VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Chitrashree K (1BM23CS081)

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 **Chitrashree K (1BM23CS081),** 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.

Sheetal V Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

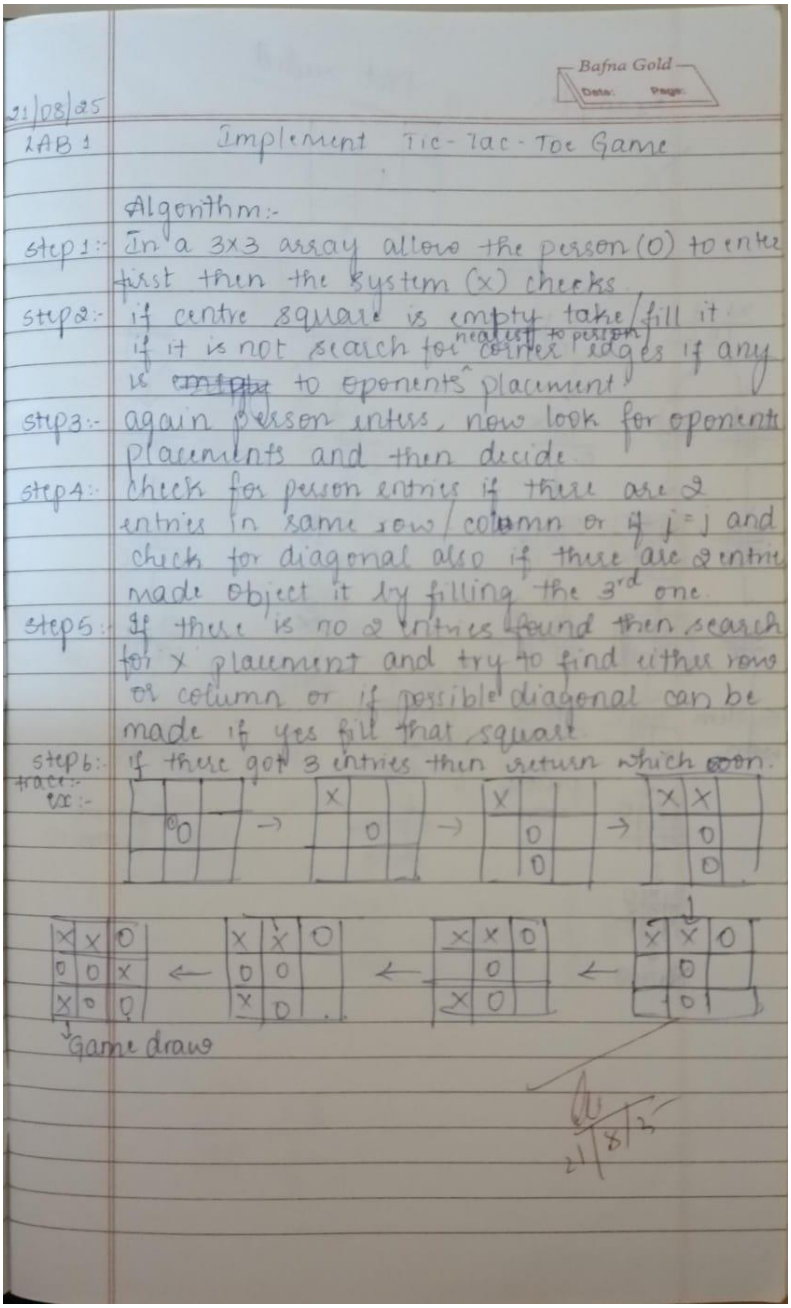
Index

Github Link:
https://github.com/Chitrashree-tech/AI

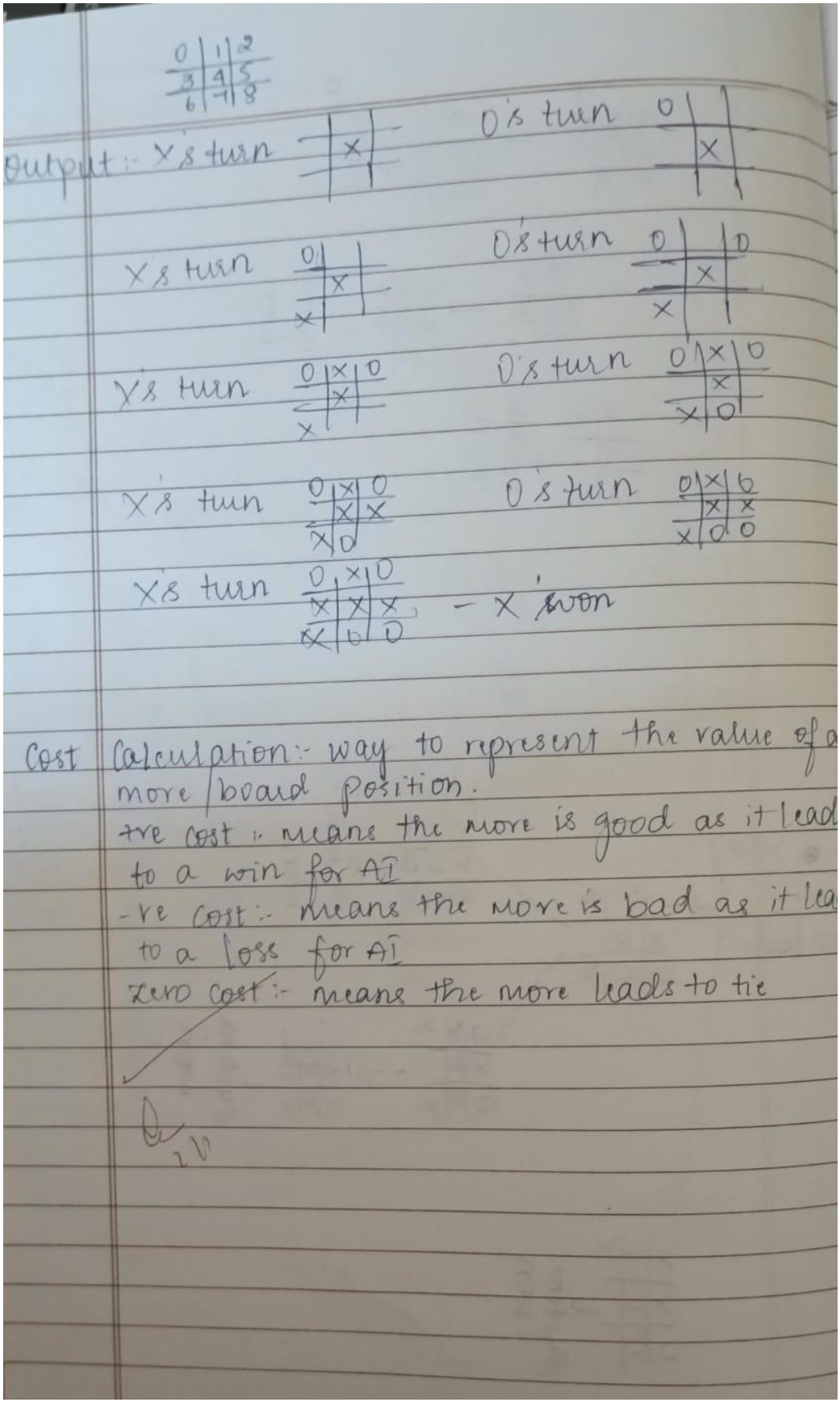
Program 1

Implement Tic –Tac –Toe Game

Algorithm



Bafna Gold
Date:
Page:
Human
Human
Human
Human
Human
Human
Human
Human



Cost Calculation :- way to represent the value of a move/board position.

+ve cost :- means the move is good as it leads to a win for AI

-ve cost :- means the move is bad as it leads to a loss for AI

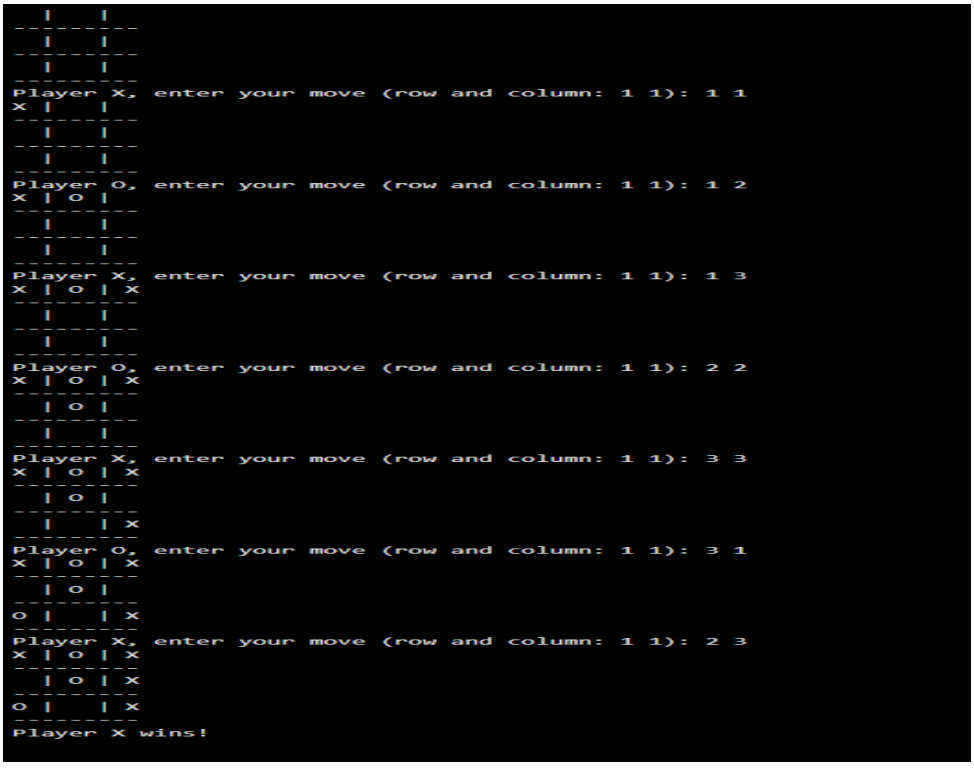
zero cost :- means the move leads to tie

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)
```

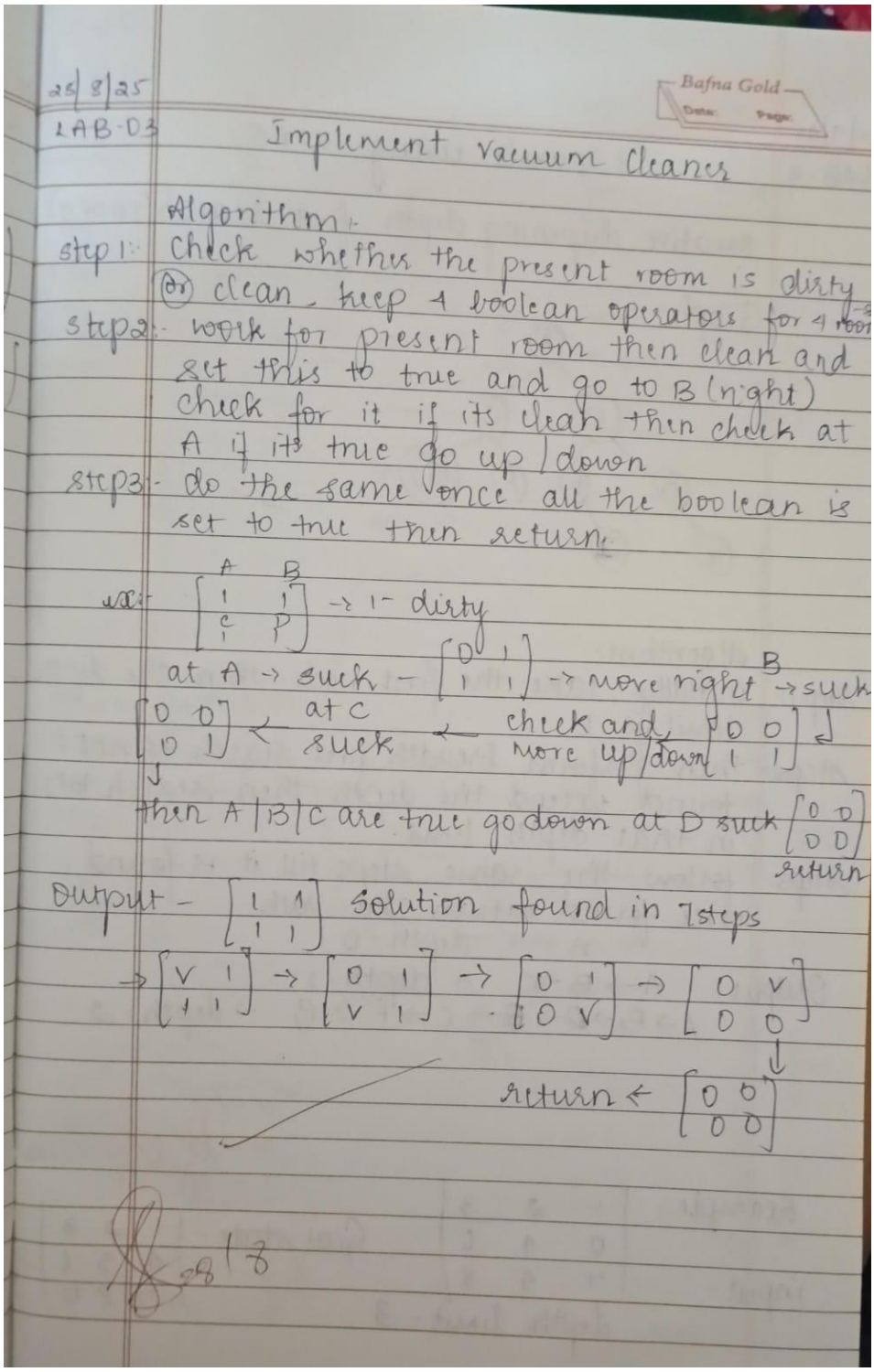
```
        if board[row][col] != " ":
            print("That spot is taken. Try again.")
            continue

        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:



Implement Vacuum Cleaner Agent

Algorithm

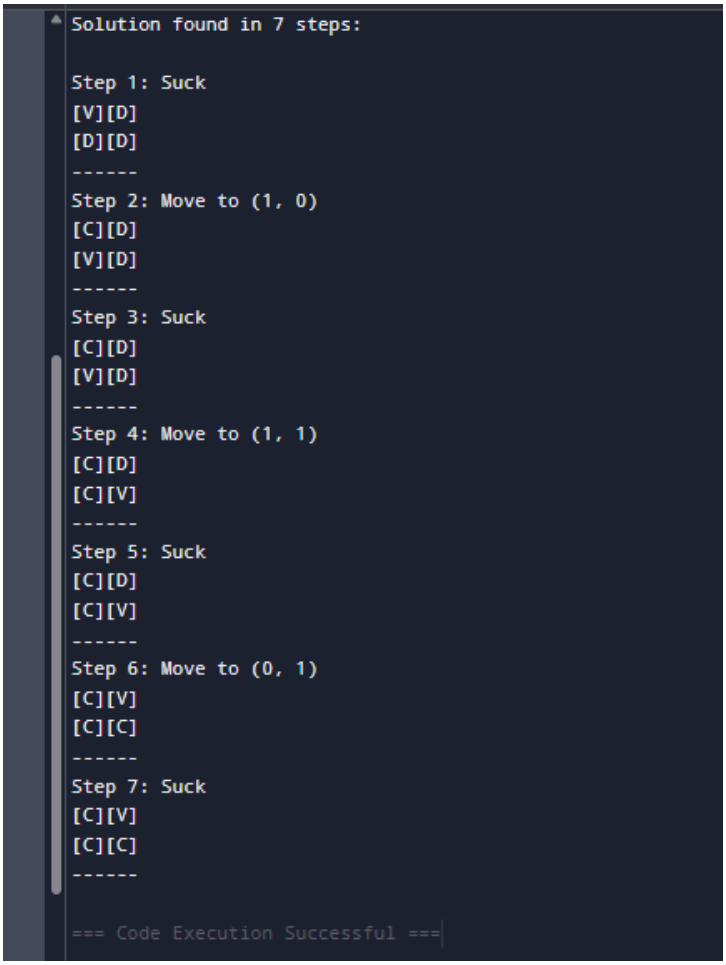


Code:

```
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()
```

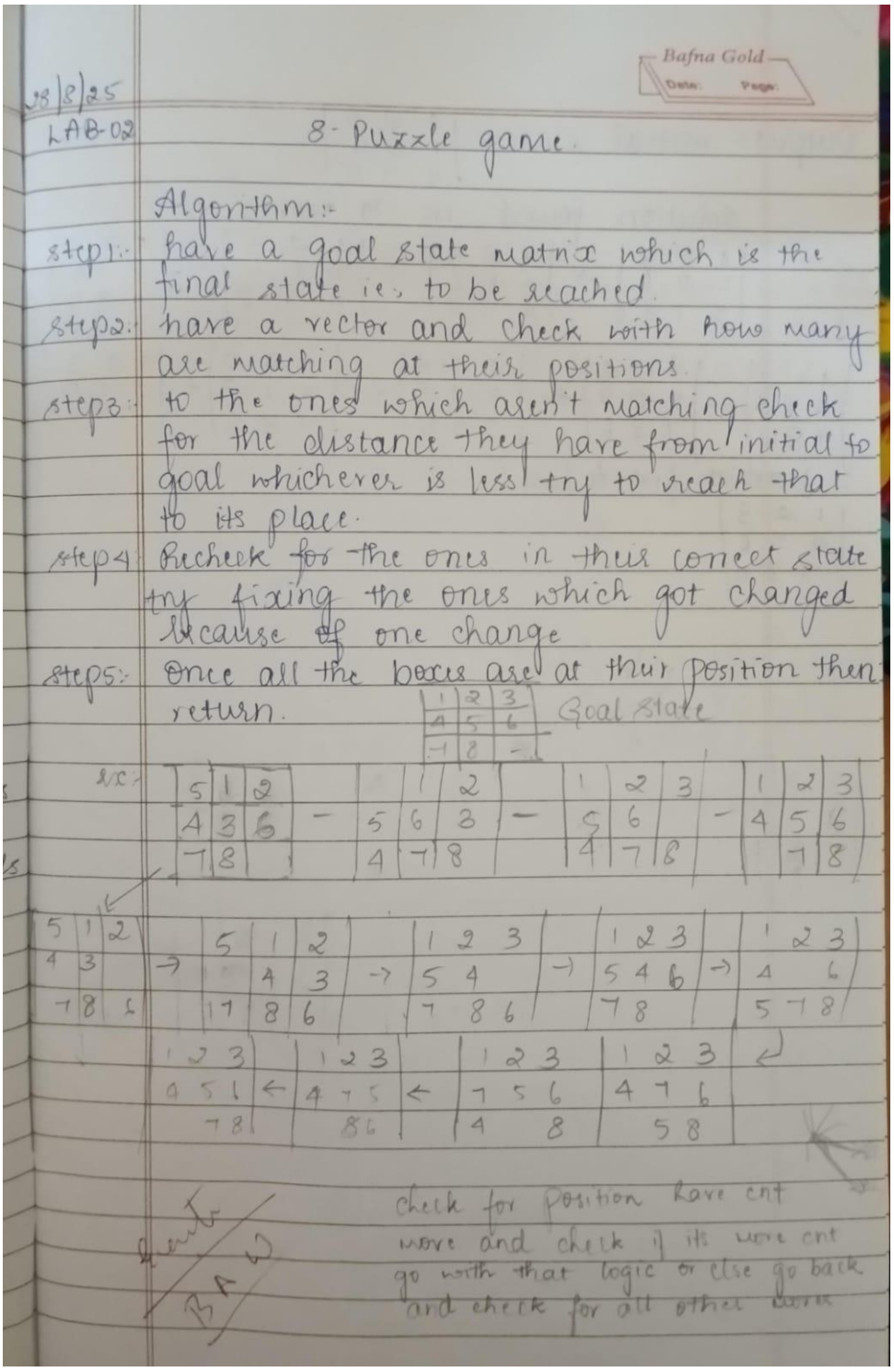
OUTPUT

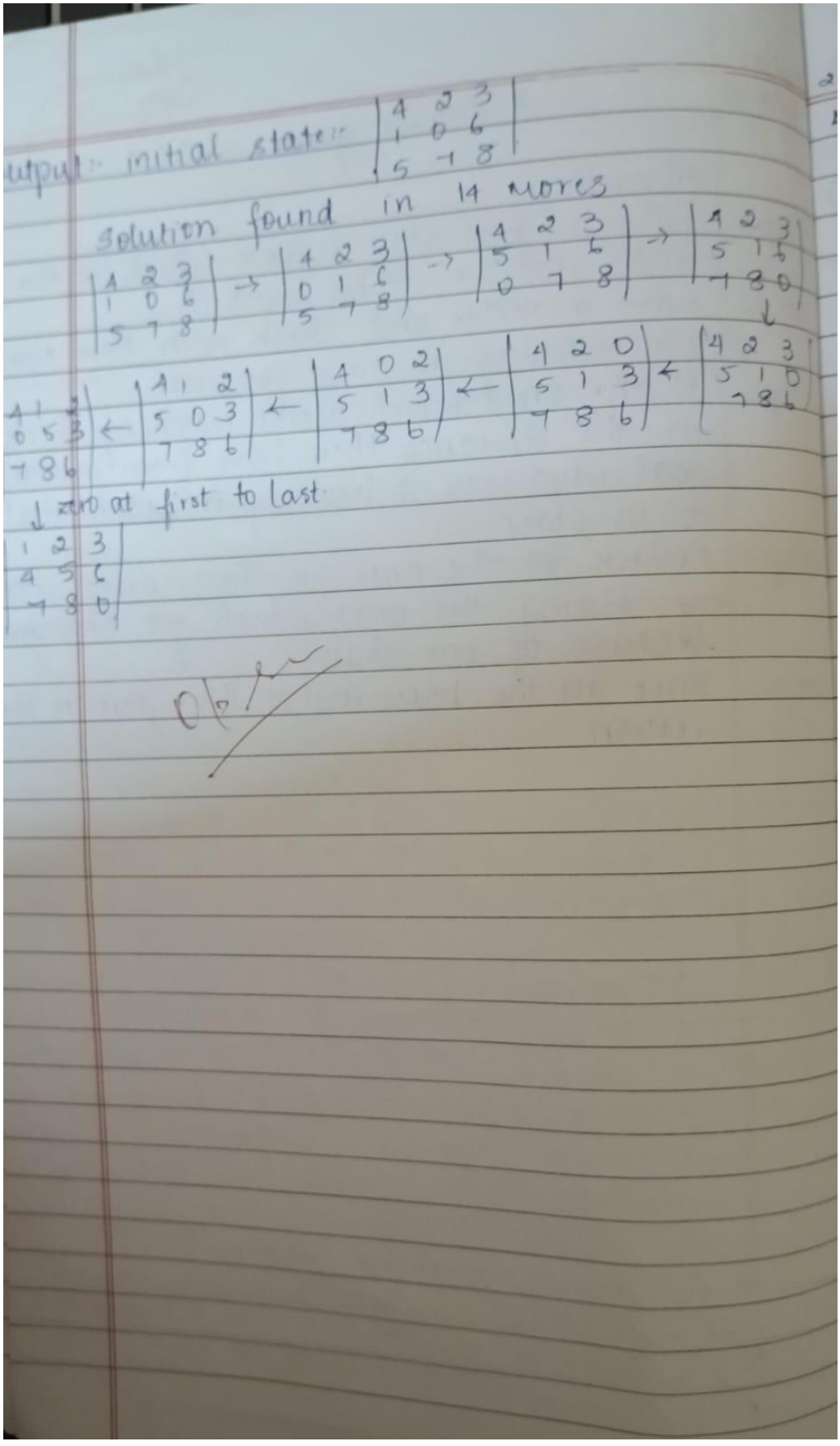


Program 2

Implement 8 puzzle problems using Depth First Search (DFS)

Algorithm




utput: initial state:
solution found in 14 moves
zero at first to last

Code:

Using DFS 8 puzzle 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
```

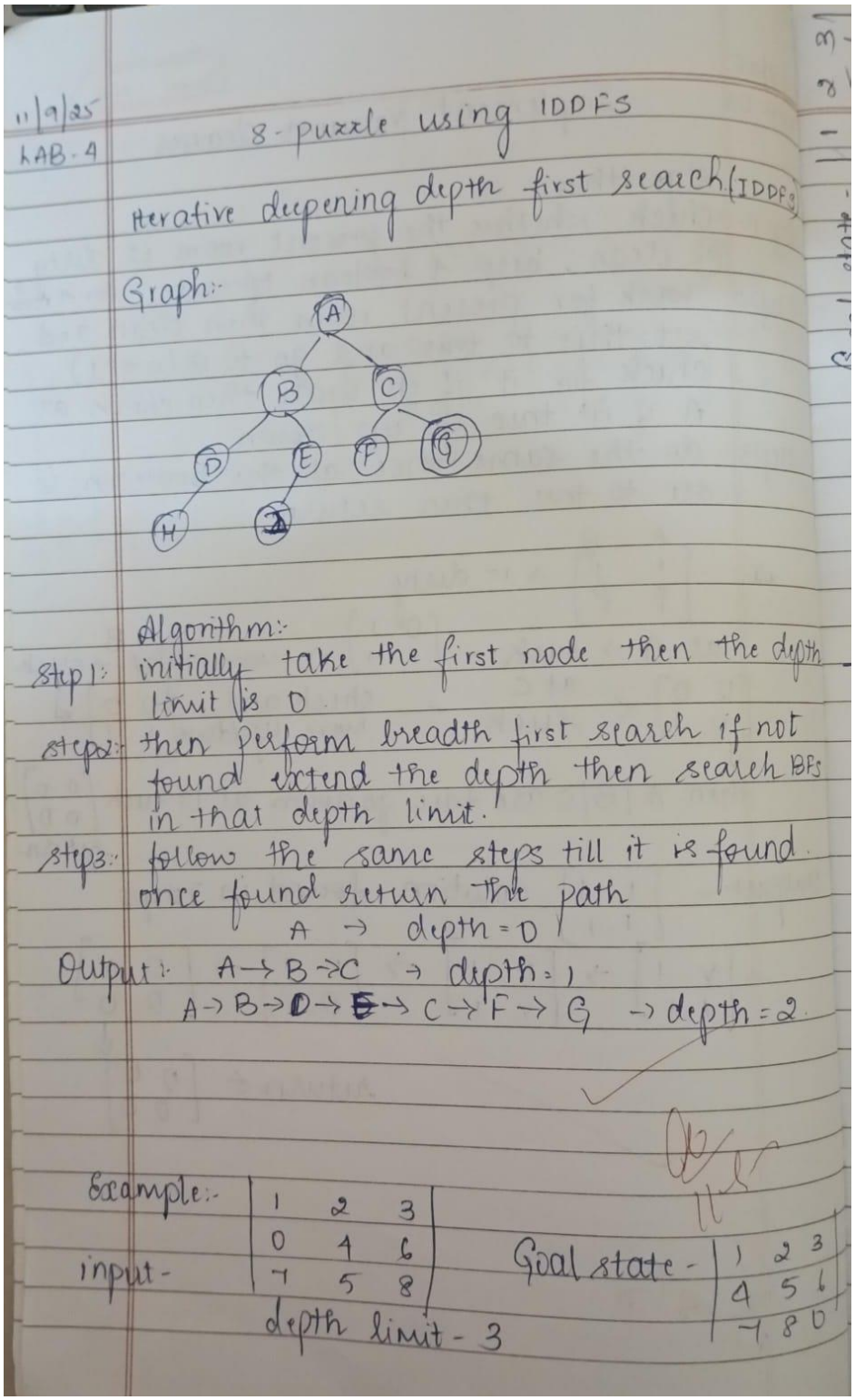
```
# Example start state
start = ((2, 8, 3),
         (1, 6, 4),
         (7, 0, 5))

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.")
```

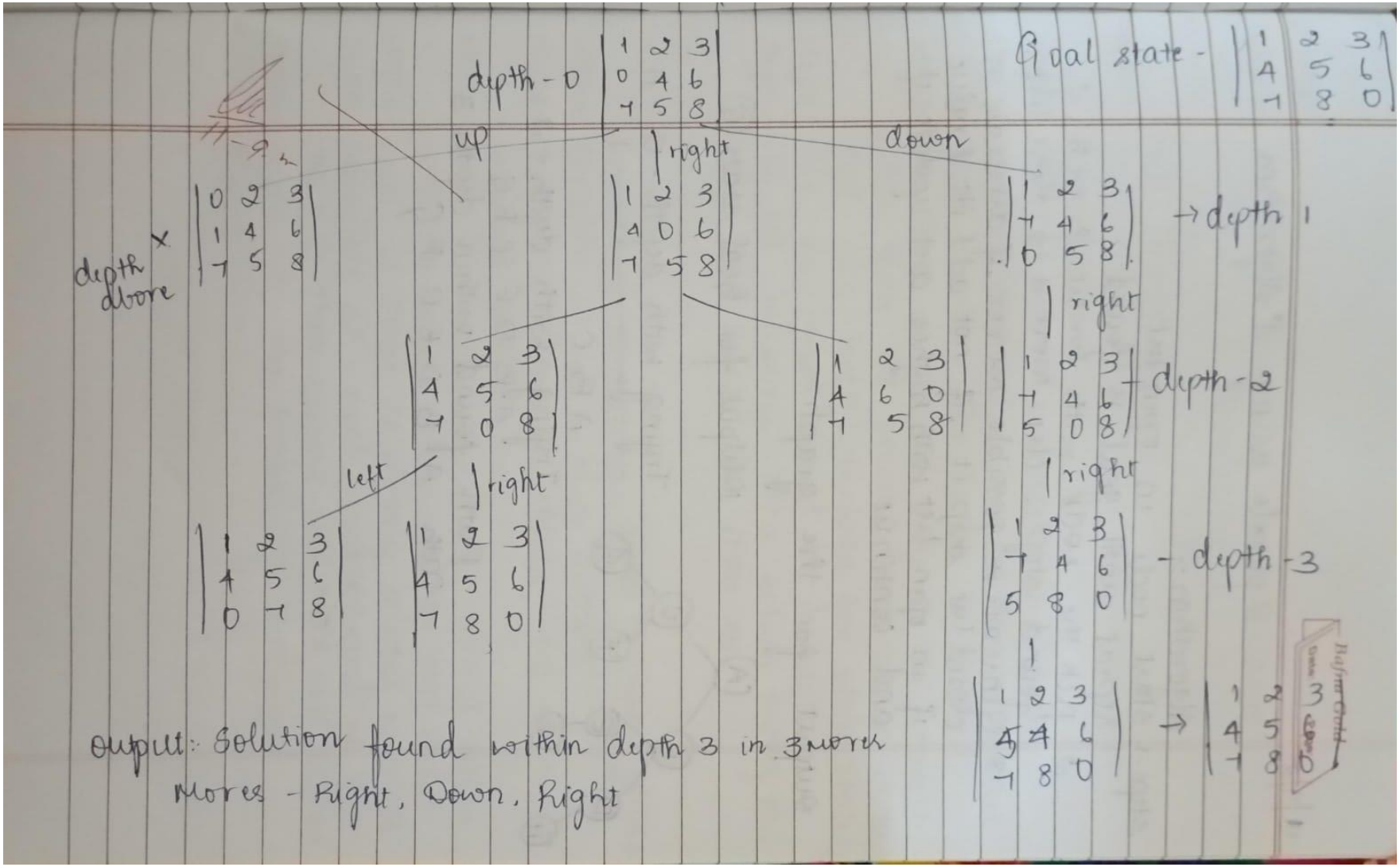
OUTPUT

```
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)

Algorithm:




depth - 0
Goal state -
up
right
down
depth above
left
right
right
right
depth - 2
depth - 3
→ depth 1
Output: Solution found within depth 3 in 3 moves
Moves - Right, Down, Right

Code:

```
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.")
```

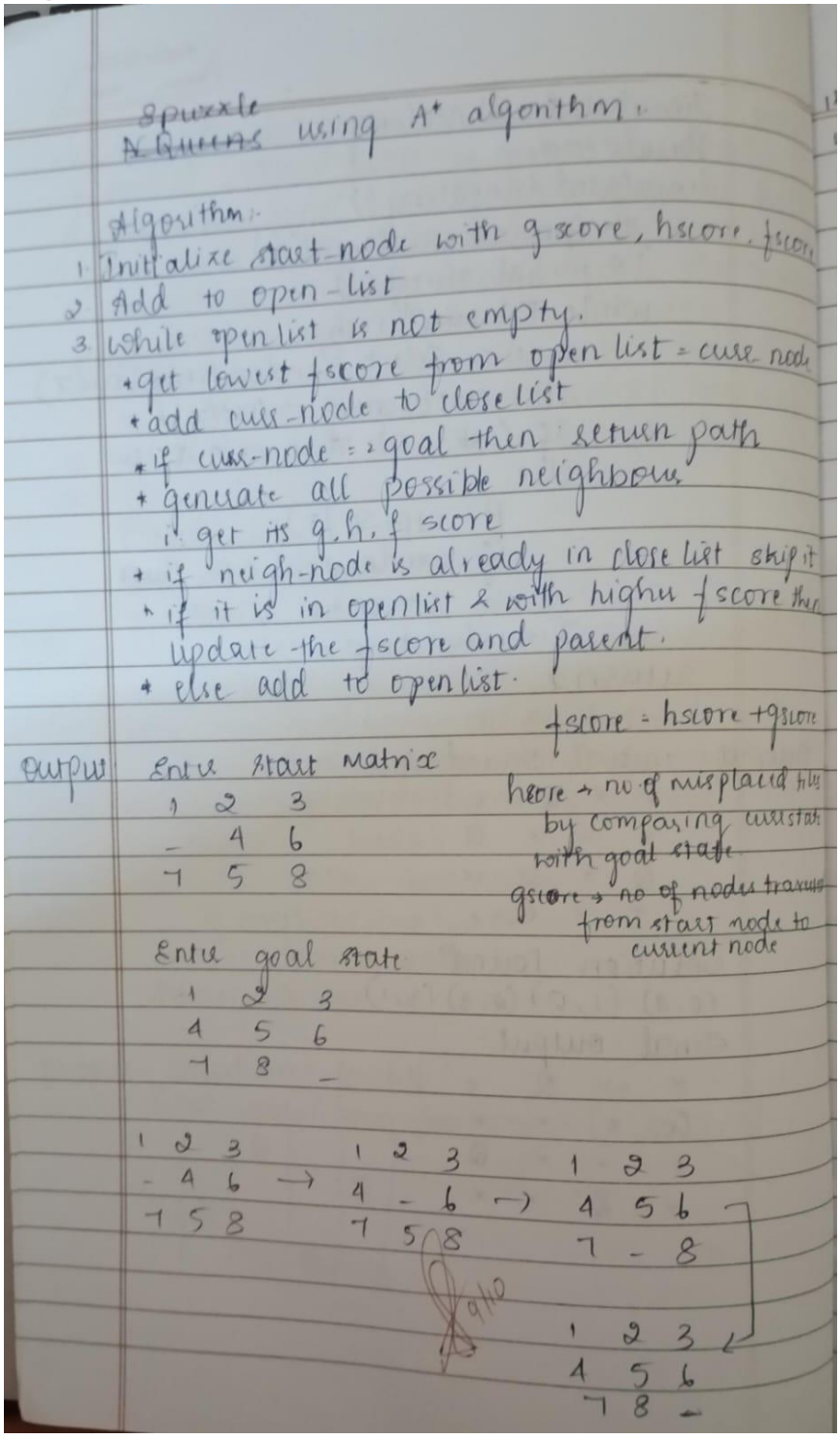
OUTPUT

```
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:



Code:

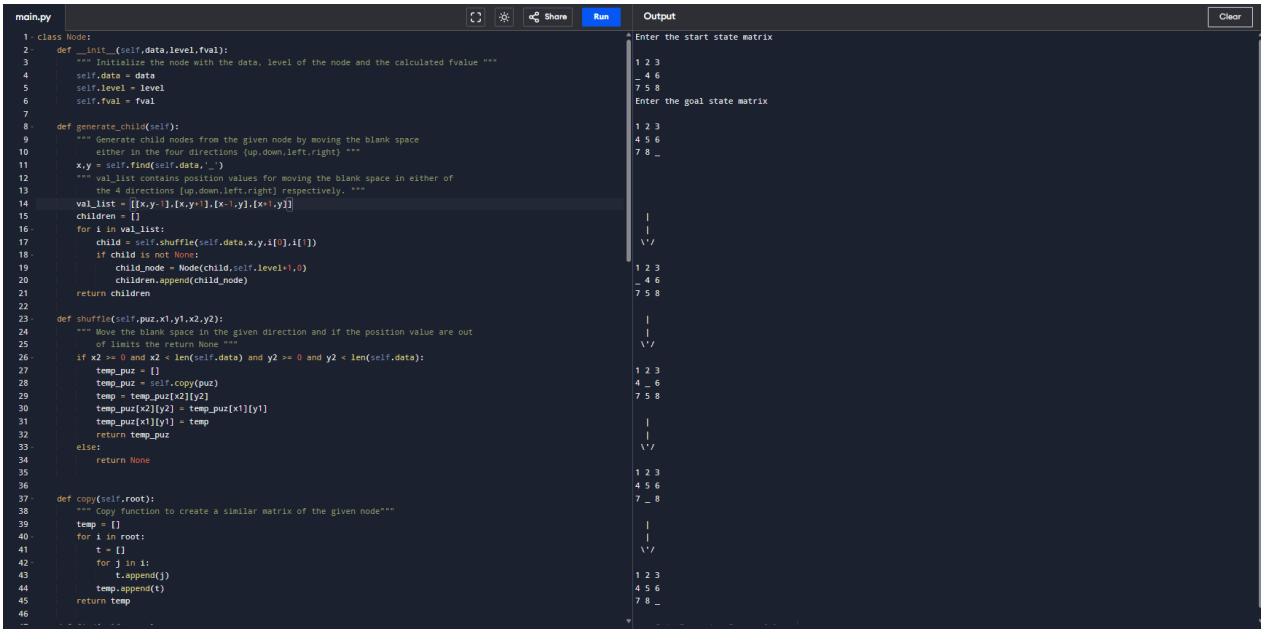
Misplace Tiles

```
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



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

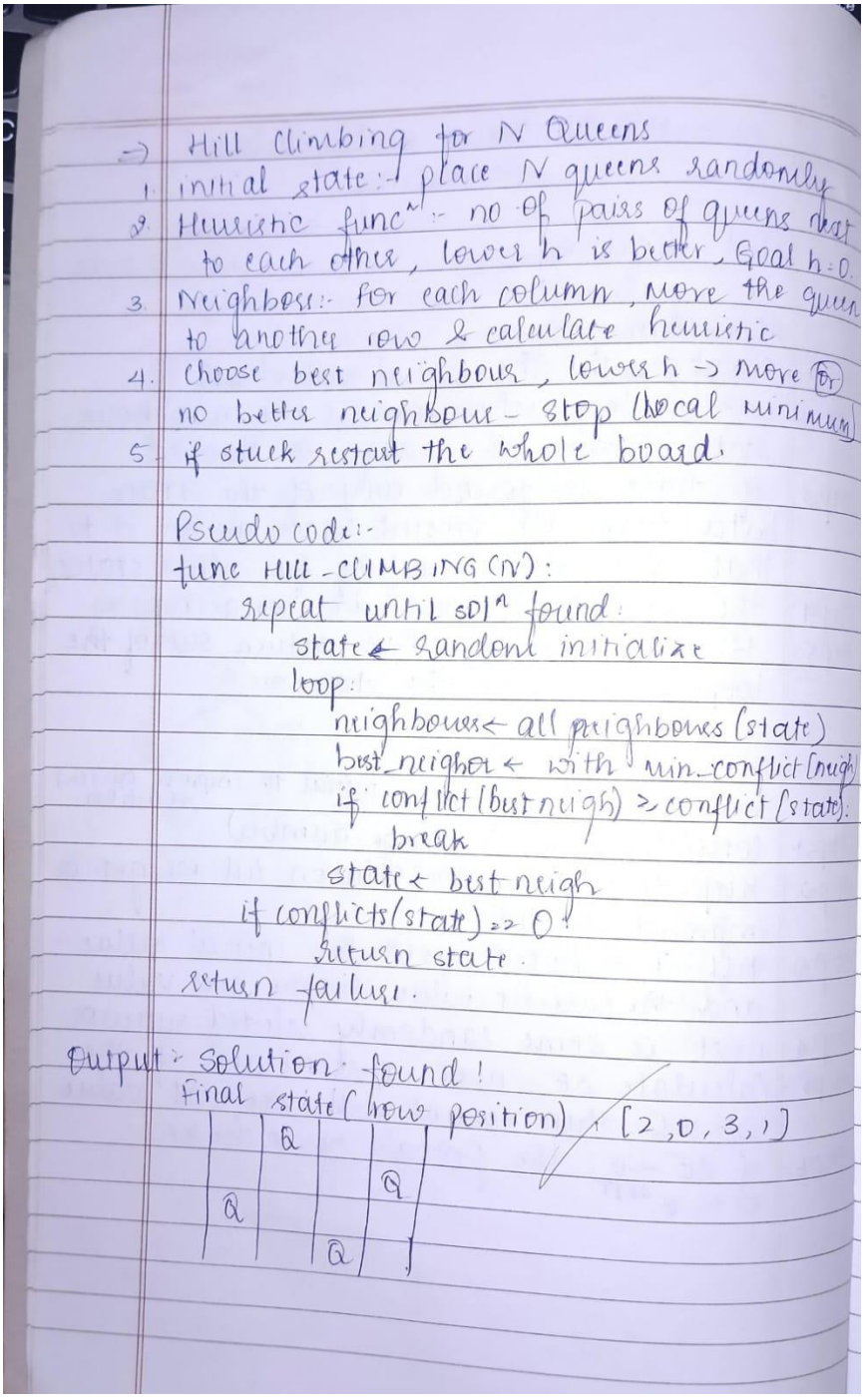
OUTPUT:

```
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:



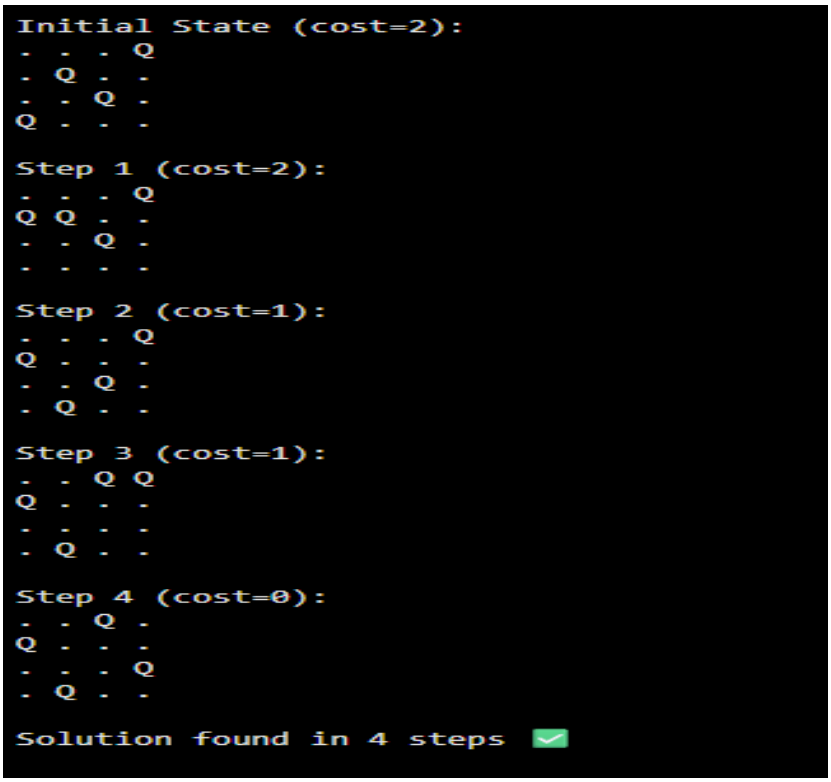
Code:

```
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)
```

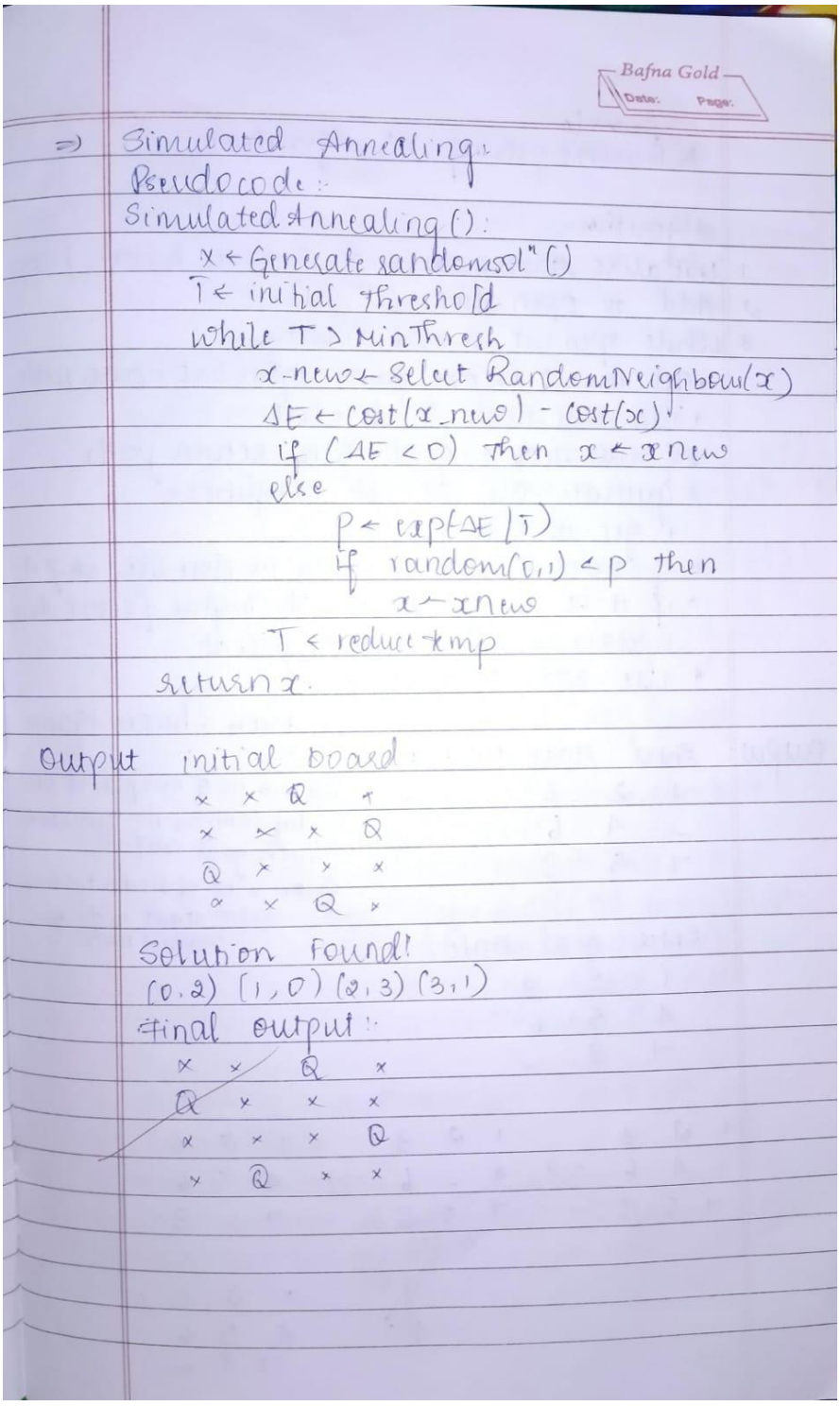
OUTPUT:



Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



=> Simulated Annealing:

Pseudocode:

```
Simulated Annealing():
    x ← Generate randomsol"()
    T ← initial threshold
    while T > MinThresh
        x_new ← Select RandomNeighbour(x)
        ΔE ← cost(x_new) - cost(x)
        if (ΔE < 0) then x ← xnew
        else
            P ← exp(ΔE/T)
            if random(0,1) < p then
                x ← xnew
        T ← reduce temp
    return x.
```

Output initial board:

× × Q ×
× × × Q
Q × × ×
× × Q ×

Solution found!
(0,2) (1,0) (2,3) (3,1)

Final output:

× × Q ×
Q × × ×
× × × Q
× Q × ×

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:

main.py

Share Run

```
    def run(self):
        board = self.board
        board_queens = self.board.queens[:]
        solutionFound = False

        for k in range(170000):
            self.temperature *= self.sch
            board.reset()
            successor_queens = board.queens[:]

            dw = (Board.calculateCostWithQueens(successor_queens) -
                  Board.calculateCostWithQueens(board_queens))

            exp = decimal.Decimal(math.e) ** (decimal.Decimal(-dw) * decimal.Decimal(self
                .temperature))

            if dw > 0 or random.uniform(0, 1) < exp:
                board_queens = successor_queens[:]

            if Board.calculateCostWithQueens(board_queens) == 0:
                print("✅ Solution Found!")
                print(Board.toString(board_queens))
                print("\nChessboard:\n")
                Board.printBoard(board_queens)
                self.elapsedTime = self.getElapsedTime()
                print("Success, Elapsed Time: %sms" % str(self.elapsedTime))
                solutionFound = True
                break

        if not solutionFound:
            self.elapsedTime = self.getElapsedTime()
            print("❌ No solution found, Elapsed Time: %sms" % str(self.elapsedTime))

        return self.elapsedTime
```

Output

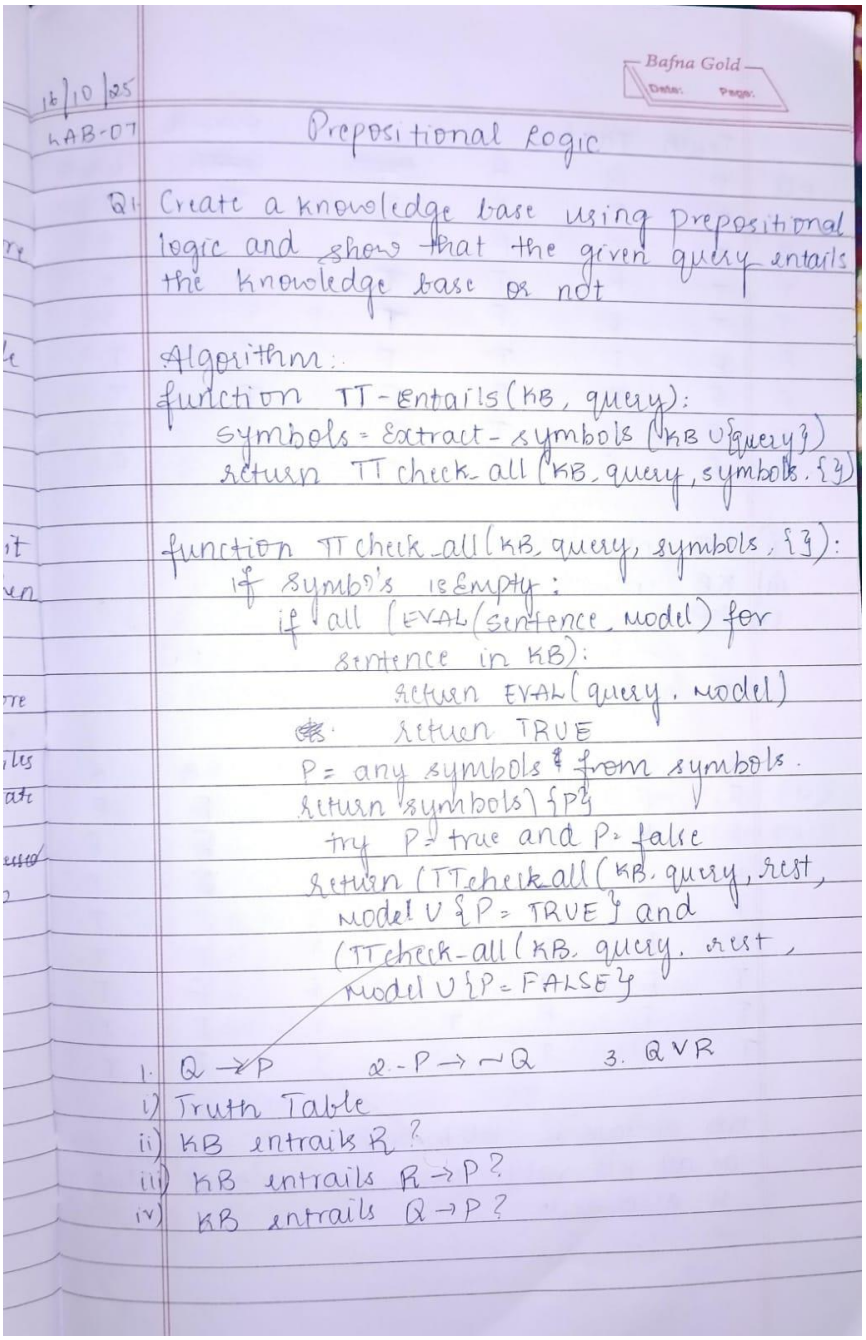
Clear

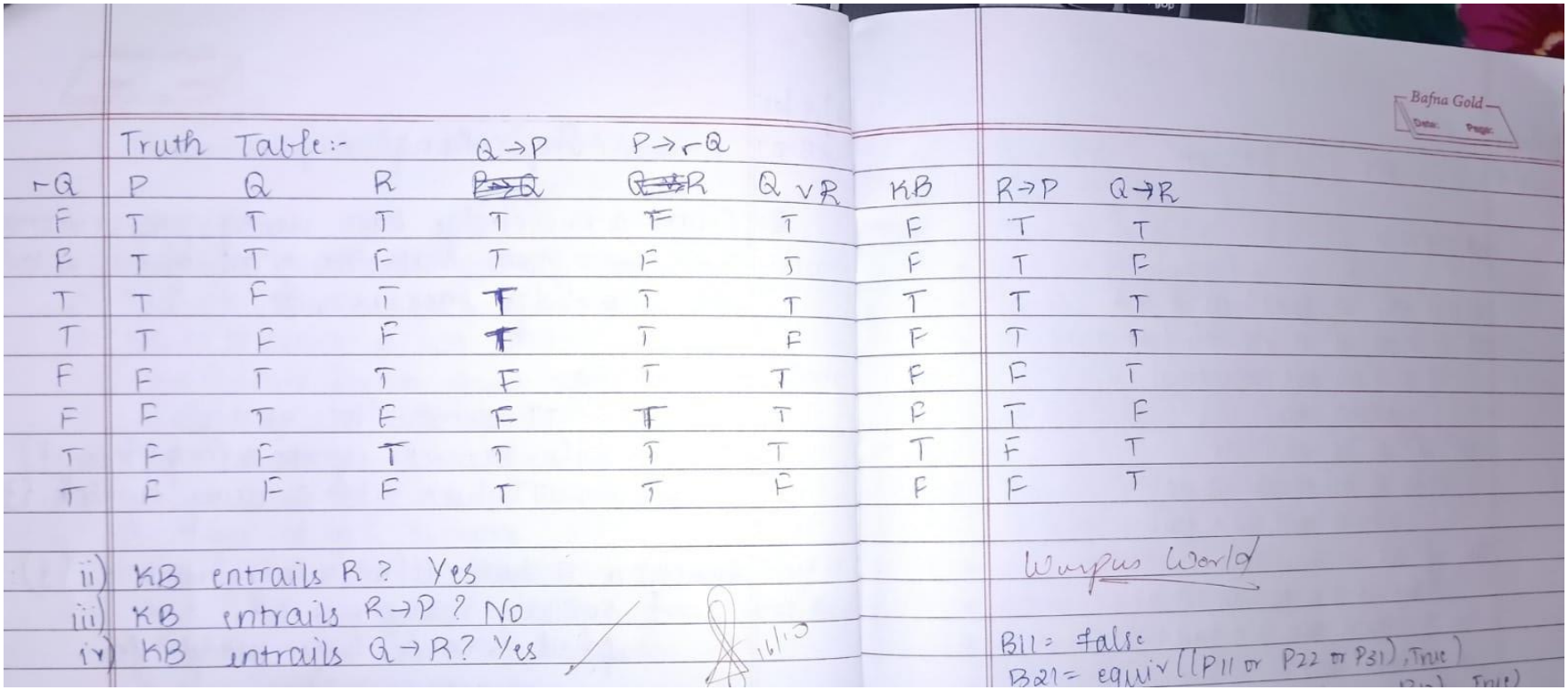
```
Initial Board (random):
 .  Q  .  .
 .  Q  .  .
 .  .  .  Q
 .  .  .  Q


✅ Solution Found!
(0, 1)
(1, 3)
(2, 0)
(3, 2)

Chessboard:

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


Success, Elapsed Time: 3.736ms

=== Code Execution Successful ===
```

Program 6

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

Algorithm:





Truth Table:-

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

ii) KB entrails R ? Yes
iii) KB entrails R→P ? No
iv) KB entrails Q→R ? Yes

Wumpus World

B11 = false
B21 = equiv([P11 or P22 or P31), True)

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

A	B	C	A∨C	B∧¬C	KB	α
F	F	F	F	T	F	F
F	F	T	T	F	F	F
F	T	F	F	T	F	T
F	T	T	T	T	T	T
T	F	F	T	T	T	T
T	F	T	T	F	F	T
T	T	F	T	T	T	T
T	T	T	T	T	T	T

KB entrails α (KB ⊨ A∨B) :.
at all KB value true, the α (A∨B) value is also true.

B21 = equiv (¬P11 or P22 or P31), True)
B12 = equiv(¬P11 or P22 or P12), True)

KB = (not P11)
KB = KB and equiv (B12, (P11 or P22 or P12))
KB = KB and equiv(B21, (P11 or P22 or P31))

S		B	P
W	S B	P	B
S		B	
start	B	P	

square to check - P31
output: the square may have PIT

DA
16-10-25

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")
```

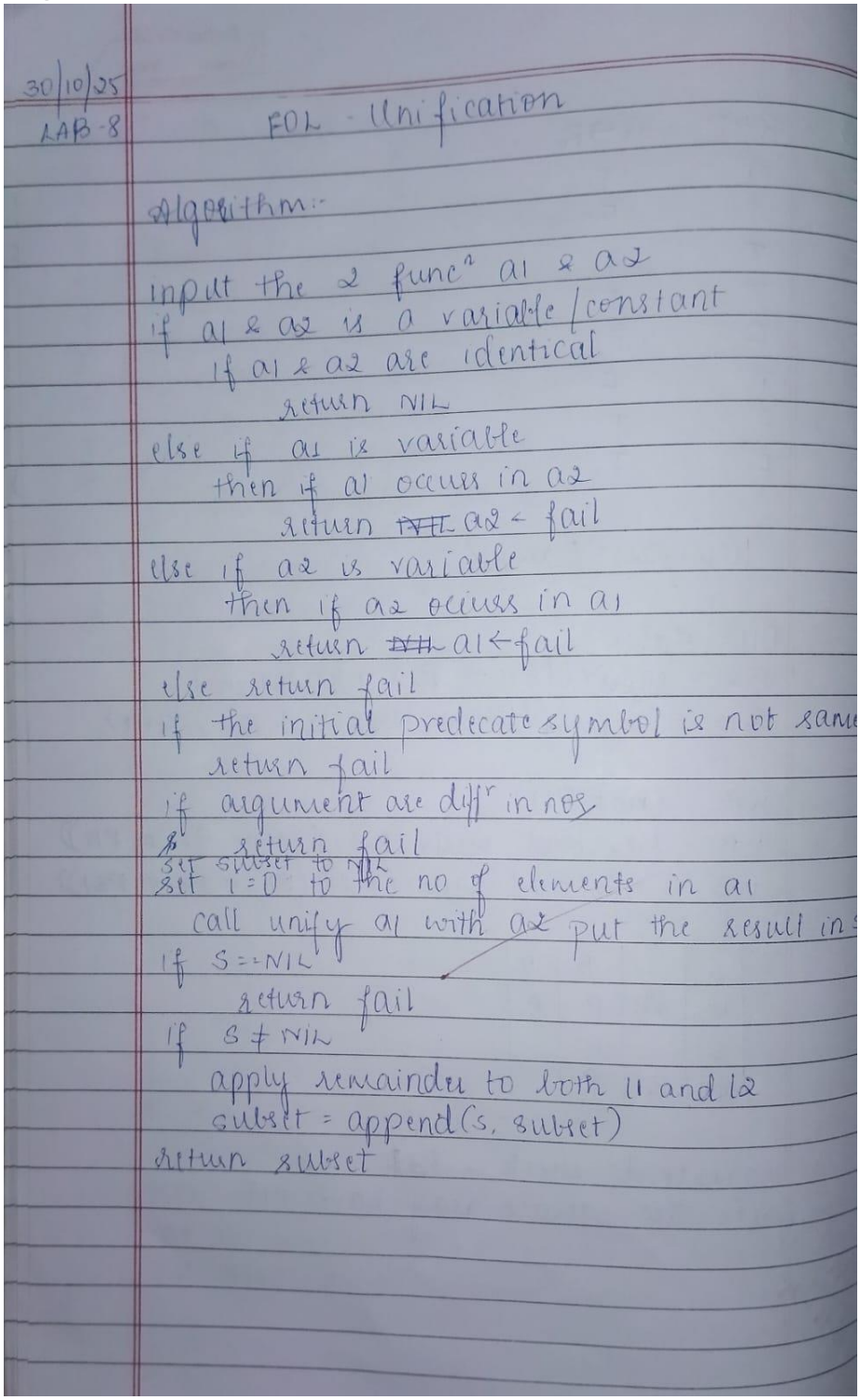
OUTPUT:

```
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:



Bafna Gold
Date: Page:

Questions:-

1. $P(f(a), g(y), y)$
   $P(f(g(x)), g(f(a)), f(a))$
   * predicate symbol is same
   * same no of arguments.
   * unification:-
     $x \to g(x)$; $y \to f(a)$
     $\theta = \{x/g(x), y/f(a)\}$ can be unified. x is free variable

2. $Q(x, f(x))$
   $Q(f(y), y)$
   * predicate symbol is same
   * same no of arguments
   * unification:-
     $x \to f(y)$; $f(x) \Rightarrow f(f(y)) \neq y$
     ~~$\theta = \{x/f(y), f(a)/y\}$~~ y occurs both sides.
     cannot be unified.

3. $P(x, g(x))$
   $P(g(y), g(g(x)))$
   * predicate symbol is same
   * same no of arguments
   * unification:-
     $x \to g(y)$; $g(x) = g(g(y)) = g(g(x))$ if $y = x$
     $y \to x$ $x \to y$.
     $\theta = \{x/g(y), y \to x\}$
     $\{x/g(x), y \to x\}$

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:

main.py

Share Run

```
    elif x in substitutions:
        return unify(var, substitutions[x], substitutions)
    elif occurs_check(var, x, substitutions):
        return None
    else:
        substitutions[var] = x
        return substitutions


def occurs_check(var, x, substitutions):
    if var == x:
        return True
    elif isinstance(x, (list, tuple)):
        return any(occurs_check(var, xi, substitutions) for xi in x)
    elif x in substitutions:
        return occurs_check(var, substitutions[x], substitutions)
    return False


def is_variable(term):
    # variable is a lowercase string
    return isinstance(term, str) and term[0].islower()


# ------------------------------------------------
# ✅ Example usage (you can replace with your own)
# ------------------------------------------------
x1 = ['P', 'f(x)', 'g(y)', 'y']
x2 = ['P', 'f(g(z))', 'g(f(a))', 'f(a)']

result = unify(x1, x2)
if result:
    print("Unification Successful! Substitutions:", result)
else:
```

Output

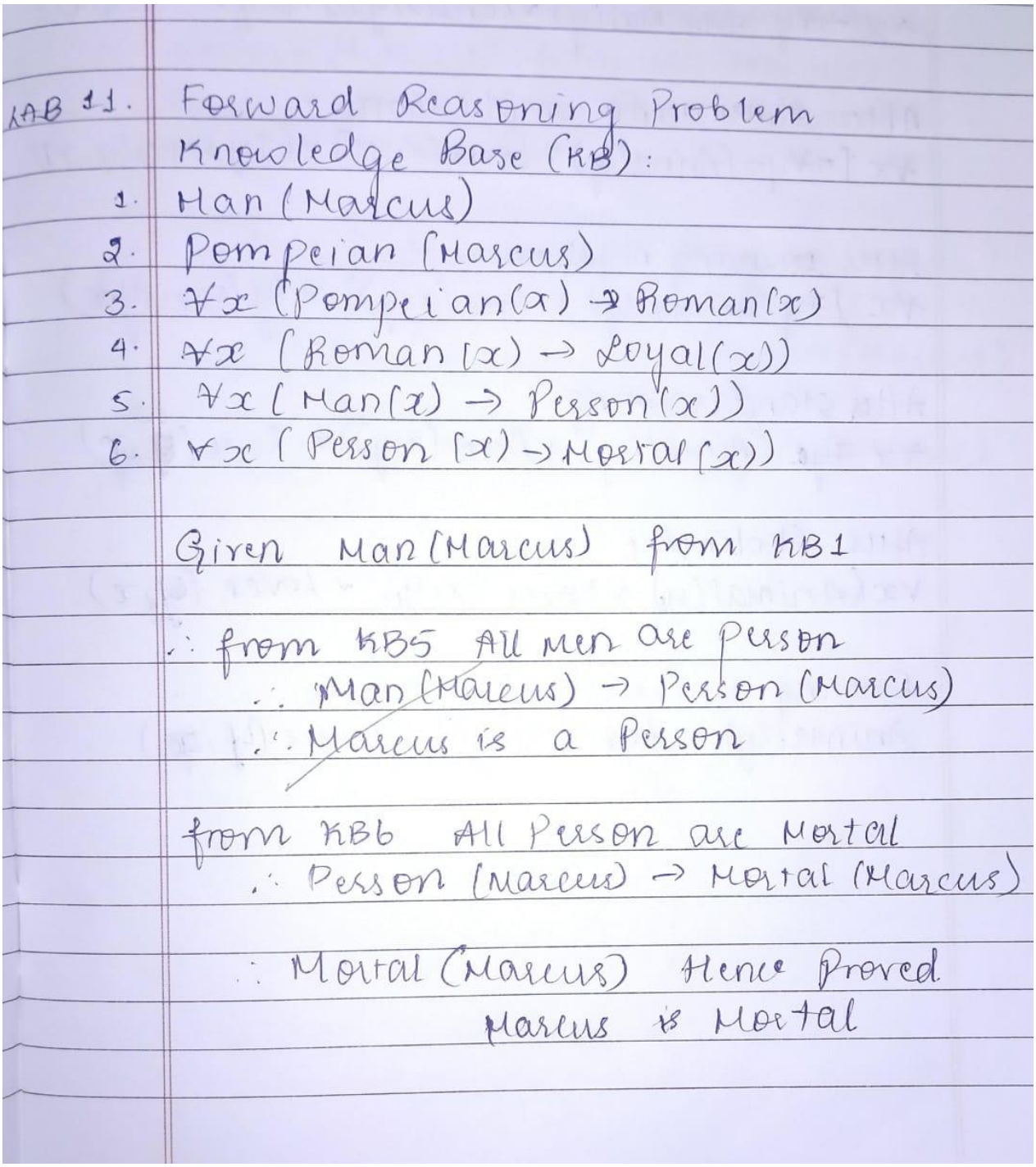
Clear

```
Unification Successful! Substitutions: {'f(x)': 'f(g(z))', 'g(y)': 'g(f(a))', 'y': 'f(a)'}

=== Code Execution Successful ===
```

Program 8

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

Algorithm:



Code:

```
import re
from collections import deque, namedtuple

# ---------------------------
# Utilities: parsing & types
# ---------------------------
Literal = namedtuple("Literal", ["pred", "args", "neg"])  # neg kept for possible extension

_VAR_RE = re.compile(r'^[a-z][a-zA-Z0-9_]*$')     # variable: starts with lowercase
_PRED_RE = re.compile(r'^([A-Za-z][A-Za-z0-9_]*)\((.*)\)$')

def is_variable(token):
    return bool(_VAR_RE.match(token))

def parse_literal(s):
    """Parse a predicate string like 'Loves(x,y)' into Literal(pred, [args])"""
    s = s.strip()
    m = _PRED_RE.match(s)
    if not m:
        raise ValueError(f"Bad literal format: {s}")
    pred = m.group(1)
    args = [a.strip() for a in m.group(2).split(',')] if m.group(2).strip() else []
    return Literal(pred=pred, args=args, neg=False)

def literal_to_str(lit):
    return f"{lit.pred}({', '.join(lit.args)})"
def unify(x, y, theta=None):
    """Unify two terms (variables/constants or lists) with substitution theta."""
    if theta is None:
```

```
        theta = {}
    # if identical after applying theta
    x = substitute_term(x, theta)
    y = substitute_term(y, theta)

    # variable cases
    if isinstance(x, str) and is_variable(x):
        return unify_var(x, y, theta)
    if isinstance(y, str) and is_variable(y):
        return unify_var(y, x, theta)

    # compound (list) case
    if isinstance(x, list) and isinstance(y, list) and len(x) == len(y):
        for xi, yi in zip(x, y):
            theta = unify(xi, yi, theta)
            if theta is None:
                return None
        return theta

    # constants / atoms
    if x == y:
        return theta
    return None

def unify_var(var, x, theta):
    if var in theta:
        return unify(theta[var], x, theta)
    if isinstance(x, str) and x in theta:
        return unify(var, theta[x], theta)
    # occurs check: variable should not appear in x
```

```
    if occurs_check(var, x, theta):
        return None
    theta2 = dict(theta)
    theta2[var] = x
    return theta2

def occurs_check(var, x, theta):
    x = substitute_term(x, theta)
    if var == x:
        return True
    if isinstance(x, list):
        return any(occurs_check(var, xi, theta) for xi in x)
    return False

def substitute_term(term, theta):
    """Apply substitution theta to a term (str or list)."""
    if isinstance(term, list):
        return [substitute_term(t, theta) for t in term]
    if isinstance(term, str) and term in theta:
        return substitute_term(theta[term], theta)
    return term

def apply_substitution_literal(lit, theta):
    """Return a new Literal with substitution applied to args."""
    new_args = [str(substitute_term(arg, theta)) for arg in lit.args]
    return Literal(pred=lit.pred, args=new_args, neg=lit.neg)
Rule = namedtuple("Rule", ["antecedents", "consequent"])  # antecedents: list of Literals, consequent:
Literal

def parse_rule(s):
    """Parse a rule string like 'A(x) & B(x,y) -> C(x)' or a fact 'Fact(a)'"""
```

```
    s = s.strip()
    if "->" in s:
        left, right = s.split("->", 1)
        ants = [parse_literal(part) for part in re.split(r'\s*&\s*', left.strip()) if part.strip()]
        cons = parse_literal(right.strip())
        return Rule(antecedents=ants, consequent=cons)
    else:
        # treat as fact with zero antecedents rule
        lit = parse_literal(s)
        return Rule(antecedents=[], consequent=lit)
def forward_chain(rules, facts, query=None, verbose=True):
    """
    rules: list of Rule objects
    facts: list of Literal objects (ground facts)
    query: string such as 'Mortal(Marcus)' or None
    Returns (entailed_bool, derived_facts_set)
    """
    # store facts as strings for quick membership; but also keep Literal objects
    derived = set(literal_to_str(f) for f in facts)
    fact_objs = {literal_to_str(f): f for f in facts}

    agenda = deque(facts)  # facts to consider (Literal objects)
    new_inferred = True

    if verbose:
        print("Initial Facts:")
        for f in facts:
            print("  ", literal_to_str(f))
        print("---- rules ----")
        for r in rules:
```

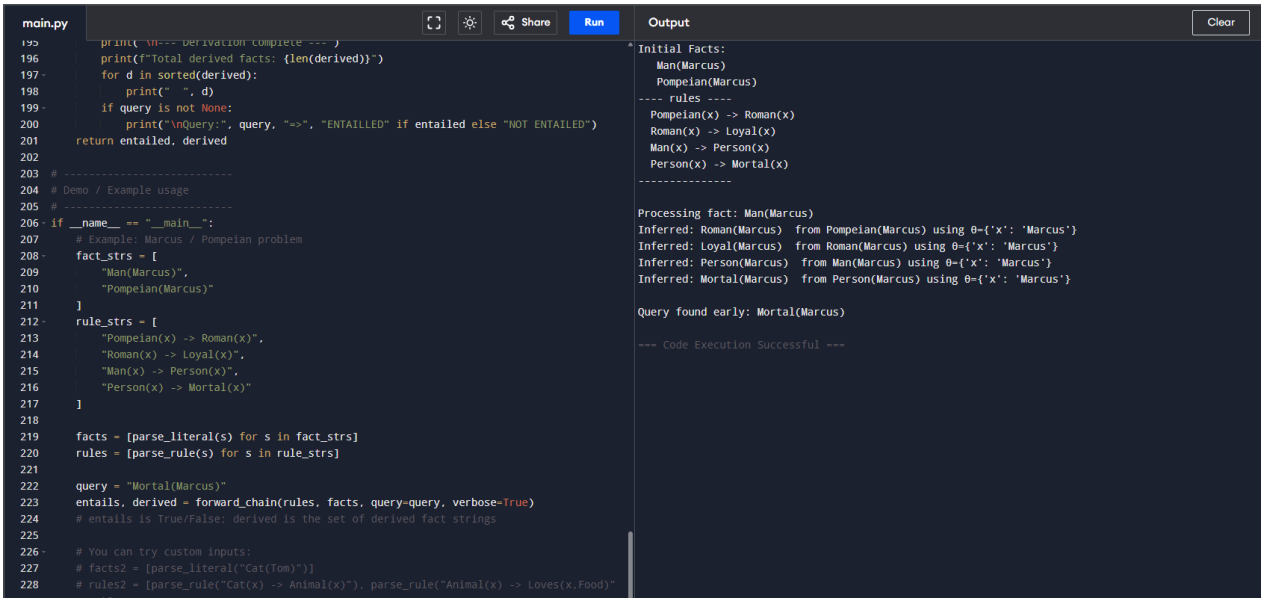
```
        ants = " & ".join(literal_to_str(a) for a in r.antecedents) if r.antecedents else "TRUE"
        print(f"  {ants} -> {literal_to_str(r.consequent)}")
    print("---------------\n")

while agenda:
    fact = agenda.popleft()
    if verbose:
        print("Processing fact:", literal_to_str(fact))
    # Try to apply each rule whose antecedents can be unified with known facts
    for rule in rules:
        # For rules with no antecedent (facts as rules), check if consequent already known
        if not rule.antecedents:
            cons_subst = {}
            # consequent may contain variables; but a fact-rule would normally be ground
            cons = apply_substitution_literal(rule.consequent, cons_subst)
            cons_str = literal_to_str(cons)
            if cons_str not in derived:
                derived.add(cons_str)
                fact_objs[cons_str] = cons
                agenda.append(cons)
                if verbose:
                    print("Inferred (from fact-rule):", cons_str)
            continue

        # For rules with antecedents, we attempt to find substitutions that make all antecedents true.
        # We perform a backtracking search over antecedents, building substitutions using unification
        def backtrack(idx, theta):
            if idx == len(rule.antecedents):
                # all antecedents unified under theta => infer consequent
                cons = apply_substitution_literal(rule.consequent, theta)
```

```
            cons_str = literal_to_str(cons)
            if cons_str not in derived:
                derived.add(cons_str)
                fact_objs[cons_str] = cons
                agenda.append(cons)
                if verbose:
                    ant_strs = [literal_to_str(apply_substitution_literal(a, theta)) for a in rule.antecedents]
                    print(f"Inferred: {cons_str}  from {', '.join(ant_strs)} using θ={theta}")
                return

            antecedent = rule.antecedents[idx]
            # We need to try to unify this antecedent with any known fact (derived)
            for known_str, known_lit in list(fact_objs.items()):
                if antecedent.pred != known_lit.pred or len(antecedent.args) != len(known_lit.args):
                    continue
                # attempt to unify antecedent.args with known_lit.args under current theta
                theta_try = unify(list(antecedent.args), list(known_lit.args), dict(theta))
                if theta_try is not None:
                    backtrack(idx + 1, theta_try)

        # Start backtracking with current fact as a potential match for antecedents[0]
        # (We could try all known facts; backtrack function already does that by iterating fact_objs)
        backtrack(0, {})

    # optional early stopping if query found
    if query is not None and query in derived:
        if verbose:
            print("\nQuery found early:", query)
        return True, derived
```

```
    # finished
    entailed = (query in derived) if query is not None else None
    if verbose:
        print("\n--- Derivation complete ---")
        print(f"Total derived facts: {len(derived)}")
        for d in sorted(derived):
            print("  ", d)
        if query is not None:
            print("\nQuery:", query, "=>", "ENTAILLED" if entailed else "NOT ENTAILED")
    return entailed, derived
if __name__ == "__main__":
    # Example: Marcus / Pompeian problem
    fact_strs = [
        "Man(Marcus)",
        "Pompeian(Marcus)"
    ]
    rule_strs = [
        "Pompeian(x) -> Roman(x)",
        "Roman(x) -> Loyal(x)",
        "Man(x) -> Person(x)",
        "Person(x) -> Mortal(x)"
    ]

    facts = [parse_literal(s) for s in fact_strs]
    rules = [parse_rule(s) for s in rule_strs]

    query = "Mortal(Marcus)"
    entails, derived = forward_chain(rules, facts, query=query, verbose=True)
    # entails is True/False; derived is the set of derived fact strings
```

```
    # You can try custom inputs:
    # facts2 = [parse_literal("Cat(Tom)")]
    # rules2 = [parse_rule("Cat(x) -> Animal(x)"), parse_rule("Animal(x) -> Loves(x,Food)")]
    # forward_chain(rules2, facts2, query="Loves(Tom,Food)")
```

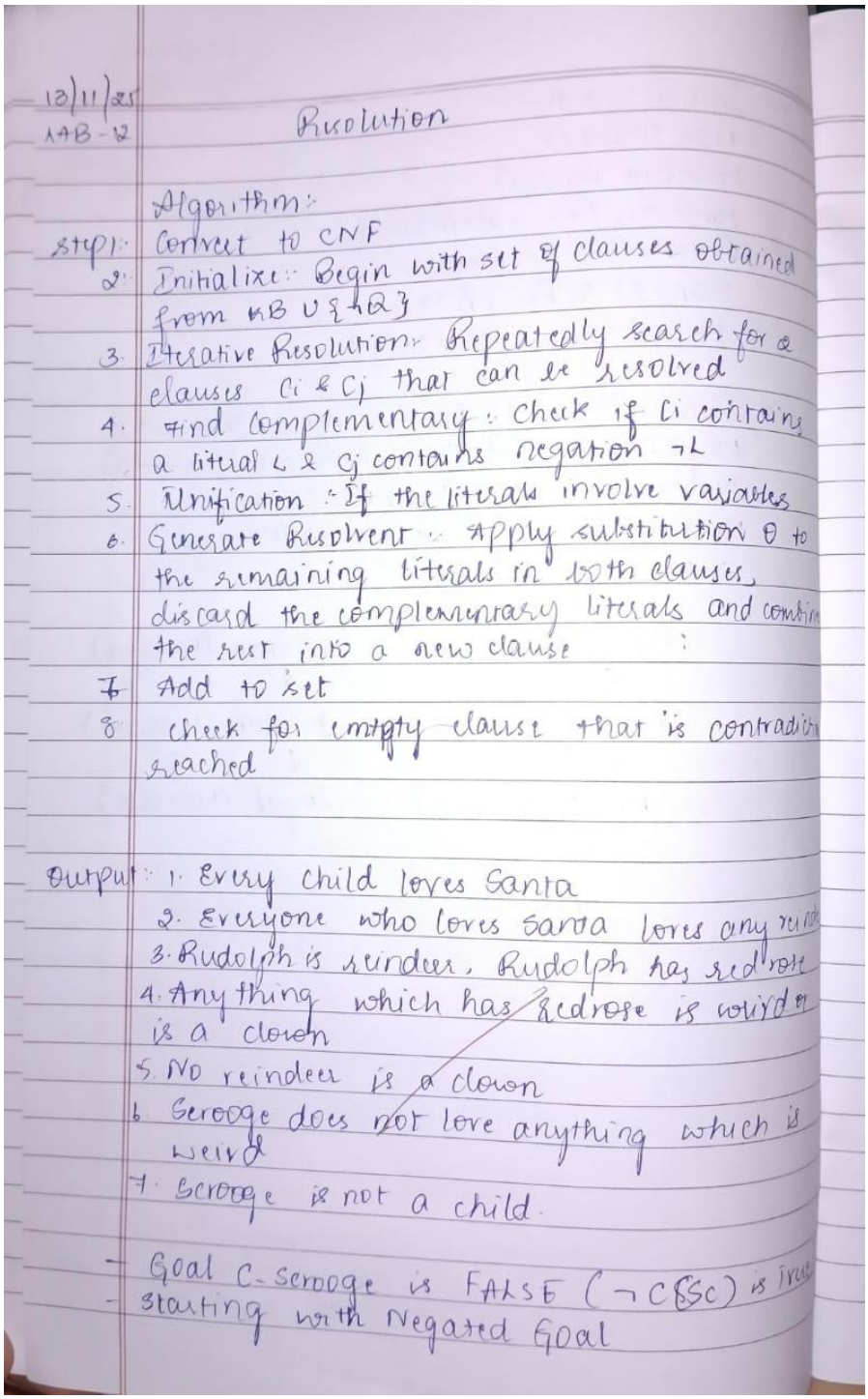
OUTPUT:



Program 9

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

Algorithm:



8. Resolved (7) & (1) → {L-Scrooge-Santa}
9. Resolved (8) & (2) → {¬R-Rudolph, L-Scrooge-Rudolph}
10. Resolved (9) & (3) → {L-Scrooge-Rudolph}
11. Resolved (3) & (4) → {A-Rudolph, W-Rudolph}
12. Resolved (11) & (5) → {¬R-Rudolph, W-Rudolph}
13. Resolved (3) & (12) → {W-Rudolph}
14. Resolved (13) & (6) → {¬L-Scrooge-Rudolph}
15. Resolved (10) & (14) → set() empty clause

The generation of empty clause confirms the contradiction with the negated goal.
Conclusion. Scrooge is not child {¬C(sc)}

¬Reindeer(Ru) ∨ Loves(Sc, Ru) ¬Weird(Ru) ∨ ¬Loves(Sc, Ru)

Reindeer(Ru)

Reindeer(Ru)

Weird(Ru) ∨ ¬Reindeer(Ru)

¬(Child(x) ∨ Loves(x, S))

Weird(Ru) ∨ (Clown(Ru)

RedNose(Ru)

¬Reindeer(x) ∨ ¬(Clown(Ru)

13/11/25

¬RedNose(x) ∨ Weird(x) ∨ (Clown(x)

empty.

Code:

```
def resolve(clause1, clause2):
    """
    Simulates the resolution step between two clauses.
    Looks for a complementary literal (L and ¬L) and returns the resolvent.

    Args:
        clause1 (set): The first parent clause.
        clause2 (set): The second parent clause.

    Returns:
        set or None: The resolvent clause (a set) or None if no resolution is possible.
    """
    # The resolution rule: Find a literal in one clause and its negation in the other.
    for literal1 in clause1:
        # Determine the negation of literal1
        # Negation is represented by adding/removing the '-' prefix
        negation_of_literal1 = literal1[1:] if literal1.startswith('-') else '-' + literal1

        if negation_of_literal1 in clause2:
            # Resolution is possible:
            # 1. Remove the literal and its negation from the parent clauses.
            # 2. Combine the remaining literals using the union operator (|).
            resolvent = (clause1 - {literal1}) | (clause2 - {negation_of_literal1})
            return resolvent
    return None

def run_resolution_proof():
    """
    Executes the Resolution Refutation proof for the 'Scrooge is not a child' problem.
```

```
"""
# --- PREDICATE MAPPING ---
# A dictionary to map human-readable names to simplified literals for the code
P = {
    "C(Sc)": "C_Scrooge",           # Scrooge is a child
    "L(Sc, S)": "L_Scrooge_Santa",  # Scrooge loves Santa
    "R(Ru)": "R_Rudolph",           # Rudolph is a reindeer
    "N(Ru)": "N_Rudolph",           # Rudolph has a red nose
    "W(Ru)": "W_Rudolph",           # Rudolph is weird
    "A(Ru)": "A_Rudolph",           # Rudolph is a clown
    "L(Sc, Ru)": "L_Scrooge_Rudolph", # Scrooge loves Rudolph
}

# --- KNOWLEDGE BASE (KB) and NEGATED GOAL (CNF Clauses) ---
# Clauses use negative signs for negation, e.g., -C_Scrooge for ¬C(Scrooge)

# 1. ¬C(x) ∨ L(x, S) -> Specific instance used: ¬C(Sc) ∨ L(Sc, S)
clause1 = {"-C_Scrooge", P["L(Sc, S)"]}

# 2. ¬L(x, S) ∨ ¬R(y) ∨ L(x, y) -> Specific instance: ¬L(Sc, S) ∨ ¬R(Ru) ∨ L(Sc, Ru)
clause2 = {"-L_Scrooge_Santa", "-R_Rudolph", P["L(Sc, Ru)"]}

# 3. R(Ru) and N(Ru)
clause3_R = {P["R(Ru)"]}
clause3_N = {P["N(Ru)"]}

# 4. ¬N(x) ∨ W(x) ∨ A(x) -> Specific instance: ¬N(Ru) ∨ W(Ru) ∨ A(Ru)
clause4 = {"-N_Rudolph", P["W(Ru)"], P["A(Ru)"]}

# 5. ¬R(x) ∨ ¬A(x) -> Specific instance: ¬R(Ru) ∨ ¬A(Ru)
```

```
clause5 = {"-R_Rudolph", "-A_Rudolph"}

# 6. ¬W(x) ∨ ¬L(Sc, x) -> Specific instance: ¬W(Ru) ∨ ¬L(Sc, Ru)
clause6 = {"-W_Rudolph", "-L_Scrooge_Rudolph"}

# 8. NEGATED GOAL: C(Sc) (Scrooge is a child)
negated_goal = {P["C(Sc)"]}

# Store all clauses for tracking (index starts at 0 for list, but logically they are clause 1 to 8)
CLAUSES = [clause1, clause2, clause3_R, clause3_N, clause4, clause5, clause6, negated_goal]
EMPTY_CLAUSE = set()

print("--- Starting Resolution Refutation Proof ---")
print(f"Goal: {P['C(Sc)']} is FALSE (i.e., ¬C(Sc) is TRUE)")
print(f"Starting with Negated Goal (Clause 8): {negated_goal}")
print("-" * 35)

# --- RESOLUTION STEPS (9 through 16) ---
try:
    # Step 9: (8) C(Sc) and (1) ¬C(Sc) ∨ L(Sc, S) -> L(Sc, S)
    c8 = CLAUSES[7] # {C_Scrooge}
    c1 = CLAUSES[0] # {-C_Scrooge, L_Scrooge_Santa}
    c9 = resolve(c8, c1)
    if c9 is None: raise Exception("Step 9 failed.")
    print(f"9. Resolved (8) and (1) -> {c9}")

    # Step 10: (9) L(Sc, S) and (2) ¬L(Sc, S) ∨ ¬R(Ru) ∨ L(Sc, Ru) -> ¬R(Ru) ∨ L(Sc, Ru)
    c2 = CLAUSES[1]
    c10 = resolve(c9, c2)
    if c10 is None: raise Exception("Step 10 failed.")
```

```
print(f"10. Resolved (9) and (2) -> {c10}")

# Step 11: (3) R(Ru) and (10) ¬R(Ru) ∨ L(Sc, Ru) -> L(Sc, Ru)
c3_R = CLAUSES[2]
c11 = resolve(c3_R, c10)
if c11 is None: raise Exception("Step 11 failed.")
print(f"11. Resolved (3) and (10) -> {c11}")

# Step 12: (3) N(Ru) and (4) ¬N(Ru) ∨ W(Ru) ∨ A(Ru) -> W(Ru) ∨ A(Ru)
c3_N = CLAUSES[3]
c4 = CLAUSES[4]
c12 = resolve(c3_N, c4)
if c12 is None: raise Exception("Step 12 failed.")
print(f"12. Resolved (3) and (4) -> {c12}")

# Step 13: (12) W(Ru) ∨ A(Ru) and (5) ¬R(Ru) ∨ ¬A(Ru) -> W(Ru) ∨ ¬R(Ru)
c5 = CLAUSES[5]
c13 = resolve(c12, c5)
if c13 is None: raise Exception("Step 13 failed.")
print(f"13. Resolved (12) and (5) -> {c13}")

# Step 14: (3) R(Ru) and (13) W(Ru) ∨ ¬R(Ru) -> W(Ru)
c14 = resolve(c3_R, c13)
if c14 is None: raise Exception("Step 14 failed.")
print(f"14. Resolved (3) and (13) -> {c14}")

# Step 15: (14) W(Ru) and (6) ¬W(Ru) ∨ ¬L(Sc, Ru) -> ¬L(Sc, Ru)
c6 = CLAUSES[6]
c15 = resolve(c14, c6)
if c15 is None: raise Exception("Step 15 failed.")
```

```
        print(f"15. Resolved (14) and (6) -> {c15}")

        # Step 16: (11) L(Sc, Ru) and (15) ¬L(Sc, Ru) -> {} (Empty Clause)
        c16 = resolve(c11, c15)

        print("-" * 35)

        if c16 == EMPTY_CLAUSE:
            print(f"16. Resolved (11) and (15) -> {c16} (EMPTY CLAUSE)")
            print("\n*** PROOF SUCCESSFUL! ***")
            print("The generation of the empty clause confirms a contradiction with the negated goal.")
            print("Conclusion: Scrooge is not a child (¬C(Sc)) is PROVEN TRUE.")
        else:
            print("Proof failed: Did not find the empty clause.")

    except Exception as e:
        print(f"\nProof failed early: {e}")
        print("A step in the manual deduction failed, meaning the specific literal pairs used in that step were not found in the clauses.")


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

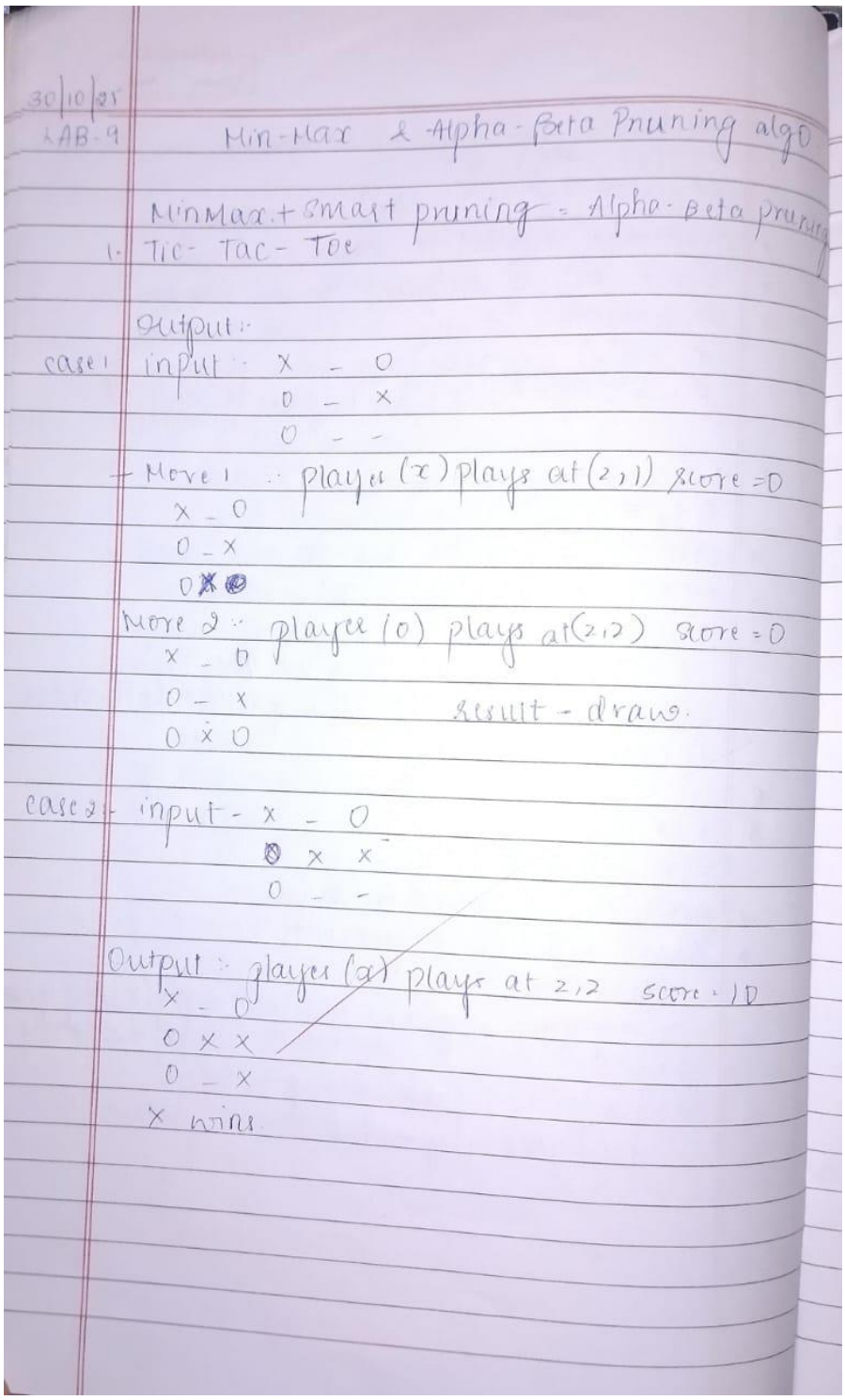
OUPUT:

```
=== 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:



Bafna Gold
Date: Page:

2. 8-puzzle

Initial State.

1	2	3
7	4	6
5	0	8

Goal state

1	2	3
4	5	6
7	8	0

Step 1 -

1	2	3
7	4	6
0	5	8

→

1	2	3
0	4	6
7	5	8

→

1	2	3
4	5	6
7	0	8

↓

1	2	3
4	5	6
7	8	0

~~Alpha - Beta algorithm~~

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
```