VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Shrinanda Shivprasad Dinde[1BM23CS324]

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

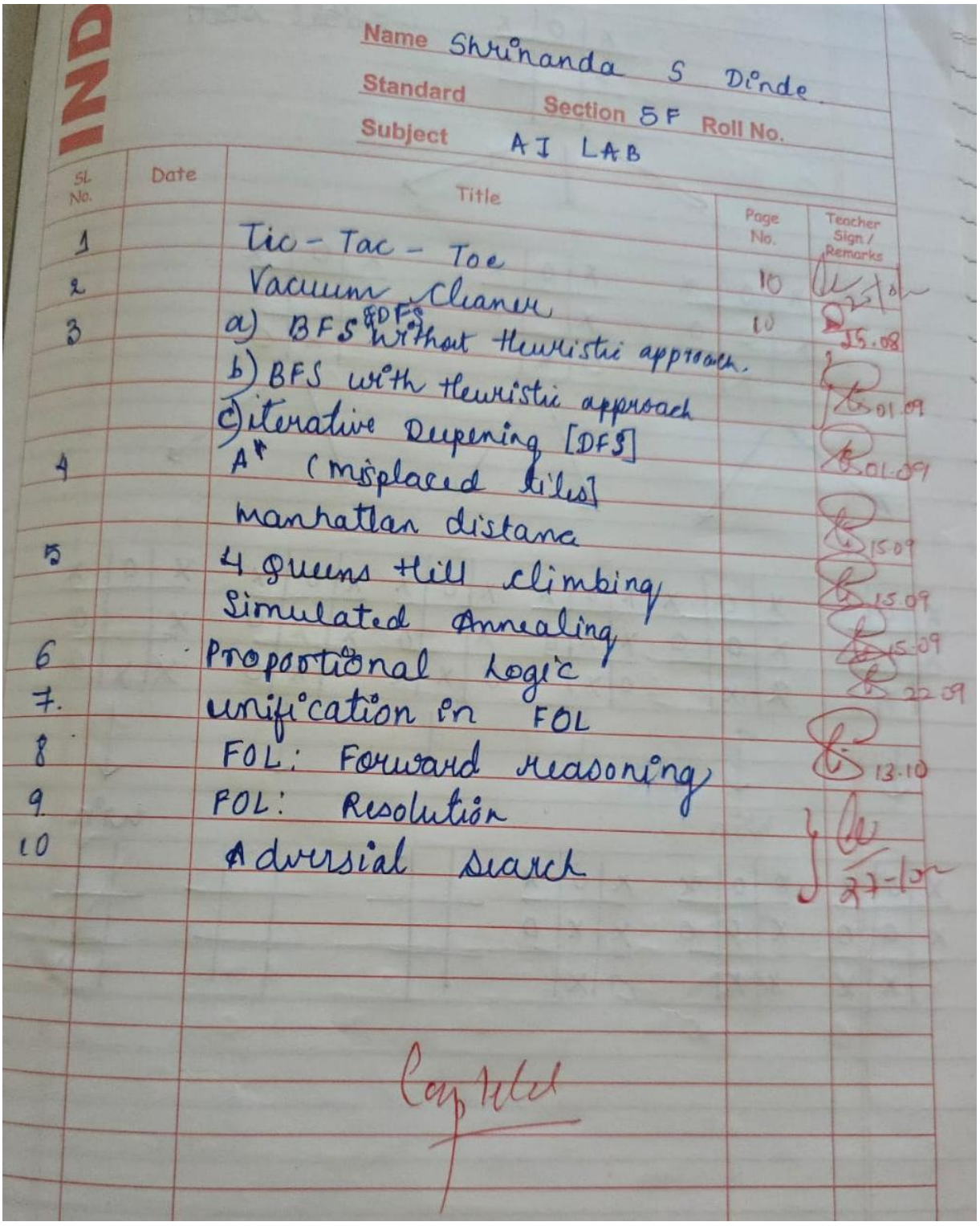
This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **ShrinandaShivprasad Dinde(1BM23CS324),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and 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.

Surabhi S Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:
https://github.com/shrinanda27/AI-LAB-1BM23CS324.git

INDEX:



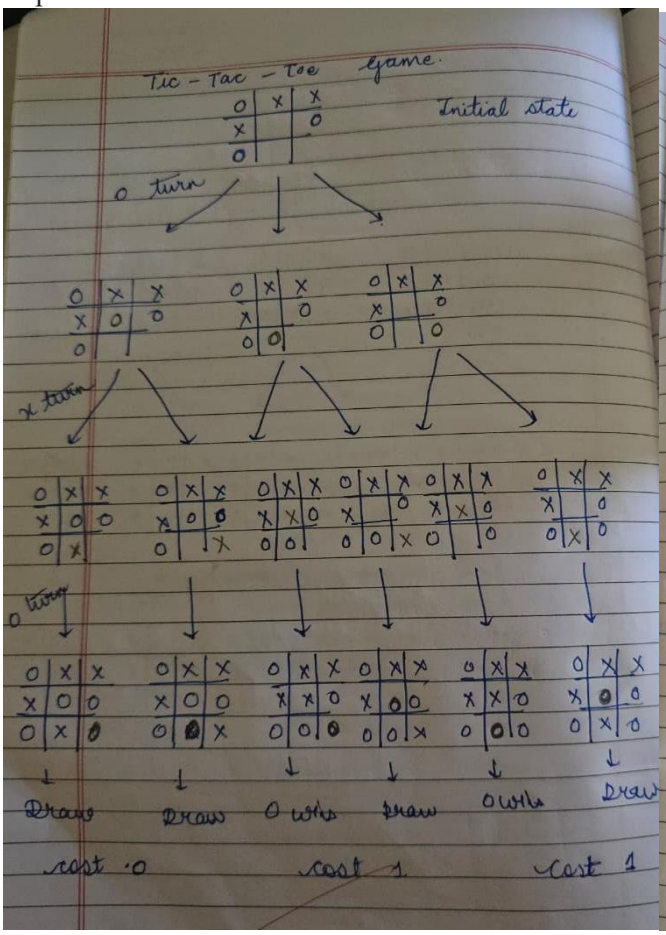
Name Shrinanda S Dinde
Standard Section 5F Roll No.
Subject AI LAB

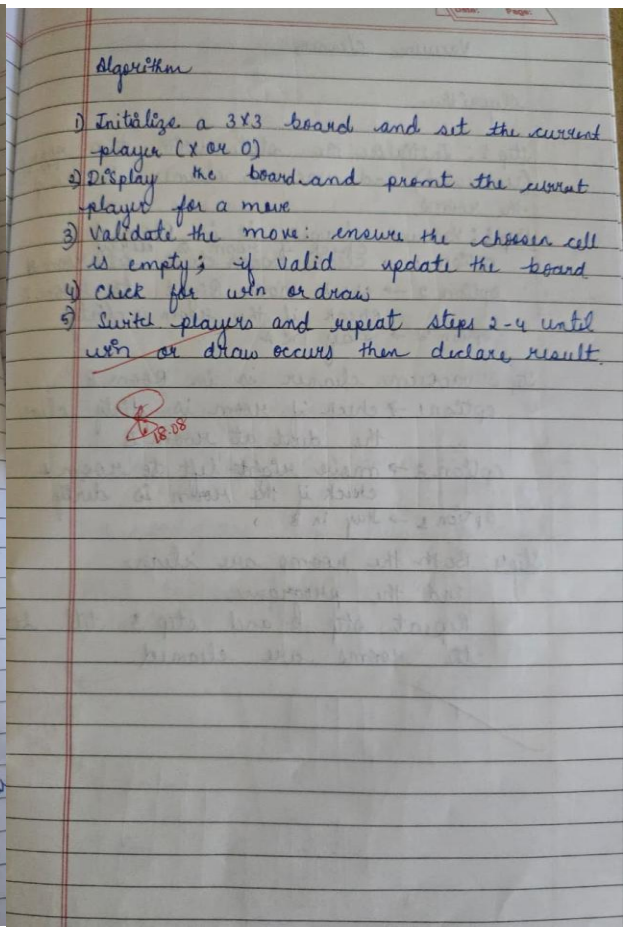
Sl No.	Date	Title	Page No.	Teacher Sign / Remarks
1		Tic - Tac - Toe	10	27/10
2		Vacuum Cleaner	10	15.08
3		a) BFS & DFS without Heuristic approach.		
		b) BFS with Heuristic approach		01.09
		c) Iterative Deepening [DFS]		
4		A* (misplaced tiles)		01.09
		manhattan distance		
5		4 Queens Hill climbing		15.09
		Simulated Annealing		15.09
6		Proportional Logic		15.09
7.		unification in FOL		22.09
8		FOL: Forward reasoning		13.10
9.		FOL: Resolution		
10		Adversial search		27-10

Completed

Program 1

Implement Tic –Tac –Toe Game





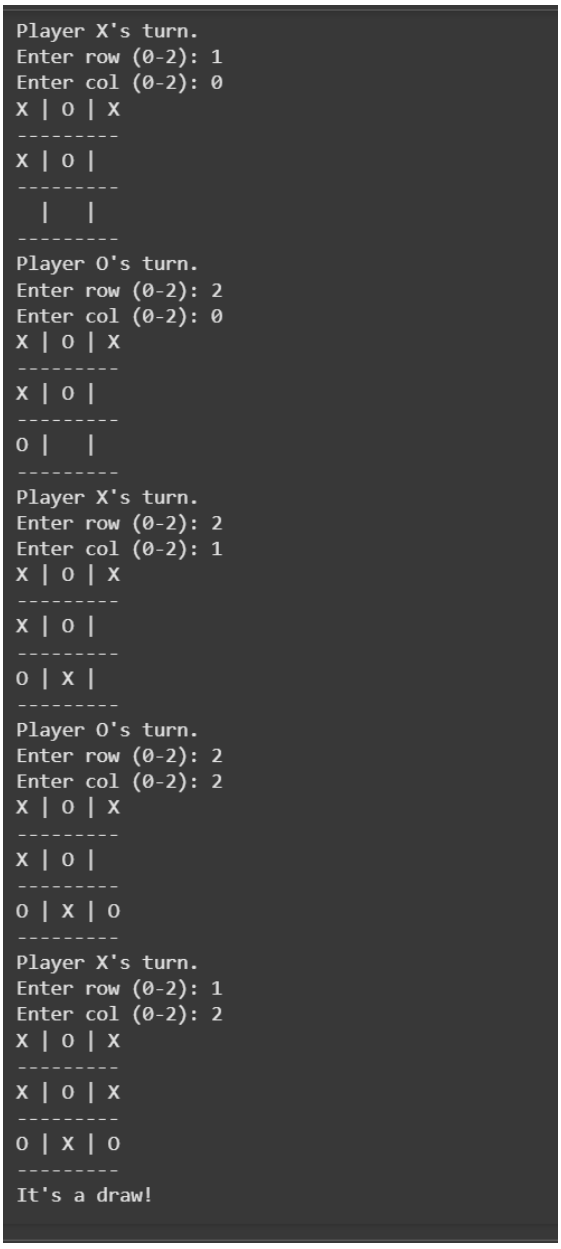
Code:

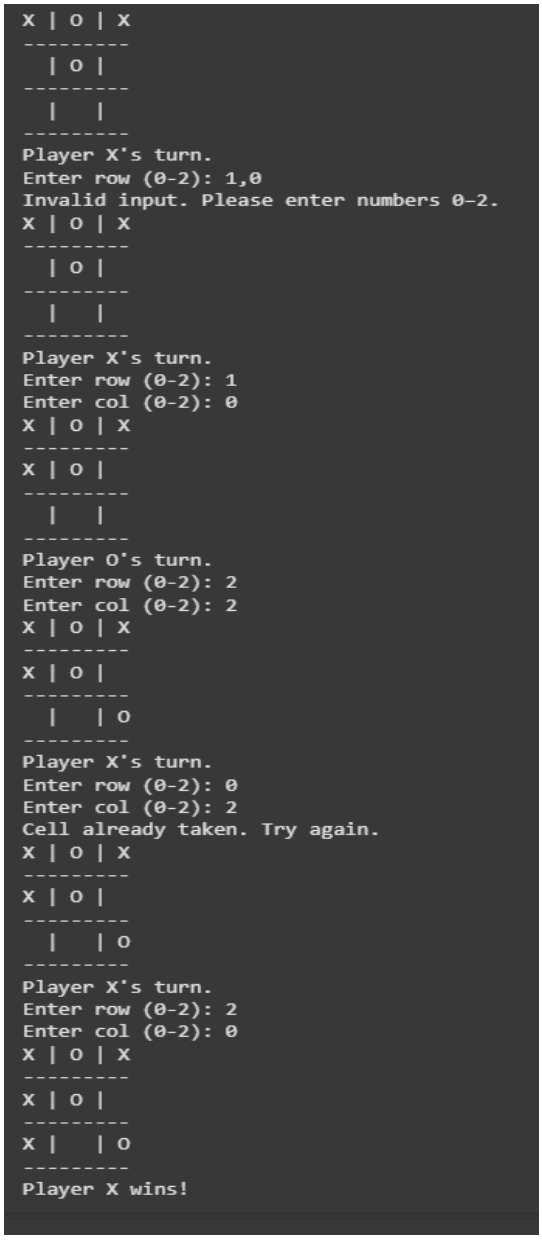
```
def print_board(board):
    for row in board:
        print(" | ".join(row))
        print("-" * 9)

def check_winner(board, player):
    for i in range(3):
        if all(board[i][j] == player for j in range(3)) or \
           all(board[j][i] == player for j in range(3)):
            return True
    return board[0][0] == board[1][1] == board[2][2] == player or \
           board[0][2] == board[1][1] == board[2][0] == player

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

```
def play_game(board):
    current_player = 'X' if sum(row.count('X') for row in board) <= sum(row.count('O') for row in
board) else 'O'
    while True:
        print_board(board)
        print(f"Player {current_player}'s turn.")
        try:
            row = int(input("Enter row (0-2): "))
            col = int(input("Enter col (0-2): "))
        except ValueError:
            print("Invalid input. Please enter numbers 0–2.")
            continue
        if not (0 <= row <= 2 and 0 <= col <= 2):
            print("Invalid position. Try again.")
            continue
        if board[row][col] != ' ':
            print("Cell already 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'


initial_board = [
    ['X', 'O', 'X'],
    [' ', 'O', ' '],
    [' ', ' ', ' ']
]

for i in range(3):
    for j in range(3):
        if initial_board[i][j] not in ['X', 'O']:
```

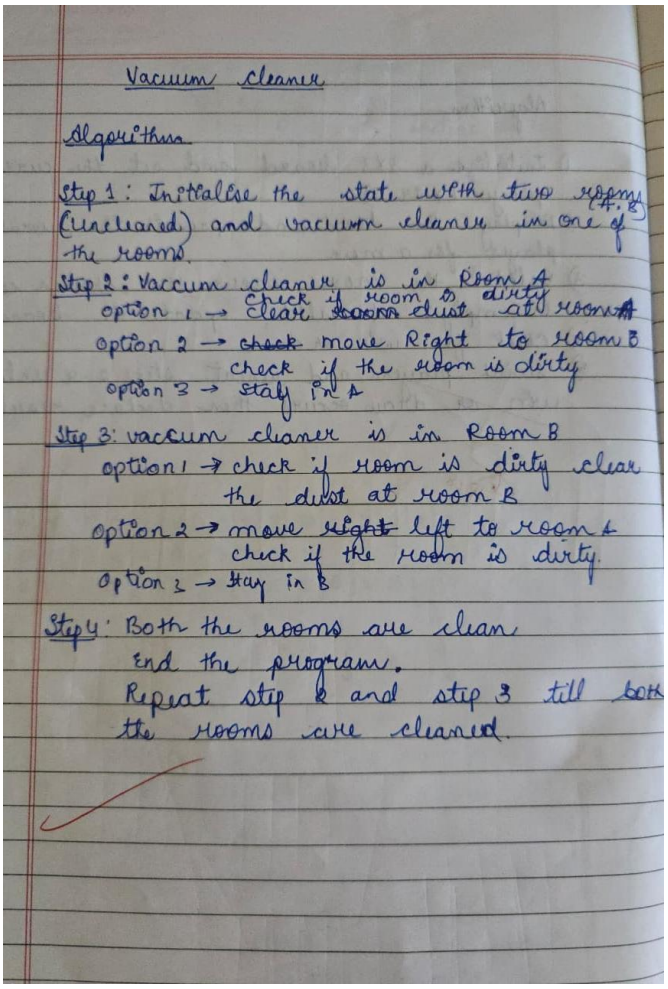
```
        initial_board[i][j] = ' '

play_game(initial_board)
```

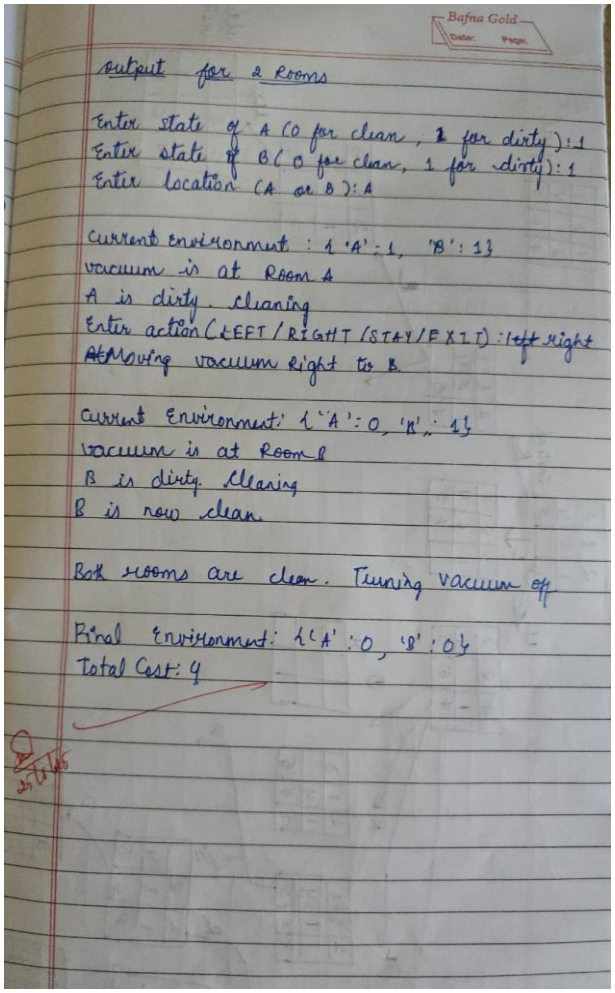
OUTPUT:





Implement vacuum cleaner agent





CODE:

```
def vacuum_cleaner():
    try:
        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()

        if state_A not in (0, 1) or state_B not in (0, 1) or location not in ('A', 'B'):
            raise ValueError("Invalid input! Please enter correctly.")
    except ValueError as e:
        print("Error:", e)
        return

    environment = {'A': state_A, 'B': state_B}
    cost = 0

    while True:
```

```
        print("\nCurrent Environment:", environment)
        print("Vacuum is at Room", location)

        if environment[location] == 1:
            print(f"{location} is dirty. Cleaning...")
            environment[location] = 0
            cost += 1
            print(f"{location} is now clean.")
        else:
            print(f"{location} is already clean.")

        if environment['A'] == 0 and environment['B'] == 0:
            print("\nBoth rooms are clean. Turning vacuum off.")
            break

        move = input("Enter action (LEFT / RIGHT / STAY / EXIT): ").upper()

        if move == "LEFT":
            if location == 'A':
                print("Already at A. Can't move LEFT.")
            else:
                location = 'A'
                print("Moving vacuum LEFT to A.")
            cost += 1

        elif move == "RIGHT":
            if location == 'B':
                print("Already at B. Can't move RIGHT.")
            else:
                location = 'B'
                print("Moving vacuum RIGHT to B.")
            cost += 1

        elif move == "STAY":
            print("Staying in the same room.")

        elif move == "EXIT":
            print("Exiting manually.")
            break

        else:
            print("Invalid action! Please enter LEFT, RIGHT, STAY, or EXIT.")

    print("\nFinal Environment:", environment)
    print("Total Cost:", cost)

vacuum_cleaner()
```

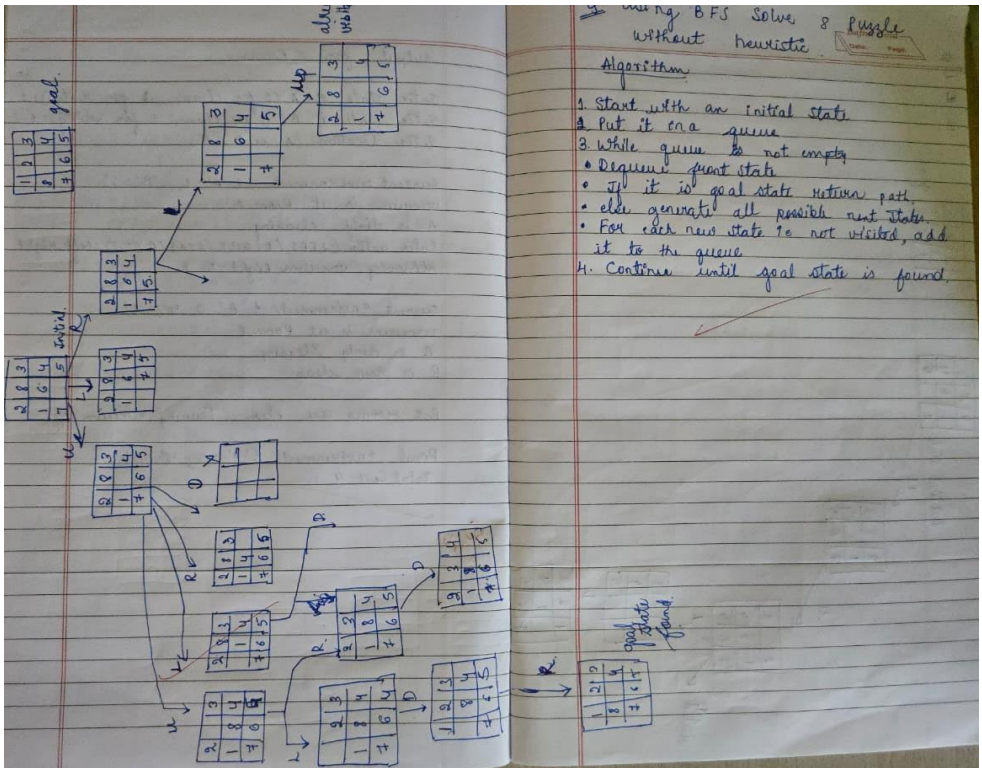
OUTPUT:

```
Name :Shrinanda Shivprasad Dinde
USN:1BM23CS324
Enter state of A (0 for clean, 1 for dirty): 1
Enter state of B (0 for clean, 1 for dirty): 1
Enter location (A or B): A

Current Environment: {'A': 1, 'B': 1}
Vacuum is at Room A
A is dirty. Cleaning...
A is now clean.
Enter action (LEFT / RIGHT / STAY / EXIT): left
Already at A. Can't move LEFT.

Current Environment: {'A': 0, 'B': 1}
Vacuum is at Room A
A is already clean.
Enter action (LEFT / RIGHT / STAY / EXIT): right
Moving vacuum RIGHT to B.

Current Environment: {'A': 0, 'B': 1}
Vacuum is at Room B
B is dirty. Cleaning...
B is now clean.

Both rooms are clean. Turning vacuum off.

Final Environment: {'A': 0, 'B': 0}
Total Cost: 4
```

```
Name :Shrinanda Shivprasad Dinde
USN:1BM23CS324
Enter state of A (0 for clean, 1 for dirty): 0
Enter state of B (0 for clean, 1 for dirty): 0
Enter location (A or B): A

Current Environment: {'A': 0, 'B': 0}
Vacuum is at Room A
A is already clean.

Both rooms are clean. Turning vacuum off.

Final Environment: {'A': 0, 'B': 0}
Total Cost: 0
```

PROGRAM-2

Implement 8 puzzle problems



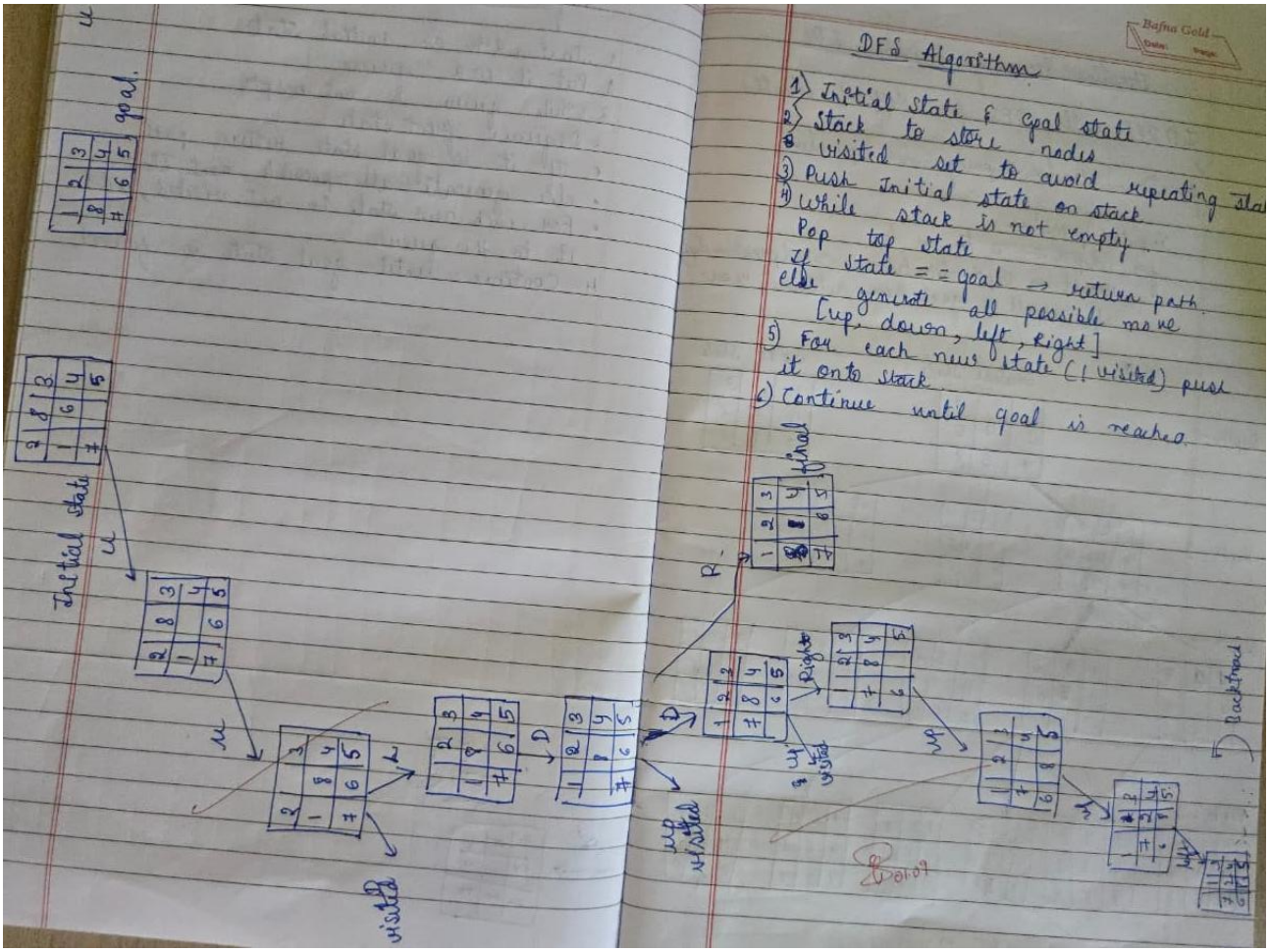
CODE:

```
from collections import deque
print("Shrinanda Shivprasad Dinde")
print("USN:1BM23CS324")

# Directions: Up, Down, Left, Right
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Convert flat list to 2D list
def to_matrix(state_list):
    return [state_list[i:i+3] for i in range(0, 9, 3)]

# Convert 2D list to tuple for hashing
def state_to_tuple(state):
    return tuple(num for row in state for num in row)

# Find position of 0 (blank)
def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
    return -1, -1

# Check boundaries
def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

# Swap tiles
def swap(state, x1, y1, x2, y2):
    new_state = [row[:] for row in state]
    new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
    return new_state

# BFS function
def bfs(start_state, goal_state):
    visited = set()
    queue = deque()
    parent = {}

    start_tuple = state_to_tuple(start_state)
    goal_tuple = state_to_tuple(goal_state)

    queue.append((start_state, 0))
    visited.add(start_tuple)
    parent[start_tuple] = None

    while queue:
        current_state, depth = queue.popleft()

        if state_to_tuple(current_state) == goal_tuple:
            print(f"\n Goal reached in {depth} moves.\n")
            print_path(parent, start_tuple, goal_tuple)
            return

        x, y = find_zero(current_state)

        for dx, dy in DIRECTIONS:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny):
                new_state = swap(current_state, x, y, nx, ny)
                new_tuple = state_to_tuple(new_state)
                if new_tuple not in visited:
                    visited.add(new_tuple)
                    queue.append((new_state, depth + 1))
                    parent[new_tuple] = state_to_tuple(current_state)

    print("No solution found.")
```

```
# Reconstruct and print solution path
def print_path(parent, start_tuple, goal_tuple):
    path = []
    current = goal_tuple
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()

    print("Solution Path:")
    for state in path:
        print_state(state)
        print()

# Pretty print a state
def print_state(state_tuple):
    for i in range(0, 9, 3):
        row = state_tuple[i:i+3]
        print(" ".join(str(x) if x != 0 else " " for x in row))

# Validate user input
def read_state(prompt):
    while True:
        try:
            nums = list(map(int, input(prompt).strip().split()))
            if len(nums) != 9 or sorted(nums) != list(range(9)):
                raise ValueError
            return to_matrix(nums)
        except ValueError:
            print("Invalid input. Please enter 9 numbers (0-8) with no duplicates.")

# Main
if __name__ == "__main__":
    print("8 Puzzle Solver using BFS (Uninformed Search)\n")
    print("Enter the initial state (use 0 for the blank tile):")
    start_state = read_state("Initial (e.g., 1 2 3 4 5 6 7 8 0): ")

    print("\nEnter the goal state:")
    goal_state = read_state("Goal (e.g., 1 2 3 4 5 6 7 8 0): ")

    bfs(start_state, goal_state)
```

```
Shrinanda Shivprasad Dinde
USN:1BM23CS324
8 Puzzle Solver using BFS (Uninformed Search)

Enter the initial state (use 0 for the blank tile):
Initial (e.g., 1 2 3 4 5 6 7 8 0): 2 8 3 1 6 4 7 0 5

Enter the goal state:
Goal (e.g., 1 2 3 4 5 6 7 8 0): 1 2 3 8 0 4 7 6 5

 Goal reached in 5 moves.

Solution Path:
2 8 3
1 6 4
7   5

2 8 3
1   4
7 6 5

2   3
1 8 4
7 6 5

  2 3
1 8 4
7 6 5

1 2 3
  8 4
7 6 5

1 2 3
8   4
7 6 5
```



CODE:

```
from collections import deque
print("Shrinanda Shivprasad Dinde")
print("USN:1BM23CS324")

# Directions: Up, Down, Left, Right
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

# Convert flat list to 2D list
def to_matrix(state_list):
    return [state_list[i:i+3] for i in range(0, 9, 3)]

# Convert 2D list to tuple
def state_to_tuple(state):
    return tuple(num for row in state for num in row)

# Find blank (0)
def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
```

```
    return -1, -1

# Check bounds
def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

# Swap tiles
def swap(state, x1, y1, x2, y2):
    new_state = [row[:] for row in state]
    new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
    return new_state

# DFS algorithm with visited set and optional depth limit
def dfs(start_state, goal_state, max_depth=50):
    visited = set()
    stack = [(start_state, 0)]  # (state, depth)
    parent = {}

    start_tuple = state_to_tuple(start_state)
    goal_tuple = state_to_tuple(goal_state)

    parent[start_tuple] = None
    visited.add(start_tuple)

    total_visited = 0

    while stack:
        current_state, depth = stack.pop()
        current_tuple = state_to_tuple(current_state)
        total_visited += 1

        if current_tuple == goal_tuple:
            print(f"\n Goal reached in {depth} moves.")
            print(f"Total states visited: {total_visited}\n")
            print_path(parent, start_tuple, goal_tuple)
            return

        if depth >= max_depth:
            continue  # Skip if depth limit is hit

        x, y = find_zero(current_state)

        for dx, dy in DIRECTIONS:
            nx, ny = x + dx, y + dy
            if is_valid(nx, ny):
                new_state = swap(current_state, x, y, nx, ny)
                new_tuple = state_to_tuple(new_state)
```

```
            if new_tuple not in visited:
                visited.add(new_tuple)
                parent[new_tuple] = current_tuple
                stack.append((new_state, depth + 1))

    print(" No solution found within depth limit.")
    print(f" Total states visited: {total_visited}")

# Reconstruct and print solution path
def print_path(parent, start_tuple, goal_tuple):
    path = []
    current = goal_tuple
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()

    print(" Solution Path:")
    for state in path:
        print_state(state)
        print()

# Pretty print state
def print_state(state_tuple):
    for i in range(0, 9, 3):
        row = state_tuple[i:i+3]
        print(" ".join(str(x) if x != 0 else " " for x in row))

# Get and validate user input
def read_state(prompt):
    while True:
        try:
            nums = list(map(int, input(prompt).strip().split()))
            if len(nums) != 9 or sorted(nums) != list(range(9)):
                raise ValueError
            return to_matrix(nums)
        except ValueError:
            print("Invalid input. Enter 9 numbers (0-8) with no duplicates.")

# Main function
if __name__ == "__main__":
    print(" 8 Puzzle Solver using DFS (Uninformed Search)\n")
    print("Enter the initial state (use 0 for the blank tile):")
    start_state = read_state("Initial (e.g., 1 2 3 4 5 6 7 8 0): ")

    print("\nEnter the goal state:")
    goal_state = read_state("Goal (e.g., 1 2 3 4 5 6 7 8 0): ")
```

```
# Optional: change max_depth for deeper searches
dfs(start_state, goal_state, max_depth=50)
```

OUTPUT:

Shrinanda Shivprasad Dinde
USN:1BM23CS324
8 Puzzle Solver using DFS (Uninformed Search)

Enter the initial state (use 0 for the blank tile):
Initial (e.g., 1 2 3 4 5 6 7 8 0): 2 8 3 1 6 4 7 0 5

Enter the goal state:
Goal (e.g., 1 2 3 4 5 6 7 8 0): 1 2 3 8 0 4 7 6 5

Goal reached in 49 moves.
Total states visited: 10111

Solution Path:

```
2 8 3
1 6 4
7   5

2 8 3
1 6 4
7 5

2 8 3
1 6
7 5 4

2 8 3
1   6
7 5 4

2 8 3
  1 6
7 5 4

2 8 3
7 1 6
  5 4

2 8 3
7 1 6
5   4
```

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

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

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

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

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

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

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

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

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

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

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

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

2 8 3
4 5 7
6 1

2 8 3
4 5
6 1 7

2 8 3
4 5
6 1 7

2 8 3
4 5
6 1 7

2 8 3
6 4 5
1 7

2 8 3
6 4 5
1 7

2 8 3
6 4 5
1 7

2 8 3
6 4
1 7 5

2 8 3
6 4
1 7 5

2 8 3
6 7 4
1 5

2 8 3
6 7 4
1 5

2 8 3
7 4

```
6 1 5

2 8 3
7  4
6 1 5

2 8 3
7 4
6 1 5

2 8
7 4 3
6 1 5

2  8
7 4 3
6 1 5

2 4 8
7  3
6 1 5

2 4 8
7 3
6 1 5

2 4
7 3 8
6 1 5

2  4
7 3 8
6 1 5

2 3 4
7  8
6 1 5

2 3 4
7 1 8
6  5

2 3 4
7 1 8
 6 5

2 3 4
```

1 8
7 6 5

2 3 4
1 8
7 6 5

2 3 4
1 8
7 6 5

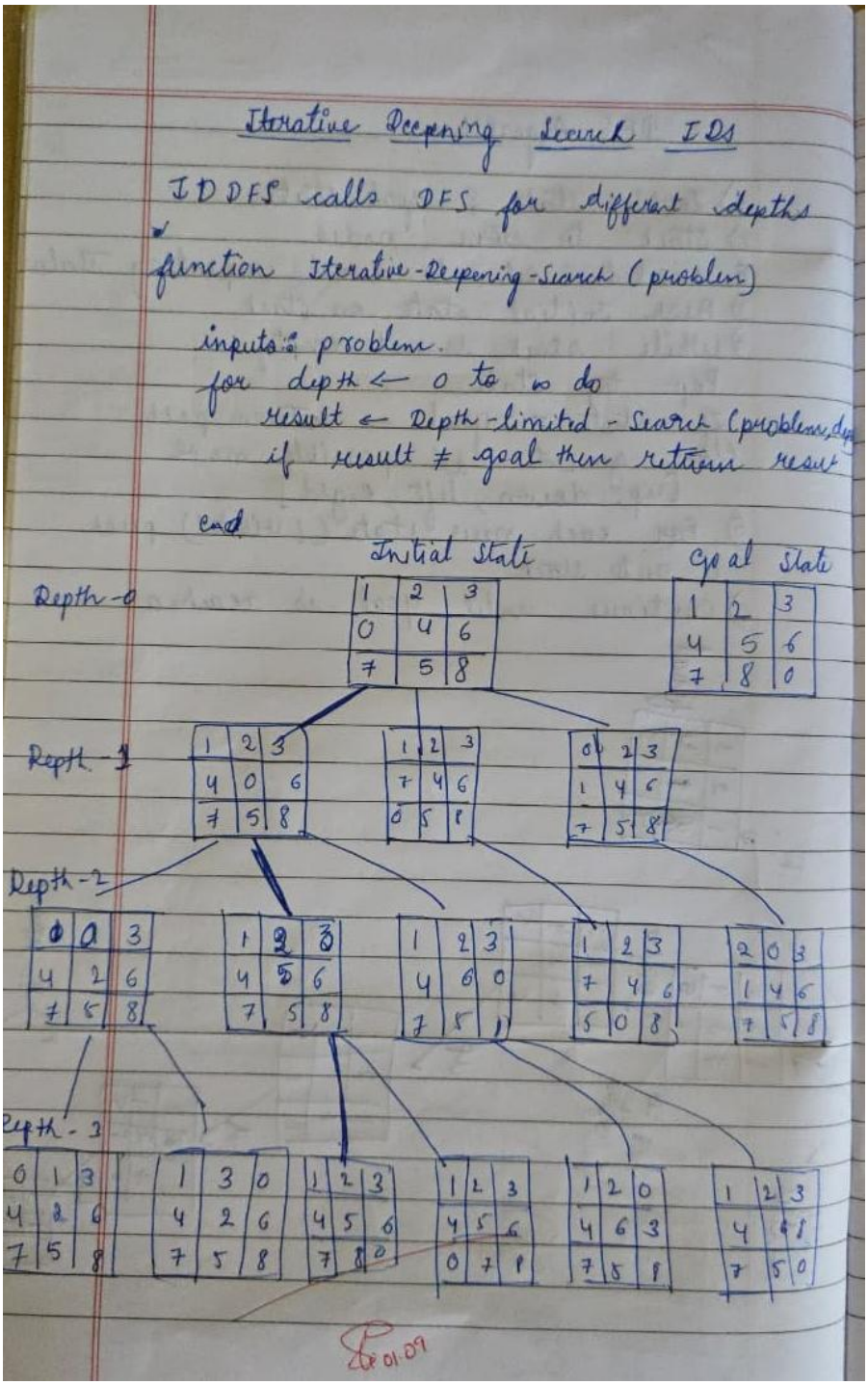
2 3
1 8 4
7 6 5

2 3
1 8 4
7 6 5

2 3
1 8 4
7 6 5

1 2 3
8 4
7 6 5

1 2 3
8 4
7 6 5



CODE:

```
from collections import deque

print("Shrinanda Shivprasad Dinde")
print("1BM23CS324")
DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

def to_matrix(state_list):
```

```
    return [state_list[i:i+3] for i in range(0, 9, 3)]

def state_to_tuple(state):
    return tuple(num for row in state for num in row)

def find_zero(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return -1, -1

def is_valid(x, y):
    return 0 <= x < 3 and 0 <= y < 3

def swap(state, x1, y1, x2, y2):
    new_state = [row[:] for row in state]
    new_state[x1][y1], new_state[x2][y2] = new_state[x2][y2], new_state[x1][y1]
    return new_state

def dls(current_state, goal_tuple, depth_limit, visited, parent, depth):
    """
    Depth-Limited Search:
    Returns True if goal found within depth_limit, else False.
    """
    current_tuple = state_to_tuple(current_state)
    if current_tuple == goal_tuple:
        return True

    if depth >= depth_limit:
        return False

    x, y = find_zero(current_state)

    for dx, dy in DIRECTIONS:
        nx, ny = x + dx, y + dy
        if is_valid(nx, ny):
            new_state = swap(current_state, x, y, nx, ny)
            new_tuple = state_to_tuple(new_state)
            if new_tuple not in visited:
                visited.add(new_tuple)
                parent[new_tuple] = current_tuple
                found = dls(new_state, goal_tuple, depth_limit, visited, parent, depth + 1)
                if found:
                    return True

                visited.remove(new_tuple)
```

```
    return False

def iddfs(start_state, goal_state, max_depth=50):
    start_tuple = state_to_tuple(start_state)
    goal_tuple = state_to_tuple(goal_state)
    parent = {start_tuple: None}

    for depth_limit in range(max_depth + 1):
        visited = set([start_tuple])
        print(f"Searching with depth limit = {depth_limit} ...")
        found = dls(start_state, goal_tuple, depth_limit, visited, parent, 0)
        if found:
            print(f"\n Goal found at depth {depth_limit}!\n")
            print_path(parent, start_tuple, goal_tuple)
            return
    print(" No solution found within max depth.")

def print_path(parent, start_tuple, goal_tuple):
    path = []
    current = goal_tuple
    while current is not None:
        path.append(current)
        current = parent[current]
    path.reverse()

    print(" Solution Path:")
    for state in path:
        print_state(state)
        print()

def print_state(state_tuple):
    for i in range(0, 9, 3):
        row = state_tuple[i:i+3]
        print(" ".join(str(x) if x != 0 else " " for x in row))

def read_state(prompt):
    while True:
        try:
            nums = list(map(int, input(prompt).strip().split()))
            if len(nums) != 9 or sorted(nums) != list(range(9)):
                raise ValueError
            return to_matrix(nums)
        except ValueError:
            print("Invalid input. Enter 9 numbers (0-8) with no duplicates.")

if __name__ == "__main__":
    print(" 8 Puzzle Solver using IDDFS\n")
```

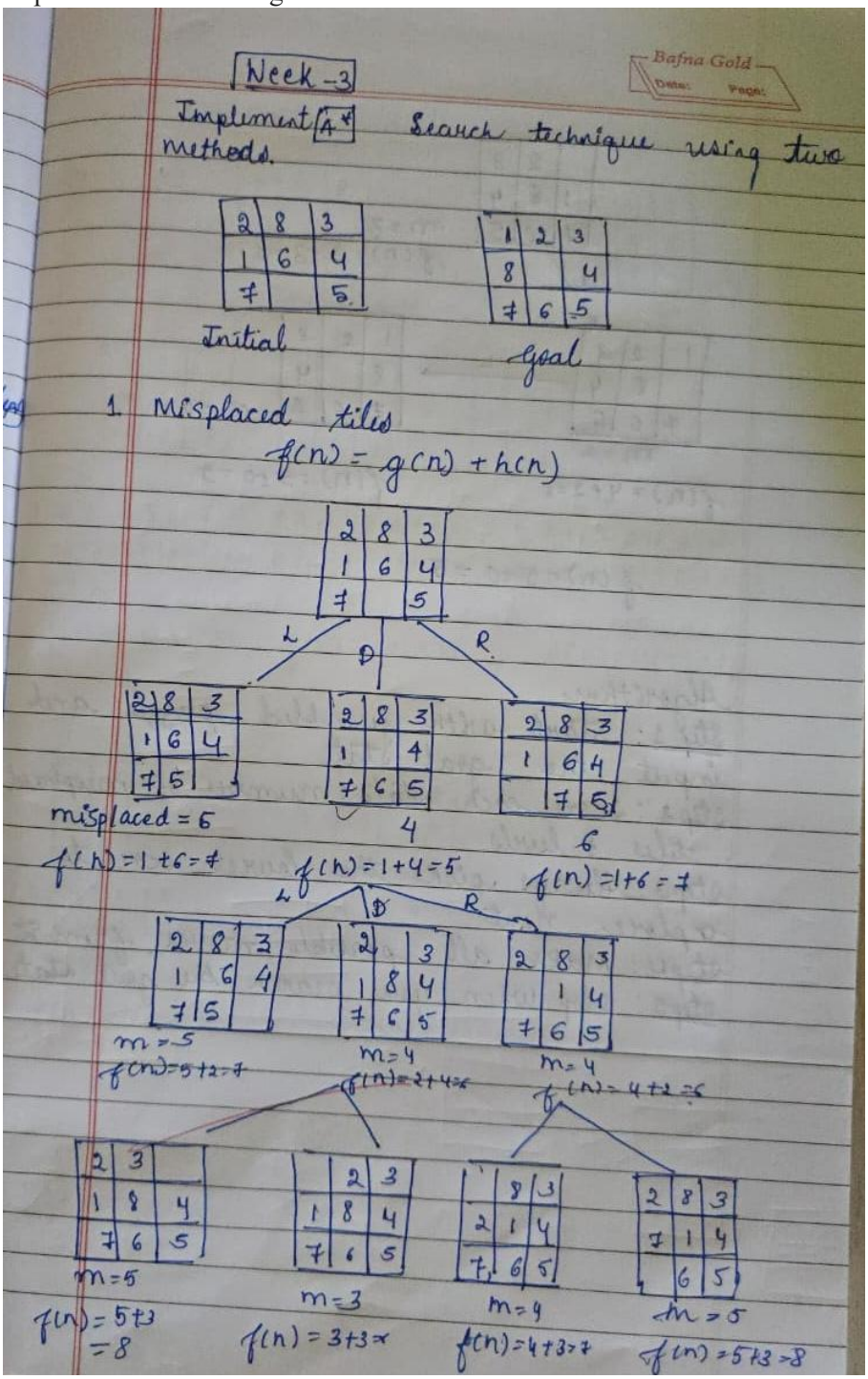
```
print("Enter the initial state (use 0 for the blank tile):")
start_state = read_state("Initial (e.g., 1 2 3 4 5 6 7 8 0): ")

print("\nEnter the goal state:")
goal_state = read_state("Goal (e.g., 1 2 3 4 5 6 7 8 0): ")

iddfs(start_state, goal_state, max_depth=30)
```

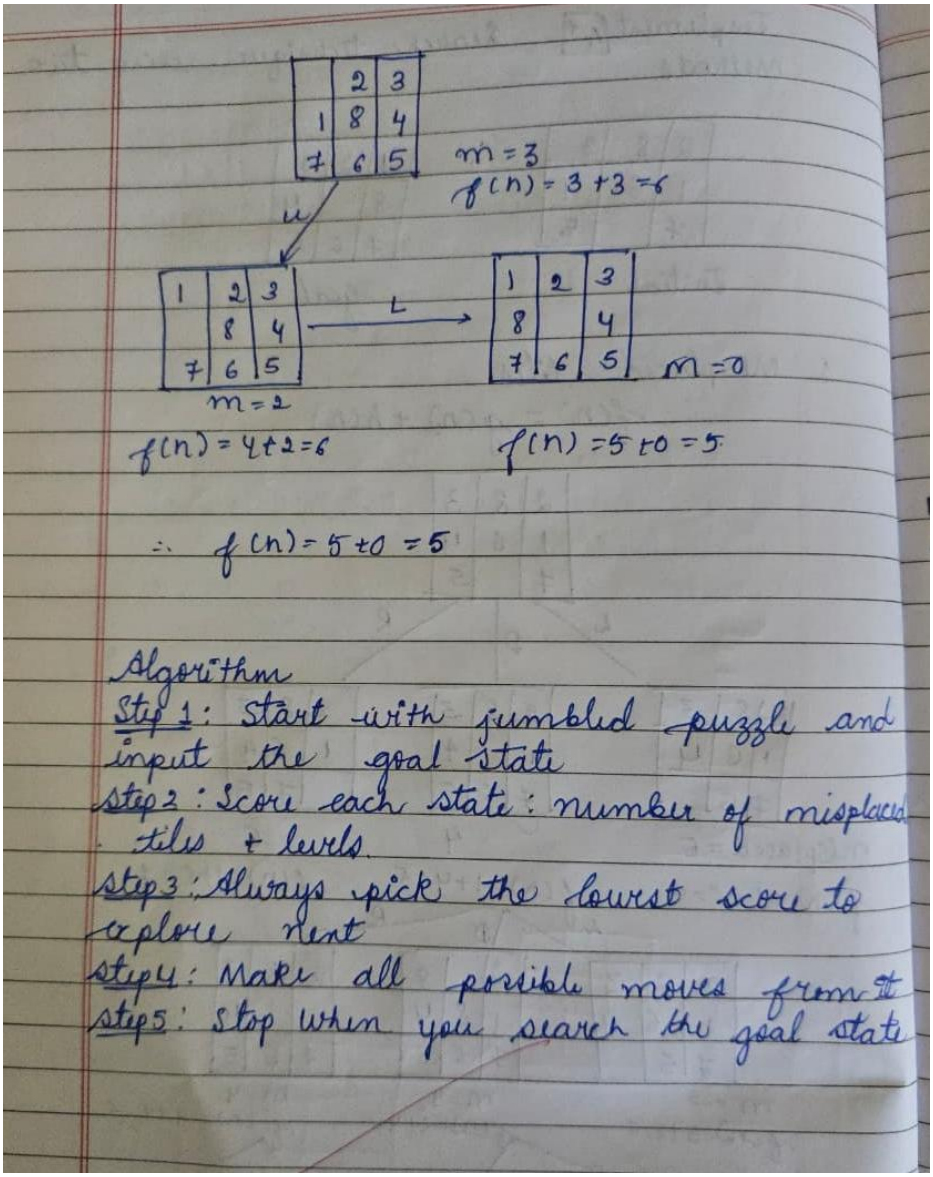
OUTPUT:

```
Shrinanda Shivprasad Dinde
1BM23CS324
 8 Puzzle Solver using IDDFS

Enter the initial state (use 0 for the blank tile):
Initial (e.g., 1 2 3 4 5 6 7 8 0): 1 2 3 0 4 6 7 5 8

Enter the goal state:
Goal (e.g., 1 2 3 4 5 6 7 8 0): 1 2 3 4 5 6 7 8 0
Searching with depth limit = 0 ...
Searching with depth limit = 1 ...
Searching with depth limit = 2 ...
Searching with depth limit = 3 ...

 Goal found at depth 3!

 Solution Path:
1 2 3
  4 6
7 5 8

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 3:

Implement A* search algorithm



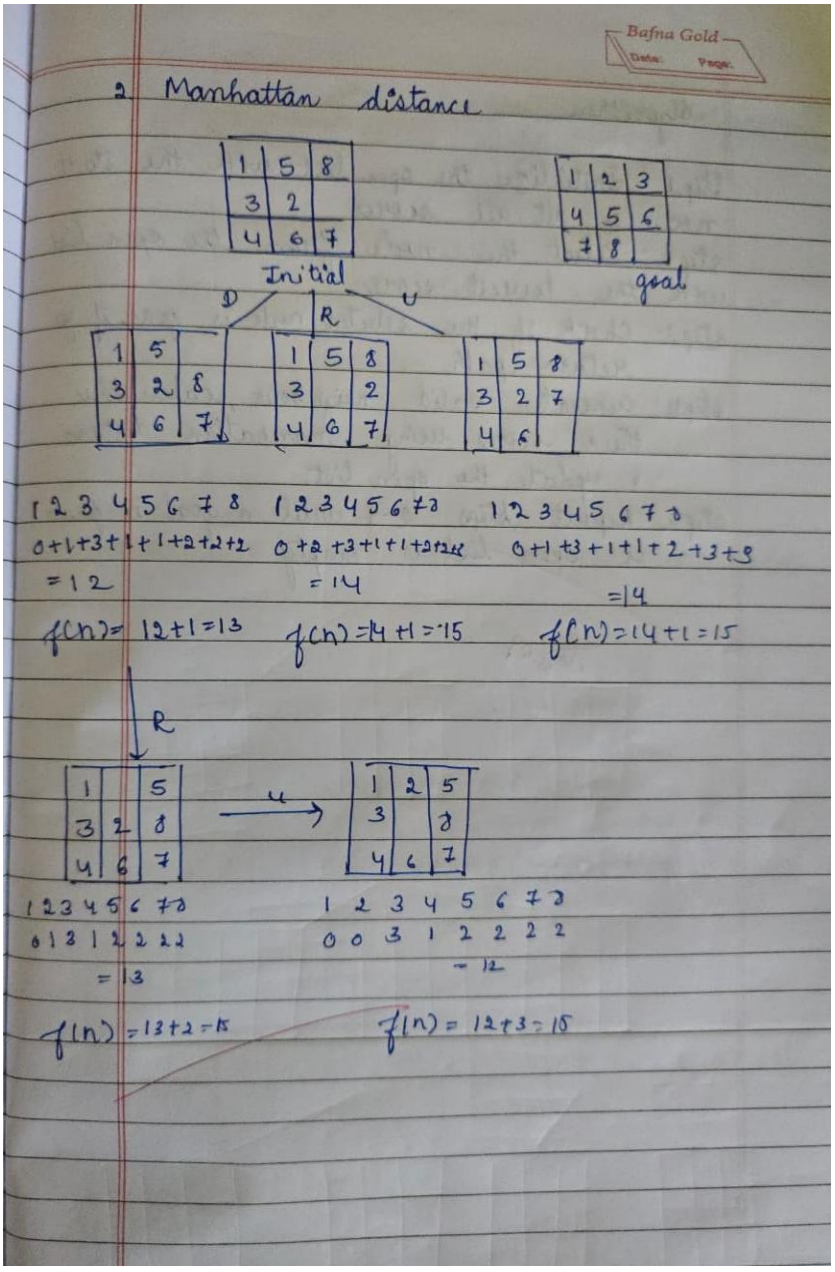


CODE:

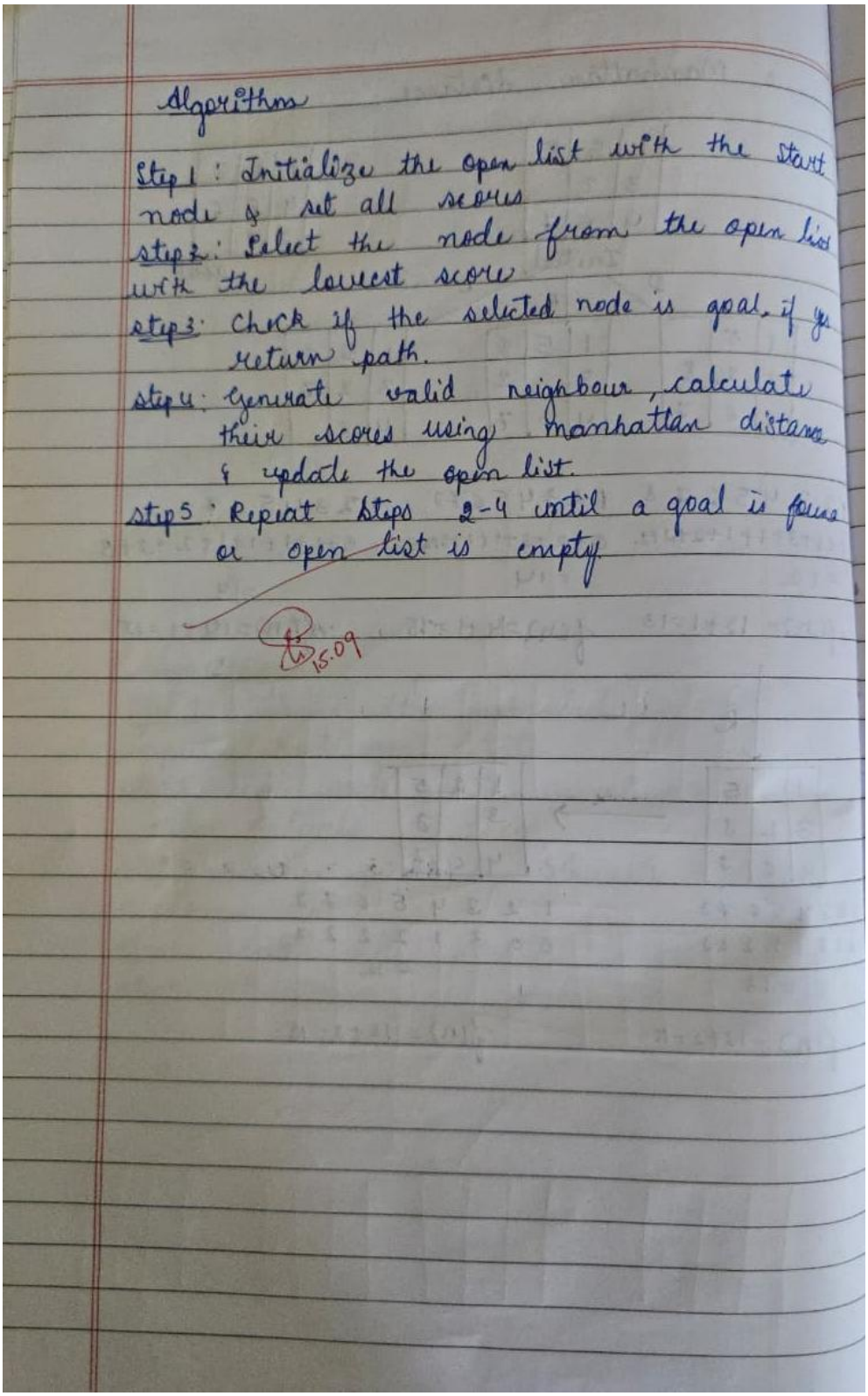
```
from heapq import heappush, heappop
print("Shrinanda Dinde")
print("1BM23CS324")



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

def misplaced_tiles(state):
    """Heuristic: count of misplaced tiles compared to the goal."""
    count = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
```

```
            count += 1
    return count

def get_neighbors(state):
    """Generate possible moves by sliding the empty tile (0) up/down/left/right."""
    neighbors = []

    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j

    directions = [(-1,0),(1,0),(0,-1),(0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:

            new_state = [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 a_star(start_state):
    """A* search using misplaced tile heuristic."""
    open_set = []
    heappush(open_set, (misplaced_tiles(start_state), 0, start_state, []))
    closed_set = set()

    while open_set:
        f, g, current, path = heappop(open_set)

        if current == GOAL_STATE:
            return path + [current]

        if current in closed_set:
            continue
        closed_set.add(current)

        for neighbor in get_neighbors(current):
            if neighbor in closed_set:
                continue
            new_g = g + 1
            new_f = new_g + misplaced_tiles(neighbor)
            heappush(open_set, (new_f, new_g, neighbor, path + [current]))

    return None
```

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

solution_path = a_star(start)
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.")
```



-



CODE:

```
import heapq
print("Shrinanda Dinde")
print("1BM23CS324")
```

```
def manhattan_distance(state, goal):
    """Calculate total Manhattan distance of all tiles from their goal positions."""
    goal_positions = {}
    for i in range(3):
        for j in range(3):
            goal_positions[goal[i][j]] = (i, j)

    dist = 0
    for i in range(3):
        for j in range(3):
            tile = state[i][j]
            if tile != 0:
                goal_i, goal_j = goal_positions[tile]
                dist += abs(i - goal_i) + abs(j - goal_j)
    return dist

def get_neighbors(state):
    neighbors = []
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                x, y = i, j
    directions = [(-1,0),(1,0),(0,-1),(0,1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [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 print_state(state):
    for row in state:
        print(' '.join(str(x) if x != 0 else ' ' for x in row))
    print()

def get_user_state(prompt):
    print(prompt)
    print("Enter 9 numbers separated by space, use 0 for blank tile (e.g. '2 8 3 1 6 4 7 0 5'):")
    while True:
        try:
            entries = list(map(int, input().strip().split()))
            if len(entries) != 9 or set(entries) != set(range(9)):
                raise ValueError
            break
        except ValueError:
```

```
            print("Invalid input! Enter exactly 9 unique digits from 0 to 8 separated by spaces.")
    return tuple(tuple(entries[i*3:(i+1)*3]) for i in range(3))

def a_star_verbose(start_state, goal_state):
    open_set = []
    start_h = manhattan_distance(start_state, goal_state)
    heapq.heappush(open_set, (start_h, 0, start_state, []))  # (f, g, state, path)

    closed_set = set()
    step_counter = 0

    while open_set:
        f, g, current, path = heapq.heappop(open_set)
        step_counter += 1

        print(f"Step {step_counter}:")
        print(f"Current state with f = g + h = {g} + {f - g} = {f}")
        print_state(current)

        if current == goal_state:
            print("Goal reached!")
            return path + [current]

        if current in closed_set:
            print("This state has already been visited. Skipping.\n")
            continue
        closed_set.add(current)

        neighbors = get_neighbors(current)
        print(f"Expanding neighbors ({len(neighbors)}):")
        for n in neighbors:
            if n not in closed_set:
                h = manhattan_distance(n, goal_state)
                new_g = g + 1
                new_f = new_g + h
                print(f"Neighbor state with g={new_g}, h={h}, f={new_f}:")
                print_state(n)
                heapq.heappush(open_set, (new_f, new_g, n, path + [current]))
            else:
                print("Neighbor already visited, skipping.")
        print("-----\n")
    return None

if __name__ == "__main__":
    start = get_user_state("Enter the START state:")
    goal = get_user_state("Enter the GOAL state:")
    print("\nStarting A* search with Manhattan distance heuristic...\n")
```

```
solution = a_star_verbose(start, goal)

if solution:
    print(f"\nSolution found in {len(solution)-1} moves:\n")
    for step_num, step in enumerate(solution):
        print(f"Step {step_num}:")
        print_state(step)
else:
    print("No solution found.")
```

PROGRAM 4:

Implement Hill Climbing search algorithm to solve N-Queens problem

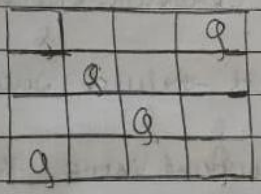
Week-4

Implement Hill climbing search Algorithm to solve N-Queens problem

Algorithm

1. Define current state
2. Loop until goal state is achieved or no more operation is applied.
3. Apply an operator
4. Compare new state with goal
5. Quit
6. Evaluate new state
7. Compare
8. If new state is close to goal state update current state

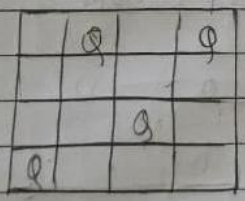
① Initial state



$\{x_0=3, x_1=1, x_2=2, x_3=0\}$

$h=2$

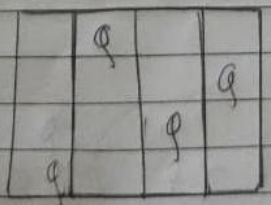
②



$\{x_0=3, x_1=0, x_2=2, x_3=0\}$

$h=1$

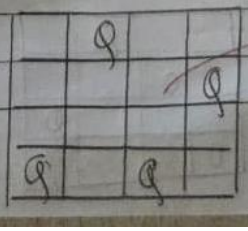
③



$\{x_0=3, x_1=0, x_2=2, x_3=1\}$

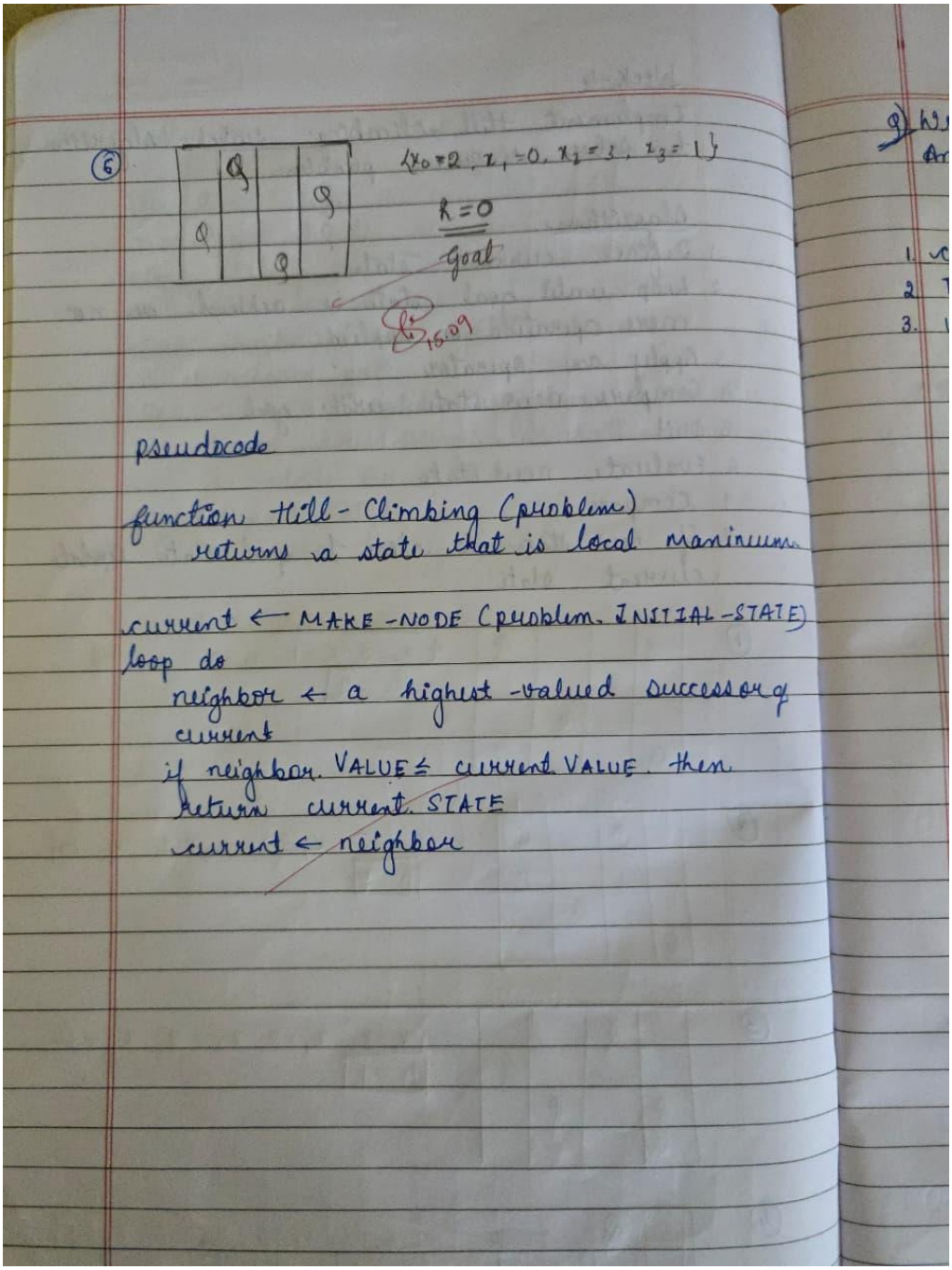
$h=1$

④



$\{x_0=3, x_1=0, x_2=3, x_3=1\}$

$h=1$




```
print("Shrinanda Dinde")
print("1BM23CS324")

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

```
def generate_neighbors(state):
    """Generate neighbors by swapping the row positions of two queens."""
    neighbors = []
    n = len(state)
    for i in range(n):
        for j in range(i+1, n):
            neighbor = state.copy()
            neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
            neighbors.append(neighbor)
    return neighbors

def hill_climbing(initial_state):
    current = initial_state
    current_cost = calculate_cost(current)
    step = 0

    print(f"Step {step}: State = {current}, Cost = {current_cost}")
    print_board(current)

    while True:
        neighbors = generate_neighbors(current)
        costs = [calculate_cost(n) for n in neighbors]

        min_cost = min(costs)
        if min_cost >= current_cost:
            # No better neighbor found, return current state
            print("No better neighbor found. Stopping.")
            break

        # Choose neighbor with minimum cost
        best_index = costs.index(min_cost)
        current = neighbors[best_index]
        current_cost = min_cost
        step += 1

        print(f"Step {step}: State = {current}, Cost = {current_cost}")
        print_board(current)

        if current_cost == 0:
            print("Goal reached!")
            break

    return current, current_cost

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

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

if __name__ == "__main__":
    import random

    initial_state = random.sample(range(4), 4)  # Random initial state
    print("Initial state:", initial_state)
    print_board(initial_state)

    solution, cost = hill_climbing(initial_state)

    print("Final state:", solution)
    print("Final cost:", cost)
    print_board(solution)
```

PROGRAM -5

Bafna Gold

Q) Write a program to implement Simulated Annealing Algorithm

```
1. current ← Initial state
2. T ← a large positive value
3. while T > 0 do
      next ← a random neighbour of current
      ΔE ← current.cost - next.cost
      if ΔE > 0 then
         current ← next
      else
         current ← next with probability p = e^(ΔE/T)
      end if
      decrease T
   end while
   return current
```

// output

Initial state : [1, 0, 2, 3]

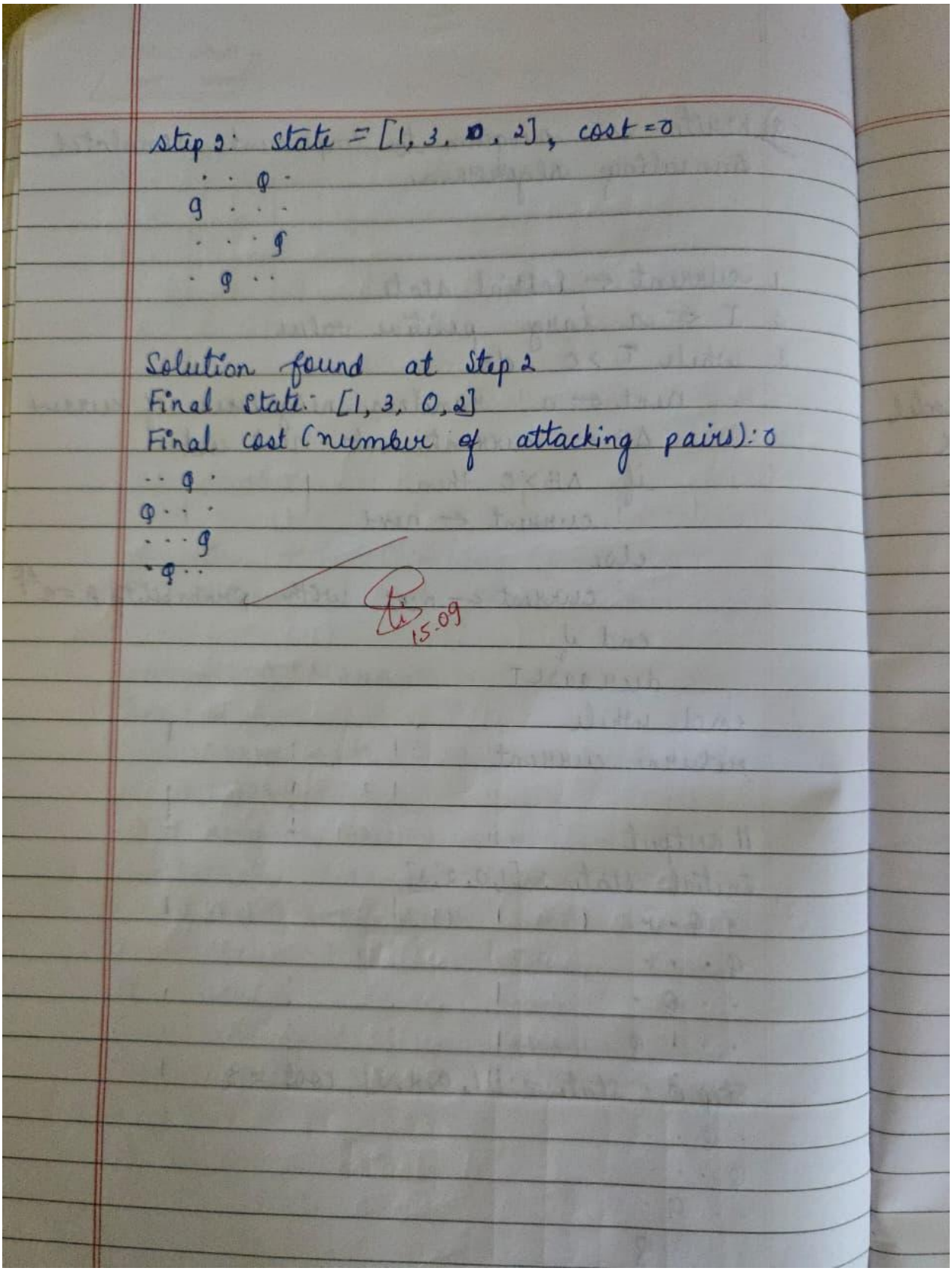
```
. q . .
q . . .
. . q .
. . . q
```

Step 0 : State = [1, 0, 2, 3], cost = 2

```
. q . .
q . . .
. . q .
. . . q
```

Step 1 : State = [1, 3, 2, 0], cost = 1

```
. . . q
q . . .
. . q .
. q . .
```




```
import random
import math
print("Shrinanda Dinde")
```

```
print("1BM23CS324")

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

def random_neighbor(state):
    """Generate a random neighbor by swapping two columns' queen positions."""
    neighbor = state.copy()
    i, j = random.sample(range(len(state)), 2)
    neighbor[i], neighbor[j] = neighbor[j], neighbor[i]
    return neighbor

def print_board(state):
    n = len(state)
    for row in range(n):
        line = ""
        for col in range(n):
            if state[col] == row:
                line += "Q "
            else:
                line += ". "
        print(line)
    print()

def simulated_annealing(initial_state, initial_temp=10000, cooling_rate=0.99, min_temp=0.1):
    current = initial_state
    current_cost = calculate_cost(current)
    T = initial_temp
    step = 0

    print(f"Step {step}: State = {current}, Cost = {current_cost}")
    print_board(current)

    while T > min_temp:
        next_state = random_neighbor(current)
        next_cost = calculate_cost(next_state)
        delta_E = current_cost - next_cost

        if delta_E > 0:
            current = next_state
```

```
            current_cost = next_cost
        else:
            p = math.exp(delta_E / T)
            if random.random() < p:
                current = next_state
                current_cost = next_cost

        step += 1
        print(f"Step {step}: State = {current}, Cost = {current_cost}")
        print_board(current)

        T *= cooling_rate

        if current_cost == 0:
            print(f"Solution found at step {step}")
            break

    return current, current_cost

if __name__ == "__main__":
    initial_state = random.sample(range(4), 4)
    print("Initial state:", initial_state)
    print_board(initial_state)

    solution, cost = simulated_annealing(initial_state)

    print("Final state:", solution)
    print("Final cost (number of attacking pairs):", cost)
    print_board(solution)
```

Propositional Logic Week-6

Implementation of truth-table enumeration algorithm for deciding propositional entailment i.e

Truth tables for connectives

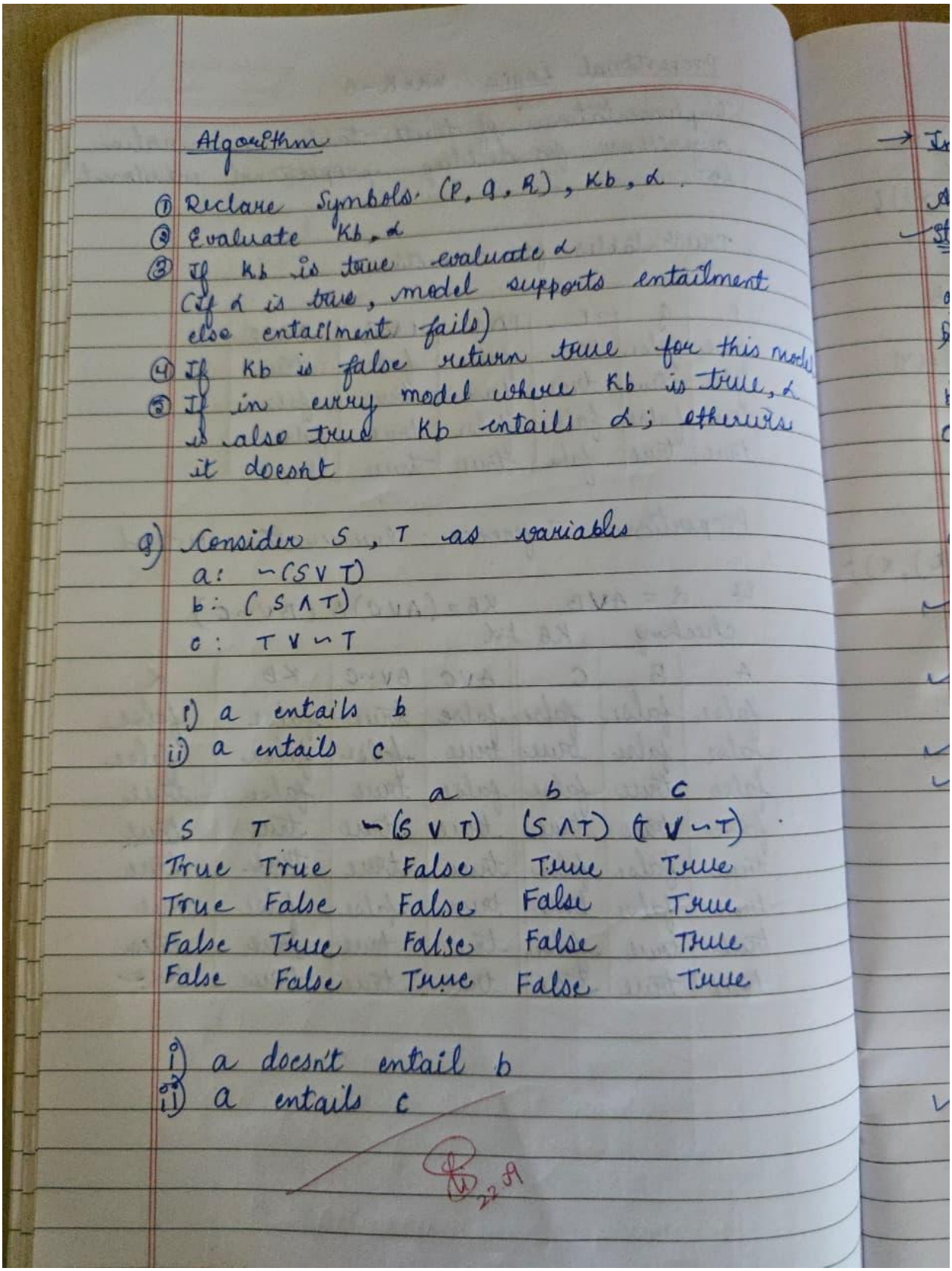
P	Q	¬P	P∧Q	P∨Q	P⇔Q
false	false	true	false	false	true
false	true	true	false	true	false
true	false	false	false	true	false
true	true	false	true	true	true

Propositional Inference: Enumeration Method

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

Checking $KB \models \alpha$

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



Algorithm

① Declare Symbols (P, Q, R), Kb, α
② Evaluate Kb, α
③ If Kb is true evaluate α
(If α is true, model supports entailment
else entailment fails)
④ If Kb is false return true for this model
⑤ If in every model where Kb is true, α
is also true Kb entails α; otherwise
it doesn't

Q) Consider S, T as variables
a: ¬(S ∨ T)
b: (S ∧ T)
c: T ∨ ¬T

i) a entails b
ii) a entails c

		a	b	c
S	T	¬(S ∨ T)	(S ∧ T)	(T ∨ ¬T)
True	True	False	True	True
True	False	False	False	True
False	True	False	False	True
False	False	True	False	True

i) a doesn't entail b
ii) a entails c

```
import itertools
print("Shrinanda Shivprasad Dinde")
print("1BM23CS324")
def truth_table_example():
```

```
    symbols = ["A", "B", "C"]

    entails = True

    header = ["A", "B", "C", "A∨C", "B∨¬C", "KB", "α"]
    print(" | ".join(f"{h:6}" for h in header))
    print("-" * (9 * len(header)))

    for values in itertools.product([False, True], repeat=3):
        A, B, C = values

        A_or_C = A or C
        B_or_notC = B or (not C)
        KB = A_or_C and B_or_notC
        alpha = A or B

        row = [A, B, C, A_or_C, B_or_notC, KB, alpha]
        print(" | ".join(f"{str(r):6}" for r in row))

        if KB and not alpha:
            entails = False

    print("\nFinal Result:")
    if entails:
        print("KB entails α ☑ (KB ⊨ α)")
    else:
        print("KB does NOT entail α ❌ (KB ⊭ α)")

truth_table_example()
```

First Order Logic

Week-7

→ Implement unification in first order logic

Algorithm Unify (ψ_1, ψ_2)

Step 1: If ψ_1 or ψ_2 is a variable or constant then:

a) If ψ_1 or ψ_2 are identical, then return NIL

b) Else if ψ_1 is variable

a). then if ψ occurs in ψ_2, then return FAILURE

b) Else return $\{(\psi_2 / \psi_1)\}$.

c) Else if ψ_2 is a variable

a. If ψ_2 occurs in ψ_1 then return FAILURE.

b Else return $\{(\psi_1 / \psi_2)\}$

d) Else return FAILURE.

Step 2: If the initial Predicate symbol in ψ_1 and ψ_2 are not same, then return FAILURE

Step 3: If ψ_1 and ψ_2 have different number of arguments, return FAILURE

Step 4: Set Substitution set (SUBST) to NIL

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

a) call unify function with the i^{th} element of ψ_1 and i^{th} element of ψ_2, and put result into S.

b) If S = failure the return Failure

c) If S ≠ NIL then do.

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

b. SUBST = APPEND (S, SUBST).

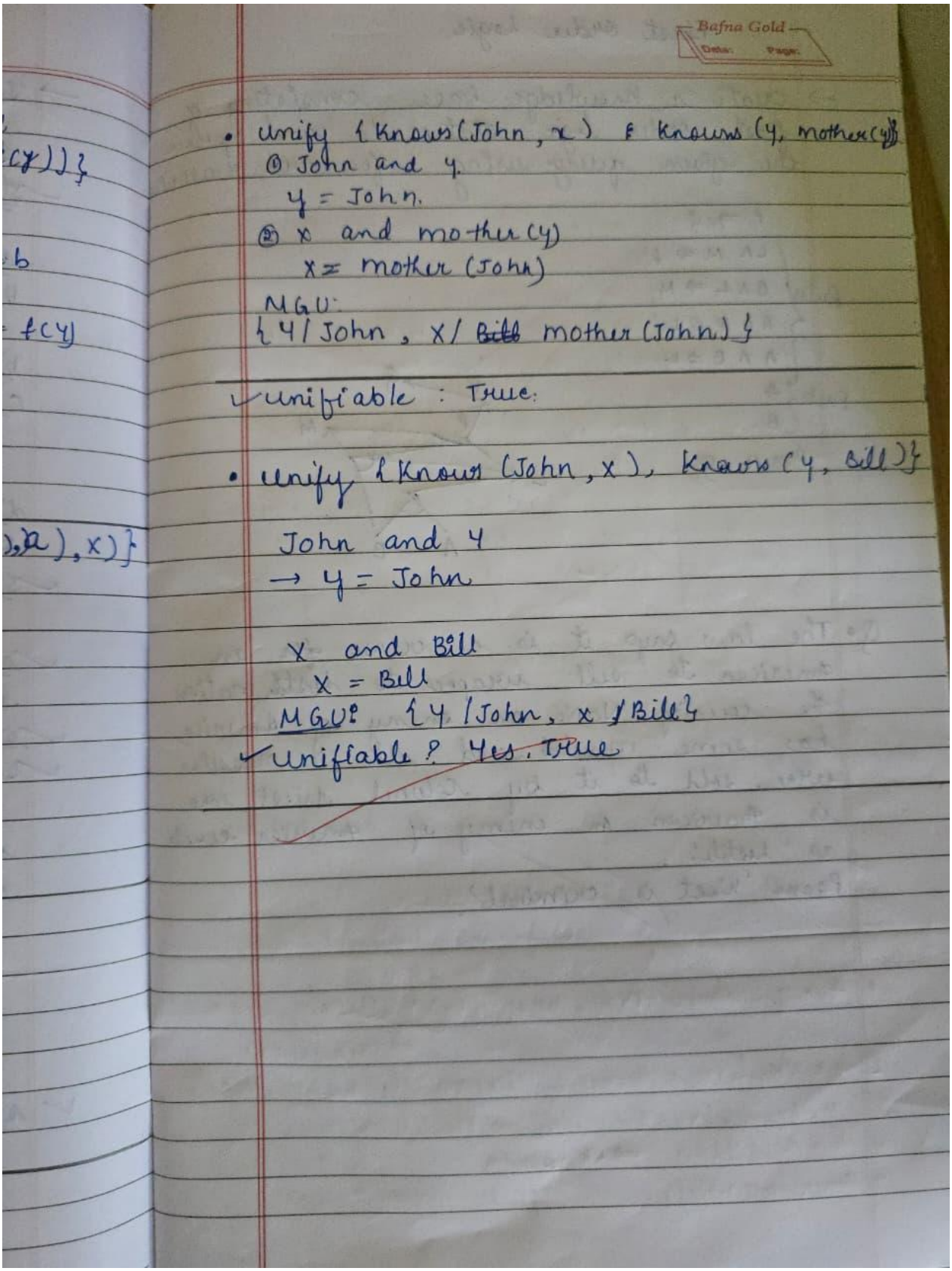
Step 6: Return SUBST.

- Find Most General Unifier (MGU) of
  {p(b, X, f(g(z)) and p(z, f(y), f(x))}
  3 arguments
  ① b and z
  b constant, z ~~const~~ variable → z = b
  ② x and f(y)
  x variable f(y) function → x = f(y)
  ③ f(g(z)) and f(y)
  y = g(z)
  MGU: {z/b, x/f(g(z)), y/g(z)}
  ✓unifiable True

- {q(a, g(x, a), f(y)), q(a, g(f(b), a), x)}
  a → a same.
  g(x, a) and g(f(b), a)
  x = f(b)
  f(y) and x
  f(y) = f(b) y = b
  MGU:
  {x/f(b), y/b}
  ✓unifiable: True

- {p(f(a), g(y)), p(x, x)}
  f(a) = x
  g(y) = x
  f(a) = g(y)
  f & g have different names
  ✗ unification fails

- unify {prime(11) and prime(y)}
  y = 11
  unifiable: True




```
print("Shrinanda Shivprasad Dinde")
print("USN: 1BM23CS324")

def occurs_check(var, term, subst):
```

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

def apply_substitution(term, subst):
    if isinstance(term, str):
        return subst.get(term, term)
    elif isinstance(term, tuple):
        return tuple(apply_substitution(t, subst) for t in term)
    else:
        return term

def unify(x, y, subst=None):
    if subst is None:
        subst = {}

    x = apply_substitution(x, subst)
    y = apply_substitution(y, subst)

    if x == y:
        return subst
    elif isinstance(x, str) and x.islower():  # variable
        if occurs_check(x, y, subst):
            return None  # failure
        subst[x] = y
        return subst
    elif isinstance(y, str) and y.islower():  # variable
        if occurs_check(y, x, subst):
            return None
        subst[y] = x
        return subst
    elif isinstance(x, tuple) and isinstance(y, tuple):
        if len(x) != len(y):
            return None
        for x_i, y_i in zip(x, y):
            subst = unify(x_i, y_i, subst)
            if subst is None:
                return None
        return subst
    else:
```

```
        return None

def is_unifiable(x, y):
    result = unify(x, y)
    return result is not None


# Example: knows(John, x) and knows(y, Bill)
term1 = ('knows', 'John', 'x')
term2 = ('knows', 'y', 'Bill')

result = unify(term1, term2)

if result is None:
    print("No unifier exists.")
else:
    print("MGU:", result)

print("Are terms unifiable?", is_unifiable(term1, term2))
```

PROGRAM-8

First order Logic.

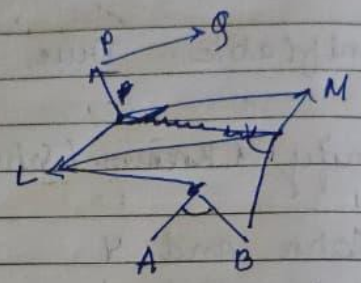
→ Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning

$P \Rightarrow Q$

Rules:
- $L \wedge M \Rightarrow P$
- $B \wedge L \Rightarrow M$
- $A \wedge P \Rightarrow L$
- $A \wedge B \Rightarrow L$

Facts:
- A
- B

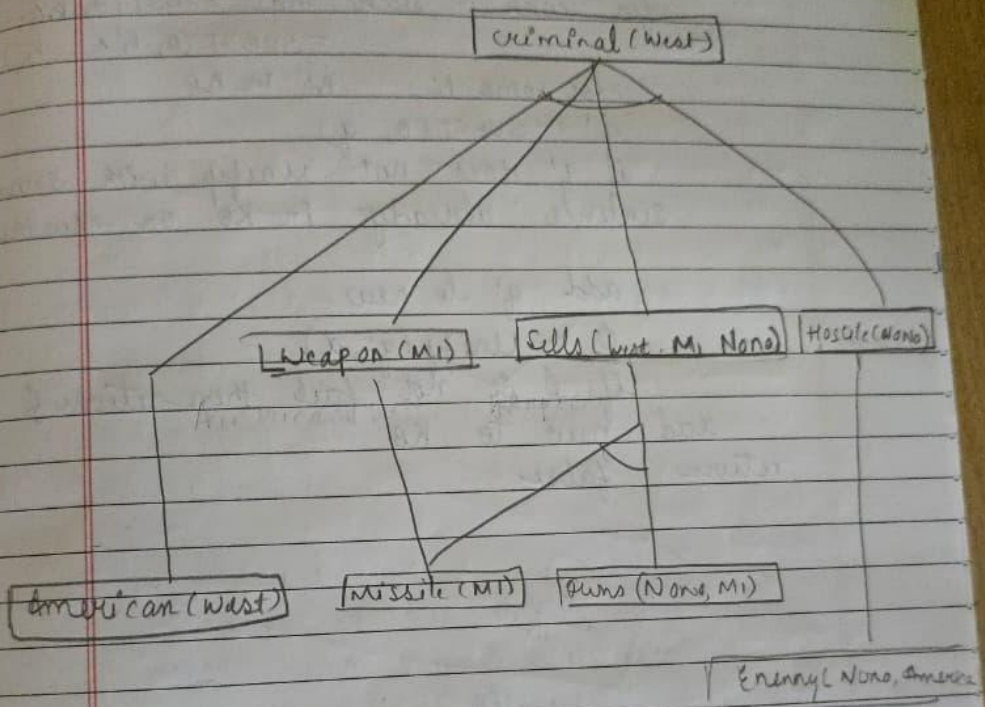
Prove Q



Q. The law says it is a crime for an American to sell weapons to hostile nations. The country Nono an enemy of America, has some missiles and all its missiles were sold to it by Colonel West who is American. An enemy of America counts as 'hostile'.
Prove 'West is criminal'.

1. $\forall x, y, z$ American$(x) \wedge$ Weapon$(y) \wedge$ Sells$(x, y, z) \wedge$ Hostile$(z) \Rightarrow$ Criminal(x)
2. $\forall x$ Missile$(x) \wedge$ Owns$(Nono, x) \Rightarrow$ Sells$(West, x, Nono)$
3. $\forall x$ Enemy$(x, America) \Rightarrow$ Hostile(x)
4. $\forall x$ Missile$(x) \Rightarrow$ Weapon(x)
5. American(West)

6 Enemy (Nono, America)
7 Owns (Nono, M1) and
8 Missile (M1)

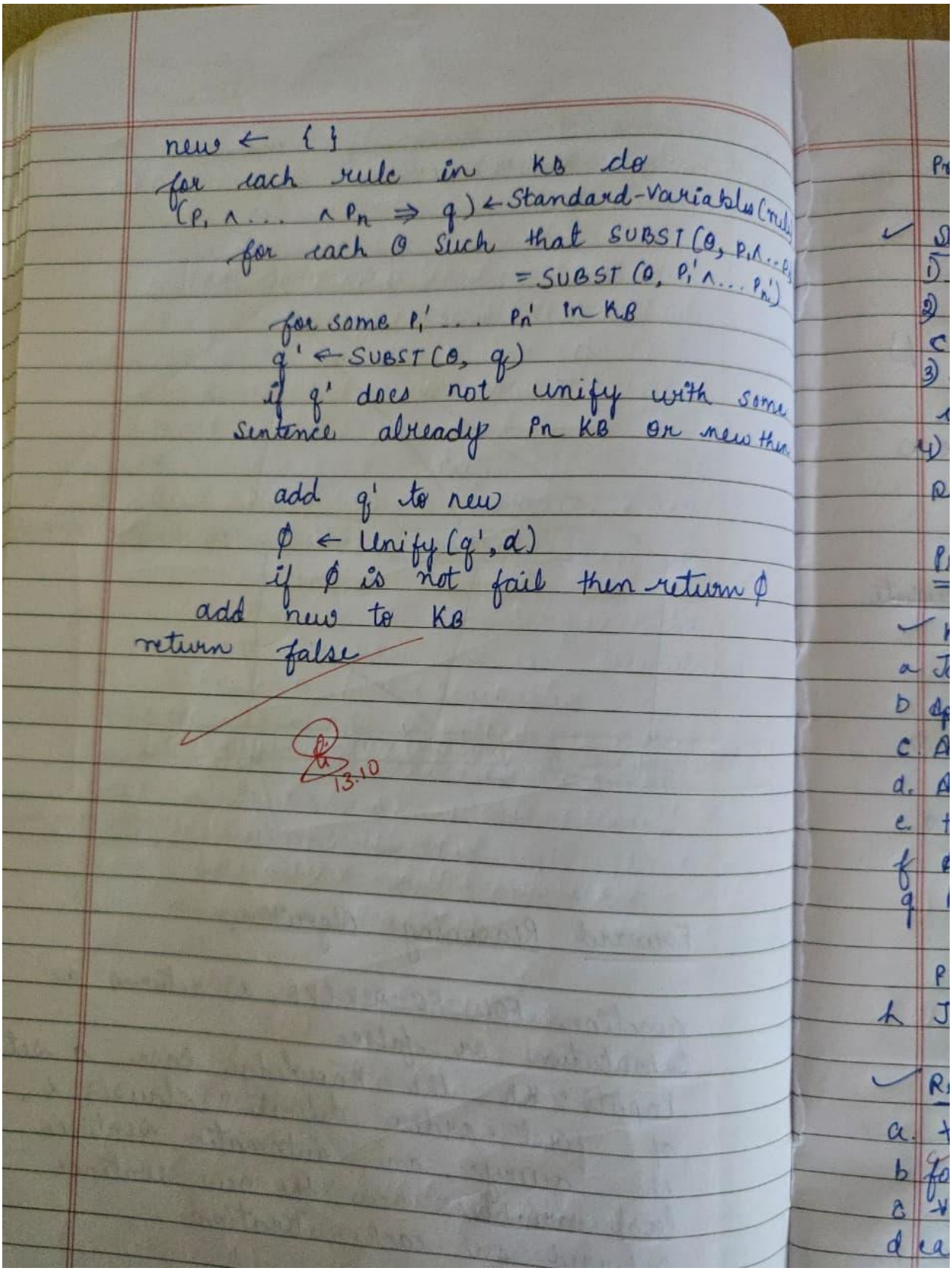


Forward Reasoning Algorithm

function FOL-FC-Ask (KB, α) returns a substitution or false

Inputs: KB, the knowledge base, a set of first-order definite clauses
α, the query, an automatic sentence
local variables: new, the new sentence inferred on each iteration

repeat until new is empty




```
print("Shrinanda Shivprasad Dinde")
print("USN: 1BM23CS324")
class Fact:
    def __init__(self, predicate, *args):
```

```
        self.predicate = predicate
        self.args = args

    def __repr__(self):
        return f"{self.predicate}({', '.join(self.args)})"

    def __eq__(self, other):
        return isinstance(other, Fact) and self.predicate == other.predicate and self.args == other.args

    def __hash__(self):
        return hash((self.predicate, self.args))

class Rule:
    def __init__(self, premises, conclusion):
        # premises: list of Fact objects
        # conclusion: Fact object
        self.premises = premises
        self.conclusion = conclusion

    def __repr__(self):
        premises_str = " ∧ ".join(map(str, self.premises))
        return f"{premises_str} ⇒ {self.conclusion}"

# Initialize facts (atomic sentences)
facts = {
    Fact("American", "Robert"),
    Fact("Enemy", "A", "America"),
    Fact("Missile", "T1"),
    Fact("Owns", "A", "T1"),
}

# Rules (implications)
rules = [
    Rule([Fact("Missile", "x")], Fact("Weapon", "x")),  # Missile(x) ⇒ Weapon(x)
    Rule([Fact("Enemy", "x", "America")], Fact("Hostile", "x")),  # Enemy(x, America) ⇒ Hostile(x)
    Rule([Fact("Missile", "x"), Fact("Owns", "A", "x")], Fact("Sells", "Robert", "x", "A")),  # ∀x Missile(x) ∧ Owns(A,x) ⇒
Sells(Robert, x, A)
    Rule([Fact("American", "p"), Fact("Weapon", "q"), Fact("Sells", "p", "q", "r"), Fact("Hostile", "r")], Fact("Criminal", "p"))
# American(p) ∧ Weapon(q) ∧ Sells(p,q,r) ∧ Hostile(r) ⇒ Criminal(p)
]

def match(fact1, fact2):
    """Match two facts allowing variables (starting with lowercase) to be replaced."""
    if fact1.predicate != fact2.predicate or len(fact1.args) != len(fact2.args):
        return None
    substitution = {}
    for arg1, arg2 in zip(fact1.args, fact2.args):
```

```
        if arg1.islower():
            if arg1 in substitution and substitution[arg1] != arg2:
                return None
            substitution[arg1] = arg2
        else:
            if arg1 != arg2:
                return None
    return substitution

def substitute(fact, substitution):
    """Substitute variables in fact according to substitution dict."""
    new_args = [substitution.get(arg, arg) for arg in fact.args]
    return Fact(fact.predicate, *new_args)

def forward_chaining(facts, rules):
    inferred = set(facts)
    new_inferred = True
    while new_inferred:
        new_inferred = False
        for rule in rules:

            substitutions_list = [{}]
            for premise in rule.premises:
                new_substitutions = []
                for substitution in substitutions_list:

                    substituted_premise = substitute(premise, substitution)
                    matched = False
                    for fact in inferred:
                        match_substitution = match(substituted_premise, fact)
                        if match_substitution is not None:

                            combined_substitution = substitution.copy()
                            combined_substitution.update(match_substitution)
                            new_substitutions.append(combined_substitution)
                            matched = True

                substitutions_list = new_substitutions

            for substitution in substitutions_list:
                conclusion_fact = substitute(rule.conclusion, substitution)
                if conclusion_fact not in inferred:
                    print(f"Inferred new fact: {conclusion_fact} using rule: {rule}")
                    inferred.add(conclusion_fact)
                    new_inferred = True
    return inferred
```

```
final_facts = forward_chaining(facts, rules)

criminal_fact = Fact("Criminal", "Robert")

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

print("\nIs Robert Criminal?")
print(criminal_fact in final_facts)
```

First Order Logic

Bafna Gold

Prove given query using Resolution

✓ Steps for resolution in FOL
1) Convert all sentences to CNF
2) Negate conclusion S & convert result to CNF
3) Add negated conclusion S to premise clauses
4) Repeat until contradiction or no progress is made

Proof by Resolution

✓ KB
a. John likes all kind of food
b. Apple and vegetables are food
c. Anything anyone eats and not killed is food
d. Anil eats peanuts and still alive
e. Harry eats everything that Anil eats
f. Anyone who is alive implies not killed
g. Anyone who is not killed implies alive

Prove
h. John likes peanuts

✓ Representation in FOL
a. $\forall x : food(x) \rightarrow likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y : eats(x, y) \wedge \neg killed(x) \rightarrow food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x : eats(Anil, x) \rightarrow eats(Harry, x)$
f. $\forall x : \neg killed(x) \rightarrow alive(x)$
g. $\forall x : alive(x) \rightarrow \neg killed(x)$
h. $likes(John, Peanuts)$

Eliminate implication
$\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

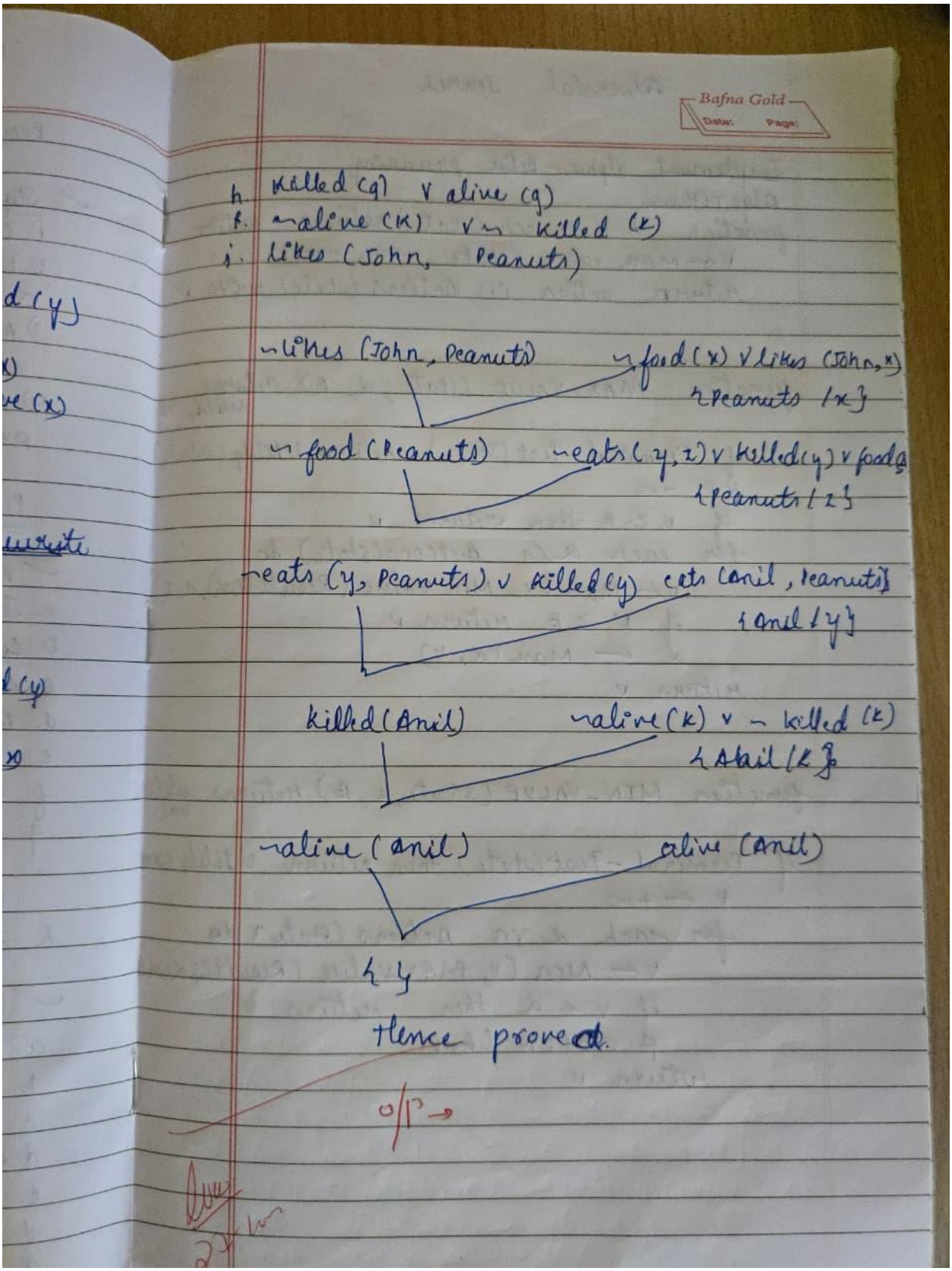
a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetable$
c. $\forall x \forall y \neg[eats(x,y) \wedge \neg killed(x)] \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x \neg[\neg killed(x)) \vee$ ~~eats~~ $Harry$ $alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

move negation ($\neg$) inwards and rewrite

a. $\forall x \neg food(x) \vee likes(John, x)$
b. $food(Apple) \wedge food(vegetables)$
c. $\forall x \forall y \neg eats(x,y) \vee killed(x) \vee food(y)$
d. $eats(Anil, Peanuts) \wedge alive(Anil)$
e. $\forall x \neg eats(Anil, x) \vee eats(Harry, x)$
f. $\forall x. killed(x) \vee alive(x)$
g. $\forall x \neg alive(x) \vee \neg killed(x)$
h. $likes(John, Peanuts)$

Drop universal quantifiers

a. $\neg food(x) \vee likes(John, x)$
b. $food(Apple)$
c. $food(vegetables)$
d. $\neg eats(y, z) \vee killed(y) \vee food(z)$
e. $eats(Anil, Peanuts)$
f. $alive(Anil)$
g. $\neg eats(Anil, w) \vee eats(Harry, w)$



print("Shrinanda Shivprasad Dinde")
print("USN:1BM23CS324")

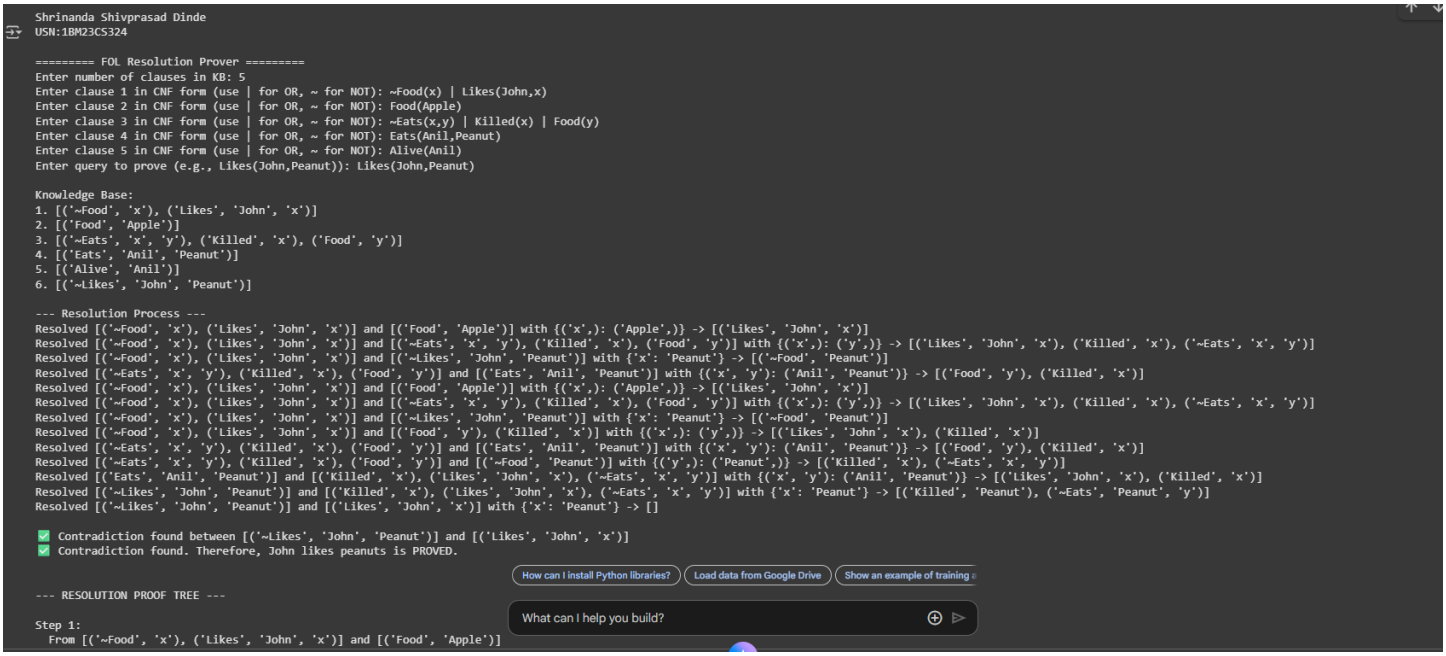
from copy import deepcopy

```
# ---------- Unification Utilities ----------
def is_variable(term):
    return term[0].islower()

def unify(x, y, subs=None):
    if subs is None:
        subs = {}
    if x == y:
        return subs
    if is_variable(x):
        return unify_var(x, y, subs)
    if is_variable(y):
        return unify_var(y, x, subs)
    if 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:]):
            subs = unify(a, b, subs)
            if subs is None:
                return None
        return subs
    return None

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

def occurs_check(var, x, subs):
    if var == x:
        return True
    elif isinstance(x, tuple):
        return any(occurs_check(var, xi, subs) for xi in x[1:])
    elif x in subs:
        return occurs_check(var, subs[x], subs)
    return False

def substitute(clause, subs):
    new_clause = []
    for lit in clause:
```

```
        pred, *args = lit
        new_args = tuple(subs.get(arg, arg) for arg in args)
        new_clause.append((pred, *new_args))
    return new_clause

# ---------- Resolution ----------
proof_steps = []  # store proof history

def resolve(ci, cj):
    for li in ci:
        for lj in cj:
            if li[0] == '~' + lj[0] or lj[0] == '~' + li[0]:
                subs = unify(li[1:], lj[1:])
                if subs is not None:
                    new_ci = [l for l in ci if l != li]
                    new_cj = [l for l in cj if l != lj]
                    resolvent = substitute(list(set(new_ci + new_cj)), subs)
                    step = {
                        "parents": (ci, cj),
                        "sub": subs,
                        "result": resolvent
                    }
                    proof_steps.append(step)
                    print(f"Resolved {ci} and {cj} with {subs} -> {resolvent}")
                    return resolvent
    return None

def resolution(KB):
    new = set()
    while True:
        n = len(KB)
        for i in range(n):
            for j in range(i + 1, n):
                resolvent = resolve(KB[i], KB[j])
                if resolvent == []:
                    print(f"\n☑ Contradiction found between {KB[i]} and {KB[j]}")
                    print("☑ Contradiction found. Therefore, John likes peanuts is PROVED.")
                    proof_steps.append({
                        "parents": (KB[i], KB[j]),
                        "sub": {},
                        "result": []
                    })
                    return True
                if resolvent is not None:
                    new.add(tuple(sorted(resolvent)))
        new_clauses = [list(c) for c in new if list(c) not in KB]
        if not new_clauses:
```

```
            return False
        KB.extend(new_clauses)

def parse_clause(text):
    text = text.replace(" ", "")
    literals = text.split("|")
    clause = []
    for lit in literals:
        neg = lit.startswith("~")
        lit = lit[1:] if neg else lit
        pred, args = lit.split("(")
        args = args.strip(")").split(",")
        pred = "~" + pred if neg else pred
        clause.append((pred, *args))
    return clause

# ---------- Proof Tree Printing ----------
def print_proof_tree():
    print("\n\n--- RESOLUTION PROOF TREE ---")
    for i, step in enumerate(proof_steps):
        p1, p2 = step["parents"]
        subs = step["sub"]
        result = step["result"]
        print(f"\nStep {i+1}:")
        print(f"  From {p1} and {p2}")
        print(f"  Substitution: {subs}")
        print(f"  ⇒ {result}")
    if proof_steps and proof_steps[-1]["result"] == []:
        print("\n{ }  Hence proved.")

# ---------- MAIN EXECUTION ----------
print("\n========= FOL Resolution Prover =========")
n = int(input("Enter number of clauses in KB: "))
KB = []
for i in range(n):
    text = input(f"Enter clause {i+1} in CNF form (use | for OR, ~ for NOT): ")
    KB.append(parse_clause(text))

goal = input("Enter query to prove (e.g., Likes(John,Peanut)): ")
neg_goal = "~" + goal if not goal.startswith("~") else goal[1:]
KB.append(parse_clause(neg_goal)[0:1])  # add negated goal

print("\nKnowledge Base:")
for i, c in enumerate(KB, 1):
    print(f"{i}. {c}")

print("\n--- Resolution Process ---")
```

```
if resolution(KB):
    print_proof_tree()
else:
    print("\n❌ No contradiction found. Query cannot be proved.")
```



Adversial Search

Implement alpha-Beta pruning.

Algorithm

function A-B-Search (State) returns action
  v ← Max-value (state, -∞, +∞)
  return action in Actions (state) with v

function MAX-VALUE (State, α, β) returns utility
  if Terminal-Test (State) return Utility (State)
  v ← -∞
  if v ≥ β then return v
  for each a in Actions (state) do
    v ← Max (v, Min Value (Result (s, a), α, β))
    if v ≥ β return v
    α ← Max (α, v)
  return v

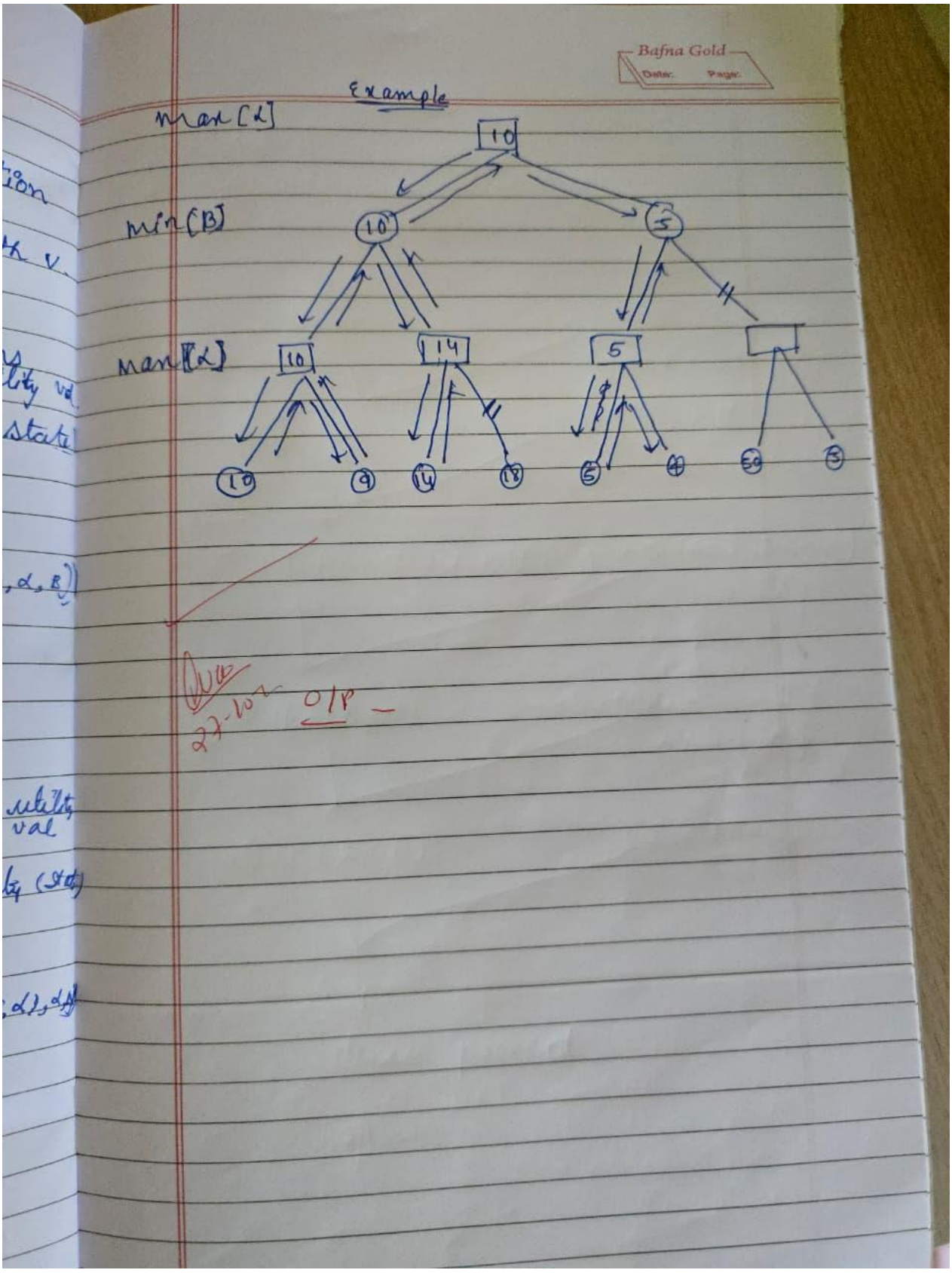
function MIN-VALVE (state, α, β) returns utility val
  if Terminal-Test (State) then return Utility (s
  v ← +∞
  for each a in Actions (state) do
    v ← Min (v, MAX-value (Result (s, a),
    if v ≤ α then return v
    β ← MIN (β, v)
  return v

max [α]

min [β]

max [α]

19




```
import math
print("Shrinanda Shivprasad Dinde")
print("1BM23CS324")
```

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

def alpha_beta(node, alpha, beta, maximizing, trace):
    if not node.children:
        trace.append(f"Reached leaf {node.name} → value = {node.value}")
        return node.value

    if maximizing:
        max_eval = -math.inf
        trace.append(f"MAX node {node.name} starts with α={alpha}, β={beta}")
        for child in node.children:
            eval_val = alpha_beta(child, alpha, beta, False, trace)
            max_eval = max(max_eval, eval_val)
            alpha = max(alpha, eval_val)
            trace.append(f"MAX node {node.name} updated α={alpha}, β={beta}")

            if beta <= alpha:
                trace.append(f" ⚠ PRUNED remaining children of {node.name} (α={alpha}, β={beta})")
                break
        return max_eval
    else:
        min_eval = math.inf
        trace.append(f"MIN node {node.name} starts with α={alpha}, β={beta}")
        for child in node.children:
            eval_val = alpha_beta(child, alpha, beta, True, trace)
            min_eval = min(min_eval, eval_val)
            beta = min(beta, eval_val)
            trace.append(f"MIN node {node.name} updated α={alpha}, β={beta}")

            if beta <= alpha:
                trace.append(f" ⚠ PRUNED remaining children of {node.name} (α={alpha}, β={beta})")
                break
        return min_eval

def build_tree():
    num_levels = int(input("Enter the number of levels in the tree (including leaf level): "))

    if num_levels < 2:
        print("Tree must have at least 2 levels (root and leaves).")
        return None

    level_types = []
    for i in range(num_levels - 1):
```

```
        t = input(f"Is level {i+1} a MAX or MIN level? (Enter MAX/MIN): ").strip().upper()
        level_types.append(t)

    num_leaves = int(input("Enter number of leaf nodes: "))
    leaf_values = []
    for i in range(num_leaves):
        v = int(input(f"Enter value for leaf node L{i+1}: "))
        leaf_values.append(Node(f"L{i+1}", value=v))

    current_level = leaf_values
    for depth in range(num_levels - 2, -1, -1):
        new_level = []
        level_type = level_types[depth]
        node_prefix = f"{level_type[0]}{depth+1}"
        for i in range(0, len(current_level), 2):
            children = current_level[i:i + 2]
            node_name = f"{node_prefix}_{i//2 + 1}"
            new_level.append(Node(node_name, children))
        current_level = new_level

    root = current_level[0]
    print("\n☑ Tree built successfully!\n")
    return root


if __name__ == "__main__":
    root = build_tree()
    if root:
        trace_output = []
        print("\nStarting Alpha-Beta Pruning...\n")
        best_value = alpha_beta(root, -math.inf, math.inf, True, trace_output)

        print("\n===== TRACE OUTPUT =====")
        for step in trace_output:
            print(step)

        print("\n===== FINAL RESULT =====")
        print(f"Best value at root: {best_value}")
```

```
Shrinanda Shivprasad Dinde
1BM23CS324
Enter the number of levels in the tree (including leaf level): 4
Is level 1 a MAX or MIN level? (Enter MAX/MIN): max
Is level 2 a MAX or MIN level? (Enter MAX/MIN): min
Is level 3 a MAX or MIN level? (Enter MAX/MIN): max
Enter number of leaf nodes: 8
Enter value for leaf node L1: 10
Enter value for leaf node L2: 9
Enter value for leaf node L3: 14
Enter value for leaf node L4: 18
Enter value for leaf node L5: 5
Enter value for leaf node L6: 4
Enter value for leaf node L7: 50
Enter value for leaf node L8: 3

✅ Tree built successfully!


Starting Alpha-Beta Pruning...


===== TRACE OUTPUT =====
MAX node M1_1 starts with α=-inf, β=inf
MIN node M2_1 starts with α=-inf, β=inf
MAX node M3_1 starts with α=-inf, β=inf
Reached leaf L1 → value = 10
MAX node M3_1 updated α=10, β=inf
Reached leaf L2 → value = 9
MAX node M3_1 updated α=10, β=inf
MIN node M2_1 updated α=-inf, β=10
MAX node M3_2 starts with α=-inf, β=10
Reached leaf L3 → value = 14
MAX node M3_2 updated α=14, β=10
⚠️ PRUNED remaining children of M3_2 (α=14, β=10)
MIN node M2_1 updated α=-inf, β=10
MAX node M1_1 updated α=10, β=inf
MIN node M2_2 starts with α=10, β=inf
MAX node M3_3 starts with α=10, β=inf
Reached leaf L5 → value = 5
MAX node M3_3 updated α=10, β=inf
Reached leaf L6 → value = 4
MAX node M3_3 updated α=10, β=inf
MIN node M2_2 updated α=10, β=5
⚠️ PRUNED remaining children of M2_2 (α=10, β=5)
MAX node M1_1 updated α=10, β=inf

===== FINAL RESULT =====
Best value at root: 10
```