# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

**"JnanaSangama", Belgaum -590014, Karnataka.**

**LAB REPORT**
**on**

# Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**Anish Arjun Budavi (1BM23CS401)**

***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 **Anish Arjun Budavi (1BM23CS401),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

| | |
|---|---|
| Prof. Shravya AR<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index

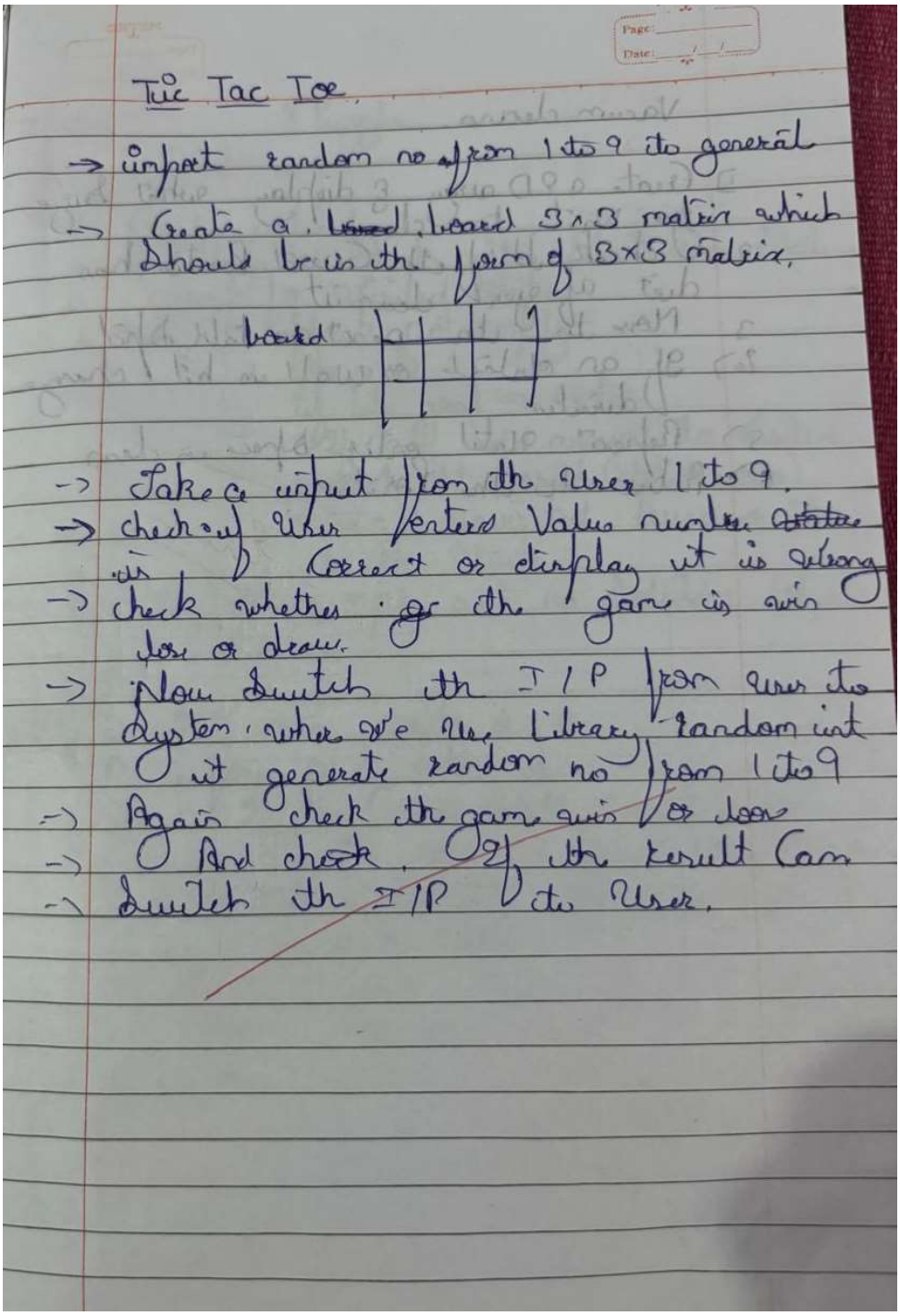

| Sl. No. | Date | Experiment Title | Page No. |
|---|---|---|---|
| 1 | 30-9-2024 | Implement Tic –Tac –Toe Game<br>Implement vacuum cleaner agent | 4-10 |
| 2 | 7-10-2024 | Implement 8 puzzle problems using Depth First Search (DFS)<br>Implement Iterative deepening search algorithm | 11-18 |
| 3 | 14-10-2024 | Implement A* search algorithm | 19-32 |
| 4 | 21-10-2024 | Implement Hill Climbing search algorithm to solve N-Queens problem | 33-37 |
| 5 | 28-10-2024 | Simulated Annealing to Solve 8-Queens problem | 38-40 |
| 6 | 11-11-2024 | Create a knowledge base using propositional logic and show that the given query entails the knowledge base or not. | 41-45 |
| 7 | 2-12-2024 | Implement unification in first order logic | 46-49 |
| 8 | 2-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using forward reasoning. | 50-53 |
| 9 | 16-12-2024 | Create a knowledge base consisting of first order logic statements and prove the given query using Resolution | 54-63 |
| 10 | 16-12-2024 | Implement Alpha-Beta Pruning. | 64-66 |


**Github Link:**
**https://github.com/Anishbudavi/AI-1BM23CS401.git**

## Program 1

### Tic-Tac-Toe:

**Algorithm:**



**Code:**

```
import random
board = [[' ' for _ in range(3)] for _ in range(3)]
def print_board():
    for row in board:
        print(' | '.join(row))
        print('-' * 9)
def check_win(player):
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] == player:
            return True
        if board[0][i] == board[1][i] == board[2][i] == player:
            return True
    if board[0][0] == board[1][1] == board[2][2] == player:
        return True
    if board[0][2] == board[1][1] == board[2][0] == player:
        return True
    return False
def check_draw():
    for row in board:
        if ' ' in row:
            return False
    return True
def computer():
    for row in range(3):
        for col in range(3):
            if board[row][col] == ' ':
                board[row][col] = 'O'
                if check_win('O'):
                    return
                board[row][col] = ' '
    for row in range(3):
        for col in range(3):
            if board[row][col] == ' ':
                board[row][col] = 'X'
                if check_win('X'):
                    board[row][col] = 'O'
                    return
                board[row][col] = ' '
    if board[1][1] == ' ':
```

```
        board[1][1] = 'O'
        return
    corners = [(0, 0), (0, 2), (2, 0), (2, 2)]
    random.shuffle(corners)
    for row, col in corners:
        if board[row][col] == ' ':
            board[row][col] = 'O'
            return
    sides = [(0, 1), (1, 0), (1, 2), (2, 1)]
    random.shuffle(sides)
    for row, col in sides:
        if board[row][col] == ' ':
            board[row][col] = 'O'
            return
flag = True
while flag:
    print_board()
    while True:
        try:
            print("Player 1's turn (X)")
            user_input = input("Enter row and column (1-3) separated by space: ")
            row, col = map(int, user_input.split())
            row -= 1
            col -= 1
            if row in range(3) and col in range(3) and board[row][col] == ' ':
                board[row][col] = 'X'
                break
            else:
                print("Invalid move, cell already taken or out of bounds.")
        except (ValueError, IndexError):
            print("Invalid input, please enter two numbers between 1 and 3 separated by space.")
    if check_win('X'):
        print_board()
        print("Player 1 wins!")
        break
    if check_draw():
        print_board()
        print("Draw!")
        break
    print_board()
    print("Player 2's turn (O)")
```

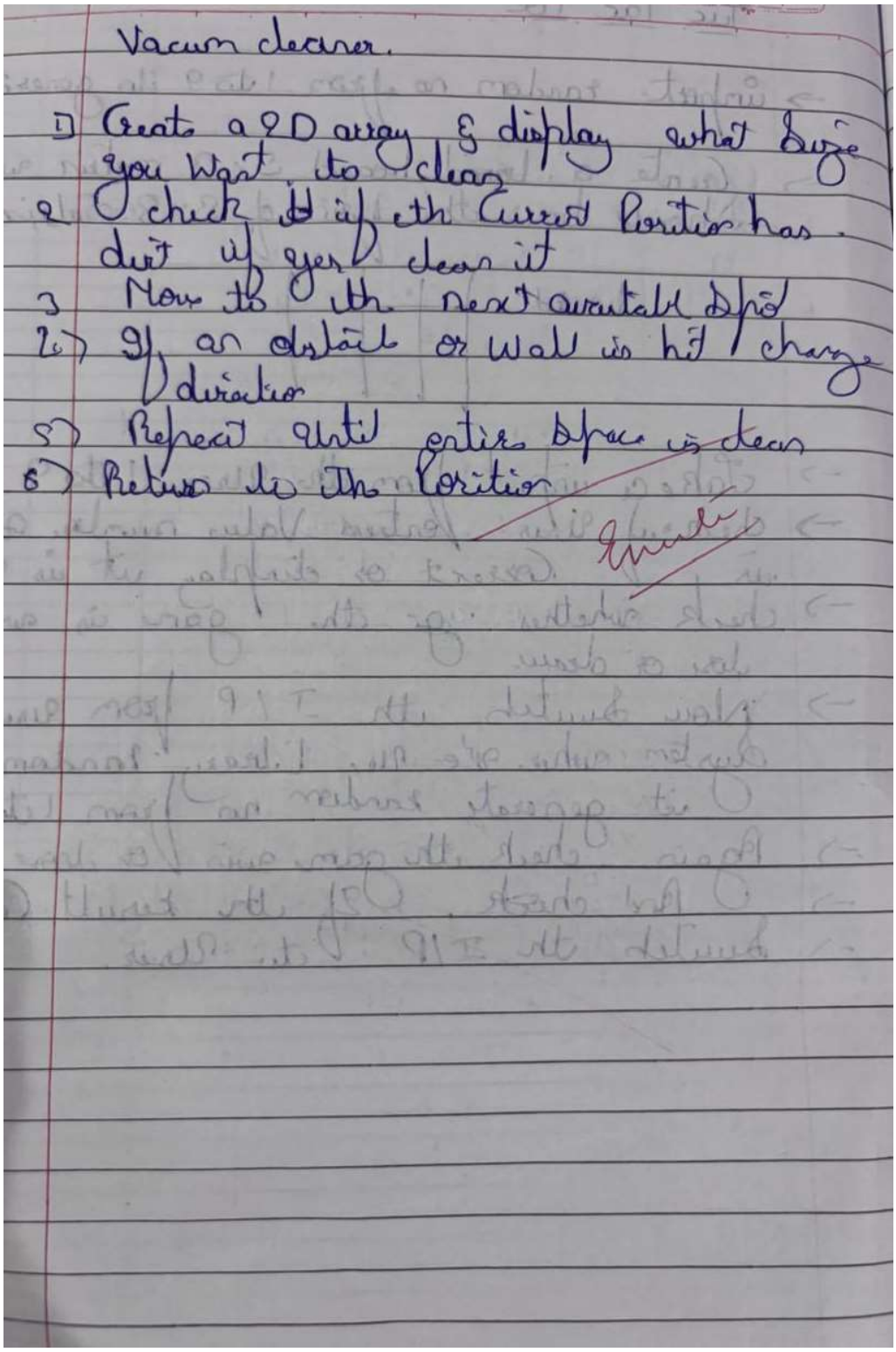
```
    computer()
    if check_win('O'):
        print_board()
        print("Player 2 wins!")
        break
    if check_draw():
        print_board()
        print("Draw!")
        break
```

**Output:**

```
Player 1's turn (X)
Enter row and column (1-3) separated by space: 1 2
X | X | O
---------
O | X | X
---------
  |   | O
---------
Player 2's turn (O)
X | X | O
---------
O | X | X
---------
  | O | O
---------
Player 1's turn (X)
Enter row and column (1-3) separated by space: 3 1
X | X | O
---------
O | X | X
---------
X | O | O
---------
Draw!
```

**Vacuum Cleaner Agent:**

**Algorithm:**



**Code:**

```
class CustomVacuumCleaner:
    def __init__(self, states, start_position):
        self.rooms = states
        self.position = start_position
        self.loc = ['A', 'B']

    def is_dirty(self):
        return self.rooms[self.position] == 1

    def clean(self):
        if self.is_dirty():
            print(f"Cleaning room {self.loc[self.position]}")
            self.rooms[self.position] = 0
        else:
            print(f"Room {self.loc[self.position]} is already clean.")

    def move(self):
        if self.position == 0:
            self.position = 1
        else:
            self.position = 0
    def run(self):
        cleaned = 0
        while cleaned <= 1:
            self.clean()
            self.move()
            cleaned += 1
        print("All rooms are clean!"
a = int(input("Enter state for room A, 0 for clean and 1 for dirty "))
b = int(input("Enter state for room B, 0 for clean and 1 for dirty "))
states = [a, b]
start_position = int(input("Enter start position, 0 for room A, 1 for room B "))
vacuum = CustomVacuumCleaner(states, start_position)
vacuum.run()
```

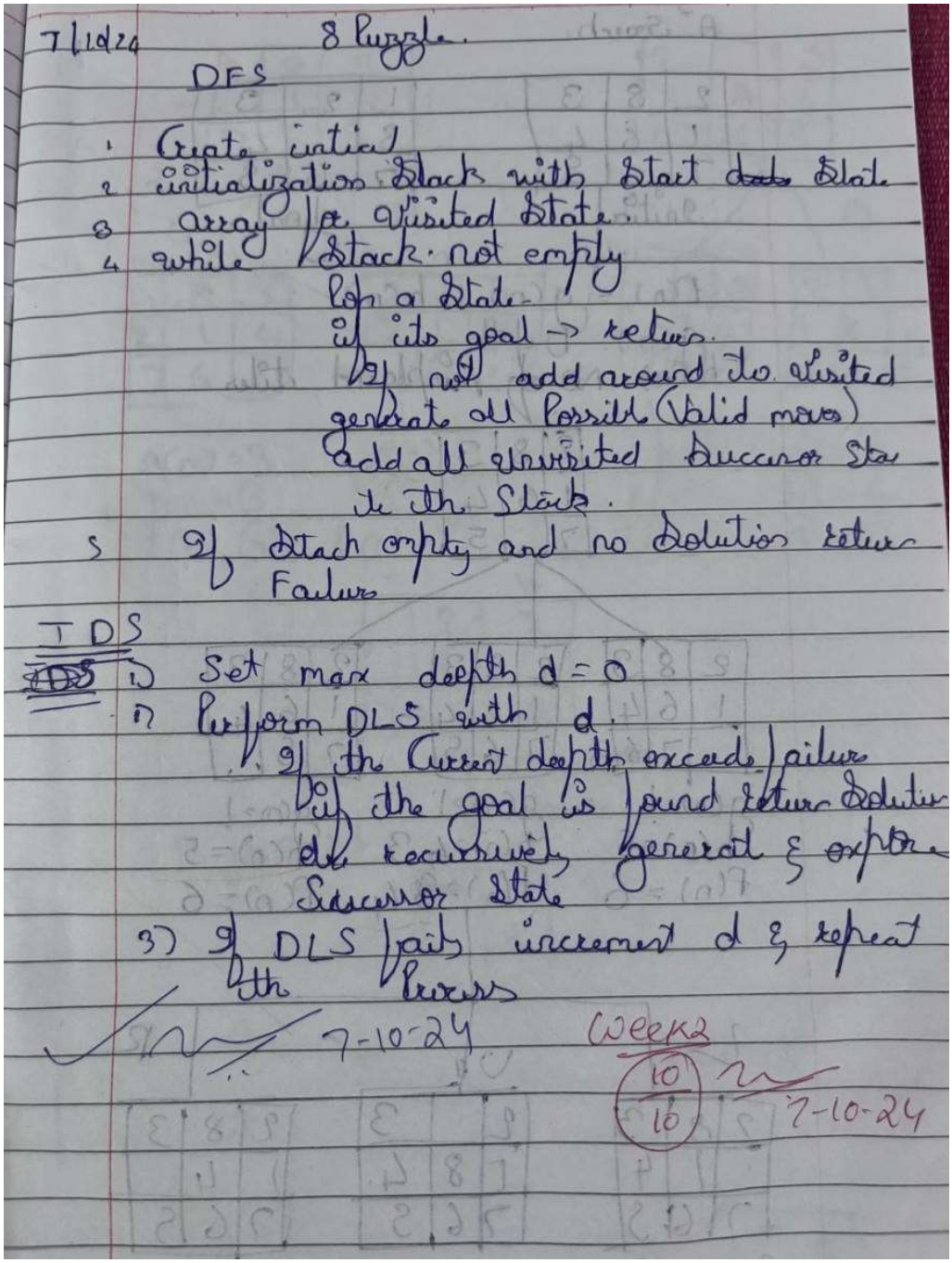
**Output:**

```
Enter state for room A, 0 for clean and 1 for dirty 0
Enter state for room B, 0 for clean and 1 for dirty 0
Enter start position, 0 for room A, 1 for room B 1
Room B is already clean.
Room A is already clean.
All rooms are clean!
```

## Program 2

**Depth First Search and Iterative Deepening Search:**

**Algorithm:**



7/10/24 8 Puzzle.

DFS

1. Create initial
2. initialization Stack with Start ~~data~~ State
3. array for visited State.
4. while Stack not empty
   - Pop a State
   - if its goal → return.
   - if not add around to visited generate all Possible (Valid moves)
   - add all unvisited Succesor Sta to the Stack.
5. if Stack empty and no Solution return Failure

IDS

1) Set max depth d = 0
2) Perform DLS with d.
   - if the Current depth exceeds failure
   - if the goal is found return Solution
   - else recursively generate & explore Successor State
3) if DLS fails increment d & repeat the process

7-10-24

WEEK2

10/10

7-10-24

**Code:**

```
from collections import deque

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

MOVES = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def is_valid_move(board, move):
    x, y = get_zero_position(board)
    dx, dy = MOVES[move]
    new_x, new_y = x + dx, y + dy
    return 0 <= new_x < 3 and 0 <= new_y < 3

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

def apply_move(board, move):
    x, y = get_zero_position(board)
    dx, dy = MOVES[move]
    new_x, new_y = x + dx, y + dy
    board[x][y], board[new_x][new_y] = board[new_x][new_y], board[x][y]

def is_goal_state(board):
    return board == GOAL_STATE

def dfs(board):
    stack = deque([(board, [])])
    visited = set()
```

```
    while stack:
        current_board, moves = stack.pop()
        if tuple(map(tuple, current_board)) in visited:
            continue
        visited.add(tuple(map(tuple, current_board)))
        if is_goal_state(current_board):
            return moves
        for move in MOVES:
            new_board = [row[:] for row in current_board]
            if is_valid_move(new_board, move):
                apply_move(new_board, move)
                stack.append((new_board, moves + [move]))
    return None

def print_steps(moves):
    board = [
        [1, 2, 3],
        [4, 5, 0],
        [7, 8, 6]

    ]
    for move in moves:
        x, y = get_zero_position(board)
        dx, dy = MOVES[move]
        new_x, new_y = x + dx, y + dy
        board[x][y], board[new_x][new_y] = board[new_x][new_y], board[x][y]
        print("Move:", move)
        print("Board:")
        for row in board:
            print(row)
        print()

initial_board = [
    [1, 2, 3],
    [4, 5, 0],
    [7, 8, 6]
]
solution = dfs(initial_board)
if solution:
    print("Solution found:")
    print_steps(solution)
```

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

**Output:**

```
Move: down
Board:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

Move: right
Board:
[1, 2, 3]
[4, 5, 6]
[7, 0, 8]

Move: right
Board:
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```

**Iterative Deepening Search:**

**Code:**

```
from collections import deque

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

MOVES = {
    'up': (-1, 0),
    'down': (1, 0),
    'left': (0, -1),
    'right': (0, 1)
}

def is_valid_move(board, move):
    x, y = get_zero_position(board)
    dx, dy = MOVES[move]
    new_x, new_y = x + dx, y + dy
    return 0 <= new_x < 3 and 0 <= new_y < 3

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

def apply_move(board, move):
    x, y = get_zero_position(board)
    dx, dy = MOVES[move]
    new_x, new_y = x + dx, y + dy
    board[x][y], board[new_x][new_y] = board[new_x][new_y], board[x][y]

def is_goal_state(board):
    return board == GOAL_STATE

def ids(board):
    for depth in range(1, 100):
```

```
        result = dls(board, depth)
        if result is not None:
            return result
    return None

def dls(board, depth):
    stack = deque([(board, [], 0)])
    visited = set()
    while stack:
        current_board, moves, current_depth = stack.pop()
        if tuple(map(tuple, current_board)) in visited:
            continue
        visited.add(tuple(map(tuple, current_board)))
        if is_goal_state(current_board):
            return moves
        if current_depth < depth:
            for move in MOVES:
                new_board = [row[:] for row in current_board]
                if is_valid_move(new_board, move):
                    apply_move(new_board, move)
                    stack.append((new_board, moves + [move], current_depth + 1))
    return None

def print_steps(moves):
    board = [
        [5, 4, 0],
        [6, 1, 8],
        [7, 3, 2]
    ]
    for move in moves:
        x, y = get_zero_position(board)
        dx, dy = MOVES[move]
        new_x, new_y = x + dx, y + dy
        board[x][y], board[new_x][new_y] = board[new_x][new_y], board[x][y]
        print("Move:", move)
        print("Board:")
        for row in board:
            print(row)
        print()

initial_board = [
```

```
    [5, 4, 0],
    [6, 1, 8],
    [7, 3, 2]
]
solution = ids(initial_board)
if solution:
    print("Solution found:")
    print_steps(solution)
else:
    print("No solution found.")
```

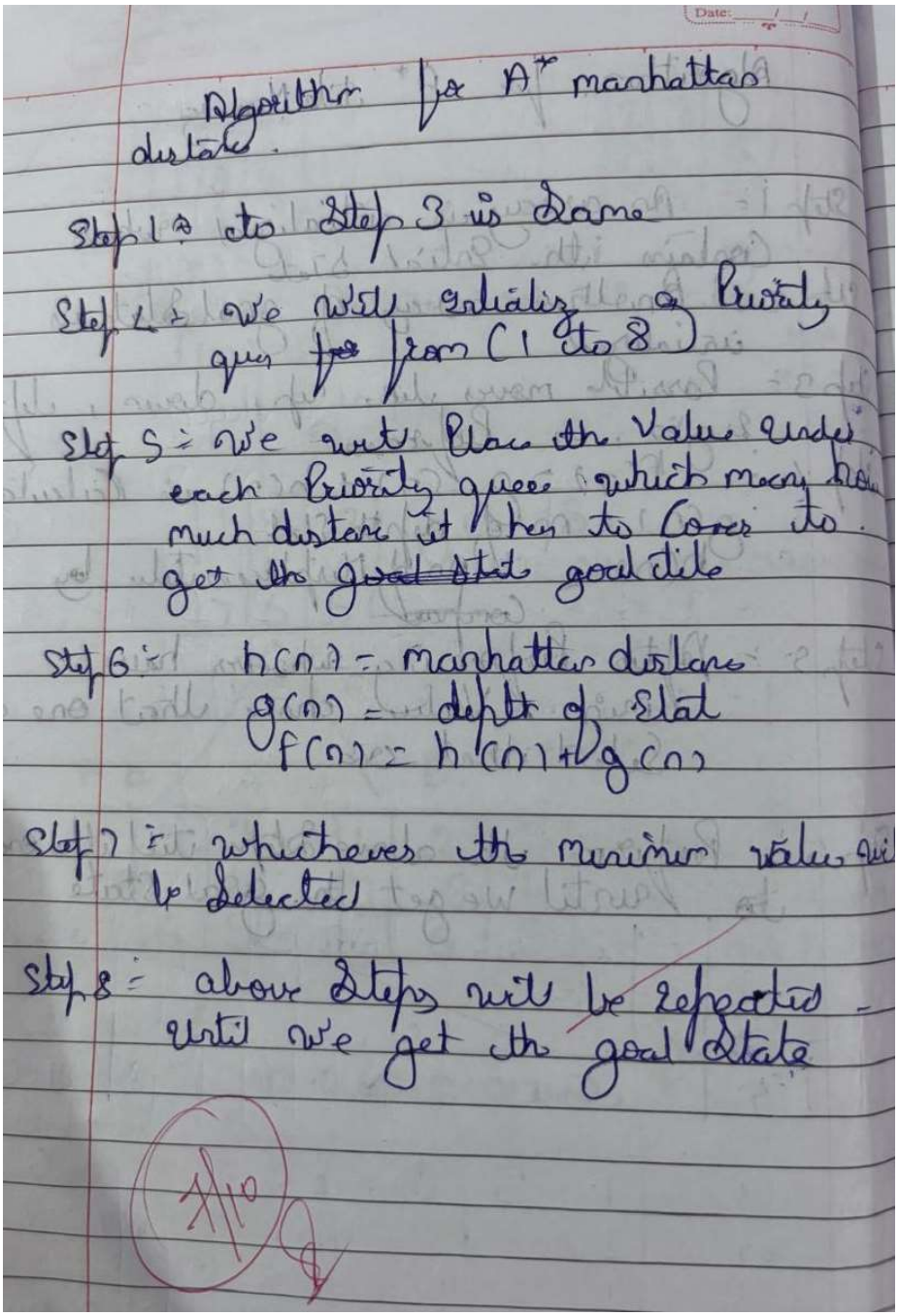
**Output:**

```
Move: left
Board:
[1, 4, 2]
[3, 0, 5]
[6, 7, 8]

Move: up
Board:
[1, 0, 2]
[3, 4, 5]
[6, 7, 8]

Move: left
Board:
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
```

## Program 3

### A* using Manhattan Distance:

**Algorithm:**



Week 3

A* implementation of 8th Puzzle

Case 1 :- g(n) depth of node h(n) 'Number of Misplaced tiles'

Case 2 :- g(n) :- depth of node h(n) manhattan distance A* f(n) = g(n) + h(n).

observation book.

1 State Space diagram for Case 2.

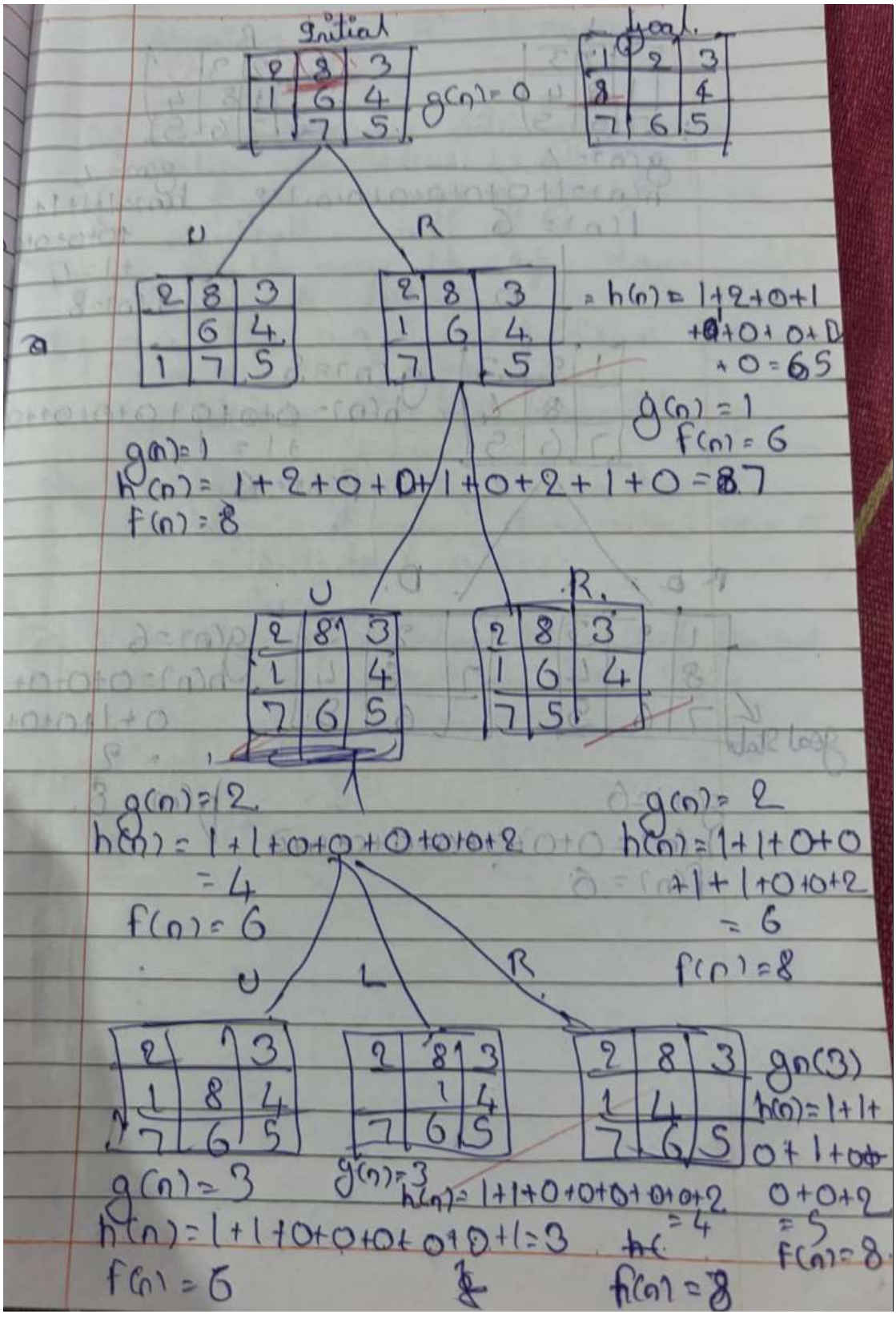
| | | |
|---|---|---|
| 2 | 8 | 3 |
| 1 | 6 | 4 |
| | 7 | 5 |

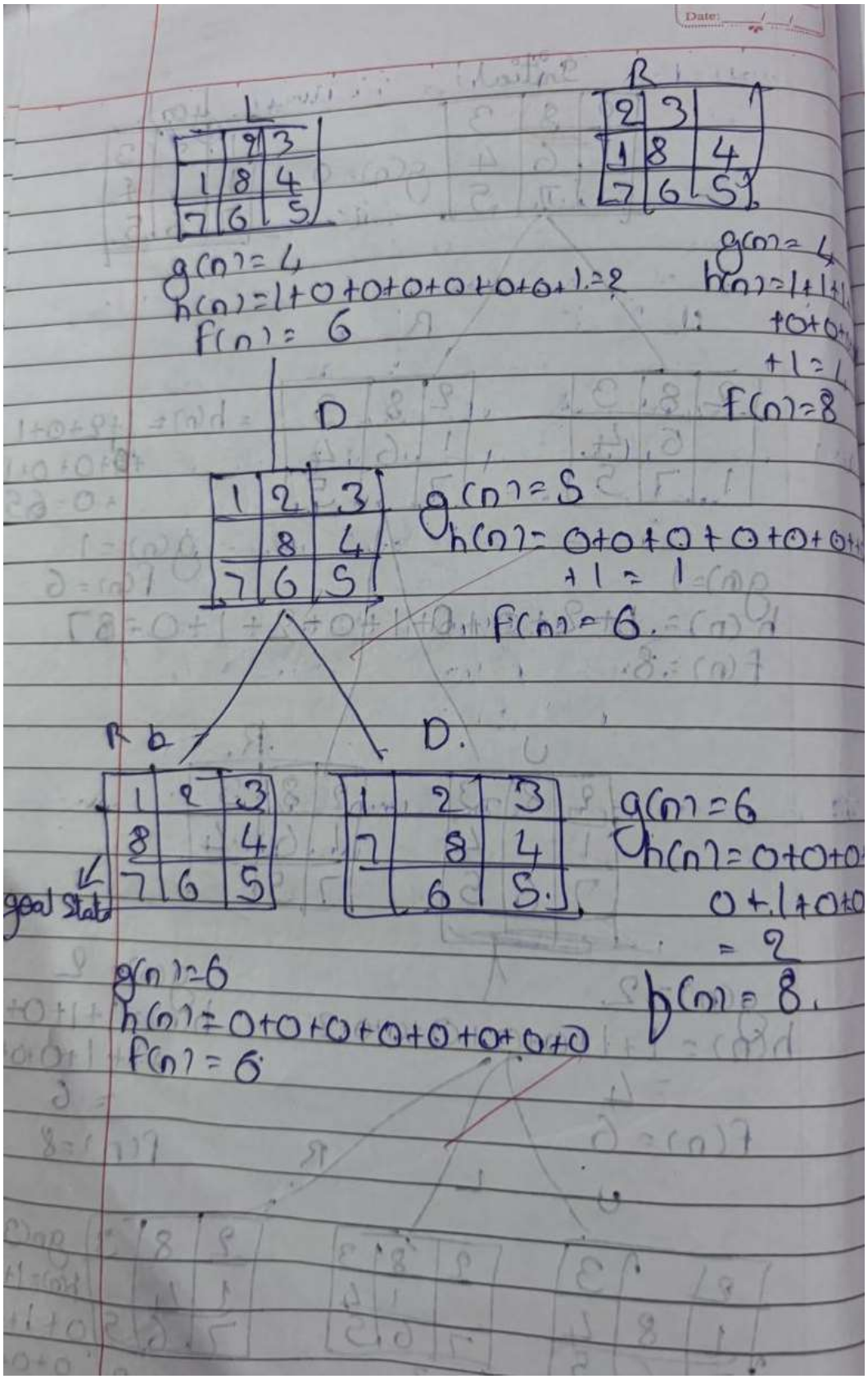
Initial

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

Goal

2 Logic Implementation for Case 1 & Case 2.


Initial
Goal
g(n)=0
U
R
h(n)=1+2+0+1
+0+0+0+0
=6
g(n)=1
f(n)=6
g(n)=1
h(n)=1+2+0+0+1+0+2+1+0=7
f(n)=8
U
R
g(n)=2
h(n)=1+1+0+0+0+0+0+2
=4
f(n)=6
g(n)=2
h(n)=1+1+0+0
+1+1+0+0+2
=6
f(n)=8
U
L
R
g(n)=3
h(n)=1+1+0+0+0+0+0+1=3
f(n)=6
g(n)=3
h(n)=1+1+0+0+0+0+0+2
=4
f(n)=7
g(n)=3
h(n)=1+1+
0+1+
0+0+2
=5
f(n)=8


L
R
D
R b
D.
goal state
g(n)=4
h(n)=1+0+0+0+0+0+0+1=2
f(n)=6
g(n)=4
h(n)=1+1+1
+0+0+
+1=4
f(n)=8
g(n)=5
h(n)=0+0+0+0+0+0+0+
+1=1
f(n)=6.
g(n)=6
h(n)=0+0+0+
0+1+0+0
=2
f(n)=8.
g(n)=6
h(n)=0+0+0+0+0+0+0+0
f(n)=6

**Code:**

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

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

  def copy(self, root):
    """ Copy function to create a similar matrix of the given node"""
    return [row[:] for row in root]

  def find(self, puz, x):
    """ Specifically used to find the position of the blank space """
    for i in range(len(self.data)):
      for j in range(len(self.data)):
```

```
            if puz[i][j] == x:
                return i, j


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

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

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

    def h(self, start, goal):
        distance = 0
        for i in range(self.n):
            for j in range(self.n):
                if start[i][j] != '_' and start[i][j] != goal[i][j]:
                    # target_value = start[i][j]
                    # target_x = (int(target_value) - 1) // self.n
                    # target_y = (int(target_value) - 1) % self.n
                    # distance += abs(target_x - i) + abs(target_y - j)
                    k=0
                    l=0
                    for m in range(self.n):
                        for n in range(self.n):
                            if goal[m][n] == target_value:
                                k = m
                                l = n
                    distance += abs(k - i) + abs(l - j)
        return distance
```

```
def process(self):

  print("Enter the start state matrix (use '_' for the blank space):\n")
  start = self.accept()
  print("Enter the goal state matrix:\n")
  goal = self.accept()

  start = Node(start, 0, 0)
  start.fval = self.f(start, goal)
  self.open.append(start)
  print("\n\n")

  while True:
     cur = self.open[0]

     print("\nCurrent Node Selected:")
     for i in cur.data:
        print(" ".join(i))
     print("")

     if self.h(cur.data, goal) == 0:
        print("\nGoal reached!")
        break

     print("\nGenerating child nodes:")
     for child in cur.generate_child():
        child.fval = self.f(child, goal)
        self.open.append(child)

        print("\nChild Node:")
        for i in child.data:
           print(" ".join(i))
        print("H-value:", child.fval - child.level)
        print("Level:", child.level)
        print("F-value:", child.fval)
        print("")

     self.closed.append(cur)
     del self.open[0]
```

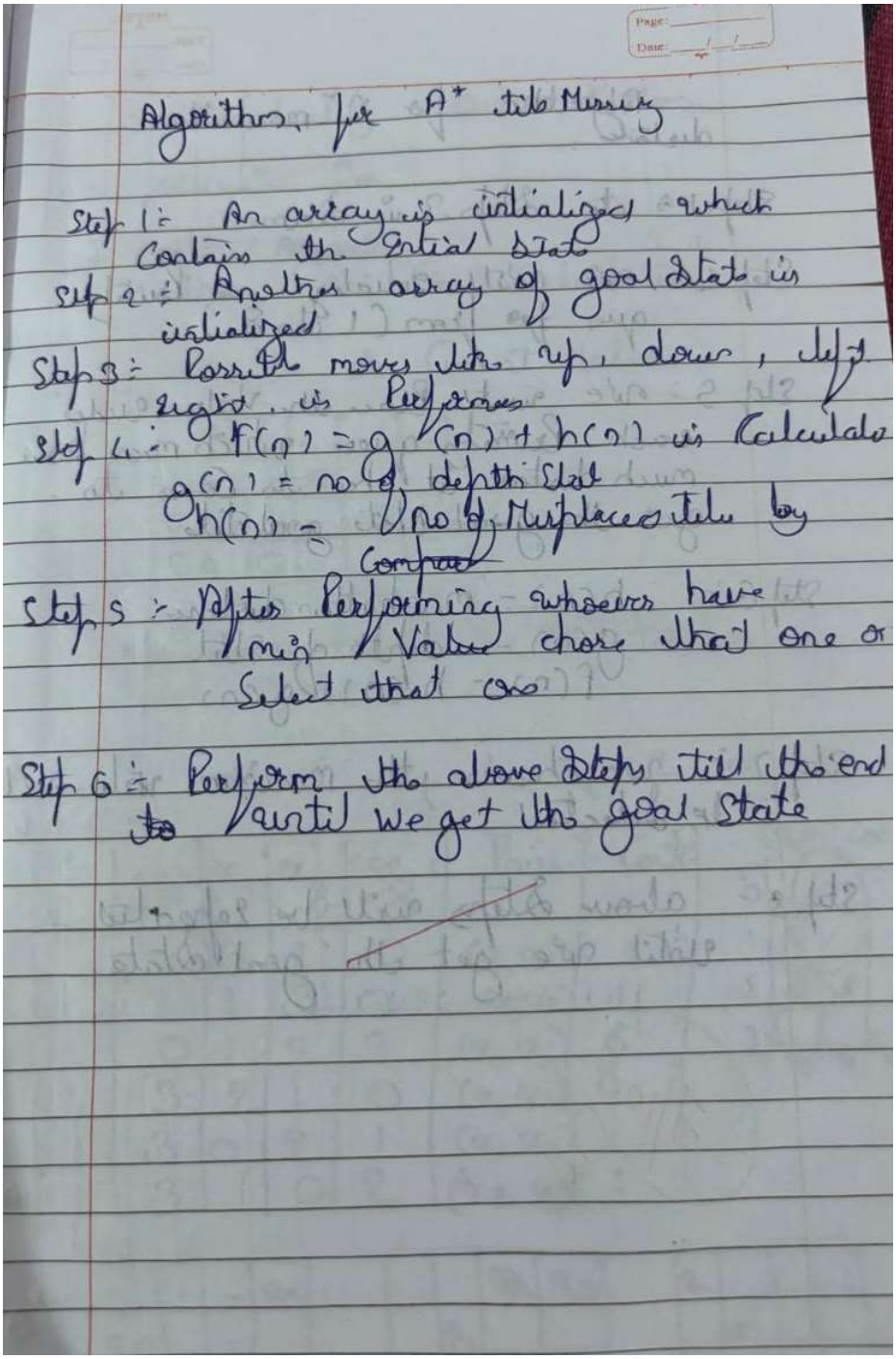
```
        self.open.sort(key=lambda x: x.fval)

puz = Puzzle(3)
puz.process()
```

**Output:**

```
Generating child nodes

Child Node:
1 2 3
8 _ 4
7 6 5
H-value: 0
Level: 6
F-value: 6


Child Node:
_ 2 3
1 8 4
7 6 5
H-value: 2
Level: 6
F-value: 8


Child Node:
1 2 3
7 8 4
_ 6 5
H-value: 2
Level: 6
F-value: 8


Current Node Selected:
1 2 3
8 _ 4
7 6 5


Goal reached!
```

**A* using Misplaced Tiles:**

**Algorithm:**



Algorithm for A* tile Misplaced

Step 1: An array is initialized which contains the initial state

Step 2: Another array of goal state is initialized

Step 3: Possible moves like up, down, left, right is Performed

Step 4: f(n) = g(n) + h(n) is Calculated
g(n) = no of depth state
h(n) = no of Misplaced tiles by Compare

Step 5: After Performing whoever have min value chose that one or Select that one

Step 6: Perform the above steps till the end or until we get the goal state

A* Search.

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

Initial

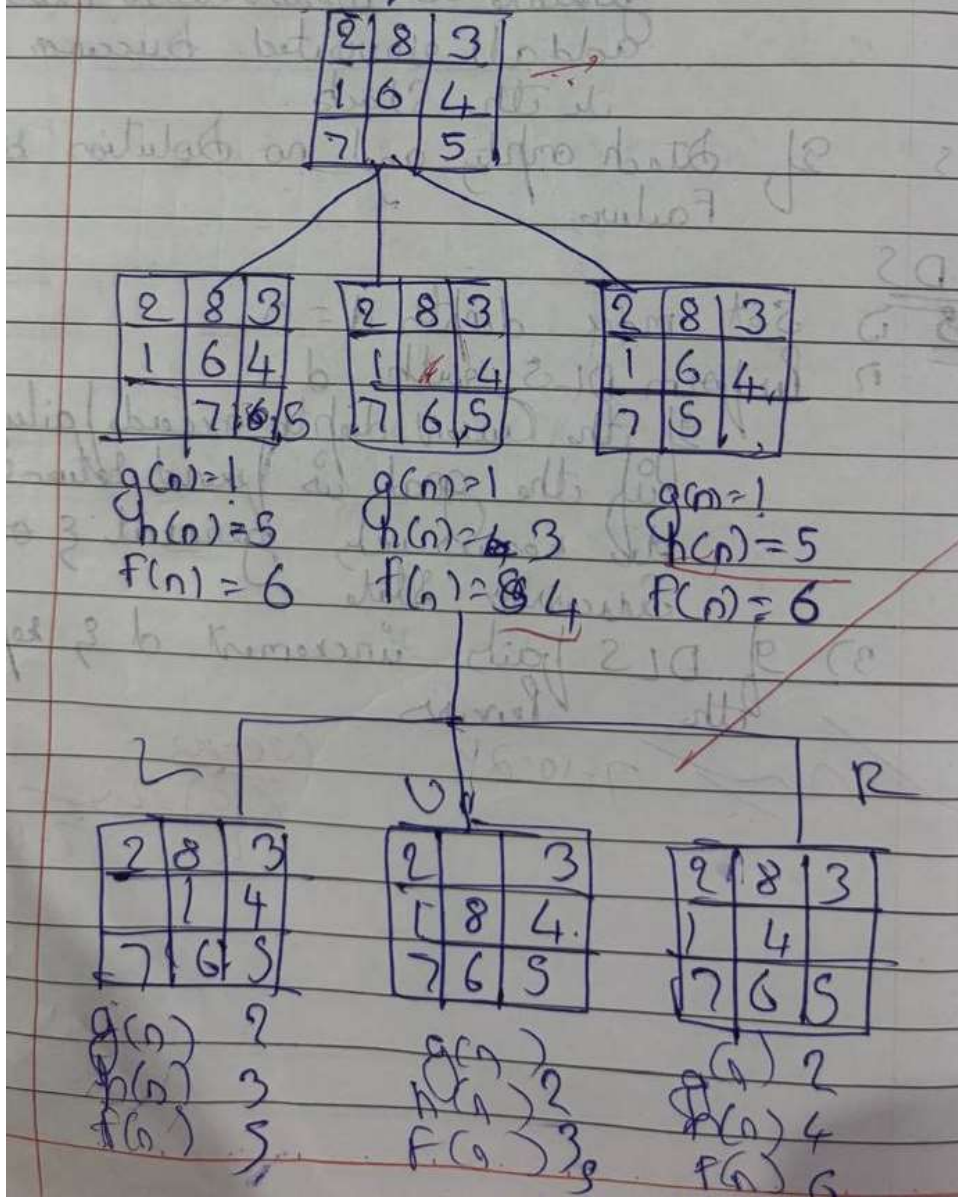
| | | |
|---|---|---|
| 1 | 2 | 3 |
| 8 | | 4 |
| 7 | 6 | 5 |

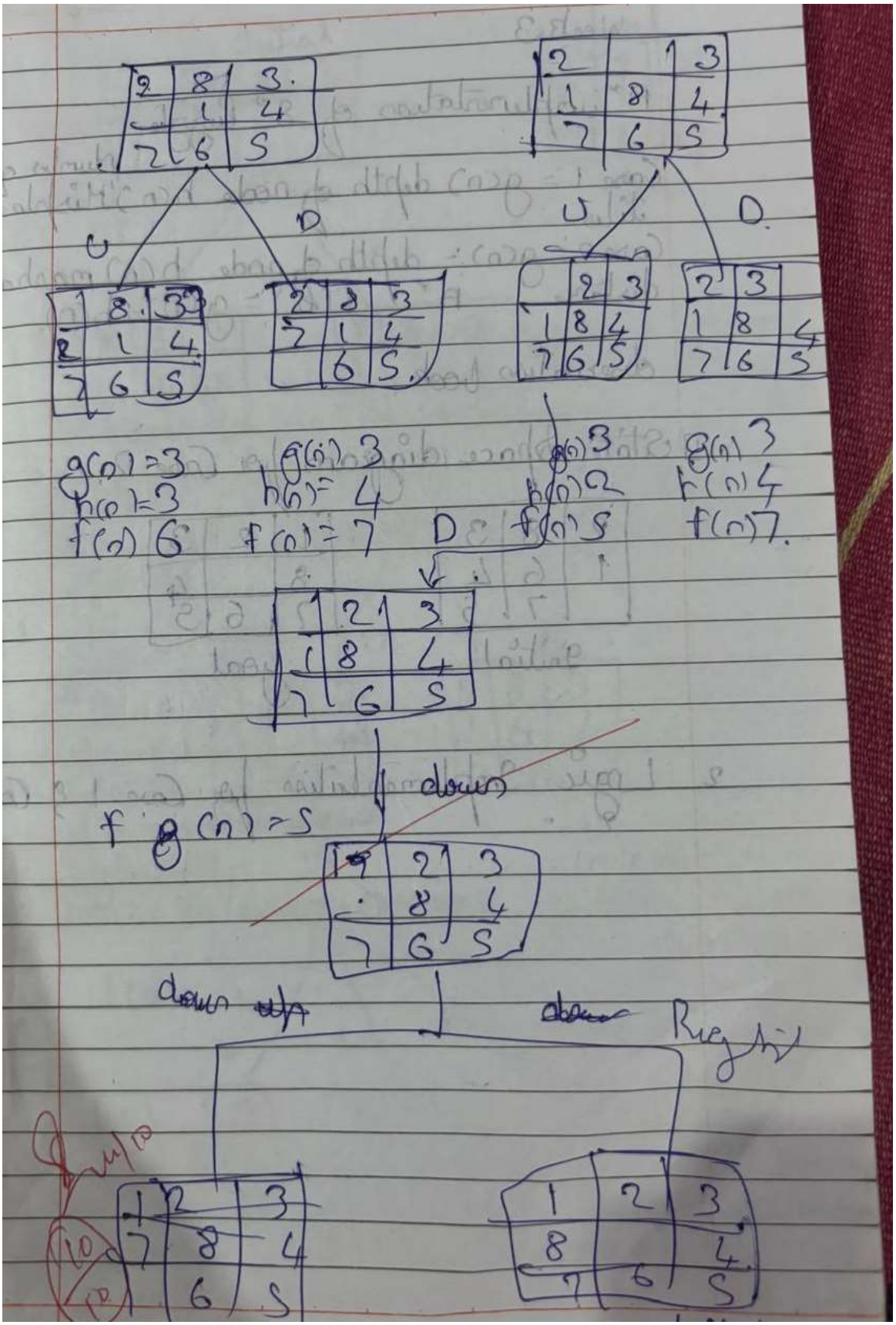
goal

$f(n) = g(n) + h(n)$.

$g(n)$ depth

$h(n)$ No of misplaced tiles




U
D
U
D
g(n)=3
h(n)=3
f(n) 6
g(n) 3
h(n)= 4
f(n)= 7
g(n)3
h(n)2
f(n)5
g(n) 3
h(n)4
f(n)7.
D
down
f g(n)=5
down
down
Right

**Code:**

```
class Node:
    def __init__(self,data,level,fval):
        self.data = data
        self.level = level
        self.fval = fval

    def generate_child(self):
        x,y = self.find(self.data,'_')
        val_list = [[x,y-1],[x,y+1],[x-1,y],[x+1,y]]
        children = []
        for i in val_list:
            child = self.shuffle(self.data,x,y,i[0],i[1])
            if child is not None:
                child_node = Node(child,self.level+1,0)
                children.append(child_node)
        return children

    def shuffle(self,puz,x1,y1,x2,y2):
        if x2 >= 0 and x2 < len(self.data) and y2 >= 0 and y2 < len(self.data):
            temp_puz = []
            temp_puz = self.copy(puz)
            temp = temp_puz[x2][y2]
            temp_puz[x2][y2] = temp_puz[x1][y1]
            temp_puz[x1][y1] = temp
            return temp_puz
        else:
            return None


    def copy(self,root):
        temp = []
        for i in root:
            t = []
            for j in i:
                t.append(j)
            temp.append(t)
        return temp

    def find(self,puz,x):
```

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


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

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

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

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


    def process(self):
      print("Enter the start state matrix (use '_' for the blank space):\n")
      start = self.accept()
      print("Enter the goal state matrix:\n")
      goal = self.accept()
```

```
        start = Node(start, 0, 0)
        start.fval = self.f(start, goal)
        self.open.append(start)
        print("\n\n")

        while True:
            cur = self.open[0]

            print("Current Node Selected:")
            for i in cur.data:
                print(" ".join(i))
            print("H-value:", cur.fval - cur.level)
            print("Level:", cur.level)
            print("F-value:", cur.fval)
            print("")

            if self.h(cur.data, goal) == 0:
                print("Goal reached!")
                break

            print("Generating child nodes:")
            for child in cur.generate_child():
                child.fval = self.f(child, goal)
                self.open.append(child)

                print("Child Node:")
                for i in child.data:
                    print(" ".join(i))
                print("H-value:", child.fval - child.level)
                print("Level:", child.level)
                print("F-value:", child.fval)
                print("")

            self.closed.append(cur)
            del self.open[0]

            self.open.sort(key=lambda x: x.fval)

puz = Puzzle(3)
puz.process()
```

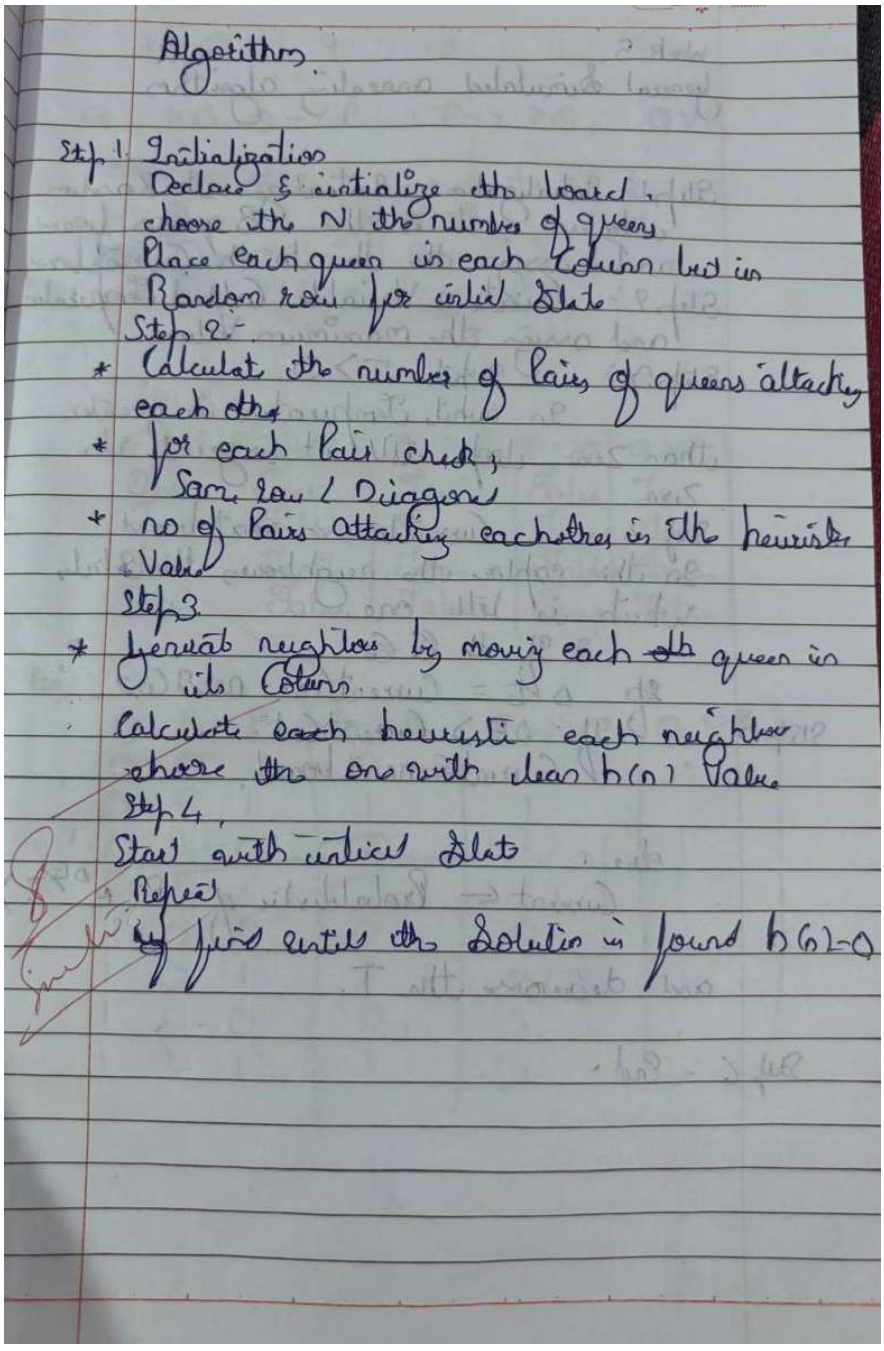
**Output:**

```
Generating child nodes
Child Node:
1 2 3
8 _ 4
7 6 5
H-value: 0
Level: 5
F-value: 5

Child Node:
_ 2 3
1 8 4
7 6 5
H-value: 2
Level: 5
F-value: 7

Child Node:
1 2 3
7 8 4
_ 6 5
H-value: 2
Level: 5
F-value: 7

Current Node Selected:
1 2 3
8 _ 4
7 6 5
H-value: 0
Level: 5
F-value: 5

Goal reached!
```

## Program 4

### Hill Climbing Search to Solve N-Queens Problem:

**Algorithm:**



Algorithm.

Step 1: Initialization
Declare & initialize the board.
choose the N the number of queens
Place each queen in each column but in
Random row for initial State
Step 2:
* Calculat the number of pairs of queens attacking each other
* for each pair check,
Same row / Diagonal
* no of pairs attacking eachother is the heuristic Value
Step 3
* Generate neighbors by moving each ~~th~~ queen in its column.
Calculate each heuristic each neighbor
choose the one with clean h(n) Value
Step 4,
Start with initial State
Repeat
Step 2 until the Solution is found h(n)=0

Hill climbing 4 Queens Problem.

Observation Bo

* Logic / Algorithm.
* Show the Cost Calculation.

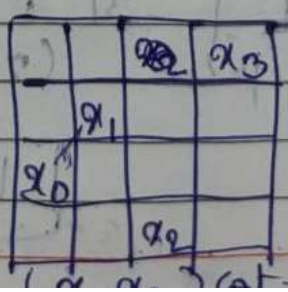
Consider the Current Matrics.

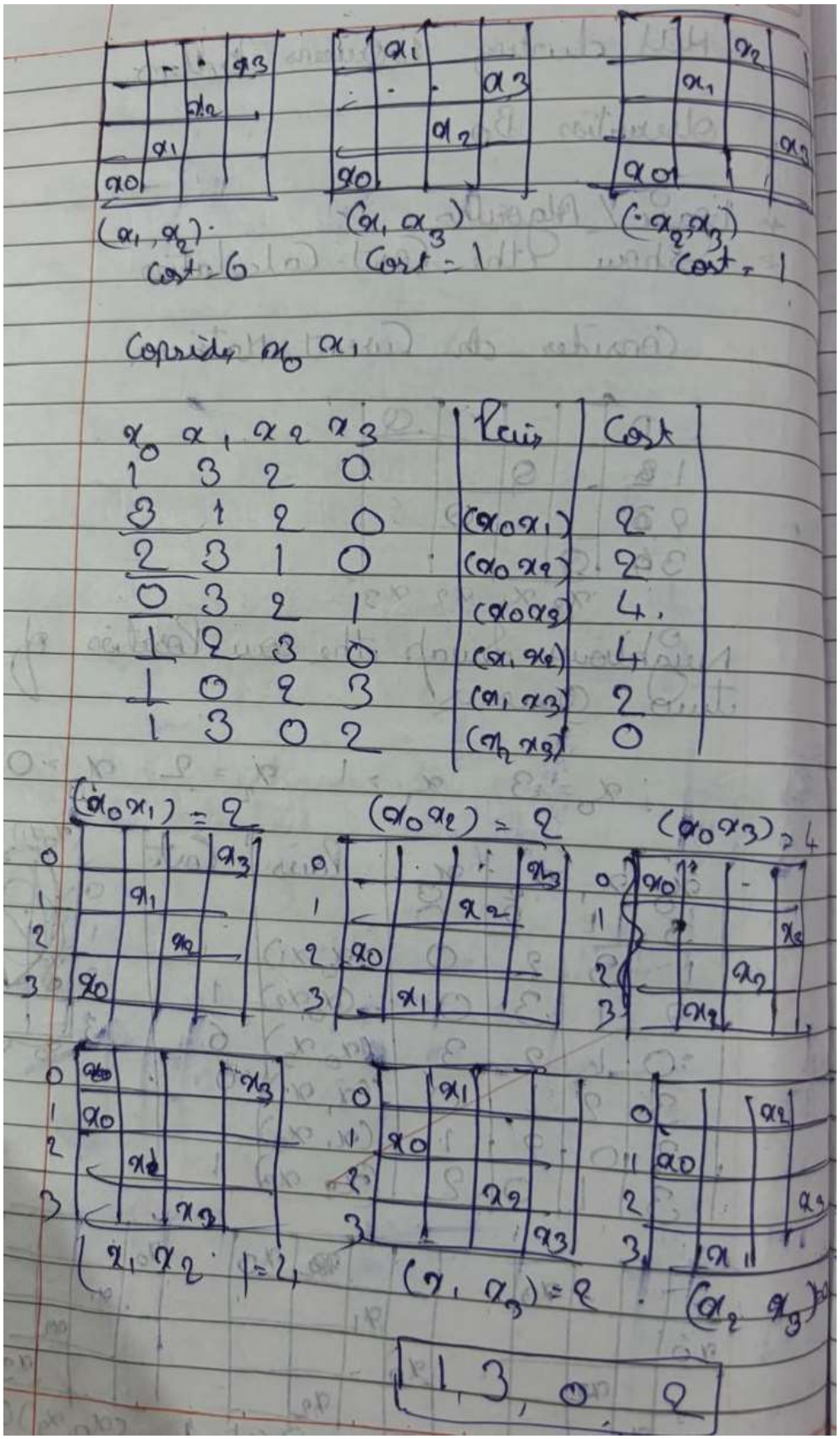

Neighbours Swap the row Position of two Queens

$x_0 = 3 \quad x_1 = 1 \quad x_2 = 2 \quad x_3 = 0$

| $x_0$ | $x_1$ | $x_2$ | $x_3$ | Pairs | Cost |
|---|---|---|---|---|---|
| 3 | 1 | 2 | 0 | | |
| 1 | 3 | 2 | 0 | $(x_0, x_1)$ | 1 |
| 2 | 1 | 3 | 0 | $(x_0, x_2)$ | 1 |
| 0 | 1 | 2 | 3 | $(x_0, x_3)$ | 6 |
| 3 | 2 | 1 | 0 | $(x_1, x_2)$ | 6 |
| 3 | 0 | 2 | 1 | $(x_1, x_3)$ | 1 |
| 3 | 1 | 0 | 2 | $(x_2, x_3)$ | 1 |







(x1, x2)
Cost = 6
(x1, x3)
Cost = 1
(x2, x3)
Cost = 1
Consider x0 x1
x0 x1 x2 x3
1 3 2 0
3 1 2 0
2 3 1 0
0 3 2 1
1 2 3 0
1 0 2 3
1 3 0 2
Pair
Cost
(x0 x1) 2
(x0 x2) 2
(x0 x3) 4
(x1 x2) 4
(x1 x3) 2
(x2 x3) 0
(x0 x1) = 2
(x0 x2) = 2
(x0 x3) = 4
(x1, x2) = 4
(x1, x3) = 2
(x2, x3)
1, 3, 0, 2

**Code:**

```
import random
def calculate_conflicts(board):
    """Calculates the number of pairs of queens attacking each other."""
    conflicts = 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]) == abs(i - j):
                conflicts += 1
    return conflicts
def get_neighbors(board):
    """Generates neighboring boards by swapping two queens."""
    neighbors = []
    n = len(board)
    for i in range(n):
        for j in range(i + 1, n):
            new_board = board[:]
            new_board[i], new_board[j] = new_board[j], new_board[i]
            neighbors.append(new_board)
    return neighbors
def hill_climbing(board):
    """Solves the N-Queens problem using Hill Climbing algorithm."""
    current_conflicts = calculate_conflicts(board)
    print(f"Initial board: {board} with {current_conflicts} conflicts")

    while True:
        neighbors = get_neighbors(board)
        next_board = None
        next_conflicts = current_conflicts
        for neighbor in neighbors:
            conflicts = calculate_conflicts(neighbor)
            if conflicts < next_conflicts:
                next_conflicts = conflicts
                next_board = neighbor

        if next_conflicts >= current_conflicts:
            break

        board = next_board
```

```
        current_conflicts = next_conflicts
        print(f"Intermediate board: {board} with {current_conflicts} conflicts")

    return board, current_conflicts

try:
    n = int(input("Enter the number of queens (size of the board): "))
    if n <= 0:
        raise ValueError("The number of queens must be a positive integer.")

    board = []
    for i in range(n):
        row = int(input(f"Enter the row index for queen {i+1} (0 to {n-1}): "))
        if row < 0 or row >= n:
            raise ValueError("Invalid row index. Must be within the range 0 to n-1.")
        board.append(row)

    solution, conflicts = hill_climbing(board)
    if conflicts == 0:
        print("Solution found:")
        print(solution)
    else:
        print("No solution found, best configuration with conflicts:")
        print(solution, "with", conflicts, "conflicts.")
except ValueError as e:
    print(f"Invalid input: {e}")
```

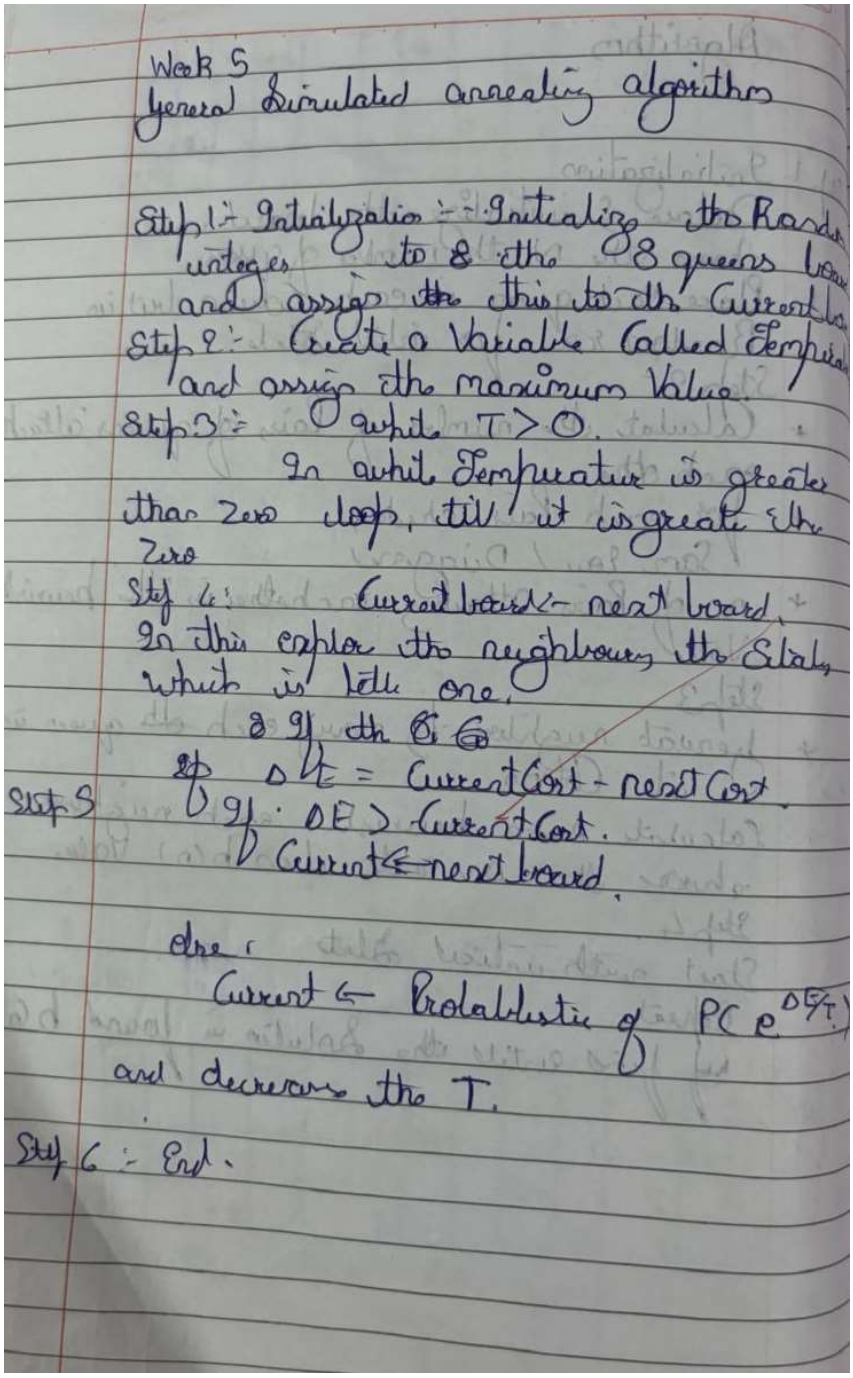
**Output:**

```
Enter the number of queens (size of the board): 4
Enter the row index for queen 1 (0 to 3): 3
Enter the row index for queen 2 (0 to 3): 1
Enter the row index for queen 3 (0 to 3): 2
Enter the row index for queen 4 (0 to 3): 0
Initial board: [3, 1, 2, 0] with 2 conflicts
Intermediate board: [1, 3, 2, 0] with 1 conflicts
Intermediate board: [1, 3, 0, 2] with 0 conflicts
Solution found:
[1, 3, 0, 2]
```

# Program 5

## Simulated Annealing to Solve 8-Queens problem:

**Algorithm:**



Week 5
General Simulated annealing algorithm

Step 1:- Initialization :- Initializing the Random integers to 8 the 8 queens board and assign the this to the Current board.
Step 2:- Create a Variable Called Temperature and assign the maximum Value.
Step 3:- while $T > 0$.
In while Temperature is greater than zero loop, till it is greate the zero
Step 4:- Current board ← next board.
In this explor the neighbours the State which is lelle one.
& If the &
If $\Delta E$ = CurrentCost - nextCost.
Step 5 If $\Delta E >$ CurrentCost.
Current ← next board.

else:
Current ← Probabilistic of $P(e^{\Delta E/T})$
and decreases the T.

Step 6:- End.

**Code:**

```
import random
import math
def count_attacks(queen_positions):
    attack_count = 0
    size = len(queen_positions)
    for i in range(size):
        for j in range(i + 1, size):
            if queen_positions[i] == queen_positions[j]:
                attack_count += 1
            if abs(queen_positions[i] - queen_positions[j]) == abs(i - j):
                attack_count += 1
    return attack_count
def generate_random_move(current_positions):
    new_state = current_positions[:]
    column_to_change = random.randint(0, len(current_positions) - 1)
    new_row_position = random.randint(0, len(current_positions) - 1)
    new_state[column_to_change] = new_row_position
    return new_state
def annealing_search(board_size, initial_configuration):
    current_positions = initial_configuration[:]
    current_attack_count = count_attacks(current_positions)
    temp = 1000
    min_temp = 0.0001
    cooling_factor = 0.99
    step_count = 0
    visited_states = set()
    while temp > min_temp and current_attack_count > 0:
        step_count += 1

        #if step_count % 150 == 0:
            #print("Step", step_count, ": Attacks =", current_attack_count)

        new_positions = generate_random_move(current_positions)

        new_positions_tuple = tuple(new_positions)

        if new_positions_tuple in visited_states:
            continue
```

```
        visited_states.add(new_positions_tuple)

        new_attack_count = count_attacks(new_positions)

        energy_difference = new_attack_count - current_attack_count

        if energy_difference < 0 or random.random() < math.exp(-energy_difference / temp):
            current_positions, current_attack_count = new_positions, new_attack_count
        temp *= cooling_factor
        if current_attack_count == 0:
            print("Solution found after", step_count, "steps!")
            break
    return current_positions, current_attack_count
board_size = int(input("Enter the size of the board (N): "))
initial_input = input("Enter the initial configuration: ")
initial_queen_positions = [int(pos) for pos in initial_input.strip('[]').split(',')]

if len(initial_queen_positions) != board_size:
    print("Error: The initial configuration must contain exactly", board_size, "integers.")
else:
    solution, conflicts = annealing_search(board_size, initial_queen_positions)

    if conflicts == 0:
        print("Solution found!")
        print("Final board configuration:", solution)
    else:
        print("No solution found. Final conflict count:", conflicts)
```

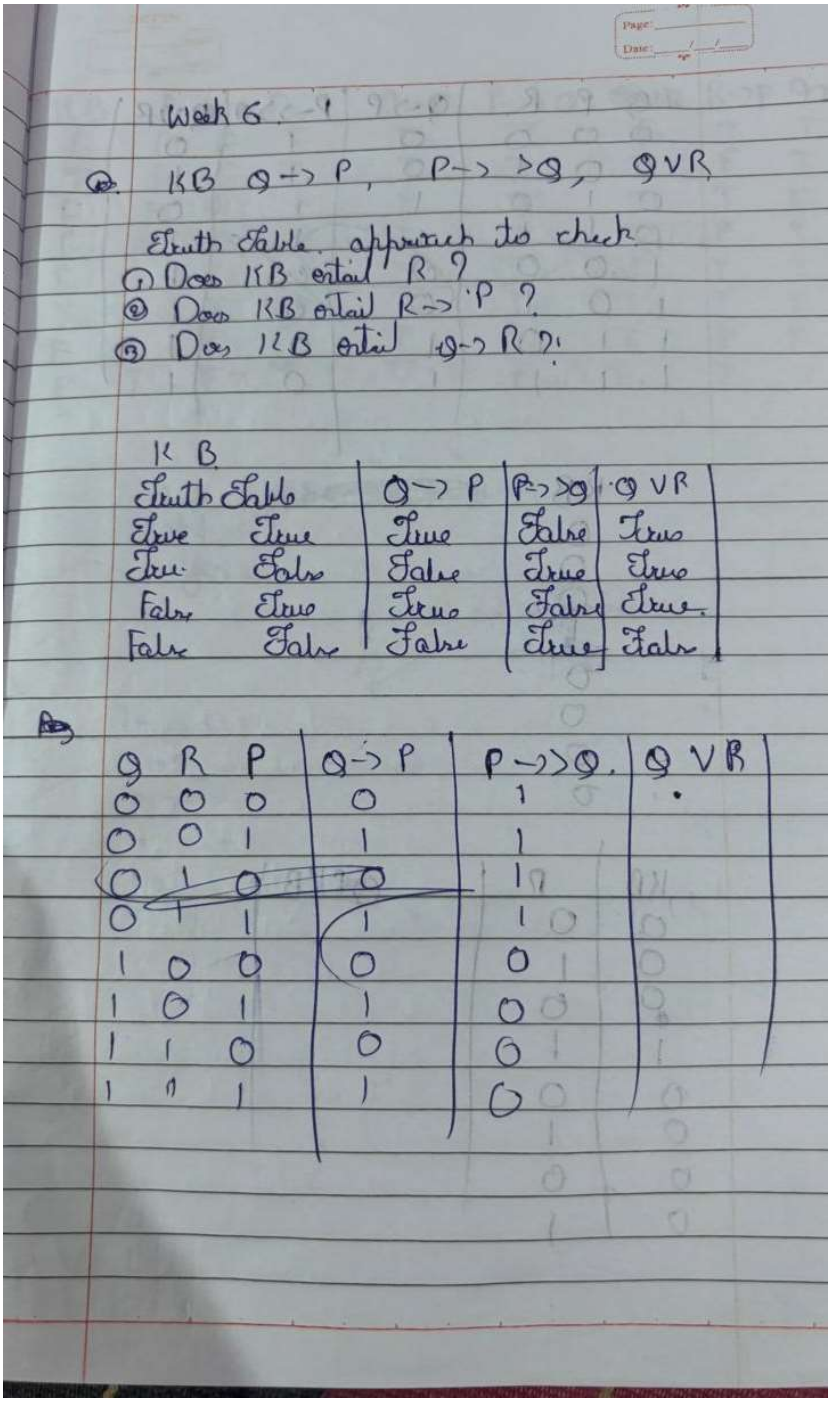
**Output:**

```
Enter the size of the board (N): 8
Enter the initial configuration: 4,5,6,3,4,5,6,5
Solution found after 1093 steps!
Solution found!
Final board configuration: [5, 3, 6, 0, 2, 4, 1, 7]
```

# Program 6

## Knowledge Base using Propositional Logic:

**Algorithm:**



Week 6.

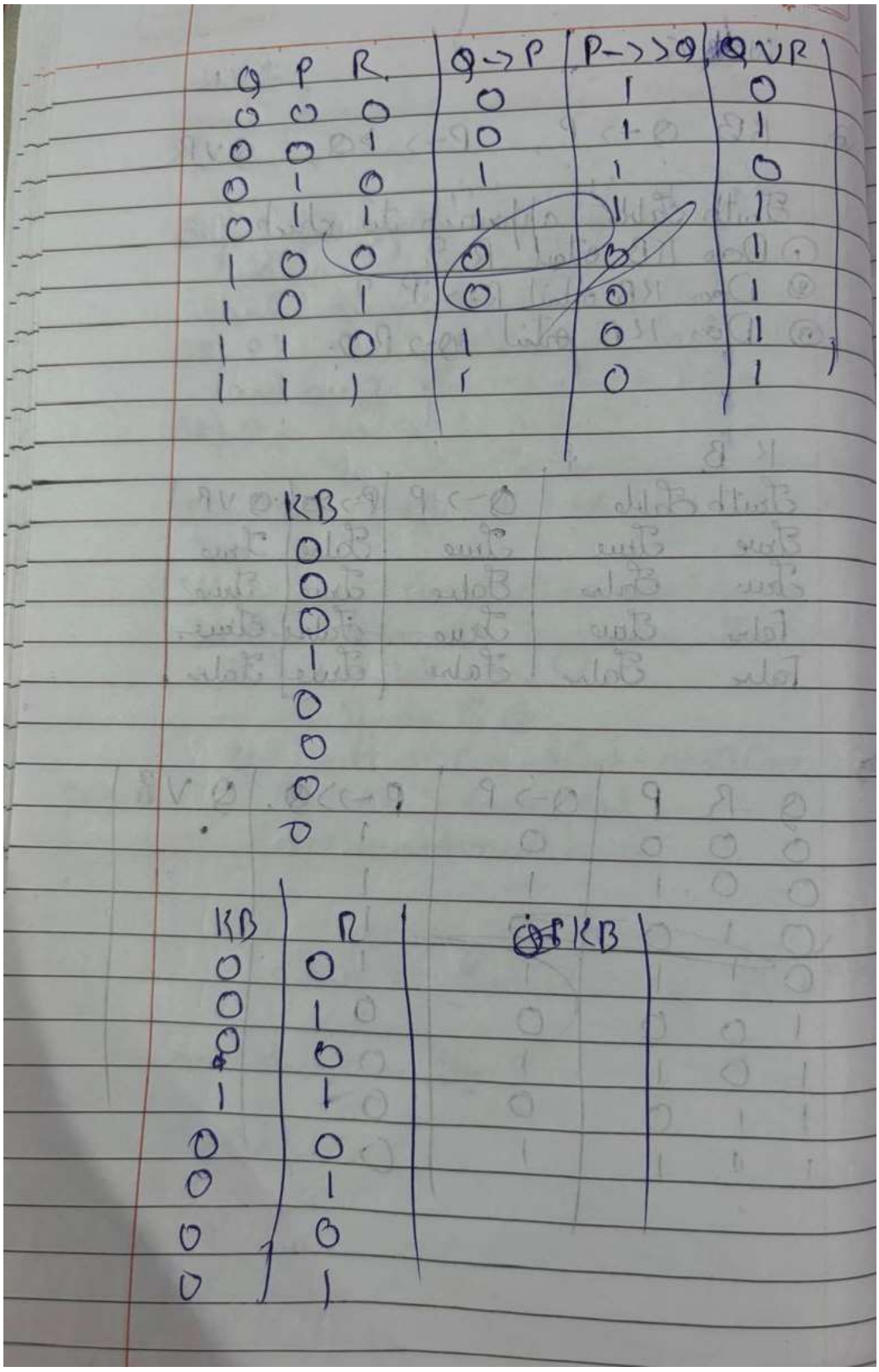
a. KB $Q \rightarrow P$, $P \rightarrow \neg Q$, $Q \vee R$

Truth Table approach to check
① Does KB entail R ?
② Does KB entail $R \rightarrow P$ ?
③ Does KB entail $Q \rightarrow R$ ?

KB

| Truth Table | | $Q \rightarrow P$ | $P \rightarrow \neg Q$ | $Q \vee R$ |
|---|---|---|---|---|
| True | True | True | False | True |
| True | False | False | True | True |
| False | True | True | True | True |
| False | False | True | True | False |

| Q | R | P | $Q \rightarrow P$ | $P \rightarrow \neg Q$ | $Q \vee R$ |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | |
| 0 | 0 | 1 | 1 | 1 | |
| 0 | 1 | 0 | 0 | 1 | |
| 0 | 1 | 1 | 1 | 1 | |
| 1 | 0 | 0 | 0 | 0 | |
| 1 | 0 | 1 | 1 | 0 | |
| 1 | 1 | 0 | 0 | 0 | |
| 1 | 1 | 1 | 1 | 0 | |

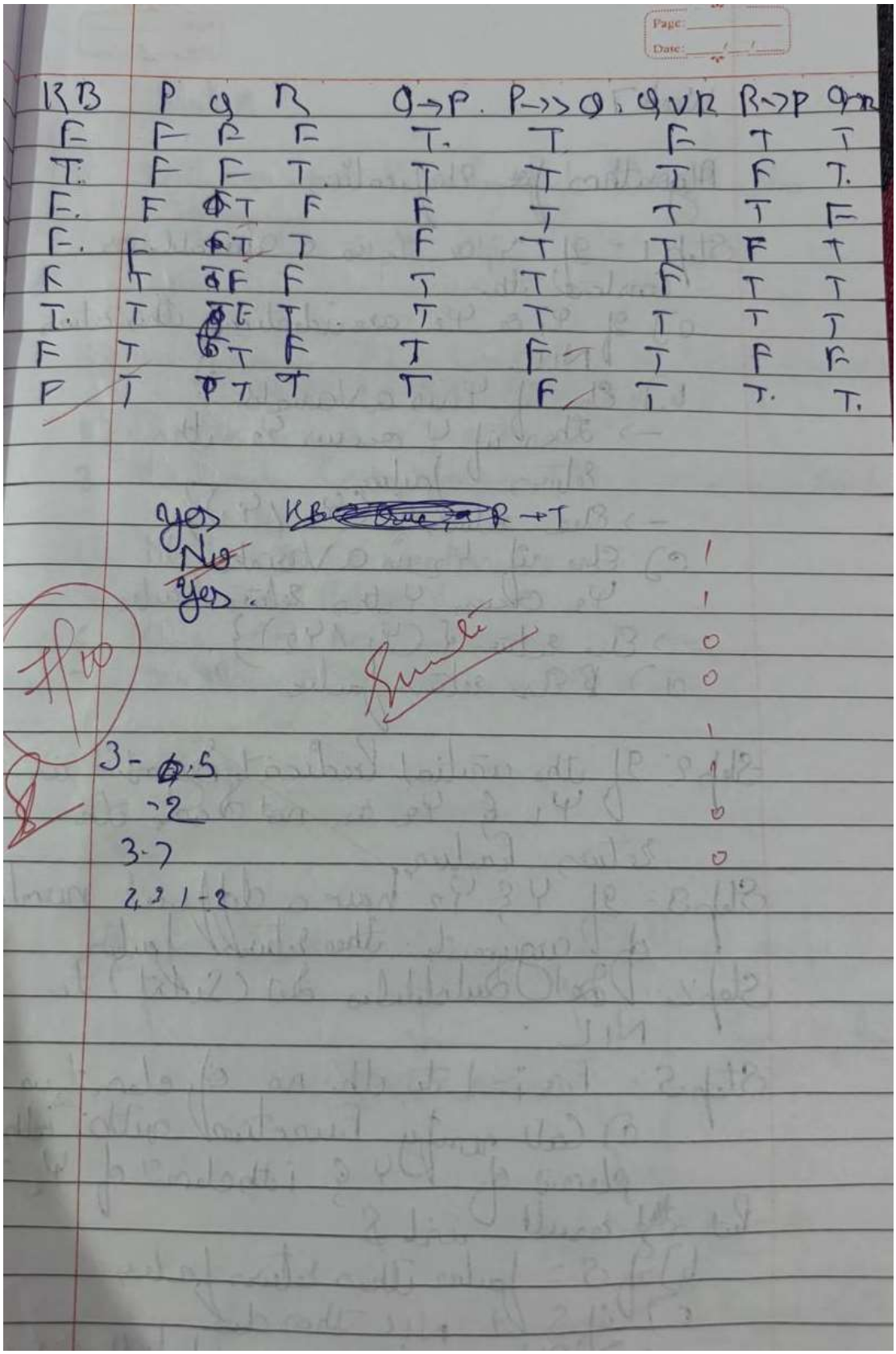


| Q | P | R | Q→P | P→Q | Q∨R |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 0 | 1 |

| KB |
|---|
| 0 |
| 0 |
| 0 |
| 1 |
| 0 |
| 0 |
| 0 |
| 0 |

| KB | R |
|---|---|
| 0 | 0 |
| 0 | 1 |
| 0 | 0 |
| 1 | 1 |
| 0 | 0 |
| 0 | 1 |
| 0 | 0 |
| 0 | 1 |

| ~~(~~KB |
|---|



| KB | P | q | R | q→P | P→q | q∨R | R→P | q |
|---|---|---|---|---|---|---|---|---|
| F | F | F | F | T | T | F | T | T |
| T | F | F | T | T | T | T | F | T |
| F | F | T | F | F | T | T | T | F |
| F | F | T | T | F | T | T | F | T |
| F | T | F | F | T | T | F | T | T |
| T | T | F | T | T | T | T | T | T |
| F | T | T | F | T | F | T | F | F |
| F | T | T | T | T | F | T | T | T |

yes KB → T

No

yes

1
1
0
0
1
1
0
0

3-0.5
-2
3.7
2.3 1-2

**Code:**

```
import random
import itertools
import math

def eval_formula(formula, assignment):
    formula = formula.replace('and', 'and').replace('or', 'or').replace('not', 'not')
    formula = formula.replace('→', ' or not ')  # Implication A → B is equivalent to (not A or B)
    formula = formula.replace('↔', '==')  # Equivalence A ↔ B is equivalent to (A == B)

    env = {var: value for var, value in zip(assignment.keys(), assignment.values())}

    return eval(formula, {}, env)

def generate_initial_state(variables):
    return {var: random.choice([True, False]) for var in variables}

def entails(KB, alpha):
    # Find all unique variables in KB and alpha
    variables = set(''.join([ch for ch in ''.join(KB + [alpha]) if ch.isalpha()]))

    # Generate all possible truth assignments for the variables
    truth_assignments = list(itertools.product([True, False], repeat=len(variables)))
    var_list = list(variables)
    for assignment in truth_assignments:
        # Map the truth assignment to each variable
        assignment_dict = dict(zip(var_list, assignment))

        # Combine all KB formulas with AND and evaluate
        kb_combined = all(eval_formula(formula, assignment_dict) for formula in KB)
        alpha_true = eval_formula(alpha, assignment_dict)

        # If KB is true and alpha is false for any assignment, KB does not entail alpha
        if kb_combined and not alpha_true:
            return False

    # If we reach here, it means KB entails alpha
    return True

# Example Usage →
```

```
KB = ["not Q or P", "not P or (not Q)", "Q or R"]
alpha = "R"

# Check if KB entails alpha
if entails(KB, alpha):
    print("KB entails R")
else:
    print("KB does not entail R")
```

**Output:**

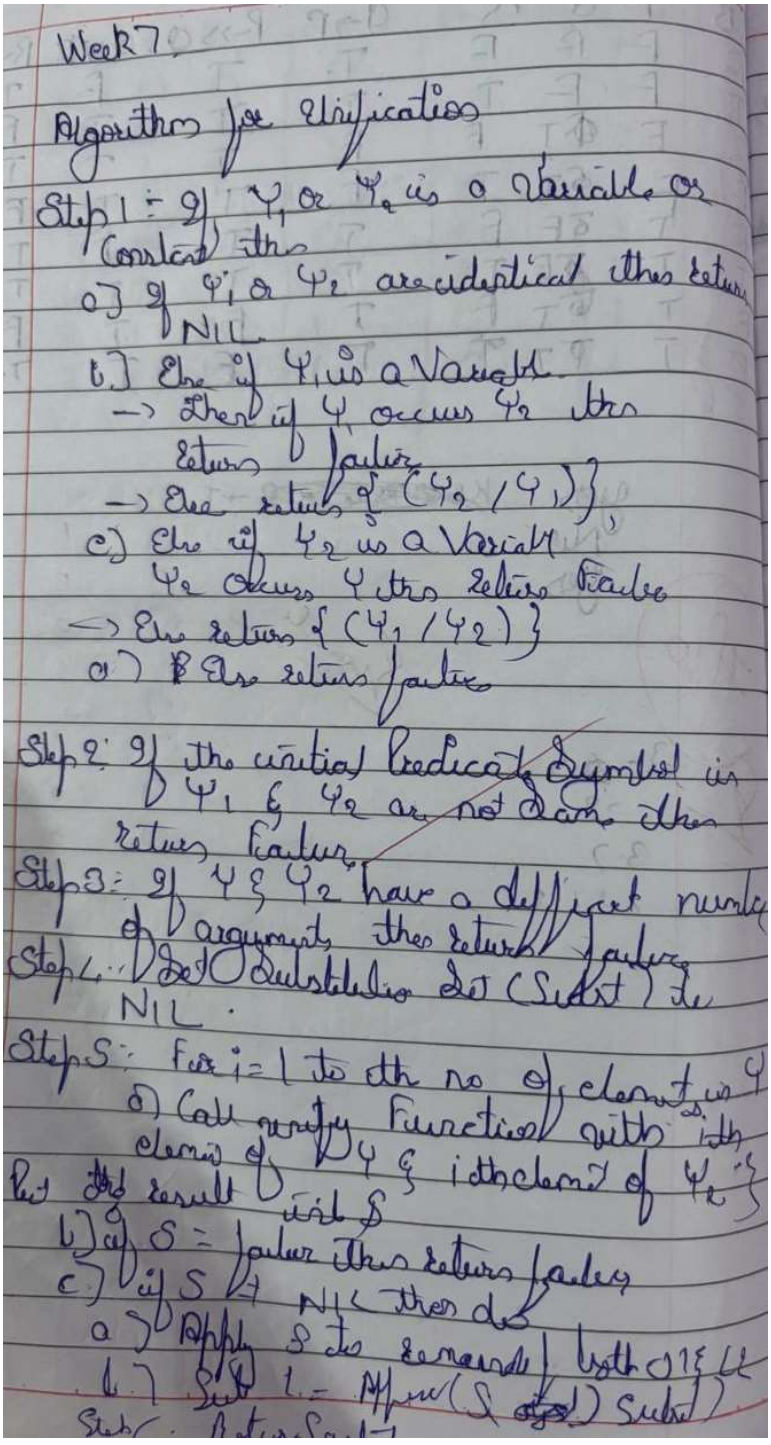
```
KB entails R
```

```
KB does not entail R
```

# Program 7

## Unification in First Order Logic:

**Algorithm:**



Week 7.

Algorithm for Unification

Step 1 :- If $\Psi_1$ or $\Psi_2$ 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.

→ Then if $\Psi_1$ occurs $\Psi_2$ then return failure.

→ Else return $\{(\Psi_2 / \Psi_1)\}$.

c) Else if $\Psi_2$ is a Variable

$\Psi_2$ occurs $\Psi_1$ then return failure

→ Else return $\{(\Psi_1 / \Psi_2)\}$

d) Else return failure.

Step 2 :- If the initial Predicate Symbol in $\Psi_1$ & $\Psi_2$ are not same then return failure.

Step 3 :- If $\Psi_1$ & $\Psi_2$ have a different number of arguments then return failure.

Step 4 :- Set Substitution set (Subst) to NIL.

Step 5 :- For i = 1 to the no of elements in $\Psi_1$

a) Call unify Function with the ith element of $\Psi_1$ & ith element of $\Psi_2$ & put the result into S.

b) If S = failure then return failure

c) If S ≠ NIL then do

a) Apply S to remainder both of $\Psi_1$ & $\Psi_2$

b) Subst = Append (S, Subst)

Step 6 :- Return Subst.

**Code:**

```
def occurs_check(var, term):
    """Checks if a variable occurs in a term."""
    if isinstance(term, str):  # Term is a constant
        return False
    elif isinstance(term, tuple):  # Term is a function (represented as a tuple)
        # Recursively check the function arguments
        if term[0] == var:
            return True
        return any(occurs_check(var, arg) for arg in term[1:])
    return False

def unify(psi1, psi2, subst=None):
    """Unify two terms psi1 and psi2 with the current substitution."""
    if subst is None:
        subst = {}

    # Step 1: If either term is a variable or constant
    if isinstance(psi1, str):  # psi1 is a variable
        if psi1 == psi2:  # Identical variables
            return subst
        elif psi1 in subst:  # psi1 already has a substitution
            return unify(subst[psi1], psi2, subst)
        elif occurs_check(psi1, psi2):  # Occurs check
            return "FAILURE"
        else:
            subst[psi1] = psi2  # Create a new substitution
            return subst

    elif isinstance(psi2, str):  # psi2 is a variable
        if psi2 == psi1:  # Identical variables
            return subst
        elif psi2 in subst:  # psi2 already has a substitution
            return unify(psi1, subst[psi2], subst)
        elif occurs_check(psi2, psi1):  # Occurs check
            return "FAILURE"
        else:
            subst[psi2] = psi1  # Create a new substitution
            return subst
```

```
    # Step 2: Check if the initial predicate symbols match
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        if psi1[0] != psi2[0]:  # Predicate symbols don't match
            return "FAILURE"

    # Step 3: Check if they have the same number of arguments
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        if len(psi1) != len(psi2):  # Different number of arguments
            return "FAILURE"

    # Step 4: Initialize the substitution set (already initialized as `subst`)

    # Step 5: Iterate through the arguments of psi1 and psi2
    if isinstance(psi1, tuple) and isinstance(psi2, tuple):
        for arg1, arg2 in zip(psi1[1:], psi2[1:]):
            # Recursively unify the arguments
            result = unify(arg1, arg2, subst)
            if result == "FAILURE":
                return "FAILURE"
            elif result != subst:
                subst = result  # Update the substitution set

    # Step 6: Return the final substitution set
    return subst

# Helper function to display the substitution in a readable format
def print_substitution(subst):
    if subst == "FAILURE":
        print("FAILURE")
    else:
        for var, val in subst.items():
            print(f"{var} -> {val}")

# Helper function to parse the user input into the required format
def parse_input(input_str):
    """Parse the input string into a tuple representing the term."""
    input_str = input_str.strip()
    if '(' in input_str and ')' in input_str:
        # Extract the predicate and arguments
        predicate, args = input_str.split('(', 1)
        args = args.rstrip(')').split(',')
```

```
        args = [parse_input(arg.strip()) if '(' in arg else arg.strip() for arg in args]
        return (predicate.strip(), *args)
    else:
        # Single term, assume it's a constant or a variable
        return input_str.strip()

# Main function to handle user input and unification
def main():
    print("Please enter the first term (e.g., f(X, g(Y))):")
    term1 = input().strip()

    print("Please enter the second term (e.g., f(a, g(b))):")
    term2 = input().strip()

    # Parse the user input into structured terms
    psi1 = parse_input(term1)
    psi2 = parse_input(term2)

    # Unify the terms and print the result
    substitution = unify(psi1, psi2)
    print_substitution(substitution)

# Run the main function
if __name__ == "__main__":
    main()
```

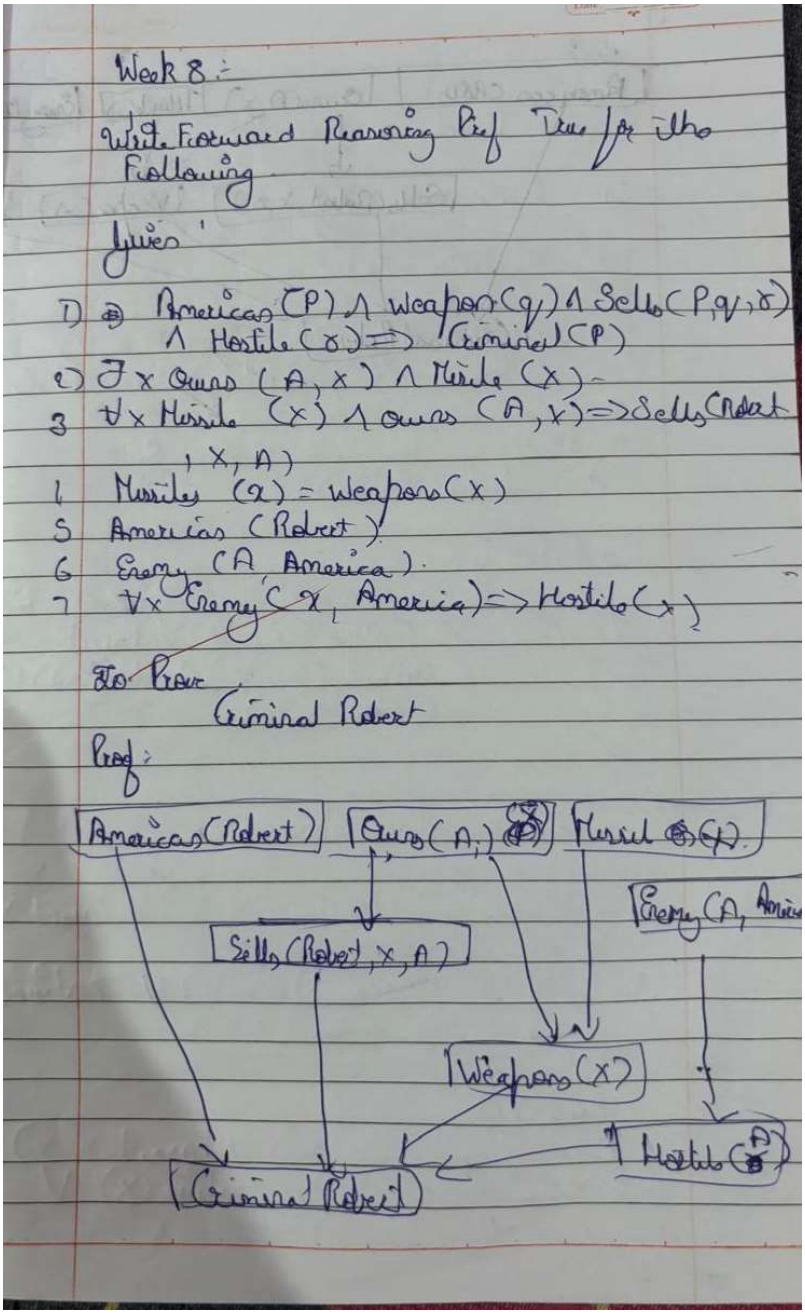
**Output:**

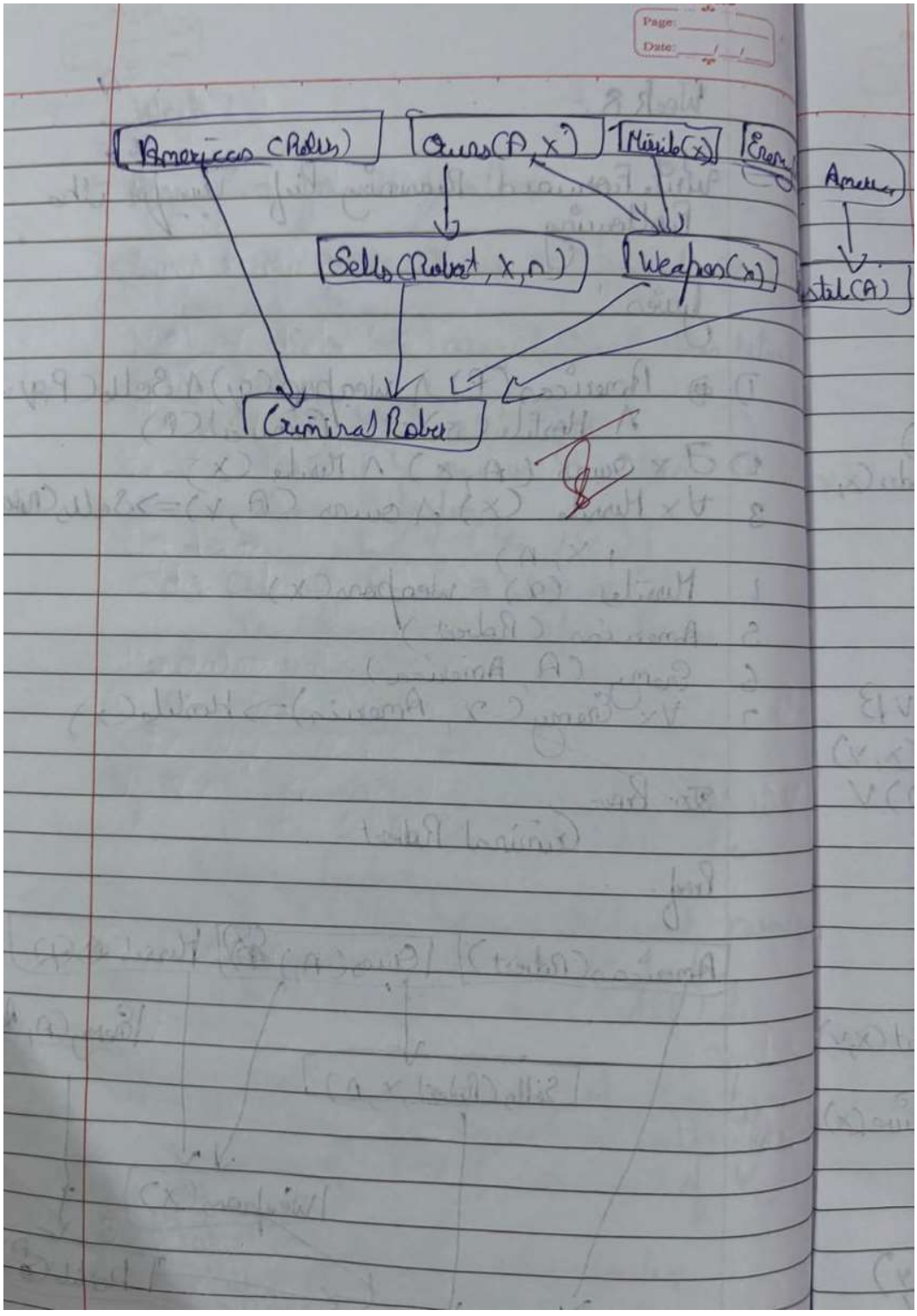
```
Please enter the first term (e.g., f(X, g(Y))):
American(robert,x)
Please enter the second term (e.g., f(a, g(b))):
American(y,enemy)
robert -> y
x -> enemy
```

## Program 8

**Knowledge Base consisting of First Order Logic Statements and proof using Forward Reasoning:**

**Algorithm:**




Criminal Robot

**Code:**

```
class ForwardChaining:
    def __init__(self, facts, rules):
        """
        Initialize the Forward Chaining algorithm with facts and rules.
        :param facts: Set of known facts (initial facts).
        :param rules: List of rules where each rule is a tuple (premise, conclusion).
        """
        self.facts = set(facts)
        self.rules = rules
        self.inferred_facts = set(facts)  # Set of facts derived during the process

    def apply_rule(self, rule):
        """
        Applies a rule to derive new facts from existing facts.
        :param rule: A rule represented as (premise, conclusion).
        :return: True if a new fact is derived, False otherwise.
        """
        premise, conclusion = rule
        premise_facts = set(premise.split(','))  # Split the premise into individual facts
        # Check if the premise of the rule is fully satisfied by current facts
        if premise_facts.issubset(self.facts):  # Ensure all premises are in facts
            if conclusion not in self.facts:
                print(f"Inferred new fact: {conclusion}")
                self.facts.add(conclusion)  # Add the conclusion to the set of facts
                return True
        return False

    def forward_chaining(self):
        """
        Applies forward chaining to derive new facts until no more facts can be derived.
        """
        new_inference = True
        while new_inference:
            new_inference = False
            # Go through all the rules and try to apply them
            for rule in self.rules:
                if self.apply_rule(rule):
                    new_inference = True
            if new_inference:
```

```
            print(f"Current facts: {self.facts}")
        print("Forward Chaining completed.")

    def is_goal_reached(self, goal):
        """
        Checks if the goal has been reached (i.e., if the goal is in the facts).
        :param goal: The goal fact to check for.
        :return: True if the goal is in the facts, otherwise False.
        """
        return goal in self.facts


def main():
    print("Forward Chaining System")

    # Define the initial facts
    facts = {
        "american(p)",
        "weapon(q)",
        "sells(p,q,r)",
        "hostile(r)",
        "american(robert)",
        "enemy(a, america)"
    }

    # Define the rules (premise -> conclusion)
    rules = [
        ("american(p),weapon(q),sells(p,q,r),hostile(r)", "criminal(p)"),  # Rule 1
        ("owns(a,x),missile(x)", "sells(robert,x,a)"),  # Rule 2
        ("missile(x)", "weapon(x)"),  # Rule 3
        ("enemy(a,america)", "hostile(a)")  # Rule 4
    ]

    # Create an instance of ForwardChaining with the facts and rules
    fc = ForwardChaining(facts, rules)

    # Perform forward chaining to infer new facts
    fc.forward_chaining()

    # Define the goal (fact you want to check)
    goal = "criminal(robert)"
```

```
    # Check if the goal is reached
    if fc.is_goal_reached(goal):
        print(f"The goal '{goal}' is reached!")
    else:
        print(f"The goal '{goal}' is reached.")

# Run the main function
if __name__ == "__main__":
    main()
```

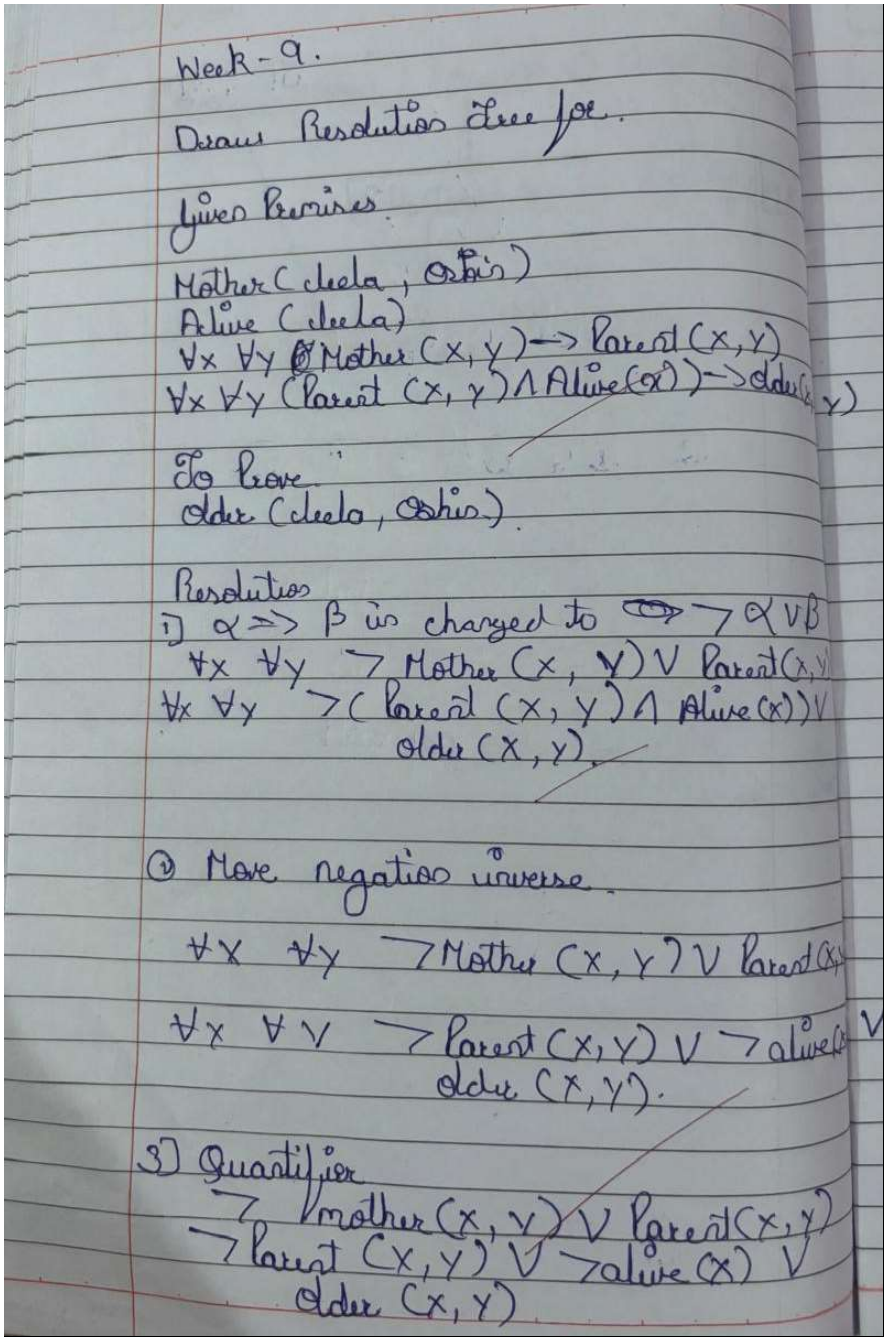
**Output:**

```
Forward Chaining System
Forward Chaining completed.
The goal 'criminal(robert)' is reached.
```

## Program 9

**Knowledge Base consisting of First Order Logic Statements and proof using Resolution:**

**Algorithm:**



Week-9.

Draw Resolution Tree for.

Given Premises

Mother (Leela, Bhis)

Alive (Leela)

∀x ∀y Mother (x, y) → Parent (x, y)

∀x ∀y (Parent (x, y) ∧ Alive (x)) → older (x, y)

To Prove

older (Leela, Bhis)

Resolution

1) α ⇒ β is changed to ¬α ∨ β

∀x ∀y ¬ Mother (x, y) ∨ Parent (x, y)

∀x ∀y ¬ (Parent (x, y) ∧ Alive (x)) ∨ older (x, y).

2) Move negation inverse.

∀x ∀y ¬Mother (x, y) ∨ Parent (x, y)

∀x ∀y ¬Parent (x, y) ∨ ¬alive (x) ∨ older (x, y).

3) Quantifier

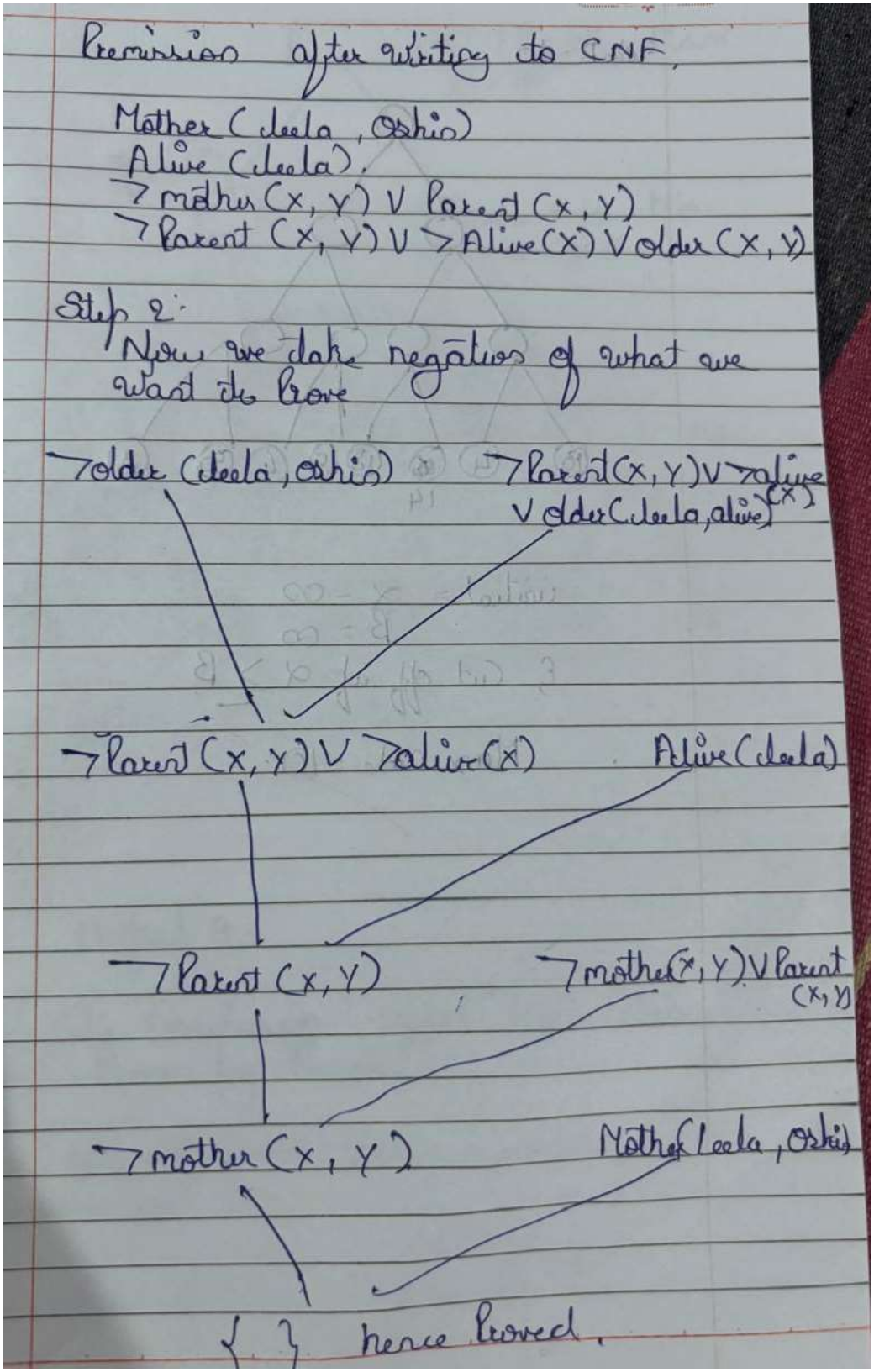
¬ mother (x, y) ∨ Parent (x, y)

¬ Parent (x, y) ∨ ¬ alive (x) ∨ older (x, y)

Premission after writing to CNF.

Mother (leela, Oshin)
Alive (leela).
¬ mother (x, y) V Parent (x, y)
¬ Parent (x, y) V ¬ Alive (x) V Older (x, y)

Step 2:
Now we take negation of what we want to prove



**Code:**

```
import time
start_time = time.time()
import re
import itertools
import collections
import copy
import queue

p=open("input.txt","r")
data=list()
data1= p.readlines()
count=0

n=int(data1[0])
queries=list()
for i in range(1,n+1):
    queries.append(data1[i].rstrip())
k=int(data1[n+1])
kbbefore=list()

def CNF(sentence):
    temp=re.split("=>",sentence)
    temp1=temp[0].split('&')
    for i in range(0,len(temp1)):
        if temp1[i][0]=='~':
            temp1[i]=temp1[i][1:]
        else:
            temp1[i]='~'+temp1[i]
    temp2='|'.join(temp1)
    temp2=temp2+'|'+temp[1]
    return temp2
variableArray = list("abcdefghijklmnopqrstuvwxyz")
variableArray2 = []
variableArray3 = []
variableArray5 = []
variableArray6 = []
for eachCombination in itertools.permutations(variableArray, 2):
    variableArray2.append(eachCombination[0] + eachCombination[1])
for eachCombination in itertools.permutations(variableArray, 3):
```

```
    variableArray3.append(eachCombination[0] + eachCombination[1] + eachCombination[2])
for eachCombination in itertools.permutations(variableArray, 4):
    variableArray5.append(eachCombination[0] + eachCombination[1] + eachCombination[2]+
eachCombination[3])
for eachCombination in itertools.permutations(variableArray, 5):
    variableArray6.append(eachCombination[0] + eachCombination[1] + eachCombination[2] +
eachCombination[3] + eachCombination[4])
variableArray = variableArray + variableArray2 + variableArray3 + variableArray5 + variableArray6
capitalVariables = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
number=0

def standardizationnew(sentence):
    newsentence=list(sentence)
    i=0
    global number
    variables=collections.OrderedDict()
    positionsofvariable=collections.OrderedDict()
    lengthofsentence=len(sentence)
    for i in range(0,lengthofsentence-1):
        if(newsentence[i]==',' or newsentence[i]=='('):
            if newsentence[i+1] not in capitalVariables:
                substitution=variables.get(newsentence[i+1])
                positionsofvariable[i+1]=i+1
                if not substitution :
                    variables[newsentence[i+1]]=variableArray[number]
                    newsentence[i+1]=variableArray[number]
                    number+=1
                else:
                    newsentence[i+1]=substitution
    return  "".join(newsentence)

def insidestandardizationnew(sentence):
    lengthofsentence=len(sentence)
    newsentence=sentence
    variables=collections.OrderedDict()
    positionsofvariable=collections.OrderedDict()
    global number
    i=0
    while i <=len(newsentence)-1 :
        if(newsentence[i]==',' or newsentence[i]=='('):
            if newsentence[i+1] not in capitalVariables:
```

```
            j=i+1
            while(newsentence[j]!=',' and newsentence[j]!=')' ):
                j+=1
            substitution=variables.get(newsentence[i+1:j])
            if not substitution :
                variables[newsentence[i+1:j]]=variableArray[number]
                newsentence=newsentence[:i+1]+variableArray[number]+newsentence[j:]
                i=i+len(variableArray[number])
                number+=1
            else:
                newsentence=newsentence[:i+1]+substitution+newsentence[j:]
                i=i+len(substitution)
        i+=1
    return newsentence

def replace(sentence,theta):
    lengthofsentence=len(sentence)
    newsentence=sentence
    i=0
    while i <=len(newsentence)-1 :
        if(newsentence[i]==',' or newsentence[i]=='('):
            if newsentence[i+1] not in capitalVariables:
                j=i+1
                while(newsentence[j]!=',' and newsentence[j]!=')' ):
                    j+=1
                nstemp=newsentence[i+1:j]
                substitution=theta.get(nstemp)
                if substitution :
                    newsentence=newsentence[:i+1]+substitution+newsentence[j:]
                    i=i+len(substitution)
        i+=1
    return newsentence
repeatedsentencecheck=collections.OrderedDict()

def insidekbcheck(sentence):
    lengthofsentence=len(sentence)
    newsentence=pattern.split(sentence)
    newsentence.sort()
    newsentence="|".join(newsentence)
    global repeatedsentencecheck
    i=0
```

```
    while i <=len(newsentence)-1 :
        if(newsentence[i]==',' or newsentence[i]=='('):
            if newsentence[i+1] not in capitalVariables:
                j=i+1
                while(newsentence[j]!=',' and newsentence[j]!=')' ):
                    j+=1
                newsentence=newsentence[:i+1]+'x'+newsentence[j:]
        i+=1
    repeatflag=repeatedsentencecheck.get(newsentence)
    if repeatflag :
        return True
    repeatedsentencecheck[newsentence]=1
    return False

for i in range(n+2,n+2+k):
     data1[i]=data1[i].replace(" ","")
     if "=>" in data1[i]:
        data1[i]=data1[i].replace(" ","")
        sentencetemp=CNF(data1[i].rstrip())
        kbbefore.append(sentencetemp)
     else:
        kbbefore.append(data1[i].rstrip())
for i in range(0,k):
    kbbefore[i]=kbbefore[i].replace(" ","")

kb={}
pattern=re.compile("\||&|=>") #we can remove the '\|" to speed up as 'OR' doesnt come in the KB
pattern1=re.compile("[(,]")
for i in range(0,k):
    kbbefore[i]=standardizationnew(kbbefore[i])
    temp=pattern.split(kbbefore[i])
    lenoftemp=len(temp)
    for j in range(0,lenoftemp):
        clause=temp[j]
        clause=clause[:-1]
        predicate=pattern1.split(clause)
        argumentlist=predicate[1:]
        lengthofpredicate=len(predicate)-1
        if predicate[0] in kb:
            if lengthofpredicate in kb[predicate[0]]:
                kb[predicate[0]][lengthofpredicate].append([kbbefore[i],temp,j,predicate[1:]])
```

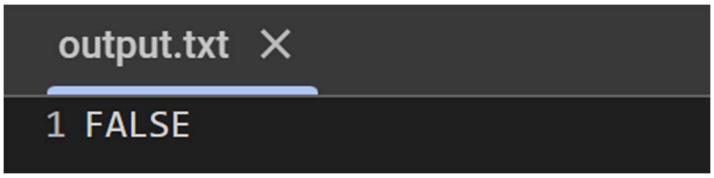
```
            else:
                kb[predicate[0]][lengthofpredicate]=[kbbefore[i],temp,j,predicate[1:]]
        else:
            kb[predicate[0]]={lengthofpredicate:[[kbbefore[i],temp,j,predicate[1:]]]}

for qi in range(0,n):
    queries[qi]=standardizationnew(queries[qi])

def substituevalue(paramArray, x, y):
    for index, eachVal in enumerate(paramArray):
        if eachVal == x:
            paramArray[index] = y
    return paramArray

def unificiation(arglist1,arglist2):
    theta = collections.OrderedDict()
    for i in range(len(arglist1)):
        if arglist1[i] != arglist2[i] and (arglist1[i][0] in capitalVariables) and (arglist2[i][0] in
capitalVariables):
            return []
        elif arglist1[i] == arglist2[i] and (arglist1[i][0] in capitalVariables) and (arglist2[i][0] in
capitalVariables):
            if arglist1[i] not in theta.keys():
                theta[arglist1[i]] = arglist2[i]
        elif (arglist1[i][0] in capitalVariables) and not (arglist2[i][0] in capitalVariables):
            if arglist2[i] not in theta.keys():
                theta[arglist2[i]] = arglist1[i]
                arglist2 = substituevalue(arglist2, arglist2[i], arglist1[i])
        elif not (arglist1[i][0] in capitalVariables) and (arglist2[i][0] in capitalVariables):
            if arglist1[i] not in theta.keys():
                theta[arglist1[i]] = arglist2[i]
                arglist1 = substituevalue(arglist1, arglist1[i], arglist2[i])
        elif not (arglist1[i][0] in capitalVariables) and not (arglist2[i][0] in capitalVariables):
            if arglist1[i] not in theta.keys():
                theta[arglist1[i]] = arglist2[i]
                arglist1 = substituevalue(arglist1, arglist1[i], arglist2[i])
            else:
                argval=theta[arglist1[i]]
                theta[arglist2[i]]=argval
                arglist2 = substituevalue(arglist2, arglist2[i], argval)
    return [arglist1,arglist2,theta]
```

```
def resolution():
    global repeatedsentencecheck
    answer=list()
    qrno=0
    for qr in queries:
        qrno+=1
        repeatedsentencecheck.clear()
        q=queue.Queue()
        query_start=time.time()
        kbquery=copy.deepcopy(kb)
        ans=qr
        if qr[0]=='~':
            ans=qr[1:]
        else:
            ans='~'+qr
        q.put(ans)
        label:outerloop
        currentanswer="FALSE"
        counter=0
        while True:
            counter+=1
            if q.empty():
                break
            ans=q.get()
            label:outerloop1
            ansclauses=pattern.split(ans)
            lenansclauses=len(ansclauses)
            flagmatchedwithkb=0
            innermostflag=0
            for ac in range(0,lenansclauses):
                insidekbflag=0
                ansclausestruncated=ansclauses[ac][:-1]
                ansclausespredicate=pattern1.split(ansclausestruncated)
                lenansclausespredicate=len(ansclausespredicate)-1
                if ansclausespredicate[0][0]=='~':
                    anspredicatenegated=ansclausespredicate[0][1:]
                else:
                    anspredicatenegated="~"+ansclausespredicate[0]
                x=kbquery.get(anspredicatenegated,{}).get(lenansclausespredicate)
                if not x:
```

```
            continue
        else:
            lenofx=len(x)
            for numofpred in range(0,lenofx):
                insidekbflag=0
                putinsideq=0
                sentenceselected=x[numofpred]

thetalist=unificiation(copy.deepcopy(sentenceselected[3]),copy.deepcopy(ansclausespredicate[1:]))
                if(len(thetalist)!=0):
                    for key in thetalist[2]:
                        tl=thetalist[2][key]
                        tl2=thetalist[2].get(tl)
                        if tl2:
                            thetalist[2][key]=tl2
                    flagmatchedwithkb=1
                    notincludedindex=sentenceselected[2]
                    senclause=copy.deepcopy(sentenceselected[1])
                    mergepart1=""
                    del senclause[notincludedindex]
                    ansclauseleft=copy.deepcopy(ansclauses)
                    del ansclauseleft[ac]
                    for am in range(0,len(senclause)):
                        senclause[am]=replace(senclause[am],thetalist[2])
                        mergepart1=mergepart1+senclause[am]+'|'
                    for remain in range(0,len(ansclauseleft)):
                        listansclauseleft=ansclauseleft[remain]
                        ansclauseleft[remain]=replace(listansclauseleft,thetalist[2])
                        if ansclauseleft[remain] not in senclause:
                            mergepart1=mergepart1+ansclauseleft[remain]+'|'
                    mergepart1=mergepart1[:-1]
                    if mergepart1=="":
                      currentanswer="TRUE"
                      break
                    ckbflag=insidekbcheck(mergepart1)
                    if not ckbflag:
                            mergepart1=insidestandardizationnew(mergepart1)
                            ans=mergepart1
                            temp=pattern.split(ans)
                            lenoftemp=len(temp)
                            for j in range(0,lenoftemp):
```

```
                                clause=temp[j]
                                clause=clause[:-1]
                                predicate=pattern1.split(clause)
                                argumentlist=predicate[1:]
                                lengthofpredicate=len(predicate)-1
                                if predicate[0] in kbquery:
                                    if lengthofpredicate in kbquery[predicate[0]]:

kbquery[predicate[0]][lengthofpredicate].append([mergepart1,temp,j,argumentlist])
                                    else:

kbquery[predicate[0]][lengthofpredicate]=[[mergepart1,temp,j,argumentlist]]
                                else:

kbquery[predicate[0]]={lengthofpredicate:[[mergepart1,temp,j,argumentlist]]}
                                q.put(ans)
                            if(currentanswer=="TRUE"):
                                break
                        if(currentanswer=="TRUE"):
                            break
                        if(counter==2000 or (time.time()-query_start)>20):
                            break
                    answer.append(currentanswer)
    return answer

if __name__ == '__main__':
    finalanswer=resolution()
    o=open("output.txt","w+")
    wc=0
    while(wc < n-1):
        o.write(finalanswer[wc]+"\n")
        wc+=1
    o.write(finalanswer[wc])
    o.close()
```

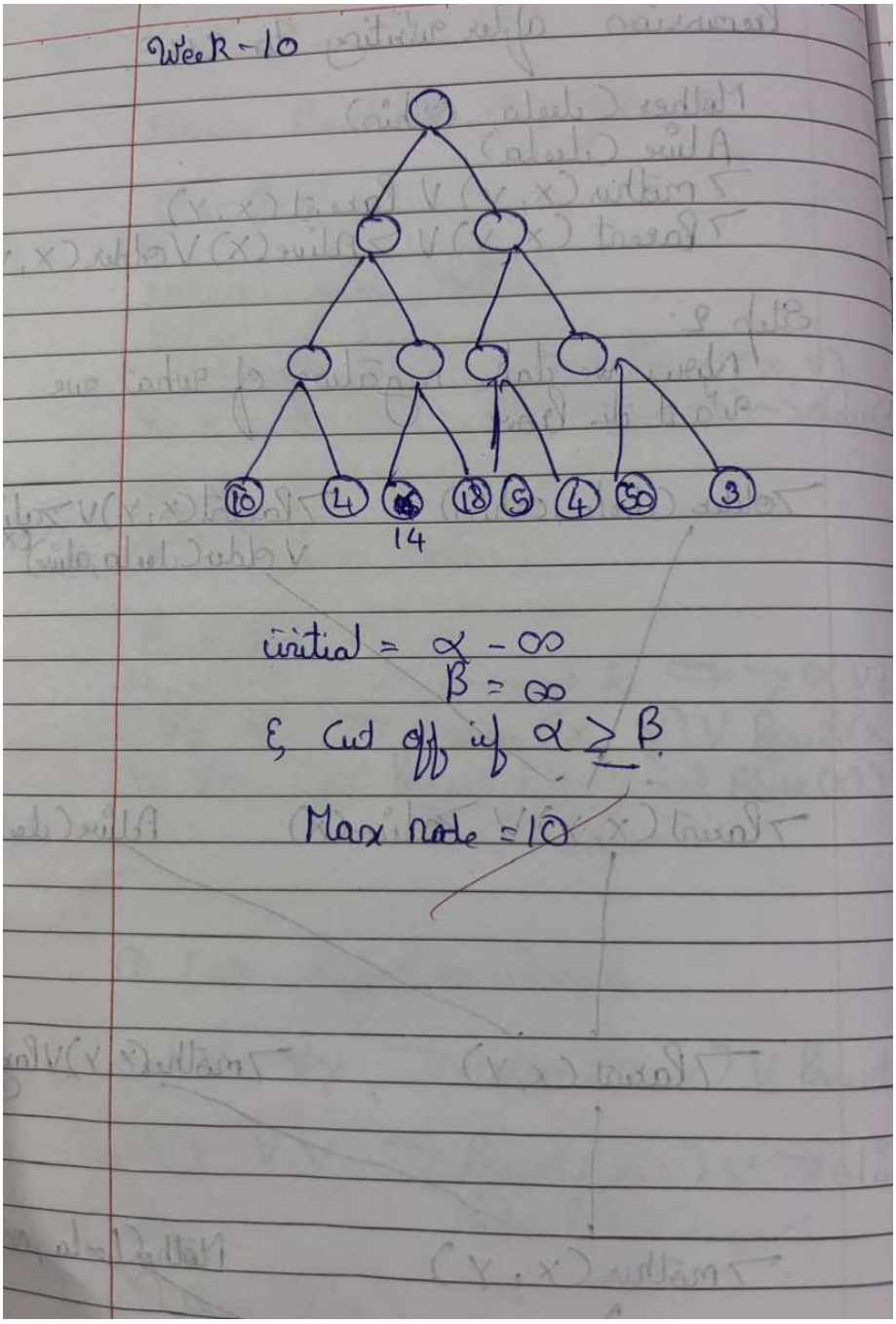
**Output:**

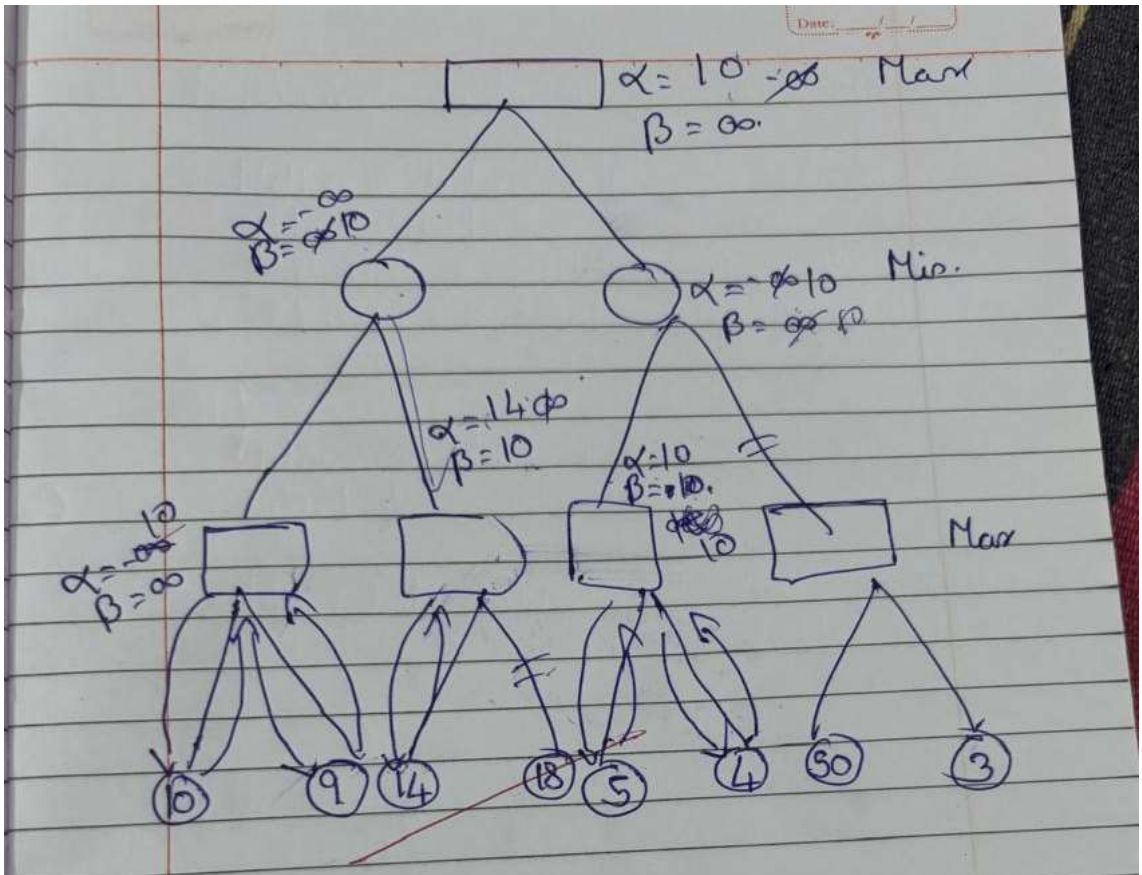


## Program 10

### Alpha-Beta Pruning

**Algorithm:**






14/12/24

output 9

The Conclusion 'older(Leela, Oshin)' is Proven by Resolution

~~the optimal Value is 10.~~

output 10

The optimal Value is 10.

**Code:**

```
def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta):
    # Base case: If the depth is 0, return the value at this node
    if depth == 0:
        return values[node_index]
    if maximizing_player:
        max_eval = -math.inf  # Start with a very small value
        for i in range(2):  # Assuming binary tree structure with two children
            eval_value = alpha_beta(depth - 1, node_index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval_value)
            alpha = max(alpha, eval_value)
            if beta <= alpha:
                break  # Beta cut-off
        return max_eval
    else:
        min_eval = math.inf  # Start with a very large value
        for i in range(2):  # Assuming binary tree structure with two children
            eval_value = alpha_beta(depth - 1, node_index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval_value)
            beta = min(beta, eval_value)
            if beta <= alpha:
                break  # Alpha cut-off
        return min_eval
def main():
    depth = int(input("Enter the depth of the tree: "))
    num_nodes = 2 ** (depth + 1) - 1  # Number of nodes in a binary tree
    values = []

    print(f"Enter the leaf node values for depth {depth}:")
    for i in range(2 ** depth):  # Leaf nodes are at the last level
        value = int(input(f"Enter value for leaf node {i+1}: "))
        values.append(value)
    alpha = -math.inf
    beta = math.inf
    result = alpha_beta(depth, 0, True, values, alpha, beta)
    print(f"Optimal value (using Alpha-Beta Pruning): {result}")

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

**Output:**

```
Enter the depth of the tree: 3
Enter the leaf node values for depth 3:
Enter value for leaf node 1: 10
Enter value for leaf node 2: 9
Enter value for leaf node 3: 14
Enter value for leaf node 4: 18
Enter value for leaf node 5: 5
Enter value for leaf node 6: 4
Enter value for leaf node 7: 50
Enter value for leaf node 8: 3
Optimal value (using Alpha-Beta Pruning): 10
```