VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT

on

Artificial Intelligence (23CS5PCAIN)

Submitted by

AMEENA YASMEEN (1BM23CS027)

in partial fulfillment for the award of the degree of

BACHELOR OF ENGINEERING

in

COMPUTER SCIENCE AND ENGINEERING

B.M.S. COLLEGE OF ENGINEERING

(Autonomous Institution under VTU)

BENGALURU-560019

Aug-2025 to Dec-2025

B.M.S. College of Engineering,

Bull Temple Road, Bangalore 560019

(Affiliated To Visvesvaraya Technological University, Belgaum)

Department of Computer Science and Engineering

CERTIFICATE

This is to certify that the Lab work entitled "Artificial Intelligence (23CS5PCAIN)" carried out by Ameena Yasmeen (1BM23CS027)**,** who is a bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence (23CS5PCAIN) work prescribed for the said degree.

Dr K R Mamatha Associate Professor Department of CSE, BMSCE	Dr. Kavitha Sooda Professor & HOD Department of CSE, BMSCE

Index

Github Link:

https://github.com/Ameena1BM23CS27/AI-LAB

Infosys®
Navigate your next

CERTIFICATE OF ACHIEVEMENT

The certificate is awarded to

Ameena Yasmeen

for successfully completing

Artificial Intelligence Foundation Certification

on November 25, 2025

Infosys | Springboard

Congratulations! You make us proud!

Satheesha B.N.
Satheesha B. Nanjappa
Senior Vice President and Head
Education, Training and Assessment
Infosys Limited

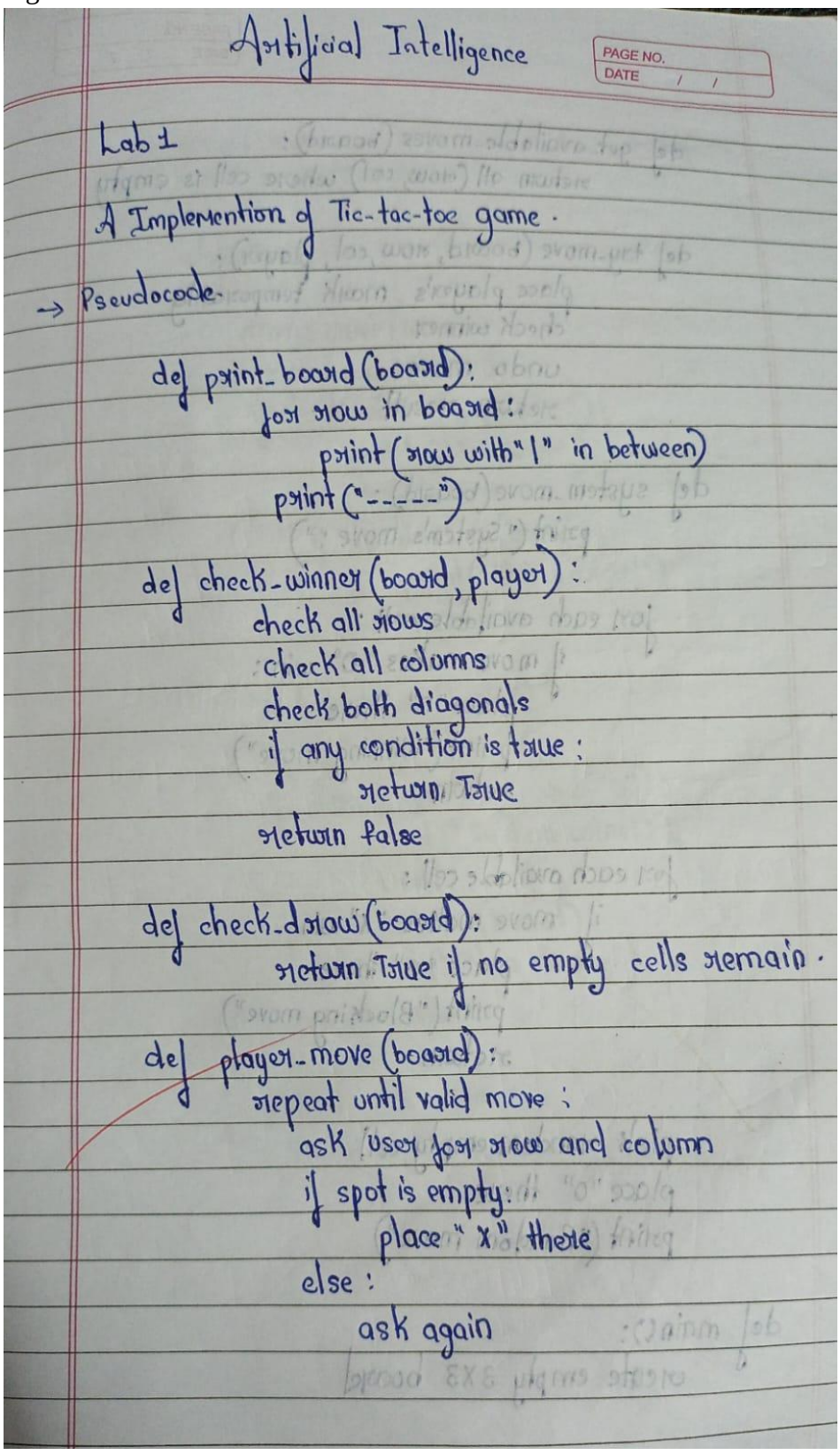
Issued on: Tuesday, November 25, 2025
To verify, scan the QR code at https://verify.onwingspan.com

Program 1

Implement Tic –Tac –Toe Game

Implement vacuum cleaner agent

Algorithm:



Artificial Intelligence

PAGE NO.
DATE / /

Lab 1

A Implemention of Tic-tac-toe game.

→ Pseudocode.

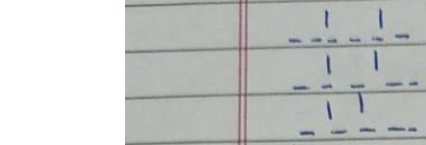
```
def print_board(board):
    for row in board:
        print(row with "|" in between)
    print("-----")

def check_winner(board, player):
    check all rows
    check all columns
    check both diagonals
    if any condition is true:
        return True
    return false

def check_draw(board):
    return True if no empty cells remain.

def player_move(board):
    repeat until valid move:
        ask user for row and column
        if spot is empty:
            place "X" there
        else:
            ask again
```

```
def get_available_moves(board):
    return all (row, col) where cell is empty

def try_move(board, row, col, player):
    place player's mark temporarily
    check winner
    undo move
    return result

def system_move(board):
    print("System's move:")

    for each available cell:
        if move makes "O" win:
            place "O" there
            print("Winning move")
            return

    for each available cell:
        if move makes "X" win:
            place "O" there.
            print("Blocking move")
            return

    pick random empty cell
    place "O" there
    print("Random move")

def main():
    create empty 3x3 board
    print("Welcome! You are X, System is O")
    print_board(board)

    while True:

        player_move(board)
        print_board(board)
        if check_winner(board, "X"):
            print("You Win!")
            break
        if check_draw(board):
            print("It is a draw!")
            break

        system_move(board)
        print_board(board)
        if check_winner(board, "O"):
            print("System wins!")
            break
        if check_draw(board):
            print("It's a draw!")
            break

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

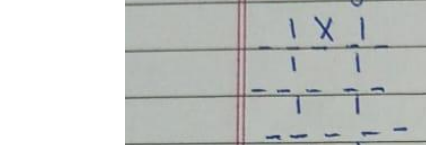
PAGE NO.
DATE / /

Output:

Welcome to tic-tac-toe! You are X, System is O.

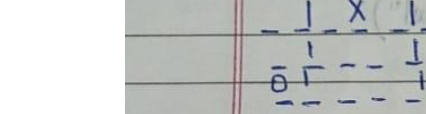


Enter your move as row, col (1-3 for both): 1, 2.

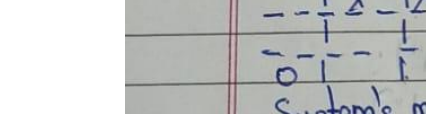


System's move:

System placed an 'O' at 3, 1 (random move)

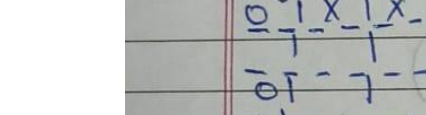


Enter your move as row, col (1-3 for both): 1, 3

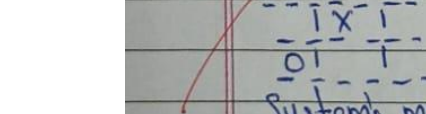


System's move

System place an 'O' at (1, 1) (blocking move)

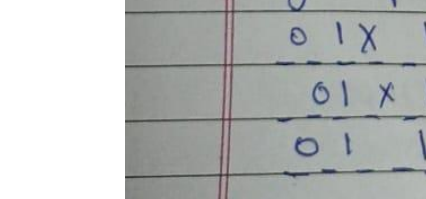


Enter your move as row, col (1-3 for both): 2, 2



System's move

System placed an 'O' at 2, 1 (winning move)



System wins! Better luck next time.

Code:

TIC-TAC-TOE :

Program:

```
def print_board(board):
print("\n  0   1   2")      for
i in range(3):
        print(f"{i} {board[i][0]} | {board[i][1]} | {board[i][2]}")
        if i < 2:
           print("  -----------")
print()

 def check_winner(board):      # Check
rows      for row in board:          if row[0]
== row[1] == row[2] != '-':
           return row[0]

    # Check columns      for col in range(3):          if
board[0][col] == board[1][col] == board[2][col] != '-':
           return board[0][col]

    # Check diagonals      if board[0][0] == board[1][1]
== board[2][2] != '-':
        return board[0][0]      if board[0][2] ==
board[1][1] == board[2][0] != '-':
        return board[0][2]

    return None

 def is_board_full(board):
for row in board:          if
'-' in row:              return
False
    return True

 def is_valid_move(board, row, col):
    return 0 <= row < 3 and 0 <= col < 3 and board[row][col] == '-'

 def get_player_move(player):
while True:          try:
           move = input(f"Player {player}, enter your move (row col): ")
```

```
            row, col = map(int, move.split())
return row, col          except (ValueError,
IndexError):
            print("Invalid input! Please enter row and column as two numbers (0-2).")

 def play_tic_tac_toe():
# Initialize empty board
    board = [['-' for _ in range(3)] for _ in range(3)]
current_player = 'X'

    print("Welcome to Tic-Tac-Toe!")
print("Enter moves as 'row col' )
print("Positions are numbered 0, 1, 2")

    # Main game loop
while True:
        print_board(board)

        # Get player move
        row, col = get_player_move(current_player)

        # Check if move is valid          if
is_valid_move(board, row, col):
            # Make the move
            board[row][col] = current_player

            # Check for winner
winner = check_winner(board)
if winner:
                print_board(board)
print(f" Player {winner} wins!")
                break

            # Check for tie
if is_board_full(board):
print_board(board)
print(" It's a tie!")
break

            # Switch players              current_player = 'O' if
current_player == 'X' else 'X'          else:
            print("Invalid move! That position is taken or out of bounds.")

 def main():
while True:
```

```
play_tic_tac_toe()

    # Ask to play again          play_again = input("\nDo you want to
play again? (y/n): ").lower()          if play_again != 'y':
        print("Thanks for playing!")
        break

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

Output:

```
Welcome to Tic-Tac-Toe!
Enter moves as 'row col' (e.g., '1 2')
Positions are numbered 0, 1, 2

  0   1   2
0 -  | -  | -
  -----------
1 -  | -  | -
  -----------
2 -  | -  | -

Player X, enter your move (row col):  0 1

  0   1   2
0 -  | X  | -
  -----------
1 -  | -  | -
  -----------
2 -  | -  | -

Player O, enter your move (row col):  1 1

  0   1   2
0 -  | X  | -
  -----------
1 -  | O  | -
  -----------
2 -  | -  | -

Player X, enter your move (row col):  0 0

  0   1   2
0 X  | X  | -
  -----------
1 -  | O  | -
  -----------
2 -  | -  | -

Player O, enter your move (row col):  1 1
Invalid move! That position is taken or out of bounds.

  0   1   2
0 X  | X  | -
  -----------
1 -  | O  | -
  -----------
2 -  | -  | -

Player O, enter your move (row col):  2 2

  0   1   2
0 X  | X  | -
  -----------
1 -  | O  | -
  -----------
2 -  | -  | O

Player X, enter your move (row col):  0 2

  0   1   2
0 X  | X  | X
  -----------
1 -  | O  | -
  -----------
2 -  | -  | O

🎉 Player X wins!

Do you want to play again? (y/n):  n
Thanks for playing!
```

Vacuum cleaner

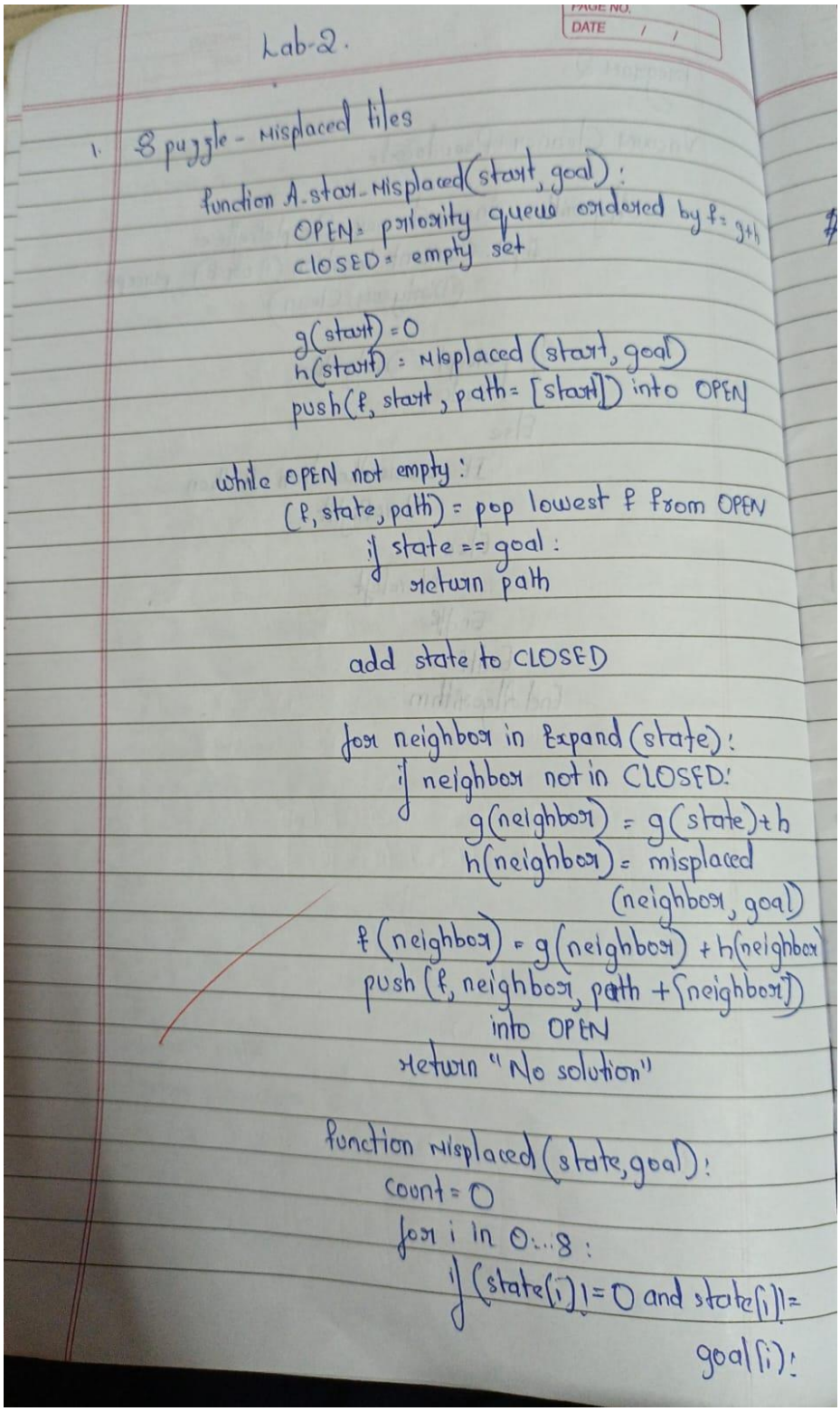
```
import random

 # Function to clean a room  def clean_room(rooms,
position):      if rooms[position] == 1:
print(f"Room {position+1} is Dirty. Cleaning...")
rooms[position] = 0          print(f"Room {position+1}
is now Clean.")      else:
        print(f"Room {position+1} is already Clean.")

 # Function to move to the next room
def move(position, total_rooms):
    position = (position + 1) % total_rooms
print(f"Moving to Room {position+1}")
    return position

 # Function to run the vacuum cleaner
def run(rooms, steps):
    position = 0  # start at first room
for _ in range(steps):
        clean_room(rooms, position)
position = move(position, len(rooms))
    print("Final Room Status:", rooms)



 # Initialize 4 rooms randomly (0 = clean, 1 = dirty)
rooms = [random.choice([0, 1]) for _ in range(4)]
 print("Initial Room Status:", rooms)

 # Run for 8 steps
run(rooms, 8) Output:
```

```
Initial Room Status: [0, 1, 0, 1]
Room 1 is already Clean.
Moving to Room 2
Room 2 is Dirty. Cleaning...
Room 2 is now Clean.
Moving to Room 3
Room 3 is already Clean.
Moving to Room 4
Room 4 is Dirty. Cleaning...
Room 4 is now Clean.
Moving to Room 1
Room 1 is already Clean.
Moving to Room 2
Room 2 is already Clean.
Moving to Room 3
Room 3 is already Clean.
Moving to Room 4
Room 4 is already Clean.
Moving to Room 1
Final Room Status: [0, 0, 0, 0]
```

Program 2

Implement 8 puzzle problems using Depth First Search (DFS)
Implement Iterative deepening search algorithm

Algorithm:



Lab-2.

1. 8 puzzle - Misplaced tiles

function A-star-Misplaced(start, goal):
OPEN = priority queue ordered by f = g+h
CLOSED = empty set

g(start) = 0
h(start) = Misplaced(start, goal)
push(f, start, path = [start]) into OPEN

while OPEN not empty:
(f, state, path) = pop lowest f from OPEN
if state == goal:
return path

add state to CLOSED

for neighbor in Expand(state):
if neighbor not in CLOSED:
g(neighbor) = g(state)+h
h(neighbor) = misplaced
(neighbor, goal)
f(neighbor) = g(neighbor) + h(neighbor)
push(f, neighbor, path + [neighbor])
into OPEN
return "No solution"

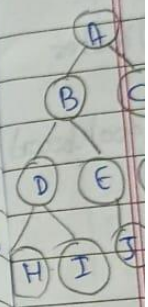
function misplaced(state, goal):
count = 0
for i in 0...8:
if (state[i] != 0 and state[i] !=
goal[i]):

2. Iterative Deepening Depth first search

PAGE NO.
DATE / /

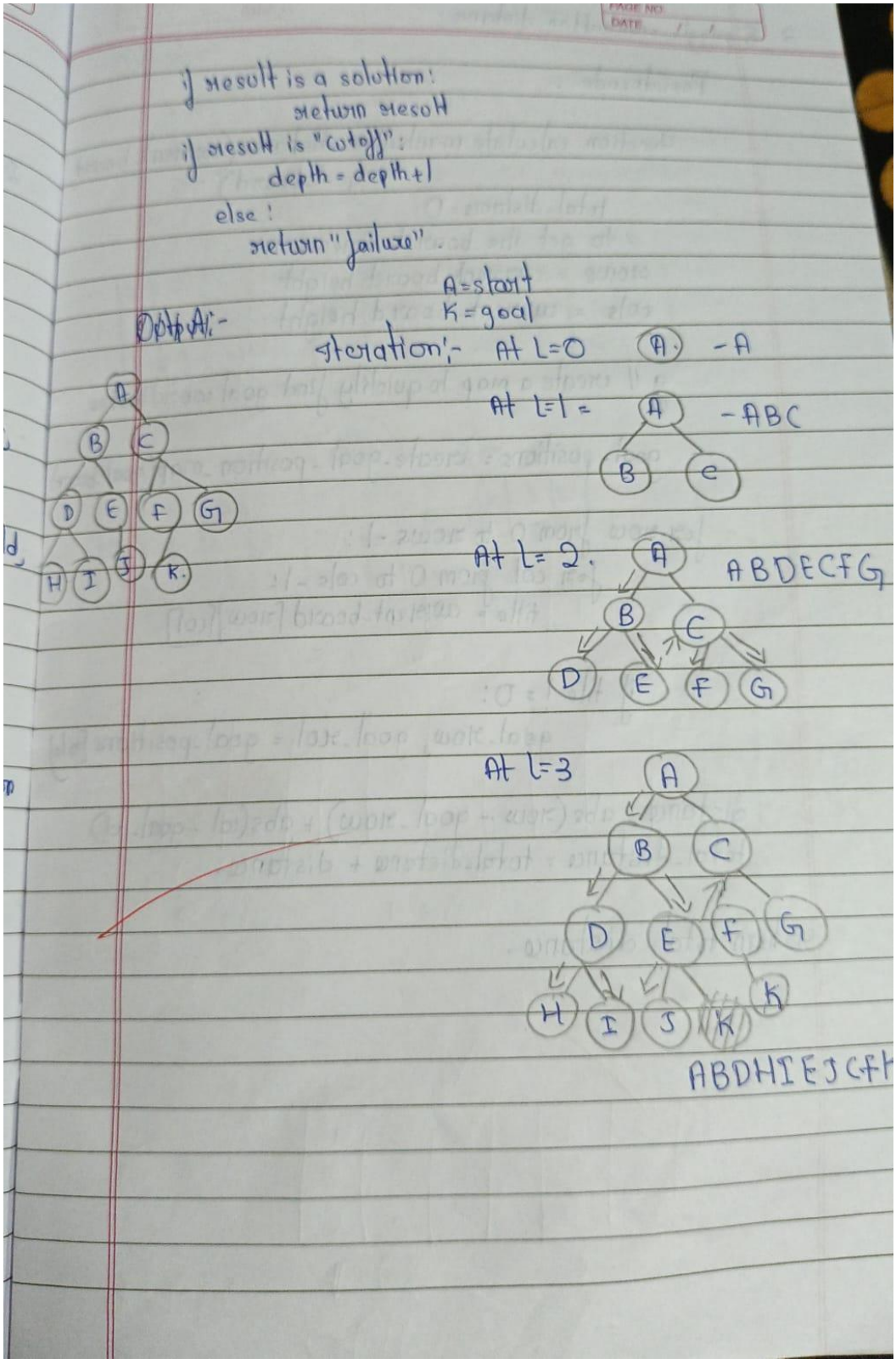
Algorithm

1. set the initial depth limit L=0
2. start a loop that will increment L in each iteration
3. Inside the loop perform a depth-limited DFS from the start node, with the current depth limit L.
4. The depth limit DFS function works as follows:
   - If the current node is the goal node, return the path to this node
   - If the current depth is equal to the depth limit, stop exploring this path.
   - For each unvisited child of the current node recursively call the depth-limited DFS function for the child with the depth increased by 1.
5. If the depth-limited DFS returns a solution, the IDDFS algorithm terminates and returns the solution.
6. If the depth-limited DFS completes without finding a solution increment the depth limit L=L+1 and repeat from step 3.
7. If the search space is finite and a solution exists, the algorithm is guaranteed to find it.



Pseudocode:

```
Function IDDFS(start_node, goal_node)
    depth = 0
    loop forever:
        result = DLS(start_node, goal_node, depth)
```



Code:

Misplaced Tiles – 8 puzzle import heapq

```
# ---------- Heuristic: Misplaced Tiles ---------- def
misplaced_tiles(state, goal):
    """Count number of misplaced tiles (ignores blank 0)."""
    return sum(1 for i in range(len(state)) if state[i] != 0 and state[i] != goal[i])

# ---------- Puzzle Neighbors ---------- def
get_neighbors(state):
    neighbors = []     idx = state.index(0) #
position of blank     x, y = divmod(idx, 3)

    moves = [(-1,0),(1,0),(0,-1),(0,1)] # up, down, left, right
for dx, dy in moves:
        nx, ny = x + dx, y + dy         if
0 <= nx < 3 and 0 <= ny < 3:
            new_idx = nx * 3 + ny             new_state = list(state)
new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
neighbors.append(tuple(new_state))     return neighbors

# ---------- A* Search ---------- def
a_star_misplaced(start, goal):
    open_list = []     heapq.heappush(open_list, (misplaced_tiles(start, goal),
0, start, [start]))     closed = set()

    while open_list:
        f, g, state, path = heapq.heappop(open_list)

        if state == goal:
            return path # solution found

        if state in closed:
continue
        closed.add(state)

        for neighbor in get_neighbors(state):
if neighbor not in closed:
                g_new = g + 1                h_new =
misplaced_tiles(neighbor, goal)                f_new =
g_new + h_new
                heapq.heappush(open_list, (f_new, g_new, neighbor, path + [neighbor]))     return
None # no solution found

# ---------- Run Example ---------- if
__name__ == "__main__":
    start = (1, 2, 3,
4, 5, 6,
```

```
        0, 7, 8)

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

    solution = a_star_misplaced(start, goal)

    if solution:
        print("Solution found in", len(solution)-1, "moves.")
for step in solution:            for i in range(0, 9, 3):
print(step[i:i+3])            print("-----")     else:
print("No solution found.") Output:
```

```
Solution found in 2 moves.
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)
-----
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
-----
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
-----
```

Manhattan distance – 8 puzzle import heapq

```
# ---------- Heuristic: Manhattan Distance ---------- def
manhattan_distance(state, goal):
    """Sum of Manhattan distances of each tile from its goal position."""
distance = 0     for i, tile in enumerate(state):         if tile != 0: # skip the
blank             goal_pos = goal.index(tile)             distance += abs(i // 3 -
goal_pos // 3) + abs(i % 3 - goal_pos % 3)     return distance

# ---------- Puzzle Neighbors ---------- def
get_neighbors(state):
```

```
   neighbors = []     idx = state.index(0)
# blank position     x, y = divmod(idx,
3)

   moves = [(-1,0),(1,0),(0,-1),(0,1)] # up, down, left, right
for dx, dy in moves:
       nx, ny = x + dx, y + dy         if
0 <= nx < 3 and 0 <= ny < 3:
           new_idx = nx * 3 + ny             new_state = list(state)
new_state[idx], new_state[new_idx] = new_state[new_idx], new_state[idx]
neighbors.append(tuple(new_state))     return neighbors

# ---------- A* Search ---------- def
a_star_manhattan(start, goal):
   open_list = []     heapq.heappush(open_list, (manhattan_distance(start, goal),
0, start, [start]))     closed = set()

   while open_list:
       f, g, state, path = heapq.heappop(open_list)

       if state == goal:
           return path # solution found

       if state in closed:
continue
       closed.add(state)

       for neighbor in get_neighbors(state):
if neighbor not in closed:
               g_new = g + 1               h_new =
manhattan_distance(neighbor, goal)               f_new =
g_new + h_new
               heapq.heappush(open_list, (f_new, g_new, neighbor, path + [neighbor]))     return
None # no solution found

# ---------- Run Example ---------- if
__name__ == "__main__":
   start = (1, 2, 3,
4, 5, 6,
           0, 7, 8)

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

```
    solution = a_star_manhattan(start, goal)

    if solution:
        print("Solution found in", len(solution)-1, "moves.")
for step in solution:             for i in range(0, 9, 3):
print(step[i:i+3])            print("-----")     else:
        print("No solution found.")
```

Output:

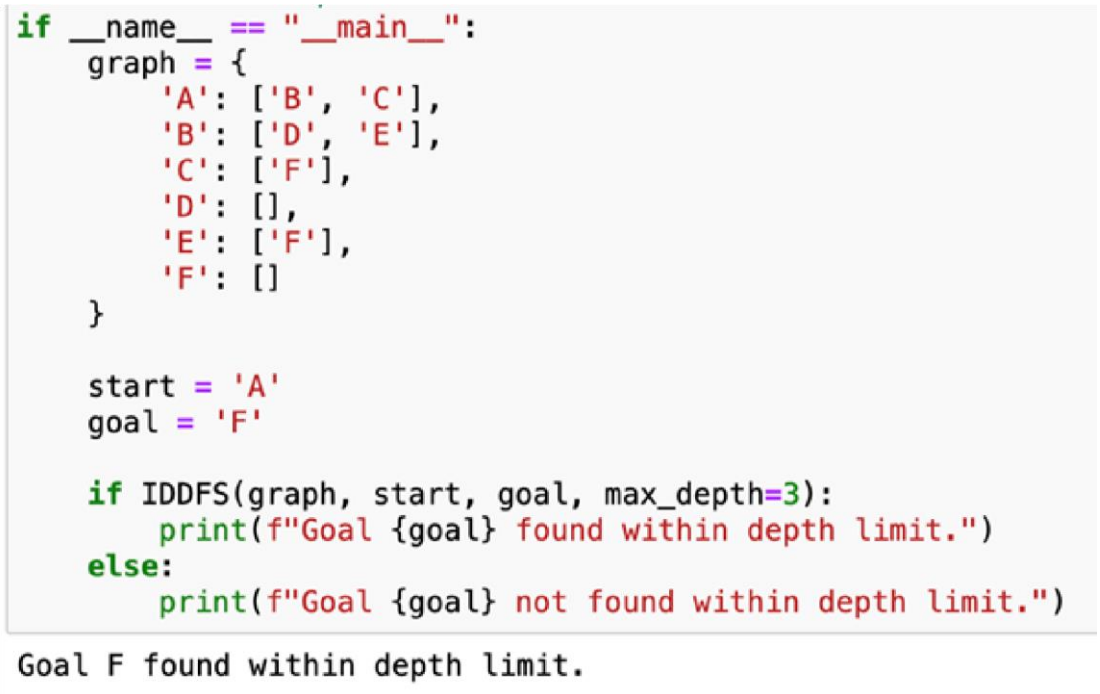
```
Solution found in 2 moves.
(1, 2, 3)
(4, 5, 6)
(0, 7, 8)
-----
(1, 2, 3)
(4, 5, 6)
(7, 0, 8)
-----
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
-----
```

Iterative Deepening Depth First Search # -

```
--------- Depth Limited Search ----------
def DLS(graph, node, goal, limit, visited):
if node == goal:
        return True
if limit == 0:
return False

    visited.add(node)     for neighbor in graph.get(node,
[]):         if neighbor not in visited:             if
DLS(graph, neighbor, goal, limit - 1, visited):
              return True
    return False

# ---------- IDDFS ---------- def IDDFS(graph,
start, goal, max_depth):     for depth in
range(max_depth + 1):         visited = set()
if DLS(graph, start, goal, depth, visited):
           return True
```

```
    return False

# ---------- Example Run ---------- if
__name__ == "__main__":
graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F'],
        'D': [],
        'E': ['F'],
        'F': []
    }

    start = 'A'
    goal = 'F'

    if IDDFS(graph, start, goal, max_depth=3):
print(f"Goal {goal} found within depth limit.")     else:
        print(f"Goal {goal} not found within depth limit.")
```

Output:

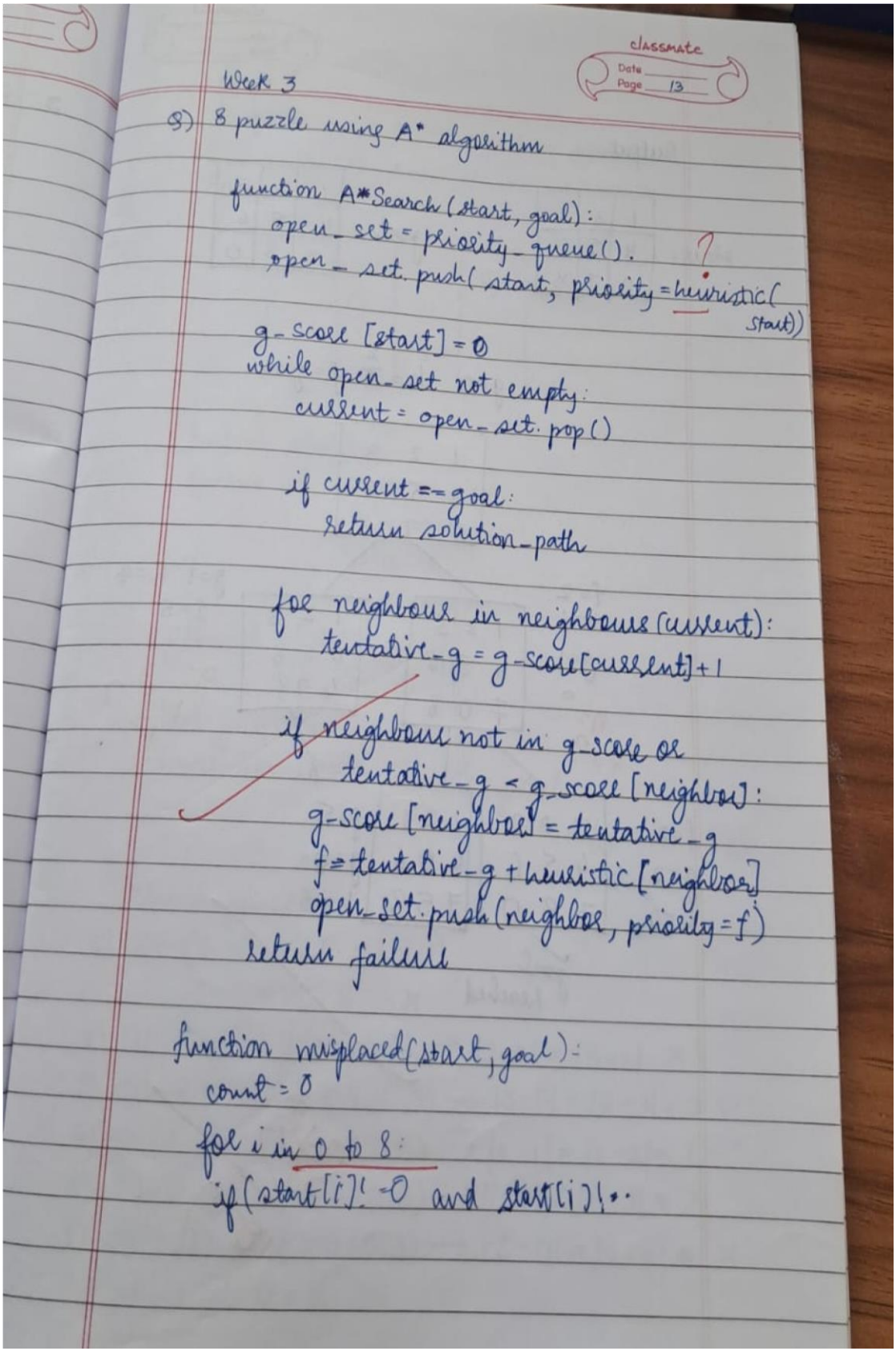


```
if __name__ == "__main__":
    graph = {
        'A': ['B', 'C'],
        'B': ['D', 'E'],
        'C': ['F'],
        'D': [],
        'E': ['F'],
        'F': []
    }

    start = 'A'
    goal = 'F'

    if IDDFS(graph, start, goal, max_depth=3):
        print(f"Goal {goal} found within depth limit.")
    else:
        print(f"Goal {goal} not found within depth limit.")
```

```
Goal F found within depth limit.
```

Program 3 Implement A* search algorithm

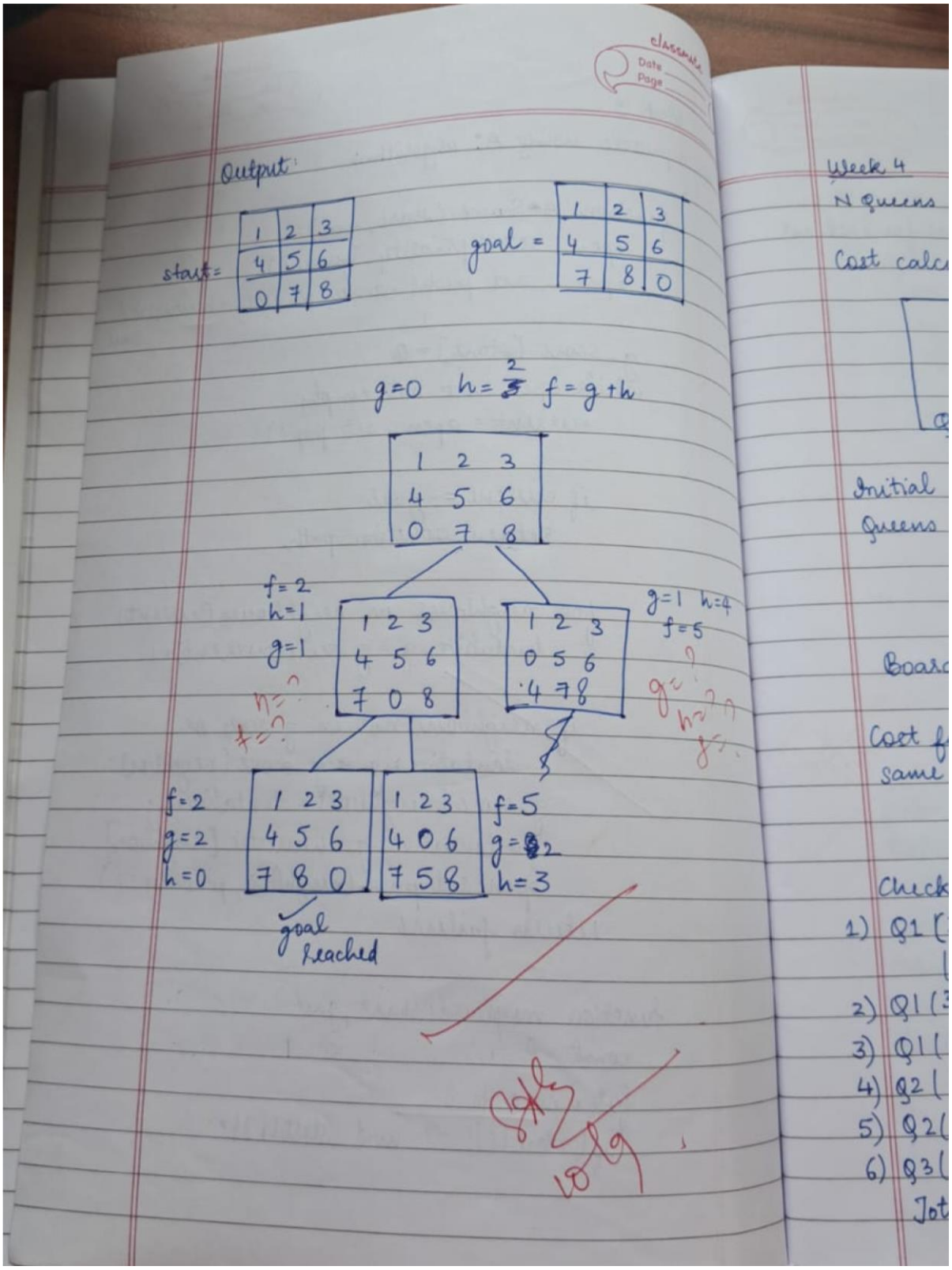
Algorithm:



Week 3

8) 8 puzzle using A* algorithm

```
function A*Search(start, goal):
    open_set = priority_queue().
    open_set.push(start, priority = heuristic(start))

    g_score[start] = 0
    while open_set not empty:
        current = open_set.pop()

        if current == goal:
            return solution_path

        for neighbour in neighbours(current):
            tentative_g = g_score[current] + 1

            if neighbour not in g_score or
                tentative_g < g_score[neighbour]:
                g_score[neighbour] = tentative_g
                f = tentative_g + heuristic[neighbour]
                open_set.push(neighbour, priority = f)
    return failure

function misplaced(start, goal):
    count = 0

    for i in 0 to 8:
        if(start[i] != 0 and start[i]!..
```


Output:
start =
goal =
g=0 h=2 f=g+h
f=2 h=1 g=1
g=1 h=4 f=5
f=2 g=2 h=0
f=5 g=2 h=3
goal Reached
Week 4
N Queens
Cost calc
Initial
Queens
Board
Cost f
same
Check
1) Q1 (
2) Q1 (
3) Q1 (
4) Q2 (
5) Q2 (
6) Q3 (
Tot

Code: import
heapq

```
# Goal state for the 8 puzzle
goal_state = [[1, 2, 3],
              [4, 5, 6],
              [7, 8, 0]]  # 0 is the blank space

# Directions: up, down, left, right
moves = [(1, 0), (-1, 0), (0, 1), (0, -1)]


def manhattan_distance(state):
    """Calculate Manhattan distance heuristic for a given state."""
distance = 0      for i in range(3):          for j in range(3):
value = state[i][j]              if value != 0:
                target_x = (value - 1) // 3                 target_y =
(value - 1) % 3                 distance += abs(i - target_x) +
abs(j - target_y)      return distance


def find_blank(state):
    """Find the position of blank (0) in the puzzle."""
for i in range(3):          for j in range(3):              if
state[i][j] == 0:
                return i, j


def state_to_tuple(state):
    """Convert list state to tuple (for hashing in sets)."""
return tuple(tuple(row) for row in state)


def a_star(start_state):
    """A* algorithm to solve 8-puzzle."""       start_h =
manhattan_distance(start_state)      pq = [(start_h, 0,
start_state, [])]  # (f, g, state, path)      visited = set()

    while pq:
        f, g, state, path = heapq.heappop(pq)

        if state == goal_state:
return path + [state]
```

```
        visited.add(state_to_tuple(state))
        x, y = find_blank(state)

        for dx, dy in moves:
            new_x, new_y = x + dx, y + dy
if 0 <= new_x < 3 and 0 <= new_y < 3:
                new_state = [row[:] for row in state]  # deep copy
                # Swap blank with neighbor
new_state[x][y], new_state[new_x][new_y] =
new_state[new_x][new_y], new_state[x][y]

                if state_to_tuple(new_state) not in visited:                    h
= manhattan_distance(new_state)                    heapq.heappush(pq,
(g + 1 + h, g + 1, new_state, path +
[state]))

    return None  # No solution found


 # Example usage:
 start_state = [[1, 2, 3],
                [4, 5, 6],
                [0, 7, 8]]

 solution = a_star(start_state)

 print("Steps to reach the goal:")
for step in solution:      for row
in step:          print(row)
print()
```

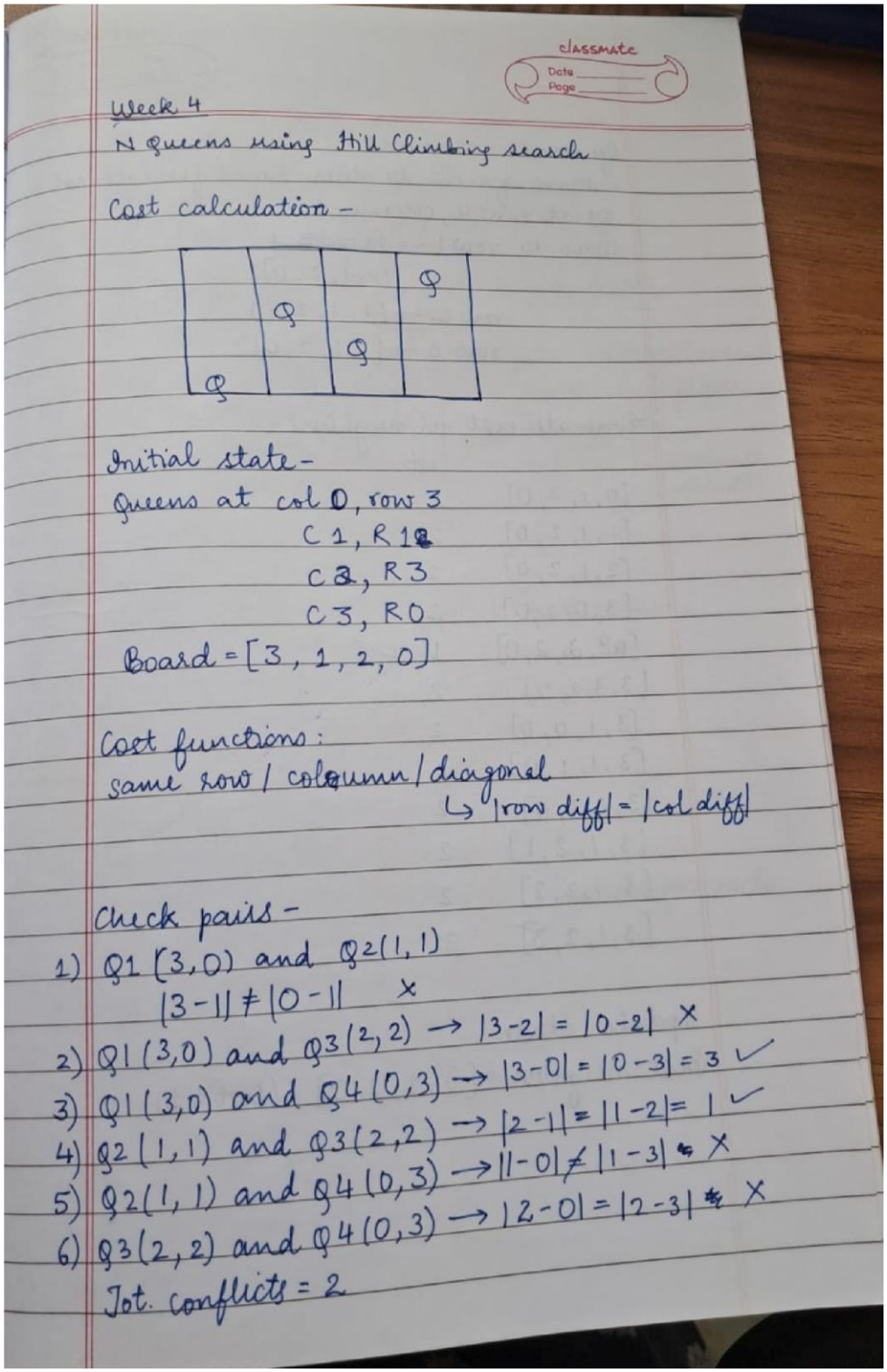
Output:

```
Steps to reach the goal:
[1, 2, 3]
[4, 5, 6]
[0, 7, 8]

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

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

Program 4

Implement Hill Climbing search algorithm to solve N-Queens problem

Algorithm:



Generate neighbors

Move queen to other rows for each col

Q1 at row 4, column 1

Move to row 1 → ~~[1, 2, 3, 4]~~

[0, 1, 2, 0]

row 2 → [1, 1, 2, 0]

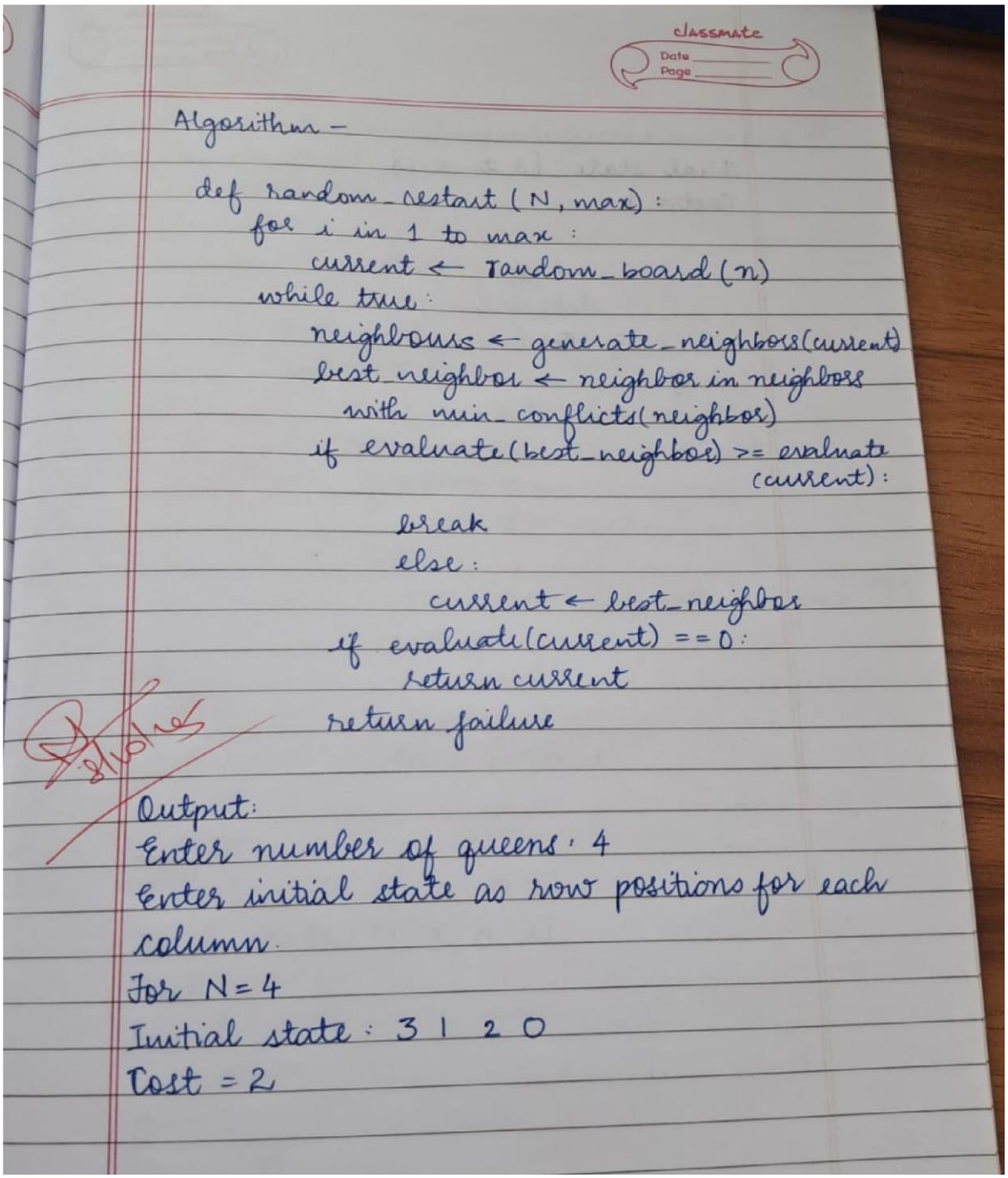
row 3 → [2, 1, 2, 0]

Evaluate cost for neighbors -

	Cost
[0, 1, 2, 0]	2
[1, 1, 2, 0]	2
[2, 1, 2, 0]	2
[3, 0, 2, 0]	2
[3, 2, 2, 0]	1 ✓
[3, 3, 2, 0]	2
[3, 1, 0, 0]	2
[3, 1, 1, 0]	2
[3, 1, 3, 0]	2
[3, 1, 2, 1]	2
[3, 1, 2, 2]	2
[3, 1, 2, 3]	2

Update State

Best neighbor = [3, 2, 2, 0] (Cost = 1)



Algorithm –

```
def random_restart (N, max):
    for i in 1 to max:
        current ← random_board (n)
    while true:
        neighbours ← generate_neighbors(current)
        best_neighbor ← neighbor in neighbors
          with min_conflicts(neighbors)
        if evaluate(best_neighbor) >= evaluate
                                      (current):
            break
        else:
            current ← best_neighbor
    if evaluate(current) == 0:
        return current
    return failure
```

Output:

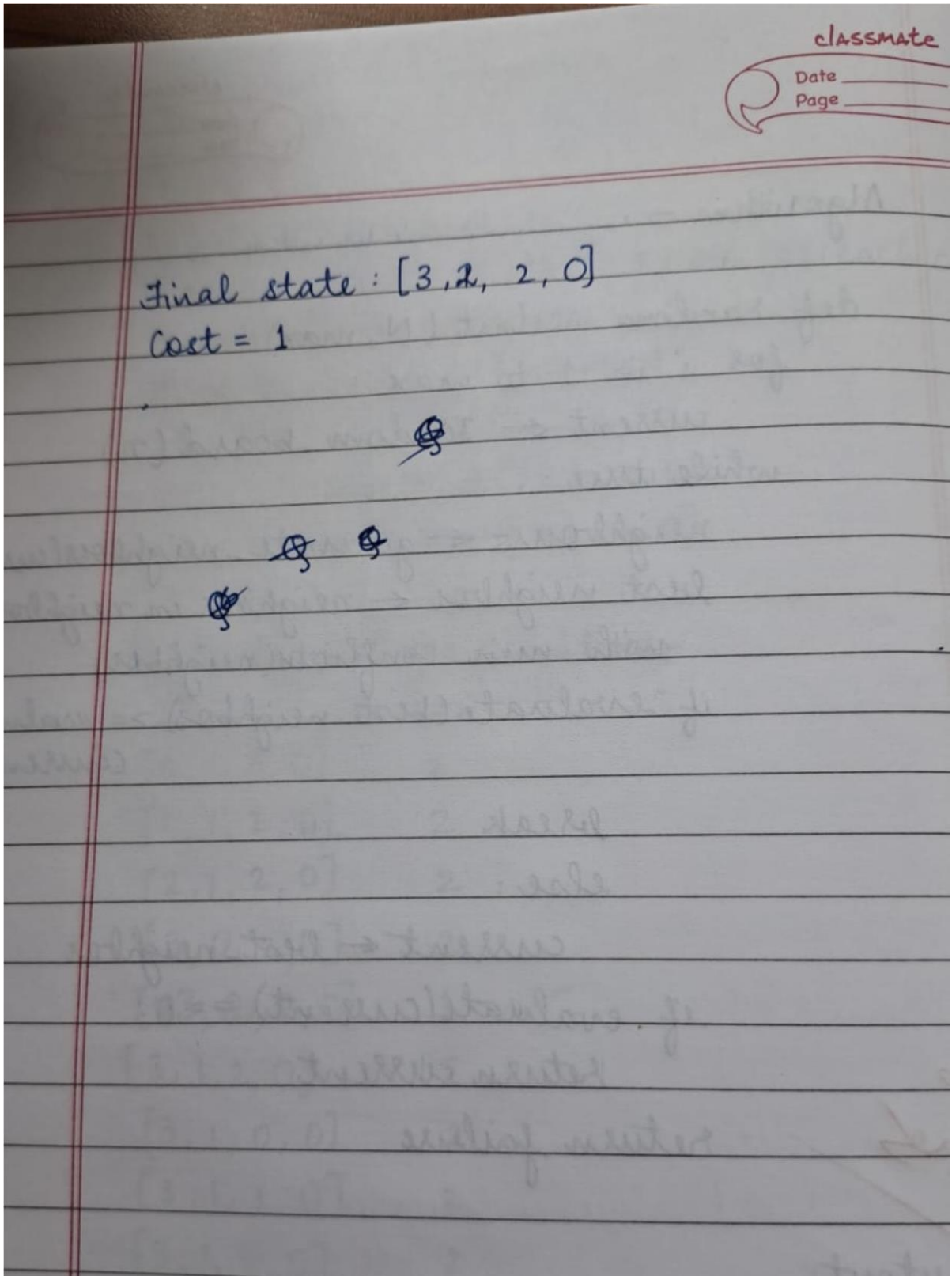
Enter number of queens : 4

Enter initial state as row positions for each column.

For N = 4

Initial state : 3 1 2 0

Cost = 2



Code:

```
import random
```

```
# Heuristic: number of pairs of queens attacking each other
def calculate_cost(state):     cost = 0
        n = len(state)     for i in range(n):         for j in range(i + 1,
n):            if state[i] == state[j] or abs(state[i] - state[j]) == abs(i -
j):                cost += 1
   return cost


# Generate all neighbors of the current state
def get_neighbors(state):     neighbors = []
        n = len(state)
for col in range(n):
for row in range(n):
         if state[col] != row: # move queen in col to new row
                neighbor = state.copy()
        neighbor[col] = row
        neighbors.append(neighbor)
   return neighbors


# Hill climbing algorithm def
hill_climb(initial_state):
current = initial_state
   current_cost = calculate_cost(current)

   print(f"Initial state: {current}, Cost = {current_cost}")

   while True:
      neighbors = get_neighbors(current)
best_neighbor = None
      best_cost = current_cost

      # Find the best neighbor
for neighbor in neighbors:
         cost = calculate_cost(neighbor)
if cost < best_cost:
                best_cost = cost
                best_neighbor = neighbor

      # If no better neighbor is found → stop        if
best_neighbor is None:            print(f"Final state: {current},
Cost = {current_cost}")           return current, current_cost

      # Move to the better neighbor        current,
current_cost = best_neighbor, best_cost
print(f"Move to: {current}, Cost = {current_cost}")
```

```
# Example usage if
__name__ == "__main__":
        n = int(input("Enter number of queens (N): "))
   print("Enter initial state as space-separated row positions for each column.")
print("Example for N=4: '1 3 0 2' means queen at (0,1), (1,3), (2,0), (3,2).")

   initial_state = list(map(int, input("Initial state: ").split()))

   if len(initial_state) != n:
      print("Invalid input: Length of initial state must be N.")
else:
      solution, cost = hill_climb(initial_state)

      if cost == 0:
         print("Goal state reached!")
else:
         print("Stuck in local minimum.")
```

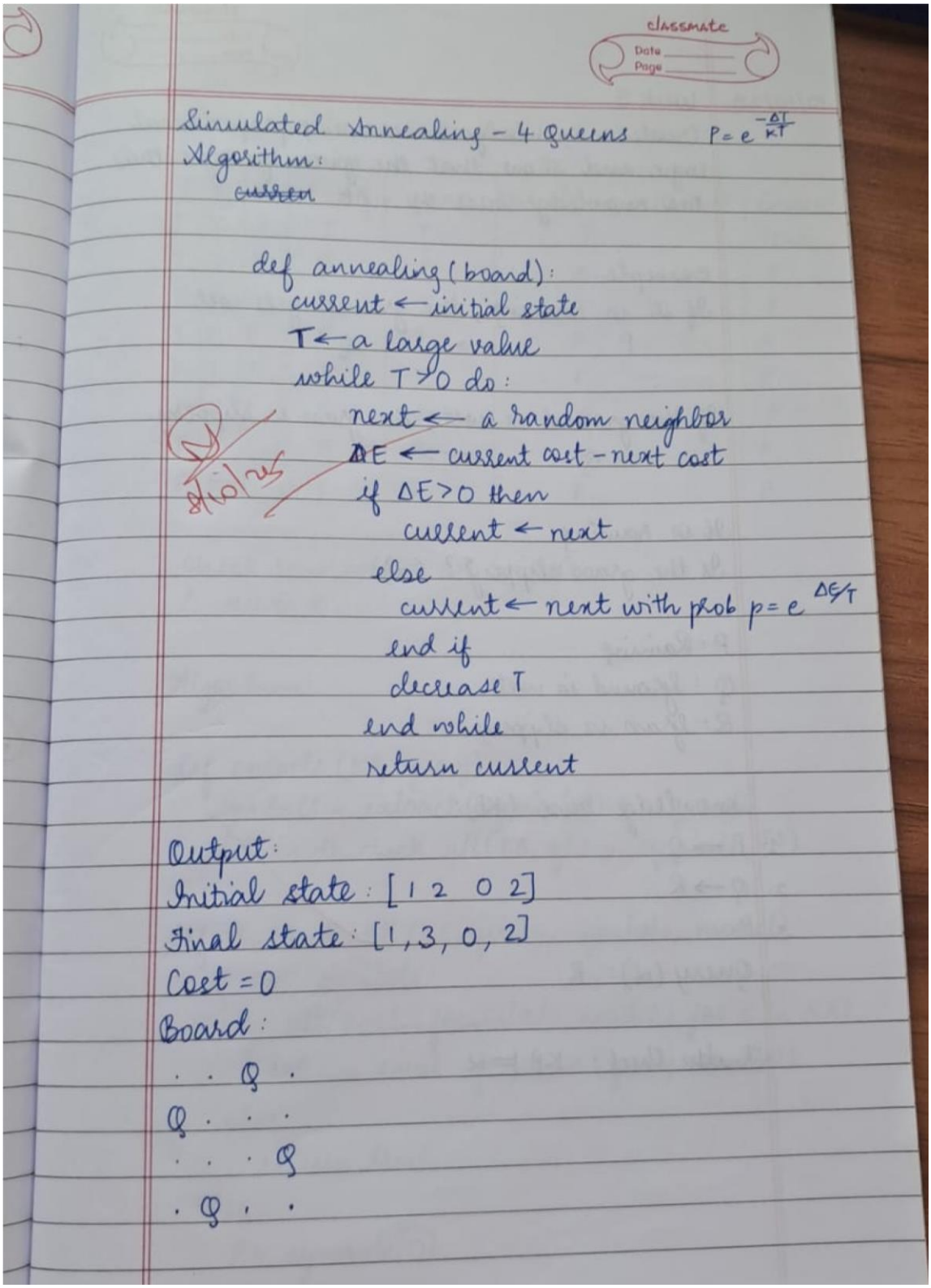
Output:

```
Enter number of queens (N):  4
Enter initial state as space-separated row positions for each column.
Example for N=4: '1 3 0 2' means queen at (0,1), (1,3), (2,0), (3,2).
Initial state:  3 1 2 0
Initial state: [3, 1, 2, 0], Cost = 2
Final state: [3, 1, 2, 0], Cost = 2
```

Program 5

Simulated Annealing to Solve 8-Queens problem

Algorithm:



Simulated Annealing - 4 Queens $P = e^{\frac{-\Delta T}{KT}}$

Algorithm:

```
def annealing(board):
    current ← initial state
    T ← a large value
    while T > 0 do:
        next ← a random neighbor
        ΔE ← current cost - next cost
        if ΔE > 0 then
            current ← next
        else
            current ← next with prob p = e^(ΔE/T)
        end if
        decrease T
    end while
    return current
```

Output:

Initial state: [1 2 0 2]

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

Cost = 0

Board:

```
. . Q .
Q . . .
. . . Q
. Q . .
```

Code:

```
import random
import math

def cost(state, N):
    """Compute number of attacking queen pairs."""     conflicts =
0     for i in range(N):         for j in range(i+1, N):             if
state[i] == state[j] or abs(state[i] - state[j]) == abs(i - j):
          conflicts += 1     return conflicts

def random_neighbor(state, N):
    """Generate a neighbor by moving one queen to another row in a random column."""
neighbor = state.copy()     col = random.randrange(N)     new_row =
random.randrange(N-1)     if new_row >= neighbor[col]:
        new_row += 1
neighbor[col] = new_row
return neighbor

def simulated_annealing(N, T0=5.0, alpha=0.995, Tmin=1e-6, max_iters=50000):
    """Solve N-Queens using simulated annealing."""
          # Random initial state: one queen per column
state = [random.randrange(N) for _ in range(N)]
    current_cost = cost(state, N)
          T = T0
    it = 0

    while T > Tmin and it < max_iters and current_cost != 0:
        neighbor = random_neighbor(state, N)
neighbor_cost = cost(neighbor, N)
        delta = neighbor_cost - current_cost

        if delta <= 0 or random.random() < math.exp(-delta / T):
            state, current_cost = neighbor, neighbor_cost

        T *= alpha
        it += 1

    return state, current_cost

def print_board(state, N):
    """Pretty-print the board with row and column numbers."""
print("   " + " ".join(str(c) for c in range(N)))   # column indices
for r in range(N):
        row = f"{r} "  # row index
for c in range(N):
```

```
            row += "Q " if state[c] == r else ". "
print(row)
    print()

# --------------------------
# User input
# --------------------------
N = int(input("Enter number of queens (N): "))

solution, c = simulated_annealing(N)

print(f"\nFinal state (col -> row): {solution}")
print("Cost:", c) print("\nBoard:")
print_board(solution, N)
```

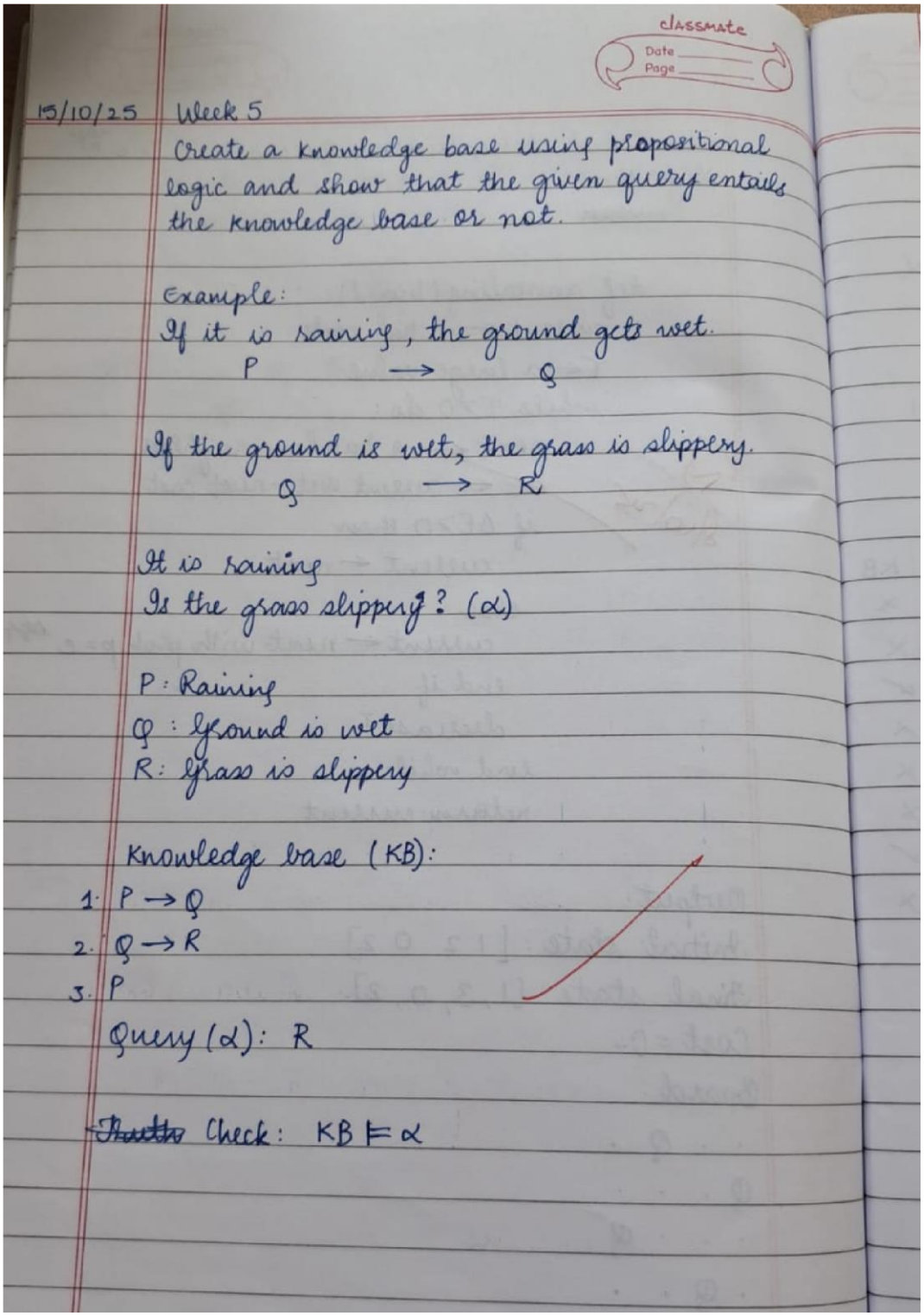
Output:

```
Enter number of queens (N):  5

Final state (col -> row): [4, 1, 3, 0, 2]
Cost: 0

Board:
   0 1 2 3 4
0 . . . Q .
1 . Q . . .
2 . . . . Q
3 . . Q . .
4 Q . . . .
```

Program 6

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

Algorithm:



15/10/25 Week 5

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

Example:

If it is raining, the ground gets wet.

$P \rightarrow Q$

If the ground is wet, the grass is slippery.

$Q \rightarrow R$

It is raining

Is the grass slippery? (α)

P: Raining

Q: Ground is wet

R: Grass is slippery

Knowledge base (KB):

1. $P \rightarrow Q$
2. $Q \rightarrow R$
3. P

Query (α): R

~~Truth~~ Check: $KB \models \alpha$

Truth table enumeration -

P	Q	R	P→Q	Q→R	KB = (P→Q)∧(Q→R)∧P	Entails R?
T	T	T	T	T	T	True
T	T	F	T	F	F	F
T	F	T	F	T	F	F
T	F	F	F	T	F	F
F	T	T	T	T	F	F
F	T	F	T	F	F	F
F	F	T	T	T	F	F
F	F	F	T	T	F	F

Check row where KB is true.
∴ KB ⊨ R

Algorithm:

```
def entails (KB, query):
    symbols = extract_symbols(KB + [query])
    return tt_check_all(KB, query, symbols, {})

def tt_check_all (KB, query, symbols, model):
    if not symbols:
        if all(eval_formula(s, model) for s in KB):
            return eval_formula (query, model)
        else:
            return true
    else:
        P = symbols[0]
```

rest = symbols[1:]
return (tt_check_all(KB, query, rest, {**model, P: True}) and
tt_check_all(KB, query, rest, {**model, P: False}))

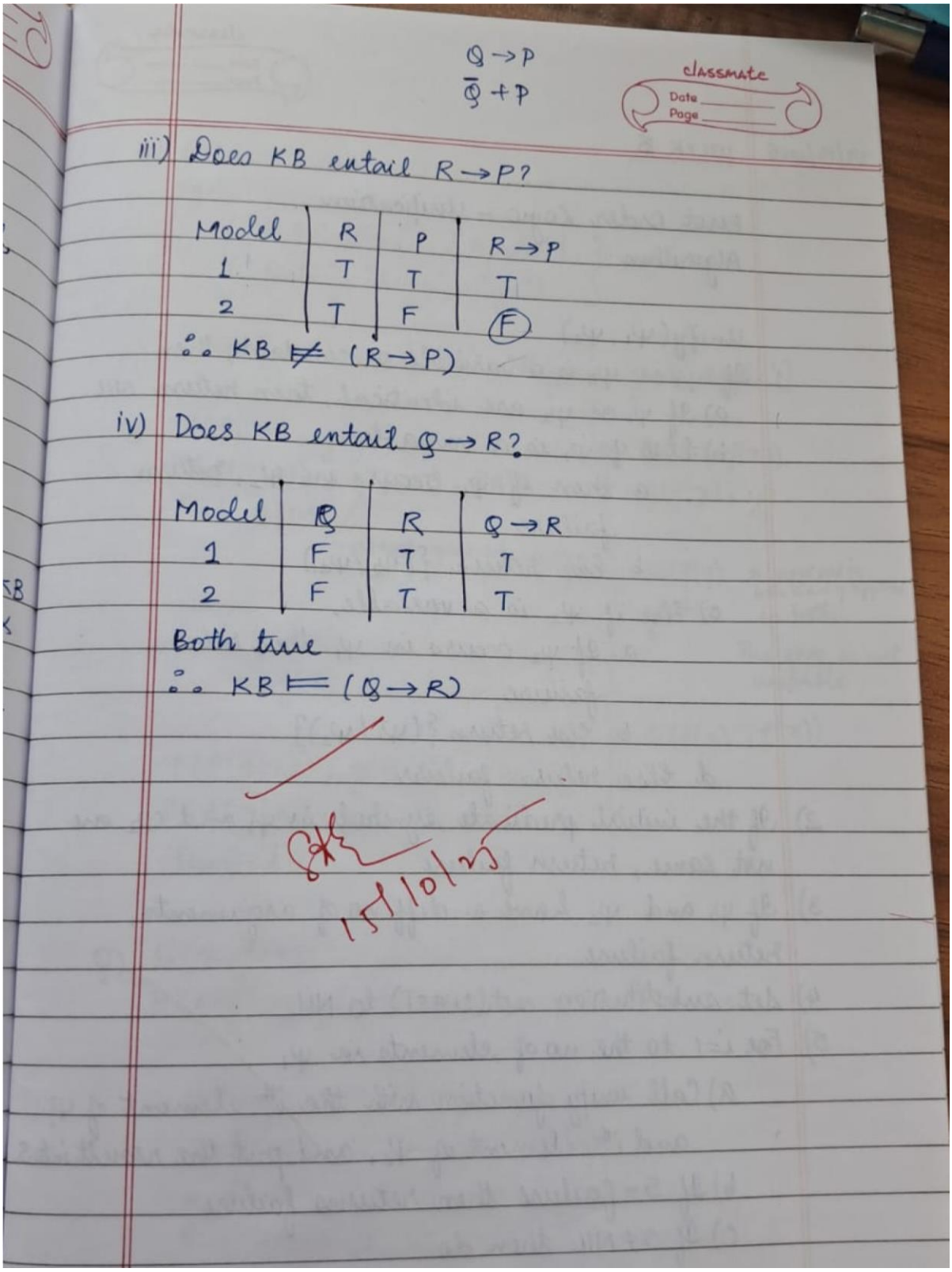
q) KB:
$Q \rightarrow P$
$P \rightarrow \neg Q$
$Q \vee R$

i)

P	Q	R	$Q \rightarrow P$	$P \rightarrow \neg Q$	$Q \vee R$	KB
T	T	T	T	F	T	✗
T	T	F	T	F	T	✗
T	F	T	T	T	T	✓
T	F	F	T	T	F	✗
F	T	T	F	T	T	✗
F	T	F	F	T	T	✗
F	F	T	T	T	T	✓
F	F	F	T	T	F	✗

KB is true in models:
(P=T, Q=F, R=T)
(P=F, Q=F, R=T)

ii) Does KB entail R?
R = True in both models
∴ KB ⊨ R



Code:

```
from itertools import product
```

```
# Define propositional logic operations def
implies(a, b):
        return (not a) or b

# Knowledge Base sentences def
KB(P, Q, R):
        s1 = implies(Q, P) # Q → P    s2 =
implies(P, not Q) # P → ¬Q
        s3 = Q or R # Q ∨ R
 return s1 and s2 and s3 # KB is true only if all hold

# All combinations of truth values for P, Q, R
values = list(product([False, True], repeat=3))

print("P\tQ\tR\tQ→P\tP→¬Q\tQ∨R\tKB") print("-"*50)

models = [] for P, Q,
R in values:
        s1 = implies(Q, P)
        s2 = implies(P, not Q)
        s3 = Q or R
        kb_val = s1 and s2 and s3
   print(f"{P}\t{Q}\t{R}\t{s1}\t{s2}\t{s3}\t{kb_val}")
        if kb_val:
      models.append((P, Q, R))

print("\n Models where KB is True:", models)

# Check entailments entails_R = all(R for P, Q, R in
models) entails_R_imp_P = all((not R) or P for P, Q, R in
models)
entails_Q_imp_R = all((not Q) or R for P, Q, R in models)

print("\nEntailments:") print("KB ⊨R :",
entails_R) print("KB ⊨R → P :",
entails_R_imp_P) print("KB ⊨Q → R :",
entails_Q_imp_R)
```

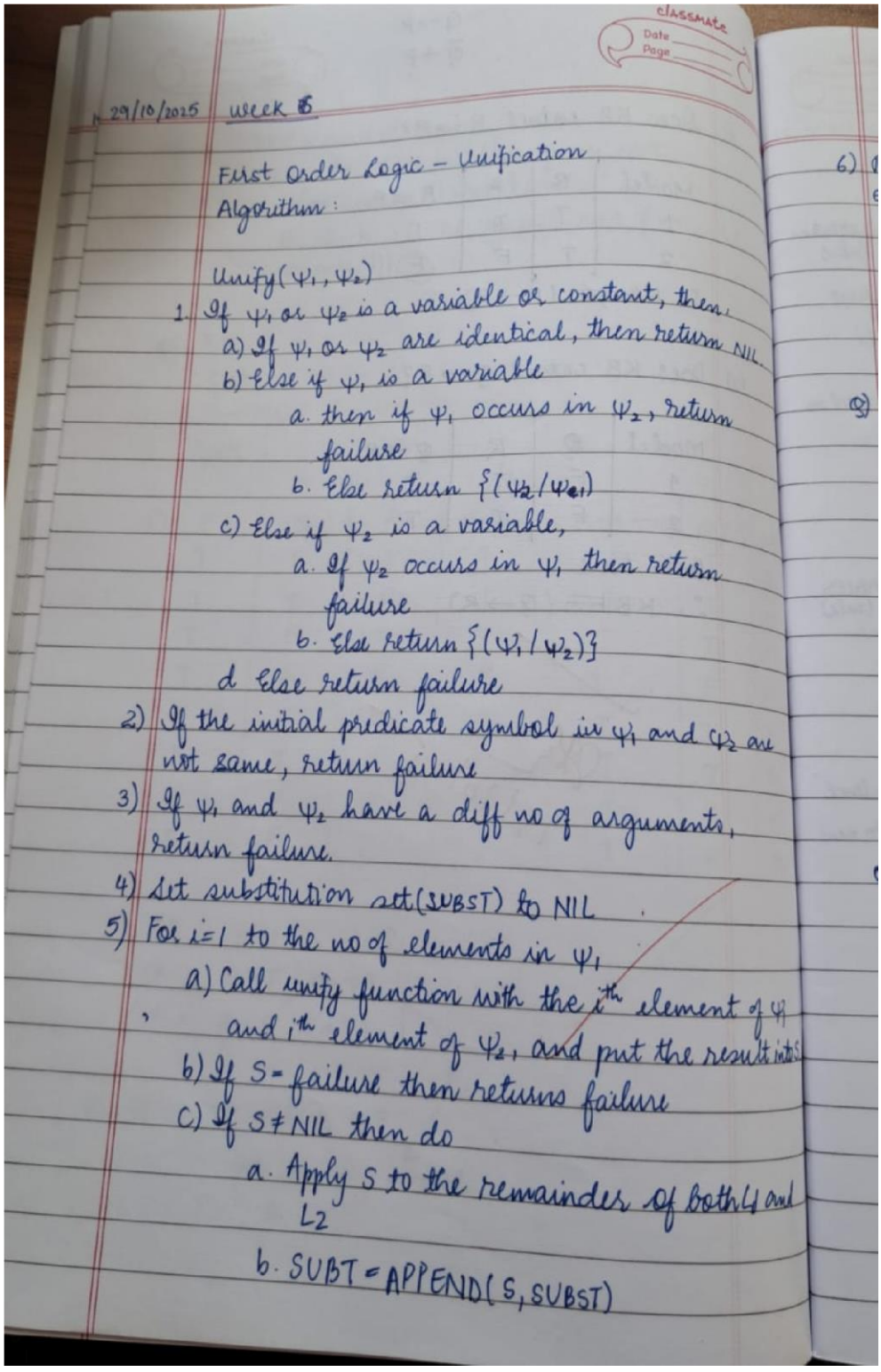
Output:

```
P       Q       R       Q→P     P→¬Q    Q∨R     KB
-------------------------------------------------
False   False   False   True    True    False   False
False   False   True    True    True    True    True
False   True    False   False   True    True    False
False   True    True    False   True    True    False
True    False   False   True    True    False   False
True    False   True    True    True    True    True
True    True    False   True    False   True    False
True    True    True    True    False   True    False

 Models where KB is True: [(False, False, True), (True, False, True)]

Entailments:
KB ⊨ R : True
KB ⊨ R → P : False
KB ⊨ Q → R : True
```

Program 7

Implement unification in first order logic.

Algorithm:



29/10/2025 WEEK 6

First Order Logic – Unification

Algorithm:

Unify(ψ_1, ψ_2)

1. If ψ_1 or ψ_2 is a variable or constant, then:
   a) If ψ_1 or ψ_2 are identical, then return NIL.
   b) Else if ψ_1 is a variable
      a. then if ψ_1 occurs in ψ_2, return failure
      b. Else return $\{(\psi_2/\psi_1)\}$
   c) Else if ψ_2 is a variable,
      a. If ψ_2 occurs in ψ_1 then return failure
      b. Else return $\{(\psi_1/\psi_2)\}$
   d Else return failure
2) If the initial predicate symbol in ψ_1 and ψ_2 are not same, return failure
3) If ψ_1 and ψ_2 have a diff no of arguments, return failure.
4) Set substitution set(SUBST) to NIL
5) For i=1 to the no of elements in ψ_1
   a) Call unify function with the i^{th} element of ψ_1 and i^{th} element of ψ_2, and put the result into S
   b) If S= failure then returns failure
   c) If S≠ NIL then do
      a. Apply S to the remainder of both L_1 and L_2
      b. SUBT = APPEND(S, SUBST)

Code:

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

    # If both are identical
if x == y:
        return substitutions

    # If x is a variable      if isinstance(x,
str) and x.islower():          return
unify_var(x, y, substitutions)

    # If y is a variable      if isinstance(y,
str) and y.islower():          return
unify_var(y, x, substitutions)

    # If both are compound expressions (like lists or tuples)
if isinstance(x, tuple) and isinstance(y, tuple):          if x[0]
!= y[0] or len(x) != len(y):
            return None          for a,
b in zip(x[1:], y[1:]):
            substitutions = unify(a, b, substitutions)
if substitutions is None:
                return None
        return substitutions

    return None



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

```
def occurs_check(var, x, substitutions):
if var == x:          return True      elif
isinstance(x, tuple):
        return any(occurs_check(var, arg, substitutions) for arg in x[1:])      elif isinstance(x, str)
and x in substitutions:
        return occurs_check(var, substitutions[x], substitutions)
return False


# Example  expr1 = ("Eats",
"x", "Apple")
expr2 = ("Eats", "Riya", "y")

result = unify(expr1, expr2)
print("Unification:", result)
```

Output:

```
Unification: {'x': 'Riya', 'y': 'Apple'}
```

Program 8

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

Algorithm:

Code:

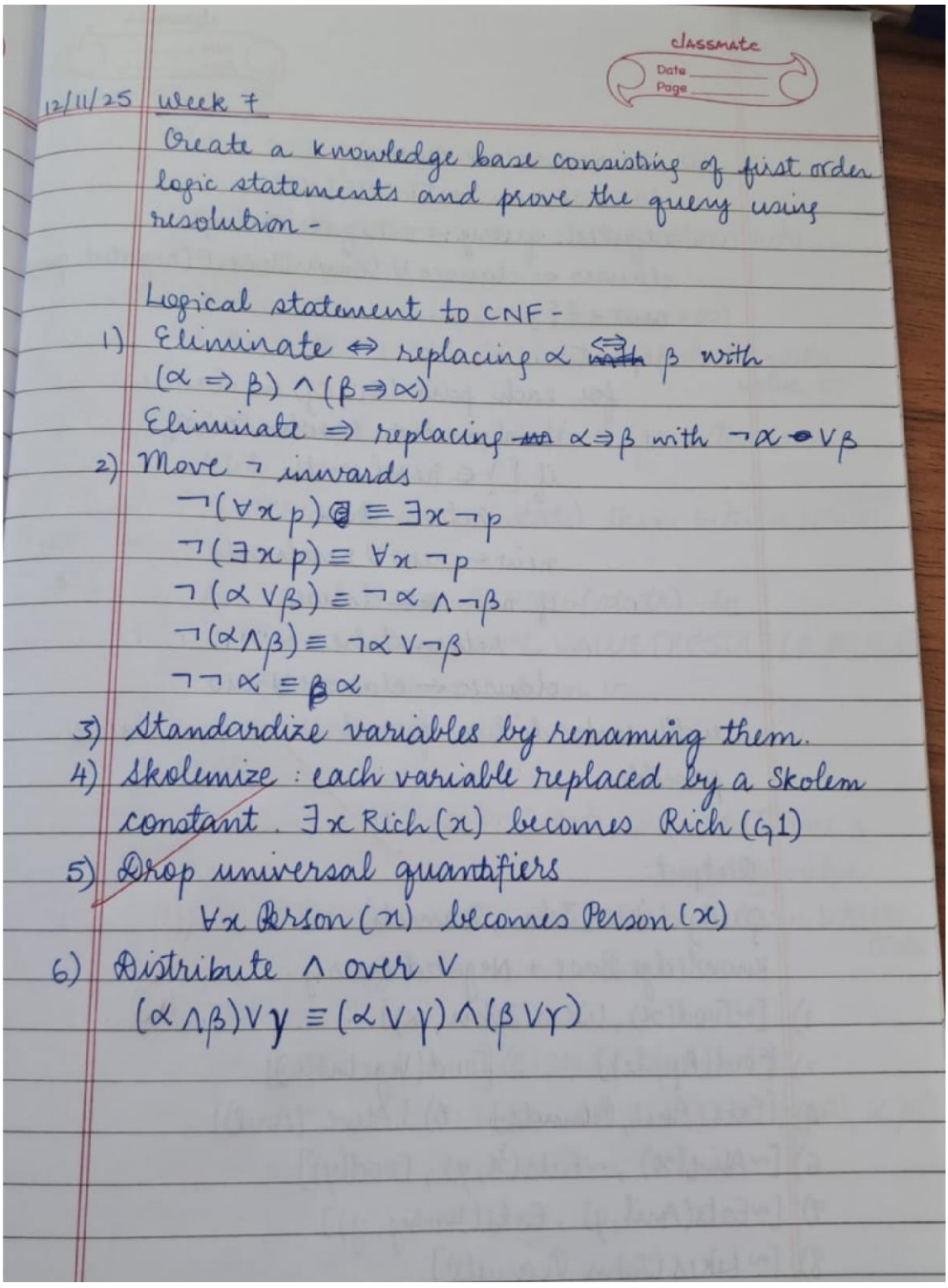
```
def forward_chaining(KB, query):
inferred = set()      new_inferred =
True

    while new_inferred:
new_inferred = False          for
rule in KB:
            premises, conclusion = rule              if all(p in inferred or p in KB for p in
premises) and conclusion not in inferred:
                inferred.add(conclusion)
new_inferred = True                  if
conclusion == query:
                    return True
return False

 # Example Knowledge Base
 KB = [
    (["American(Robert)", "Weapon(x)", "Sells(Robert, x, A)", "Hostile(A)"],
"Criminal(Robert)"),
    (["Missile(x)"], "Weapon(x)"),
    (["Owns(A, x)", "Missile(x)"], "Sells(Robert, x, A)"),
    (["Enemy(A, America)"], "Hostile(A)")
 ]

 facts = {
    "American(Robert)",
    "Enemy(A, America)",
    "Owns(A, T1)",
    "Missile(T1)"
 }
 # Add base facts to KB
for fact in facts:
    KB.append(([], fact))
 # Query  query =
"Criminal(Robert)"
 print("Is Robert a criminal?", forward_chaining(KB, query)) Output:
```

```
Is Robert a criminal? False
```

Program 9

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

Algorithm:



12/11/25 Week 7

Create a knowledge base consisting of first order logic statements and prove the query using resolution -

Logical statement to CNF -

1) Eliminate ⇔ replacing $\alpha \Leftrightarrow \beta$ with $(\alpha \Rightarrow \beta) \wedge (\beta \Rightarrow \alpha)$

   Eliminate ⇒ replacing $\alpha \Rightarrow \beta$ with $\neg\alpha \vee \beta$

2) Move ¬ inwards

   $\neg(\forall x\, p) \equiv \exists x \neg p$

   $\neg(\exists x\, p) \equiv \forall x \neg p$

   $\neg(\alpha \vee \beta) \equiv \neg\alpha \wedge \neg\beta$

   $\neg(\alpha \wedge \beta) \equiv \neg\alpha \vee \neg\beta$

   $\neg\neg\alpha \equiv \alpha$

3) Standardize variables by renaming them.

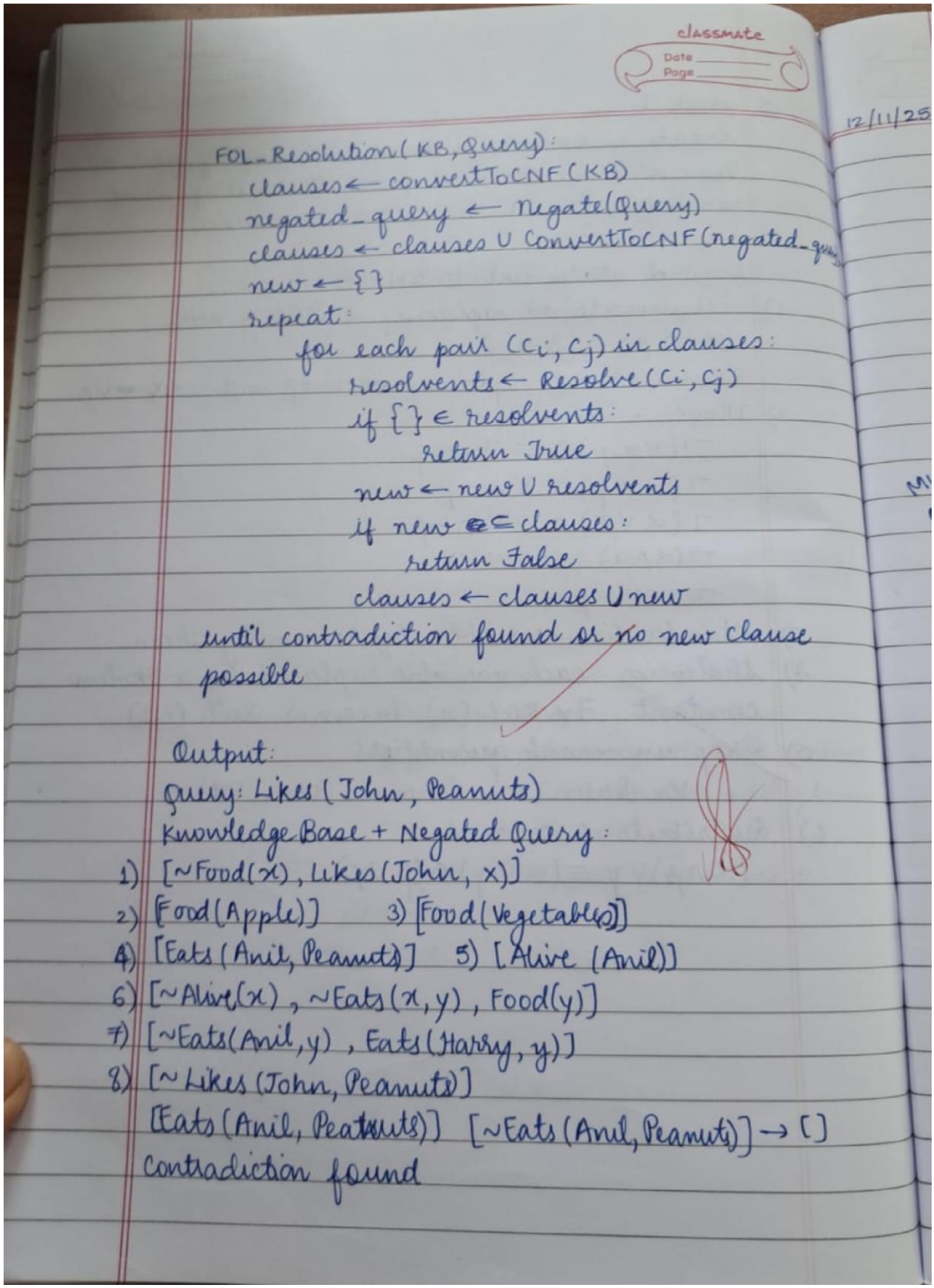
4) Skolemize : each variable replaced by a Skolem constant. $\exists x$ Rich(x) becomes Rich(G1)

5) Drop universal quantifiers

   $\forall x$ Person(x) becomes Person(x)

6) Distribute $\wedge$ over $\vee$

   $(\alpha \wedge \beta) \vee \gamma \equiv (\alpha \vee \gamma) \wedge (\beta \vee \gamma)$



```
FOL_Resolution(KB, Query):
    clauses ← convertToCNF(KB)
    negated_query ← negate(Query)
    clauses ← clauses U ConvertToCNF(negated_query)
    new ← {}
    repeat:
        for each pair (Ci, Cj) in clauses:
            resolvents ← Resolve(Ci, Cj)
            if {} ∈ resolvents:
                return True
            new ← new U resolvents
            if new ⊆ clauses:
                return False
            clauses ← clauses U new
    until contradiction found or no new clause possible
```

Output:

Query: Likes(John, Peanuts)

Knowledge Base + Negated Query:

1) [~Food(x), Likes(John, x)]
2) [Food(Apple)] 3) [Food(Vegetables)]
4) [Eats(Anil, Peanuts)] 5) [Alive(Anil)]
6) [~Alive(x), ~Eats(x, y), Food(y)]
7) [~Eats(Anil, y), Eats(Harry, y)]
8) [~Likes(John, Peanuts)]

[Eats(Anil, Peanuts)] [~Eats(Anil, Peanuts)] → []

Contradiction found

Code:

```
import copy
```

```
# ------------------------------
# Predicate Structure
# ------------------------------ class Predicate:
def __init__(self, name, args, negated=False):
        self.name = name         self.args = args if
isinstance(args, tuple) else tuple(args)         self.negated =
negated

    def __eq__(self, other):
        return (self.name == other.name and
            self.args == other.args and
                    self.negated == other.negated)

    def __hash__(self):
        return hash((self.name, self.args, self.negated))

    def __repr__(self):
        neg = "~" if self.negated else ""
args_str = ",".join(str(a) for a in self.args)
        return f"{neg}{self.name}({args_str})"

    def negate(self):
        return Predicate(self.name, self.args, not self.negated)

    def substitute(self, theta):
        """Apply substitution theta to this predicate"""         new_args =
tuple(substitute_term(arg, theta) for arg in self.args)         return
Predicate(self.name, new_args, self.negated)

def substitute_term(term, theta):     """Apply
substitution to a term"""     if isinstance(term, str) and
term.islower():  # variable         if term in theta:
return substitute_term(theta[term], theta)         return
term     elif isinstance(term, tuple):
        return tuple(substitute_term(t, theta) for t in term)
return term

# ------------------------------
# Unification Algorithm
# ------------------------------
def unify(x, y, theta=None):
if theta is None:         theta =
{}     if theta == "FAIL":
        return "FAIL"     elif x == y:         return theta
elif isinstance(x, str) and x.islower():  # variable
return unify_var(x, y, theta)     elif isinstance(y,
```

```
str) and y.islower():  # variable         return
unify_var(y, x, theta)     elif isinstance(x, tuple)
and isinstance(y, tuple):         if len(x) != len(y):
return "FAIL"         theta = unify(x[0], y[0], theta)
if theta == "FAIL":             return "FAIL"
return unify(x[1:], y[1:], theta)     else:
        return "FAIL"

def unify_var(var, x, theta):
if var in theta:
        return unify(theta[var], x, theta)     elif
isinstance(x, str) and x.islower() and x in theta:
        return unify(var, theta[x], theta)
elif occurs_check(var, x, theta):
        return "FAIL"
else:
        new_theta = copy.deepcopy(theta)
new_theta[var] = x
        return new_theta

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

# ------------------------------
# Variable Standardization
# ------------------------------
var_counter = 0

def standardize_variables(clause):
    """Rename all variables in a clause to unique names"""
global var_counter     mapping = {}     new_clause = []

    for pred in clause:
new_args = []         for
arg in pred.args:
            if isinstance(arg, str) and arg.islower():  # variable
            if arg not in mapping:
                    mapping[arg] = f"{arg}{var_counter}"
            var_counter += 1
            new_args.append(mapping[arg])             else:
```

```
                new_args.append(arg)
new_clause.append(Predicate(pred.name, new_args, pred.negated))

    return new_clause

# ----------------------------
# Resolution Algorithm
# ---------------------------- def
resolve(ci, cj):
    """Resolve two clauses using FOL resolution"""
    ci = standardize_variables(ci)
cj = standardize_variables(cj)

    resolvents = []

    for i, pi in enumerate(ci):
for j, pj in enumerate(cj):
        # Check if predicates can be resolved (opposite signs, same name)
if pi.negated != pj.negated and pi.name == pj.name:
                # Try to unify the arguments
        theta = unify(pi.args, pj.args)
                                if
theta != "FAIL":
                # Create resolvent by removing resolved predicates and applying substitution
        new_clause = []

                # Add literals from ci except pi
        for k, pred in enumerate(ci):
        if k != i:
                    new_clause.append(pred.substitute(theta))

                # Add literals from cj except pj
        for k, pred in enumerate(cj):
        if k != j:
                    new_clause.append(pred.substitute(theta))

                # Remove duplicates
        new_clause = list(set(new_clause))
resolvents.append(new_clause)

    return resolvents

def fol_resolution(kb, query):
"""FOL resolution algorithm"""      #
Negate query and add to KB
```

```
    clauses = [clause[:] for clause in kb]  # deep copy
clauses.append([query.negate()])

    print(f"\nKnowledge Base + Negated Query:")
for i, clause in enumerate(clauses):
        print(f"  {i+1}. {clause}")
    print()

    iteration = 0
while True:
        iteration += 1         n = len(clauses)         pairs = [(clauses[i],
clauses[j]) for i in range(n) for j in range(i + 1, n)]

        new_clauses = []
for (ci, cj) in pairs:
            resolvents = resolve(ci, cj)

            for resolvent in resolvents:
            if len(resolvent) == 0:
                print(f"Iteration {iteration}: Derived empty clause from:")
print(f"  {ci}")                    print(f"  {cj}")                    print("  → []
(Contradiction found!)")                return True

                    # Check if this is a new clause                    if resolvent not
in clauses and resolvent not in new_clauses:
                new_clauses.append(resolvent)

        if not new_clauses:
            print(f"Iteration {iteration}: No new clauses derived. Query cannot be proved.")
return False

        print(f"Iteration {iteration}: Generated {len(new_clauses)} new clause(s)")
for clause in new_clauses:            clauses.append(clause)

# ------------------------------
# Example Usage
# ------------------------------
if __name__ == "__main__":
        # Define knowledge base
    kb = [
        # John likes all food: Food(x) => Likes(John, x)
        [Predicate("Food", ("x",), negated=True), Predicate("Likes", ("John", "x"))],

        # Food(Apple)
        [Predicate("Food", ("Apple",))],
```

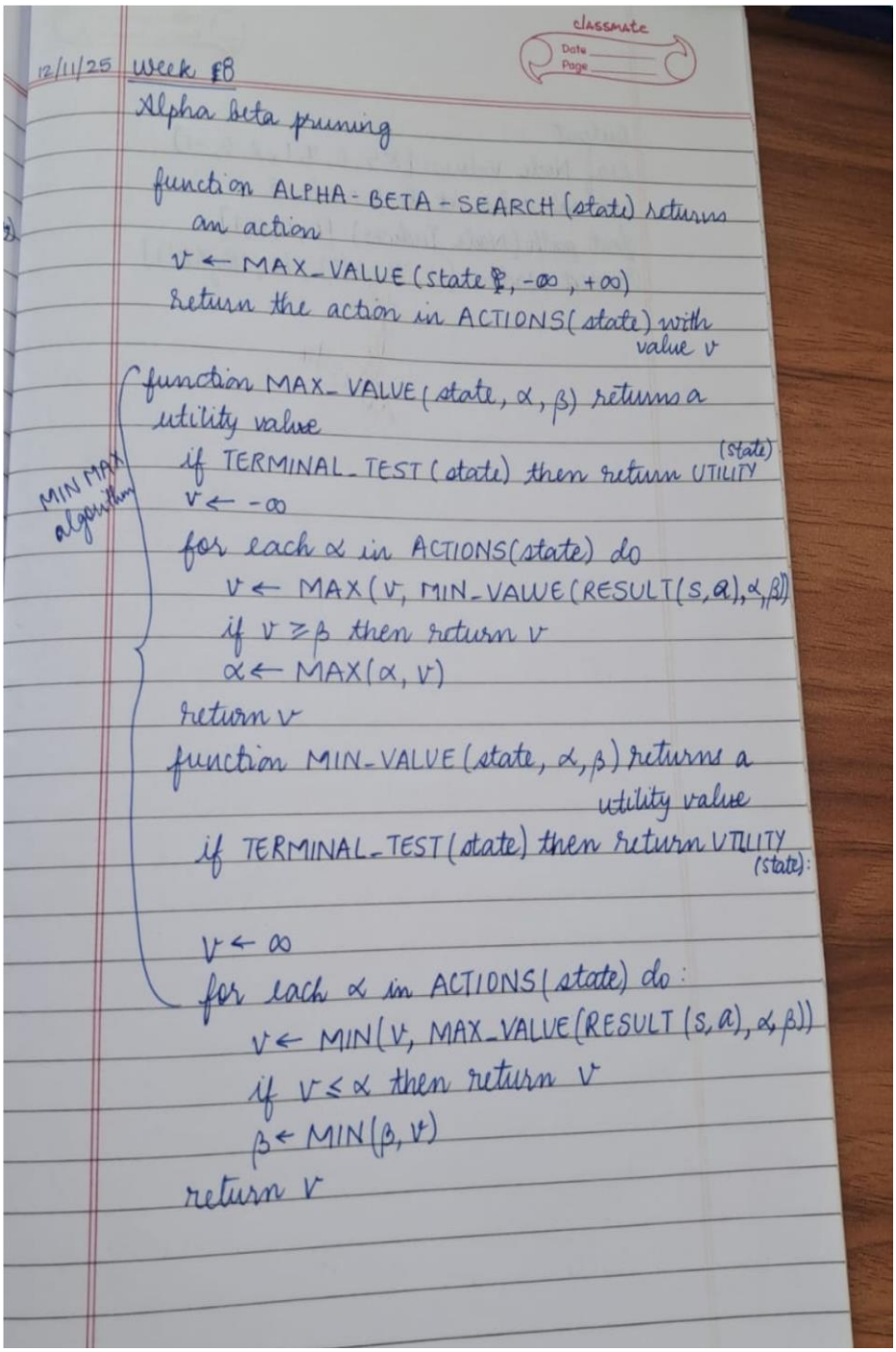
```
        # Food(Vegetables)
        [Predicate("Food", ("Vegetables",))],

        # Eats(Anil, Peanuts)
        [Predicate("Eats", ("Anil", "Peanuts"))],

        # Alive(Anil)
        [Predicate("Alive", ("Anil",))],

        # If alive and eats something, that thing is food: Alive(x) ∧ Eats(x,y) => Food(y)
        [Predicate("Alive", ("x",), negated=True),
         Predicate("Eats", ("x", "y"), negated=True),
         Predicate("Food", ("y",))],

        # Harry eats everything Anil eats: Eats(Anil,y) => Eats(Harry,y)
        [Predicate("Eats", ("Anil", "y"), negated=True),
         Predicate("Eats", ("Harry", "y"))]
          ]

          # Query: Does John like Peanuts?
    query = Predicate("Likes", ("John", "Peanuts"))

    print("=" * 60)
    print("FIRST-ORDER LOGIC RESOLUTION THEOREM PROVER")
print("=" * 60)     print(f"\nQuery: {query}")
    print("-" * 60)

    result = fol_resolution(kb, query)

    print("\n" + "=" * 60)
if result:
        print("✅Query is PROVED using resolution!")
else:
        print("❌Query CANNOT be proved.")
print("=" * 60)
```

Output:

```
============================================================
FIRST-ORDER LOGIC RESOLUTION THEOREM PROVER
============================================================

Query: Likes(John,Peanuts)
------------------------------------------------------------

Knowledge Base + Negated Query:
  1. [~Food(x), Likes(John,x)]
  2. [Food(Apple)]
  3. [Food(Vegetables)]
  4. [Eats(Anil,Peanuts)]
  5. [Alive(Anil)]
  6. [~Alive(x), ~Eats(x,y), Food(y)]
  7. [~Eats(Anil,y), Eats(Harry,y)]
  8. [~Likes(John,Peanuts)]

Iteration 1: Generated 8 new clause(s)
Iteration 2: Generated 16 new clause(s)
Iteration 3: Derived empty clause from:
  [Eats(Anil,Peanuts)]
  [~Eats(Anil,Peanuts)]
  → [] (Contradiction found!)

============================================================
✅ Query is PROVED using resolution!
============================================================
```

Program 10 Implement Alpha-Beta Pruning.

Algorithm:



12/11/25 Week 8

Alpha beta pruning

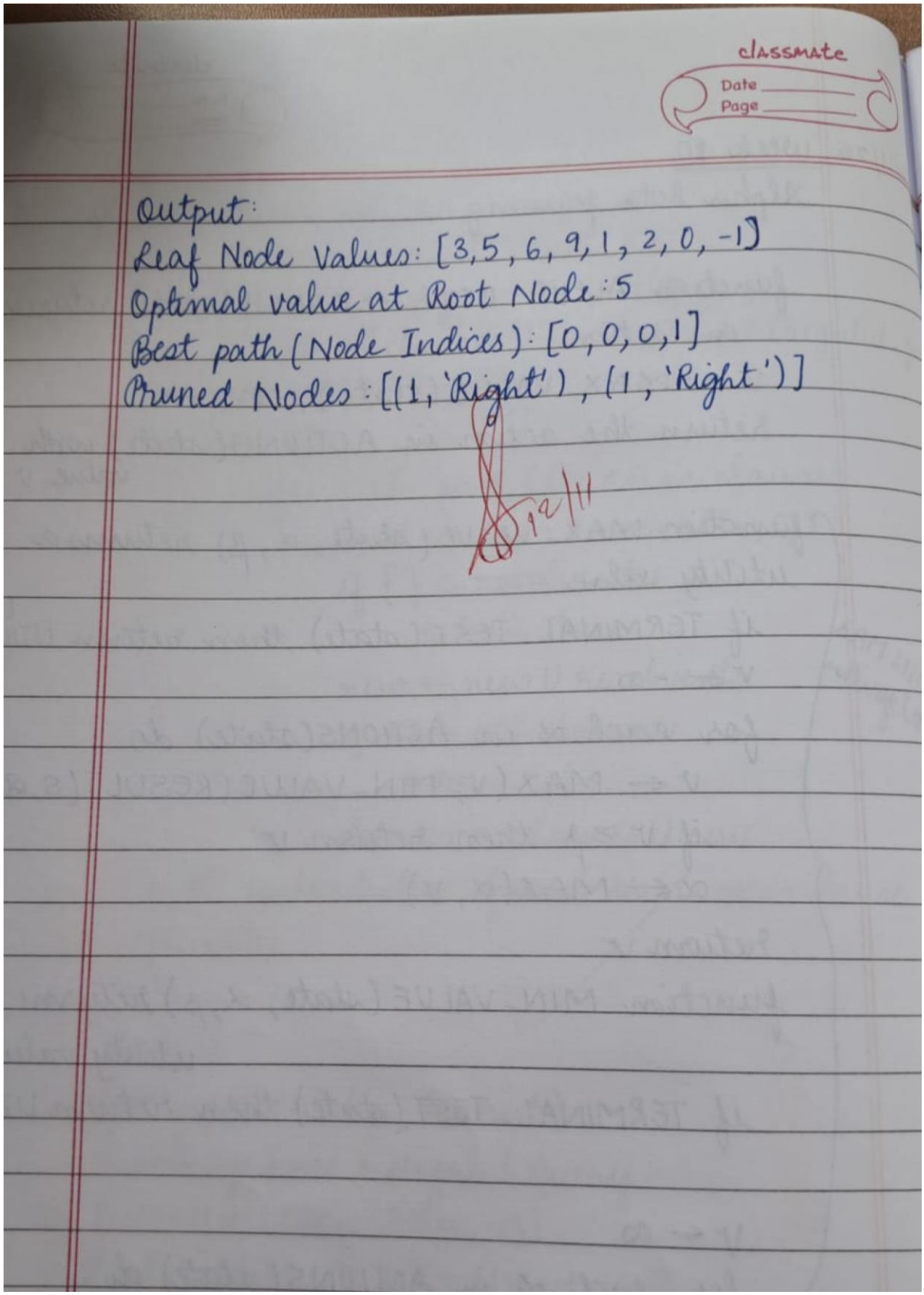
function ALPHA-BETA-SEARCH (state) returns an action
v ← MAX_VALUE (state, -∞, +∞)
return the action in ACTIONS(state) with value v

function MAX_VALUE (state, α, β) returns a utility value
if TERMINAL_TEST (state) then return UTILITY (state)
v ← -∞
for each α in ACTIONS(state) do
v ← MAX(v, MIN_VALUE(RESULT(s,a), α, β))
if v ≥ β then return v
α ← MAX(α, v)
return v

MIN MAX algorithm

function MIN_VALUE (state, α, β) returns a utility value
if TERMINAL_TEST (state) then return UTILITY (state):

v ← ∞
for each α in ACTIONS(state) do:
v ← MIN(v, MAX_VALUE(RESULT(s,a), α, β))
if v ≤ α then return v
β ← MIN(β, v)
return v


classmate
Date
Page
Output:
Leaf Node Values: [3,5,6,9,1,2,0,-1]
Optimal value at Root Node: 5
Best path (Node Indices): [0,0,0,1]
Pruned Nodes: [(1, 'Right'), (1, 'Right')]
9/2/11

Code:

```
import math
# Alpha-Beta Pruning Algorithm def alpha_beta(depth, node_index, maximizing_player, values, alpha, beta, max_depth, path, pruned):
        # Base case: leaf node
if depth == max_depth:
    return values[node_index], [node_index]

  if maximizing_player:
    best = -math.inf
best_path = []
    for i in range(2):  # two children per node          val, child_path = alpha_beta(depth + 1, node_index * 2 + i, False, values, alpha, beta, max_depth, path, pruned)          if val > best:
                best = val                best_path = [node_index] + child_path
         alpha = max(alpha, best)
if beta <= alpha:
                pruned.append((node_index, "Right" if i == 0 else "Left"))
        break
    return best, best_path
else:
    best = math.inf
best_path = []       for i in range(2):
         val, child_path = alpha_beta(depth + 1, node_index * 2 + i, True, values, alpha, beta, max_depth, path, pruned)          if val < best:
                best = val                best_path = [node_index] + child_path          beta = min(beta, best)          if beta <= alpha:
                pruned.append((node_index, "Right" if i == 0 else "Left"))
        break
    return best, best_path


# Example usage if __name__ == "__main__":
        # Example game tree (leaf node values)
  values = [3, 5, 6, 9, 1, 2, 0, -1]

  print("Leaf Node Values:", values)
  path = []
  pruned = []

  max_depth = 3     result, best_path = alpha_beta(0, 0, True, values, -math.inf, math.inf, max_depth, path, pruned)
```

print("\nOptimal Value at Root Node:", result)
print("Best Path (Node Indices):", best_path)
print("Pruned Nodes:", pruned) Output:

```
Leaf Node Values: [3, 5, 6, 9, 1, 2, 0, -1]

Optimal Value at Root Node: 5
Best Path (Node Indices): [0, 0, 0, 1]
Pruned Nodes: [(1, 'Right'), (1, 'Right')]
```