room A...
# Room A is already Clean

# Vacuum enters Room B...
# Room B is Dirty. Cleaning...
# Room B is now Clean

# Vacuum enters Room C...
# Room C is Dirty. Cleaning...
# Room C is now Clean

# Vacuum enters Room D...
# Room D is already Clean

# All rooms have been checked and cleaned!


# Current Room Status:
# Room A: Clean
# Room B: Clean
# Room C: Clean
# Room D: Clean
```

Program 1.2: Implement Tic Tac Toe Game

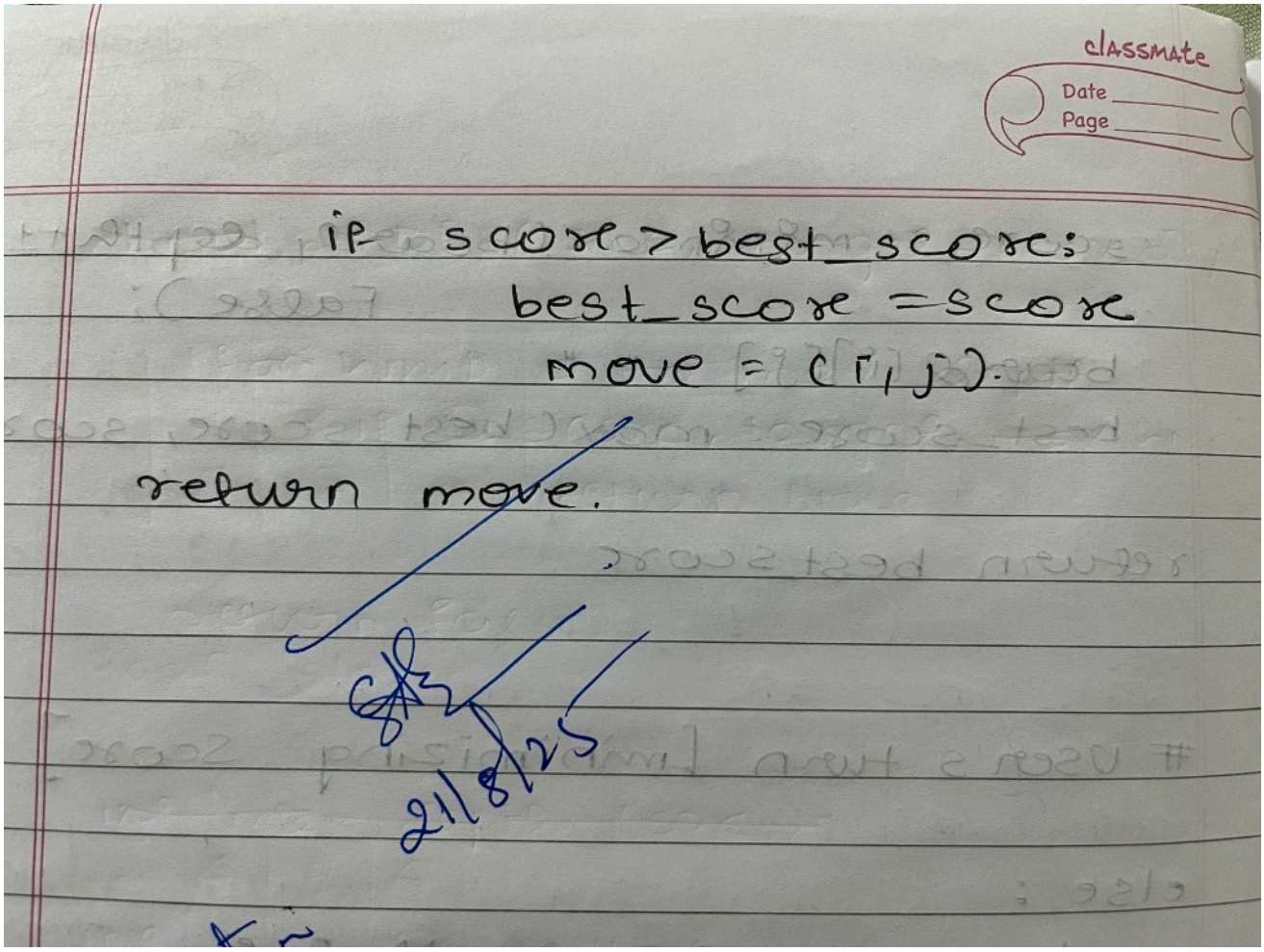
Algorithm:



```
TIC TAC TOE GAME
import math.

def print_board(board):
    print()
    for i in range(3):
        print(" | ".join(board[i]))
        if i<2:
            print("---------");
    print()

→ checks for winner

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j]==player for j in
               range(3)): #Rows.
            return true
    if all(board[j][i]==player for j in
               range(3)): # columns
            return true;

for
if (board[i][i]==player for i in range.

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
```

```
            score = minimax(board, depth+1,
                                False);
            board[i][j] = " "
            best_score = max(best_score, score);

    return best_score


# User's turn [minimizing score]

else:
    best_score = -math.Pnf
    For i Pn range(3):
        For j Pn range(3):
            if board[i][j] == " ":
                board[i][j] = "X"
                score = minimax(board,
                                depth+1, True)
                board[i][j] = " "
                best_score = min(best_score,
                                 score)
    return best_score.

def best_move(board):
    best score = -math.inf
    move = None
    For i Pn range(3):
        For j Pn range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                score = minimax(board,
                                0, False)
                board[i][j] = " "
```

```
# if the board is full => draw.

def is_full(board):
    return all(board[i][j] != " "
for i in range(3) for j in range
    (3)).


# backtracking algo.

def minimax(board, depth, maxi):

    if check_winner(board, "O"):
        return 1; #Computer

    if check_winner(board, "X"):
        return -1 # User.

    if is_full(board):
        return 0


# In computer's turn maximize
the score.

    if maxi:
        best_score = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == " ":
                    board[i][j] = "O"
```



Code:

```
import math

def print_board(board):
    print()
    for i in range(3):
        print(" | ".join(board[i]))
        if i < 2:
            print("---------")
    print()

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)):
            return True
        if 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(board[i][j] != " " for i in range(3) for j in range(3))

def minimax(board, depth, is_maximizing):
    if check_winner(board, "O"):
        return 1
    if check_winner(board, "X"):
        return -1
    if is_full(board):
        return 0

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

def best_move(board):
    best_score = -math.inf
    move = None
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                board[i][j] = "O"
                score = minimax(board, 0, False)
                board[i][j] = " "
                if score > best_score:
                    best_score = score
                    move = (i, j)
    return move
```

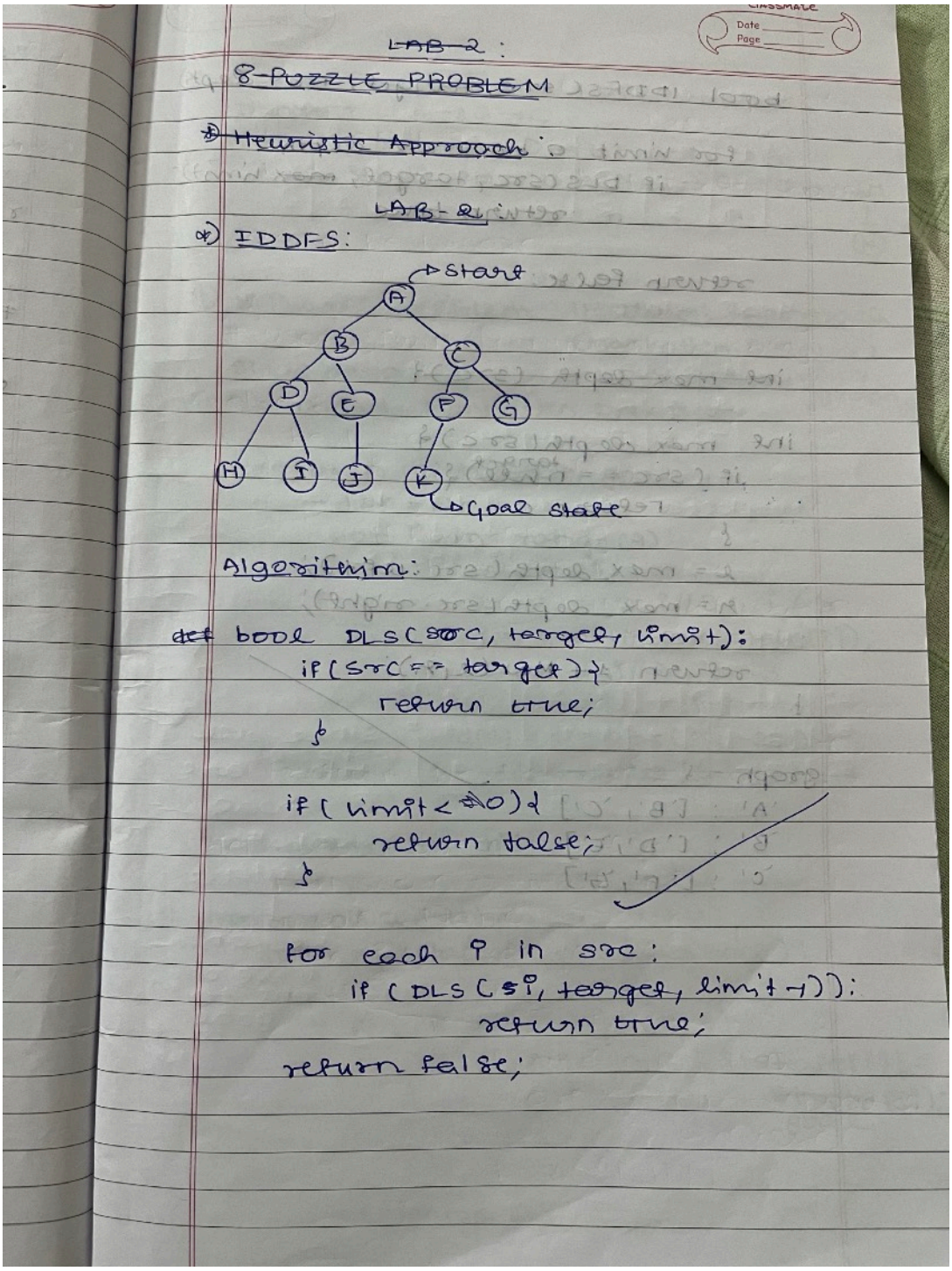
```
def play_game():
    board = [[" "]*3 for _ in range(3)]
    print("Welcome to Tic Tac Toe! You are X, Computer is O.\n")

    print_board(board)

    while True:
        while True:
            try:
                move = int(input("Enter your move (1-9): ")) - 1
                row, col = divmod(move, 3)
                if board[row][col] == " ":
                    board[row][col] = "X"
                    break
                else:
                    print("That spot is taken, try again.")
            except (ValueError, IndexError):
                print("Invalid input! Enter a number between 1-9.")

        print_board(board)

        if check_winner(board, "X"):
            print("You win!")
            break
        if is_full(board):
            print("It's a draw!")
            break

        print("Computer is thinking...")
        row, col = best_move(board)
        board[row][col] = "O"

        print_board(board)

        if check_winner(board, "O"):
            print("Computer wins!")
            break
        if is_full(board):
            print("It's a draw!")
            break

play_game()
```

```
# #OUTPUT
# Welcome to Tic Tac Toe! You are X, Computer is O .


#   |   |
# ---------
#   |   |
# ---------
#   |   |

# Enter your move (1-9):  o
# Invalid input! Enter a number between 1-9.
# Enter your move (1-9):  1

# X |   |
# ---------
#   |   |
# ---------
#   |   |

# Computer is thinking...

# X |   |
# ---------
#   | O |
# ---------
#   |   |

# Enter your move (1-9):  4

# X |   |
# ---------
# X | O |
# ---------
#   |   |

# Computer is thinking...

# X |   |
# ---------
# X | O |
# ---------
```

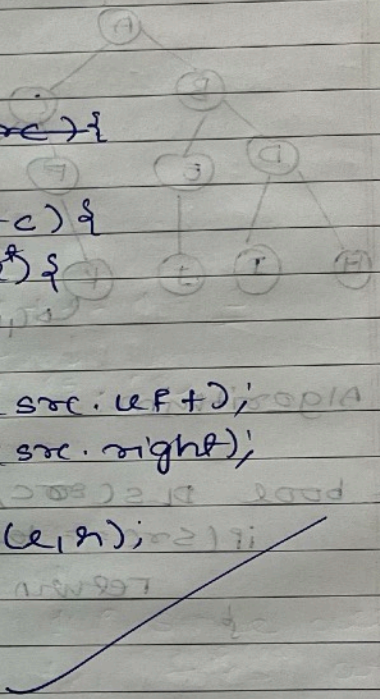
```
# O |   |

# Enter your move (1-9):  6

# X |   |
# ---------
# X | O | X
# ---------
# O |   |

# Computer is thinking...

# X | O |
# ---------
# X | O | X
# ---------
# O |   |

# Enter your move (1-9):  7
# That spot is taken, try again.
# Enter your move (1-9):  3

# X | O | X
# ---------
# X | O | X
# ---------
# O |   |

# Computer is thinking...

# X | O | X
# ---------
# X | O | X
# ---------
# O | O |

# Computer wins!
```

Program 2:

Program 2.1: 8 puzzle using IDDFS and DFS



```
bool IDDFS(src, target, max_depth)

    for limit 0 to max_depth
        if DLS(src, target, limit);
            return true

    return false;


int max_depth (src){

int max_depth(src) {
    if (src == target) {
        return 0;
    }
    l = max_depth(src.left);
    r = max_depth(src.right);
    return 1 + Max(l, r);
}.

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



Code:

```
# ---------- Depth Limited Search ----------
def dls(node, goal, depth, graph):
    """Depth Limited Search (recursive DFS with depth limit)."""
    if depth == 0 and node == goal:
        return True
    if depth > 0:
        for child in graph.get(node, []):
            if dls(child, goal, depth - 1, graph):
                return True
    return False


# ---------- IDDFS ----------
def iddfs(start, goal, max_depth, graph):
    """Iterative Deepening DFS"""
    for depth in range(max_depth + 1):
        print(f"Searching at depth {depth} ...")
        if dls(start, goal, depth, graph):
            print(f"Goal {goal} found at depth {depth}")
            return True
    print("Goal not found")
    return False


# ---------- Example Run ----------
if __name__ == "__main__":
    # Example tree (like your diagram)
    graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F', 'G'],
        'D': ['H', 'I'],
        'E': ['J'],
        'F': ['K'],   # Goal
        'G': [],
        'H': [],
        'I': [],
        'J': [],
        'K': []
    }

    start = 'A'
```

```
    goal = 'K'
    iddfs(state)
```

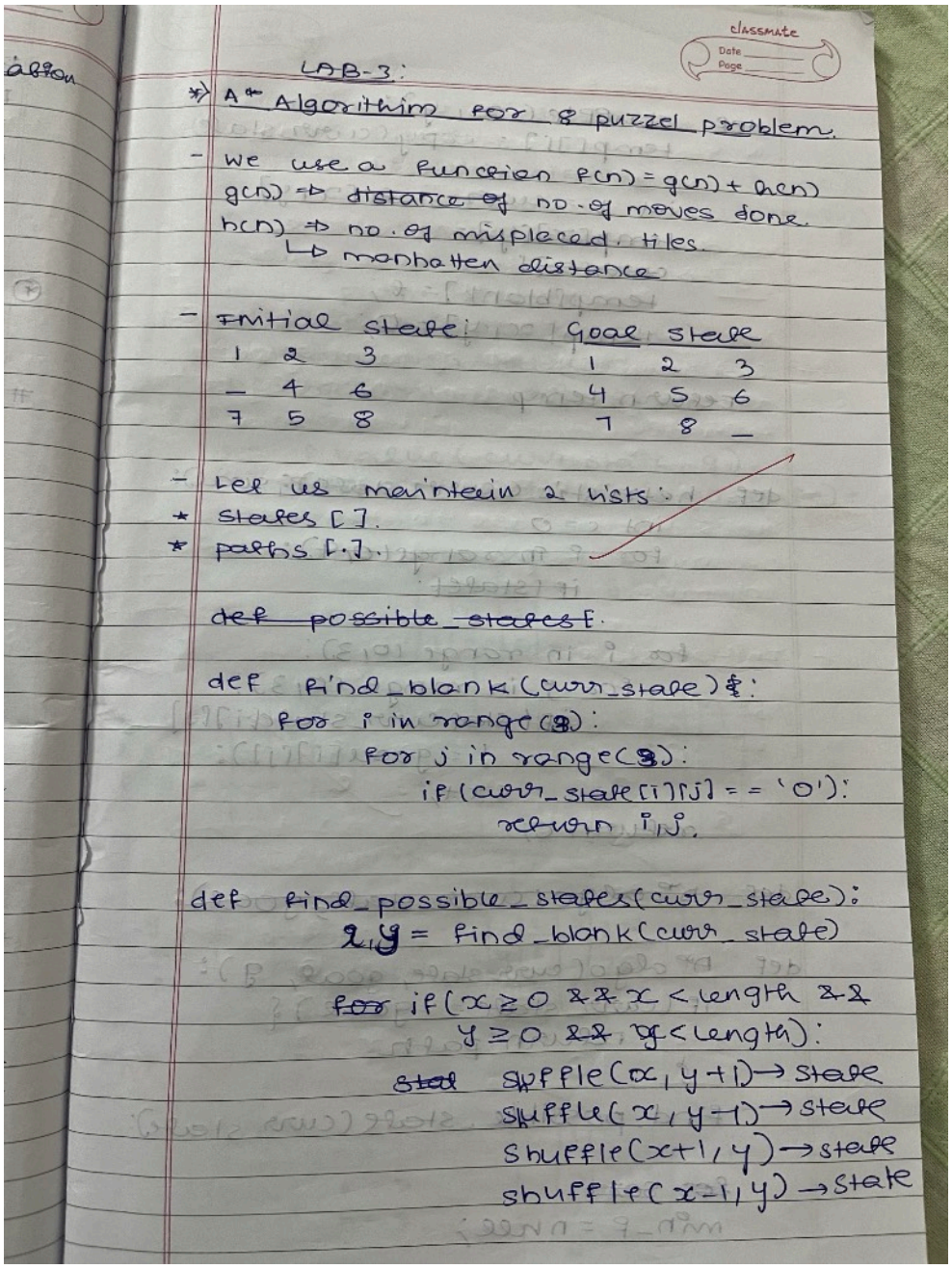
DFS

```
from collections import deque

def to_tuple(state):
    return tuple(state)

def print_board(state):
    for i in range(0, 9, 3):
        print(state[i], state[i+1], state[i+2])
    print()

def get_neighbors(state):
    neighbors = []
    blank = state.index(0)

    moves = {
        0: [1, 3],
        1: [0, 2, 4],
        2: [1, 5],
        3: [0, 4, 6],
        4: [1, 3, 5, 7],
        5: [2, 4, 8],
        6: [3, 7],
        7: [4, 6, 8],
        8: [5, 7]
    }

    for m in moves[blank]:
        new_state = list(state)
        new_state[blank], new_state[m] = new_state[m], new_state[blank]
        neighbors.append(new_state)

    return neighbors

def dfs(state, goal, visited):

    if state == goal:
```

```
        return [state]

    visited.add(to_tuple(state))

    for neighbor in get_neighbors(state):

        neighbor_t = to_tuple(neighbor)

        if neighbor_t not in visited:

            path = dfs(neighbor, goal, visited)
            if path:
                return [state] + path

    return None

def solve_8_puzzle_dfs(start, goal):
    visited = set()
    path = dfs(start, goal, visited)

    if path:
        for step in path:
            print_board(step)
    else:
        print("No solution found.")

    return path


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

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

solve_8_puzzle_dfs(start_state, goal_state)
```

Program 3: Implement A* Algorithm

Algorithm:



LAB-3:

A* Algorithm for 8 puzzel problem.

- We use a function f(n) = g(n) + h(n)
  g(n) => distance of no. of moves done.
  h(n) => no. of misplaced tiles.
  └> manhatten distance

- Initial state:

1	2	3
_	4	6
7	5	8

Goal state

1	2	3
4	5	6
7	8	_

- Let us maintain 2 lists:
  * states[].
  * paths[].

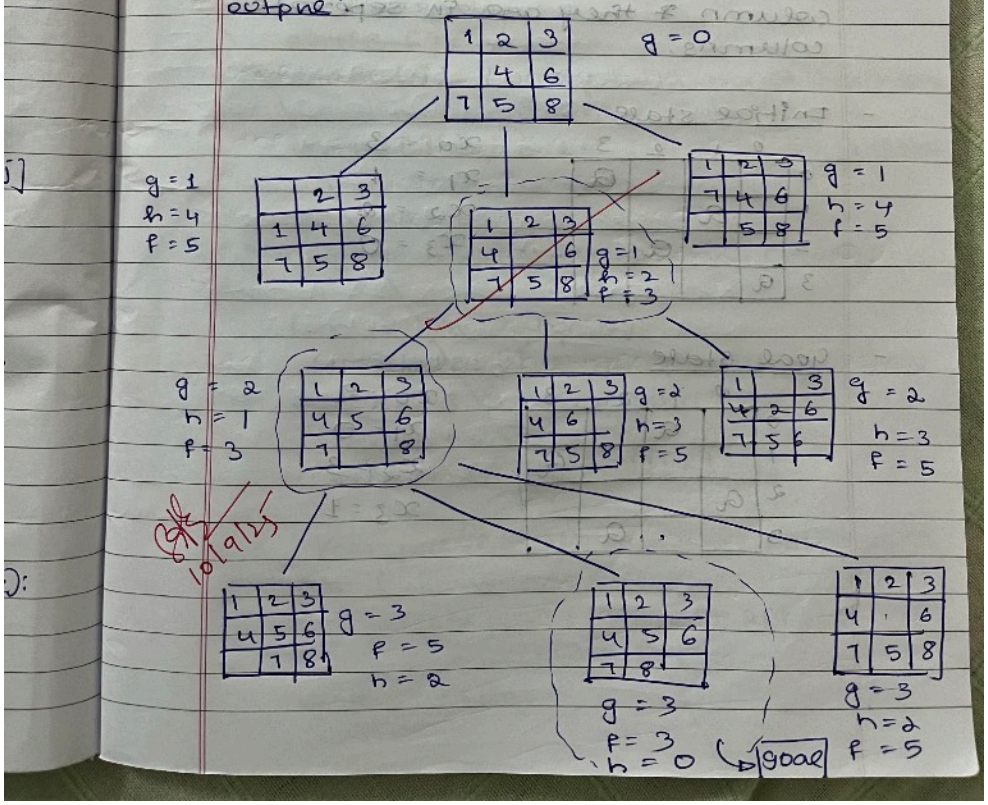
```
def possible_states():

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

def find_possible_states(curr_state):
    x,y = find_blank(curr_state)
    if(x≥0 && x<length &&
       y≥0 && y<length):
        shuffle(x,y+1)→state
        shuffle(x,y-1)→state
        shuffle(x+1,y)→state
        shuffle(x-1,y)→state
```

```
def shuffle(x, y):
    temp[][] = copy(curr_state);

    int t = curr_state[x, y]
    temp[x][y] =
    curr[x]
    temp[blank] = t;
    temp[x, y] = 0;

return temp


def heuristic(curr_state, goal):
    int c = 0
    for p in range(0, 9):
        if (state
    for i in range(0, 3):
        for j in range(0, 3):
            if (curr_state[i][j]
                != goal[i][j]):
                c++;
    return c;

def A* algo(curr_state, goal)

def A* algo(curr_state, goal, g):
    if (curr_state == goal) {
        return path
    }
    find_possible_state(curr_state);
    for sort
    min_f = null;
```

g = 1
h = 4
f = 5

```
for states in state:
    ~~h f_value~~
    if (f_value(states,g) < curr_val):
        curr_value = f_value(states,g)
        min = states.

        ~~path.add(sta~~
        ~~path.append(states).~~
path.append(min)
A*_algo(min, goal, g+1);

def f_value(curr_state, g):
    return heuristic(curr_state) + g
```

output:



Code:

```
class Node:
    def __init__(self,data,level,fval):
        """ Initialize the node with the data, level of the node and the calculated fvalue """
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        """ Generate child nodes from the given node by moving the blank space
            either in the four directions {up,down,left,right} """
        x,y = self.find(self.data,'_')
        """ val_list contains position values for moving the blank space in either of
            the 4 directions [up,down,left,right] respectively. """
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        """ Move the blank space in the given direction and if the position value are out
            of limits the return None """
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):
        """ Copy function to create a similar matrix of the given node"""
        temp = []
        for i in root:
            t = []
            for j in i:
```

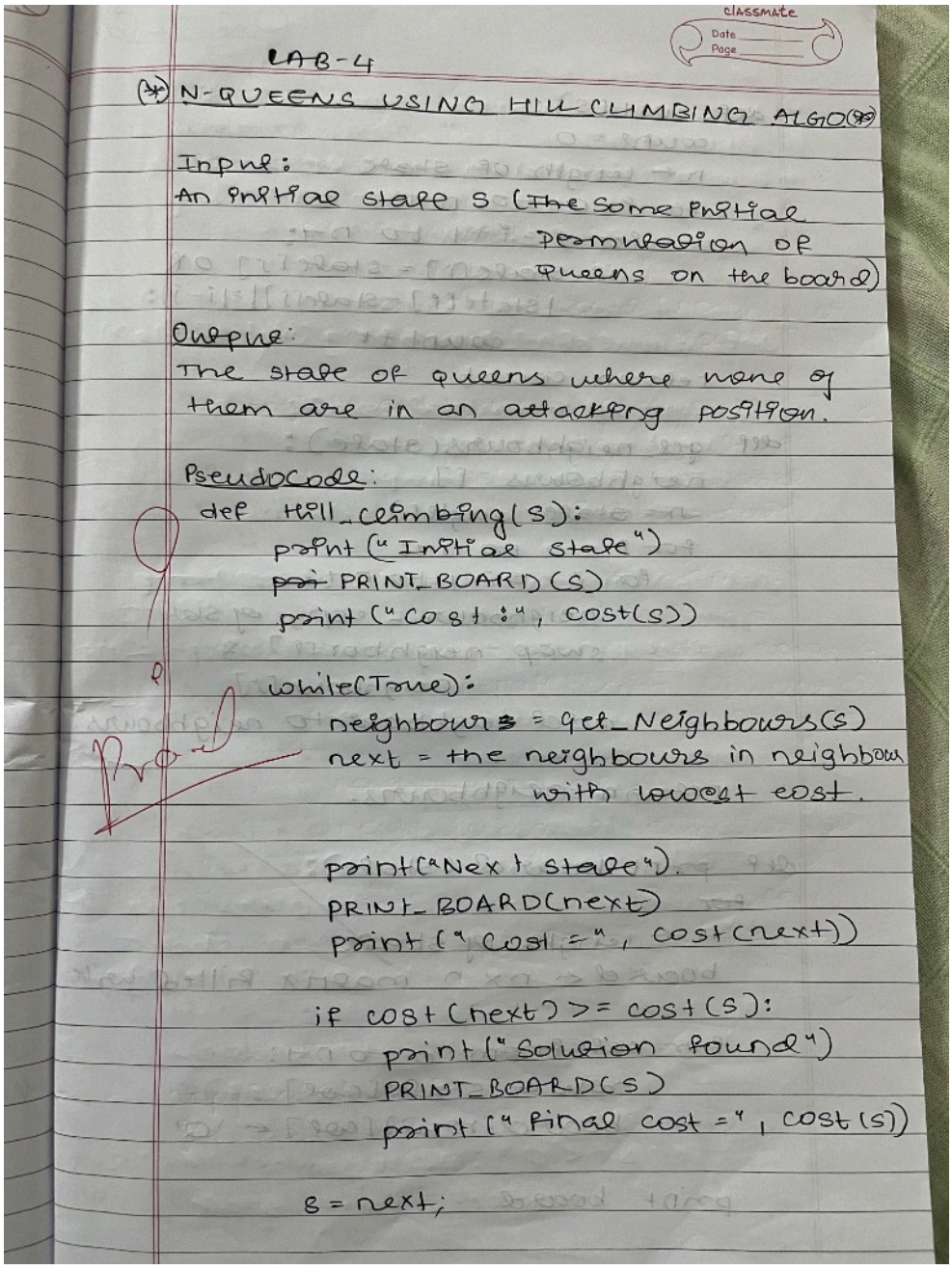
```
            t.append(j)
        temp.append(t)
    return temp

def find(self,puz,x):
    """ Specifically used to find the position of the blank space """
    for i in range(0,len(self.data)):
        for j in range(0,len(self.data)):
            if puz[i][j] == x:
                return i,j


class Puzzle:
    def __init__(self,size):
        """ Initialize the puzzle size by the specified size,open and closed lists to empty """
        self.n = size
        self.open = []
        self.closed = []

    def accept(self):
        """ Accepts the puzzle from the user """
        puz = []
        for i in range(0,self.n):
            temp = input().split(" ")
            puz.append(temp)
        return puz

    def f(self,start,goal):
        """ Heuristic Function to calculate hueristic value f(x) = h(x) + g(x) """
        return self.h(start.data,goal)+start.level

    def h(self,start,goal):
        """ Calculates the different between the given puzzles """
        temp = 0
        for i in range(0,self.n):
            for j in range(0,self.n):
                if start[i][j] != goal[i][j] and start[i][j] != '_':
                    temp += 1
        return temp


    def process(self):
        """ Accept Start and Goal Puzzle state"""
        print("Enter the start state matrix \n")
```

```
        start = self.accept()
        print("Enter the goal state matrix \n")
        goal = self.accept()

        start = Node(start,0,0)
        start.fval = self.f(start,goal)
        """ Put the start node in the open list"""
        self.open.append(start)
        print("\n\n")
        while True:
            cur = self.open[0]
            print("")
            print("  | ")
            print("  | ")
            print(" \\\'/ \n")
            for i in cur.data:
                for j in i:
                    print(j,end=" ")
                print("")
            """ If the difference between current and goal node is 0 we have reached the goal node"""
            if(self.h(cur.data,goal) == 0):
                break
            for i in cur.generate_child():
                i.fval = self.f(i,goal)
                self.open.append(i)
            self.closed.append(cur)
            del self.open[0]

            """ sort the opne list based on f value """
            self.open.sort(key = lambda x:x.fval,reverse=False)


puz = Puzzle(3)
puz.process()




## INPUT
# Enter the start state matrix

# 1 2 3
# 4 _ 6
# 7 5 8
```

```
# Enter the goal state matrix

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



## OUTPUT
#   |
#   |
#  \'/

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

#   |
#   |
#  \'/

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

#   |
#   |
#  \'/

# 1 2 3
# 4 5 6
# 7 8 _
```

Program 4: Implementing Hill Climbing Algorithm to solve 8 puzzle problem

Algorithm:



LAB-4

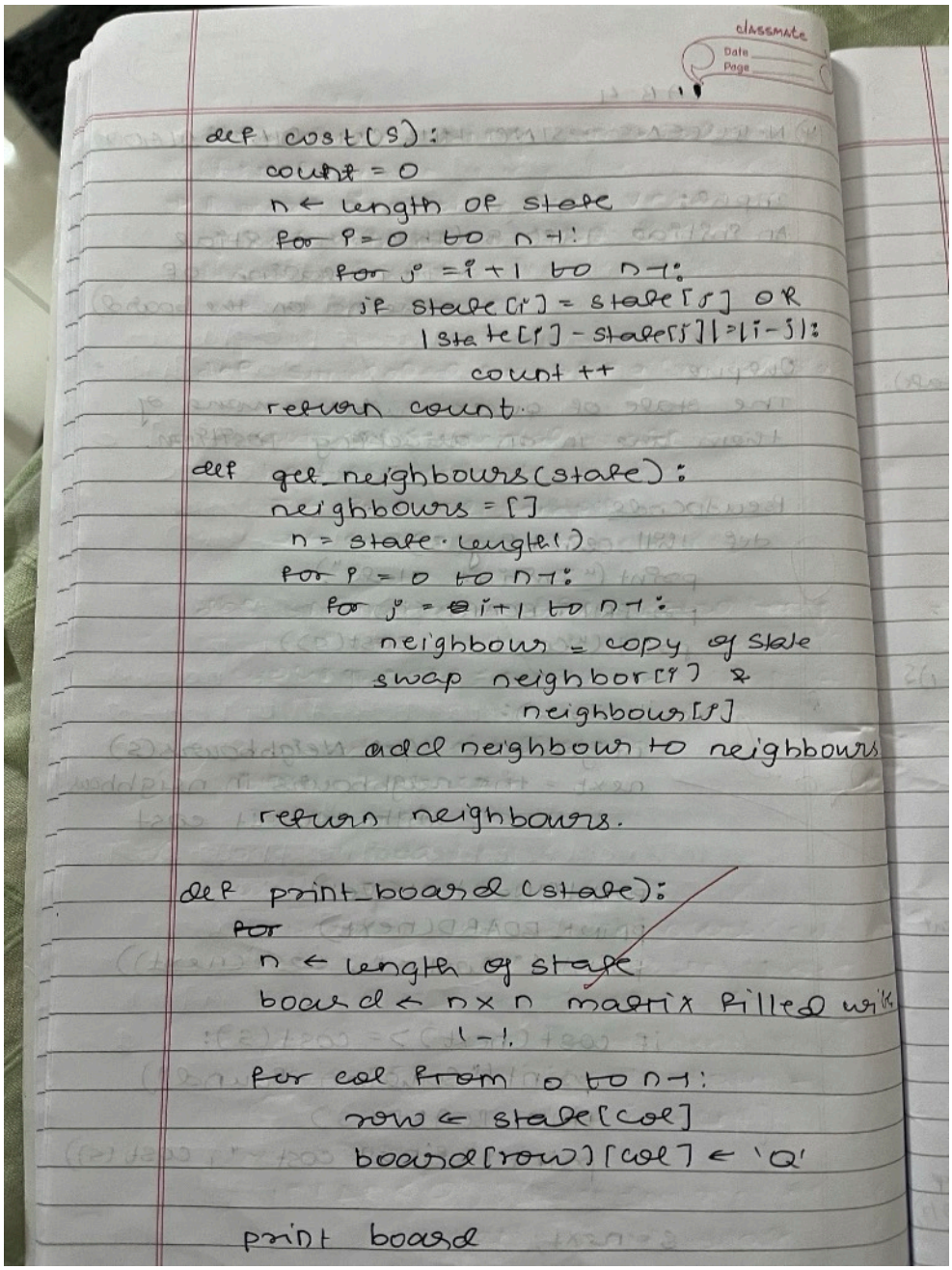
(*) N-QUEENS USING HILL CLIMBING ALGO (*)

Input:
An initial state S (The some initial permutation of queens on the board)

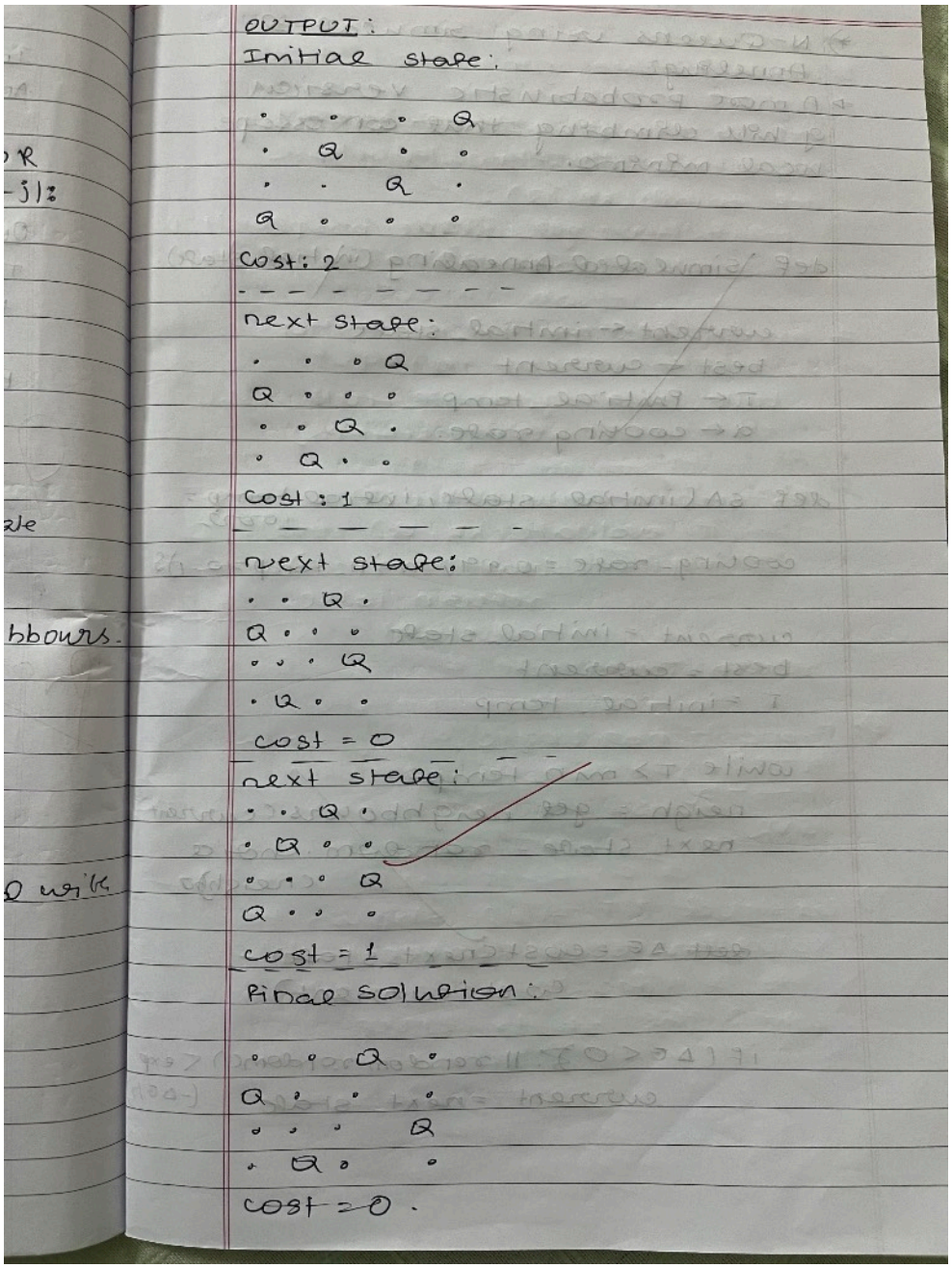
Output:
The state of queens where none of them are in an attacking position.

Pseudocode:

```
def Hill_climbing(S):
    print("Initial state")
    PRINT_BOARD(S)
    print("Cost:", cost(S))

    while(True):
        neighbours = get_Neighbours(S)
        next = the neighbours in neighbour with lowest cost.

        print("Next state").
        PRINT_BOARD(next)
        print("Cost =", cost(next))

        if cost(next) >= cost(S):
            print("Solution found")
            PRINT_BOARD(S)
            print("Final cost =", cost(S))

        S = next;
```



```
def cost(s):
    count = 0
    n ← length of state
    for i = 0 to n-1:
        for j = i+1 to n-1:
            if state[i] = state[j] OR
                |state[i] - state[j]| = |i-j|:
                    count ++
    return count.

def get_neighbours(state):
    neighbours = []
    n = state.length()
    for i = 0 to n-1:
        for j = i+1 to n-1:
            neighbour = copy of state
            swap neighbor[i] &
                 neighbours[j]
    add neighbour to neighbours

    return neighbours.

def print_board(state):
    n ← length of state
    board ← n x n matrix filled with
            '-'.
    for col from 0 to n-1:
        row ← state[col]
        board[row][col] ← 'Q'

    print board
```



OUTPUT:

Initial state:

. . . Q
. Q . .
. . Q .
Q . . .

cost: 2

next state:

. . . Q
Q . . .
. . Q .
. Q . .

cost : 1

next state:

. . Q .
Q . . .
. . . Q
. Q . .

cost = 0

next state:

. . Q .
. Q . .
. . . Q
Q . . .

cost = 1

Final solution:

. . . Q
Q . . .
. . . Q
. Q . .

cost = 0.

Code:

```
import random

def cost(state):
    """Calculate the number of attacking pairs of queens in the current state."""
    attacking_pairs = 0
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
                attacking_pairs += 1
    return attacking_pairs

def print_board(state):
    """Represent the state as a 4x4 board."""
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for i in range(n):
        board[state[i]][i] = 'Q'

    for row in board:
        print(" ".join(row))

def get_neighbors(state):
    """Generate all possible neighbors by swapping two queens."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]  # Swap queens
            neighbors.append(tuple(neighbor))
    return neighbors

def hill_climbing(initial_state):
    """Hill climbing algorithm to solve the N-Queens problem."""
    current = initial_state
    print(f"Initial state:")
    print_board(current)
    print(f"Cost: {cost(current)}")
    print('-' * 20)

    while True:
```

```
        neighbors = get_neighbors(current)
        # Select the neighbor with the lowest cost
        next_state = min(neighbors, key=lambda x: cost(x))
        print(f"Next state:")
        print_board(next_state)
        print(f"Cost: {cost(next_state)}")
        print('-' * 20)

        if cost(next_state) >= cost(current):
            # If no better state is found, return the current state as the solution
            print(f"Solution found:")
            print_board(current)
            print(f"Cost: {cost(current)}")
            return current
        current = next_state

if __name__ == "__main__":
    # Initial state for 4-Queens, random placement
    initial_state = (3, 1, 2, 0)  # Example initial state, where each index represents a column

    # Run Hill Climbing algorithm
    solution = hill_climbing(initial_state)
```

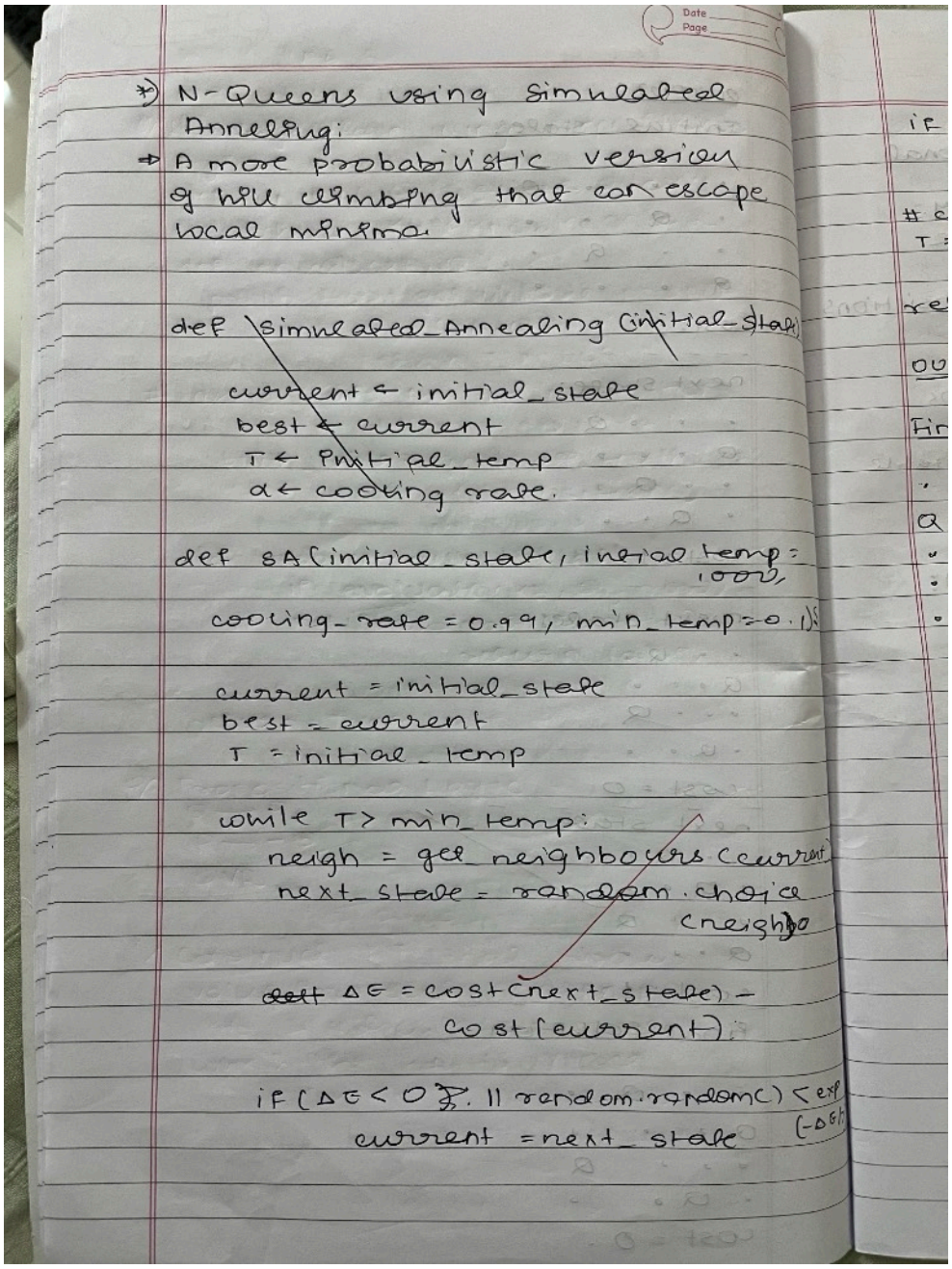
##OUTPUT

```
# Initial state:
# . . . Q
# . Q . .
# . . Q .
# Q . . .
# Cost: 2
# --------------------
# Next state:
# . . . Q
# Q . . .
# . . Q .
# . Q . .
# Cost: 1
# --------------------
# Next state:
# . . Q .
```

```
# Q . . .
# . . . Q
# . Q . .
# Cost: 0
# --------------------
# Next state:
# . . Q .
# . Q . .
# . . . Q
# Q . . .
# Cost: 1
# --------------------
# Solution found:
# . . Q .
# Q . . .
# . . . Q
# . Q . .
# Cost: 0
```

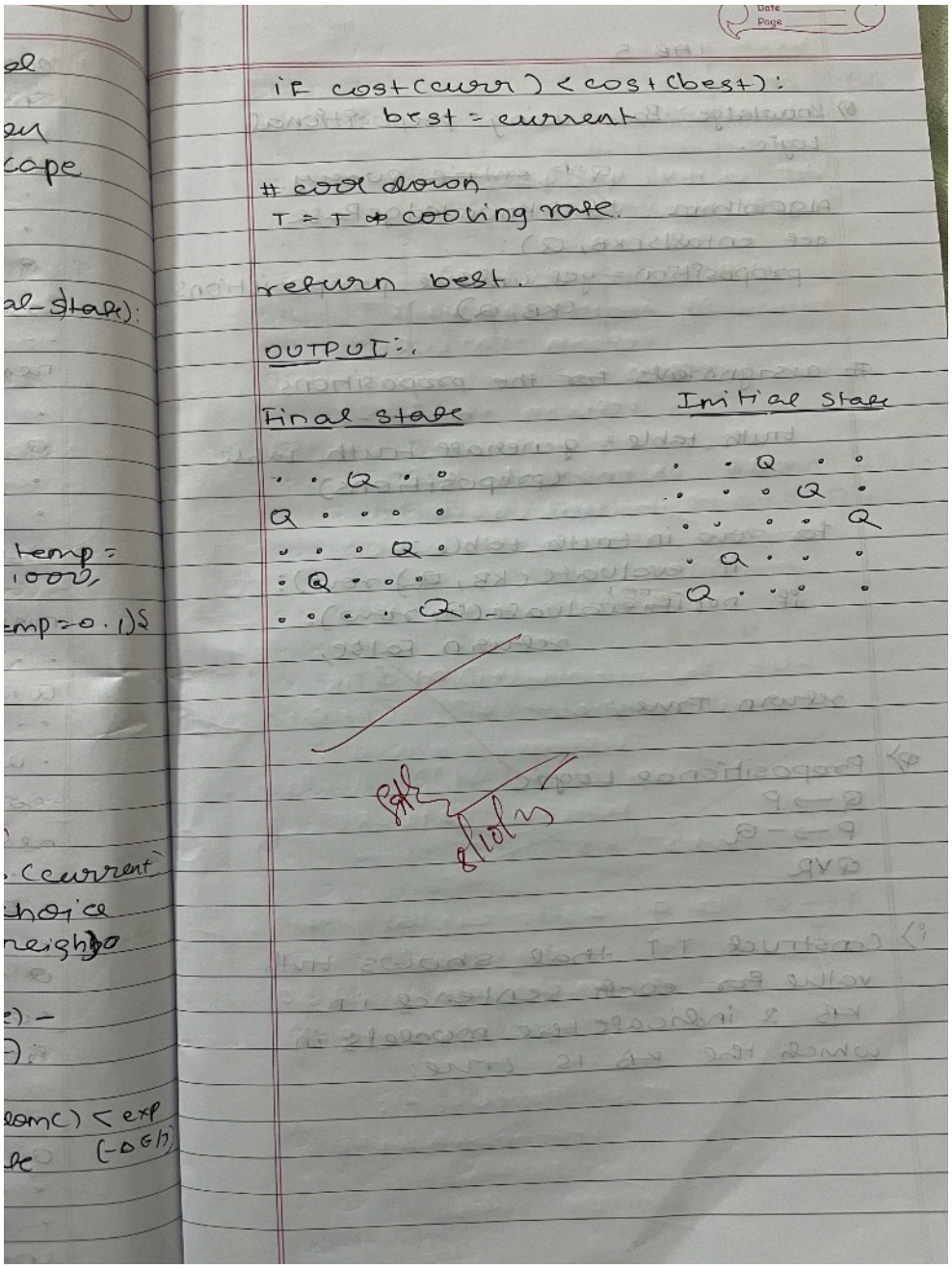
Program 5:Using Simulated Annealing to solve 8 puzzle problem

Algorithm:



*) N-Queens using simulated Annealing:
→ A more probabilistic version of hill climbing that can escape local minima.

```
def simulated_Annealing(initial_state):
    current ← initial_state
    best ← current
    T ← Initial_temp
    α ← cooling rate.

def SA(initial_state, initial_temp=1000,
    cooling_rate = 0.99, min_temp = 0.1):
    current = initial_state
    best = current
    T = initial_temp

    while T > min_temp:
        neigh = get_neighbours(current)
        next_state = random.choice(neighbo
        ΔE = cost(next_state) -
             cost(current)

        if (ΔE < 0 or random.random() < exp
            current = next_state
```



if cost(curr) < cost(best):
    best = current

# cool down
T = T * cooling rate

return best

OUTPUT:

Final state

. . Q . .
Q . . . .
. . . Q .
. Q . . .
. . . . Q

Initial state

. . Q . .
. . . Q .
. . . . Q
. Q . . .
Q . . . .

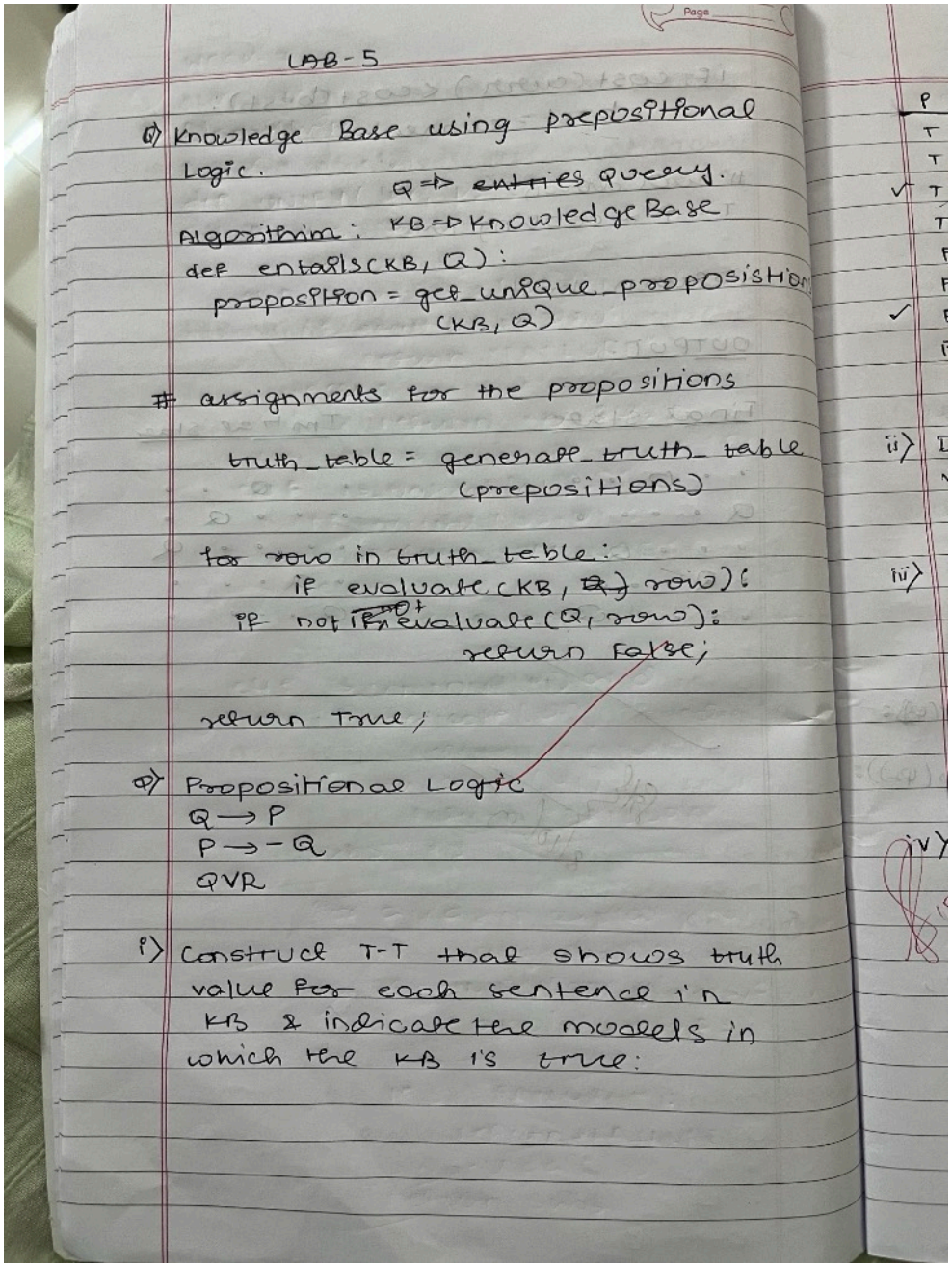
Code:

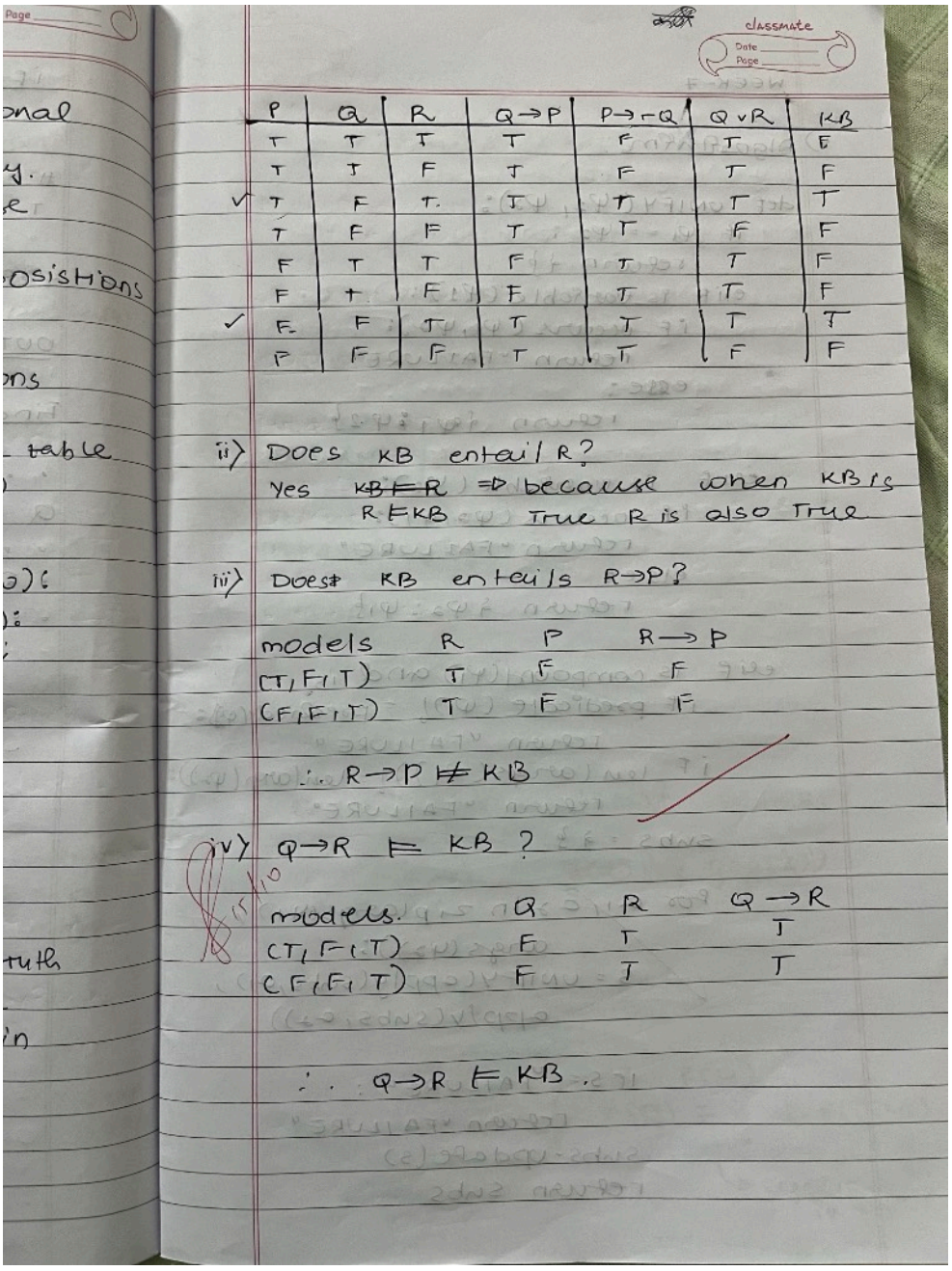
```
import random
import math

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

def print_board(state):
    n = len(state)
    board = [['.' for _ in range(n)] for _ in range(n)]
    for col in range(n):
        row = state[col]
        board[row][col] = 'Q'
    for row in board:
        print(" ".join(row))
    print()

def get_neighbors(state):
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i + 1, n):
            neighbor = list(state)
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(tuple(neighbor))
    return neighbors

def simulated_annealing(initial_state, initial_temp=100, cooling_rate=0.99, min_temp=0.1):
    current = initial_state
    best = current
    T = initial_temp
    print("Initial State:")
    print_board(current)
    print(f"Initial Cost: {cost(current)}")
    print('-' * 30)
    while T > min_temp:
        neighbors = get_neighbors(current)
```

```
        next_state = random.choice(neighbors)
        delta_E = cost(next_state) - cost(current)
        if delta_E < 0 or random.random() < math.exp(-delta_E / T):
            current = next_state
        if cost(current) < cost(best):
            best = current
        T = T * cooling_rate
        print(f"Temperature: {T:.2f} | Cost: {cost(current)} | Best: {cost(best)}")
    print('-' * 30)
    print("Final Solution Found:")
    print_board(best)
    print(f"Final Cost: {cost(best)}")
    return best

if __name__ == "__main__":
    n = 5
    initial_state = tuple(random.sample(range(n), n))
    solution = simulated_annealing(initial_state)
```

Program 6:Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not.

Algorithm;





P	Q	R	Q→P	P→¬Q	Q∨R	KB
T	T	T	T	F	T	F
T	T	F	T	F	T	F
✓ T	F	T	T	T	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	F
F	T	F	F	T	T	F
✓ F	F	T	T	T	T	T
F	F	F	T	T	F	F

ii) Does KB entail R?

Yes KB ⊨ R ⇒ because when KB is R ⊨ KB True R is also True

iii) Does KB entails R→P?

models	R	P	R→P
(T, F, T)	T	F	F
(F, F, T)	T	F	F

∴ R→P ⊭ KB

iv) Q→R ⊨ KB ?

models	Q	R	Q→R
(T, F, T)	F	T	T
(F, F, T)	F	T	T

∴ Q→R ⊨ KB.

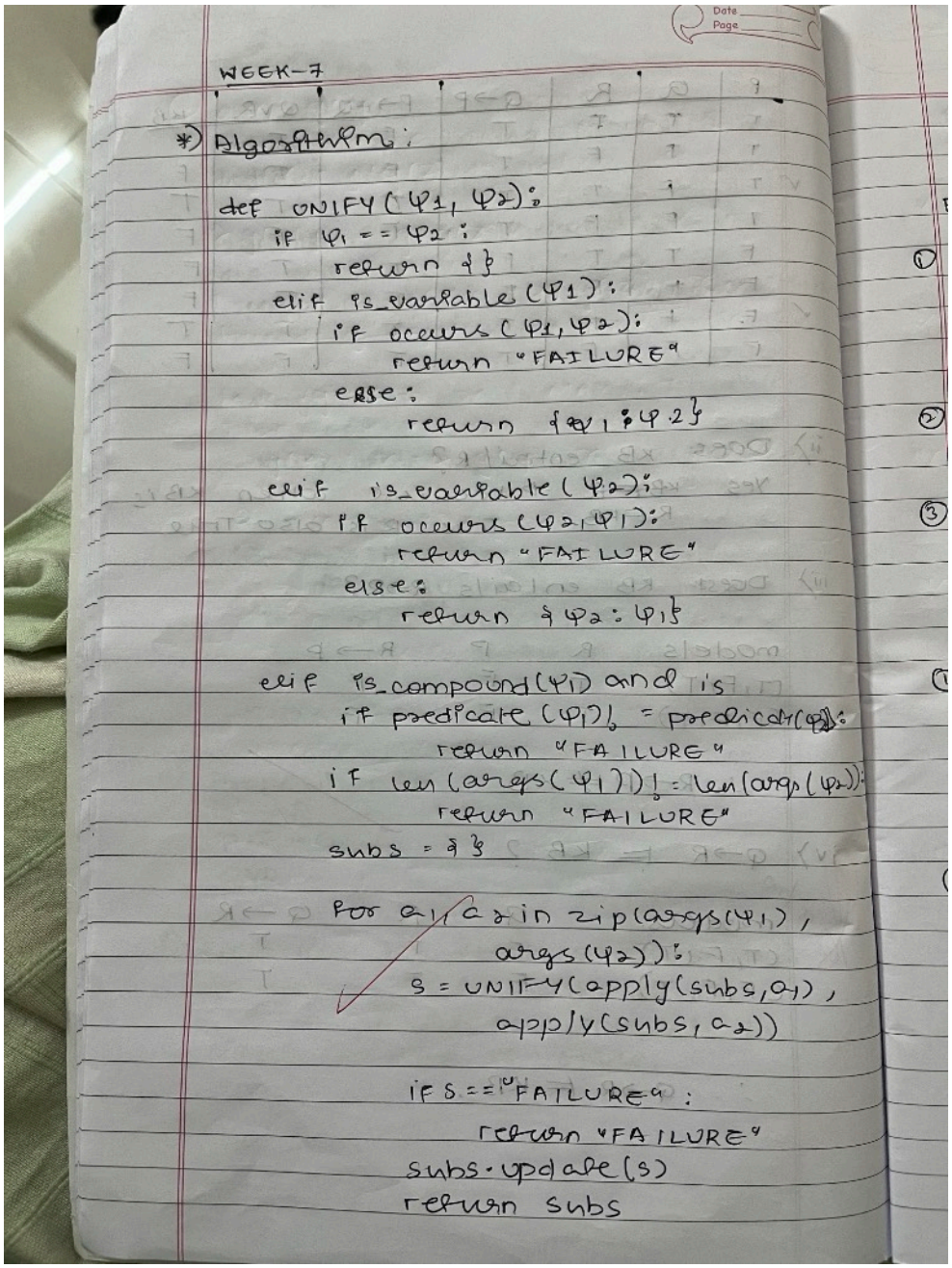
Code:

```
import itertools
import pandas as pd

vars = ['P', 'Q', 'R']

def implies(a, b):
    return (not a) or b

def KB(P, Q, R):
    s1 = implies(Q, P)
    s2 = implies(P, not Q)
    s3 = Q or R
    return s1, s2, s3, (s1 and s2 and s3)

def entails_R(P, Q, R):
    return R

def entails_R_implies_P(P, Q, R):
    return implies(R, P)

def entails_Q_implies_R(P, Q, R):
    return implies(Q, R)

data = []
models_KB_true = []

for P, Q, R in itertools.product([False, True], repeat=3):
    s1, s2, s3, kb = KB(P, Q, R)
    r = entails_R(P, Q, R)
    r_imp_p = entails_R_implies_P(P, Q, R)
    q_imp_r = entails_Q_implies_R(P, Q, R)
    data.append({
        'P': P, 'Q': Q, 'R': R,
        'Q→P': s1, 'P→¬Q': s2, 'Q∨R': s3,
        'KB': kb, 'R': r, 'R→P': r_imp_p, 'Q→R': q_imp_r
    })
    if kb:
        models_KB_true.append((P, Q, R))

df = pd.DataFrame(data)

def entails(KB_true_models, query_func):
    for P, Q, R in KB_true_models:
        if not query_func(P, Q, R):
            return False
    return True

print("\nModels where KB is TRUE:")
for m in models_KB_true:
    print(f"P={m[0]}, Q={m[1]}, R={m[2]}")

print("\nEntailment Results:")
print("KB ⊨ R:", entails(models_KB_true, entails_R))
print("KB ⊨ (R → P):", entails(models_KB_true, entails_R_implies_P))
```

```
print("KB ⊨ (Q → R):", entails(models_KB_true, entails_Q_implies_R))
print("TRUTH TABLE")
df


# OUTPUT:




#     P     Q     R   Q→P  P→¬Q   Q∨R    KB   R→P   Q→R
# False False False  True  True False False  True  True
# False False  True  True  True  True  True False  True
# False  True False False  True  True False  True False
# False  True  True False  True  True False False  True
#  True False False  True  True False False  True  True
#  True False  True  True  True  True  True  True  True
#  True  True False  True False  True False  True False
#  True  True  True  True False  True False  True  True

# Models where KB is TRUE:
# P=False, Q=False, R=True
# P=True, Q=False, R=True

# Entailment Results:
# KB ⊨ R: True
# KB ⊨ (R → P): False
# KB ⊨ (Q → R): True
```

Program 7:Implement unification in first order logic

Algorithm:



WEEK-7

*) Algorithm:

```
def UNIFY(ψ1, ψ2):
    if ψ1 == ψ2:
        return {}
    elif is_variable(ψ1):
        if occurs(ψ1, ψ2):
            return "FAILURE"
        else:
            return {ψ1: ψ2}

    elif is_variable(ψ2):
        if occurs(ψ2, ψ1):
            return "FAILURE"
        else:
            return {ψ2: ψ1}

    elif is_compound(ψ1) and is
        if predicate(ψ1) != predicate(ψ2):
            return "FAILURE"
        if len(args(ψ1)) != len(args(ψ2)):
            return "FAILURE"
        subs = {}

        for a1, a2 in zip(args(ψ1),
                          args(ψ2)):
            s = UNIFY(apply(subs, a1),
                      apply(subs, a2))

            if s == "FAILURE":
                return "FAILURE"
            subs.update(s)
        return subs
```

else:
return "FAILURE".

PROBLEMS:

① P(F(x), g(y), y)
P(F(g(z)), g(F(a)), F(a))

Find θ(MGU)

② Q(x, F(x))
Q(F(y), y)

③ H(x, g(x))
H(g(y), g(g(z)))

ANSWERS:

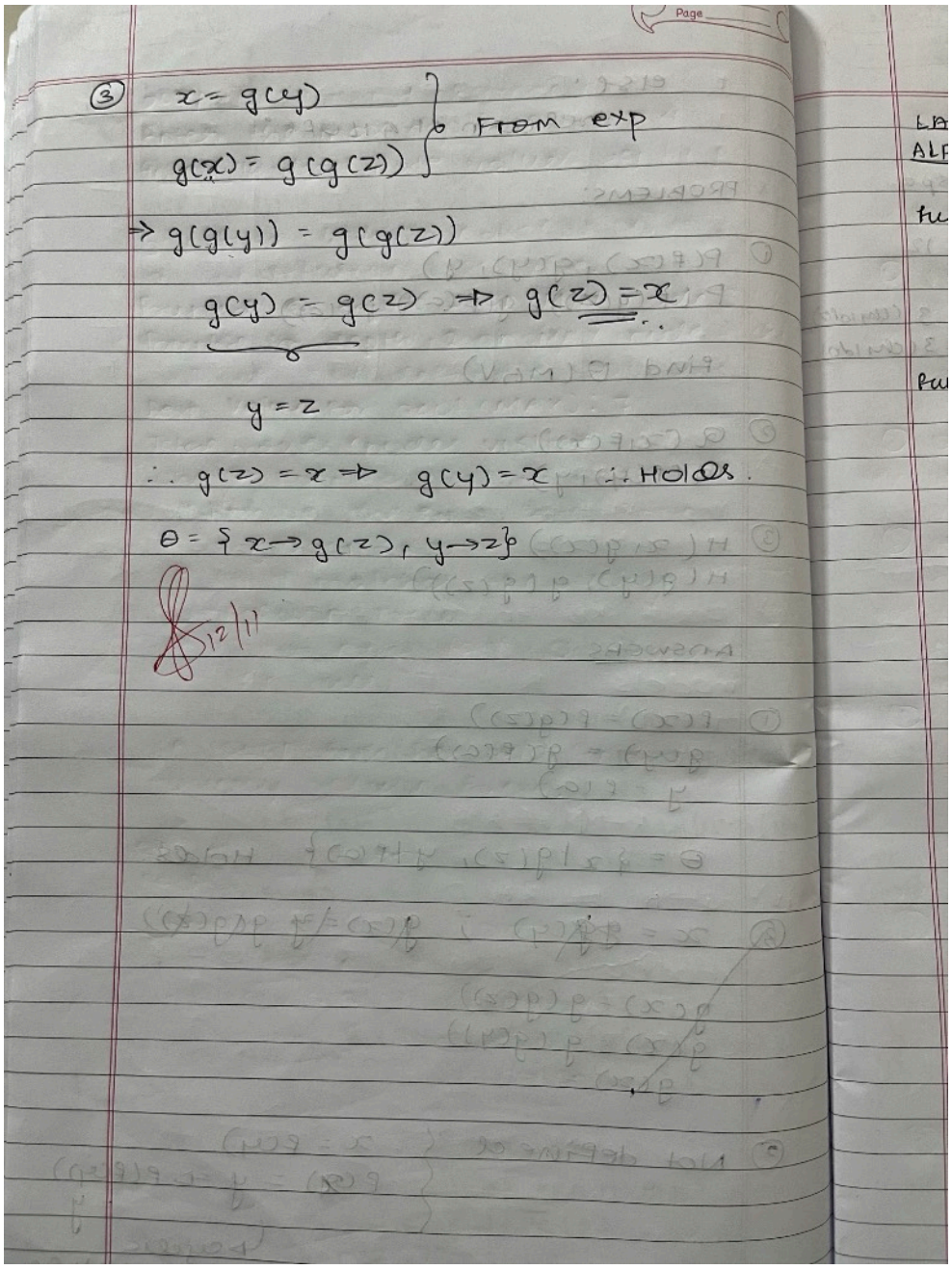
① F(x) = F(g(z))
g(y) = g(F(a))
y = F(a)

θ = {x | g(z), y | F(a)} Holds.

② x = g(y) ; g(x) = g(g(z))

g(x) = g(g(z))
g(x) = g(g(y))
g(x) =

③ Not defined { x = F(y)
F(x) = y ⇒ F(F(y))
= y
↳ cyclic in Nature.


③
x = g(y)
g(x) = g(g(z))
From exp
⇒ g(g(y)) = g(g(z))
g(y) = g(z) ⇒ g(z) = x
y = z
∴ g(z) = x ⇒ g(y) = x ∴ Hold.
θ = {x → g(z), y → z}
12/11

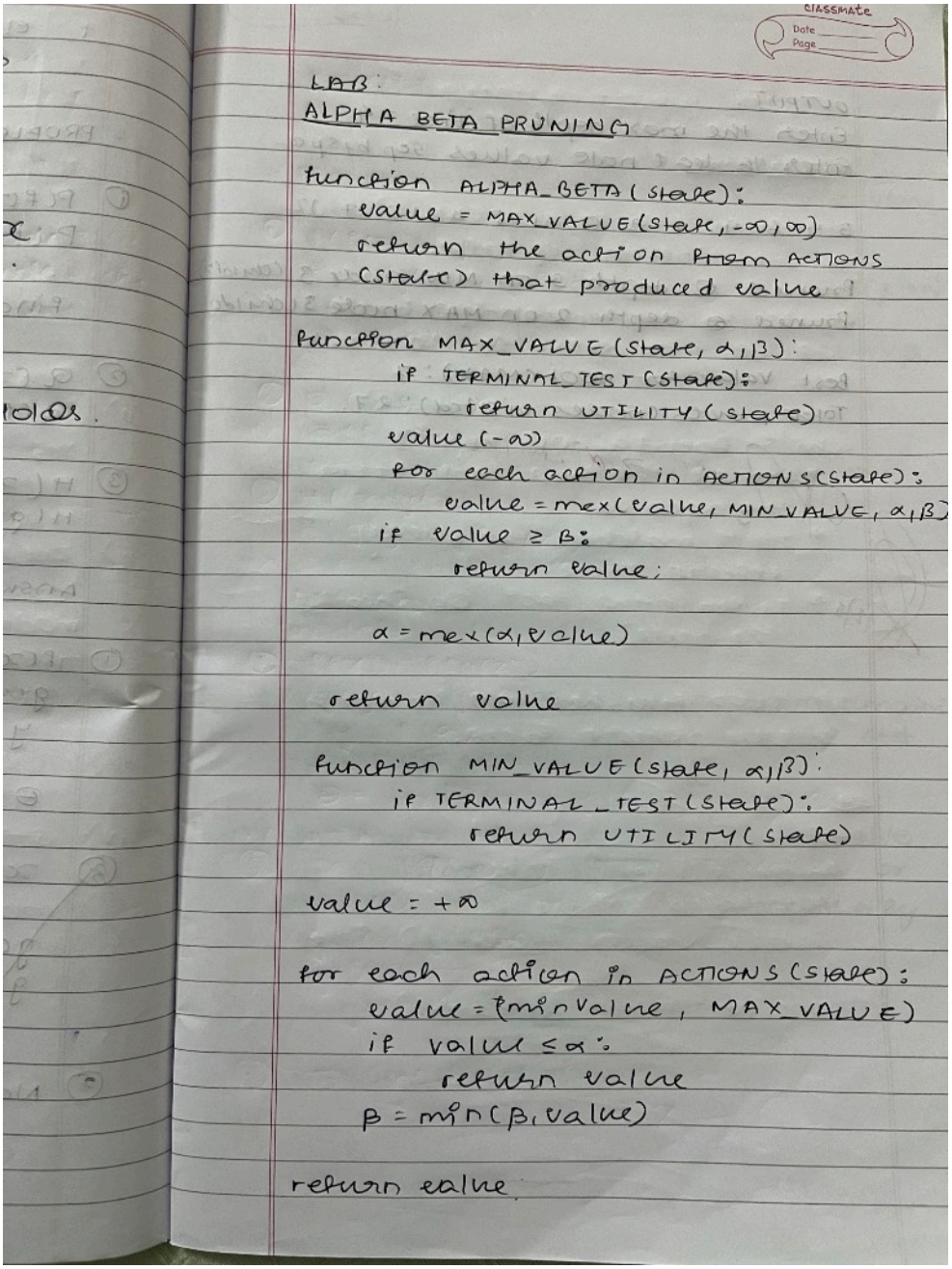
Code:

```
def occurs_check(var, term, subst):
    if var == term:
        return True
    elif isinstance(term, tuple):
        return any(occurs_check(var, t, subst) for t in term)
    elif term in subst:
        return occurs_check(var, subst[term], subst)
    return False

def unify(x, y, subst):
    if subst is None:
        return None
    elif x == y:
        return subst
    elif isinstance(x, str) and x.isupper():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.isupper():
        return unify_var(y, x, subst)
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if x[0] != y[0] or len(x) != len(y):
            return None
        for a, b in zip(x[1:], y[1:]):
            subst = unify(a, b, subst)
            if subst is None:
                return None
        return subst
    else:
        return None

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif x in subst:
        return unify(var, subst[x], subst)
    elif occurs_check(var, x, subst):
        return None
    else:
        subst[var] = x
        return subst

def parse_expr(s):
    s = s.replace(" ", "")
    if '(' not in s:
        return s
    name_end = s.index('(')
    name = s[:name_end]
    args = []
    depth = 0
    current = ""
    for c in s[name_end+1:-1]:
        if c == ',' and depth == 0:
            args.append(parse_expr(current))
            current = ""
```

```
        else:
            if c == '(':
                depth += 1
            elif c == ')':
                depth -= 1
            current += c
    if current:
        args.append(parse_expr(current))
    return tuple([name] + args)

def expr_to_str(expr):
    if isinstance(expr, tuple):
        return expr[0] + "(" + ",".join(expr_to_str(e) for e in expr[1:]) + ")"
    else:
        return expr

expr1_input = input("Enter first expression: ")
expr2_input = input("Enter second expression: ")

expr1 = parse_expr(expr1_input)
expr2 = parse_expr(expr2_input)

subst = unify(expr1, expr2, {})

if subst:
    formatted_subst = {var: expr_to_str(val) for var, val in subst.items()}
else:
    formatted_subst = None

print("Most General Unifier (MGU):", formatted_subst)
```

Program 8:Implement Alpha-Beta Pruning.

Algorithm:



LAB:

ALPHA BETA PRUNING

function ALPHA_BETA (state):
    value = MAX_VALUE (state, -∞, ∞)
    return the action from ACTIONS (state) that produced value

function MAX_VALUE (state, α, β):
    if TERMINAL_TEST (state):
        return UTILITY (state)
    value (-∞)
    for each action in ACTIONS (state):
        value = max (value, MIN_VALUE, α, β)
    if value ≥ β:
        return value;

    α = max (α, value)

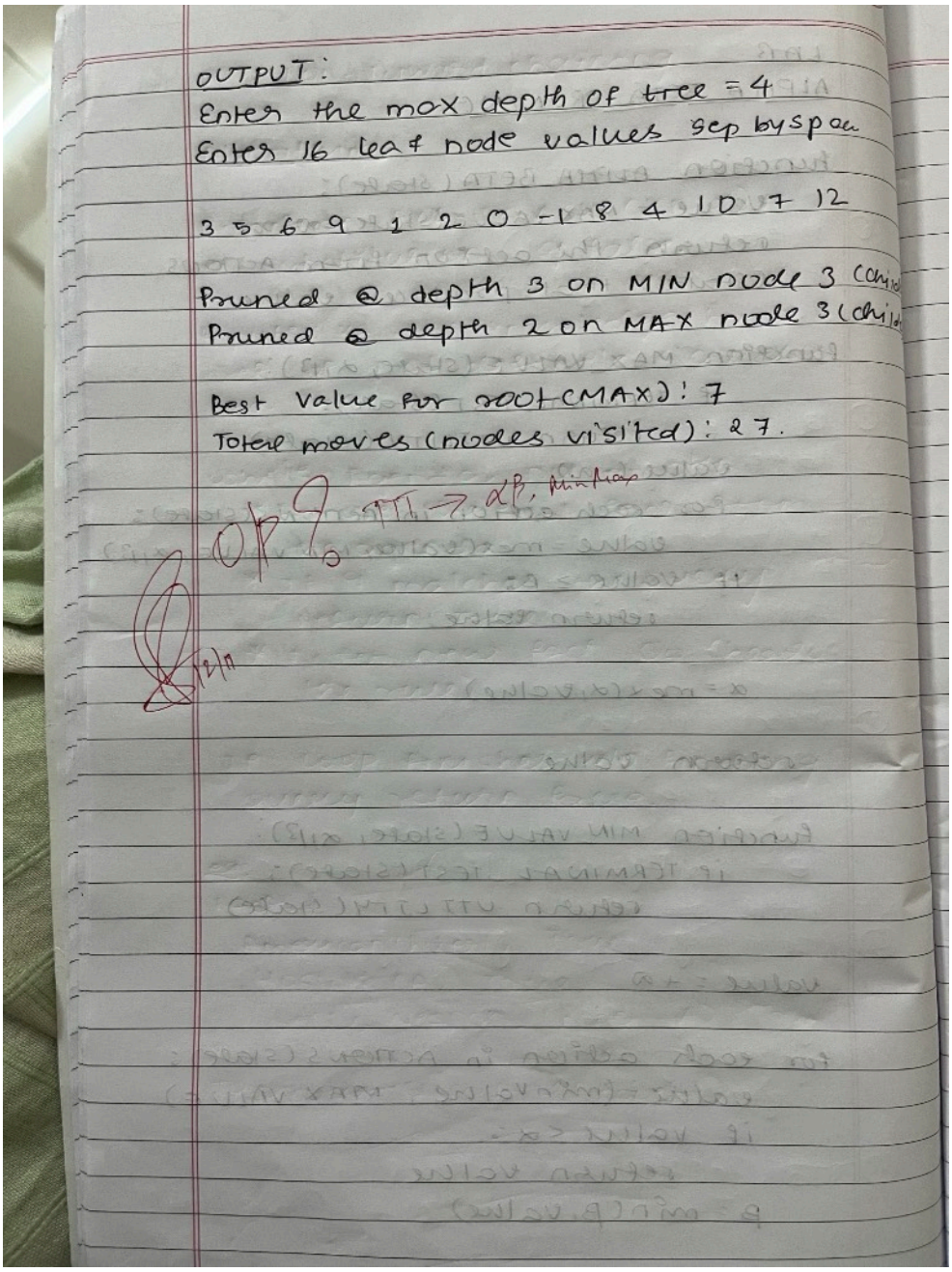
    return value

function MIN_VALUE (state, α, β):
    if TERMINAL_TEST (state):
        return UTILITY (state)

value = +∞

for each action in ACTIONS (state):
    value = (minValue, MAX_VALUE)
    if value ≤ α:
        return value
    β = min (β, value)

return ealue



OUTPUT:
Enter the max depth of tree = 4
Enter 16 leaf node values sep by space

3 5 6 9 1 2 0 -1 8 4 10 7 12

Pruned @ depth 3 on MIN node 3 (chi
Pruned @ depth 2 on MAX node 3 (chil

Best value for root (MAX): 7
Total moves (nodes visited): 27.

OP 9. TTT → αβ, MinMax

12/11

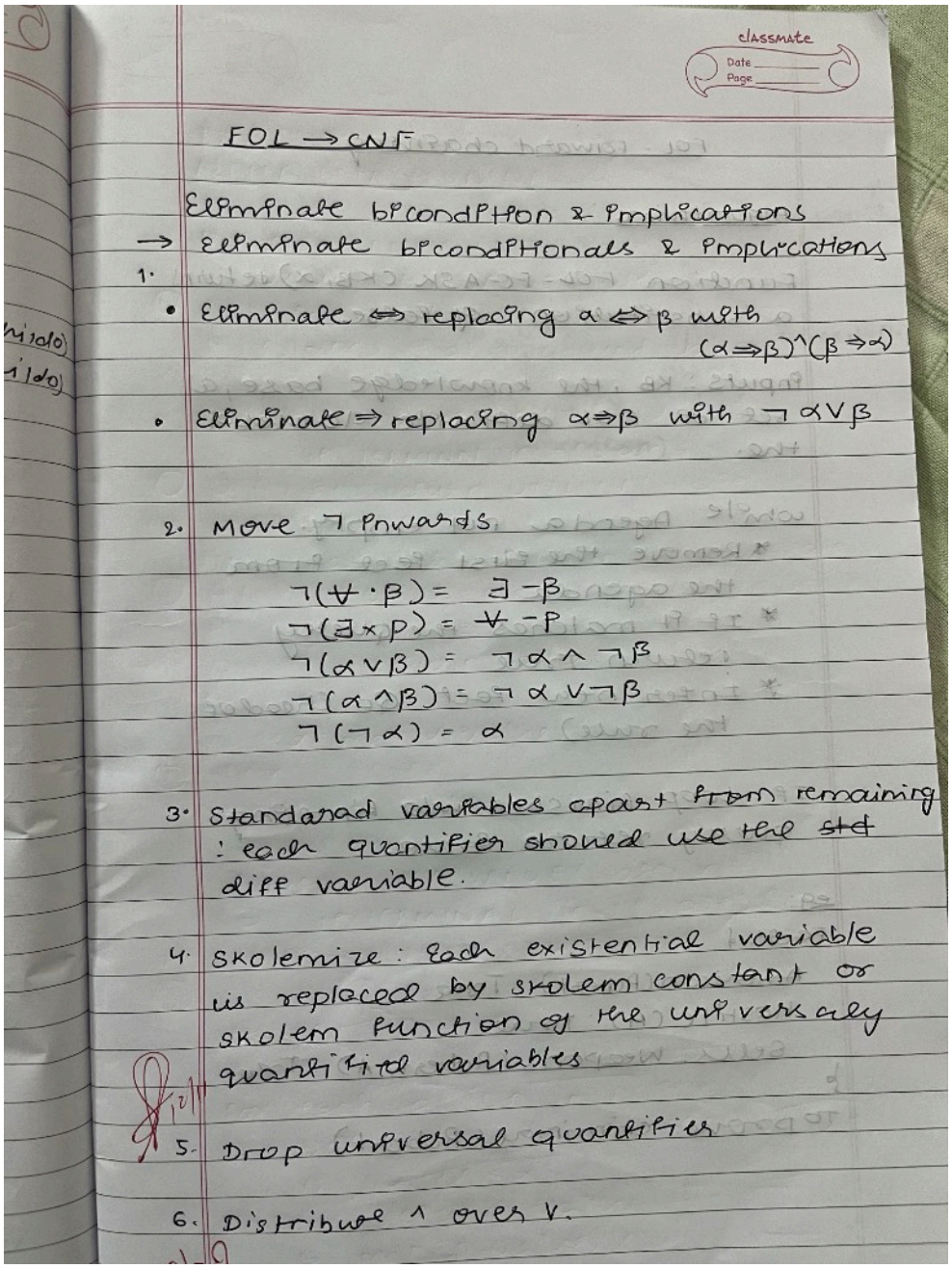
Code:

```
# alpha_beta.py
move_count = 0

def alpha_beta(depth, node_index, is_maximizing, values, alpha, beta, max_depth):
    global move_count
    move_count += 1

    # If we've reached a leaf, return its value
    if depth == max_depth:
        return values[node_index]

    if is_maximizing:
        best = float('-inf')
        for i in range(2):  # binary tree: left (0) then right (1)
            val = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth} on MAX node {node_index} (child {i})")
                break
        return best
    else:
        best = float('inf')
        for i in range(2):
            val = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                print(f"Pruned at depth {depth} on MIN node {node_index} (child {i})")
                break
        return best

def main():
    try:
        max_depth = int(input("Enter the maximum depth of the tree: ").strip())
        if max_depth < 0:
            raise ValueError
    except ValueError:
        print("Error: Please enter a non-negative integer for depth.")
        return

    num_leaves = 2 ** max_depth
    print(f"Enter {num_leaves} leaf node values separated by spaces:")
    try:
        values = list(map(int, input().strip().split()))
    except ValueError:
        print("Error: Please enter integer values for leaves.")
        return

    if len(values) != num_leaves:
        print(f"Error: Number of values does not match 2^{max_depth} = {num_leaves}.")
        return
```

```
    global move_count
    move_count = 0
    best_value = alpha_beta(0, 0, True, values, float('-inf'), float('inf'), max_depth)

    print("\nBest value for root (MAX):", best_value)
    print(f"Total moves (nodes visited): {move_count}")

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

Program 9:Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

Algorithm:



FOL → CNF

Eliminate bicondition & implications

→ Eliminate biconditionals & implications

1.

- Eliminate ⇔ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$
- Eliminate ⇒ replacing $\alpha \Rightarrow \beta$ with $\neg \alpha \vee \beta$

2. Move ¬ inwards.

$\neg(\forall \cdot \beta) = \exists \neg\beta$

$\neg(\exists x\, p) = \forall \neg p$

$\neg(\alpha \vee \beta) = \neg\alpha \wedge \neg\beta$

$\neg(\alpha \wedge \beta) = \neg\alpha \vee \neg\beta$

$\neg(\neg\alpha) = \alpha$

3. Standarad variables apart from remaining : each quantifier should use the std diff variable.

4. Skolemize : Each existential variable is replaced by skolem constant or skolem function of the universaly quantified variables

5. Drop universal quantifier

6. Distribute ∧ over ∨.

Code:

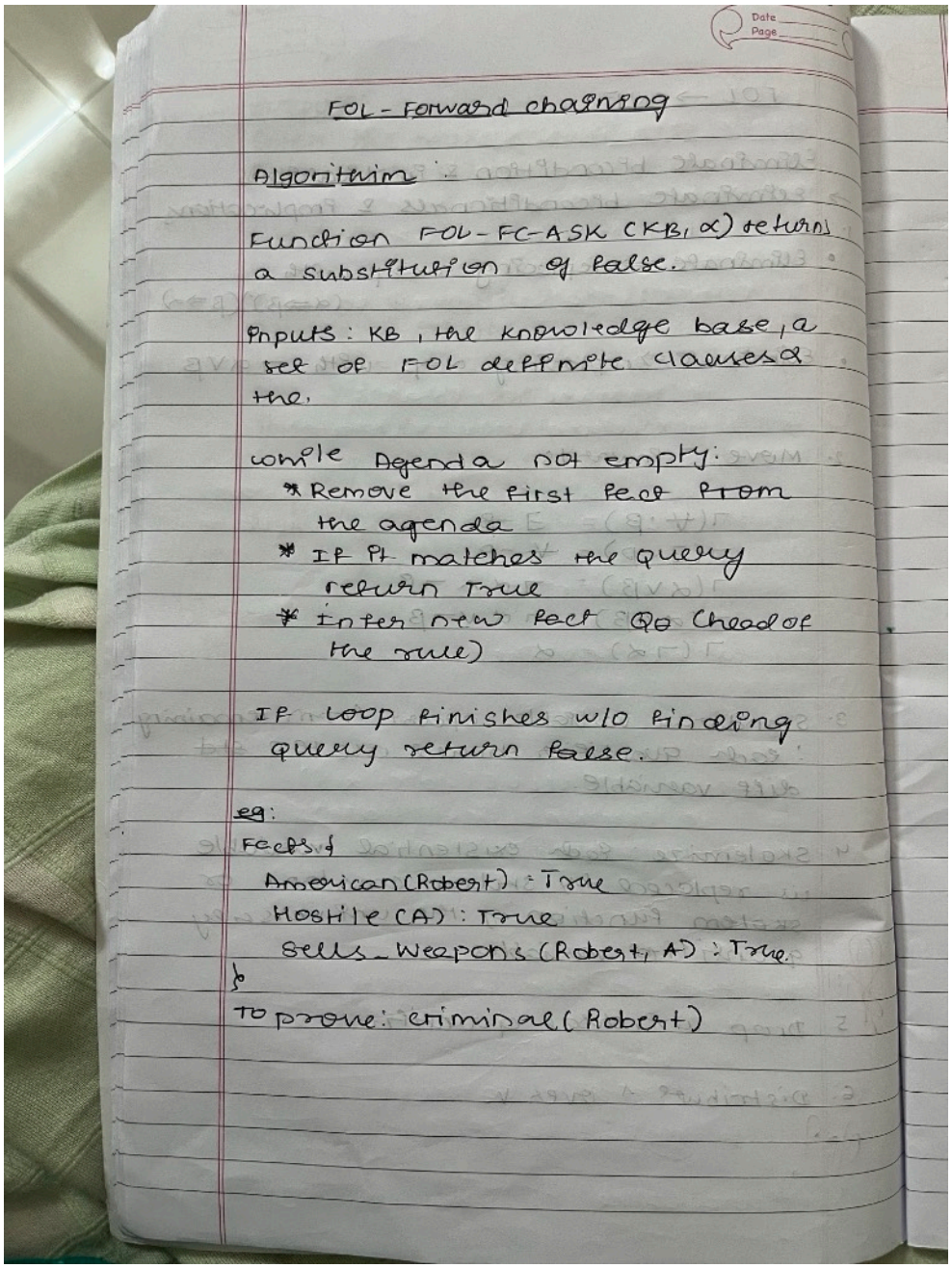
```
import itertools

def remove_implications(expr):
    if isinstance(expr, str):
        return expr

    op = expr[0]

    if op == 'implies':
        return ['or', ['not', remove_implications(expr[1])], remove_implications(expr[2])]

    if op == 'iff':
        A = remove_implications(expr[1])
        B = remove_implications(expr[2])
        return ['and',
            ['or', ['not', A], B],
            ['or', A, ['not', B]]
        ]

    return [op] + [remove_implications(e) for e in expr[1:]]

def move_negation_inward(expr):
    if isinstance(expr, str):
        return expr

    op = expr[0]

    if op == 'not':
        sub = expr[1]
        if isinstance(sub, list):
            if sub[0] == 'not':
                return move_negation_inward(sub[1])
            if sub[0] == 'and':
                return ['or'] + [move_negation_inward(['not', x]) for x in sub[1:]]
            if sub[0] == 'or':
                return ['and'] + [move_negation_inward(['not', x]) for x in sub[1:]]
        return ['not', move_negation_inward(sub)]

    return [op] + [move_negation_inward(e) for e in expr[1:]]

var_counter = itertools.count()

def standardize(expr, mapping=None):
    if mapping is None:
        mapping = {}

    if isinstance(expr, str):
        if expr[0].islower():
            if expr not in mapping:
                mapping[expr] = f"v{next(var_counter)}"
            return mapping[expr]
        return expr
```

```
    return [expr[0]] + [standardize(e, mapping) for e in expr[1:]]

skolem_counter = itertools.count()

def skolemize(expr, context=None):
    if context is None:
        context = []

    if isinstance(expr, str):
        return expr

    op = expr[0]

    if op == 'forall':
        return skolemize(expr[2], context + [expr[1]])

    if op == 'exists':
        var = expr[1]
        name = f"S{next(skolem_counter)}"
        if context:
            return skolemize(replace(expr[2], var, [name] + context), context)
        else:
            return skolemize(replace(expr[2], var, name), context)

    return [op] + [skolemize(e, context) for e in expr[1:]]

def replace(expr, old, new):
    if isinstance(expr, str):
        return new if expr == old else expr
    return [expr[0]] + [replace(e, old, new) for e in expr[1:]]

def drop_quantifiers(expr):
    if isinstance(expr, str):
        return expr
    if expr[0] in ('forall', 'exists'):
        return drop_quantifiers(expr[2])
    return [expr[0]] + [drop_quantifiers(e) for e in expr[1:]]

def distribute(expr):
    if isinstance(expr, str):
        return expr

    if expr[0] == 'or':
        A = distribute(expr[1])
        B = distribute(expr[2])
        if isinstance(A, list) and A[0] == 'and':
            return ['and', distribute(['or', A[1], B]), distribute(['or', A[2], B])]
        if isinstance(B, list) and B[0] == 'and':
            return ['and', distribute(['or', A, B[1]]), distribute(['or', A, B[2]])]
        return ['or', A, B]

    if expr[0] == 'and':
        return ['and'] + [distribute(e) for e in expr[1:]]

    return [expr[0]] + [distribute(e) for e in expr[1:]]
```

```
def fol_to_cnf(expr):
    expr = remove_implications(expr)
    expr = move_negation_inward(expr)
    expr = standardize(expr)
    expr = skolemize(expr)
    expr = drop_quantifiers(expr)
    expr = distribute(expr)
    return expr

formula = ['implies', ['and', 'P(x)', 'Q(x)'], ['exists', 'y', 'R(y)']]
print(fol_to_cnf(formula))
```

Program 10:Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning.

Algorithm:



FOL - Forward chaining

Algorithm:

Function FOL-FC-ASK (KB, α) returns a substitution or false.

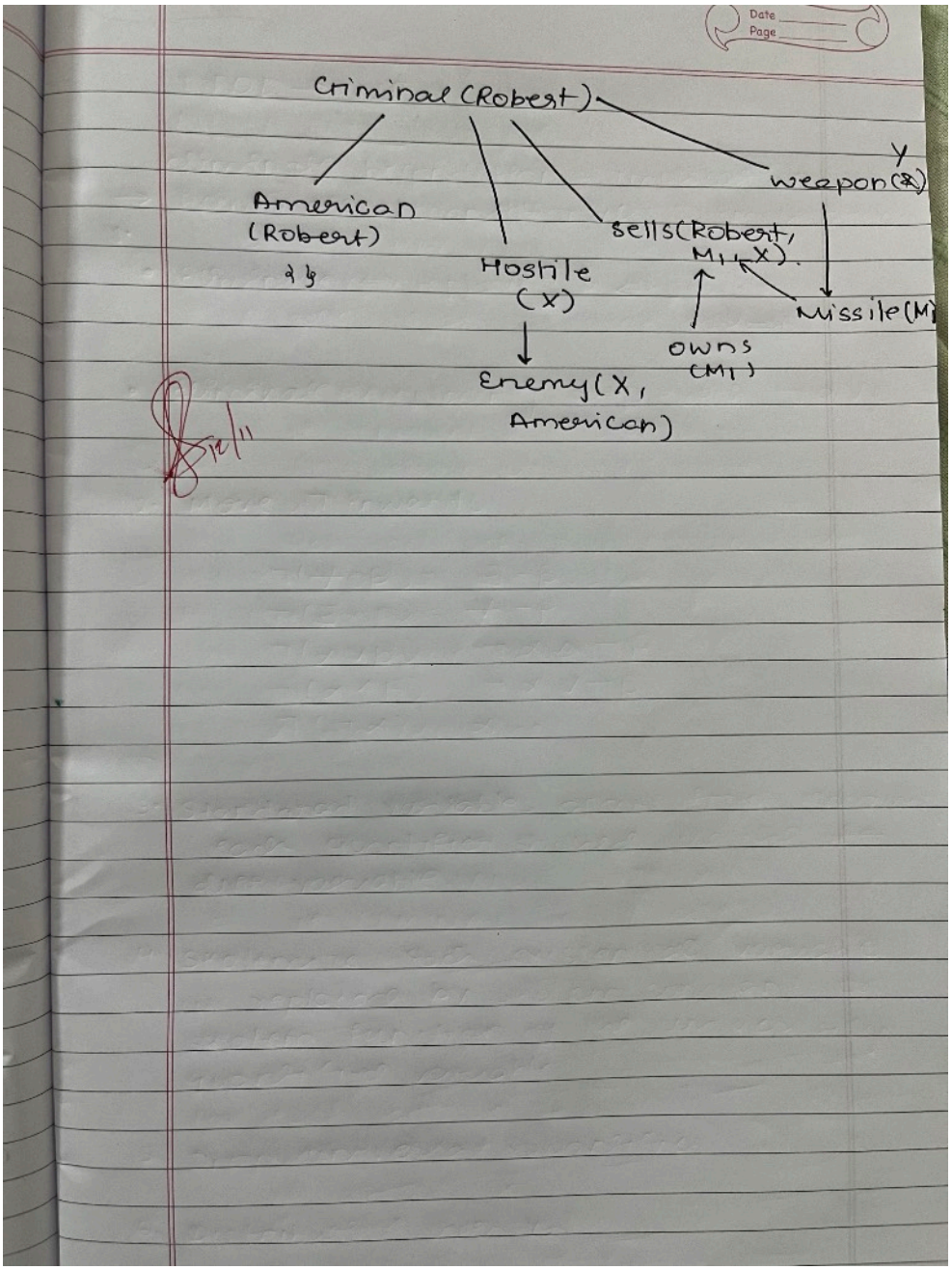
Inputs: KB, the knowledge base, a set of FOL definite clauses
α.

while Agenda not empty:
* Remove the first fact from the agenda
* If it matches the query return True
* Infer new fact to (head of the rule)

If loop finishes w/o finding query return false.

eg:
Facts:
American(Robert) : True
Hostile(A) : True
Sells_Weapons(Robert, A) : True.

To prove: criminal(Robert)


Criminal (Robert)
American (Robert)
Hostile (X)
sells (Robert, M1, X)
weapon (Y)
Missile (M1)
owns (M1)
Enemy (X, American)

Code:

```
def fol_forward_chaining(kb, query):
    inferred = set()
    agenda = []
    count = {}

    for rule in kb:
        if isinstance(rule, tuple):
            premises, conclusion = rule
            count[rule] = len(premises)
            for p in premises:
                if p in kb:
                    agenda.append(p)
        else:
            agenda.append(rule)

    while agenda:
        p = agenda.pop(0)
        if p == query:
            return True
        if p not in inferred:
            inferred.add(p)
            for rule in kb:
                if isinstance(rule, tuple):
                    premises, conclusion = rule
                    if p in premises:
                        count[rule] -= 1
                        if count[rule] == 0:
                            if conclusion == query:
                                return True
                            agenda.append(conclusion)
    return False

kb = [
    "A",
    (["A"], "B"),
    (["B"], "C"),
    (["C"], "D")
]

print(fol_forward_chaining(kb, "D"))
```

Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Ritesh Mohan Nayak

for successfully completing

Principles of Generative AI Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com