VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB RECORD

Artificial Intelligence (23CS5PCAIN)

Submitted by

Varsha P(1BM22CS320)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
Sep-2024 to Jan-2025

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

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by **Varsha P(1BM22CS320),** who is a 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 of the above mentioned subject and the work prescribed for the said degree.
Swathi Sridharan

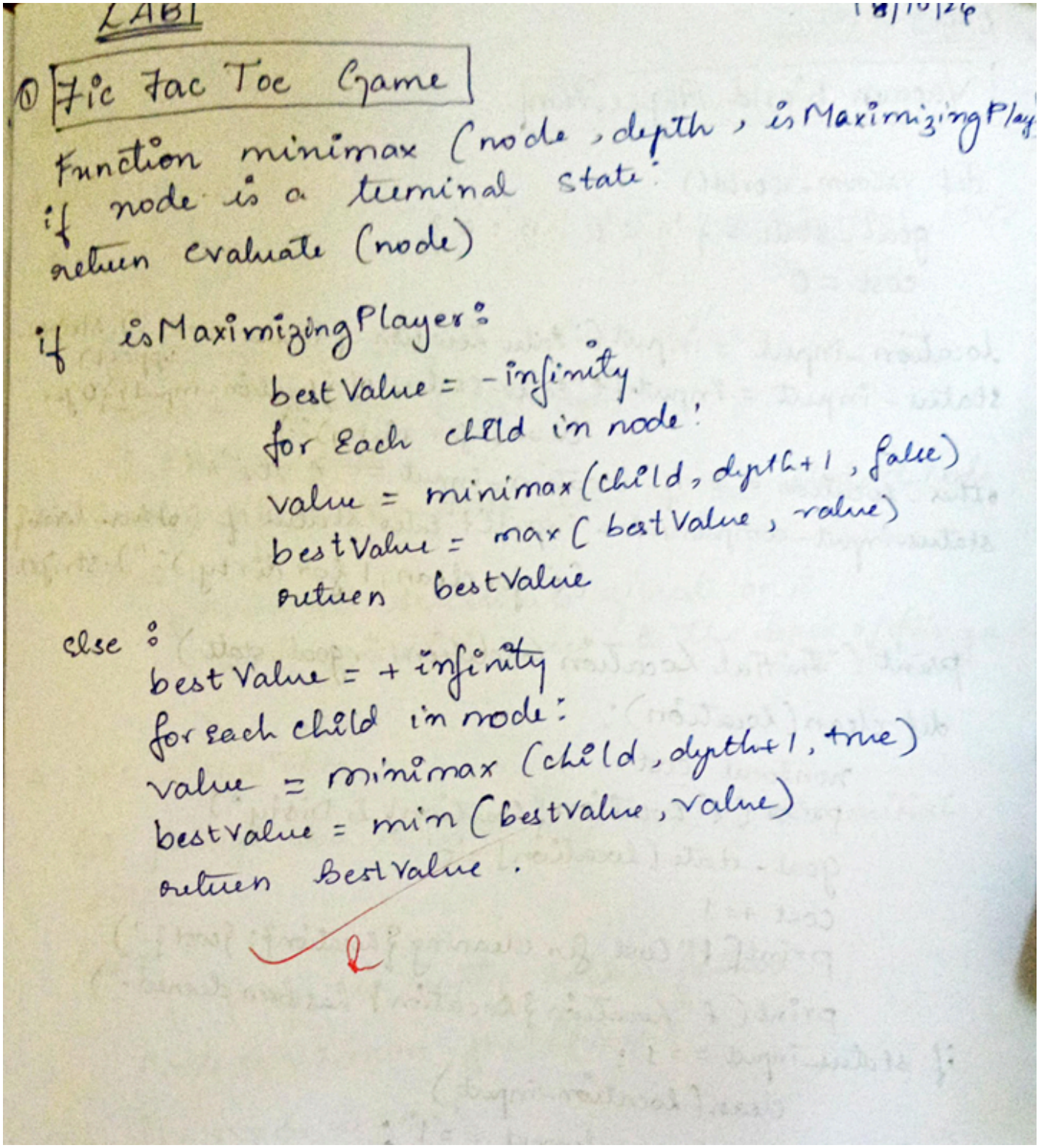
Swathi Sridharan Assistant Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor and Head Department of CSE, BMSCE

Index

Program 1

Implement Tic –Tac –Toe Game

Algorithm :



Code :

```
board={1:' ',2:' ',3:' ',
      4:' ',5:' ',6:' ',
      7:' ',8:' ',9:' '
}



def printBoard(board):
   print(board[1]+'|'+board[2]+'|'+board[3])
   print('-+-+-')
   print(board[4] + '|' + board[5] + '|' + board[6])
   print('-+-+-')
   print(board[7] + '|' + board[8] + '|' + board[9])
   print('\n')

def spaceFree(pos):
   if(board[pos]==' '):
       return True
   else:
       return False

def checkWin():
   if(board[1]==board[2] and board[1]==board[3] and board[1]!=' '):
       return True
   elif(board[4]==board[5] and board[4]==board[6] and board[4]!=' '):
       return True
   elif(board[7]==board[8] and board[7]==board[9] and board[7]!=' '):
       return True
   elif (board[1] == board[5] and board[1] == board[9] and board[1] != ' '):
       return True
   elif (board[3] == board[5] and board[3] == board[7] and board[3] != ' '):
       return True
   elif (board[1] == board[4] and board[1] == board[7] and board[1] != ' '):
       return True
   elif (board[2] == board[5] and board[2] == board[8] and board[2] != ' '):
       return True
   elif (board[3] == board[6] and board[3] == board[9] and board[3] != ' '):
       return True
   else:
       return False

def checkMoveForWin(move):
   if (board[1]==board[2] and board[1]==board[3] and board[1] ==move):
       return True
   elif (board[4]==board[5] and board[4]==board[6] and board[4] ==move):
       return True
```

```
    elif (board[7]==board[8] and board[7]==board[9] and board[7] ==move):
        return True
    elif (board[1]==board[5] and board[1]==board[9] and board[1] ==move):
        return True
    elif (board[3]==board[5] and board[3]==board[7] and board[3] ==move):
        return True
    elif (board[1]==board[4] and board[1]==board[7] and board[1] ==move):
        return True
    elif (board[2]==board[5] and board[2]==board[8] and board[2] ==move):
        return True
    elif (board[3]==board[6] and board[3]==board[9] and board[3] ==move):
        return True
    else:
        return False

def checkDraw():
    for key in board.keys():
        if (board[key]==' '):
            return False
    return True


def insertLetter(letter, position):
    if (spaceFree(position)):
        board[position] = letter
        printBoard(board)

        if (checkDraw()):
            print('Draw!')
        elif (checkWin()):
            if (letter == 'X'):
                print('Bot wins!')
            else:
                print('You win!')
        return

    else:
        print('Position taken, please pick a different position.')
        position = int(input('Enter new position: '))
        insertLetter(letter, position)
        return

player = 'O'
bot ='X'

def playerMove():
    position=int(input('Enter position for O:'))
```

```
    insertLetter(player, position)
    return

def compMove():
    bestScore=-1000
    bestMove=0
    for key in board.keys():
        if (board[key]==' '):
            board[key]=bot
            score = minimax(board, False)
            board[key] = ' '
            if (score > bestScore):
                bestScore = score
                bestMove = key

    insertLetter(bot, bestMove)
    return


def minimax(board, isMaximizing):
    if (checkMoveForWin(bot)):
        return 1
    elif (checkMoveForWin(player)):
        return -1
    elif (checkDraw()):
        return 0

    if isMaximizing:
        bestScore = -1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = bot
                score = minimax(board, False)
                board[key] = ' '
                if (score > bestScore):
                    bestScore = score
        return bestScore
    else:
        bestScore = 1000

        for key in board.keys():
            if board[key] == ' ':
                board[key] = player
                score = minimax(board, True)
                board[key] = ' '
                if (score < bestScore):
```

```
                    bestScore = score
        return bestScore

while not checkWin():
    playerMove()
    compMove()
print("Varsha P(1BM22Cs320)")
```

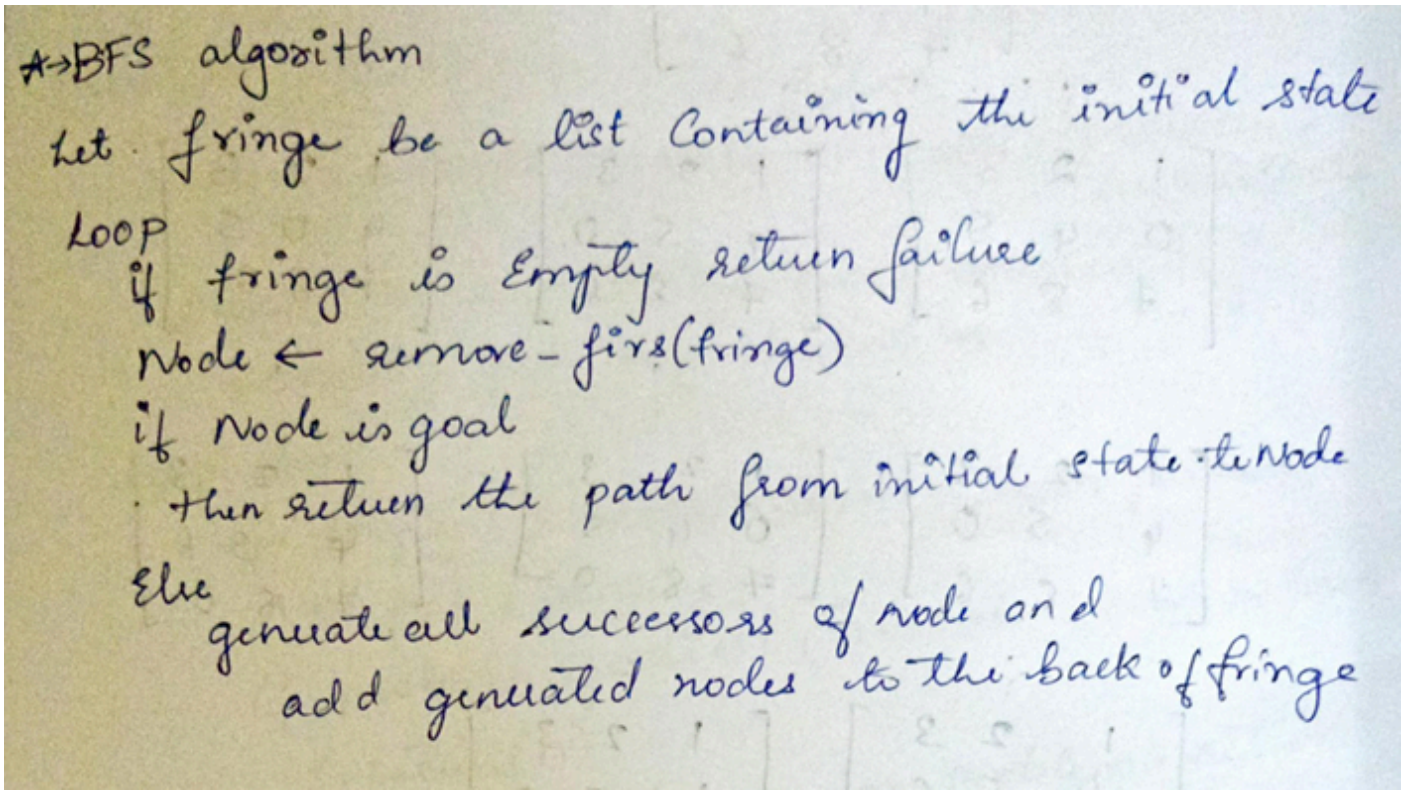
Output :

```
Enter position for O:1
O| |
-+-+-
 | |
-+-+-
 | |


O| |
-+-+-
 |X|
-+-+-
 | |


Enter position for O:2
O|O|
-+-+-
 |X|
-+-+-
 | |


O|O|X
-+-+-
 |X|
-+-+-
 | |


Enter position for O:4
O|O|X
-+-+-
O|X|
-+-+-
 | |


O|O|X
-+-+-
O|X|
-+-+-
X| |


Bot wins!
Varsha P(1BM22CS320)
```

Program 2

Solve 8 puzzle problems
1.BFS

Algorithm :



Code :

```
from collections import deque

class PuzzleState:
    def __init__(self, board, zero_position, path=[]):
        self.board = board
        self.zero_position = zero_position
        self.path = path

    def is_goal(self):
        return self.board == [1, 2, 3, 4, 5, 6, 7, 8, 0]

    def get_possible_moves(self):
        moves = []
        row, col = self.zero_position
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
                new_board = self.board[:]
                # Swap zero with the adjacent tile
                new_board[row * 3 + col], new_board[new_row * 3 + new_col] =
new_board[new_row * 3 + new_col], new_board[row * 3 + col]
                moves.append(PuzzleState(new_board, (new_row, new_col), self.path +
[new_board]))

        return moves

def bfs(initial_state):
    queue = deque([initial_state])
    visited = set()

    while queue:
        current_state = queue.popleft()

        # Show the current board
        print("Current Board State:")
        print_board(current_state.board)
        print()

        if current_state.is_goal():
            return current_state.path

        visited.add(tuple(current_state.board))
```

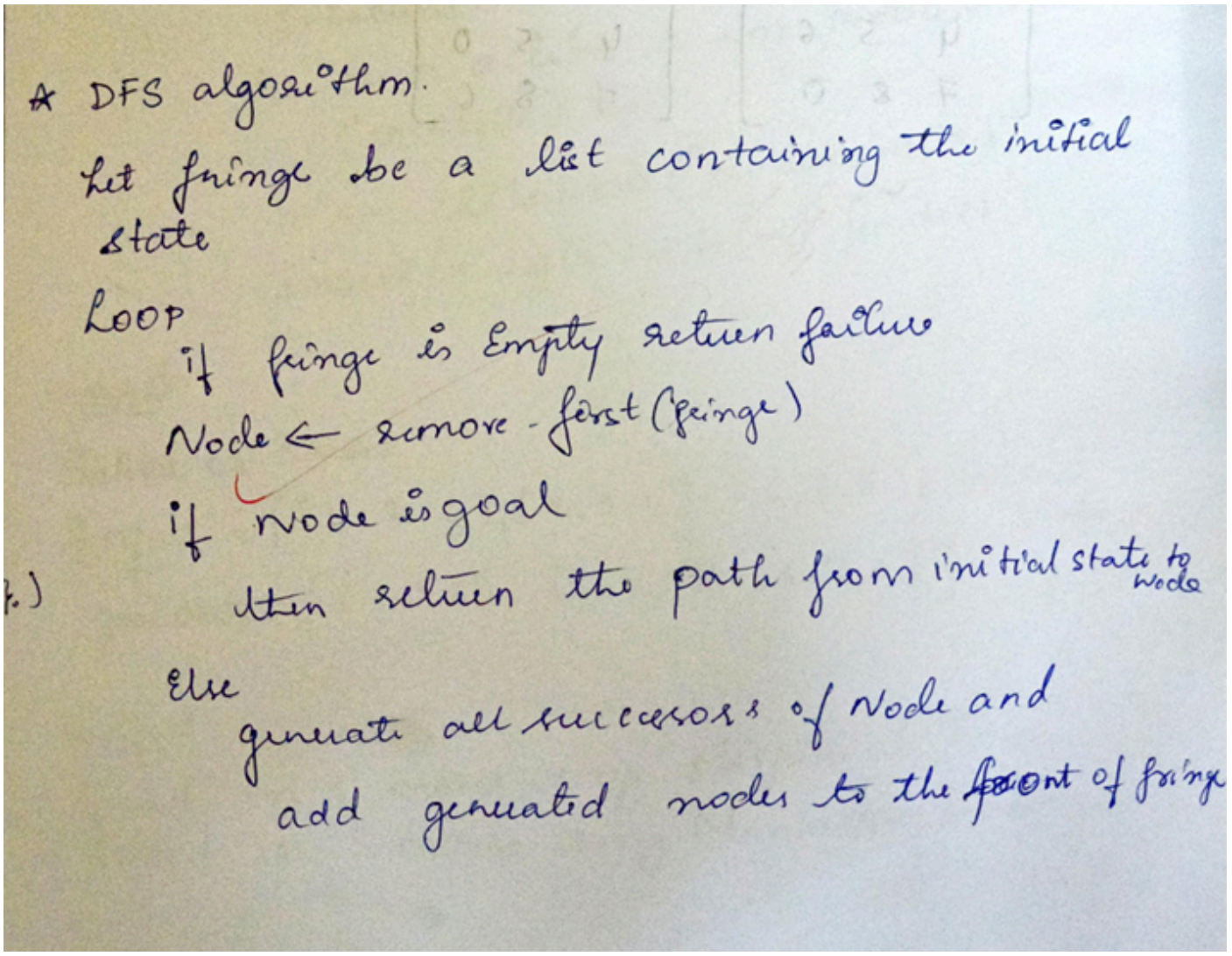
```
        for next_state in current_state.get_possible_moves():
            if tuple(next_state.board) not in visited:
                queue.append(next_state)

    return None

def print_board(board):
    for i in range(3):
        print(board[i * 3:i * 3 + 3])

def main():
    print("Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g.,
'1 2 3 4 5 6 7 8 0'): ")
    user_input = input()
    initial_board = list(map(int, user_input.split()))

    if len(initial_board) != 9 or set(initial_board) != set(range(9)):
        print("Invalid input! Please enter 9 numbers from 0 to 8.")
        return

    zero_position = initial_board.index(0)
    initial_state = PuzzleState(initial_board, (zero_position // 3, zero_position %
3))

    solution_path = bfs(initial_state)

    if solution_path is None:
        print("No solution found.")
    else:
        print("Solution found in", len(solution_path), "steps.")
        for step in solution_path:
            print_board(step)
            print()

if __name__ == "__main__":
    main()

print("-------------")
print("Varsha P(1BM22CS320)")
```

Output :

```
Enter the initial state of the 8-puzzle (use 0 for the blank tile, e.g., '1 2 3 4 5 6 7 8 0'):
1 2 3 4 0 6 7 5 8
Current Board State:
[1, 2, 3]
[4, 0, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 0]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Current Board State:
[1, 2, 3]
[0, 4, 6]
[7, 5, 8]

Current Board State:
[1, 0, 3]
[4, 2, 6]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 6, 8]
[7, 5, 0]


Current Board State:
[1, 2, 0]
[4, 6, 3]
[7, 5, 8]

Current Board State:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]

Solution found in 2 steps.
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

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

-------------
Varsha P(1BM22CS320)
```

2.DFS

Algorithm :



* DFS algorithm.

Let fringe be a list containing the initial state

LOOP

if fringe is Empty return failure

Node ← remove-first(fringe)

if Node is goal

then return the path from initial state to Node

Else

generate all successors of Node and

add generated nodes to the front of fringe

Code:

```
from collections import deque
print("VARSHA P(1BM22CS320)")
print("--------")

def get_user_input(prompt):
    board = []
    print(prompt)
    for i in range(3):
        row = list(map(int, input(f"Enter row {i+1} (space-separated numbers, use 0
for empty space): ").split()))
        board.append(row)
    return board

def is_solvable(board):
    flattened_board = [tile for row in board for tile in row if tile != 0]
    inversions = 0
    for i in range(len(flattened_board)):
        for j in range(i + 1, len(flattened_board)):
            if flattened_board[i] > flattened_board[j]:
                inversions += 1
    return inversions % 2 == 0

class PuzzleState:
    def __init__(self, board, moves=0, previous=None):
        self.board = board
        self.empty_tile = self.find_empty_tile()
        self.moves = moves
        self.previous = previous

    def find_empty_tile(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return (i, j)

    def is_goal(self, goal_state):
        return self.board == goal_state

    def get_possible_moves(self):
        row, col = self.empty_tile
        possible_moves = []
        directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]  # down, up, right, left

        for dr, dc in directions:
            new_row, new_col = row + dr, col + dc
            if 0 <= new_row < 3 and 0 <= new_col < 3:
```

```
                # Make the move
                new_board = [row[:] for row in self.board]  # Deep copy
                new_board[row][col], new_board[new_row][new_col] =
new_board[new_row][new_col], new_board[row][col]
                possible_moves.append(PuzzleState(new_board, self.moves + 1, self))

        return possible_moves

def dfs(initial_state, goal_state):
    stack = [initial_state]
    visited = set()

    while stack:
        current_state = stack.pop()

        if current_state.is_goal(goal_state):
            return current_state

        # Convert board to a tuple for the visited set
        state_tuple = tuple(tuple(row) for row in current_state.board)

        if state_tuple not in visited:
            visited.add(state_tuple)
            for next_state in current_state.get_possible_moves():
                stack.append(next_state)

    return None  # No solution found

def print_solution(solution):
    path = []
    while solution:
        path.append(solution.board)
        solution = solution.previous
    for state in reversed(path):
        for row in state:
            print(row)
        print()

if __name__ == "__main__":
    # Get user input for initial and goal states
    initial_board = get_user_input("Enter the initial state of the puzzle:")
    goal_board = get_user_input("Enter the goal state of the puzzle:")

    if is_solvable(initial_board):
        initial_state = PuzzleState(initial_board)
        solution = dfs(initial_state, goal_board)
```

```
        if solution:
            print("Solution found in", solution.moves, "moves:")
            print_solution(solution)
        else:
            print("No solution found.")
    else:
        print("This puzzle is unsolvable.")

print("VARSHA P(1BM22CS320)")
```

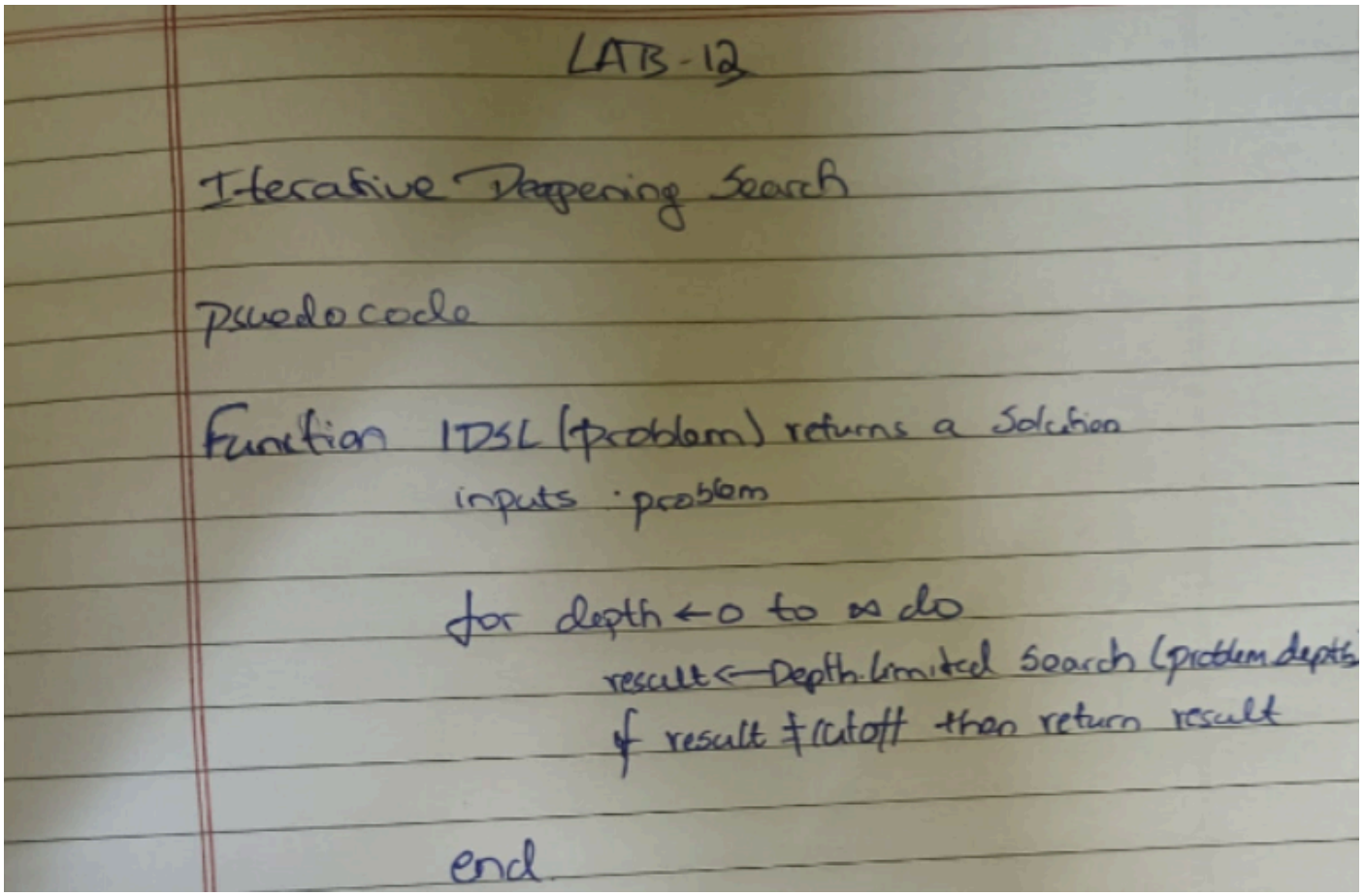
Output

```
VARSHA P(1BM22CS320)
--------
Enter the initial state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 0 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 4 7 8
Enter the goal state of the puzzle:
Enter row 1 (space-separated numbers, use 0 for empty space): 1 2 3
Enter row 2 (space-separated numbers, use 0 for empty space): 4 5 6
Enter row 3 (space-separated numbers, use 0 for empty space): 7 8 0
Solution found in 431 moves:
[1, 2, 3]
[0, 5, 6]
[4, 7, 8]
```

Program 3

Implement Iterative deepening search algorithm.

Algorithm :



Code :

```
from copy import deepcopy

DIRECTIONS = [(-1, 0), (1, 0), (0, -1), (0, 1)]

class PuzzleState:
    def __init__(self, board, parent=None, move=""):
        self.board = board
        self.parent = parent
        self.move = move

    def get_blank_position(self):
        for i in range(3):
            for j in range(3):
                if self.board[i][j] == 0:
                    return i, j

    def generate_successors(self):
        successors = []
        x, y = self.get_blank_position()

        for dx, dy in DIRECTIONS:
            new_x, new_y = x + dx, y + dy

            if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_board = deepcopy(self.board)
                new_board[x][y], new_board[new_x][new_y] = new_board[new_x][new_y],
new_board[x][y]
                successors.append(PuzzleState(new_board, parent=self))

        return successors

    def is_goal(self, goal_state):
        return self.board == goal_state

    def __str__(self):
        return "\n".join([" ".join(map(str, row)) for row in self.board])

def depth_limited_search(current_state, goal_state, depth):
    if depth == 0 and current_state.is_goal(goal_state):
        return current_state

    if depth > 0:
        for successor in current_state.generate_successors():
            found = depth_limited_search(successor, goal_state, depth - 1)
            if found:
```

```
                return found
    return None

def iterative_deepening_search(start_state, goal_state, max_depth):
    for depth in range(max_depth + 1):
        print(f"\nSearching at depth level: {depth}")
        result = depth_limited_search(start_state, goal_state, depth)
        if result:
            return result
    return None

def get_user_input():
    print("Enter the start state (use 0 for the blank):")
    start_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        start_state.append(row)

    print("Enter the goal state (use 0 for the blank):")
    goal_state = []
    for _ in range(3):
        row = list(map(int, input().split()))
        goal_state.append(row)

    max_depth = int(input("Enter the maximum depth for search: "))

    return start_state, goal_state, max_depth

def main():
    start_board, goal_board, max_depth = get_user_input()
    start_state = PuzzleState(start_board)
    goal_state = goal_board

    result = iterative_deepening_search(start_state, goal_state, max_depth)

    if result:
        print("\nGoal reached!")
        path = []
        while result:
            path.append(result)
            result = result.parent
        path.reverse()
        for state in path:
            print(state, "\n")
    else:
        print("Goal state not found within the specified depth.")
```

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

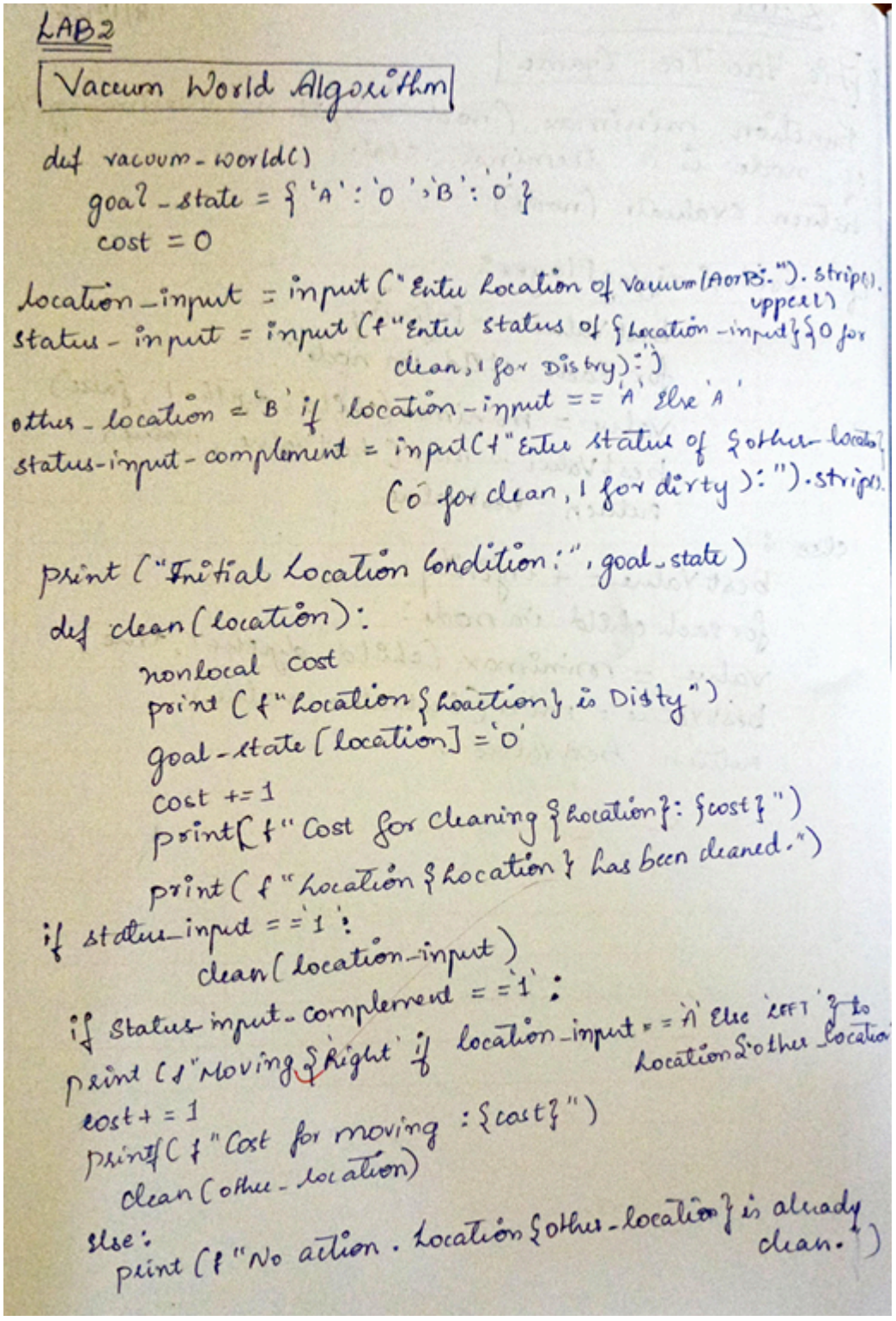
Output :

```
Enter the start state (use 0 for the blank):
2 8 3
1 6 4
7 0 5
Enter the goal state (use 0 for the blank):
1 2 3
8 0 4
7 6 5
Enter the maximum depth for search: 5

Searching at depth level: 0

Searching at depth level: 1

Searching at depth level: 2

Searching at depth level: 3

Searching at depth level: 4

Searching at depth level: 5

Goal reached!
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

-----------
Varsha P(1BM22CS320)
```

Program 4

Implement a vacuum cleaner agent.

Algorithm :



LAB2

Vacuum World Algorithm

```
def vacuum_world()
    goal_state = {'A': '0', 'B': '0'}
    cost = 0

location_input = input("Enter location of Vacuum (A or B: ").strip().upper()
status_input = input(f"Enter status of {location_input} (0 for clean, 1 for Dirty): ")

other_location = 'B' if location_input == 'A' else 'A'
status_input_complement = input(f"Enter status of {other_location} (0 for clean, 1 for dirty): ").strip()

print("Initial Location Condition:", goal_state)

def clean(location):
        nonlocal cost
        print(f"Location {location} is Dirty")
        goal_state[location] = '0'
        cost += 1
        print(f"Cost for cleaning {location}: {cost}")
        print(f"Location {location} has been cleaned.")

if status_input == '1':
        clean(location_input)

if Status_input_complement == '1':
print(f"Moving 'Right' if location_input == 'A' else 'LEFT' to Location {other_location}")
    cost += 1
    print(f"Cost for moving : {cost}")
    clean(other_location)
else:
    print(f"No action. Location {other_location} is already clean.")
```

Code :

```
def vacuum_world():
    # Initializing goal_state
    # 0 indicates Clean and 1 indicates Dirty
    goal_state = {'A': '0', 'B': '0'}
    cost = 0



    location_input = input("Enter Location of Vacuum (A or B): ").strip().upper()
# User input of location
    status_input = input(f"Enter status of {location_input} (0 for Clean, 1 for
Dirty): ").strip()
    status_input_complement = input("Enter status of other room (0 for Clean, 1 for
Dirty): ").strip()

    print("Initial Location Condition: " + str(goal_state))

    if location_input == 'A':
        print("Vacuum is placed in Location A")
        if status_input == '1':
            print("Location A is Dirty.")
            goal_state['A'] = '0'  # Clean A
            cost += 1  # Cost for sucking
            print("Cost for CLEANING A: " + str(cost))
            print("Location A has been Cleaned.")

            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving right to Location B.")
                cost += 1  # Cost for moving right
                print("Cost for moving RIGHT: " + str(cost))

                goal_state['B'] = '0'  # Clean B
                cost += 1  # Cost for sucking
                print("Cost for SUCK: " + str(cost))
                print("Location B has been Cleaned.")
            else:
                print("Location B is already clean.")
        else:
            print("Location A is already clean.")
            if status_input_complement == '1':
                print("Location B is Dirty.")
                print("Moving RIGHT to Location B.")
                cost += 1  # Cost for moving right
                print("Cost for moving RIGHT: " + str(cost))

                goal_state['B'] = '0'  # Clean B
```

```
            cost += 1  # Cost for sucking
            print("Cost for SUCK: " + str(cost))
            print("Location B has been Cleaned.")
        else:
            print("Location B is already clean.")

elif location_input == 'B':
    print("Vacuum is placed in Location B")
    if status_input == '1':
        print("Location B is Dirty.")
        goal_state['B'] = '0'  # Clean B
        cost += 1  # Cost for sucking
        print("Cost for CLEANING B: " + str(cost))
        print("Location B has been Cleaned.")

        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1  # Cost for moving left
            print("Cost for moving LEFT: " + str(cost))

            goal_state['A'] = '0'  # Clean A
            cost += 1  # Cost for sucking
            print("Cost for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")
    else:
        print("Location B is already clean.")
        if status_input_complement == '1':
            print("Location A is Dirty.")
            print("Moving LEFT to Location A.")
            cost += 1  # Cost for moving left
            print("Cost for moving LEFT: " + str(cost))

            goal_state['A'] = '0'  # Clean A
            cost += 1  # Cost for sucking
            print("Cost for SUCK: " + str(cost))
            print("Location A has been Cleaned.")
        else:
            print("Location A is already clean.")

# Done cleaning
print("GOAL STATE: ")
print(goal_state)
print("Performance Measurement: " + str(cost))
```

```
# Output
vacuum_world()
print("-----------")
print("Varsha P(1BM22CS320)")
```

Output :

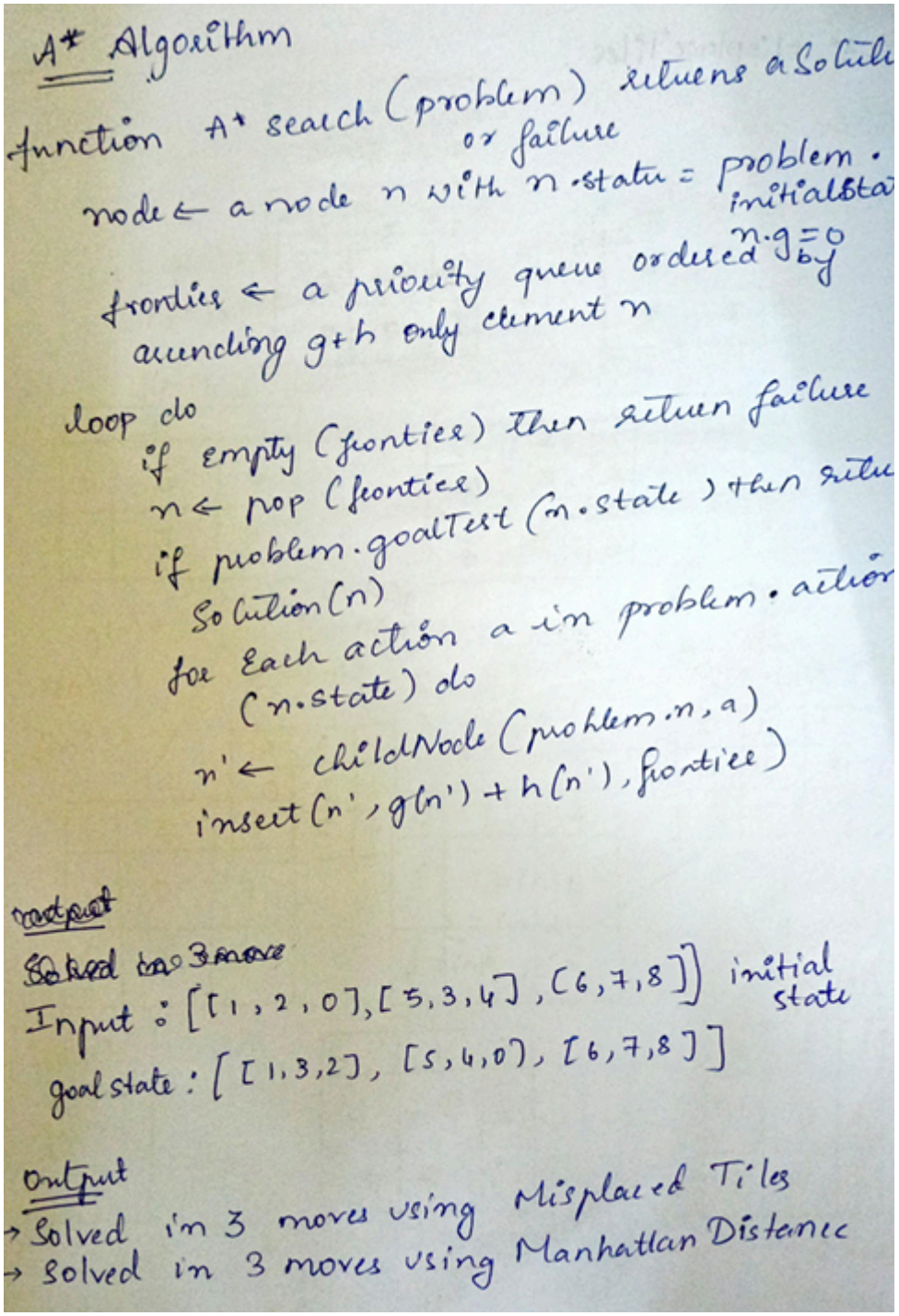
```
Enter Location of Vacuum (A or B): A
Enter status of A (0 for Clean, 1 for Dirty): 1
Enter status of other room (0 for Clean, 1 for Dirty): 1
Initial Location Condition: {'A': '0', 'B': '0'}
Vacuum is placed in Location A
Location A is Dirty.
Cost for CLEANING A: 1
Location A has been Cleaned.
Location B is Dirty.
Moving right to Location B.
Cost for moving RIGHT: 2
Cost for SUCK: 3
Location B has been Cleaned.
GOAL STATE:
{'A': '0', 'B': '0'}
Performance Measurement: 3
-----------
Varsha P(1BM22CS320)
```

Program 5

Implement A* search algorithm.

1.Using Misplaced tiles

Algorithm :



Code :

```
import heapq
print("Varsha P(1BM22CS320)")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
    [2, 8, 1],
    [0, 4, 3],
    [7, 6, 5]
]

# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0),  # Up
    (1, 0),   # Down
    (0, -1),  # Left
    (0, 1)    # Right
]

class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g  # Cost from start to current node
        self.h = h  # Heuristic cost to goal
        self.f = g + h  # Total cost

    def __lt__(self, other):
        return self.f < other.f

def misplaced_tiles(state):
    """Heuristic function that counts the number of misplaced tiles."""
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != GOAL_STATE[i][j]:
                misplaced += 1
    return misplaced

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None
```

```
def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)

    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[zero_x][zero_y]
            h = misplaced_tiles(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1,
h=h))
    return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=misplaced_tiles(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state):
            return reconstruct_path(current_node)

        closed_set.add(tuple(map(tuple, current_node.state)))

        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
```

```
                continue
            heapq.heappush(open_list, successor)

    return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8):
").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try
again.")
        return None

    return state

# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()

    solution = a_star(start_state)

    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
        for step in solution:
            for row in step:
```

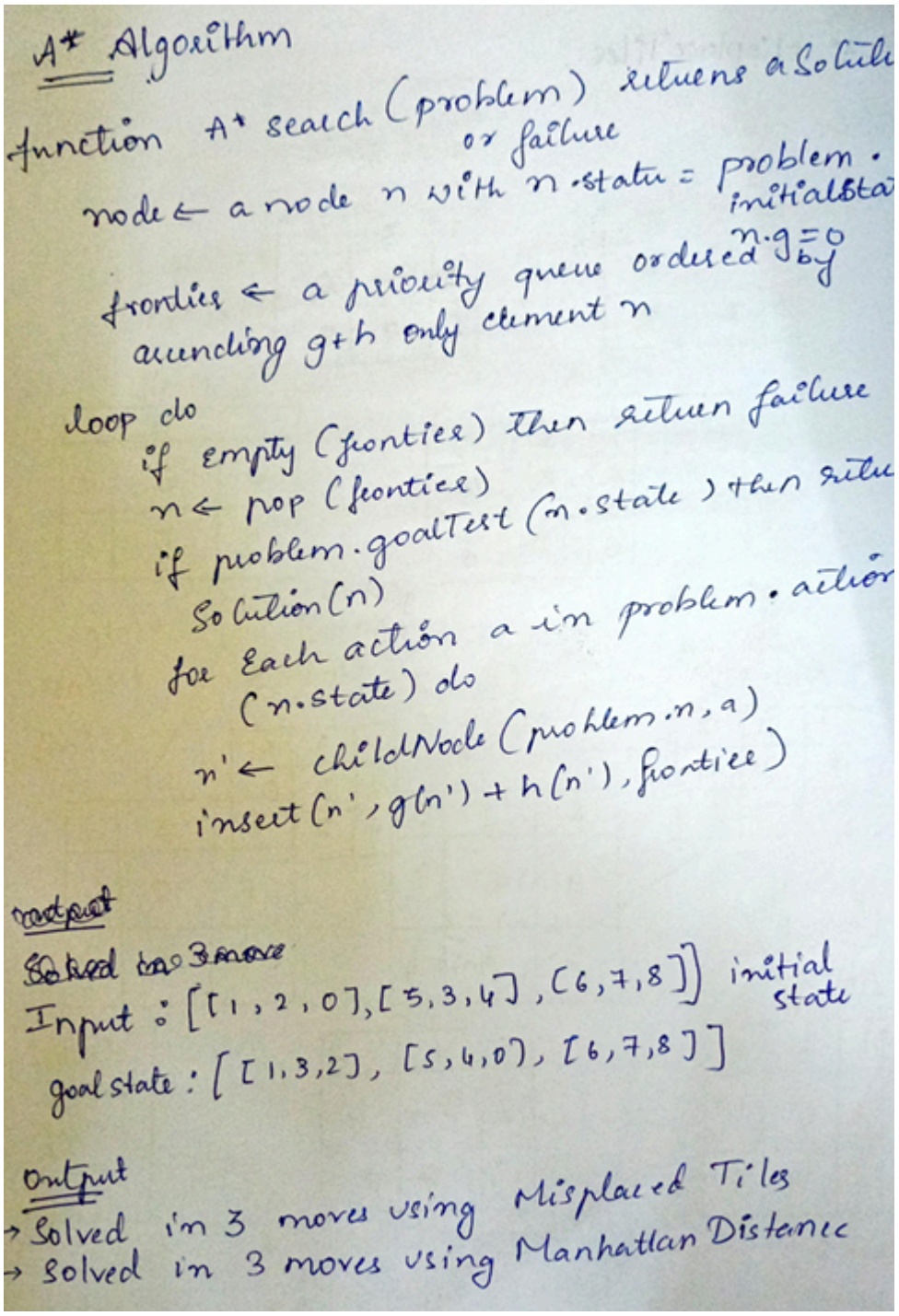
```
                print(row)
            print()
    else:
        print("No solution found.")

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

Output

```
Varsha P(1BM22CS320)
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 4 0 6
Enter row 3 (space-separated numbers between 0 and 8): 7 8 5
Solution found in 17 moves:
[1, 2, 3]
[4, 0, 6]
[7, 8, 5]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

2.Using Manhattan distance

Algorithm:



Code:

```
import heapq
print("Varsha P(1BM22CS320)")

# Define the goal state for the 8-puzzle
GOAL_STATE = [
     [2, 8, 1],
     [0, 4, 3],
     [7, 6, 5]

]

# Define the position moves (up, down, left, right)
MOVES = [
    (-1, 0),  # Up
    (1, 0),   # Down
    (0, -1),  # Left
    (0, 1)    # Right
]

class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g  # Cost from start to current node
        self.h = h  # Heuristic cost to goal (Manhattan distance)
        self.f = g + h  # Total cost

    def __lt__(self, other):
        return self.f < other.f

def manhattan_distance(state):
    """Heuristic function that calculates the Manhattan distance for each tile."""
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:  # Skip the empty tile
                # Calculate goal position for this value
                goal_x, goal_y = (value - 1) // 3, (value - 1) % 3
                # Add the Manhattan distance for this tile
                distance += abs(i - goal_x) + abs(j - goal_y)
    return distance

def get_zero_position(state):
    """Find the position of the zero (empty tile) in the puzzle."""
```

```
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
    return None

def generate_successors(node):
    """Generate successors by moving the empty tile in all possible directions."""
    successors = []
    zero_x, zero_y = get_zero_position(node.state)

    for move_x, move_y in MOVES:
        new_x, new_y = zero_x + move_x, zero_y + move_y
        if 0 <= new_x < 3 and 0 <= new_y < 3:
            new_state = [row[:] for row in node.state]
            new_state[zero_x][zero_y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[zero_x][zero_y]
            h = manhattan_distance(new_state)
            successors.append(PuzzleNode(new_state, parent=node, g=node.g + 1,
h=h))
    return successors

def is_goal(state):
    """Check if the current state is the goal state."""
    return state == GOAL_STATE

def reconstruct_path(node):
    """Reconstruct the path from the start state to the goal state."""
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return path[::-1]

def a_star(start_state):
    """A* algorithm to solve the 8-puzzle problem."""
    start_node = PuzzleNode(start_state, g=0, h=manhattan_distance(start_state))
    open_list = []
    closed_set = set()

    heapq.heappush(open_list, start_node)

    while open_list:
        current_node = heapq.heappop(open_list)

        if is_goal(current_node.state):
            return reconstruct_path(current_node)
```

```
        closed_set.add(tuple(map(tuple, current_node.state)))

        for successor in generate_successors(current_node):
            if tuple(map(tuple, successor.state)) in closed_set:
                continue
            heapq.heappush(open_list, successor)

    return None

def get_user_input():
    """Get a valid 8-puzzle input state from the user."""
    print("Enter your 8-puzzle configuration (0 represents the empty tile):")
    state = []
    values = set()

    for i in range(3):
        row = input(f"Enter row {i+1} (space-separated numbers between 0 and 8):
").split()
        if len(row) != 3:
            print("Each row must have exactly 3 numbers. Please try again.")
            return None

        row = [int(x) for x in row]

        if not all(0 <= x <= 8 for x in row):
            print("Values must be between 0 and 8. Please try again.")
            return None

        state.append(row)
        values.update(row)

    if values != set(range(9)):
        print("All numbers from 0 to 8 must be present exactly once. Please try
again.")
        return None

    return state
# Main function
def main():
    start_state = None
    while start_state is None:
        start_state = get_user_input()
    solution = a_star(start_state)
    # Print the solution steps
    if solution:
        print("Solution found in", len(solution) - 1, "moves:")
```

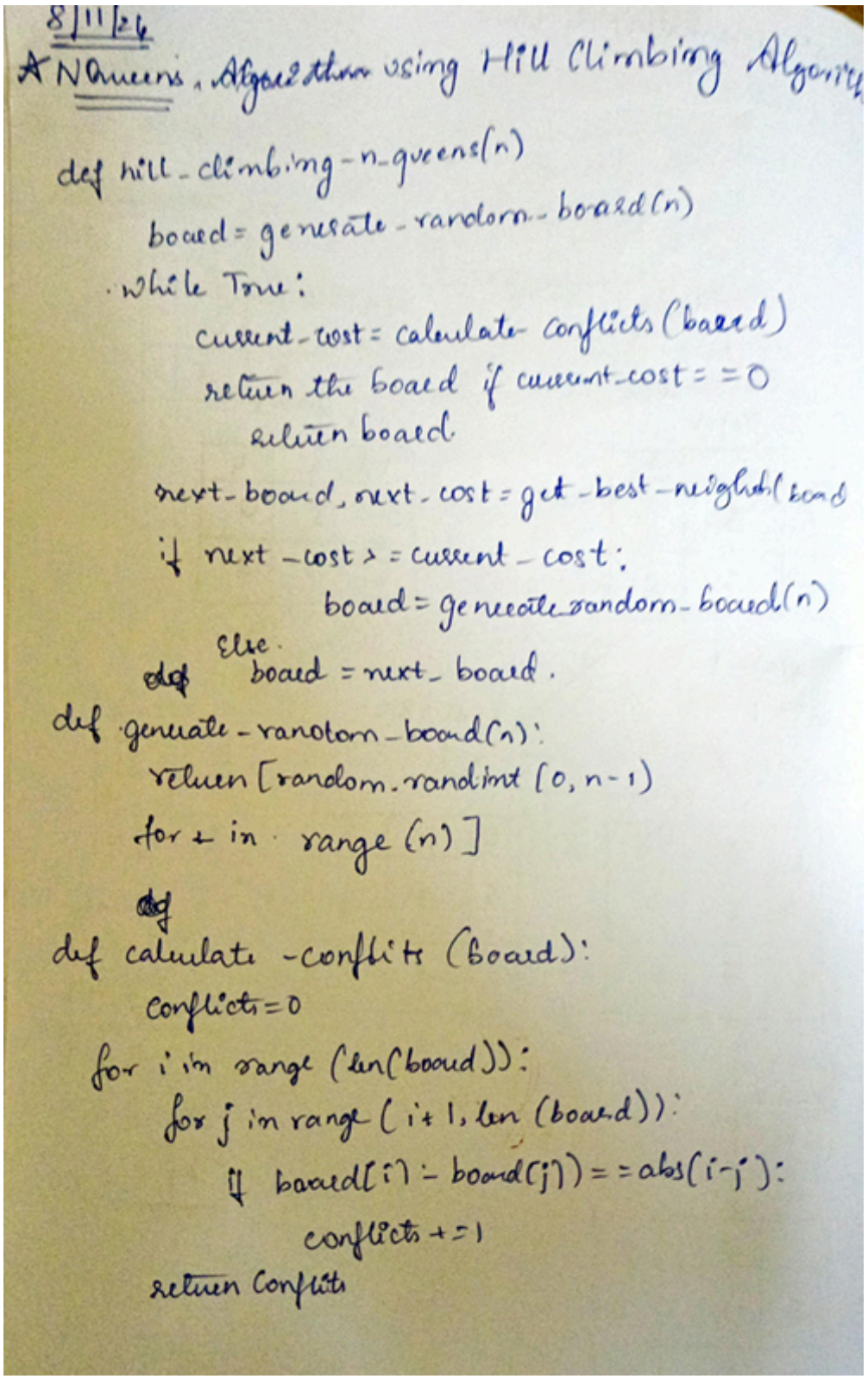
```
        for step in solution:
            for row in step:
                print(row)
            print()
    else:
        print("No solution found.")

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

Output

```
Varsha P(1BM22CS320)
Enter your 8-puzzle configuration (0 represents the empty tile):
Enter row 1 (space-separated numbers between 0 and 8): 1 2 3
Enter row 2 (space-separated numbers between 0 and 8): 4 0 6
Enter row 3 (space-separated numbers between 0 and 8): 7 8 5
Solution found in 17 moves:
[1, 2, 3]
[4, 0, 6]
[7, 8, 5]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Implement Hill Climbing Algorithm.

Algorithm :



8/11/24

A NQueens Algorithm using Hill Climbing Algorithm

```
def hill_climbing_n_queens(n)
    board = generate_random_board(n)
    while True:
        current_cost = calculate_conflicts(board)
        return the board if current_cost == 0
            return board
        next_board, next_cost = get_best_neighbor(board)
        if next_cost >= current_cost:
                board = generate_random_board(n)
        Else:
            board = next_board.
def generate_random_board(n):
    return [random.randint(0, n-1)
    for _ in range(n)]

def calculate_conflicts(board):
    conflicts = 0
    for i in range(len(board)):
        for j in range(i+1, len(board)):
            if board[i] == board[j] == abs(i-j):
                conflicts += 1
    return conflicts
```

Code :

```
import random

def hill_climbing_n_queens(n):
    # Step 1: Initialize a random board with one queen in each column
    board = generate_random_board(n)

    while True:
        # Calculate the current number of conflicts
        current_cost = calculate_conflicts(board)

        # If solution is found (no conflicts), return the board
        if current_cost == 0:
            return board

        # Step 3: Find the neighbor with the lowest number of conflicts
        next_board, next_cost = get_best_neighbor(board)

        # Step 4: Check if we've reached a local minimum
        if next_cost >= current_cost:
            # If stuck in local minimum, restart with a new board
            board = generate_random_board(n)
        else:
            # Move to the better board configuration
            board = next_board

def generate_random_board(n):
    # Generates a random board with one queen in each column
    return [random.randint(0, n - 1) for _ in range(n)]

def calculate_conflicts(board):
    # Counts the number of pairs of queens that are in conflict
    conflicts = 0
    for i in range(len(board)):
        for j in range(i + 1, len(board)):
            if board[i] == board[j] or abs(board[i] - board[j]) == abs(i - j):
                conflicts += 1
    return conflicts

def get_best_neighbor(board):
    n = len(board)
    best_board = board[:]
    best_cost = calculate_conflicts(board)

    # Try moving each queen in each column to a different row
    for col in range(n):
```

```
        original_row = board[col]

        # Check each possible row for this column
        for row in range(n):
            if row != original_row:
                board[col] = row
                cost = calculate_conflicts(board)

                # Keep track of the best board with minimum conflicts
                if cost < best_cost:
                    best_cost = cost
                    best_board = board[:]

        # Revert the queen to its original row
        board[col] = original_row

    return best_board, best_cost

# Example usage:
n=int(input("No of queens: "))  # Change n to desired board size
solution = hill_climbing_n_queens(n)
print("Solution for", n, "queens:")
print(solution)


print("---------")
print("VARSHA P(1BM22CS320)")
```

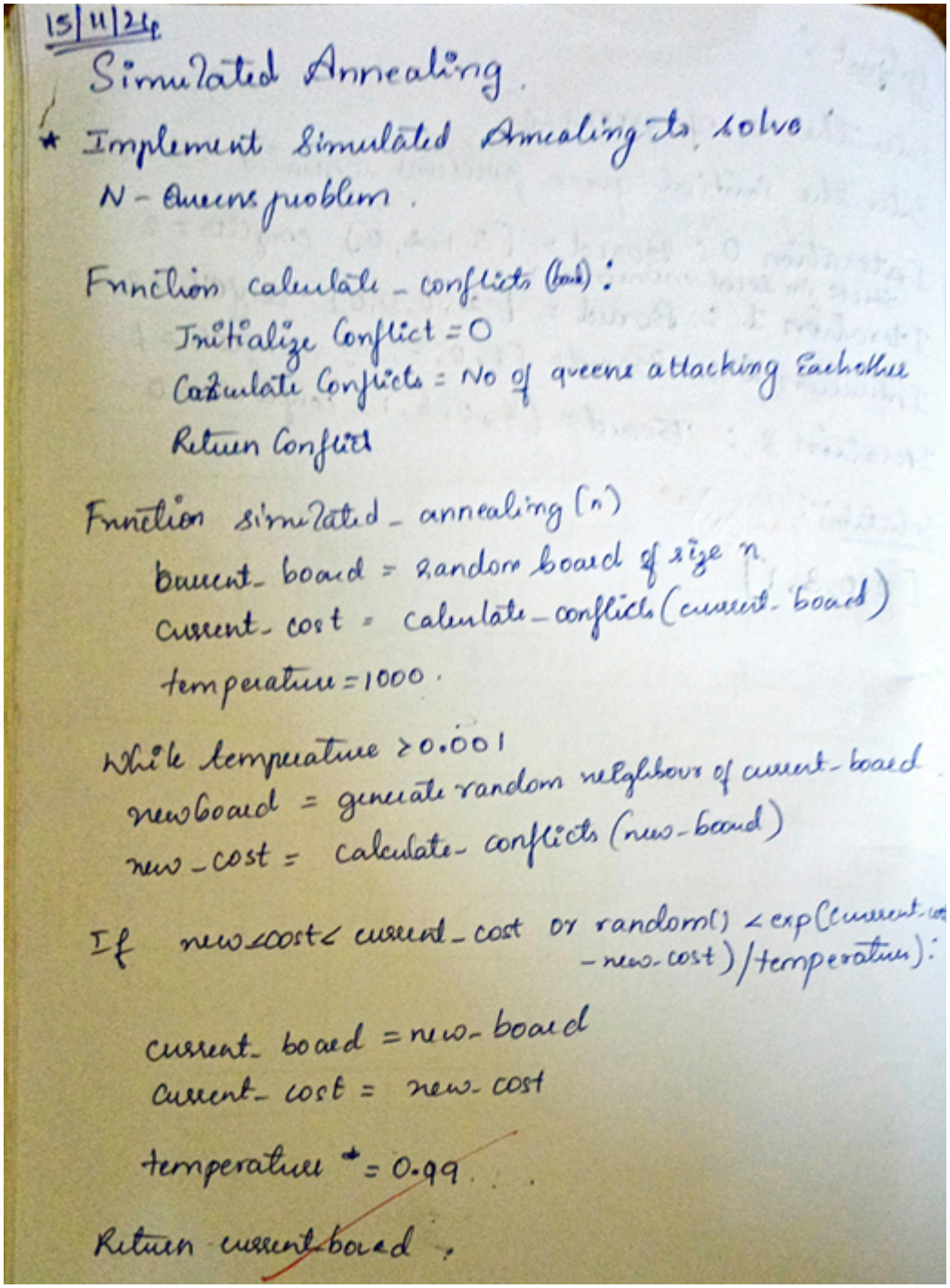
Output :

```
No of queens: 8
Solution for 8 queens:
[2, 5, 7, 0, 4, 6, 1, 3]
---------
VARSHA P(1BM22CS320)


No of queens: 4
Solution for 4 queens:
[2, 0, 3, 1]
---------
VARSHA P(1BM22CS320)
```

Program 6

Write a program to implement Simulated Annealing Algorithm

Algorithm :



15/11/24

Simulated Annealing.

* Implement Simulated Annealing to solve N-Queens problem.

Function calculate_conflicts (board):
    Initialize Conflict = 0
    Calculate Conflicts = No of queens attacking each other
    Return Conflict

Function simulated_annealing (n)
    current_board = random board of size n
    current_cost = calculate_conflicts (current_board)
    temperature = 1000.

While temperature > 0.001
    newboard = generate random neighbour of current_board
    new_cost = calculate_conflicts (new_board)

If new_cost < current_cost or random() < exp((current_cost - new_cost)/temperature):

    current_board = new_board
    current_cost = new_cost

    temperature *= 0.99

Return current_board.

Code :

```
import random
import math


def calculate_conflicts(board):
    conflicts = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if two queens are attacking each other
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                conflicts += 1
    return conflicts

# Function to simulate the annealing process
def simulated_annealing(board, max_iterations=10000, initial_temperature=1000,
cooling_rate=0.99):
    current_board = board[:]
    current_cost = calculate_conflicts(current_board)
    temperature = initial_temperature

    for iteration in range(max_iterations):
        if current_cost == 0:
            return current_board, iteration


        new_board = current_board[:]
        queen_index = random.randint(0, len(board) - 1)
        new_board[queen_index] = random.randint(0, len(board) - 1)

        new_cost = calculate_conflicts(new_board)


        if new_cost < current_cost or random.random() < math.exp((current_cost -
new_cost) / temperature):
            current_board = new_board
            current_cost = new_cost


        print(f"Iteration {iteration}: Cost = {current_cost}, Temperature =
{temperature:.2f}")
        print_board(current_board)
        print("\n")


        temperature *= cooling_rate
```

```
        if temperature < 1e-3:
            break

    return current_board, iteration


def print_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if board[j] == i else '.' for j in range(n)]
        print(" ".join(row))


def main():
    n = int(input("Enter the number of queens: "))

    print("Enter the initial positions of the queens as a list of row indices (0-indexed):")
    board = list(map(int, input().split()))

    if len(board) != n:
        print("Error: The number of positions provided does not match the number of queens.")
        return

    solution, iterations = simulated_annealing(board)

    print(f"\nSolution found in {iterations} iterations:")
    print_board(solution)

main()

print("Varsha P(1BMS22CS320)")
```

Output:

```
Enter the number of queens: 4
Enter the initial positions of the queens as a list of row indices (0-indexed):
3 1 2 0
Iteration 0: Cost = 4, Temperature = 1000.00
. . . Q
. . . .
. Q Q .
Q . . .


Iteration 1: Cost = 4, Temperature = 990.00
. . . .
. . . .
. Q Q Q
Q . . .
```

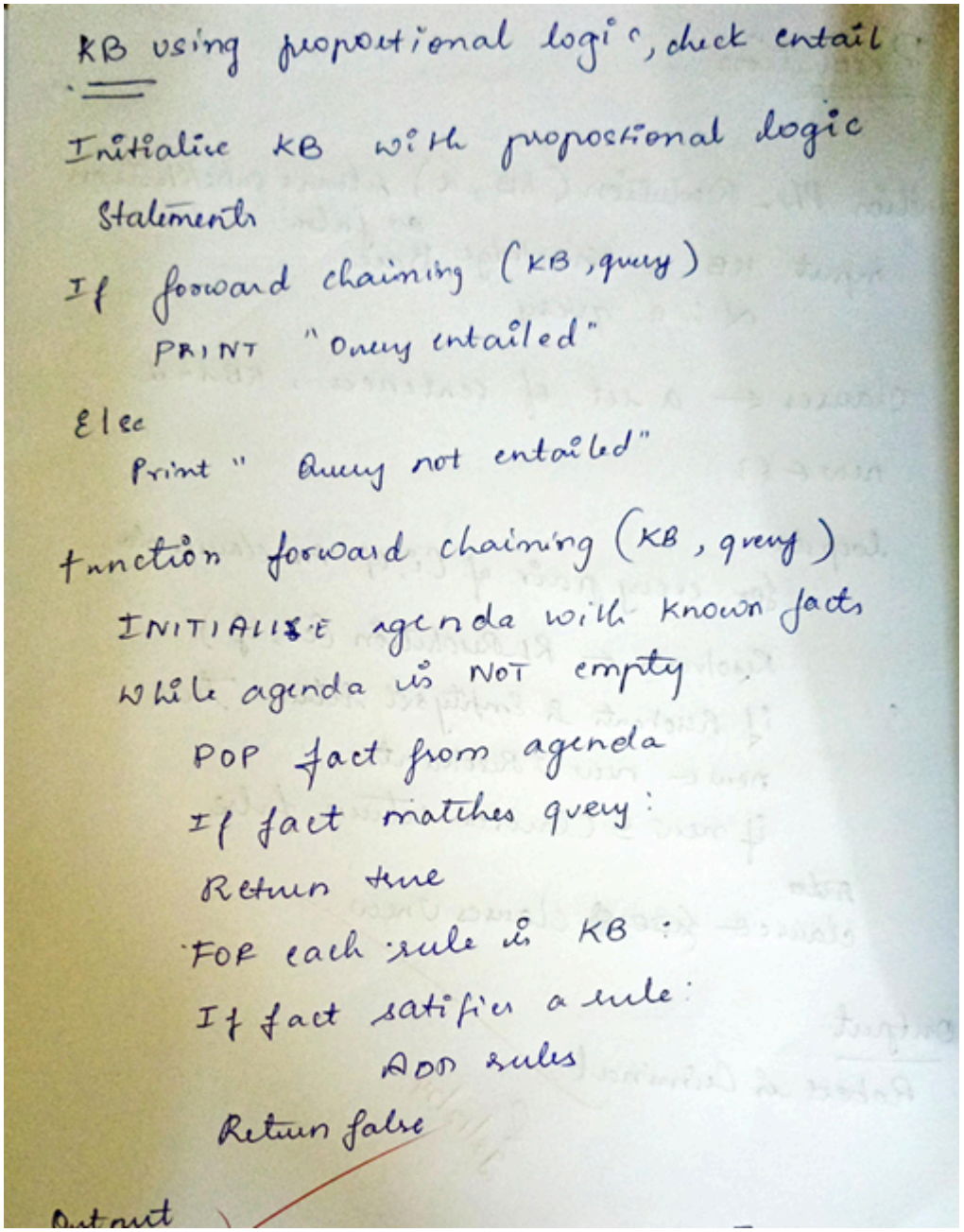
Last 5 Iterations:

```
Iteration 152: Cost = 4, Temperature = 217.04
Q . . .
. Q Q .
. . . .
. . . Q


Iteration 153: Cost = 6, Temperature = 214.87
Q . . .
. Q . .
. . Q .
. . . Q


Iteration 154: Cost = 4, Temperature = 212.73
Q . . .
. Q . .
. . . .
. . Q Q


Iteration 155: Cost = 3, Temperature = 210.60
. . . .
. Q . .
Q . . .
. . Q Q


Iteration 156: Cost = 1, Temperature = 208.49
. Q . .
. . . .
Q . . .
. . Q Q


Iteration 157: Cost = 0, Temperature = 206.41
. Q . .
. . . Q
Q . . .
. . Q .



Solution found in 158 iterations:
. Q . .
. . . Q
Q . . .
. . Q .
Varsha P(1BMS22CS320)
```

Program 7

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

Algorithm :



KB using propositional logic, check entail

Initialize KB with propositional logic statements

If forward chaining (KB, query)
    PRINT "Query entailed"

Else
    Print "Query not entailed"

function forward chaining (KB, query)
    INITIALISE agenda with known facts
    While agenda is NOT empty
        POP fact from agenda
        If fact matches query:
        Return true
        FOR each rule in KB:
        If fact satisfies a rule:
            ADD rules
        Return false

Output

Code :

```
from itertools import product

# Define a function to evaluate a propositional expression
def evaluate(expr, model):
    """
    Evaluates the given expression based on the values in the model.
    """
    for var, val in model.items():
        expr = expr.replace(var, str(val))
    return eval(expr)

# Define the truth-table enumeration algorithm
def truth_table_entails(KB, query, symbols):
    """
    Checks if KB entails query using truth-table enumeration.
    KB: list of propositional expressions (strings)
    query: propositional expression (string)
    symbols: list of symbols (propositions) in the KB and query
    """
    # Generate all possible truth assignments
    assignments = list(product([False, True], repeat=len(symbols)))

    entailing_models = []

    # Iterate over each assignment to check entailment
    for assignment in assignments:
        model = dict(zip(symbols, assignment))

        # Check if KB is true in this model
        KB_is_true = all(evaluate(expr, model) for expr in KB)

        # If KB is true, check if query is also true
        if KB_is_true:
            query_is_true = evaluate(query, model)
            if query_is_true:
                entailing_models.append(model)  # Store the model
            else:
                return False, []
               # Found a model where KB is true but query is false

    return True, entailing_models  # KB entails query if no counterexample was
found

# Get input from the user
```

```
symbols = input("Enter the propositions (symbols) separated by spaces: ").split()
KB = []
n = int(input("Enter the number of statements in the knowledge base: "))

for i in range(n):
    expr = input(f"Enter statement {i + 1} in the knowledge base: ")
    KB.append(expr)

query = input("Enter the query: ")

# Check entailment
result, models = truth_table_entails(KB, query, symbols)
if truth_table_entails(KB, query, symbols):
    print("KB entails the query.")
    print("Models where KB entails query:")
    for model in models:
        print(model)
else:
    print("KB does not entail the query.")


print("Varsha P(1BMS22CS320)")
```

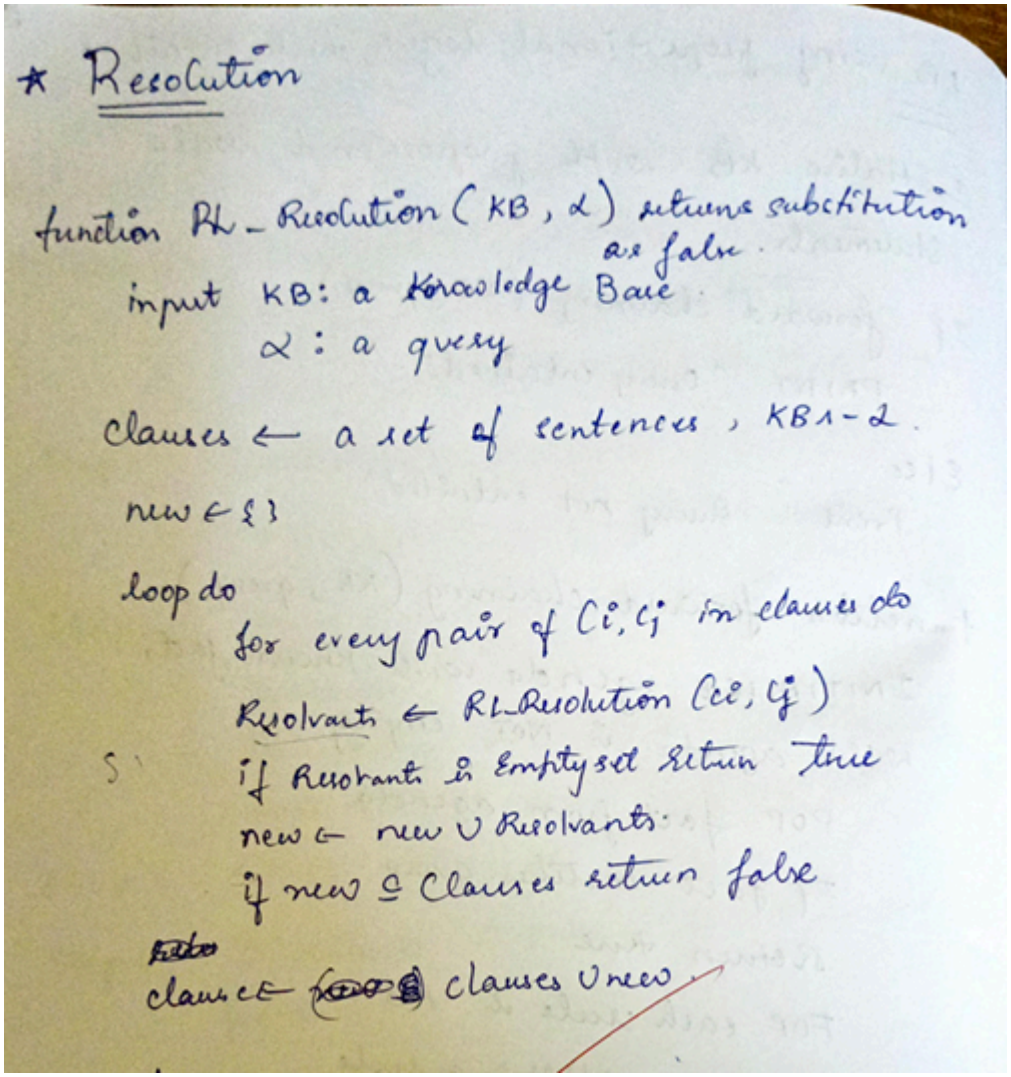
Output :

```
Enter the propositions (symbols) separated by spaces: A B C
Enter the number of statements in the knowledge base: 2
Enter statement 1 in the knowledge base: A or C
Enter statement 2 in the knowledge base: B or not C
Enter the query: A or B
KB entails the query.
Models where KB entails query:
{'A': False, 'B': True, 'C': True}
{'A': True, 'B': False, 'C': False}
{'A': True, 'B': True, 'C': False}
{'A': True, 'B': True, 'C': True}
Varsha P(1BMS22CS320)
```

Program 8

Create a knowledge base using propositional logic and prove the given query using resolution.

Algorithm :



* Resolution

function PL_Resolution (KB, α) returns substitution or false.

input KB: a Knowledge Base
α : a query

clauses ← a set of sentences, KB ∧ ¬α.

new ← {}

loop do
for every pair of Ci, Cj in clauses do
Resolvents ← RL_Resolution (Ci, Cj)
if Resolvents is Empty set return true
new ← new ∪ Resolvents.
if new ⊆ Clauses return false
clauses ← clauses ∪ new

Code :

```
# Open In Colab

# Step 1: Helper function to parse user inputs into clauses (with '!' for negation)
def parse_clause(clause_input):
    """
    Parses a user input string into a tuple of literals for the clause.
    Replaces '!' with '¬' for negation handling.
    Example: "!Food(x), Likes(John, x)" -> ("¬Food(x)", "Likes(John, x)")
    """
    return tuple(literal.strip().replace("!", "¬") for literal in
clause_input.split(","))

# Step 2: Collect knowledge base (KB) from user
def get_knowledge_base():
    print("Enter the premises of the knowledge base, one by one.")
    print("Use ',' to separate literals in a clause. Use '!' for negation.")
    print("Example: !Food(x), Likes(John, x)")
    print("Type 'done' when finished.")

    kb = []
    while True:
        clause_input = input("Enter a clause (or 'done' to finish): ").strip()
        if clause_input.lower() == "done":
            break
        kb.append(parse_clause(clause_input))

    return kb

# Step 3: Add negated conclusion
def get_negated_conclusion():
    print("\nEnter the conclusion to be proved.")
    print("It will be negated automatically for proof by contradiction.")
    conclusion = input("Enter the conclusion (e.g., Likes(John, Peanuts)):
").strip()
    negated = f"!{conclusion}" if not conclusion.startswith("!") else
conclusion[1:]
    return (negated.replace("!", "¬"),)  # Convert '!' to '¬' for consistency

# Step 4: Resolution algorithm
def resolve(clause1, clause2):
    """
    Resolves two clauses and returns the resolvent (new clause).
    If no resolvable literals exist, returns an empty set.
    """
    resolved = set()
```

```
    for literal in clause1:
        if "¬" + literal in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove("¬" + literal)
            resolved = temp1.union(temp2)
        elif literal.startswith("¬") and literal[1:] in clause2:
            temp1 = set(clause1)
            temp2 = set(clause2)
            temp1.remove(literal)
            temp2.remove(literal[1:])
            resolved = temp1.union(temp2)
    return tuple(resolved)

def resolution(kb):
    """
    Runs the resolution algorithm on the knowledge base (KB).
    Returns True if an empty clause is derived (proving the conclusion),
    or False if resolution fails.
    """
    clauses = set(kb)
    new = set()

    while True:
        # Generate all pairs of clauses
        pairs = [(ci, cj) for ci in clauses for cj in clauses if ci != cj]

        for (ci, cj) in pairs:
            resolvent = resolve(ci, cj)
            if not resolvent:  # Empty clause found
                return True
            new.add(resolvent)

        if new.issubset(clauses):  # No new clauses
            return False
        clauses = clauses.union(new)

# Step 5: Main execution
if __name__ == "__main__":
    print("=== Resolution Proof System ===")
    print("Provide the knowledge base and conclusion to prove.")

    # Collect user inputs
    kb = get_knowledge_base()
    negated_conclusion = get_negated_conclusion()
    kb.append(negated_conclusion)
```

```
    # Show the knowledge base
    print("\nKnowledge Base (KB):")
    for clause in kb:
        print("   ", " ∨ ".join(clause))  # Join literals with OR for readability

    # Perform resolution
    print("\nStarting resolution process...")
    result = resolution(kb)
    if result:
        print("\nProof complete: The conclusion is TRUE.")
    else:
        print("\nResolution failed: The conclusion could not be proved.")


print("Varsha P(1BMS22CS320)")
```

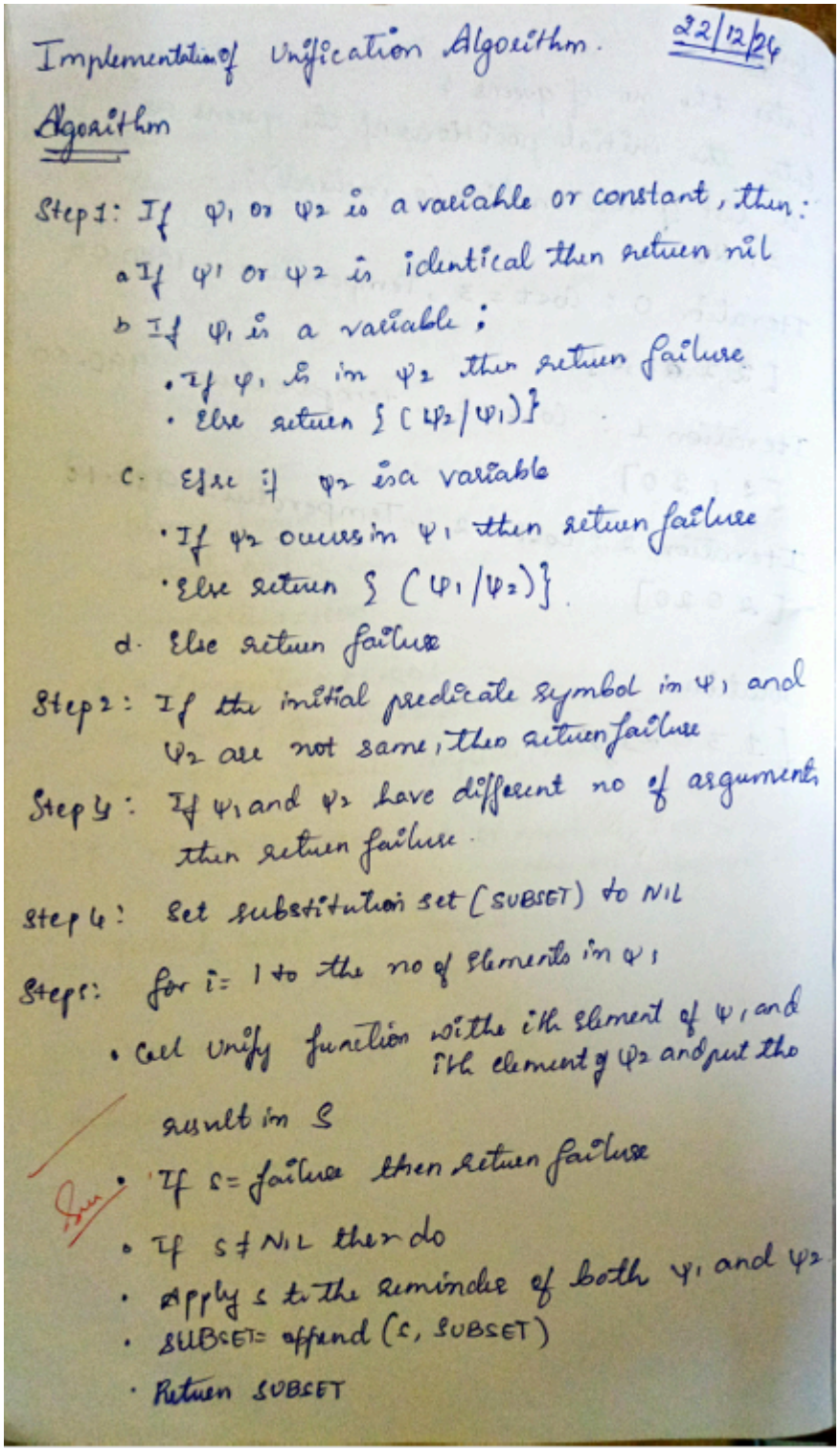
Output :

```
=== Resolution Proof System ===
Provide the knowledge base and conclusion to prove.
Enter the premises of the knowledge base, one by one.
Use ',' to separate literals in a clause. Use '!' for negation.
Example: !Food(x), Likes(John, x)
Type 'done' when finished.
Enter a clause (or 'done' to finish): !Food(x), Likes(John,x)
Enter a clause (or 'done' to finish): Food(Apple)
Enter a clause (or 'done' to finish): Food(Vegetables)
Enter a clause (or 'done' to finish): !Eats(x,y), !Killed(x), Food(y)
Enter a clause (or 'done' to finish): Eats(Anil,Peanuts)
Enter a clause (or 'done' to finish): !Killed(Anil)
Enter a clause (or 'done' to finish): done

Enter the conclusion to be proved.
It will be negated automatically for proof by contradiction.
Enter the conclusion (e.g., Likes(John, Peanuts)): Likes(John,Peanuts)

Knowledge Base (KB):
   ¬Food(x) ∨ Likes(John ∨ x)
   Food(Apple)
   Food(Vegetables)
   ¬Eats(x ∨ y) ∨ ¬Killed(x) ∨ Food(y)
   Eats(Anil ∨ Peanuts)
   ¬Killed(Anil)
   ¬Likes(John,Peanuts)

Starting resolution process...

Proof complete: The conclusion is TRUE.
Varsha P(1BMS22CS320)
```

Program 9

Implement unification in first order logic.

Algorithm :



Implementation of Unification Algorithm. 22/12/24

Algorithm

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

a. If ψ_1 or ψ_2 is identical then return nil

b. If ψ_1 is a variable;

- If ψ_1 is in ψ_2 then return failure
- Else return $\{(\psi_2/\psi_1)\}$

c. Else if ψ_2 is a variable

- If ψ_2 occurs in ψ_1 then return failure
- 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 no of arguments then return failure.

Step 4: Set substitution set (SUBSET) to NIL

Step 5: for i = 1 to the no of elements in ψ_1

- Call Unify function with the ith element of ψ_1 and ith element of ψ_2 and put the result in S
- If S = failure then return failure
- If S ≠ NIL then do
- Apply S to the remainder of both ψ_1 and ψ_2
- SUBSET = append (S, SUBSET)
- Return SUBSET

Code :

```
def unify(s1, s2, theta={}):

    if theta is None:
        return None

    if s1 == s2:
        return theta

    if isinstance(s1, str) and s1.islower():
        return unify_var(s1, s2, theta)

    if isinstance(s2, str) and s2.islower():
        return unify_var(s2, s1, theta)

    if isinstance(s1, tuple) and isinstance(s2, tuple) and len(s1) == len(s2):
        return unify(s1[1:], s2[1:], unify(s1[0], s2[0], theta))

    return None

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

def occurs_check(var, x, theta):
    if var == x:
        return True
    elif isinstance(x, str) and x.islower() and x in theta:
        return occurs_check(var, theta[x], theta)
    elif isinstance(x, tuple):
        for arg in x:
            if occurs_check(var, arg, theta):
                return True
    return False



s1 = ('p', 'x', ('f', 'x'), ('y'))
s2 = ('p', 'a', 'y', ('f', 'x'))

substitution = unify(s1, s2)
```

```
if substitution:
    print("Unification successful:")
    print(f"Substitution: {substitution}")
else:
    print("Unification failed.")
```

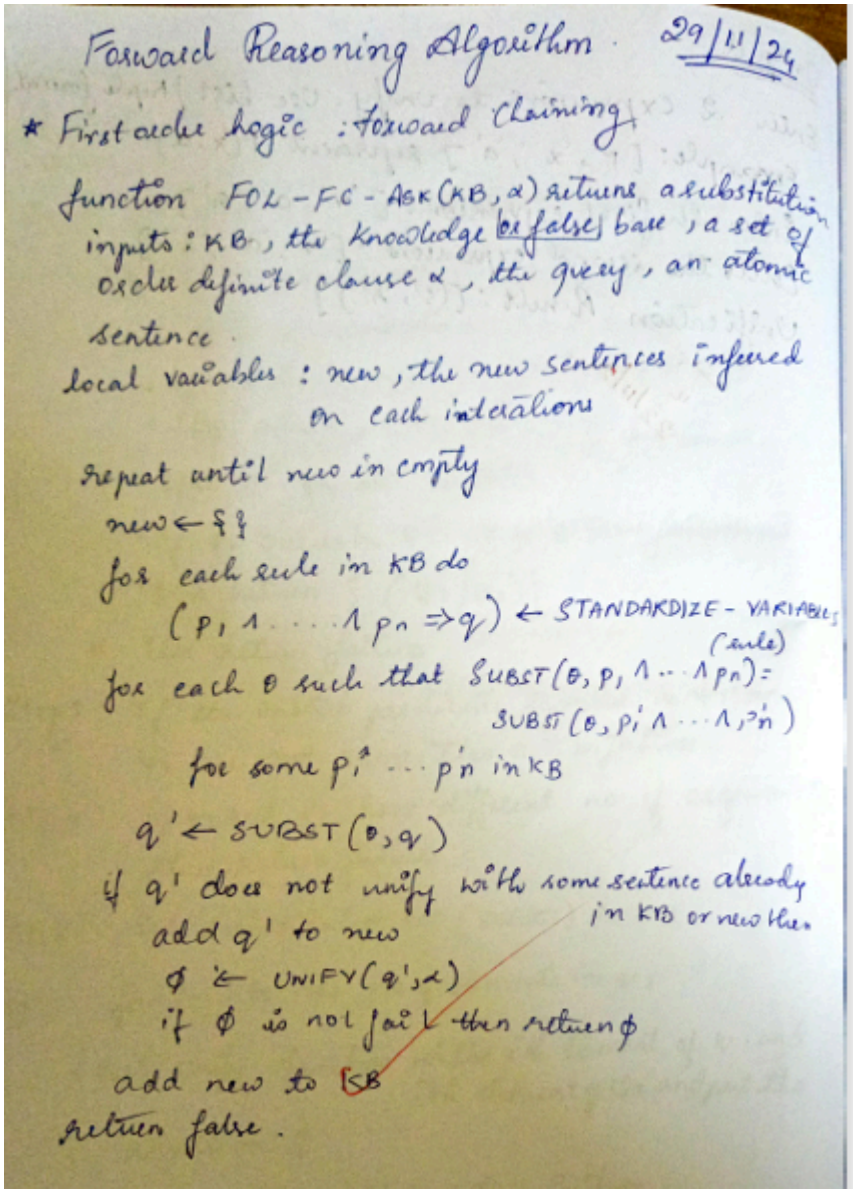
Output:

```
Unification successful:
Substitution: {'x': 'a', 'y': ('f', 'x')}
```

Program 10

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

Algorithm :



Forward Reasoning Algorithm 29/11/24

* First order logic : forward chaining

function FOL-FC-ASK(KB, α) returns a substitution or false
inputs : KB, the knowledge base, a set of first order definite clause
α, the query, an atomic sentence
local variables : new, the new sentences inferred on each iterations

repeat until new is empty
new ← { }
for each rule in KB do
(p1 ∧ ... ∧ pn ⇒ q) ← STANDARDIZE-VARIABLES (rule)
for each θ such that SUBST(θ, p1 ∧ ... ∧ pn) = SUBST(θ, p'1 ∧ ... ∧ p'n)
for some p'1 ... p'n in KB
q' ← SUBST(θ, q)
if q' does not unify with some sentence already in KB or new then
add q' to new
φ ← UNIFY(q', α)
if φ is not fail then return φ
add new to KB
return false.

Code :

```
# Define the knowledge base as a list of rules and facts

class KnowledgeBase:
    def __init__(self):
        self.facts = set()    # Set of known facts
        self.rules = []       # List of rules

    def add_fact(self, fact):
        self.facts.add(fact)

    def add_rule(self, rule):
        self.rules.append(rule)

    def infer(self):
        inferred = True
        while inferred:
            inferred = False
            for rule in self.rules:
                if rule.apply(self.facts):
                    inferred = True

# Define the Rule class
class Rule:
    def __init__(self, premises, conclusion):
        self.premises = premises  # List of conditions
        self.conclusion = conclusion  # Conclusion to add if premises are met

    def apply(self, facts):
        if all(premise in facts for premise in self.premises):
            if self.conclusion not in facts:
                facts.add(self.conclusion)
                print(f"Inferred: {self.conclusion}")
                return True
        return False

# Initialize the knowledge base
kb = KnowledgeBase()

# Facts in the problem
kb.add_fact("American(Robert)")
kb.add_fact("Missile(T1)")
kb.add_fact("Owns(A, T1)")
kb.add_fact("Enemy(A, America)")
```

```
# Rules based on the problem
# 1. Missile(x) implies Weapon(x)
kb.add_rule(Rule(["Missile(T1)"], "Weapon(T1)"))

# 2. Enemy(x, America) implies Hostile(x)
kb.add_rule(Rule(["Enemy(A, America)"], "Hostile(A)"))

# 3. Missile(x) and Owns(A, x) imply Sells(Robert, x, A)
kb.add_rule(Rule(["Missile(T1)", "Owns(A, T1)"], "Sells(Robert, T1, A)"))

# 4. American(p) and Weapon(q) and Sells(p, q, r) and Hostile(r) imply Criminal(p)
kb.add_rule(Rule(["American(Robert)", "Weapon(T1)", "Sells(Robert, T1, A)",
"Hostile(A)"], "Criminal(Robert)"))

# Infer new facts based on the rules
kb.infer()

# Check if Robert is a criminal
if "Criminal(Robert)" in kb.facts:
    print("Conclusion: Robert is a criminal.")
else:
    print("Conclusion: Unable to prove Robert is a criminal.")



print("Varsha P(1BMS22CS320)")
```

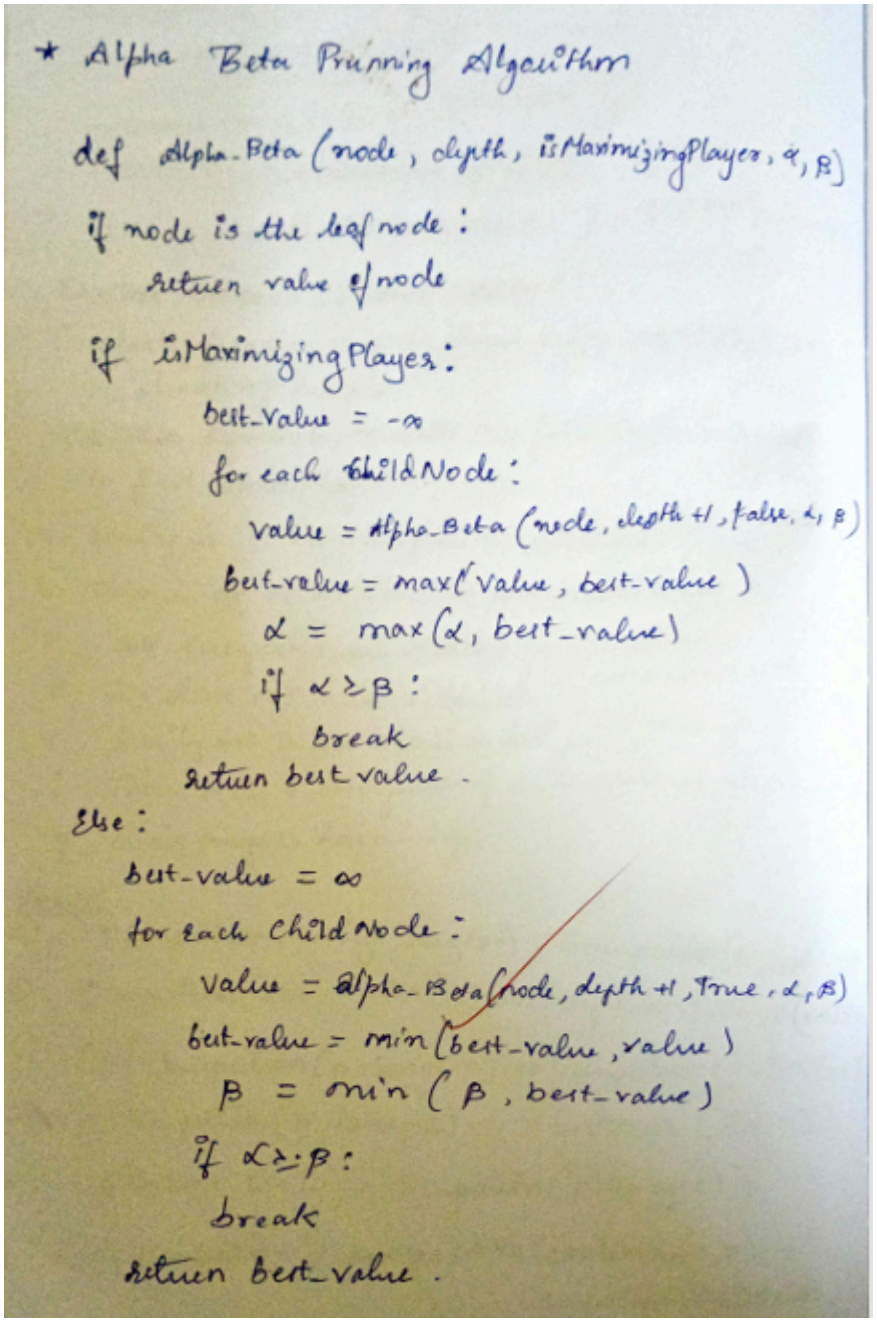
Output :

```
Inferred: Weapon(T1)
Inferred: Hostile(A)
Inferred: Sells(Robert, T1, A)
Inferred: Criminal(Robert)
Conclusion: Robert is a criminal.
Varsha P(1BMS22CS320)
```

Program 11

Implement Alpha-Beta Pruning.

Algorithm :



* Alpha Beta Pruning Algorithm

```
def Alpha-Beta (node, depth, isMaximizingPlayer, α, β)
if node is the leaf node :
    return value of node
if isMaximizingPlayer :
        best_value = -∞
        for each childNode :
            value = Alpha-Beta (node, depth+1, false, α, β)
        best_value = max( value, best_value )
            α = max (α, best_value)
            if α ≥ β :
                break
        return best_value .
Else :
    best_value = ∞
    for each childNode :
        value = Alpha-Beta(node, depth+1, True, α, β)
        best_value = min (best_value, value )
            β = min ( β , best_value )
        if α ≥ β :
            break
    return best_value .
```

Code :

```
# Open In Colab

# Python3 program to demonstrate
# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta
MAX, MIN = 1000, -1000

# Returns optimal value for current player
#(Initially called for root and maximizer)
def minimax(depth, nodeIndex, maximizingPlayer,
      values, alpha, beta):

  # Terminating condition. i.e
  # leaf node is reached
  if depth == 3:
    return values[nodeIndex]

  if maximizingPlayer:

    best = MIN

    # Recur for left and right children
    for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
            False, values, alpha, beta)
      best = max(best, val)
      alpha = max(alpha, best)

      # Alpha Beta Pruning
      if beta <= alpha:
        break

    return best

  else:
    best = MAX

    # Recur for left and
    # right children
    for i in range(0, 2):

      val = minimax(depth + 1, nodeIndex * 2 + i,
              True, values, alpha, beta)
      best = min(best, val)
```

```
      beta = min(beta, best)

      # Alpha Beta Pruning
      if beta <= alpha:
        break

    return best

# Driver Code
if __name__ == "__main__":

  values = [3, 5, 6, 9, 1, 2, 0, -1]
  print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))


print("Varsha P(1BMS22CS320)")
```

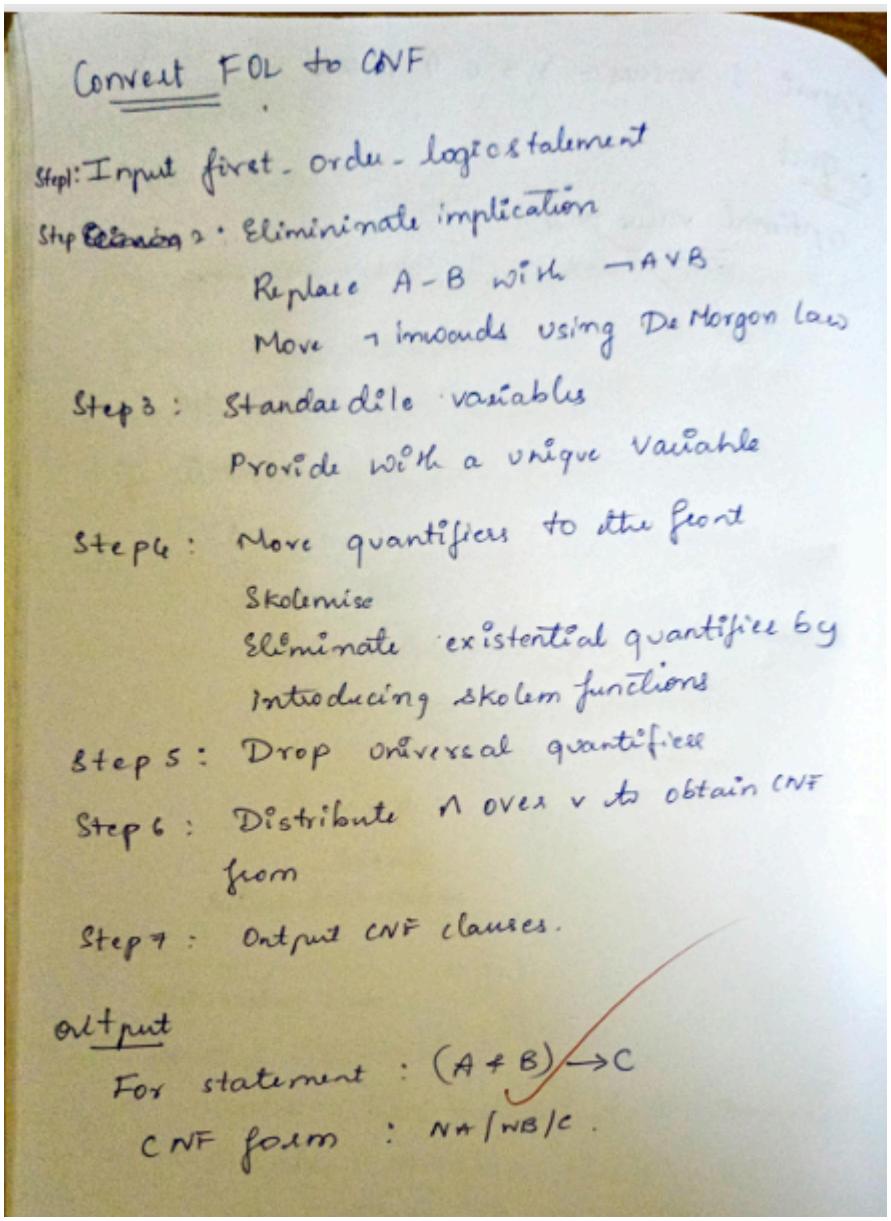
Output :

```
The optimal value is : 5
Varsha P(1BMS22CS320)
```

Program 12

Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm :



Code :

```
from sympy import symbols, Not, Or, And, Implies, Equivalent
from sympy.logic.boolalg import to_cnf


def fol_to_cnf(fol_expr):
    fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b),
Implies(b, a)))
    fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))
    cnf_form = to_cnf(fol_expr, simplify=True)
    return cnf_form


def main():
    P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")


    fol_expr1 = Implies(P, Q)
    print("Example 1: P → Q")
    print("Original FOL Expression:")
    print(fol_expr1)


    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
    print(cnf1)


    fol_expr2 = Implies(Or(P, Not(Q)), Or(Q, R))
    print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)


    cnf2 = fol_to_cnf(fol_expr2)
    print("\nCNF Form:")
    print(cnf2)


if __name__ == "__main__":
    main()
    print("---------")
    print("Varsha P(1BM22CS320)")
```

Output :

```
Example 1: P → Q
Original FOL Expression:
Implies(P, Q)

CNF Form:
Q | ~P

Example 2: (P ∨ ¬Q) → (Q ∨ R)
Original FOL Expression:
Implies(P | ~Q, Q | R)

CNF Form:
Q | R
---------
Varsha P(1BM22CS320)
```