Artificial Intelligence Lab Report

Submitted by

Vyom Gupta (1BM22CS333)
Batch: 3

Course: Artificial Intelligence
Course Code: 24CS5PCAIN
Sem & Section: 5F

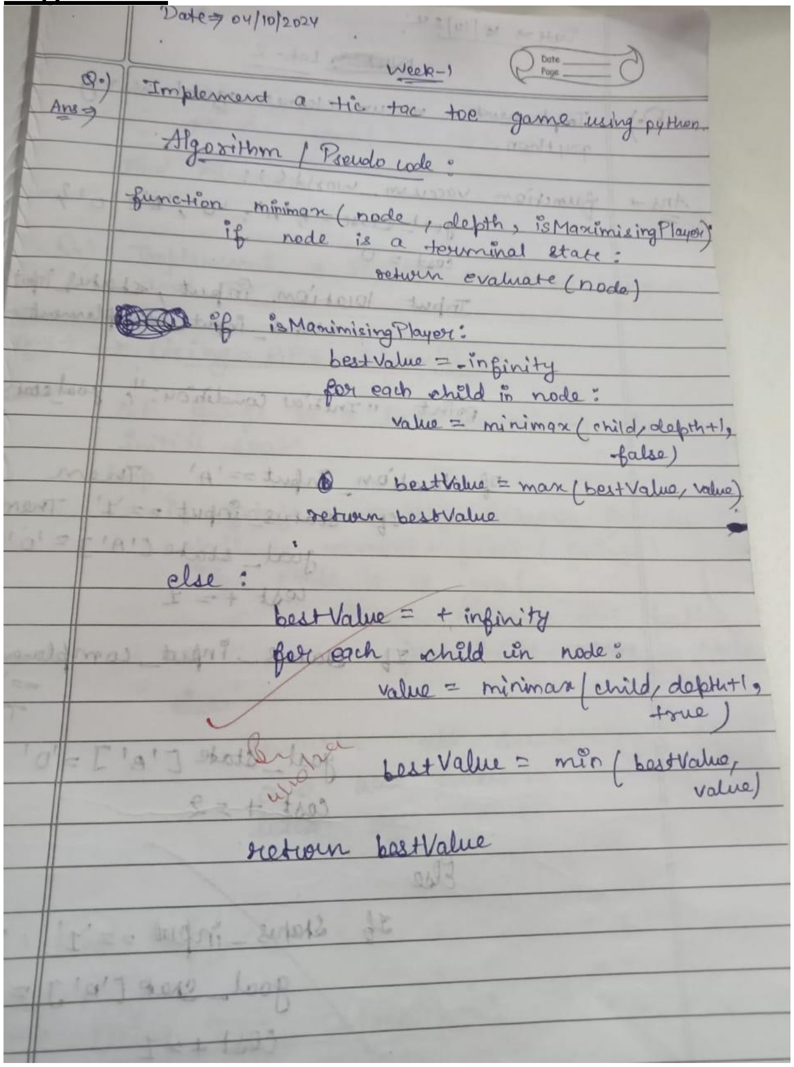
BACHELOR OF ENGINEERING
in
COMPUTER SCIENCE AND ENGINEERING

B. M. S. COLLEGE OF ENGINEERING
(Autonomous Institution under VTU)
BENGALURU-560019
2023-2024

Table of contents

Program 1 - Tic Tac toe

Algorithm



Date ⇒ 04/10/2024

Week-1

Q.) Implement a tic tac toe game using python.

Ans ⇒ Algorithm / Pseudo code :

```
function minimax ( node , depth , isMaximisingPlayer)
        if node is a terminal state :
                return evaluate (node)
        if isMaximisingPlayer:
                bestValue = -infinity
                for each child in node :
                        value = minimax ( child, depth+1,
                                          false)
                        bestValue = max (bestValue, value)
                return bestValue

        else :
                bestValue = + infinity
                for each child in node :
                        value = minimax ( child, depth+1,
                                          true )

                        bestValue = min ( bestValue,
                                          value)

                return bestValue
```

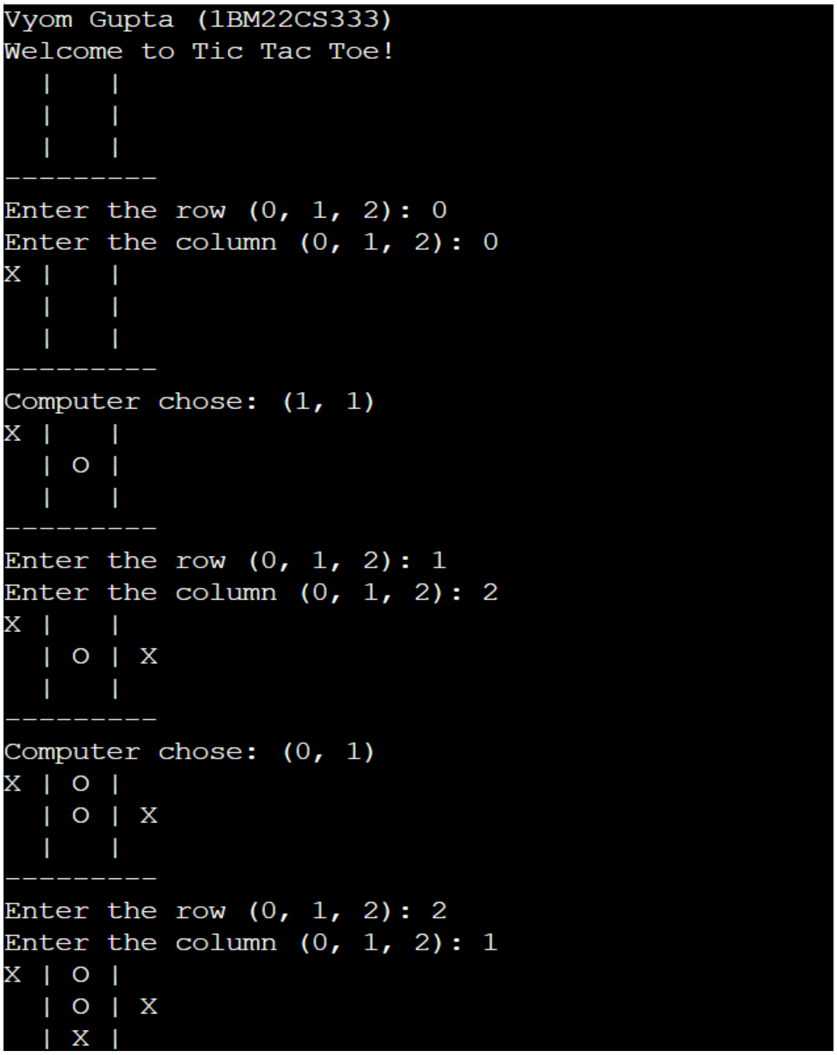
Code

```
import random
import math


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


def check_winner(board, mark):
    # Check rows, columns, and diagonals for a win
    for row in board:
        if all(cell == mark for cell in row):
            return True

    for col in range(3):
        if all(board[row][col] == mark for row in range(3)):
            return True
    if all(board[i][i] == mark for i in range(3)) or all(board[i][2 - i] == mark for i in range(3)):
        return True

    return False

def get_available_moves(board):
    return [(r, c) for r in range(3) for c in range(3) if board[r][c] == " "]

def minimax(board, depth, is_maximizing):
    if check_winner(board, "O"):
        return 10 - depth
    if check_winner(board, "X"):
        return depth - 10
    if not get_available_moves(board):
        return 0

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


def computer_move(board):
    best_score = -math.inf
    best_move = None
    for (row, col) in get_available_moves(board):
        board[row][col] = "O"
        score = minimax(board, 0, False)
        board[row][col] = " "
        if score > best_score:
            best_score = score
            best_move = (row, col)
    return best_move


def main():
    print("Vyom Gupta (1BM22CS333)")
    print("Welcome to Tic Tac Toe!")
    board = [[" " for _ in range(3)] for _ in range(3)]

    print_board(board)

    for turn in range(9):
        if turn % 2 == 0:
            # Player's turn
            while True:
                try:
```

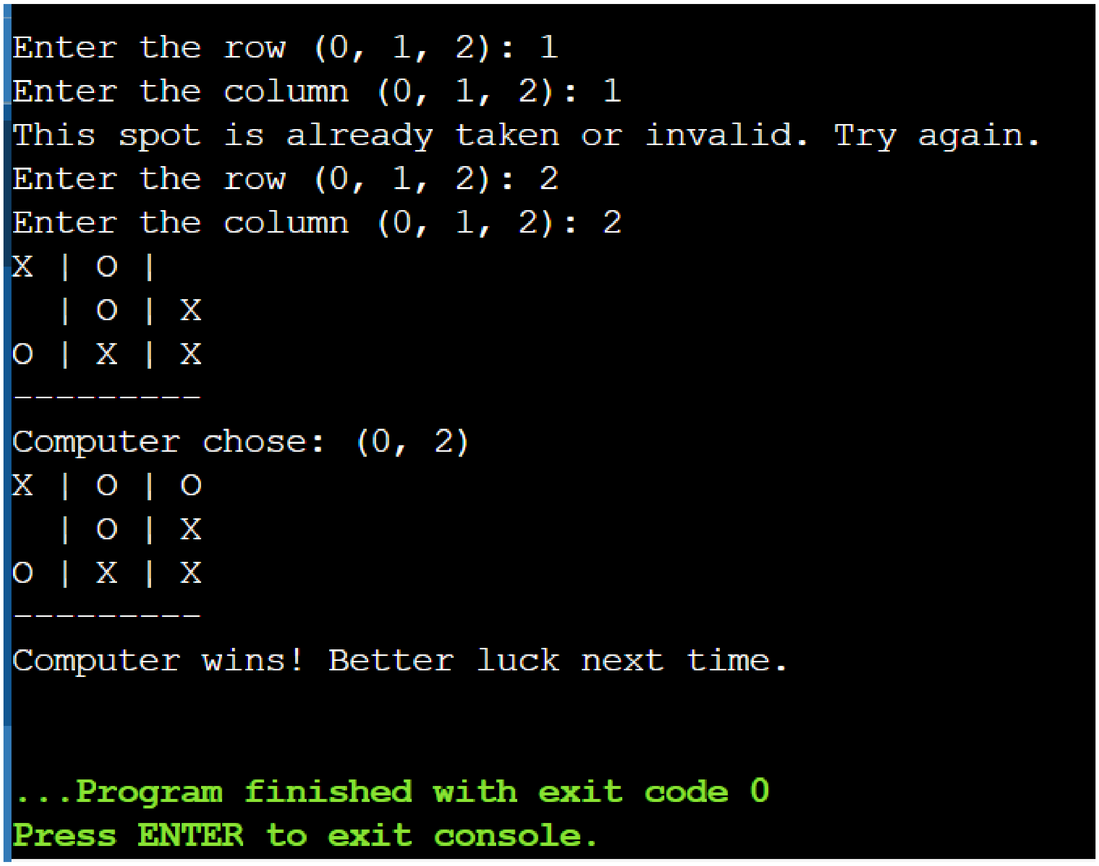
```
            row = int(input("Enter the row (0, 1, 2): "))
            col = int(input("Enter the column (0, 1, 2): "))

            if (row, col) not in get_available_moves(board):
                print("This spot is already taken or invalid. Try again.")
            else:
                board[row][col] = "X"
                break
        except ValueError:
            print("Invalid input. Please enter numbers 0, 1, or 2.")
else:
    # Computer's turn
    row, col = computer_move(board)
    board[row][col] = "O"
    print(f"Computer chose: ({row}, {col})")

print_board(board)

# Check for a winner
```

```
        if check_winner(board, "X"):
            print("Congratulations! You win!")
            return
        elif check_winner(board, "O"):
            print("Computer wins! Better luck next time.")
            return

    print("It's a tie!")

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

Output



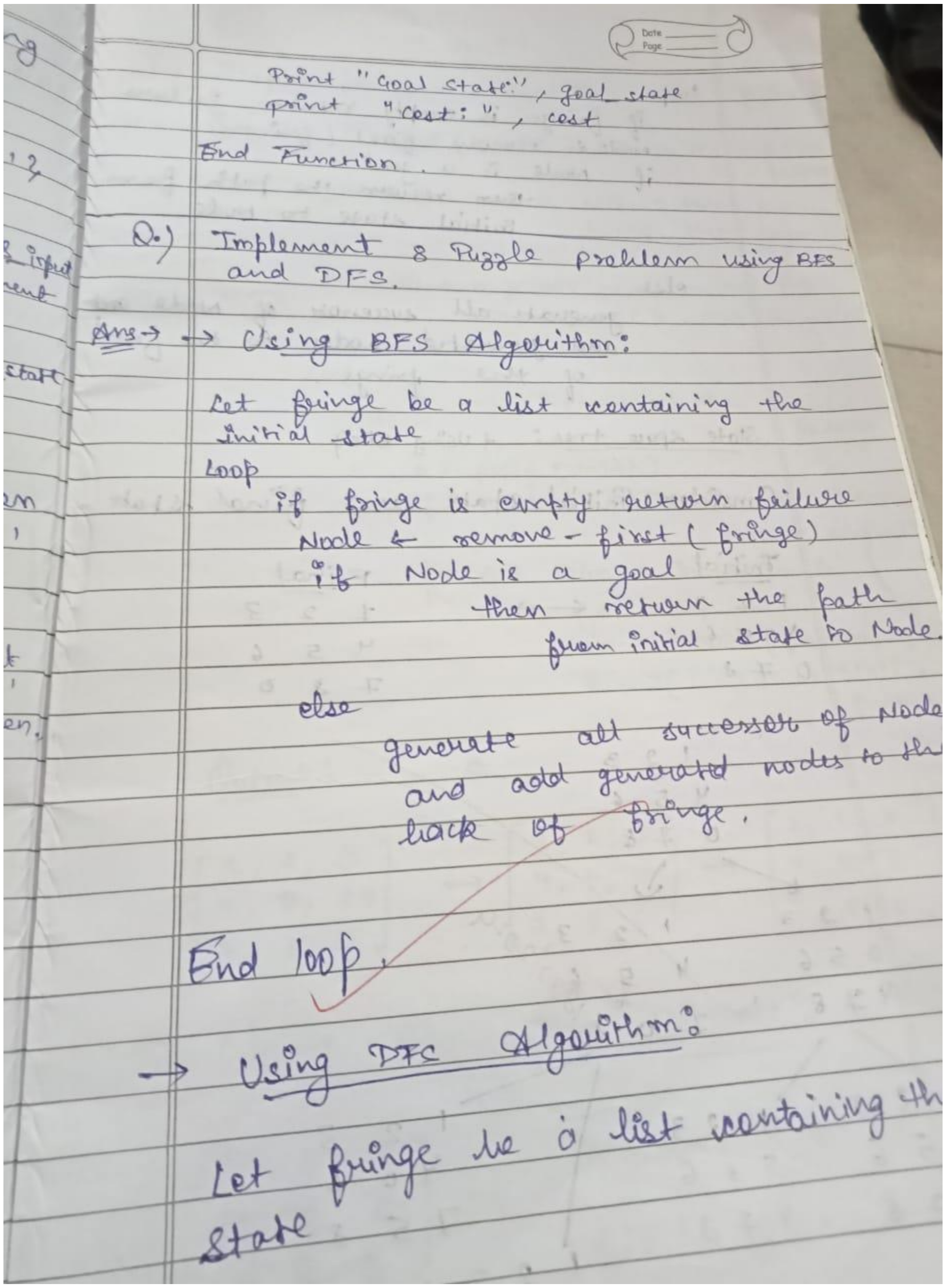
```
Vyom Gupta (1BM22CS333)
Welcome to Tic Tac Toe!
  |   |
  |   |
  |   |
---------
Enter the row (0, 1, 2): 0
Enter the column (0, 1, 2): 0
X |   |
  |   |
  |   |
---------
Computer chose: (1, 1)
X |   |
  | O |
  |   |
---------
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 2
X |   |
  | O | X
  |   |
---------
Computer chose: (0, 1)
X | O |
  | O | X
  |   |
---------
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 1
X | O |
  | O | X
  | X |
```



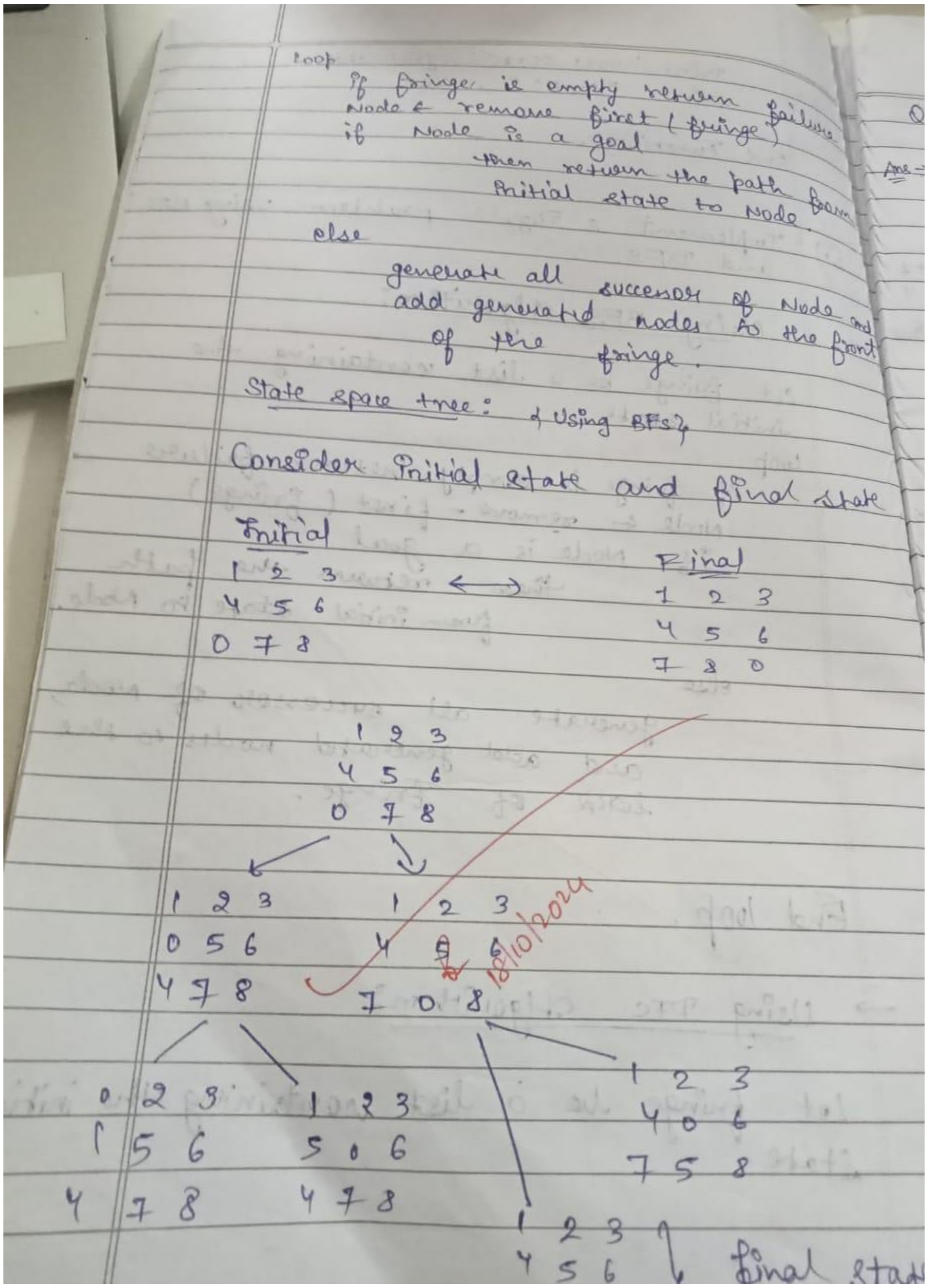
```
Enter the row (0, 1, 2): 1
Enter the column (0, 1, 2): 1
This spot is already taken or invalid. Try again.
Enter the row (0, 1, 2): 2
Enter the column (0, 1, 2): 2
X | O |
  | O | X
O | X | X
---------
Computer chose: (0, 2)
X | O | O
  | O | X
O | X | X
---------
Computer wins! Better luck next time.


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 2 - **8 Puzzle BFS and DFS**

Algorithm



```
        Print "Goal State:", goal_state
        print "Cost: ", cost

End Function.

Q.) Implement 8 Puzzle problem using BFS
    and DFS.

Ans→ → Using BFS Algorithm:

Let fringe be a list containing the
initial state
Loop
    if fringe is empty return failure
      Node ← remove-first (fringe)
      if Node is a goal
              then return the path
                   from initial state to Node.

      else
          generate all successor of Node
          and add generated nodes to the
          back of fringe.

End loop.

→ Using DFS Algorithm:

Let fringe be a list containing th
state
```



```
Loop
    if fringe is empty return failure
    Node ← remove first (fringe)
    if Node is a goal
        then return the path from
            initial state to Node.
else
    generate all successor of Node and
    add generated nodes to the front
        of the fringe.
```

State space tree: {Using BFS}

Consider initial state and final state

Initial ⟷ Final

1 2 3 ⟷ 1 2 3
4 5 6 ⟷ 4 5 6
0 7 8 ⟷ 7 8 0

1 2 3
4 5 6
0 7 8

1 2 3 | 1 2 3
0 5 6 | 4 0 6
4 7 8 | 7 5 8

0 2 3 | 1 2 3
1 5 6 | 5 0 6
4 7 8 | 4 7 8

1 2 3 | 1 2 3
4 5 6 | 4 0 6
7 0 8 | 7 5 8

1 2 3
4 5 6 } final state

18/10/2024

Code (BFS)

```
from collections import deque


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


def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()
```

```
def get_neighbors(state):
    rows, cols = len(state), len(state[0])
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors


def bfs(initial, goal):
    queue = deque([(initial, [])])
    visited = set()
    visited.add(tuple(tuple(row) for row in initial))

    while queue:
        current, path = queue.popleft()
        if current == goal:
            return path + [current]

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                queue.append((neighbor, path + [current]))
    return None

def main():
    print("Vyom Gupta(1BM22CS333)")
    print("8-Puzzle Solver Using BFS")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
```

```
goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]    # Goal state

print_state(initial_state, label="Initial State:")
print_state(goal_state, label="Goal State:")

if not is_solvable(initial_state):
    print("This puzzle is not solvable.")
    return

solution = bfs(initial_state, goal_state)
if solution:
    print("Solution found in {} steps:\n".format(len(solution) - 1))
    for i, step in enumerate(solution):
        if i == 0:
            print_state(step, label="Initial State:")
        elif i == len(solution) - 1:
            print_state(step, label="Final State:")
        else:
            print_state(step, label=f"Step {i}:")
    else:

        print("No solution exists.")



if _name___== "_main_":

    main()
```

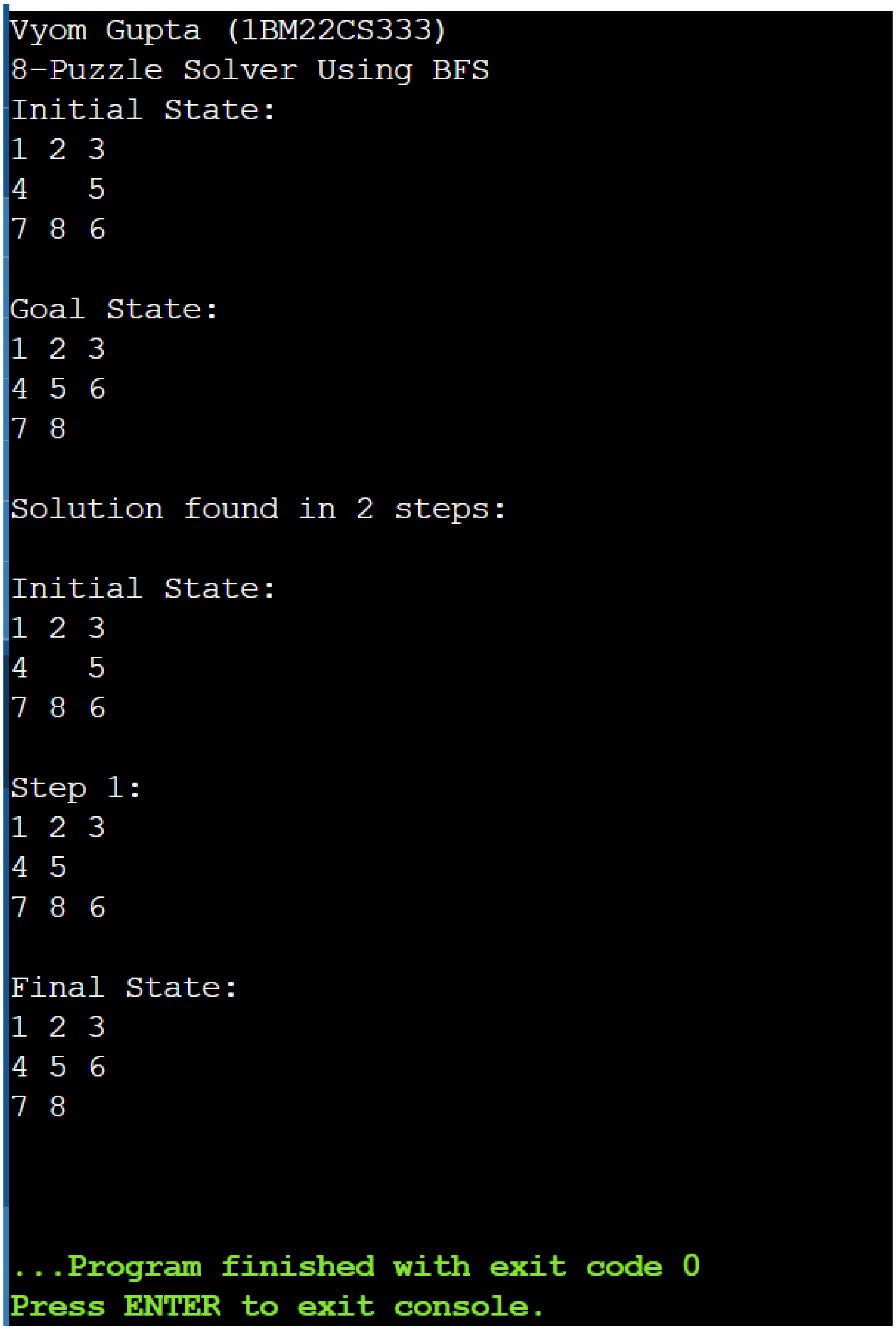
Code (DFS)

```
def is_solvable(state):

    inversions = 0

    flattened = [num for row in state for num in row if num != 0]

    for i in range(len(flattened)):

        for j in range(i + 1, len(flattened)):

            if flattened[i] > flattened[j]:

                inversions += 1

    return inversions % 2 == 0



def print_state(state, label=None):

    if label:

        print(label)

    for row in state:

        print(" ".join(str(num) if num != 0 else " " for num in row))

    print()



def get_neighbors(state):

    rows, cols = len(state), len(state[0])
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc], new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors


def dfs(initial, goal):
    stack = [(initial, [])]
    visited = set()
    visited.add(tuple(tuple(row) for row in initial))
    while stack:
        current, path = stack.pop()
        if current == goal:
            return path + [current]

        for neighbor in get_neighbors(current):
            neighbor_tuple = tuple(tuple(row) for row in neighbor)
            if neighbor_tuple not in visited:
                visited.add(neighbor_tuple)
                stack.append((neighbor, path + [current]))
    return None


def main()
    print("Vyom Gupta (1BM22CS333)")
    print("8-Puzzle Solver Using DFS")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]      # Goal state
```

```
print_state(initial_state, label="Initial State:")
print_state(goal_state, label="Goal State:")

if not is_solvable(initial_state):
    print("This puzzle is not solvable.")
    return

solution = dfs(initial_state, goal_state)
if solution:
    print("Solution found in {} steps:\n".format(len(solution) - 1))
    for i, step in enumerate(solution):
        if i == 0:
            print_state(step, label="Initial State:")
        elif i == len(solution) - 1:
            print_state(step, label="Final State:")
        else:
            print_state(step, label=f"Step {i}:")
else:
    print("No solution exists.")
```

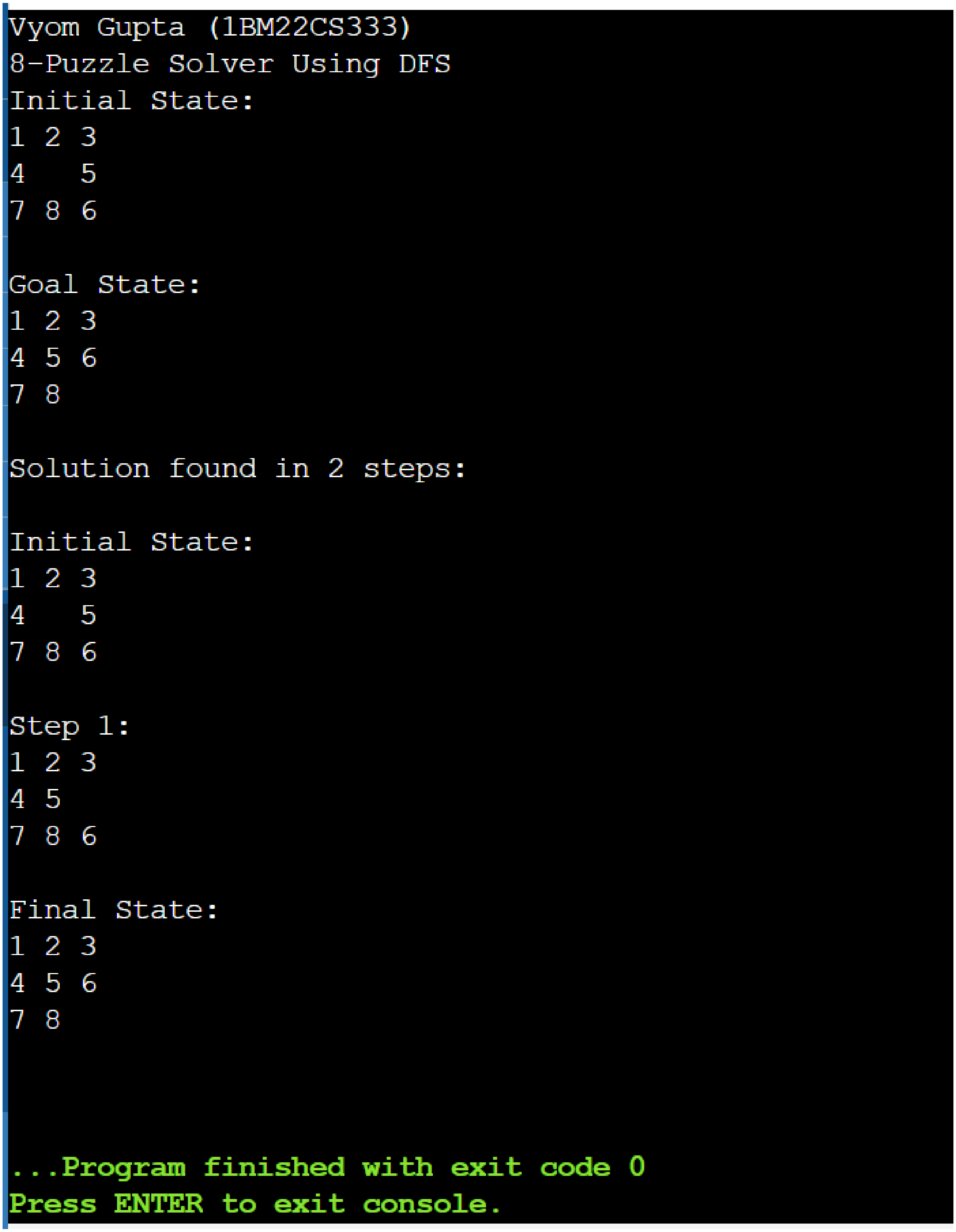
```
if _name___== "_main_":

    main()
```

Output (BFS)



```
Vyom Gupta (1BM22CS333)
8-Puzzle Solver Using BFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8



...Program finished with exit code 0
Press ENTER to exit console.
```

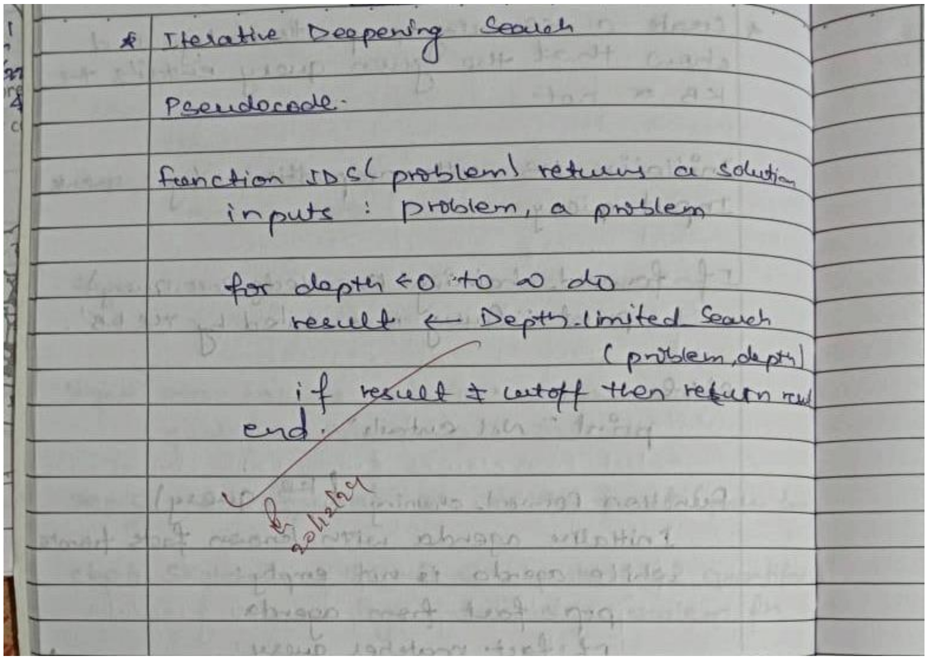
Output (DFS)



```
Vyom Gupta (1BM22CS333)
8-Puzzle Solver Using DFS
Initial State:
1 2 3
4   5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 2 steps:

Initial State:
1 2 3
4   5
7 8 6

Step 1:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8



...Program finished with exit code 0
Press ENTER to exit console.
```

Program 3 - Iterative Deepening Search

Algorithm



* Iterative Deepening Search

Pseudocode.

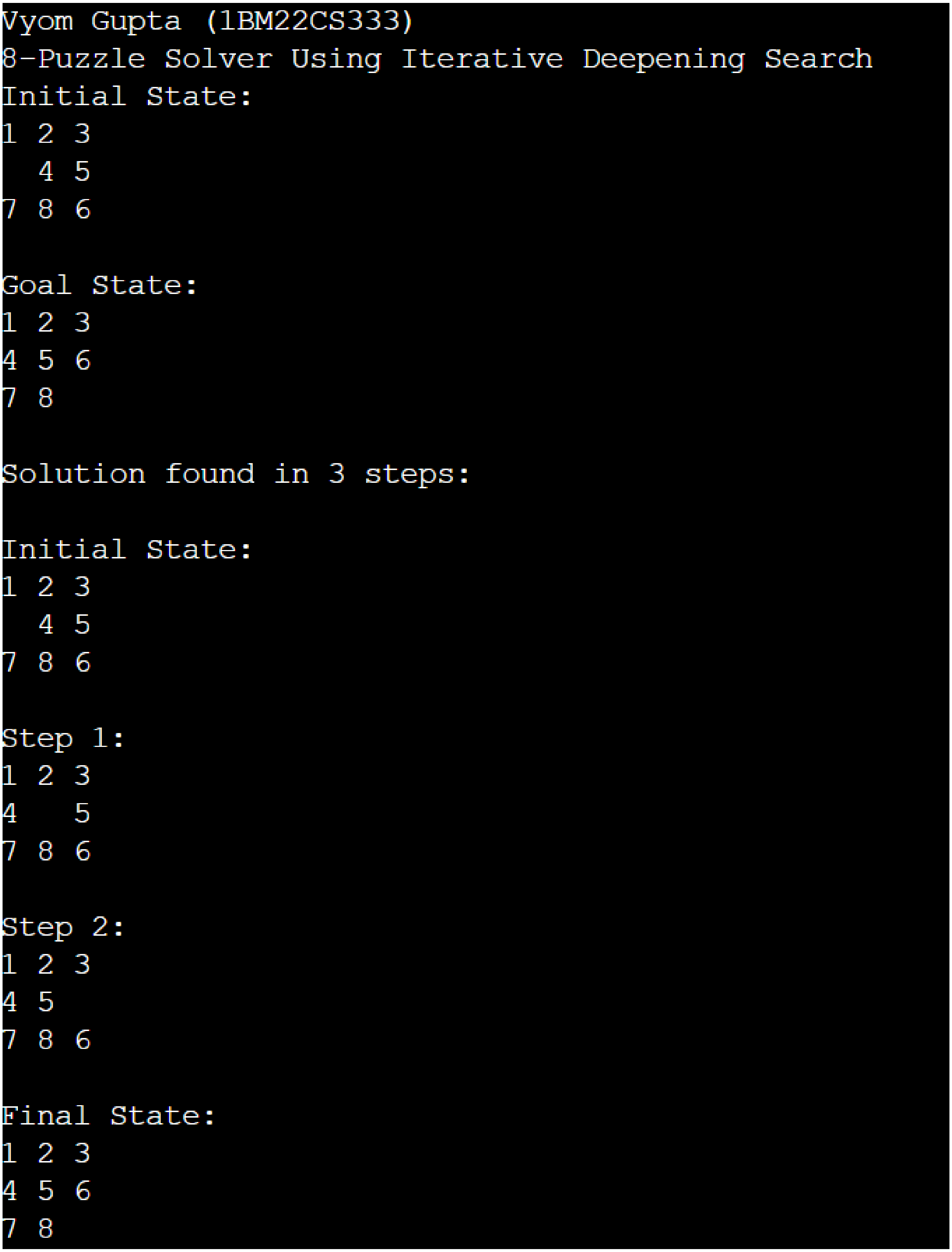
```
function IDS( problem) returns a solution
        inputs : problem, a problem

        for depth ← 0 to ∞ do
            result ← Depth-Limited Search
                            ( problem, depth)
            if result ≠ cutoff then return result
        end.
```

20/2/24

Code

```
print("Vyom Gupta (1BM22CS333)")

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

def print_state(state, label=None):
    if label:
        print(label)
    for row in state:
        print(" ".join(str(num) if num != 0 else " " for num in row))
    print()

def get_neighbors(state):
    rows, cols = len(state), len(state[0])
    for r in range(rows):
        for c in range(cols):
            if state[r][c] == 0:
                zero_pos = (r, c)
                break
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    neighbors = []
    for dr, dc in directions:
        nr, nc = zero_pos[0] + dr, zero_pos[1] + dc
        if 0 <= nr < rows and 0 <= nc < cols:
            new_state = [row[:] for row in state]
            new_state[zero_pos[0]][zero_pos[1]], new_state[nr][nc] = new_state[nr][nc],
new_state[zero_pos[0]][zero_pos[1]]
            neighbors.append(new_state)
    return neighbors

def ids(initial, goal, depth_limit):
    def dls(state, path, depth):
        if state == goal:
            return path + [state]
        if depth == 0:
            return None
        for neighbor in get_neighbors(state):
            if tuple(tuple(row) for row in neighbor) not in visited:
                visited.add(tuple(tuple(row) for row in neighbor))
                result = dls(neighbor, path + [state], depth - 1)
                if result:
                    return result
        return None

    for depth in range(depth_limit):
        visited = set()
        visited.add(tuple(tuple(row) for row in initial))
        result = dls(initial, [], depth)
        if result:
            return result
    return None

def main():
    print("8-Puzzle Solver Using Iterative Deepening Search")
    initial_state = [[1, 2, 3], [4, 0, 5], [7, 8, 6]]  # Example initial state
    goal_state = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]       # Goal state

    print_state(initial_state, label="Initial State:")
    print_state(goal_state, label="Goal State:")

    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return

    depth_limit = 20
    solution = ids(initial_state, goal_state, depth_limit)
    if solution:
        print("Solution found in {} steps:\n".format(len(solution) - 1))
        for i, step in enumerate(solution):
            if i == 0:
                print_state(step, label="Initial State:")
            elif i == len(solution) - 1:
                print_state(step, label="Final State:")
            else:
                print_state(step, label=f"Step {i}:")
    else:
        print("No solution exists within depth limit {}.".format(depth_limit))

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

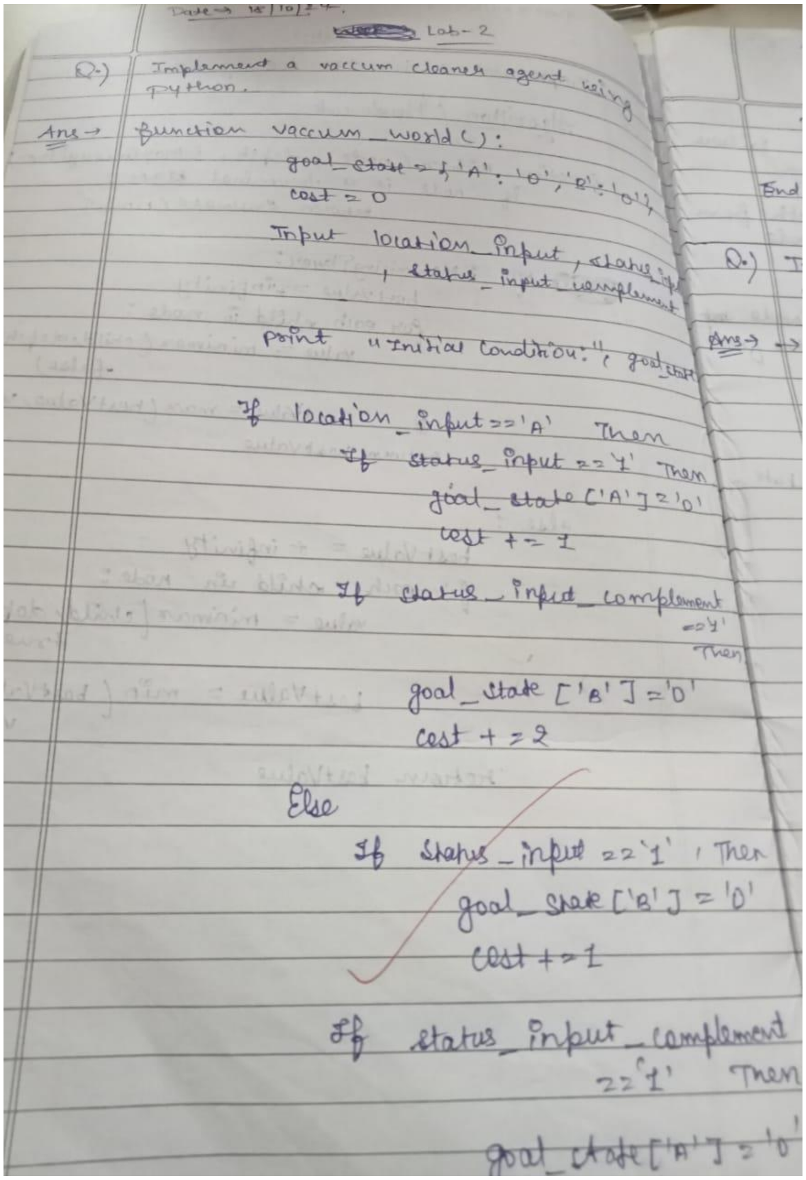
Output



```
Vyom Gupta (1BM22CS333)
8-Puzzle Solver Using Iterative Deepening Search
Initial State:
1 2 3
  4 5
7 8 6

Goal State:
1 2 3
4 5 6
7 8

Solution found in 3 steps:

Initial State:
1 2 3
  4 5
7 8 6

Step 1:
1 2 3
4   5
7 8 6

Step 2:
1 2 3
4 5
7 8 6

Final State:
1 2 3
4 5 6
7 8
```

Program 4 - Vacuum Cleaner Agent

Algorithm



Date → 18/10/24.

Lab - 2

Q.) Implement a vaccum cleaner agent using python.

Ans →

```
function vaccum_world():
        goal_state = {'A': '0', 'B': '0'}
        cost = 0

        Input location_input, status_input
              , status_input_complement

        print "Initial Condition:", goal_state

    If location_input == 'A'   Then
        If status_input == '1'  Then
            goal_state ['A'] = '0'
            cost += 1

        If status_input_complement
                            == '1'
                            Then

            goal_state ['B'] = '0'
            cost += 2

    Else
        If status_input == '1' , Then
            goal_state ['B'] = '0'
            cost += 1

        If status_input_complement
                        == '1'   Then

            goal_state ['A'] = '0'
```

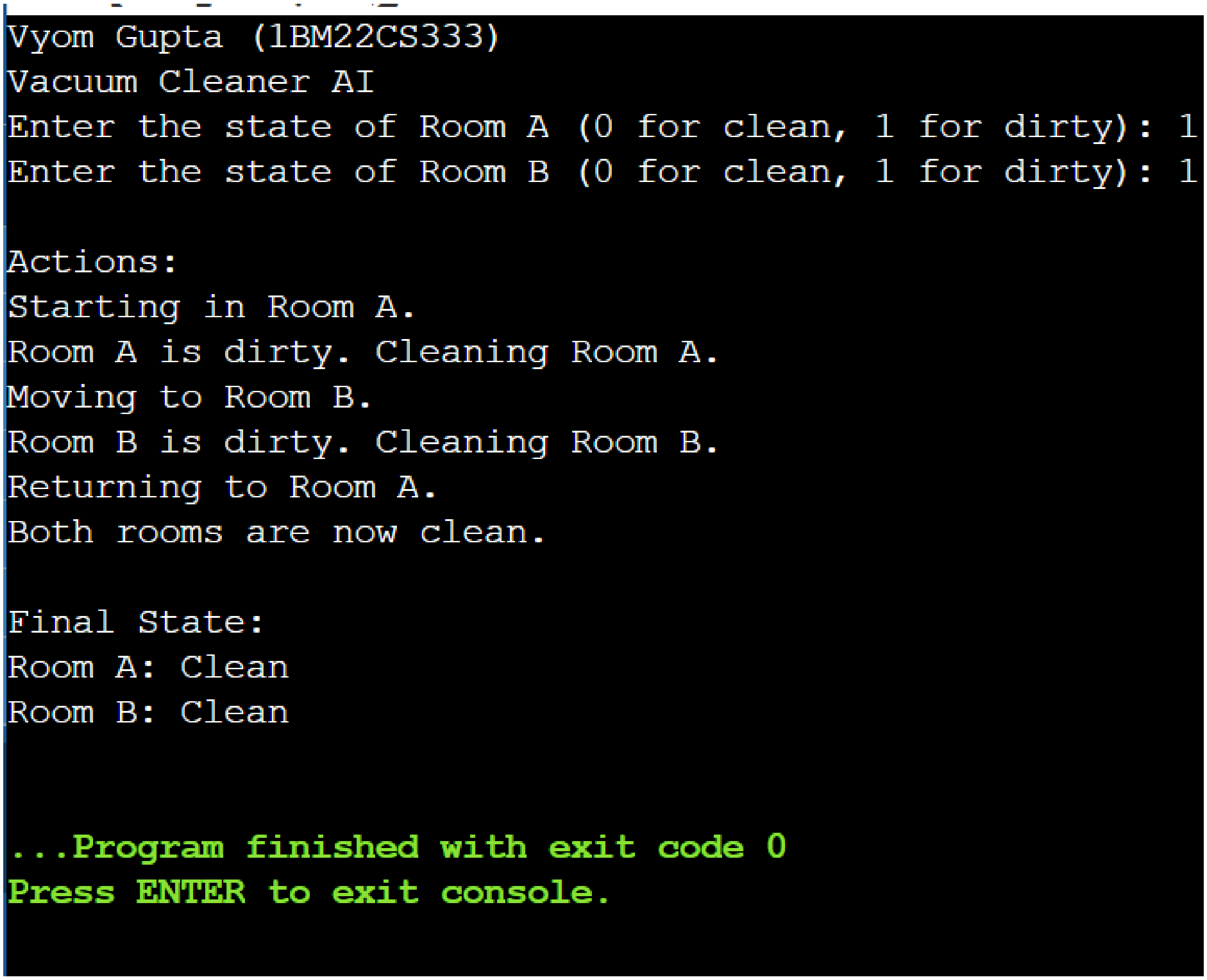
Code

```
print("Vyom Gupta (1BM22CS333)")


def vacuum_cleaner(initial_state):
    # Initial states of rooms A and B
    room_A, room_B = initial_state

    # Trace of actions
    actions = []

    # Start in Room A
    actions.append("Starting in Room A.")

    # Check room A
    if room_A == 1:
        actions.append("Room A is dirty. Cleaning Room A.")
        room_A = 0
    else:
        actions.append("Room A is already clean.")
```

```
# Move to Room B
actions.append("Moving to Room B.")

# Check room B
if room_B == 1:
    actions.append("Room B is dirty. Cleaning Room B.")
    room_B = 0
else:
    actions.append("Room B is already clean.")

# Move back to Room A
actions.append("Returning to Room A.")

# Final state
final_state = (room_A, room_B)
actions.append("Both rooms are now clean.")

return final_state, actions
```

```
def main():
    print("Vacuum Cleaner AI")

    # Input initial states of Room A and Room B
    room_A = int(input("Enter the state of Room A (0 for clean, 1 for dirty): "))
    room_B = int(input("Enter the state of Room B (0 for clean, 1 for dirty): "))

    # Validate input
    if room_A not in (0, 1) or room_B not in (0, 1):
        print("Invalid input. Please enter 0 or 1.")
        return

    # Solve using vacuum cleaner AI
    final_state, actions = vacuum_cleaner((room_A, room_B))

    # Output actions and final state
    print("\nActions:")
    for action in actions:
        print(action)



    print("\nFinal State:")

    print(f"Room A: {'Clean' if final_state[0] == 0 else 'Dirty'}")

    print(f"Room B: {'Clean' if final_state[1] == 0 else 'Dirty'}")



if __name____== "__main__":

    main()
```

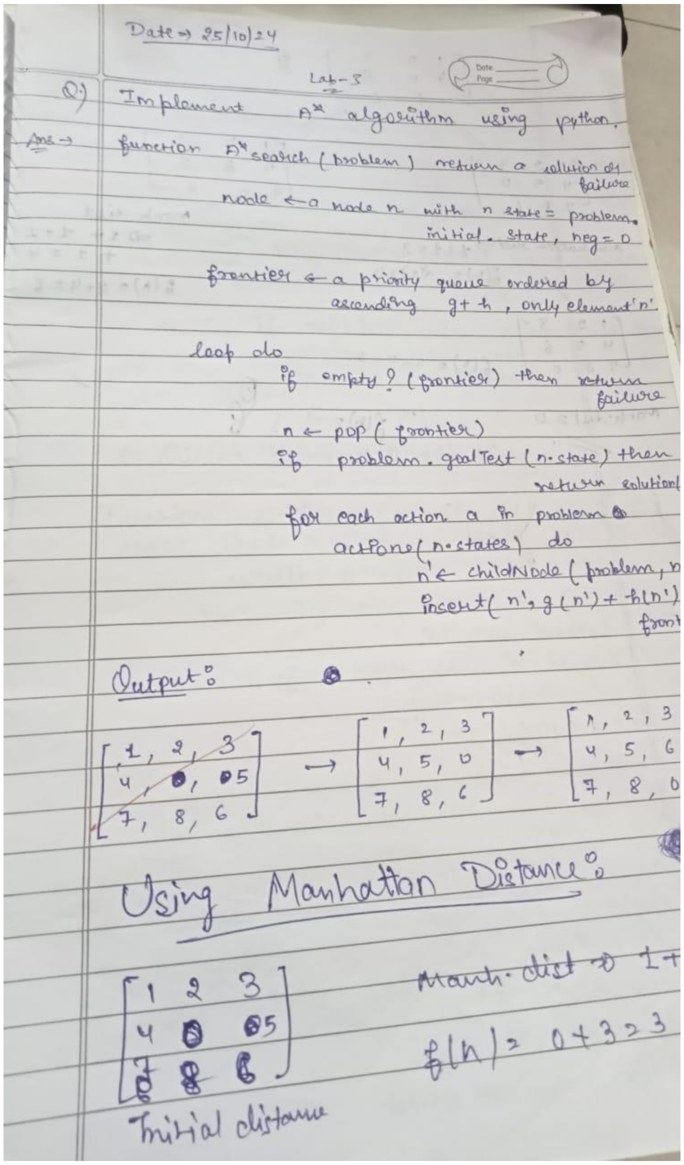
Output



```
Vyom Gupta (1BM22CS333)
Vacuum Cleaner AI
Enter the state of Room A (0 for clean, 1 for dirty): 1
Enter the state of Room B (0 for clean, 1 for dirty): 1

Actions:
Starting in Room A.
Room A is dirty. Cleaning Room A.
Moving to Room B.
Room B is dirty. Cleaning Room B.
Returning to Room A.
Both rooms are now clean.

Final State:
Room A: Clean
Room B: Clean


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 5 - A* Search Algorithm and Hill Climbing Algorithm

Algorithm



Date → 25/10/24

Lab-5

Date ____ Page ____

Q.) Implement A* algorithm using python.

Ans →

```
function A*search (problem) return a solution or
                                          failure
    node ← a node n with n state = problem.
                       initial. state, neg = 0
    frontier ← a priority queue ordered by
             ascending g+h, only element 'n'.
    loop do
        if empty? (frontier) then return
                                  failure
        n ← pop ( frontier)
        if problem. goalTest (n.state) then
                            return solution(
        for each action a in problem
            actions( n.states) do
                n' ← childNode ( problem, n
                insert( n', g(n') + h(n')
                                     front
```

Output:

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

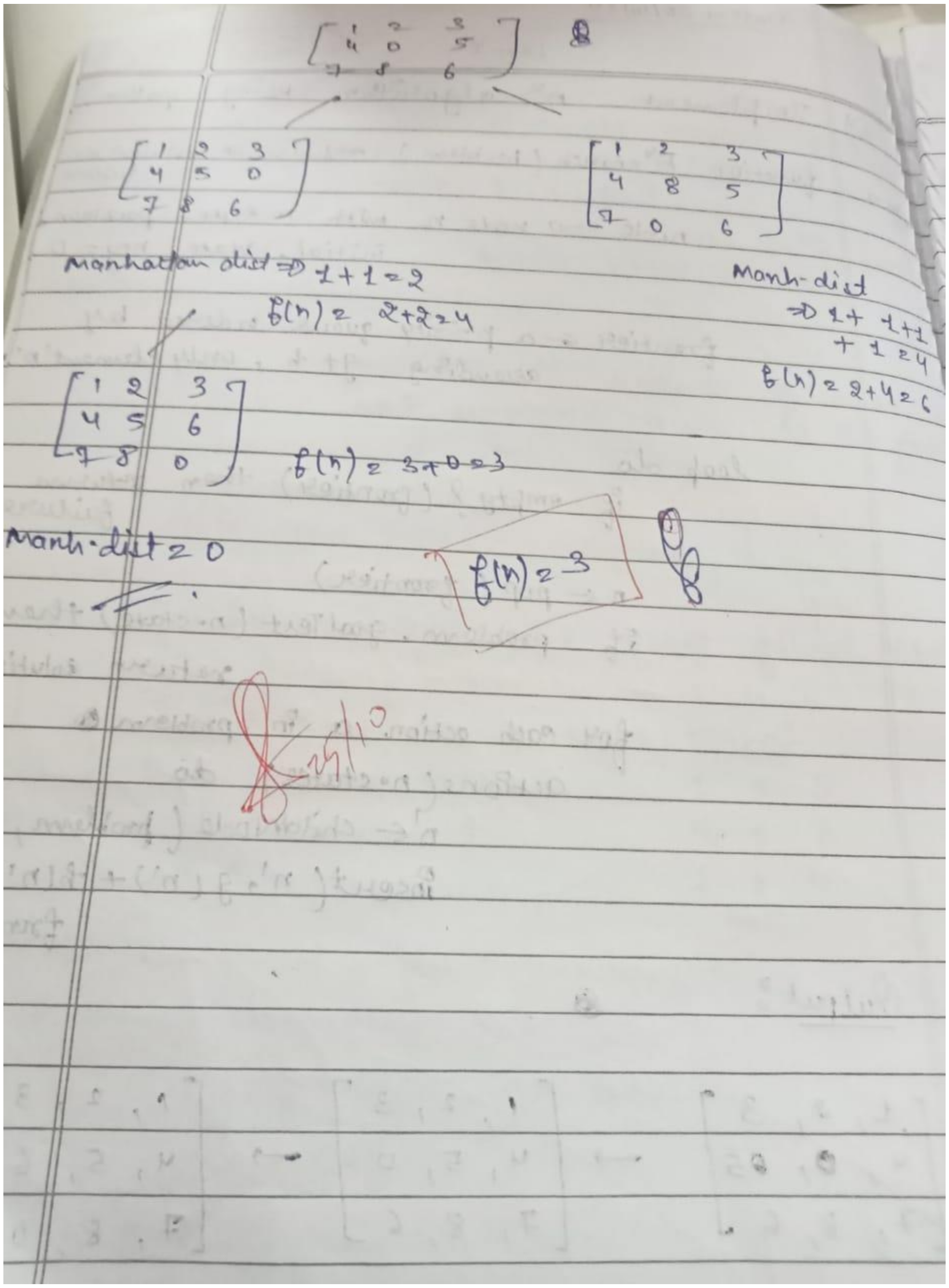
Using Manhattan Distance:

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

Initial distance

~~Manh. dist~~ → 1+

$f(n) = 0 + 3 = 3$



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

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

Manhattan dist ⇒ $1+1=2$

$f(n) = 2+2 = 4$

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

Manh-dist

⇒ $1+1+1+1 = 4$

$f(n) = 2+4 = 6$

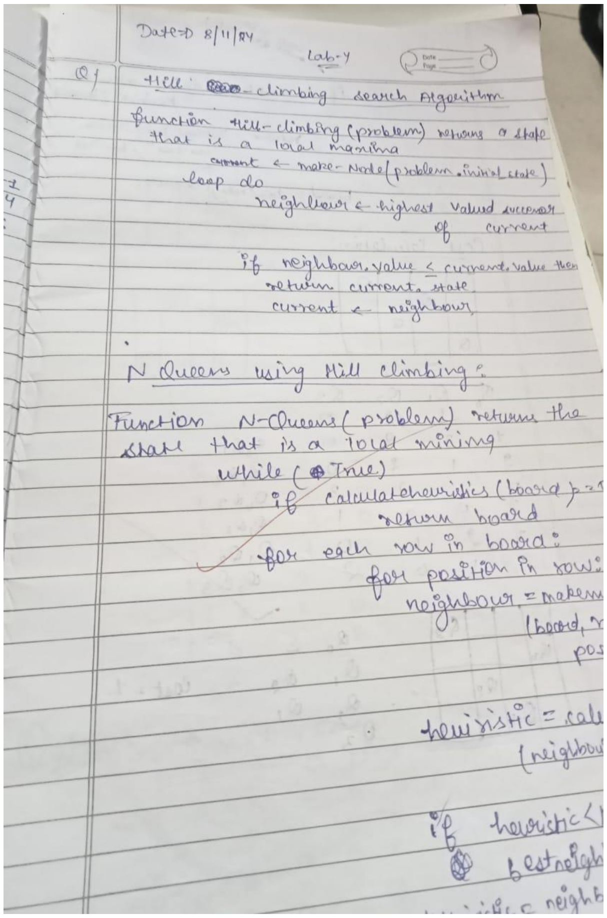
$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 0 \end{bmatrix}$$

$f(n) = 3+0 = 3$

Manh.dist = 0

$\boxed{f(n) = 3}$

24/10



Date=D 8/11/24

Lab-4

Date ____ Page ____

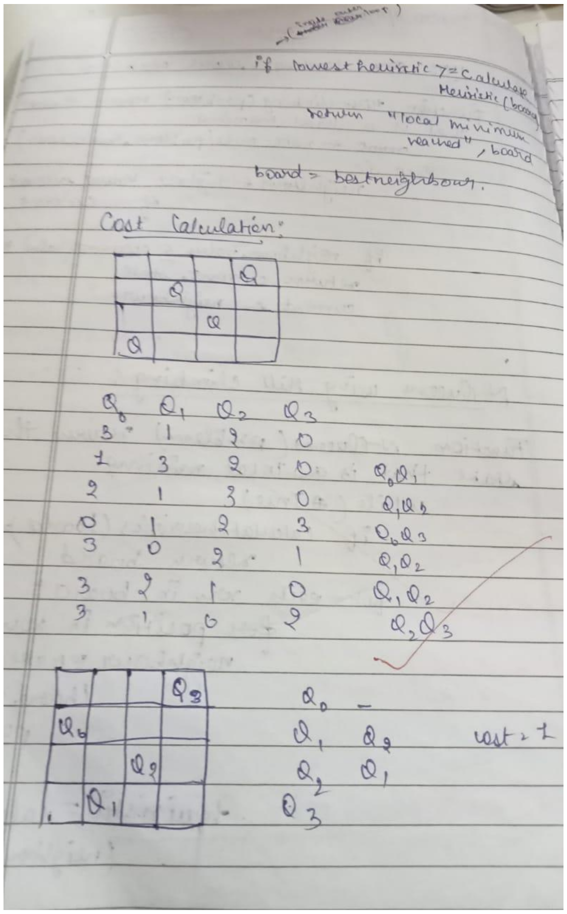
Q1

Hill climbing search Algorithm

```
function Hill-climbing (problem) returns a state
  that is a local maxima
        current ← make-Node(problem.initial state)
    loop do
            neighbour ← highest valued successor
                                      of current

            if neighbour.value ≤ current.value then
                return current.state
                current ← neighbour
```

N Queens using Hill climbing:

```
Function N-Queens( problem) returns the
state that is a local mining
        while (True)
            if calculateheuristics (board) ==
                        return board
            for each row in board:
                    for position in row:
                        neighbour = makem
                                    (board, r
                                        pos

                        heuristic = calc
                                    (neighbou

                        if heuristic<
                            bestneigh
                            ...ic = neighb
```



→(inside outer loop)

```
                if lowest heuristic >= calculate
                                        Heuristic (board)
                    return "local minimum
                                reached", board
                board = bestneighbour.
```

Cost Calculation:

			Q
	Q		
		Q	
Q			

Q_0	Q_1	Q_2	Q_3	
3	1	2	0	
1	3	2	0	Q_0Q_1
2	1	3	0	Q_1Q_2
0	1	2	3	Q_0Q_3
3	0	2	1	Q_1Q_2
3	2	1	0	Q_1Q_2
3	1	0	2	Q_2Q_3

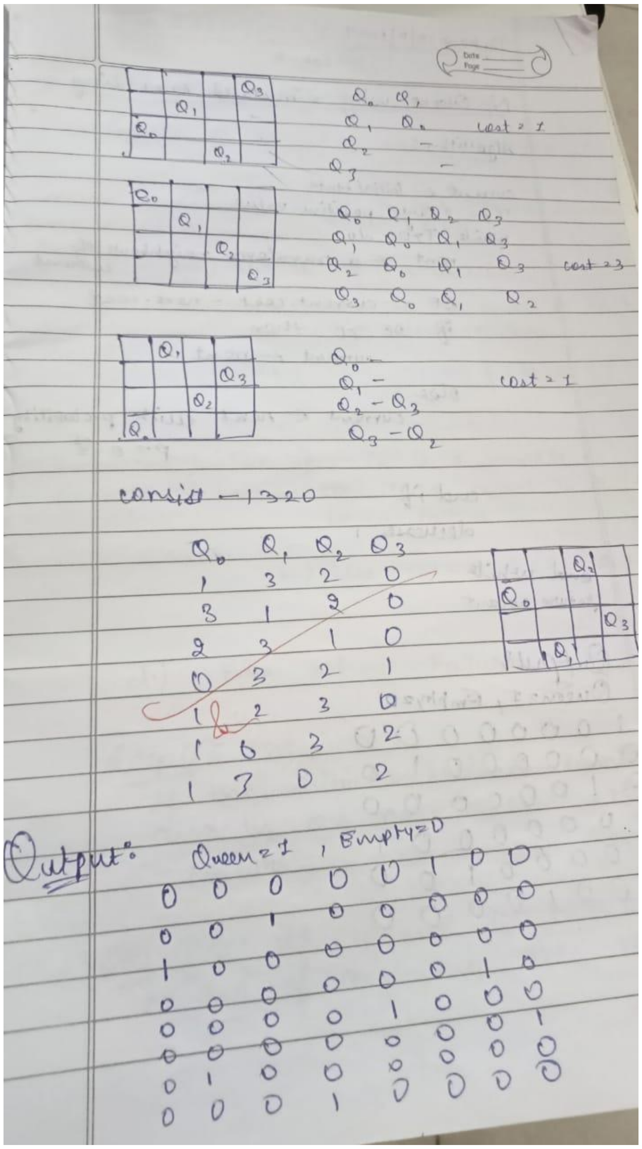
			Q_3
Q_0			
		Q_2	
	Q_1		

Q_0 –

Q_1 Q_2 cost = 1

Q_2 Q_1

Q_3



			Q_3
	Q_1		
Q_0			
		Q_2	

Q_0 Q_1

Q_1 Q_0 cost = 1

Q_2 –

Q_3 –

Q_0			
	Q_1		
		Q_2	
			Q_3

Q_0 Q_1 Q_2 Q_3

Q_1 Q_0 Q_1 Q_3

Q_2 Q_0 Q_1 Q_3 cost = 3

Q_3 Q_0 Q_1 Q_2

	Q_1		
			Q_3
		Q_2	
Q_0			

Q_0 –

Q_1 – cost = 1

Q_2 – Q_3

Q_3 – Q_2

consist – 1320

Q_0	Q_1	Q_2	Q_3
1	3	2	0
3	1	2	0
2	3	1	0
0	3	2	1
1	2	3	0
1	0	3	2
1	3	0	2

		Q_2	
Q_0			
			Q_3
	Q_1		

Output: Queen = 1 , Empty = 0

```
0 0 0 0 0 1 0 0
0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 1
0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0
```

Code (A* algorithm using N – displaced Tiles)

```
import heapq

 # Goal state

goal_state = (

    (1, 2, 3),

    (4, 5, 6),

    (7, 8, 0)

)



# Function to compute the heuristic (misplaced tiles)

def misplaced_tiles(state):

    misplaced = 0

    for i in range(3):

        for j in range(3):

            if state[i][j] != goal_state[i][j] and state[i][j] != 0:

                misplaced += 1

    return misplaced



# Function to get possible moves (neighbors)
```

```
def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    i, j = zero_pos

    # Possible moves: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = list(list(row) for row in state)  # Create a copy of the state
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            neighbors.append(tuple(tuple(row) for row in new_state))  # Convert back to tuple
    return neighbors

# Function to count the number of inversions in the puzzle
def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
    inversions = 0
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return None


    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tiles(initial_state), 0, initial_state, []))  # (f(n), g(n),
state, path)
```

```
closed_list = set()

while open_list:
    f, g, current_state, path = heapq.heappop(open_list)
    closed_list.add(current_state)

    # If goal state is reached
    if current_state == goal_state:
        return path + [current_state]

    # Generate neighbors
    for neighbor in get_neighbors(current_state):
        if neighbor not in closed_list:
            heapq.heappush(open_list, (
                g + 1 + misplaced_tiles(neighbor),  # f(n) = g(n) + h(n)
                g + 1,  # Increment g(n) by 1 for each move
                neighbor,
                path + [current_state]
            ))
    return None  # No solution found


# Function to display the puzzle state
def display_state(state, label):
    print(f"{label} state:")
    for row in state:
        print(" ".join(str(x) for x in row))
    print()


# Example initial state (this one is solvable)
initial_state = (
    (1, 2, 3),
    (5, 6, 4),
    (7, 8, 0)
)


# Solving the puzzle
solution = a_star(initial_state)
```

```
# Displaying the result
if solution:
    # Print Vyom's information
    print("Vyom Gupta (1BM22CS333)\n")

    # Print the initial state
    display_state(initial_state, "Initial")

    # Print the final state
    display_state(goal_state, "Goal")

    # Displaying the solution path
    print("Solution path:")
    for step in solution:
        display_state(step, "Step")
else:
    print("No solution found.")
```

Code (A* algorithm using Manhattan distance)

```
import heapq



# Goal state

goal_state = (

    (1, 2, 3),

    (4, 5, 6),

    (7, 8, 0)

)



# Function to compute the Manhattan distance heuristic

def manhattan_distance(state):

    distance = 0

    for i in range(3):

        for j in range(3):

            tile = state[i][j]

            if tile != 0:

                goal_i, goal_j = divmod(tile - 1, 3)

                distance += abs(goal_i - i) + abs(goal_j - j)
    return distance


# Function to get possible moves (neighbors)
def get_neighbors(state):
    neighbors = []
    zero_pos = [(i, j) for i in range(3) for j in range(3) if state[i][j] == 0][0]
    i, j = zero_pos

    # Possible moves: up, down, left, right
    moves = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for move in moves:
        new_i, new_j = i + move[0], j + move[1]
        if 0 <= new_i < 3 and 0 <= new_j < 3:
            new_state = list(list(row) for row in state)  # Create a copy of the state
            new_state[i][j], new_state[new_i][new_j] = new_state[new_i][new_j], new_state[i][j]
            neighbors.append(tuple(tuple(row) for row in new_state))  # Convert back to tuple
    return neighbors


# Function to count the number of inversions in the puzzle
```

```
def count_inversions(state):
    one_d_state = [tile for row in state for tile in row if tile != 0]
    inversions = 0
    for i in range(len(one_d_state)):
        for j in range(i + 1, len(one_d_state)):
            if one_d_state[i] > one_d_state[j]:
                inversions += 1
    return inversions


# Check if the puzzle is solvable
def is_solvable(state):
    inversions = count_inversions(state)
    return inversions % 2 == 0


# A* Algorithm
def a_star(initial_state):
    if not is_solvable(initial_state):
        print("This puzzle is not solvable.")
        return None
    open_list = []
    heapq.heappush(open_list, (0 + manhattan_distance(initial_state), 0, initial_state, []))  # (f(n), g(n), state, path)
    closed_list = set()

    while open_list:
        f, g, current_state, path = heapq.heappop(open_list)
        closed_list.add(current_state)

        # Print the current state and its f(n) value
        print(f"State: {current_state}")
        print(f"f(n) = g(n) + h(n) = {g} + {manhattan_distance(current_state)} = {f}")
        print()

        # If goal state is reached
        if current_state == goal_state:
            return path + [current_state]

        # Generate neighbors
        for neighbor in get_neighbors(current_state):
            if neighbor not in closed_list:
                heapq.heappush(open_list, (
                    g + 1 + manhattan_distance(neighbor),  # f(n) = g(n) + h(n)
                    g + 1,  # Increment g(n) by 1 for each move
                    neighbor,
                    path + [current_state]
                ))

    return None  # No solution found


# Function to display the puzzle state
def display_state(state, label):
    print(f"{label} state:")
    for row in state:
        print(" ".join(str(x) for x in row))
    print()


# Example initial state (this one is solvable)
```

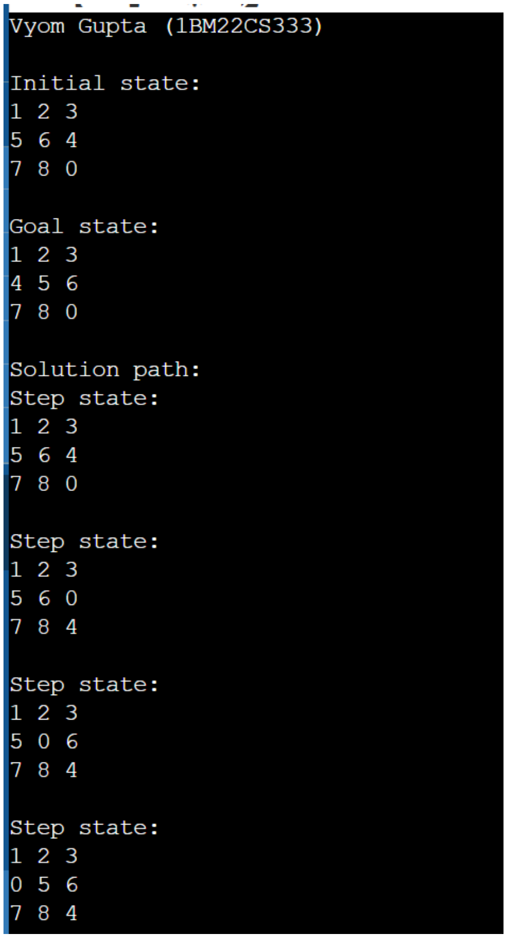
```
initial_state = (

    (1, 2, 3),

    (5, 6, 4),

    (7, 8, 0)

)



# Solving the puzzle

solution = a_star(initial_state)



# Displaying the result

if solution:

    print("Vyom Gupta(1BM22CS333)\n")



    # Print the initial state

    display_state(initial_state, "Initial")



    # Print the final state

    display_state(goal_state, "Goal")
    # Displaying the solution path

    print("Solution path:")

    for step in solution:

        display_state(step, "Step")

else:

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

Code (Hill Climbing algorithm)

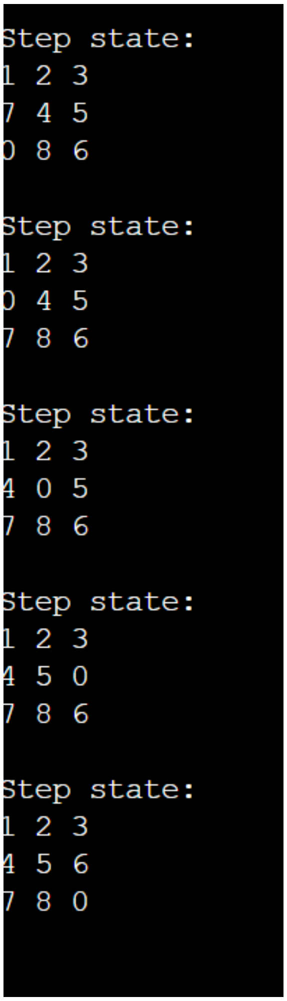
```
import random

print("Vyom Gupta (1BM22CS333)")

# Function to calculate the number of attacking pairs of queens

def calculate_attacks(board):

    attacks = 0

    n = len(board)

    for i in range(n):

        for j in range(i + 1, n):

            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                attacks += 1
    return attacks


# Function to generate a random initial state
def generate_initial_state(n):
    return [random.randint(0, n - 1) for _ in range(n)]


# Function to generate neighbors by moving one queen to a different row
def generate_neighbors(board):
    neighbors = []
    n = len(board)
    for col in range(n):
        for row in range(n):
            if row != board[col]:  # Make sure we are not moving the queen to its current row
                neighbor = board[:]
                neighbor[col] = row
                neighbors.append(neighbor)
    return neighbors
```

```
# Hill Climbing algorithm with random restarts
def hill_climbing(n, max_restarts=100):
    for restart in range(max_restarts):
        current_state = generate_initial_state(n)
        current_attacks = calculate_attacks(current_state)

        while True:
            # Generate all neighbors
            neighbors = generate_neighbors(current_state)

            # Find the neighbor with the minimum number of attacks
            next_state = None
            next_attacks = current_attacks

            for neighbor in neighbors:
                attacks = calculate_attacks(neighbor)
                if attacks < next_attacks:
                    next_state = neighbor
                    next_attacks = attacks
            # If no improvement, return the solution or terminate
            if next_attacks == current_attacks:
                break

            current_state = next_state
            current_attacks = next_attacks

        # If a solution is found, return the current state
        if current_attacks == 0:
            return current_state

    # If no solution found after max_restarts, return None
    return None

# Function to display the board
def display_board(board):
    n = len(board)
    for i in range(n):
        row = ['Q' if i == board[col] else '.' for col in range(n)]

        print(' '.join(row))

    print()



# Set the size of the board (N)

N = 8



# Solve the N-Queens problem with random restarts

solution = hill_climbing(N)



# Display the result

if solution:

    print(f"Solution for {N}-Queens:")

    display_board(solution)

else:

    print(f"No solution found for {N}-Queens.")
```

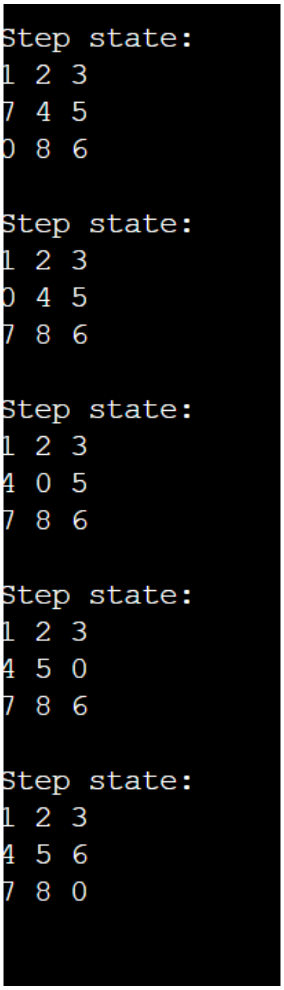
Output (N-displaced Tiles)



```
Vyom Gupta (1BM22CS333)

Initial state:
1 2 3
5 6 4
7 8 0

Goal state:
1 2 3
4 5 6
7 8 0

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

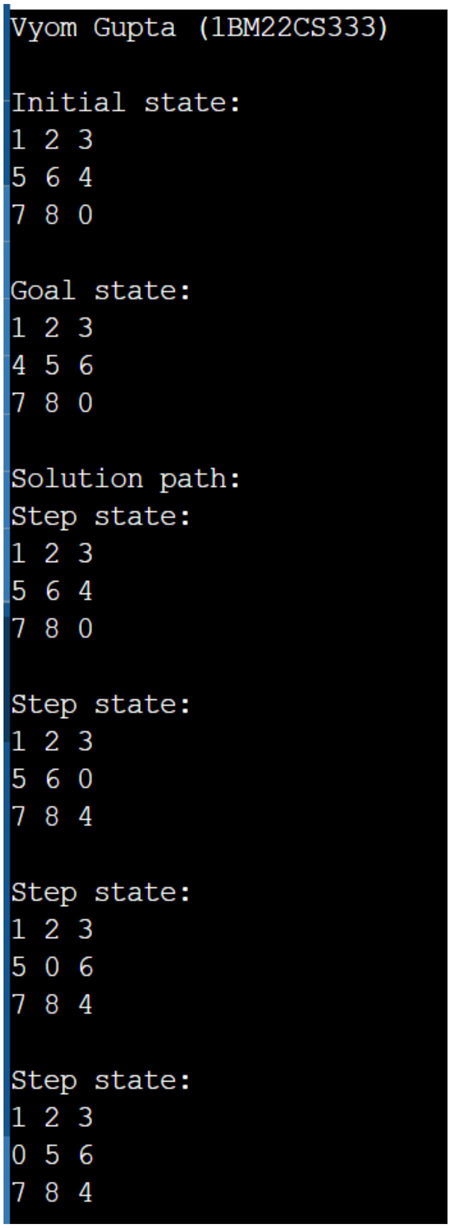


```
Step state:
1 2 3
7 4 5
0 8 6

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

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

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

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

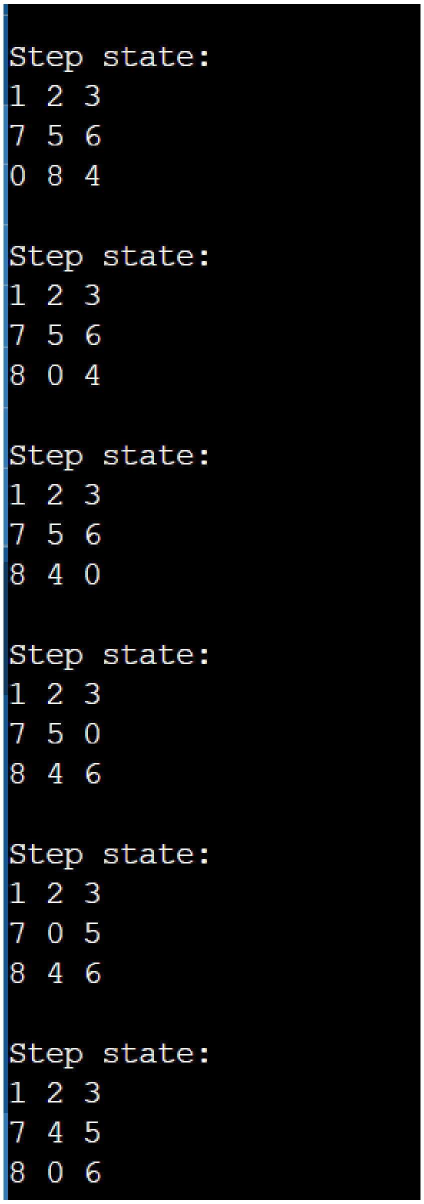


```
Step state:
1 2 3
7 4 5
0 8 6

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

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

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

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

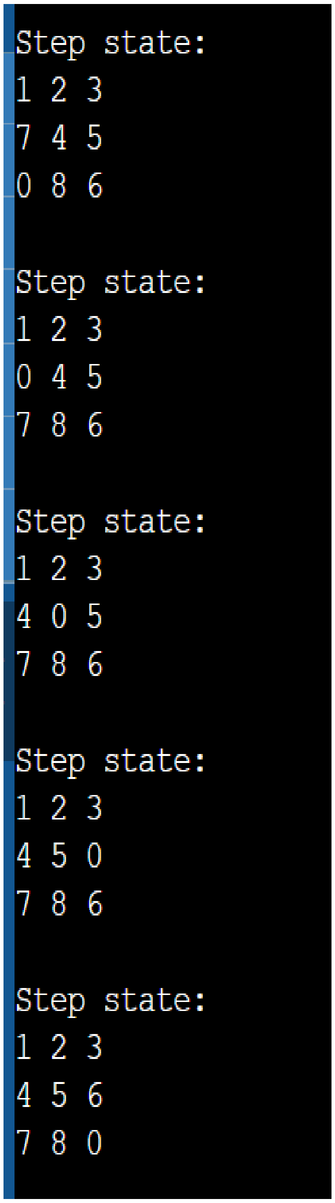
Output (Manhattan Distance)



```
Vyom Gupta (1BM22CS333)

Initial state:
1 2 3
5 6 4
7 8 0

Goal state:
1 2 3
4 5 6
7 8 0

Solution path:
Step state:
1 2 3
5 6 4
7 8 0

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

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

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

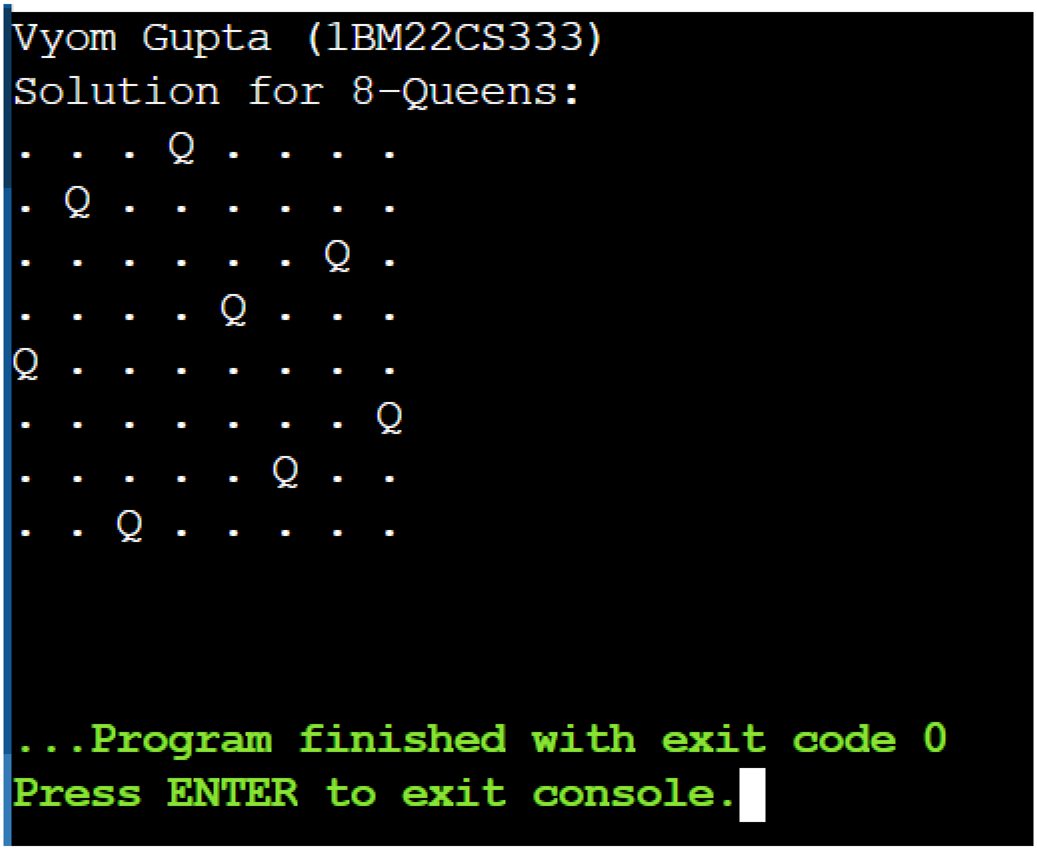


```
Step state:
1 2 3
7 5 6
0 8 4

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

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

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

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

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



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

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

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

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

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

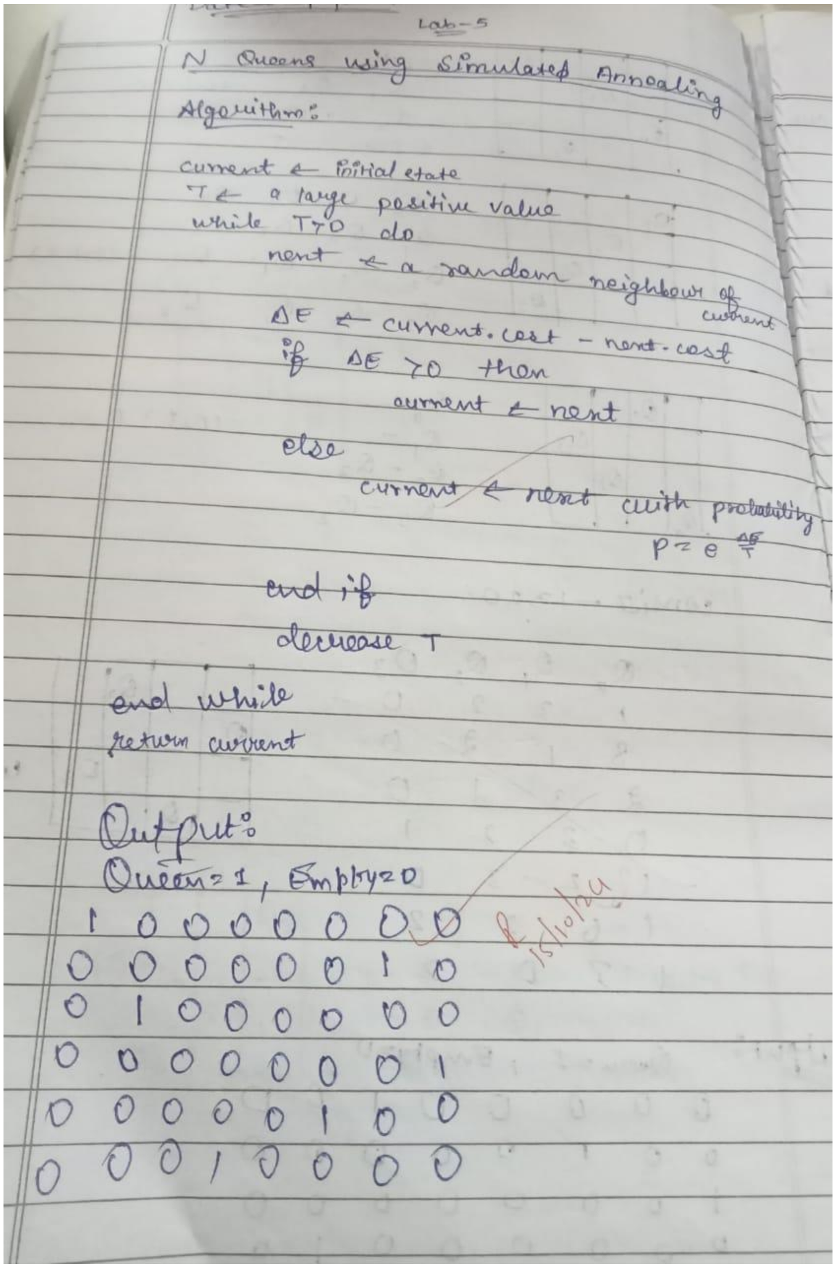
Output (Hill Climbing)



```
Vyom Gupta (1BM22CS333)
Solution for 8-Queens:
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . . . Q . . .
Q . . . . . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .



...Program finished with exit code 0
Press ENTER to exit console.
```

Program 6 - Simulated Annealing

Algorithm



Lab-5

N Queens using Simulated Annealing

Algorithm:

```
current ← initial state
T ← a large positive value
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
                                        $p = e^{\frac{\Delta E}{T}}$
        end if
        decrease T
end while
return current
```

Output:

Queen=1, Empty=0

```
1 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
0 0 0 1 0 0 0 0
```

15/10/24

Code

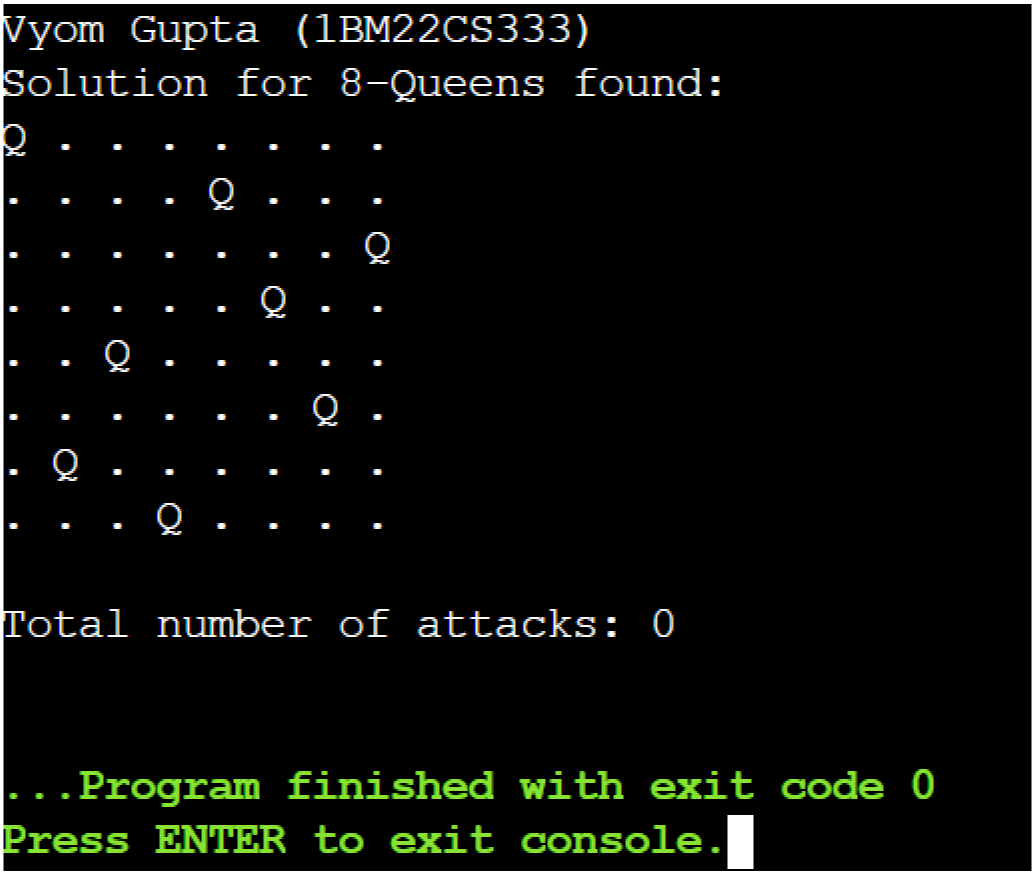
```
import random

import math

print("Vyom Gupta (1BM22CS333)")

# Objective function: count the number of attacking pairs of queens
```

```
def calculate_attacks(board):
attacks = 0
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            # Check if two queens are in the same row, column, or diagonal
            if board[i] == board[j] or abs(board[i] - board[j]) == j - i:
                attacks += 1
    return attacks


# Function to generate a random initial state (random queen positions in each column)
def generate_initial_state(n):
    return [random.randint(0, n - 1) for _ in range(n)]


# Function to generate a neighboring solution by moving one queen in a column
def generate_neighbor(board):
    neighbor = board[:]
    column = random.randint(0, len(board) - 1)
    # Randomly select a new row for the queen in the chosen column
    neighbor[column] = random.randint(0, len(board) - 1)
    return neighbor


# Simulated Annealing algorithm to solve the N-Queens problem
def simulated_annealing(n, max_iterations, initial_temperature, cooling_rate):
    current_state = generate_initial_state(n)
    current_attacks = calculate_attacks(current_state)
    temperature = initial_temperature

    best_state = current_state
    best_attacks = current_attacks

    for iteration in range(max_iterations):
        # Generate a neighbor solution
        neighbor = generate_neighbor(current_state)
        neighbor_attacks = calculate_attacks(neighbor)

        # Calculate the energy difference (how much worse the new state is)
        delta_attacks = neighbor_attacks - current_attacks
        # Accept the neighbor if it has fewer attacks or with a probability if it's worse
        if delta_attacks < 0 or random.random() < math.exp(-delta_attacks / temperature):
            current_state = neighbor
            current_attacks = neighbor_attacks

        # Update the best solution if necessary
        if current_attacks < best_attacks:
            best_state = current_state
            best_attacks = current_attacks

        # Cool down the temperature
        temperature *= cooling_rate

        # If no attacks, we found the solution
        if best_attacks == 0:
            break

    return best_state, best_attacks
```

```
# Function to display the board (where 'Q' is a queen and '.' is an empty space)

def display_board(board):

    n = len(board)

    for i in range(n):

        row = ['Q' if i == board[col] else '.' for col in range(n)]

        print(' '.join(row))

    print()


# Parameters for Simulated Annealing

N = 8  # Set the size of the board (N x N)

max_iterations = 10000  # Higher number of iterations for better convergence

initial_temperature = 1000  # High initial temperature

cooling_rate = 0.995  # Cooling rate (temperature decreases by 0.5% every iteration)


# Solve the N-Queens problem using Simulated Annealing

solution, attacks = simulated_annealing(N, max_iterations, initial_temperature, cooling_rate)


# Output the result
```

```
print(f"Solution for {N}-Queens found:")

display_board(solution)

print(f"Total number of attacks: {attacks}")
```

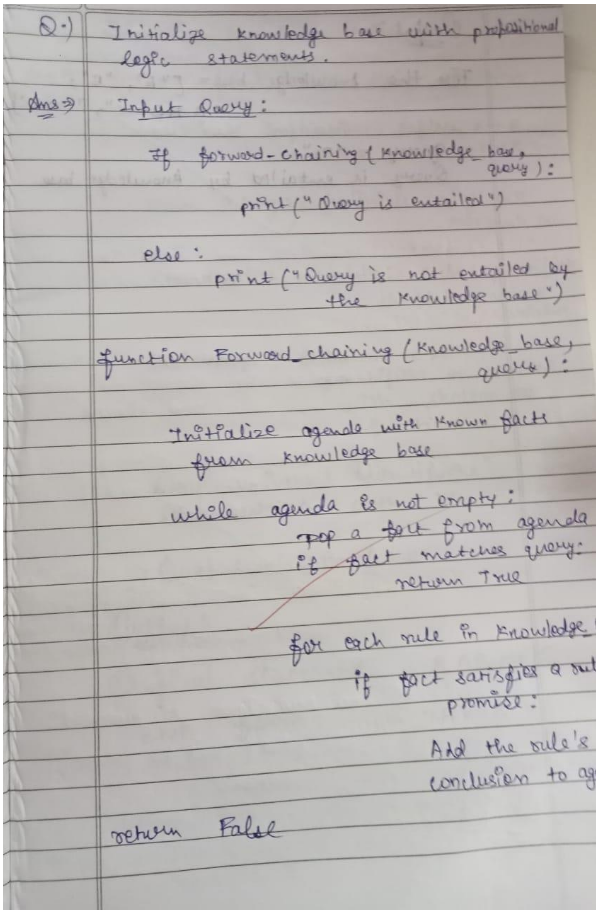
Output



```
Vyom Gupta (1BM22CS333)
Solution for 8-Queens found:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Total number of attacks: 0


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 7 - Knowledge base using prepositional logic and show that the given query entails the knowledge base or not.

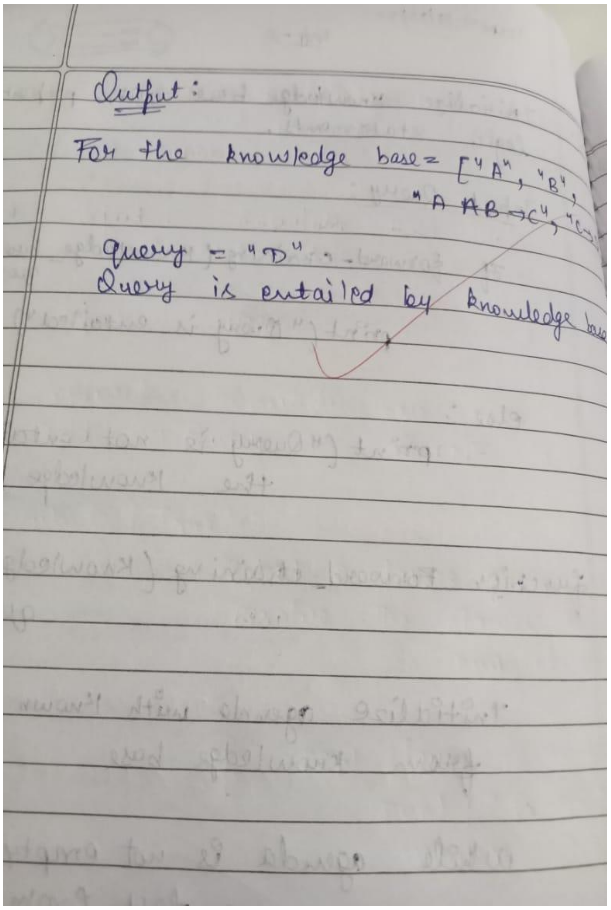
Algorithm



Q.) Initialize knowledge base with propositional logic statements.

Ans ⇒

```
Input Query:

    If forward-chaining (knowledge_base,
                                 query):
            print ("Query is entailed")

    else:
            print ("Query is not entailed by
                       the knowledge base")

function Forward_chaining (knowledge_base,
                                 query):

    Initialize agenda with known facts
      from knowledge base

    while agenda is not empty:
            Pop a fact from agenda
            if fact matches query:
                    return True

            for each rule in knowledge_
                    if fact satisfies a rul
                            promise:
                            Add the rule's
                            conclusion to ag

return False
```



Output:

For the knowledge base = ["A", "B",
"A ~~AB~~ →C", "C→

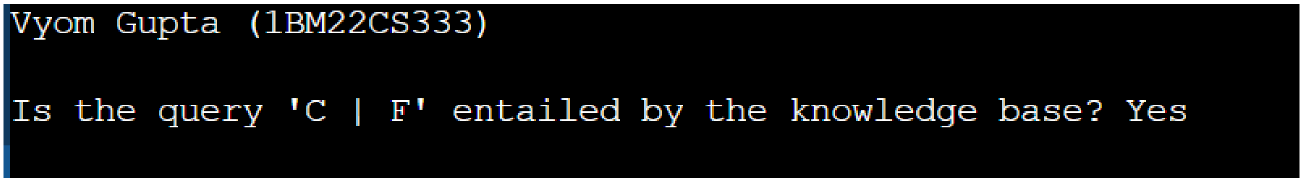
query = "D".
Query is entailed by knowledge base

Code

```
from sympy.logic.boolalg import Or, And, Not
from sympy.abc import A, B, C, D, E, F
from sympy import simplify_logic


def is_entailment(kb, query):
    # Negate the query
    negated_query = Not(query)

    # Combine the knowledge base with the negated query
    kb_with_negated_query = And(*kb, negated_query)  # Combine all KB clauses and the negated query

    # Simplify the combined KB to CNF (Conjunctive Normal Form)
    simplified_kb = simplify_logic(kb_with_negated_query, form="cnf")

    # If the simplified KB evaluates to False, the query is entailed
    return simplified_kb == False

# Define a larger Knowledge Base (kb)
kb = [
    Or(A, B),         # A ∨ B
    Or(Not(A), C),   # ¬A ∨ C
    Or(Not(B), D),   # ¬B ∨ D
    Or(Not(D), E),   # ¬D ∨ E
    Or(Not(E), F),   # ¬E ∨ F
]

# Query to check (C ∨ F)
query = Or(C, F)

# Check entailment
```

```
result = is_entailment(kb, query)


# Output the result with Vyom Gupta's details
print(f"Vyom Gupta (1BM22CS333)\n")
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

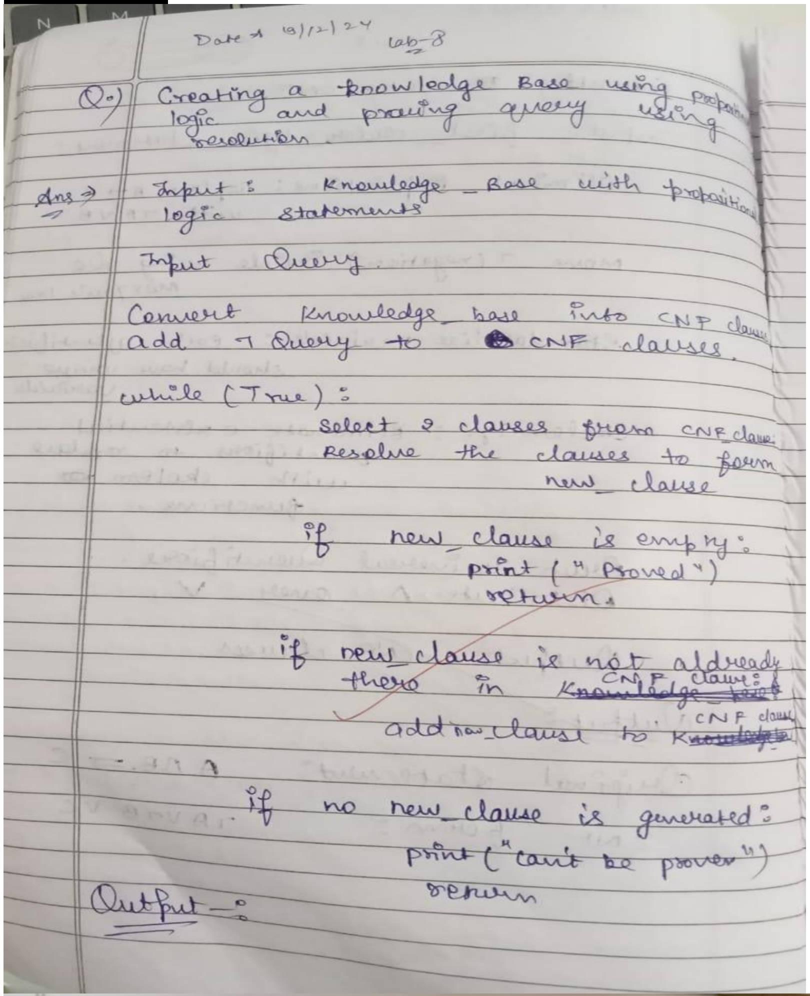
OUTPUT:



```
Vyom Gupta (1BM22CS333)

Is the query 'C | F' entailed by the knowledge base? Yes
```

Program 8 - Knowledge base using prepositional logic and prove the given query using resolution.

Algorithm



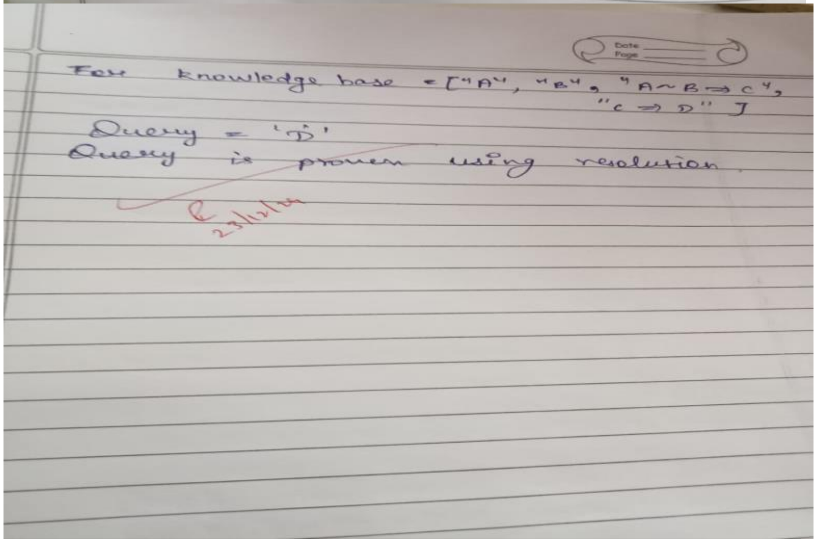
Date → 19/12/24 lab-8

Q.) Creating a knowledge Base using propositional logic and proving query using resolution.

Ans → Input : Knowledge_Base with propositional logic statements

```
Input Query.

Convert Knowledge_base into CNF clause
add ¬ Query to CNF clauses.

while (True) :
            Select 2 clauses from CNF_clause:
            Resolve the clauses to form
                        new_clause

        if new_clause is empty:
                print ("Proved")
                return.

        if new_clause is not aldready
            there in ~~Knowledge_base~~ CNF clause:
            add new_clause to ~~Knowledge~~ CNF clause

      if no new_clause is generated:
                print ("can't be proven")
                return
```

Output - :



For knowledge base = ["A", "B", "A~B → C",
"C → D"]

Query = 'D'
Query is proven using resolution.

23/12/24

Code

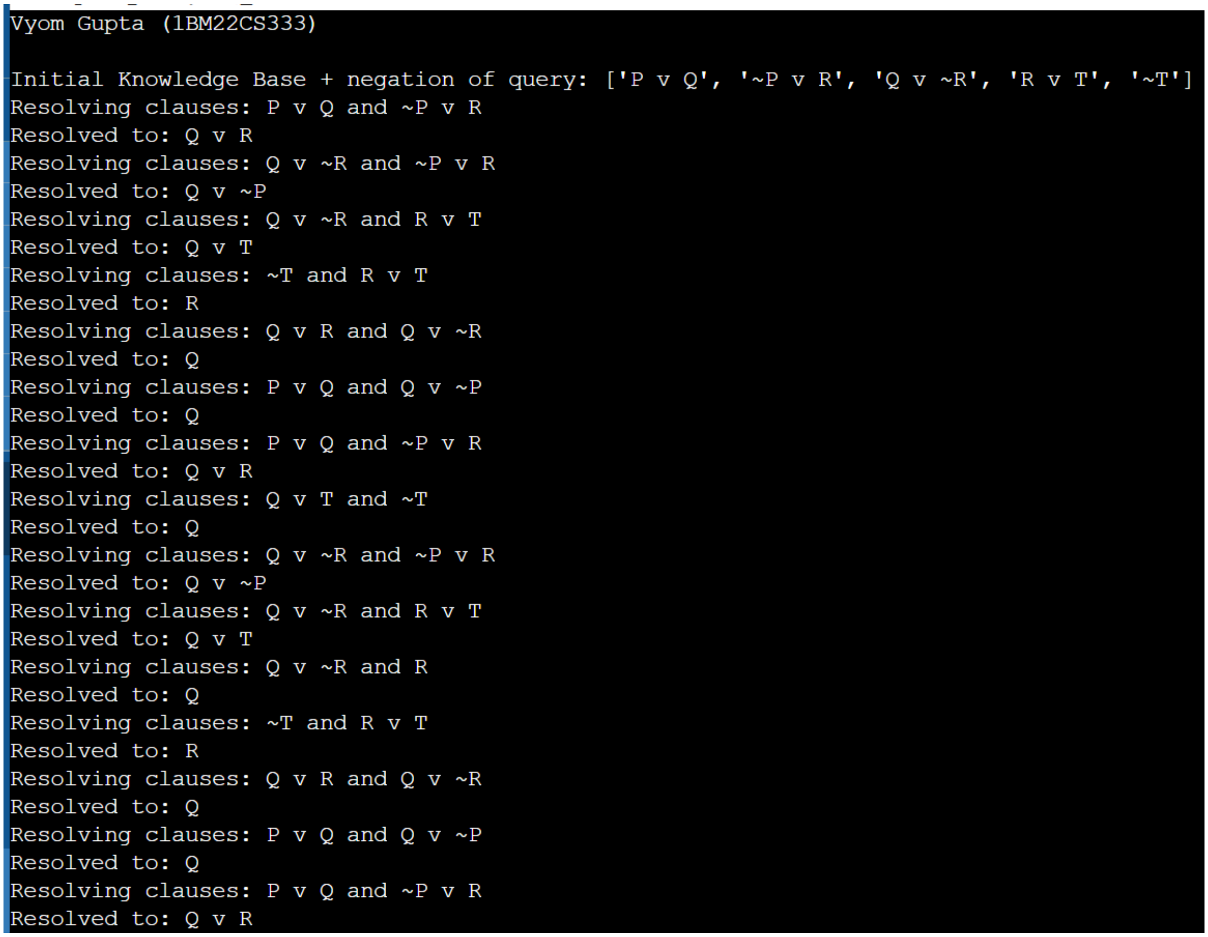
```
def negation(p):

    """Negate a literal."""

    if p.startswith("~"):

        return p[1:]  # remove the '~' from negated literals

    return f"~{p}"



def resolution(kb, query):

    """Perform resolution on the knowledge base to prove the query."""



    # Add the negation of the query to the knowledge base (for proof by contradiction)

    kb.append(negation(query))



    # Apply the resolution rule until we reach an empty clause (which means contradiction)

    new_clauses = set(kb)  # Keep track of all unique clauses in the knowledge base

    print(f"Initial Knowledge Base + negation of query: {kb}")



    while True:

        added_new_clause = False
```

```
# Try to resolve every pair of clauses
clauses = list(new_clauses)
for i in range(len(clauses)):
    for j in range(i + 1, len(clauses)):
        clause1 = clauses[i]
        clause2 = clauses[j]

        # Try to resolve these two clauses
        resolvent = resolve(clause1, clause2)

        if resolvent is not None:
            print(f"Resolving clauses: {clause1} and {clause2}")
            print(f"Resolved to: {resolvent}")

            # If resolvent is empty, we found a contradiction
            if not resolvent:
                return True  # Found a contradiction, so the query is provable

            # Add the new clause if it's not already in the set
```

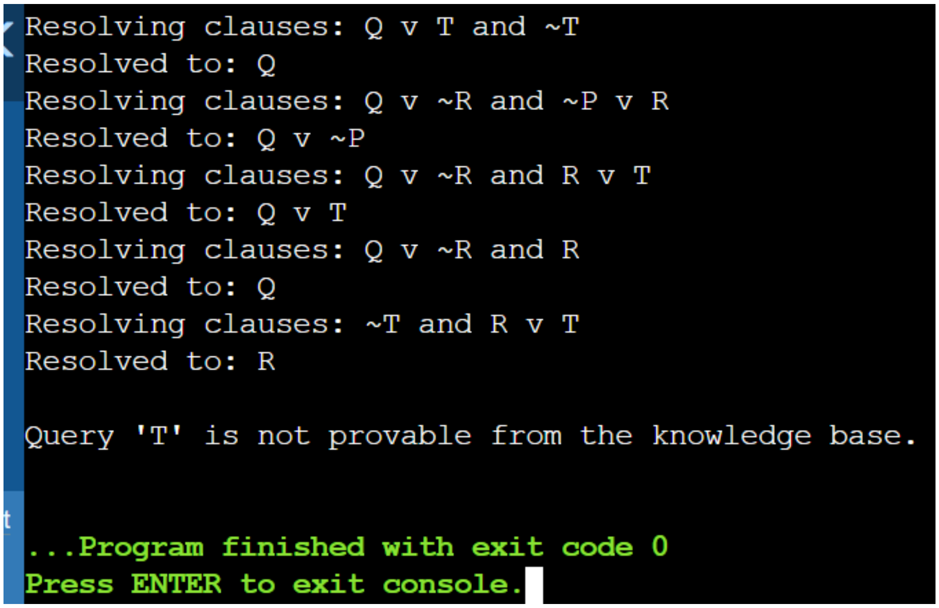
```
                    if resolvent not in new_clauses:
                        new_clauses.add(resolvent)
                        added_new_clause = True

        # If no new clause was added, resolution has terminated without a contradiction
        if not added_new_clause:
            break

    return False  # No contradiction found, so the query is not provable


def resolve(clause1, clause2):
    """Resolve two clauses if possible and return the resolvent."""
    # Split clauses into literals
    literals1 = set(clause1.split(" v "))
    literals2 = set(clause2.split(" v "))

    # Try to find complementary literals
    for literal in literals1:
        neg_literal = negation(literal)
        if neg_literal in literals2:
            # Resolve the two clauses by removing complementary literals
            new_clause = literals1.union(literals2) - {literal, neg_literal}
            return " v ".join(sorted(new_clause))  # Return the resolved clause as a string
    return None  # No resolvent found


# Example knowledge base and query (where T is provable)
kb = [
    "P v Q",           # P or Q
    "~P v R",          # Not P or R
    "Q v ~R",          # Q or Not R
    "R v T"            # R or T
]

query = "T"  # Query to prove (e.g., prove T)

# Perform resolution to prove the query
```

```
result = resolution(kb, query)


if result:
    print(f"\nQuery '{query}' is provable from the knowledge base.")
else:
    print(f"\nQuery '{query}' is not provable from the knowledge base.")
```

Output



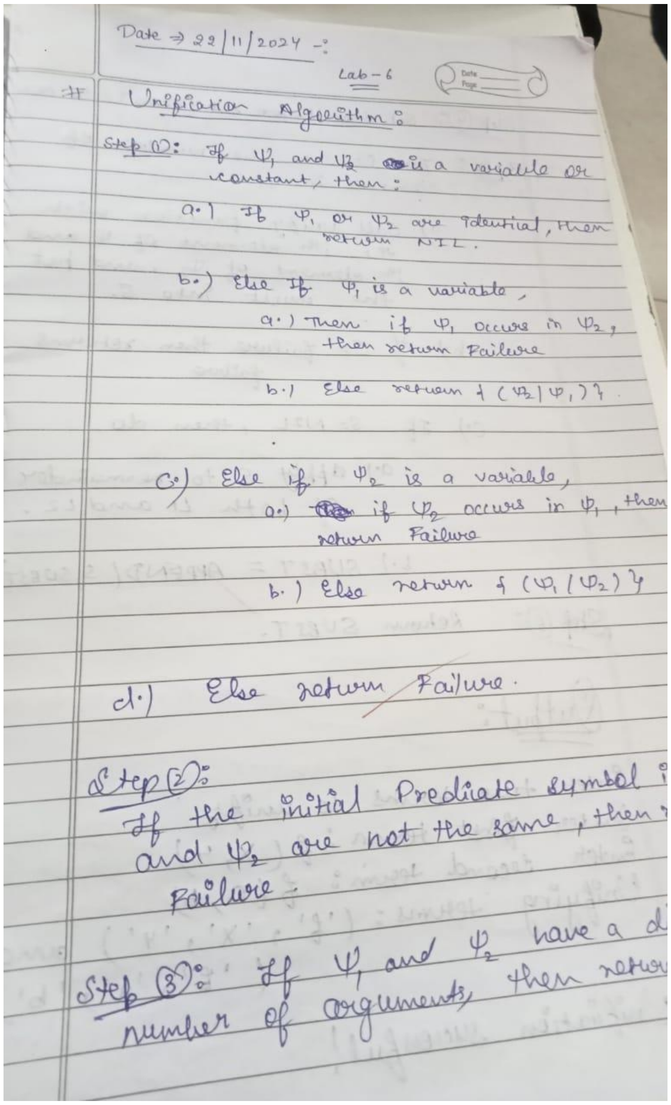
```
Vyom Gupta (1BM22CS333)

Initial Knowledge Base + negation of query: ['P v Q', '~P v R', 'Q v ~R', 'R v T', '~T']
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: ~T and R v T
Resolved to: R
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: Q v ~R and R
Resolved to: Q
Resolving clauses: ~T and R v T
Resolved to: R
Resolving clauses: Q v R and Q v ~R
Resolved to: Q
Resolving clauses: P v Q and Q v ~P
Resolved to: Q
Resolving clauses: P v Q and ~P v R
Resolved to: Q v R
```



```
Resolving clauses: Q v T and ~T
Resolved to: Q
Resolving clauses: Q v ~R and ~P v R
Resolved to: Q v ~P
Resolving clauses: Q v ~R and R v T
Resolved to: Q v T
Resolving clauses: Q v ~R and R
Resolved to: Q
Resolving clauses: ~T and R v T
Resolved to: R

Query 'T' is not provable from the knowledge base.


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 9 – Unification in First Order Logic.

Algorithm



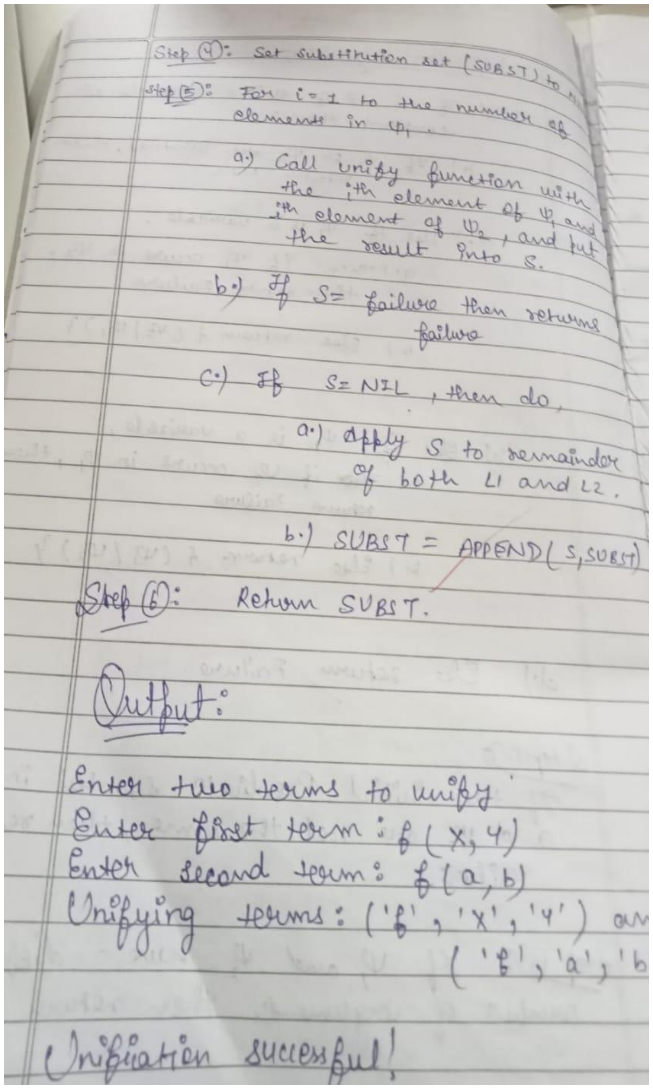
Date ⇒ 22/11/2024 -:

Lab – 6

Date
Page

Unification Algorithm:

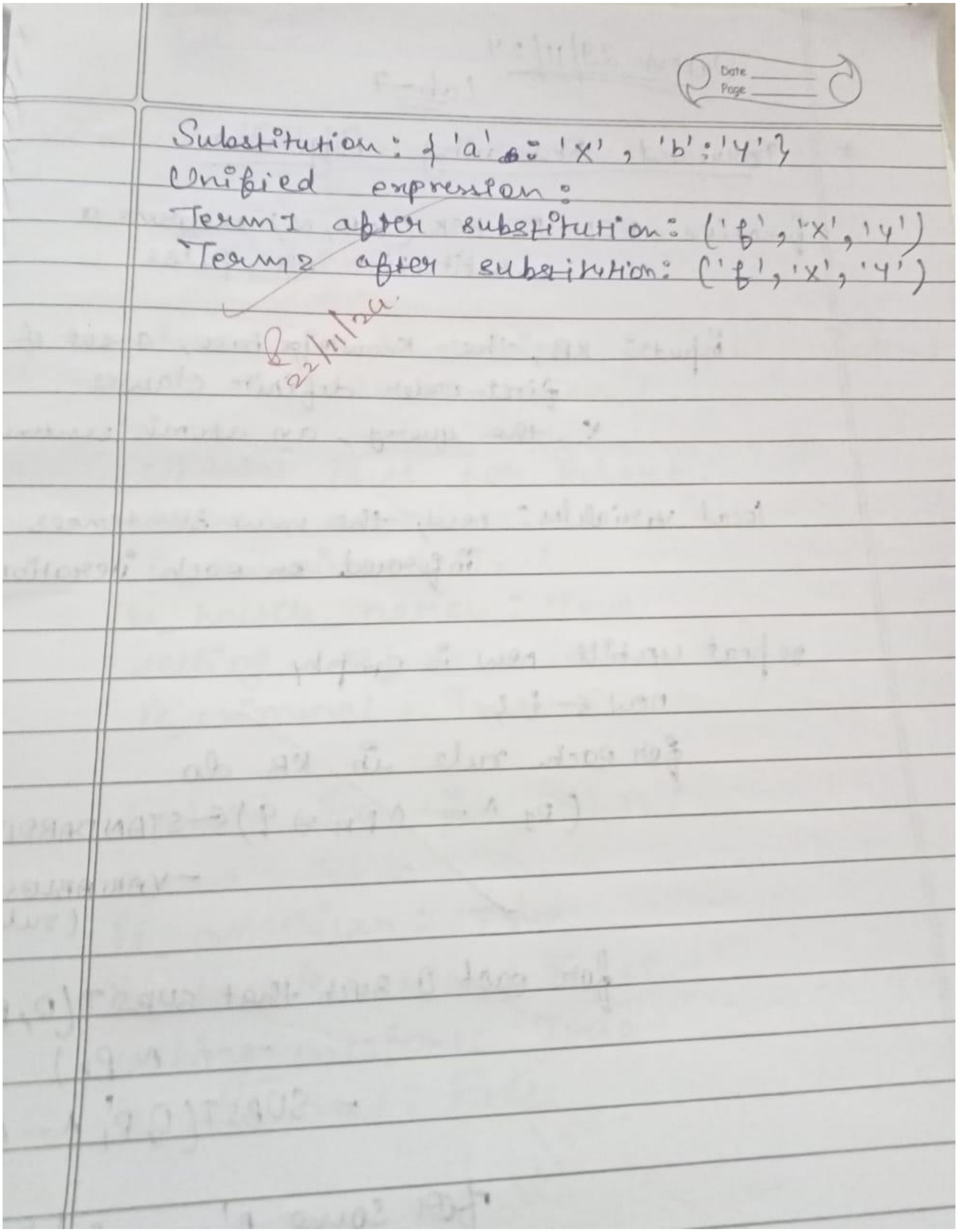
```
Step ①: If $\Psi_1$ and $\Psi_2$ is a variable or
        constant, then:
        a.) If $\Psi_1$ or $\Psi_2$ are Identical, then
                return NIL.
        b.) Else If $\Psi_1$ is a variable,
                a.) Then if $\Psi_1$ occurs in $\Psi_2$,
                        then return Failure
                b.) Else return { ($\Psi_2$ / $\Psi_1$) }.
        C.) Else if $\Psi_2$ is a variable,
                a.) if $\Psi_2$ occurs in $\Psi_1$, then
                        return Failure
                b.) Else return { ($\Psi_1$ / $\Psi_2$) }
    d.) Else return Failure.

Step ②:
    If the initial Prediate symbol i
    and $\Psi_2$ are not the same, then
        Failure.

Step ③: If $\Psi_1$ and $\Psi_2$ have a d
    number of arguments, then retur
```



```
Step ④: Set substitution set (SUBST) to
Step ⑤: For i = 1 to the number of
        elements in $\Psi_1$.
        a.) Call unify function with
            the ith element of $\Psi_1$ and
            ith element of $\Psi_2$, and put
            the result into S.
        b.) If S = failure then returns
                failure
        C.) If S ≠ NIL, then do,
                a.) Apply S to remainder
                    of both L1 and L2.
                b.) SUBST = APPEND(S, SUBST)
Step ⑥: Return SUBST.
```

Output:

```
Enter two terms to unify
Enter first term: f(X, Y)
Enter second term: f(a, b)
Unifying terms: ('f', 'X', 'Y') an
                ('f', 'a', 'b

Unification successful!
```



Substitution: {'a': 'x', 'b':'y'}
Unified expression:
Term1 after substitution: ('f', 'x', 'y')
Term2 after subsitution: ('f', 'x', 'y')

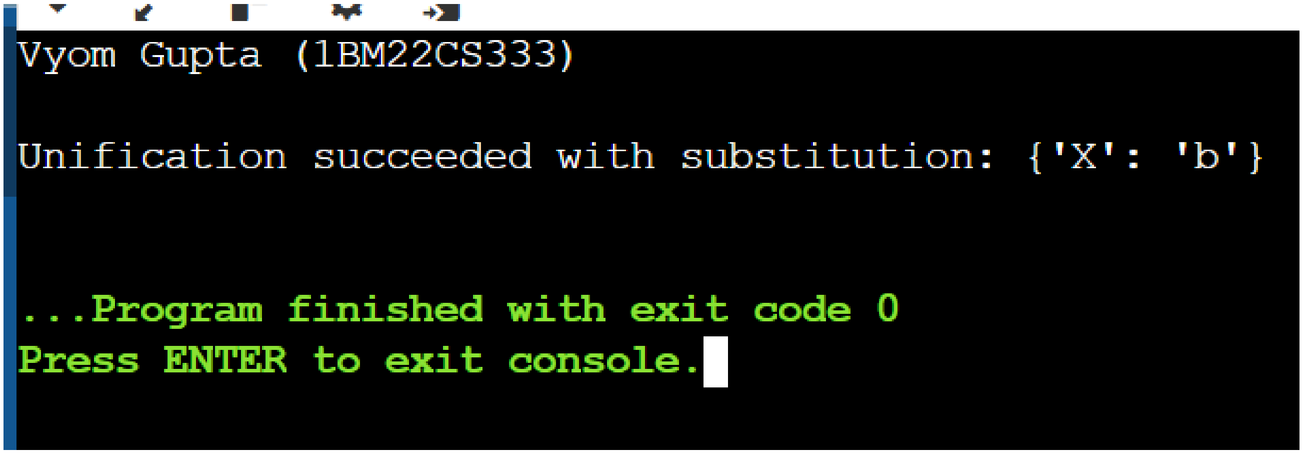
22/11/24

Code

```
print("Vyom Gupta (1BM22CS333)\n")


def occurs_check(var, term):
    """Check if a variable occurs in a term."""
    if var == term:
        return True
    elif isinstance(term, tuple):  # If the term is a function or a tuple
        return any(occurs_check(var, t) for t in term[1:])
    return False


def unify(term1, term2, substitution=None):
    """Attempt to unify two terms (or predicates)."""
    if substitution is None:
        substitution = {}

    # If both terms are the same, no unification needed
    if term1 == term2:
        return substitution
```

```
# If term1 is a variable, try to unify it with term2

if isinstance(term1, str) and term1.isupper():

    if term1 in substitution:

        return unify(substitution[term1], term2, substitution)

    if occurs_check(term1, term2):

        return None  # Avoid circular unification (occurs check)

    substitution[term1] = term2

    return substitution



# If term2 is a variable, try to unify it with term1

if isinstance(term2, str) and term2.isupper():

    return unify(term2, term1, substitution)



# If both terms are functions or predicates (tuples), unify their components

if isinstance(term1, tuple) and isinstance(term2, tuple):

    if len(term1) != len(term2):

        return None  # Different number of arguments

    for t1, t2 in zip(term1[1:], term2[1:]):
            substitution = unify(t1, t2, substitution)

            if substitution is None:

                return None  # If any unification fails, return None

        return substitution



    return None  # If no other cases match, return None (failure)



# Example usage

term1 = ('P', 'X', 'a')  # Predicate P(X, a)

term2 = ('P', 'b', 'a')  # Predicate P(b, a)



# Attempt to unify

substitution = unify(term1, term2)

if substitution is not None:

    print("Unification succeeded with substitution:", substitution)

else:

    print("Unification failed")
```

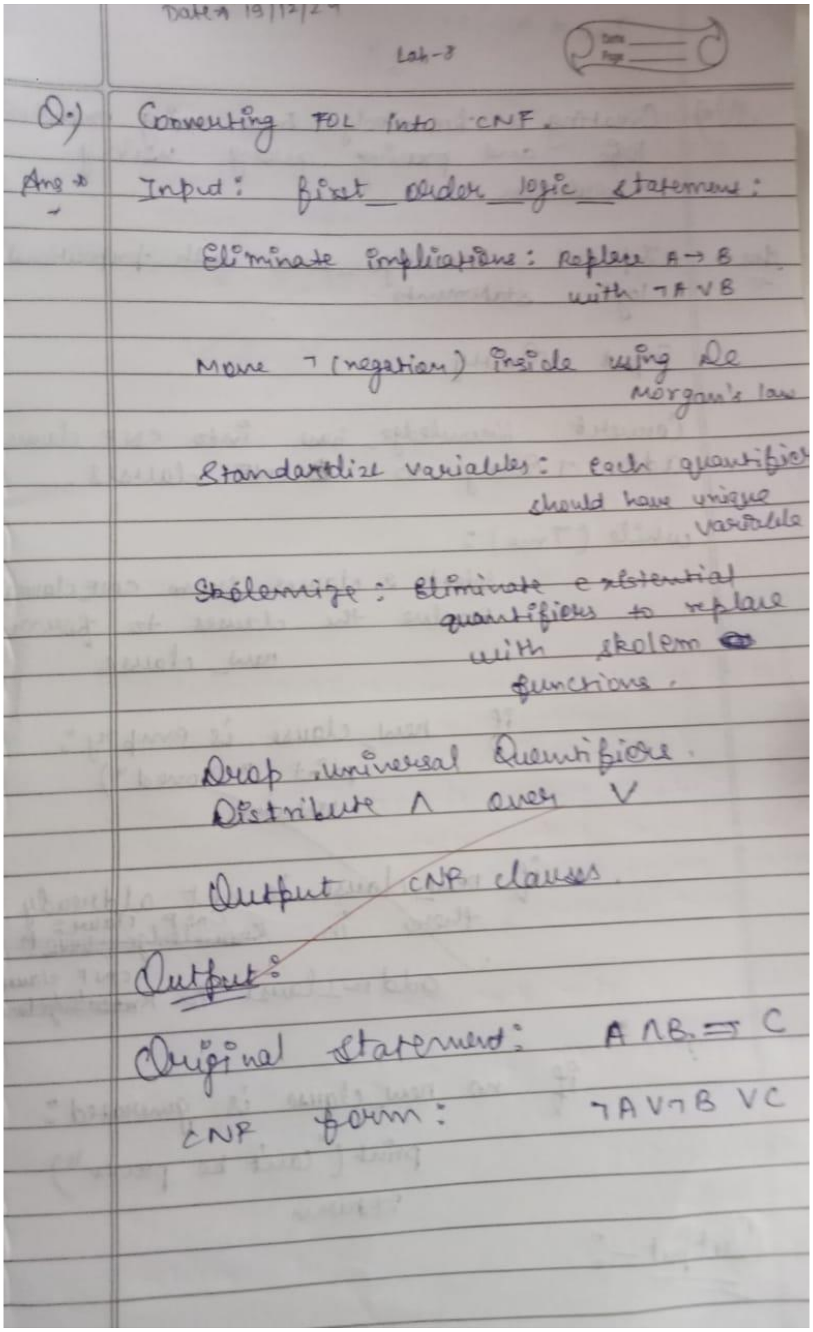
Output



```
Vyom Gupta (1BM22CS333)

Unification succeeded with substitution: {'X': 'b'}


...Program finished with exit code 0
Press ENTER to exit console.
```

Program 10 - Convert a given first order logic statement into Conjunctive Normal Form (CNF).

Algorithm



Date: 19/12/24

Lab-8

Date ____ Page ____

Q.) Converting FOL into CNF.

Ans:- Input: first_order_logic_statement:

Eliminate implications: Replace $A \rightarrow B$ with $\neg A \vee B$

Move $\neg$ (negation) inside using De Morgan's law

Standardize variables: each quantifier should have unique variable

Skolemize: Eliminate existential quantifiers to replace with skolem functions.

Drop universal Quantifiers.

Distribute $\wedge$ over $\vee$

Output CNF clauses.

Output:

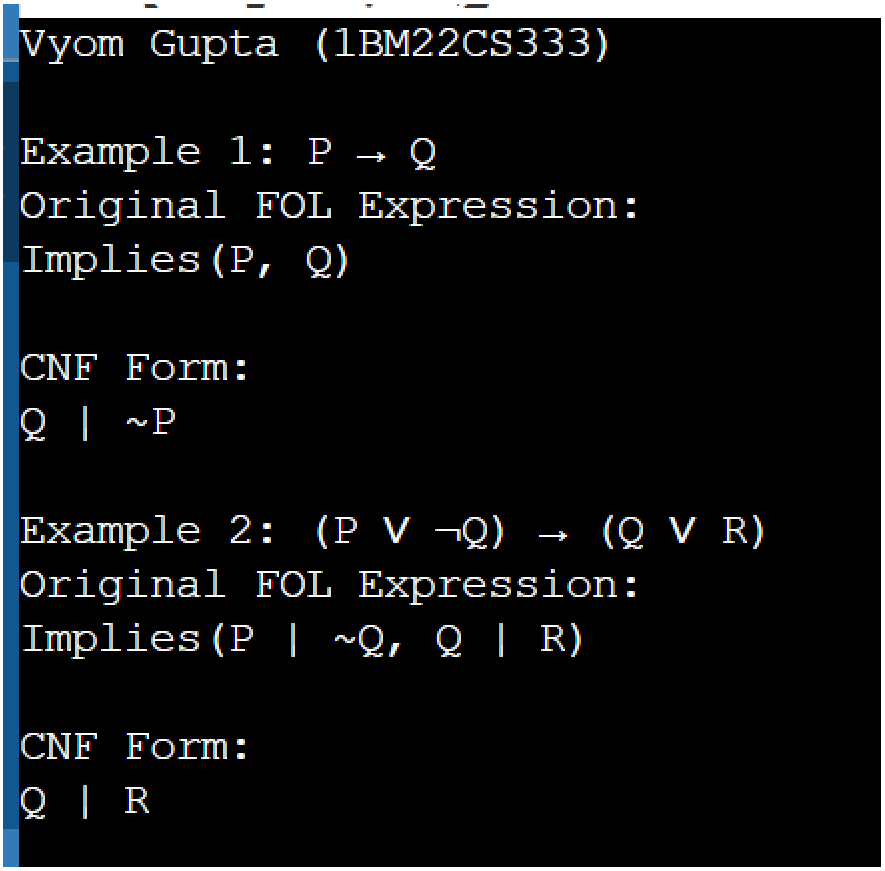
Original statement: $A \wedge B \Rightarrow C$

CNF form: $\neg A \vee \neg B \vee C$

Code

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

def fol_to_cnf(fol_expr):
    """
    Converts a First-Order Logic (FOL) statement to Conjunctive Normal Form (CNF).

    Arguments:
    fol_expr: A sympy logical expression representing the FOL statement.

    Returns:
    The CNF equivalent of the input expression.
    """
    # Step 1: Eliminate equivalences (A ↔ B) using (A → B) ∧ (B → A)
    fol_expr = fol_expr.replace(Equivalent, lambda a, b: And(Implies(a, b), Implies(b, a)))

    # Step 2: Eliminate implications (A → B) using (¬A ∨ B)
    fol_expr = fol_expr.replace(Implies, lambda a, b: Or(Not(a), b))

    # Step 3: Convert to CNF
    cnf_form = to_cnf(fol_expr, simplify=True)

    return cnf_form

def main():
    # Define propositional symbols instead of first-order predicates
    P = symbols("P")
    Q = symbols("Q")
    R = symbols("R")

    # Example 1: P → Q
    fol_expr1 = Implies(P, Q)
    print("Example 1: P → Q")
    print("Original FOL Expression:")
    print(fol_expr1)
    # Convert to CNF
    cnf1 = fol_to_cnf(fol_expr1)
    print("\nCNF Form:")
    print(cnf1)

    # Example 2: (P ∨ ¬Q) → (Q ∨ R)
    fol_expr2 = Implies(Or(P, Not(Q)), Or(Q, R))
    print("\nExample 2: (P ∨ ¬Q) → (Q ∨ R)")
    print("Original FOL Expression:")
    print(fol_expr2)
    # Convert to CNF
    cnf2 = fol_to_cnf(fol_expr2)
    print("\nCNF Form:")
    print(cnf2)

# Print name and USN at the start
print("Vyom Gupta (1BM22CS333)\n")

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

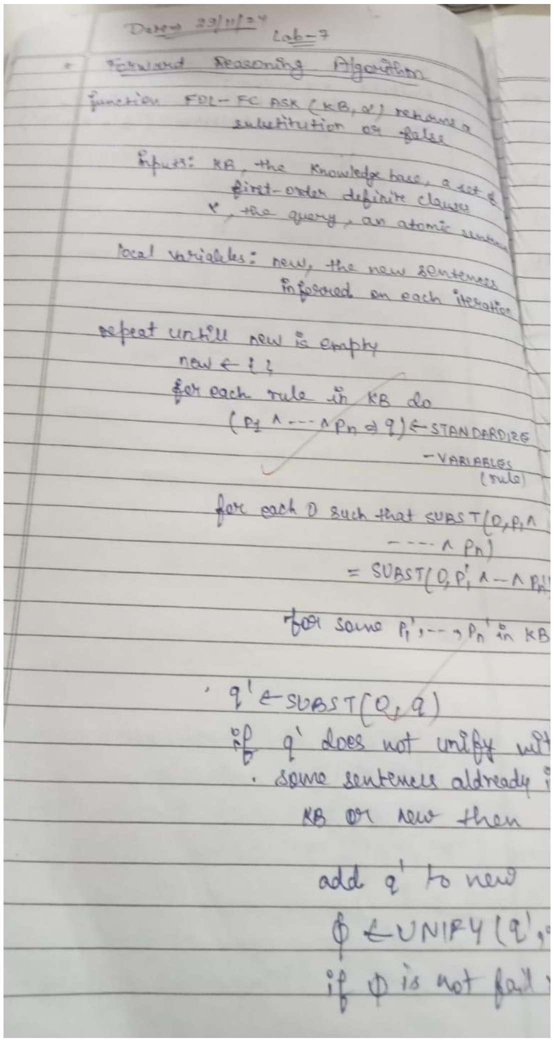
OUTPUT:



```
Vyom Gupta (1BM22CS333)

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
```

Program 11 - Knowledge base consisting of first order logic statements and prove the given query using forward reasoning..

Algorithm

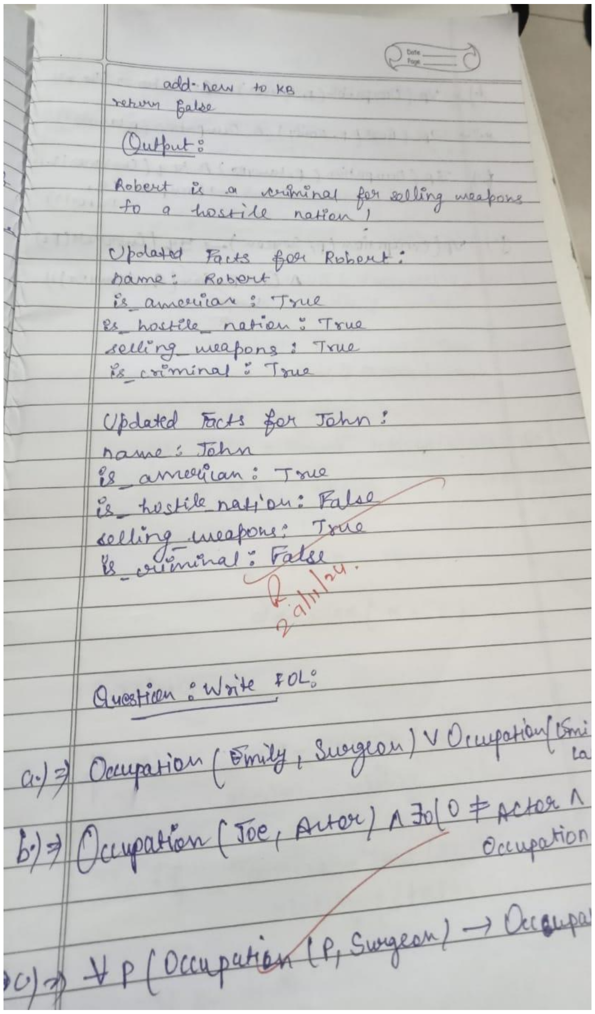


Date → 29/11/24

Lab-7

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 atomic senten

    local variables: new, the new sentences
            inferred on each iteration

repeat until new is empty
    new ← { }
    for each rule in KB do
        ($p_1 \wedge --- \wedge p_n \Rightarrow q$) ← STANDARDIZE
                                    -VARIABLES
                                    (rule)

        for each θ such that SUBST(θ, $p_1 \wedge$
                        ---- $\wedge p_n$)
                = SUBST(θ, $p_1' \wedge -- \wedge p_n'$

            for some $p_1', --, p_n'$ in KB

            q' ← SUBST(θ, q)
            if q' does not unify wit
              some sentences already i
              KB or new then

                add q' to new
                φ ← UNIFY(q',
                if φ is not fail
```



Date ______
Page ______

```
        add new to KB
return false
```

Output:

```
Robert is a criminal for selling weapons
to a hostile nation!

Updated Facts for Robert:
name: Robert
is_american: True
is_hostile_nation: True
selling_weapons: True
is_criminal: True

Updated Facts for John:
name: John
is_american: True
is_hostile_nation: False
selling_weapons: True
is_criminal: False
```

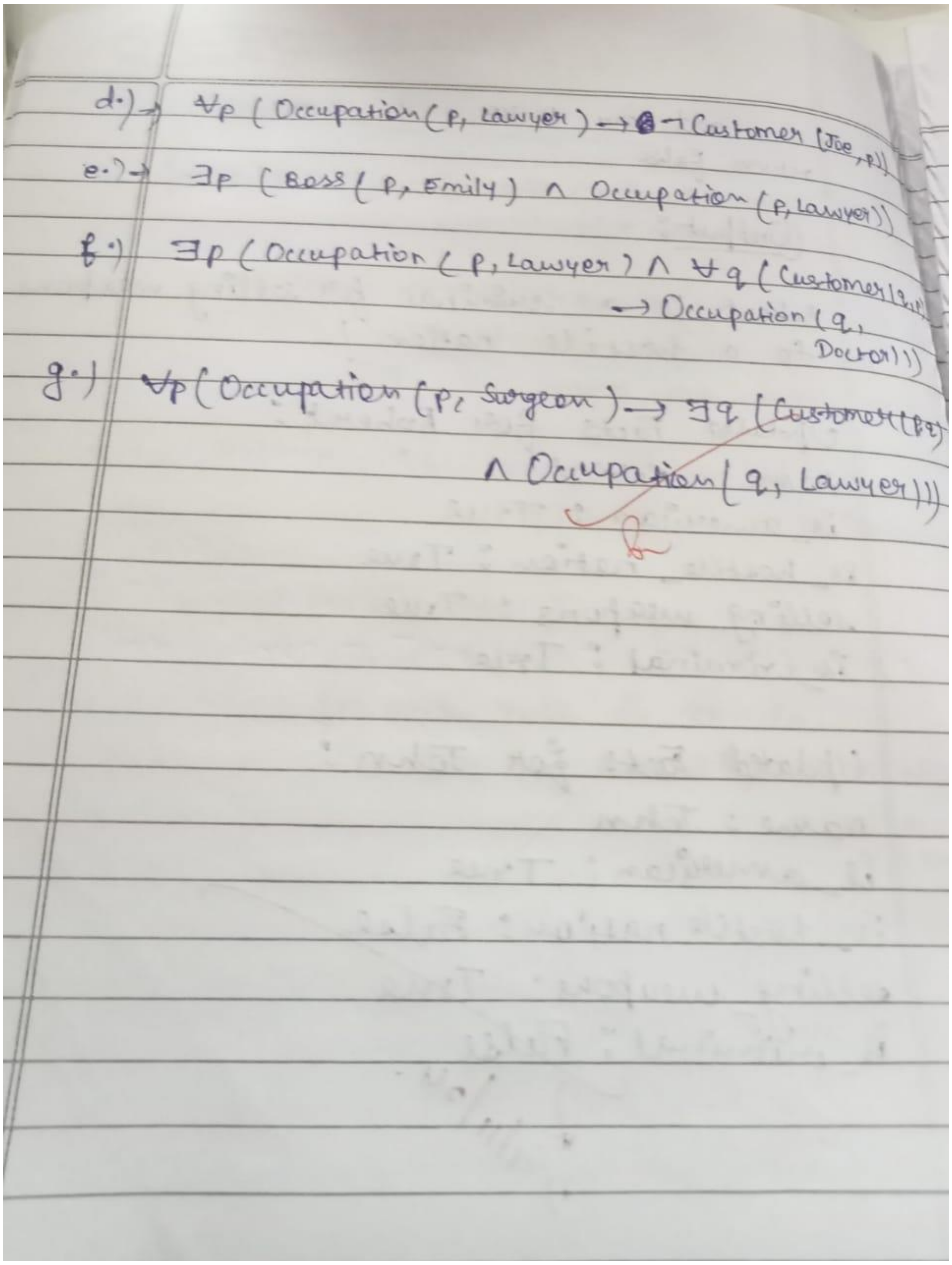
R
29/11/24.

Question: Write FOL:

a.) ⇒ Occupation (Emily, Surgeon) ∨ Occupation (Emi
La

b.) ⇒ Occupation (Joe, Actor) ∧ ∃o (o ≠ Actor ∧
Occupation

c.) ⇒ ∀ p (Occupation (P, Surgeon) → Occupa



d.)→ $\forall p\,(\text{Occupation}(p, \text{Lawyer}) \rightarrow \neg\text{Customer}(\text{Joe}, p))$

e.)→ $\exists p\,(\text{Boss}(p, \text{Emily}) \wedge \text{Occupation}(p, \text{Lawyer}))$

f.) $\exists p\,(\text{Occupation}(p, \text{Lawyer}) \wedge \forall q\,(\text{Customer}(q, p$
$\rightarrow \text{Occupation}(q,$
$\text{Doctor})))$

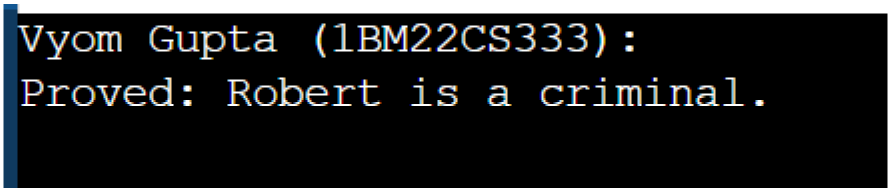
g.) $\forall p\,(\text{Occupation}(p, \text{Surgeon}) \rightarrow \exists q\,(\text{Customer}(p,q)$
$\wedge\, \text{Occupation}(q, \text{Lawyer})))$

Code

```
knowledge_base = {
"facts": {
"American(Robert)": True,
"Enemy(A, America)": True,
"Owns(A, T1)": True,
"Missile(T1)": True,
},
"rules": [
{"if": ["Missile(x)"], "then": ["Weapon(x)"]},
{"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
{"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},
{
"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"],
"then": ["Criminal(p)"],
},
],
}
def forward_chaining(kb):
facts = kb["facts"].copy()
rules = kb["rules"]
inferred = set()
while True:
43new_inferences = set()
for rule in rules:
if_conditions = rule["if"]
then_conditions = rule["then"]
substitutions = {}
all_conditions_met = True
for condition in if_conditions:
predicate, args = condition.split("(")
args = args[:-1].split(",")
matched = False
for fact in facts:
fact_predicate, fact_args = fact.split("(")
fact_args = fact_args[:-1].split(",")
if predicate == fact_predicate and len(args) == len(fact_args):
temp_subs = {}
for var, val in zip(args, fact_args):
if var.islower():
if var in temp_subs and temp_subs[var] != val:
break
```

```
temp_subs[var] = val
elif var != val:
break
else:
matched = True
substitutions.update(temp_subs)
break
if not matched:
all_conditions_met = False
break
44if all_conditions_met:
for condition in then_conditions:
predicate, args = condition.split("(")
args = args[:-1].split(",")
new_fact = predicate + "(" + ",".join(substitutions.get(arg, arg) for arg in args)
+ ")"
new_inferences.add(new_fact)
if new_inferences - inferred:
inferred.update(new_inferences)
facts.update({fact: True for fact in new_inferences})
else:
break
return inferred
result = forward_chaining(knowledge_base)
print('Vyom Gupta (1BM22CS333):')
if "Criminal(Robert)" in result:
print("Proved: Robert is a criminal.")
else:
print("Could not prove that Robert is a criminal.")
```

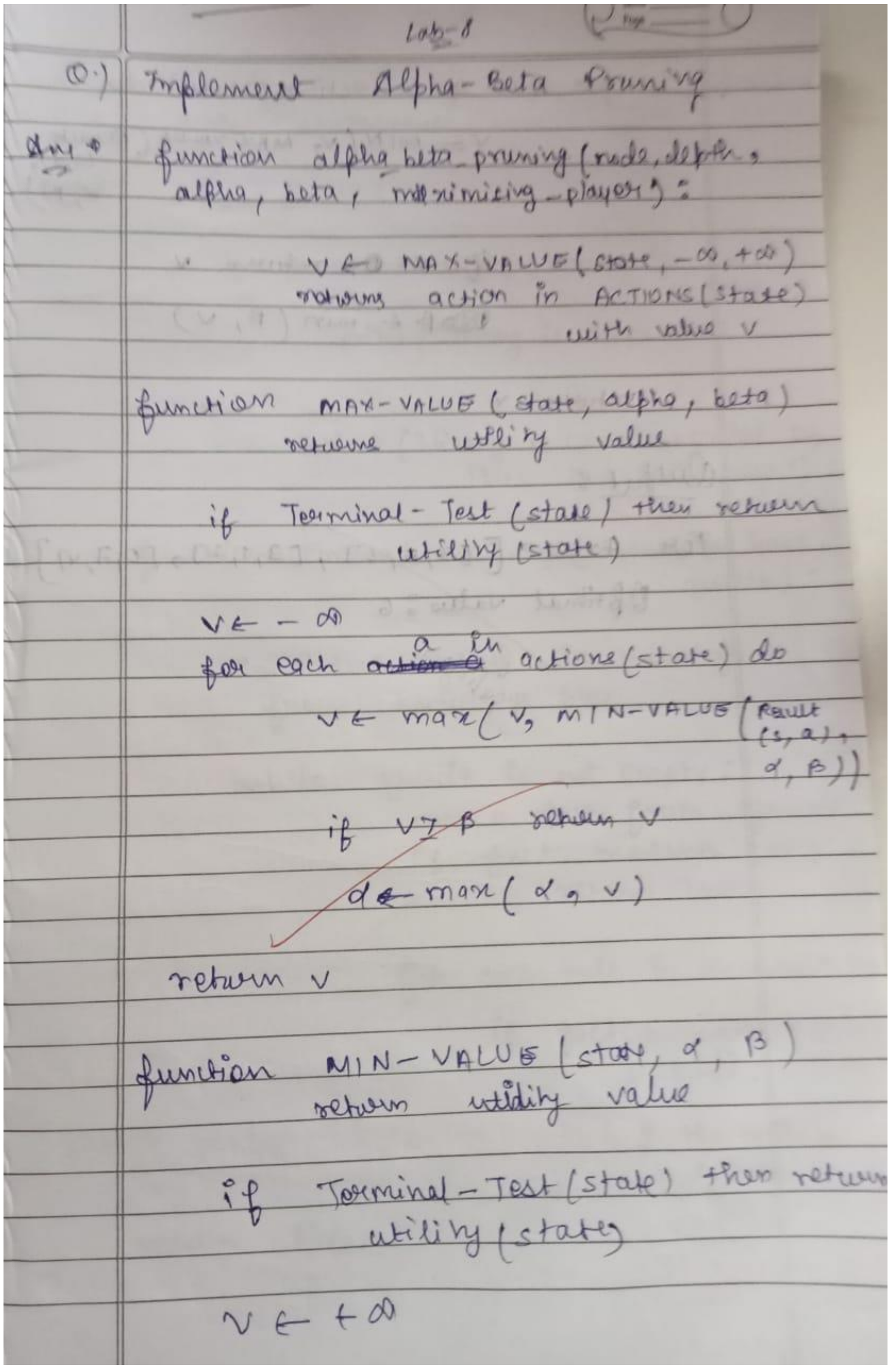
OUTPUT:



```
Vyom Gupta (1BM22CS333):
Proved: Robert is a criminal.
```

Program 12 - Implement Alpha-Beta Pruning.

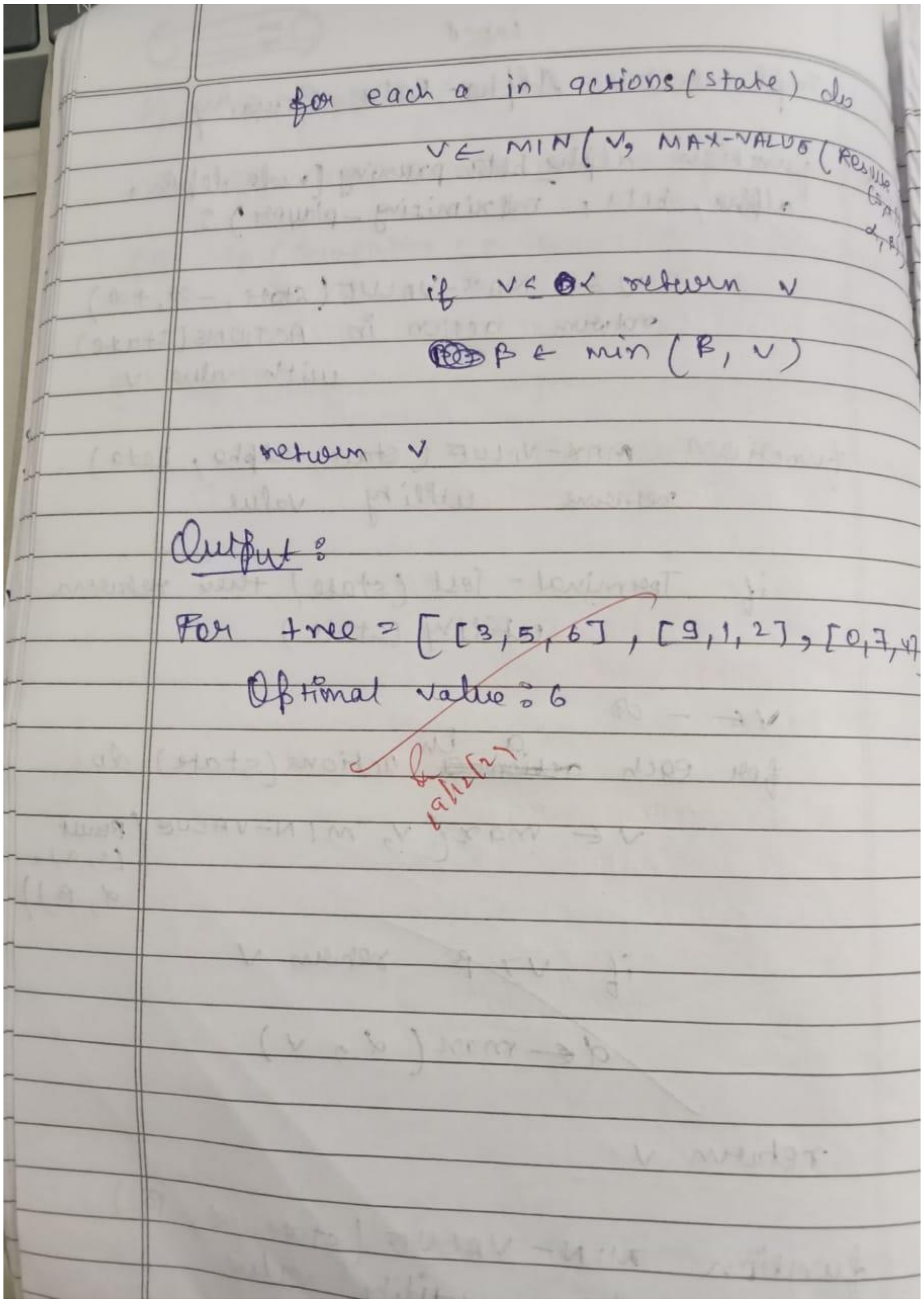
Algorithm



Lab-8

Q.) Implement Alpha-Beta Pruning

```
Ans → function alpha beta_pruning (node, depth,
      alpha, beta, maximising_player):

          $v \leftarrow$ MAX-VALUE(state, $-\infty$, $+\infty$)
          returns action in ACTIONS(state)
                        with value v

function MAX-VALUE (state, alpha, beta)
          returns utility value

      if Terminal-Test (state) then return
                utility (state)

      $v \leftarrow -\infty$
      for each ~~action~~ a in actions(state) do
          $v \leftarrow$ max(v, MIN-VALUE(Result(s, a), $\alpha$, $\beta$))

          if $v \geq \beta$ return v

          $\alpha \leftarrow$ max($\alpha$, v)

   return v

function MIN-VALUE (state, $\alpha$, $\beta$)
          return utility value

      if Terminal-Test (state) then return
            utility (state)

      $v \leftarrow +\infty$
```



```
for each a in actions(state) do
        v ← MIN(v, MAX-VALUE(Result(s,a), α, β))

        if v ≤ α return v
        β ← min(β, v)

return v
```

Output:

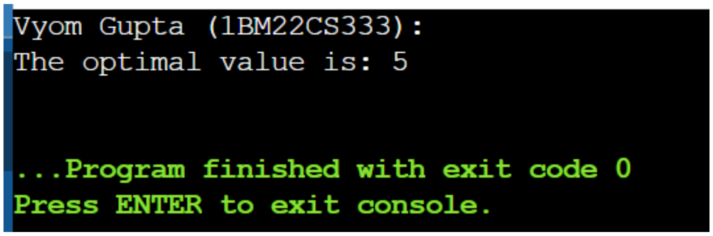
For tree = [[3,5,6], [9,1,2], [0,7,4]
    Optimal value: 6

19/12/24

Code

```
import math
def alpha_beta_pruning(depth, node_index, is_maximizing_player, values, alpha, beta,
max_depth):
# Base case: when the maximum depth is reached
if depth == max_depth:
return values[node_index]
if is_maximizing_player:
best = -math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, False, values, alpha, beta,
max_depth)
best = max(best, val)
alpha = max(alpha, best)
# Prune the remaining nodes
if beta <= alpha:
break
return best
else:
best = math.inf
# Recur for left and right children
for i in range(2):
val = alpha_beta_pruning(depth + 1, node_index * 2 + i, True, values, alpha, beta,
max_depth)
best = min(best, val)
beta = min(beta, best)
49# Prune the remaining nodes
if beta <= alpha:
break
return best
print(" Vyom Gupta (1BM22CS333):")
# Example usage
if __name___== "__main__":
# Example tree represented as a list of leaf node values
values = [3, 5, 6, 9, 1, 2, 0, -1]
max_depth = 3 # Height of the tree
result = alpha_beta_pruning(0, 0, True, values, -math.inf, math.inf, max_depth)
print("The optimal value is:", result)
```

OUTPUT:



```
Vyom Gupta (1BM22CS333):
The optimal value is: 5


...Program finished with exit code 0
Press ENTER to exit console.
```