# VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

**LAB REPORT**
**on**

# Artificial Intelligence (23CS5PCAIN)

***Submitted by***

**Ranjan Devi (1BM22CS219)**

***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 **Ranjan Devi (1BM22CS219),** who is Bonafide student of **B.M.S. College of Engineering.** It is in partial fulfilment 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.

| | |
|---|---|
| Radhika A D<br>Assistant Professor<br>Department of CSE, BMSCE | Dr. Kavitha Sooda<br>Professor & HOD<br>Department of CSE, BMSCE |

# Index

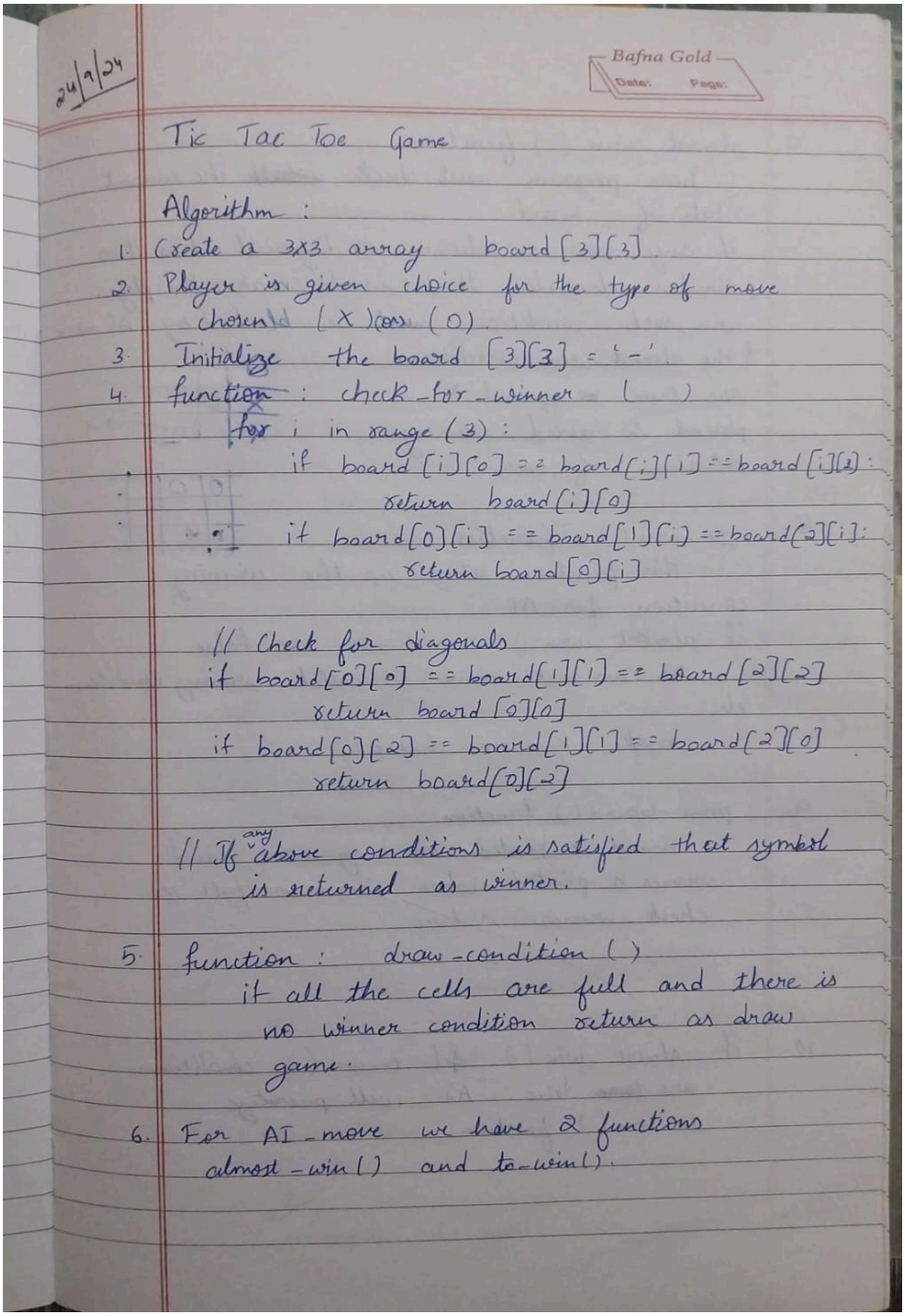


**Github Link:**

https://github.com/ranjandevi219/AI

**Program 1**
**Implement Tic –Tac –Toe Game**

**Algorithm:**



24/9/24

Bafna Gold
Date: Page:

Tic Tac Toe Game

Algorithm :

1. Create a 3x3 array board[3][3].
2. Player is given choice for the type of move chosen (X) or (O).
3. Initialize the board [3][3] = '-'.
4. function : check_for_winner ( )
   for i in range (3):
   if board[i][0] == board[i][1] == board[i][2]:
   return board[i][0]
   if board[0][i] == board[1][i] == board[2][i]:
   return board[0][i]

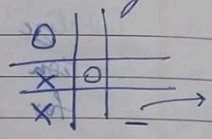
   // Check for diagonals
   if board[0][0] == board[1][1] == board[2][2]
   return board[0][0]
   if board[0][2] == board[1][1] == board[2][0]
   return board[0][2]

   // If any above conditions is satisfied that symbol is returned as winner.

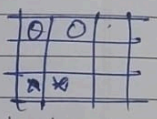
5. function : draw_condition ( )
   if all the cells are full and there is no winner condition return as draw game.

6. For AI_move we have 2 functions almost_win() and to_win().

7. almost_win() function
here program must check the current state of board.
if any row or column or diagonal has two similar elements that would make the player win, such condition must be blocked by AI move.
if the almost_win conditions are true symbol 'X' is placed to avoid winning situation of user.



8. to_win() function
this function brings up the winning condition for AI.
if almost_win() conditions are false
AI can make move to match winning condition.
else
make random choice.



9. print_board() function
board is printed on every move and winner is printed when board is full or check_winner is true.

10. if almost_win() & to_win() conditions are true AI will prioritize to_win() function.

**Code:**

```
import random
def print_board(board):
    for row in board:
        print(" | ".join(row))
```

```
        print("-" * 5)

def check_winner(board):
    # Check rows and columns
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] and board[i][0] != " ":
            return board[i][0]
        if board[0][i] == board[1][i] == board[2][i] and board[0][i] != " ":
            return board[0][i]
    # Check diagonals
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != " ":
        return board[0][0]
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != " ":
        return board[0][2]
    # Check for a tie
    for row in board:
        if " " in row:
            return None
    return "Tie"

def find_best_move(board):
    # A simple AI to choose the first available move or block the player
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                # Simulate AI move and check if it leads to a win
                board[i][j] = "O"
                if check_winner(board) == "O":
                    board[i][j] = " "  # Undo move
                    return (i, j)
                board[i][j] = " "
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                # Simulate user move and block if they can win
                board[i][j] = "X"
                if check_winner(board) == "X":
                    board[i][j] = " "  # Undo move
                    return (i, j)
                board[i][j] = " "
```

```
    # Otherwise, pick the first empty spot
    for i in range(3):
        for j in range(3):
            if board[i][j] == " ":
                return (i, j)

def main():
    board = [[" " for _ in range(3)] for _ in range(3)]
    print("Welcome to Tic Tac Toe!")
    print("You are X and AI is O.")
    while True:
        print_board(board)
        winner = check_winner(board)
        if winner:
            if winner == "Tie":
                print("It's a tie!")
            else:
                print(f"{winner} wins!")
            break
        # User's turn
        while True:
            try:
                user_input = input("Enter your move (row and column: 1 1 for top-left): ").split()
                row, col = int(user_input[0]) - 1, int(user_input[1]) - 1
                if board[row][col] == " ":
                    board[row][col] = "X"
                    break
                else:
                    print("Invalid move, try again.")
            except (ValueError, IndexError):
                print("Invalid input. Enter row and column as two numbers between 1 and 3.")

        # Check for winner after user's move
        if check_winner(board):
            continue

        # AI's turn
        move = find_best_move(board)
        if move:
            board[move[0]][move[1]] = "O"
```

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

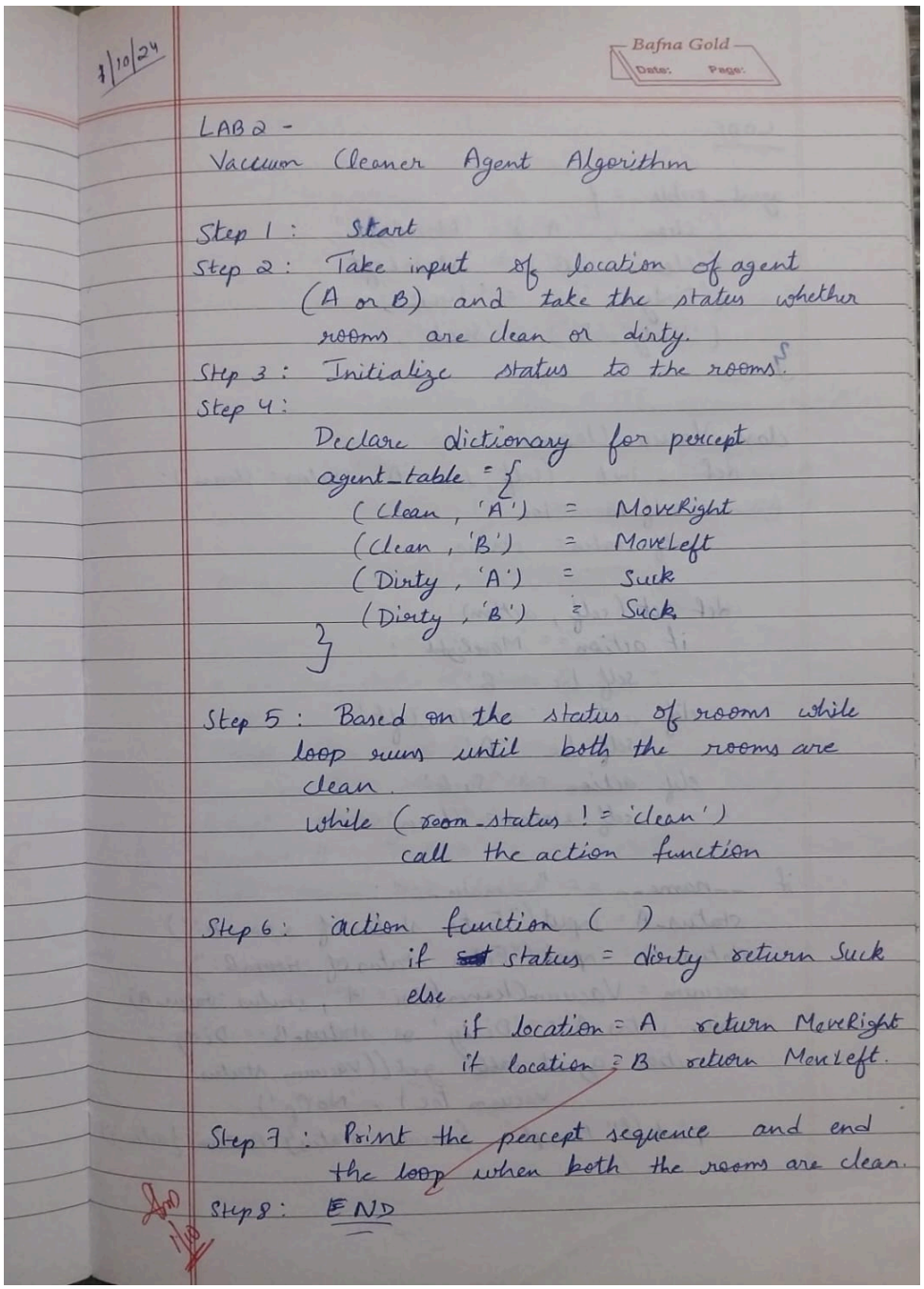
**Output:**

```
Ranjan Devi-1BM22CS219
Welcome to Tic Tac Toe!
You are X and AI is O.
  |   |
--------
  |   |
--------
  |   |
--------
Enter your move (row and column: 1 1 for top-left): 1 1
X | O |
--------
  |   |
--------
  |   |
--------
Enter your move (row and column: 1 1 for top-left): 2 2
X | O |
--------
  | X |
--------
  |   | O
--------
```

```
Enter your move (row and column: 1 1 for top-left): 3 1
X | O | O
--------
  | X |
--------
X |   | O
--------
Enter your move (row and column: 1 1 for top-left): 2 1
X | O | O
--------
X | X |
--------
X |   | O
--------
X wins!
```

**Implement vacuum cleaner agent**

**Algorithm:**



7/10/24

Bafna Gold
Date: Page:

LAB 2 -
Vacuum Cleaner Agent Algorithm

Step 1 : Start
Step 2 : Take input of location of agent (A or B) and take the status whether rooms are clean or dirty.
Step 3 : Initialize status to the rooms.
Step 4 :
Declare dictionary for percept
agent_table = {
(Clean, 'A') = MoveRight
(Clean, 'B') = MoveLeft
(Dirty, 'A') = Suck
(Dirty, 'B') = Suck
}

Step 5 : Based on the status of rooms while loop runs until both the rooms are clean.
while (room_status != 'clean')
call the action function

Step 6 : action function ( )
if status = dirty return Suck
else
if location = A return MoveRight
if location = B return MoveLeft.

Step 7 : Print the percept sequence and end the loop when both the rooms are clean.
Step 8 : END

**Code:**

#For two quadrants
def vacuum_cleaner_simulation():

```
    current_room = input("Enter current room either A or B: ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))

    cost = 0

    def display_rooms():
        print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
        print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")

    print("\nInitial status of rooms:")
    display_rooms()
    print()

    while room_A == 1 or room_B == 1:
        if current_room == 'A' and room_A == 1:
            print("Cleaning Room A...")
            room_A = 0
            cost += 1
        elif current_room == 'B' and room_B == 1:
            print("Cleaning Room B...")
            room_B = 0
            cost += 1
        else:
            current_room = 'B' if current_room == 'A' else 'A'
            print(f"Moving to Room {current_room}...")
        print("Current status:")
        display_rooms()

    print(f"\nBoth rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()

#For four quadrants
def vacuum_cleaner_simulation():
    current_room = input("Enter current room (A, B, C, or D): ").upper()
    room_A = int(input("Is Room A dirty? (yes:1/no:0): "))
    room_B = int(input("Is Room B dirty? (yes:1/no:0): "))
    room_C = int(input("Is Room C dirty? (yes:1/no:0): "))
    room_D = int(input("Is Room D dirty? (yes:1/no:0): "))

    cost = 0
    count=2
    def display_rooms():
        print(f"Room A: {'Clean' if room_A == 0 else 'Dirty'}")
        print(f"Room B: {'Clean' if room_B == 0 else 'Dirty'}")
        print(f"Room C: {'Clean' if room_C == 0 else 'Dirty'}")
```

```
    print(f"Room D: {'Clean' if room_D == 0 else 'Dirty'}")

  print("\nInitial status of rooms:")
  display_rooms()
  print()

  while room_A == 1 or room_B == 1 or room_C == 1 or room_D == 1:
    if count==0:
      print("Vacuum is recharging")
      count=2
    else:
      if current_room == 'A' and room_A == 1:
        print("Cleaning Room A...")
        room_A = 0
        cost += 1
        count-=1
      elif current_room == 'B' and room_B == 1:
        print("Cleaning Room B...")
        room_B = 0
        cost += 1
        count-=1
      elif current_room == 'C' and room_C == 1:
        print("Cleaning Room C...")
        room_C = 0
        cost += 1
        count-=1
      elif current_room == 'D' and room_D == 1:
        print("Cleaning Room D...")
        room_D = 0
        cost += 1
        count-=1
      else:
        if current_room == 'A':
          current_room = 'B'
        elif current_room == 'B':
          current_room = 'C'
        elif current_room == 'C':
          current_room = 'D'
        else:
          current_room = 'A'
        print(f"Moving to Room {current_room}...")

  print("\nCurrent status:")
  display_rooms()
  print(f"\nAll rooms are now clean! Total cost: {cost}")

vacuum_cleaner_simulation()
```

**Output:**

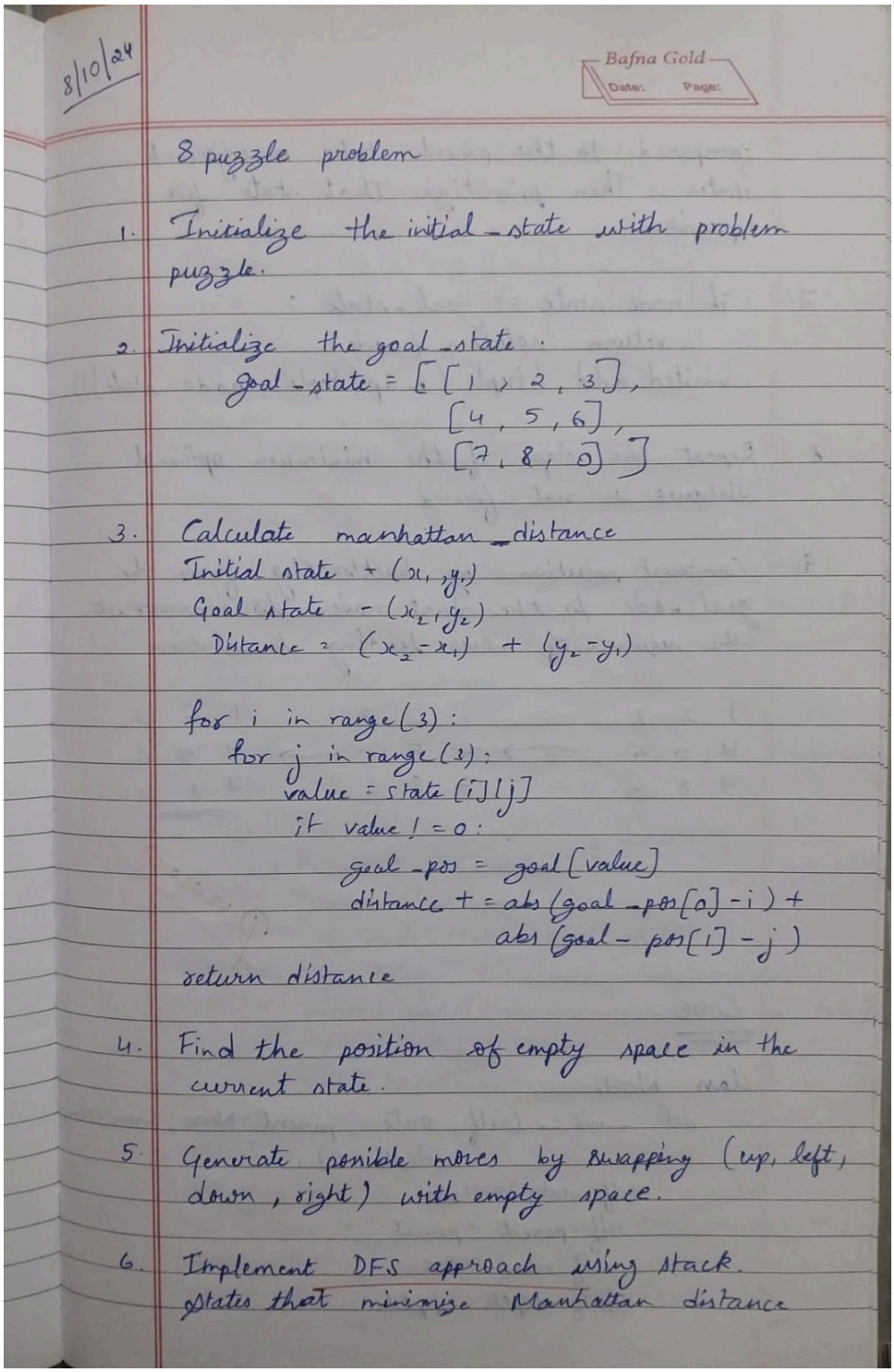
```
Enter current room either A or B: A
Is Room A dirty? (yes:1/no:0): 1
Is Room B dirty? (yes:1/no:0): 0

Initial status of rooms:
Room A: Dirty
Room B: Clean

Cleaning Room A...
Current status:
Room A: Clean
Room B: Clean

Both rooms are now clean! Total cost: 1
Ranjan Devi-1BM22CS219
Enter current room (A, B, C, or D): C
Is Room A dirty? (yes:1/no:0): 00
Is Room B dirty? (yes:1/no:0): 0
Is Room C dirty? (yes:1/no:0): 1
Is Room D dirty? (yes:1/no:0): 1
```

```
Initial status of rooms:
Room A: Clean
Room B: Clean
Room C: Dirty
Room D: Dirty

Cleaning Room C...
Moving to Room D...
Cleaning Room D...

Current status:
Room A: Clean
Room B: Clean
Room C: Clean
Room D: Clean

All rooms are now clean! Total cost: 2
```

**Program 2**

**Implement 8 puzzle problems using Depth First Search (DFS)**

**Algorithm:**



8/10/24

Bafna Gold

Date: Page:

8 puzzle problem

1. Initialize the initial_state with problem puzzle.

2. Initialize the goal_state :

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

3. Calculate manhattan_distance

Initial state = $(x_1, y_1)$

Goal state = $(x_2, y_2)$

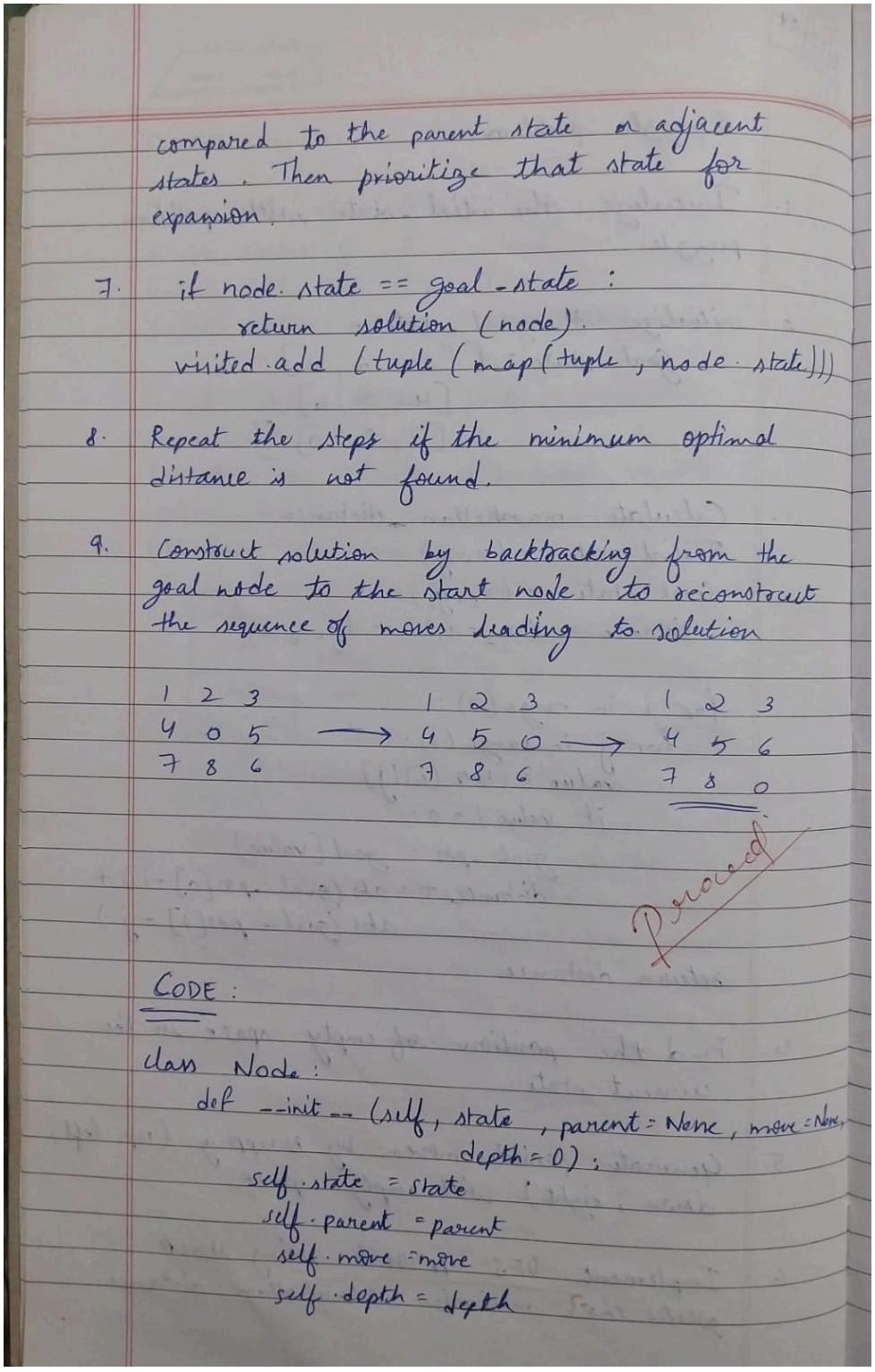
Distance = $(x_2 - x_1) + (y_2 - y_1)$

```
for i in range(3):
    for j in range(3):
        value = state[i][j]
        if value != 0:
            goal_pos = goal[value]
            distance += abs(goal_pos[0] - i) +
                        abs(goal_pos[1] - j)
return distance
```

4. Find the position of empty space in the current state.

5. Generate posible moves by swapping (up, left, down, right) with empty space.

6. Implement DFS approach using stack. States that minimize Manhattan distance

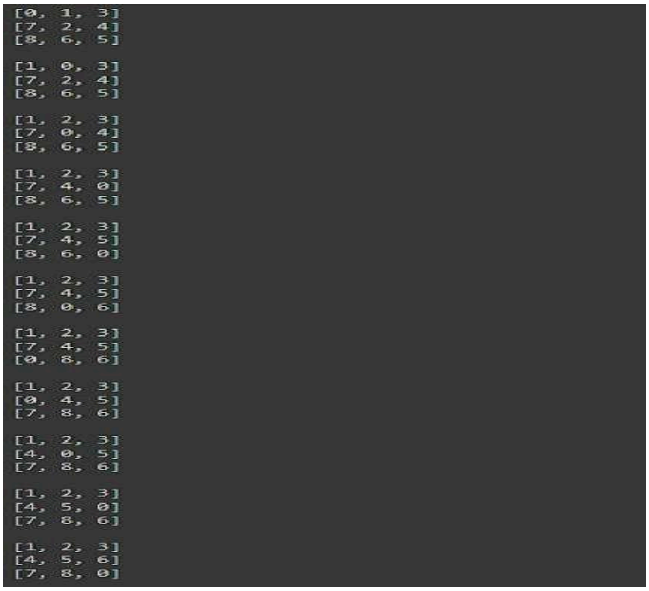



**Code:**

```
def dfs(initial_board, zero_pos):
    stack = [(initial_board, zero_pos, [])]
    visited = set()
```

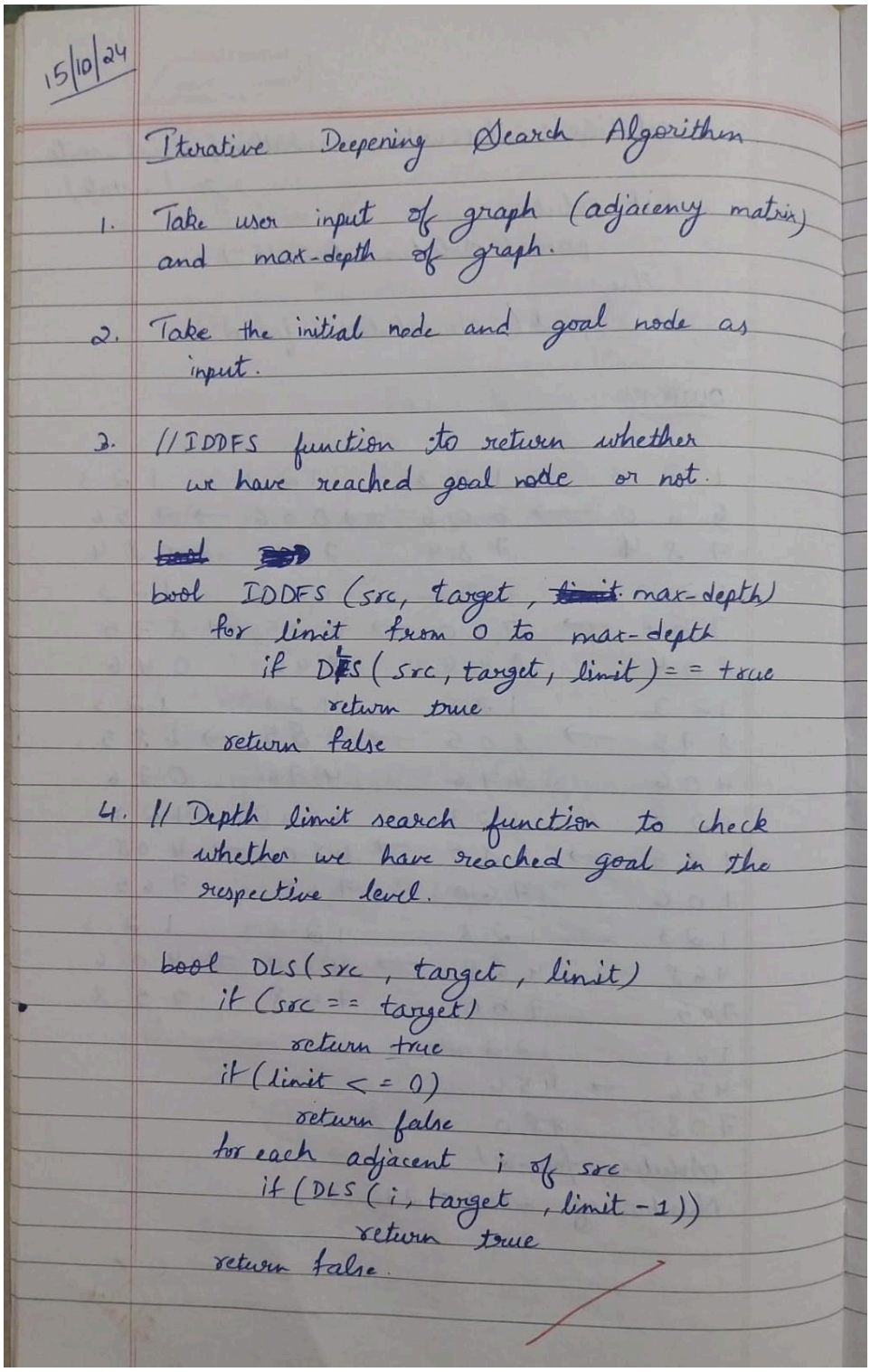
```
  while stack:
      current_board, zero_pos, moves = stack.pop()
      if is_goal(current_board):
          return moves, len(moves) # Return moves and their count

      visited.add(tuple(current_board))

      for neighbor_board, neighbor_pos in get_neighbors(current_board, zero_pos):
          if tuple(neighbor_board) not in visited:
              stack.append((neighbor_board, neighbor_pos, moves + [neighbor_board])

    return None, 0 # No solution found, return count as 0
# Initial state of the puzzle
initial_board = [1, 2, 3, 0, 4, 6, 7, 5,
8]
zero_position = (1, 0) # Position of the empty tile (0)

# Solve the puzzle using DFS
solution, move_count = dfs(initial_board, zero_position)

if solution:
    print("Solution found with moves ({}
    moves):".format(move_count)) for move in solution:
        print_board(move)
        print() # Print an empty line between moves
else:
    print("No solution found.")
```

**Output:**



**Implement Iterative deepening search algorithm**

**Algorithm:**



15/10/24

Iterative Deepening Search Algorithm

1. Take user input of graph (adjacency matrix) and max-depth of graph.

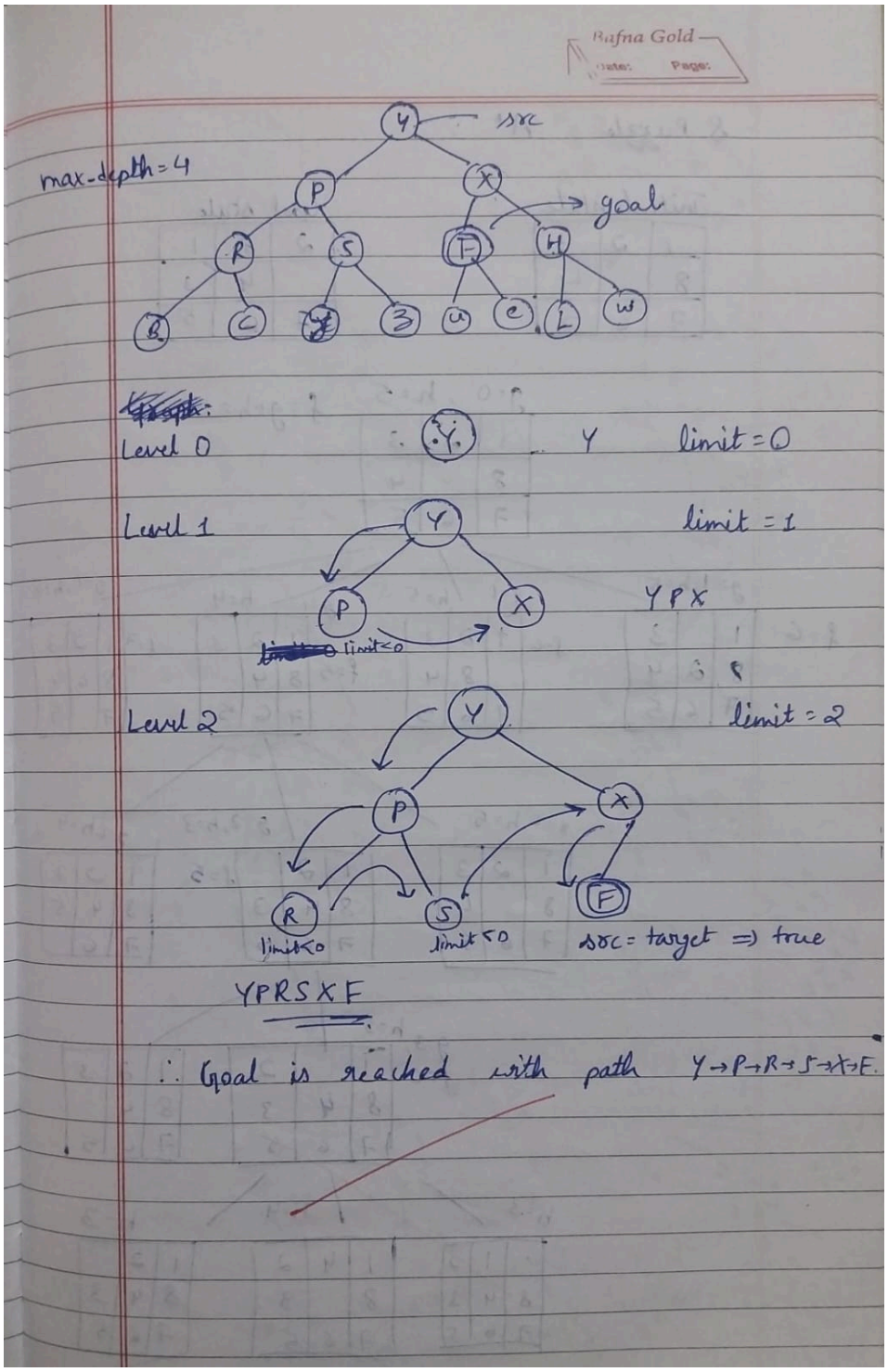
2. Take the initial node and goal node as input.

3. // IDDFS function to return whether we have reached goal node or not.

```
bool IDDFS (src, target, max_depth)
    for limit from 0 to max_depth
        if DLS(src, target, limit) == true
            return true
    return false
```

4. // Depth limit search function to check whether we have reached goal in the respective level.

```
bool DLS(src, target, limit)
    if (src == target)
        return true
    if (limit <= 0)
        return false
    for each adjacent i of src
        if (DLS(i, target, limit-1))
            return true
    return false
```




**Code:**

```
import copy
def is_goal(state, goal):
    return state == goal
def get_neighbors(state):
    neighbors = []
```

```
    blank_row, blank_col = next((r, c) for r, row in enumerate(state) for c, val in enumerate(row) if val
== 0)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]  # Up, Down, Left, Right

    for dr, dc in directions:
        new_row, new_col = blank_row + dr, blank_col + dc
        if 0 <= new_row < 3 and 0 <= new_col < 3:
            new_state = copy.deepcopy(state)
            new_state[blank_row][blank_col], new_state[new_row][new_col] =
new_state[new_row][new_col], new_state[blank_row][blank_col]
            neighbors.append(new_state)

    return neighbors

def dls_with_path(state, goal, depth, path):
    if depth == 0:
        return None
    if is_goal(state, goal):
        return path

    for neighbor in get_neighbors(state):
        if neighbor not in path:
            result = dls_with_path(neighbor, goal, depth - 1, path + [neighbor])
            if result is not None:
                return result

    return None

def iddfs_with_path(initial_state, goal_state):
    depth = 0

    while True:
        result = dls_with_path(initial_state, goal_state, depth, [initial_state])
        if result is not None:
            return result
        depth += 1

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

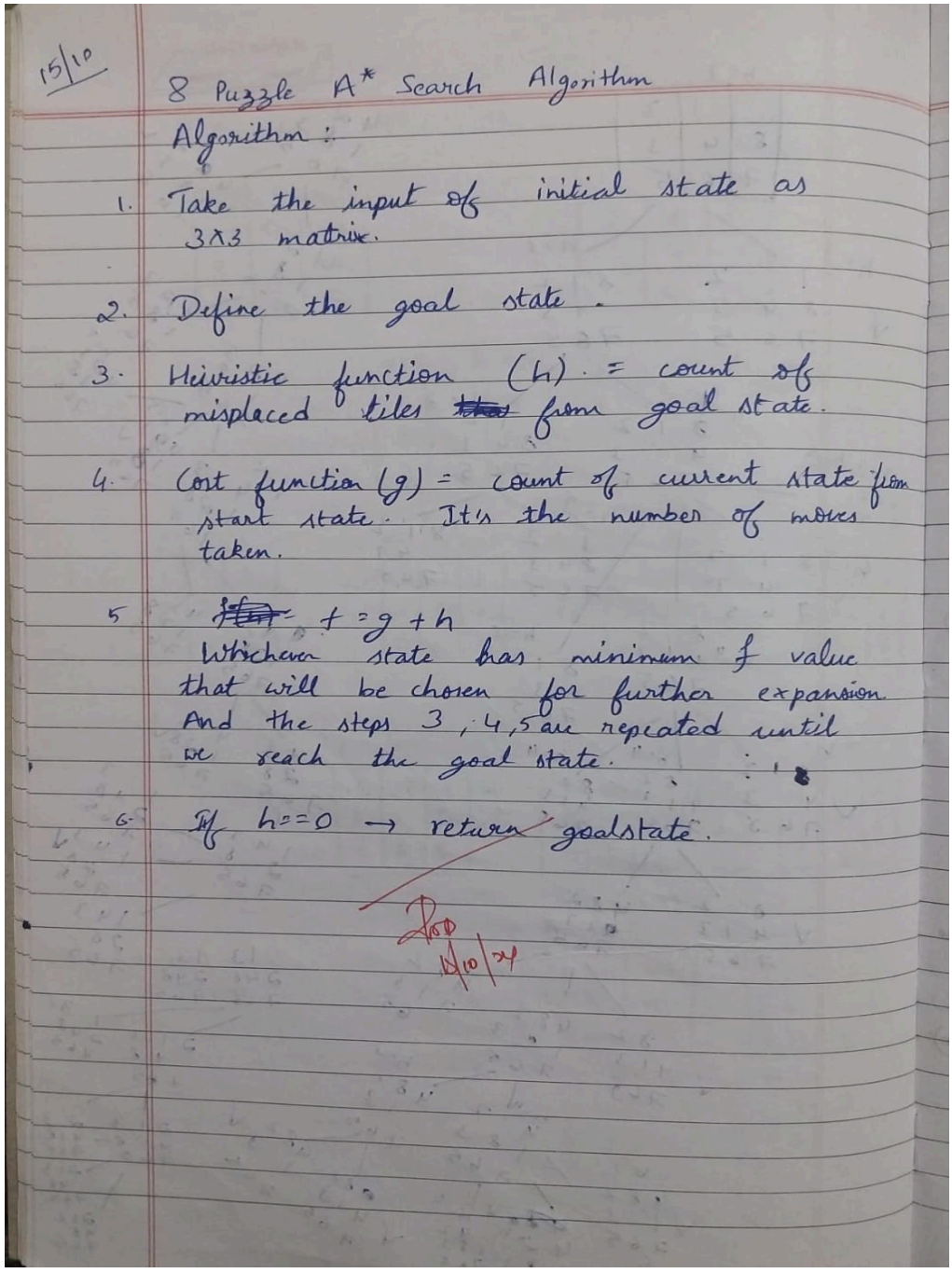
```
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 0]
]

solution = iddfs_with_path(initial_state, goal_state)

if solution:
    print("Intermediate states leading to the goal:")
    for step in solution:
        for row in step:
            print(row)
        print("---")
else:
    print("No solution found.")
```

Output:

```
Intermediate states leading to the goal:
[1, 2, 3]
[4, 0, 5]
[7, 8, 6]
---
[1, 2, 3]
[4, 5, 0]
[7, 8, 6]
---
[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
---
```

**Program 3**
**Implement A* search algorithm**

**Algorithm:**



15/10

8 Puzzle A* Search Algorithm

Algorithm :

1. Take the input of initial state as 3*3 matrix.
2. Define the goal state.
3. Heiristic function (h) = count of misplaced tiles from goal state.
4. Cost function (g) = count of current state from start state. It's the number of moves taken.
5. f = g + h
   Whichever state has minimum f value that will be chosen for further expansion. And the steps 3, 4, 5 are repeated until we reach the goal state.
6. If h==0 → return goalstate.

16/10/24

**Code:**

```
import heapq
```

```
def misplaced_tile(state, goal_state):
    misplaced = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0 and state[i][j] != goal_state[i][j]:
                misplaced += 1
    return misplaced

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))
    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path

def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + misplaced_tile(start, goal), 0, start))
    g_score = {start: 0}
    came_from = {}
    visited = set()
    while open_list:
        _, g, current = heapq.heappop(open_list)
        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g
        visited.add(current)
        for neighbor in generate_neighbors(current):
```

```
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1
            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + misplaced_tile(neighbor, goal)  # f(n) = g(n) + h(n)
                heapq.heappush(open_list, (f_score, tentative_g, neighbor))
    return None, None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_state_from_user(prompt):
    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
    goal_state = get_state_from_user("Goal state")
    print("\nInitial State:")
    print_state(start_state)
    print("\nGoal State:")
    print_state(goal_state)
    solution, cost = a_star(start_state, goal_state)
    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

**Output:**

```
Enter the initial state:
Initial state row 1 (space-separated): 1 2 3
Initial state row 2 (space-separated): 4 0 5
Initial state row 3 (space-separated): 7 8 6

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 4 5 6
Goal state row 3 (space-separated): 7 8 0

Initial State:
(1, 2, 3)
(4, 0, 5)
(7, 8, 6)


Goal State:
(1, 2, 3)
(4, 5, 6)
(7, 8, 0)
```

```

Solution found with cost: 2
Steps:
(1, 2, 3)
(4, 0, 5)
(7, 8, 6)

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

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

Manhattan Distance:

```
import heapq

def manhattan_distance(state, goal_state):
    distance = 0
    for i in range(3):
        for j in range(3):
            value = state[i][j]
            if value != 0:
                goal_i, goal_j = find_position(value, goal_state)
                distance += abs(i - goal_i) + abs(j - goal_j)
    return distance

def find_position(value, state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == value:
                return i, j

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

def generate_neighbors(state):
    neighbors = []
    x, y = find_blank(state)
    directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]

    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            new_state = [list(row) for row in state]
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]
            neighbors.append(tuple(tuple(row) for row in new_state))

    return neighbors

def reconstruct_path(came_from, current):
    path = [current]
    while current in came_from:
        current = came_from[current]
        path.append(current)
    path.reverse()
    return path
```

```
def a_star(start, goal):
    open_list = []
    heapq.heappush(open_list, (0 + manhattan_distance(start, goal), 0, start))

    g_score = {start: 0}
    came_from = {}

    visited = set()

    while open_list:
        _, g, current = heapq.heappop(open_list)

        if current == goal:
            path = reconstruct_path(came_from, current)
            return path, g

        visited.add(current)

        for neighbor in generate_neighbors(current):
            if neighbor in visited:
                continue
            tentative_g = g_score[current] + 1

            if tentative_g < g_score.get(neighbor, float('inf')):
                came_from[neighbor] = current
                g_score[neighbor] = tentative_g
                f_score = tentative_g + manhattan_distance(neighbor, goal)

                heapq.heappush(open_list, (f_score, tentative_g, neighbor))

    return None, None

def print_state(state):
    for row in state:
        print(row)
    print()

def get_state_from_user(prompt):
    state = []
    for i in range(3):
        row = input(f"{prompt} row {i+1} (space-separated): ")
        state.append(tuple(map(int, row.split())))
    return tuple(state)

if __name__ == "__main__":
    print("Enter the initial state:")
    start_state = get_state_from_user("Initial state")
    print("\nEnter the goal state:")
```

```
    goal_state = get_state_from_user("Goal state")

    print("\nInitial State:")
    print_state(start_state)

    print("\nGoal State:")
    print_state(goal_state)

    solution, cost = a_star(start_state, goal_state)

    if solution:
        print(f"\nSolution found with cost: {cost}")
        print("Steps:")
        for step in solution:
            print_state(step)
    else:
        print("\nNo solution found.")
```

**Output:**

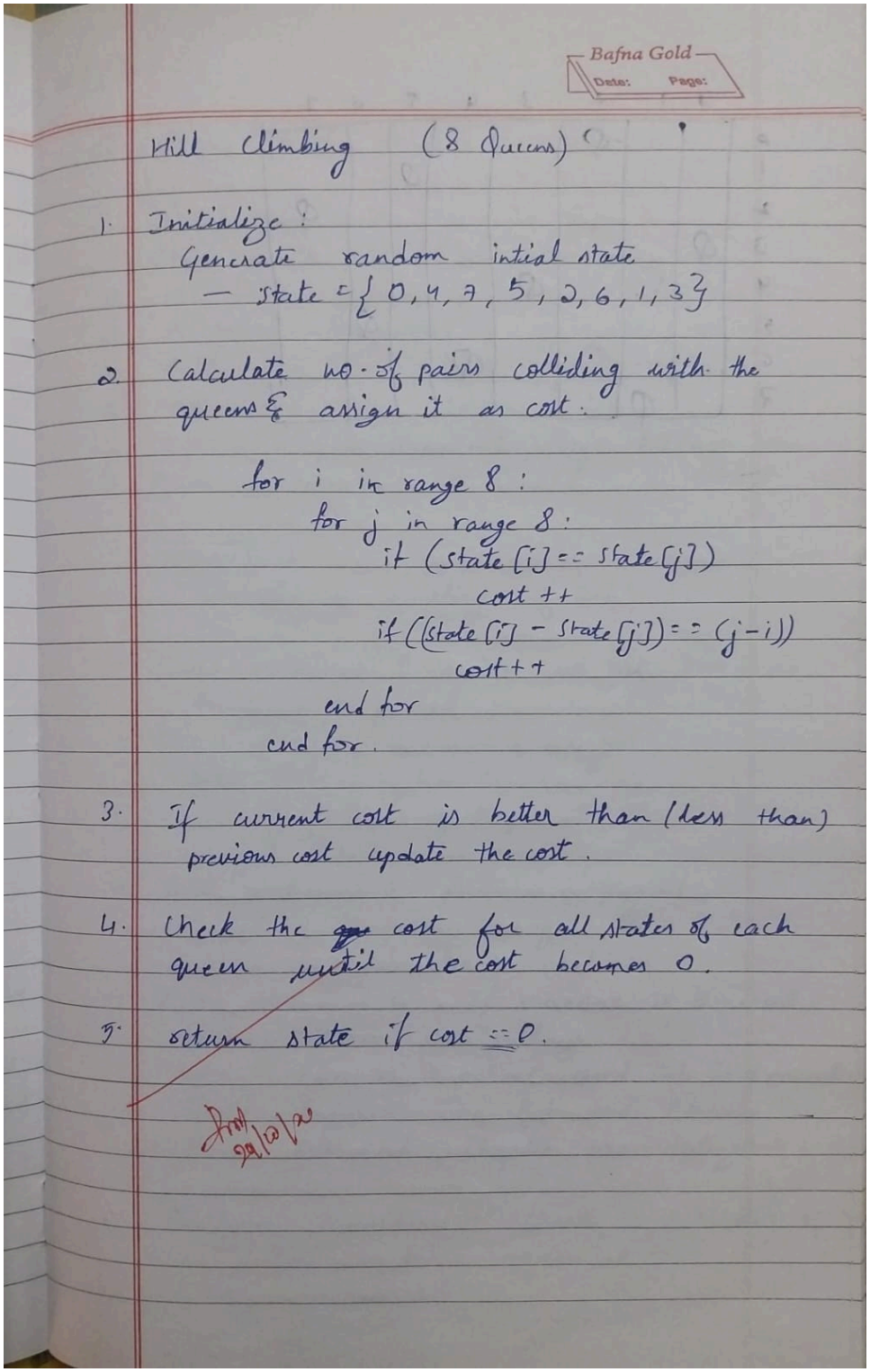
```
Enter the initial state:
Initial state row 1 (space-separated): 2 8 3
Initial state row 2 (space-separated): 1 6 4
Initial state row 3 (space-separated): 7 0 5

Enter the goal state:
Goal state row 1 (space-separated): 1 2 3
Goal state row 2 (space-separated): 8 0 4
Goal state row 3 (space-separated): 7 6 5

Initial State:
(2, 8, 3)
(1, 6, 4)
(7, 0, 5)


Goal State:
(1, 2, 3)
(8, 0, 4)
(7, 6, 5)
```

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

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

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

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

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

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

**Program 4**

**Implement Hill Climbing search algorithm to solve N-Queens problem**

**Algorithm:**



Bafna Gold
Date: Page:

Hill climbing (8 Queens)

1. Initialize:
   Generate random intial state
   - state = {0, 4, 7, 5, 2, 6, 1, 3}

2. Calculate no. of pairs colliding with the queens & assign it as cost.

```
for i in range 8:
    for j in range 8:
        if (state[i] == state[j])
            cost ++
        if ((state[i] - state[j]) == (j-i))
            cost ++
    end for
end for.
```

3. If current cost is better than (less than) previous cost update the cost.

4. Check the cost for all states of each queen until the cost becomes 0.

5. return state if cost == 0.

**Code:**

```
from random import randint
N = int(input("Enter the number of queens:"))
def configureRandomly(board, state):
	for i in range(N):
		state[i] = randint(0, 100000) % N;
		board[state[i]][i] = 1;

def printBoard(board):
	for i in range(N):
		print(*board[i])

def printState( state):
	print(*state)

def compareStates(state1, state2):
	for i in range(N):
		if (state1[i] != state2[i]):
			return False;
	return True;

def fill(board, value):
	for i in range(N):
		for j in range(N):
			board[i][j] = value;

def calculateObjective( board, state):
	attacking = 0;
	for i in range(N):
		row = state[i]
		col = i - 1;
		while (col >= 0 and board[row][col] != 1) :
			col -= 1
		if (col >= 0 and board[row][col] == 1) :
			attacking += 1;
		row = state[i]
		col = i + 1;
		while (col < N and board[row][col] != 1):
			col += 1;
		if (col < N and board[row][col] == 1) :
			attacking += 1;
		row = state[i] - 1
```

```
                col = i - 1;
                while (col >= 0 and row >= 0 and board[row][col] != 1) :
                        col-= 1;
                        row-= 1;
                if (col >= 0 and row >= 0 and board[row][col] == 1) :
                        attacking+= 1;
                row = state[i] + 1
                col = i + 1;
                while (col < N and row < N and board[row][col] != 1) :
                        col+= 1;
                        row+= 1;
                if (col < N and row < N and board[row][col] == 1) :
                        attacking += 1;
                row = state[i] + 1
                col = i - 1;
                while (col >= 0 and row < N and board[row][col] != 1) :
                        col -= 1;
                        row += 1;
                if (col >= 0 and row < N and board[row][col] == 1) :
                        attacking += 1;
                row = state[i] - 1
                col = i + 1;
                while (col < N and row >= 0 and board[row][col] != 1) :
                        col += 1;
                        row -= 1;
                if (col < N and row >= 0 and board[row][col] == 1) :
                        attacking += 1;
        return int(attacking / 2);

def generateBoard( board, state):
        fill(board, 0);
        for i in range(N):
                board[state[i]][i] = 1;

def copyState( state1, state2):
        for i in range(N):
                state1[i] = state2[i];

def getNeighbour(board, state):
        opBoard = [[0 for _ in range(N)] for _ in range(N)]
        opState = [0 for _ in range(N)]
```

```
        copyState(opState, state);
        generateBoard(opBoard, opState);
        opObjective = calculateObjective(opBoard, opState);
        NeighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
        NeighbourState = [0 for _ in range(N)]
        copyState(NeighbourState, state);
        generateBoard(NeighbourBoard, NeighbourState);
        for i in range(N):
                for j in range(N):
                        if (j != state[i]) :
                                NeighbourState[i] = j;
                                NeighbourBoard[NeighbourState[i]][i] = 1;
                                NeighbourBoard[state[i]][i] = 0;
                                temp = calculateObjective( NeighbourBoard, NeighbourState);
                                if (temp <= opObjective) :
                                        opObjective = temp;
                                        copyState(opState, NeighbourState);
                                        generateBoard(opBoard, opState);
                                NeighbourBoard[NeighbourState[i]][i] = 0;
                                NeighbourState[i] = state[i];
                                NeighbourBoard[state[i]][i] = 1;
        copyState(state, opState);
        fill(board, 0);
        generateBoard(board, state);

def hillClimbing(board, state):
        neighbourBoard = [[0 for _ in range(N)] for _ in range(N)]
        neighbourState = [0 for _ in range(N)]
        copyState(neighbourState, state);
        generateBoard(neighbourBoard, neighbourState);
        while True:
                copyState(state, neighbourState);
                generateBoard(board, state);
                getNeighbour(neighbourBoard, neighbourState);
                if (compareStates(state, neighbourState)) :
                        printBoard(board);
                        break;
                elif (calculateObjective(board, state) == calculateObjective(
neighbourBoard,neighbourState)):
                        neighbourState[randint(0, 100000) % N] = randint(0, 100000) % N;
                        generateBoard(neighbourBoard, neighbourState);
```

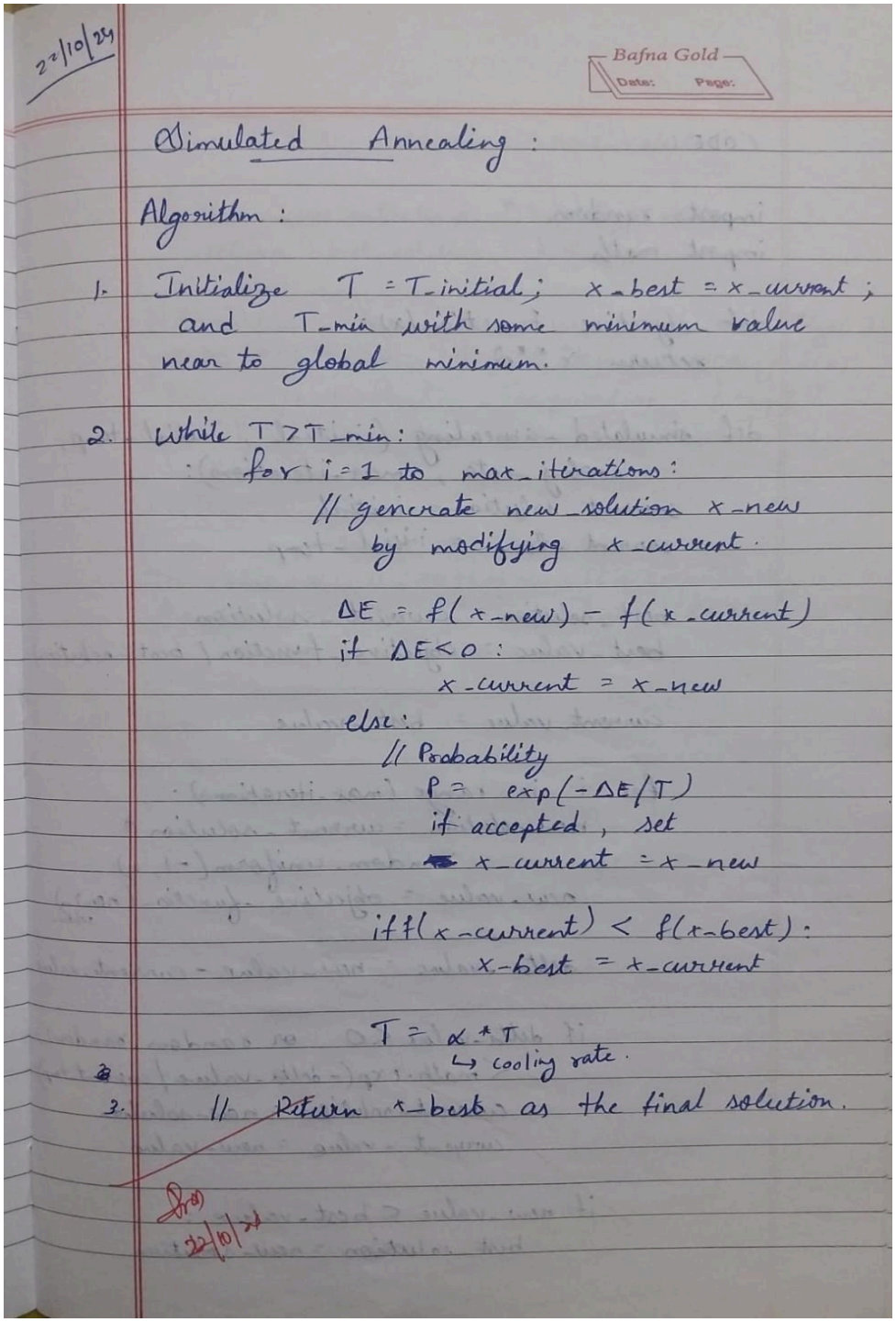
```
# Driver code
state = [0] * N
board = [[0 for _ in range(N)] for _ in range(N)]
configureRandomly(board, state);
hillClimbing(board, state);
```

**Output:**

```
Enter the number of queens:7
0 0 1 0 0 0 0
0 0 0 0 0 1 0
0 1 0 0 0 0 0
0 0 0 0 1 0 0
1 0 0 0 0 0 0
0 0 0 1 0 0 0
0 0 0 0 0 0 1
```

**Program 5**
**Simulated Annealing to Solve 8-Queens problem**

**Algorithm:**




**Code:**

```
import numpy as np
from scipy.optimize import dual_annealing
```

```
def queens_max(position):
    # This function calculates the number of pairs of queens that are not attacking each other
    position = np.round(position).astype(int)  # Round and convert to integers for queen positions
    n = len(position)
    queen_not_attacking = 0

    for i in range(n - 1):
        no_attack_on_j = 0
        for j in range(i + 1, n):
            # Check if queens are on the same row or on the same diagonal
            if position[i] != position[j] and abs(position[i] - position[j]) != (j - i):
                no_attack_on_j += 1
        if no_attack_on_j == n - 1 - i:
            queen_not_attacking += 1
    if queen_not_attacking == n - 1:
        queen_not_attacking += 1

    return -queen_not_attacking  # Negative because we want to maximize this value

# Bounds for each queen's position (0 to 7 for an 8x8 chessboard)
bounds = [(0, 7) for _ in range(8)]

# Use dual_annealing for simulated annealing optimization
result = dual_annealing(queens_max, bounds)

# Display the results
best_position = np.round(result.x).astype(int)
best_objective = -result.fun  # Flip sign to get the number of non-attacking queens

print('The best position found is:', best_position)
print('The number of queens that are not attacking each other is:', best_objective)
```

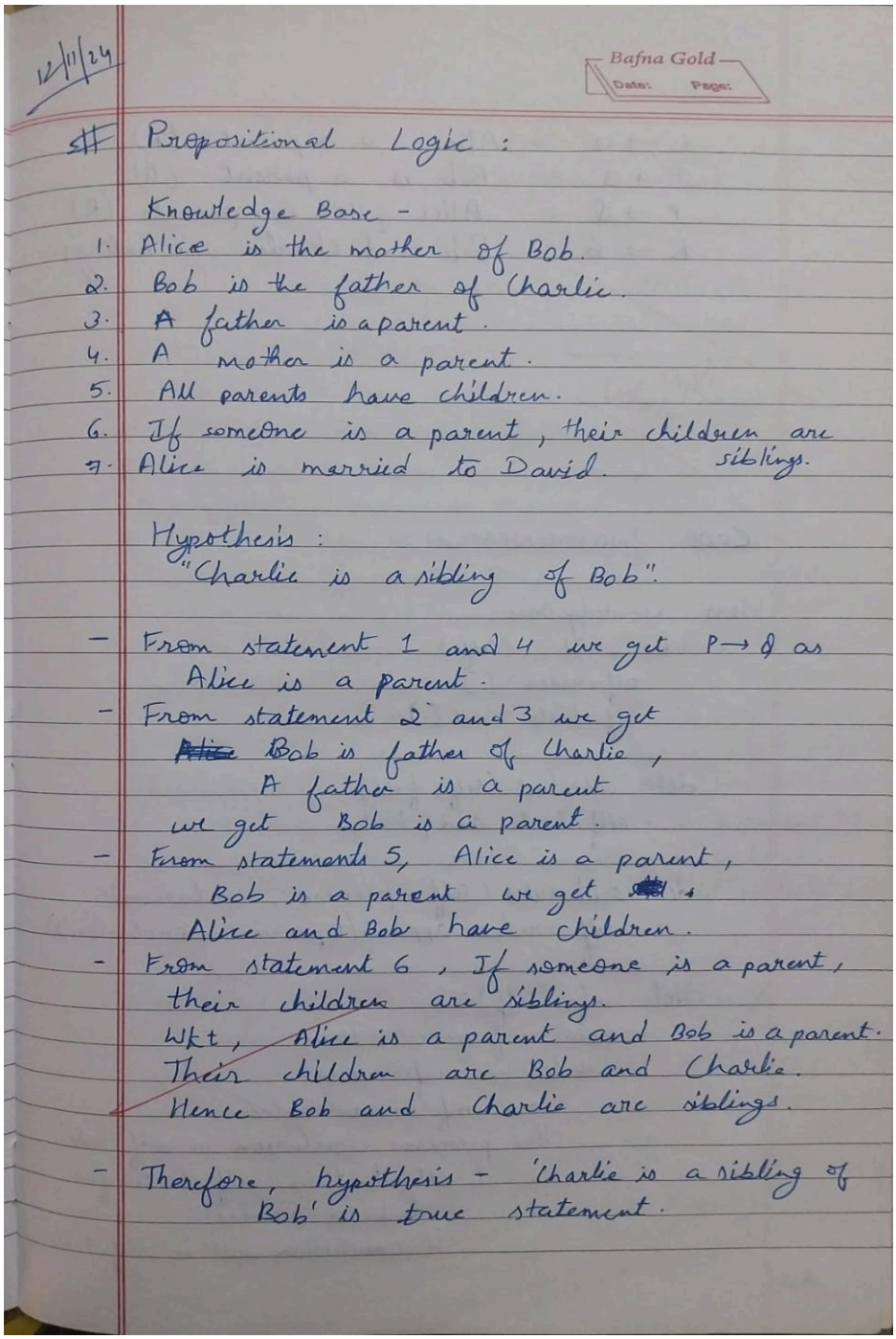
**Output:**

```
The best position found is: [0 8 5 2 6 3 7 4]
The number of queens that are not attacking each other is: 8
```

**Program 6**

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

**Algorithm:**



12/11/24

Bafna Gold

Date: Page:

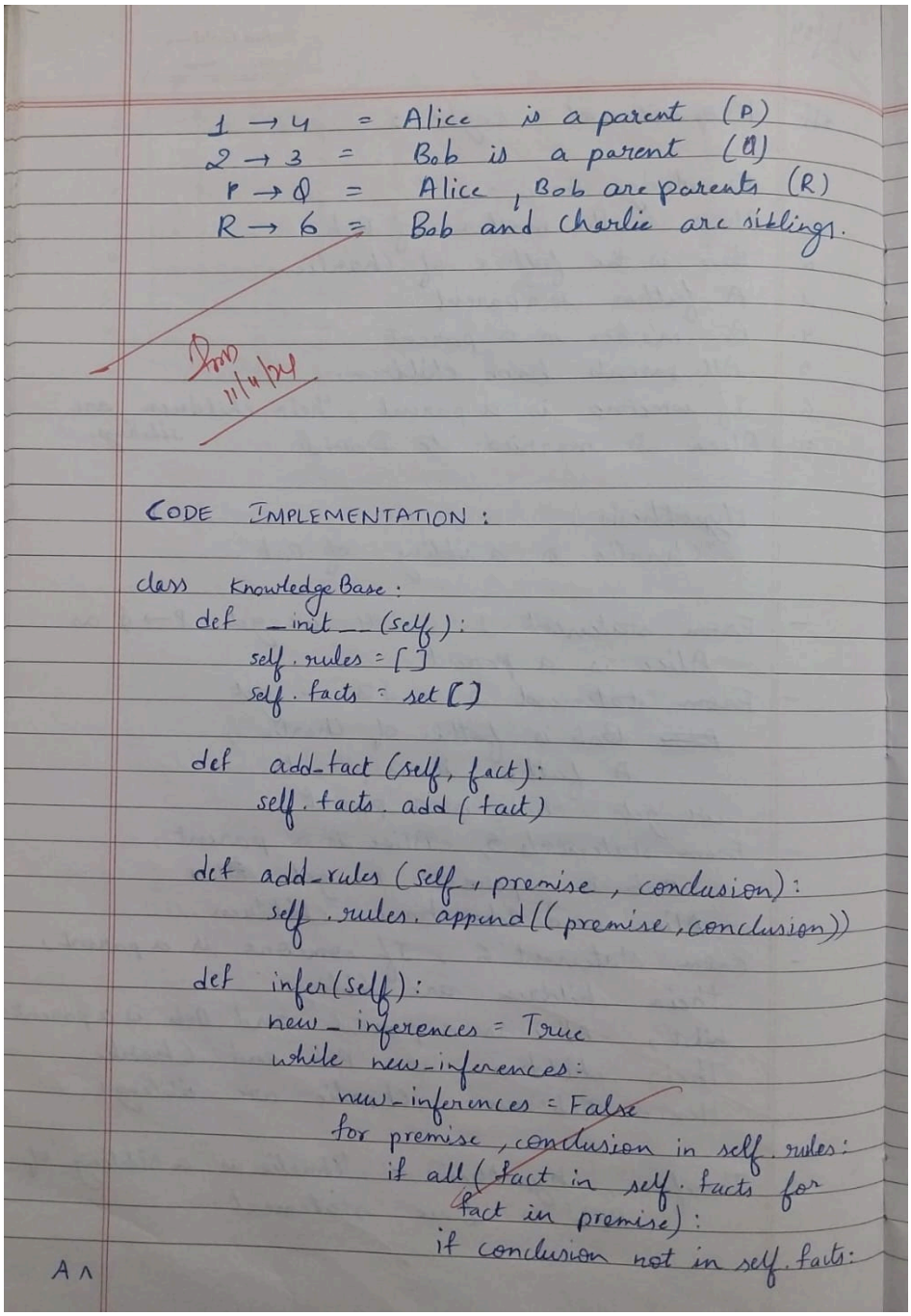
Propositional Logic :

Knowledge Base -

1. Alice is the mother of Bob.
2. Bob is the father of Charlie.
3. A father is a parent.
4. A mother is a parent.
5. All parents have children.
6. If someone is a parent, their children are siblings.
7. Alice is married to David.

Hypothesis :
"Charlie is a sibling of Bob".

- From statement 1 and 4 we get $P \rightarrow Q$ as Alice is a parent.
- From statement 2 and 3 we get Bob is father of Charlie, A father is a parent we get Bob is a parent.
- From statements 5, Alice is a parent, Bob is a parent we get Alice and Bob have children.
- From statement 6, If someone is a parent, their children are siblings.
  Wkt, Alice is a parent and Bob is a parent.
  Their children are Bob and Charlie.
  Hence Bob and Charlie are siblings.
- Therefore, hypothesis - 'Charlie is a sibling of Bob' is true statement.




**Code:**

```
import itertools

def evaluate_formula(formula, valuation):
    env = {var: valuation[i] for i, var in enumerate(['A', 'B', 'C'])}

    # Replace logical operators with Python equivalents
    formula = formula.replace('AND', 'and').replace('OR', 'or').replace('NOT', 'not')
```

```
    # Replace variables in the formula with their corresponding truth values
    for var in env:
        formula = formula.replace(var, str(env[var]))

    # Evaluate the formula and return the result (True or False)
    try:
        return eval(formula)
    except Exception as e:
        raise ValueError(f"Error in evaluating formula: {e}")

def truth_table(variables):
    """
    Generate all possible truth assignments for the given variables.
    """
    return list(itertools.product([False, True], repeat=len(variables)))

def entails(KB, alpha):
    """
    Decide if KB entails alpha using a truth-table enumeration algorithm.
    KB is a propositional formula (string), and alpha is another propositional formula (string).
    """
    # Generate all possible truth assignments for A, B, and C
    assignments = truth_table(['A', 'B', 'C'])

    print(f"{'A':<10}{'B':<10}{'C':<10}{'KB':<15}{'alpha':<15}{'KB entails alpha?'}")  # Header for the truth
table
    print("-" * 70)  # Separator for readability

    for assignment in assignments:
        # Evaluate KB and alpha under the current assignment
        KB_value = evaluate_formula(KB, assignment)
        alpha_value = evaluate_formula(alpha, assignment)
print(f"{str(assignment[0]):<10}{str(assignment[1]):<10}{str(assignment[2]):<10}{str(KB_value):<15}{str(al
pha_value):<15}{'Yes' if KB_value and alpha_value else 'No'}")

        # If KB is true and alpha is false, then KB does not entail alpha
        if KB_value and not alpha_value:
            return False
    return True

# Define the formulas for KB and alpha
alpha = 'A OR B'
KB = '(A OR C) AND (B OR NOT C)'

# Check if KB entails alpha
result = entails(KB, alpha)

# Print the final result of entailment
```

```
print(f"\nDoes KB entail alpha? {result}")
```

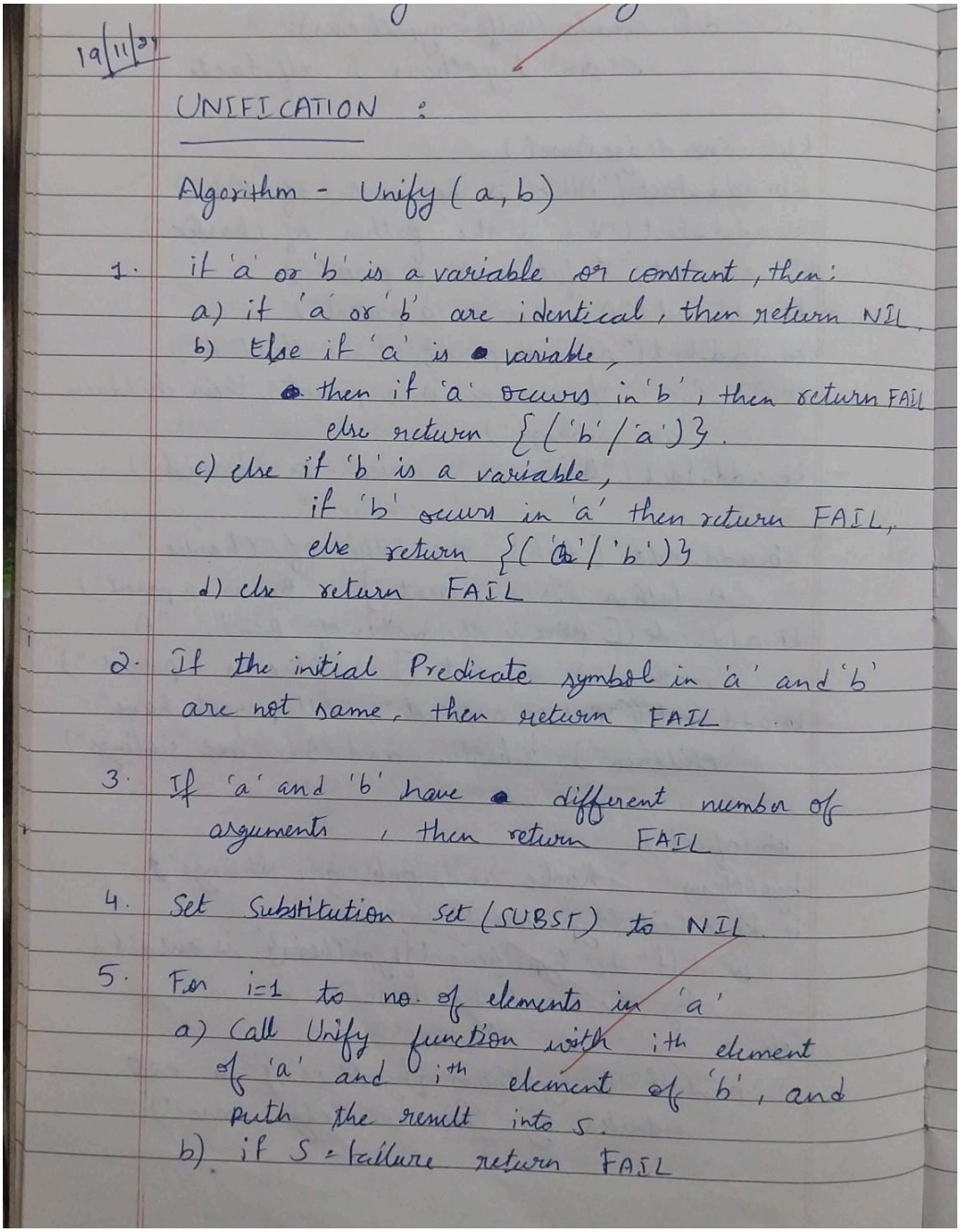
**Output:**

```
A       B       C       KB          alpha       KB entails alpha?
------------------------------------------------------------------------
False   False   False   False       False       No
False   False   True    False       False       No
False   True    False   False       True        No
False   True    True    True        True        Yes
True    False   False   True        True        Yes
True    False   True    False       True        No
True    True    False   True        True        Yes
True    True    True    True        True        Yes

Does KB entail alpha? True
```

**Program 7**

**Implement unification in first order logic**

**Algorithm:**



19/11/21

UNIFICATION :

Algorithm - Unify (a, b)

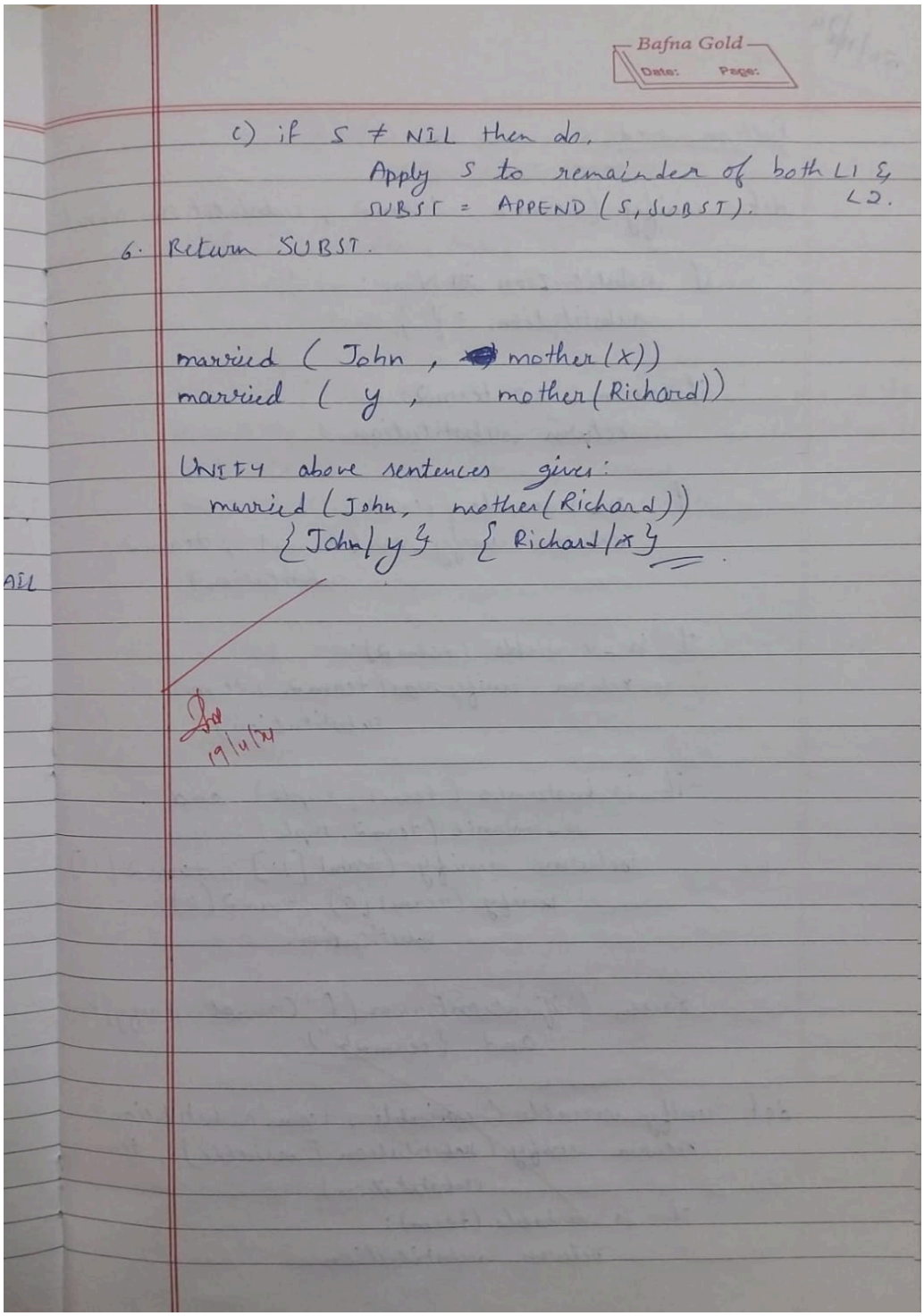
1. if 'a' or 'b' is a variable or constant, then:
   a) if 'a' or 'b' are identical, then return NIL.
   b) Else if 'a' is a variable,
      then if 'a' occurs in 'b', then return FAIL
      else return {('b'/'a')}.
   c) else if 'b' is a variable,
      if 'b' occurs in 'a' then return FAIL,
      else return {('a'/'b')}
   d) else return FAIL

2. If the initial Predicate symbol in 'a' and 'b' are not same, then return FAIL

3. If 'a' and 'b' have a different number of arguments, then return FAIL

4. Set Substitution set (SUBST) to NIL

5. For i=1 to no. of elements in 'a'
   a) Call Unify function with ith element of 'a' and ith element of 'b', and puth the result into S.
   b) if S = failure return FAIL




**Code:**

```
class Term:
    def __init__(self, symbol, args=None):
        self.symbol = symbol
        self.args = args if args else []
```

```
    def __str__(self):
        if not self.args:
            return str(self.symbol)
        return f"{self.symbol}({','.join(str(arg) for arg in self.args)})"

    def is_variable(self):
        return isinstance(self.symbol, str) and self.symbol.isupper() and not self.args

def occurs_check(var, term, substitution):
    if term.is_variable():
        if term.symbol in substitution:
            return occurs_check(var, substitution[term.symbol], substitution)
        return var.symbol == term.symbol
    return any(occurs_check(var, arg, substitution) for arg in term.args)

def substitute(term, substitution):
    if term.is_variable() and term.symbol in substitution:
        return substitute(substitution[term.symbol], substitution)
    if not term.args:
        return term
    return Term(term.symbol, [substitute(arg, substitution) for arg in term.args])

def unify(term1, term2, substitution=None, iteration=1):
    if substitution is None:
        substitution = {}
    print(f"\nIteration {iteration}:")
    print(f"Attempting to unify: {term1} and {term2}")
    print(f"Current substitution: {', '.join(f'{k}->{v}' for k,v in substitution.items()) or 'empty'}")
    term1 = substitute(term1, substitution)
    term2 = substitute(term2, substitution)
    if term1.symbol == term2.symbol and not term1.args and not term2.args:
        print("Terms are identical - no substitution needed")
        return substitution
    if term1.is_variable():
        if occurs_check(term1, term2, substitution):
            print(f"Occurs check failed: {term1.symbol} occurs in {term2}")
            return None
        substitution[term1.symbol] = term2
        print(f"Added substitution: {term1.symbol} -> {term2}")
        return substitution

    if term2.is_variable():
        if occurs_check(term2, term1, substitution):
            print(f"Occurs check failed: {term2.symbol} occurs in {term1}")
            return None
        substitution[term2.symbol] = term1
        print(f"Added substitution: {term2.symbol} -> {term1}")
        return substitution
```

```
    if term1.symbol != term2.symbol or len(term1.args) != len(term2.args):
        print(f"Unification failed: Different predicates or argument lengths")
        return None

    for arg1, arg2 in zip(term1.args, term2.args):
        result = unify(arg1, arg2, substitution, iteration + 1)
        if result is None:
            return None
        substitution = result
    return substitution

def parse_term(s):
    """Parse terms like P(X,f(Y)) or X"""
    s = s.strip()
    if '(' not in s:
        return Term(s)
    pred = s[:s.index('(')]
    args_str = s[s.index('(')+1:s.rindex(')')]
    args = []
    current = ''
    depth = 0
    for c in args_str:
        if c == '(' or c == '[':
            depth += 1
        elif c == ')' or c == ']':
            depth -= 1
        elif c == ',' and depth == 0:
            args.append(parse_term(current.strip()))
            current = ''
            continue
        current += c
    if current:
        args.append(parse_term(current.strip()))
    return Term(pred, args)

def print_examples():
    print("\nExample format:")
    print("1. Simple terms: P(X,Y)")
    print("2. Nested terms: P(f(X),g(Y))")
    print("3. Mixed terms: Knows(John,X)")
    print("4. Complex nested terms: P(f(g(X)),h(Y,Z))")
    print("\nNote: Use capital letters for variables (X,Y,Z) and lowercase for constants and predicates.")

def validate_input(expr):
    """Basic validation for input expressions"""
    if not expr:
        return False
```

```
    count = 0
    for char in expr:
        if char == '(':
            count += 1
        elif char == ')':
            count -= 1
        if count < 0:
            return False
    return count == 0

def main():
    while True:
        print("\n=== First Order Predicate Logic Unification ===")
        print("1. Start Unification")
        print("2. Show Examples")
        print("3. Exit")
        choice = input("\nEnter your choice (1-3): ")
        if choice == '1':
            print("\nEnter two expressions to unify.")
            print_examples()
            while True:
                expr1 = input("\nEnter first expression (or 'back' to return): ")
                if expr1.lower() == 'back':
                    break

                if not validate_input(expr1):
                    print("Invalid expression! Please check the format and try again.")
                    continue
                expr2 = input("Enter second expression: ")
                if not validate_input(expr2):
                    print("Invalid expression! Please check the format and try again.")
                    continue

                try:
                    term1 = parse_term(expr1)
                    term2 = parse_term(expr2)
                    print("\nUnification Process:")
                    result = unify(term1, term2)
                    print("\nFinal Result:")
                    if result is None:
                        print("Unification failed!")
                    else:
                        print("Unification successful!")
                        print("Final substitutions:", ', '.join(f'{k}->{v}' for k,v in result.items()))

                    retry = input("\nTry another unification? (y/n): ")
                    if retry.lower() != 'y':
                        break
```

```
            except Exception as e:
                print(f"Error processing expressions: {str(e)}")
                print("Please check your input format and try again.")

        elif choice == '2':
            print("\n=== Example Expressions ===")
            print("1. P(X,h(Y)) and P(a,f(Z))")
            print("2. P(f(a),g(Y)) and P(X,X)")
            print("3. Knows(John,X) and Knows(X,Elisabeth)")
            print("\nPress Enter to continue...")
            input()

        elif choice == '3':
            print("\nThank you for using the Unification Program!")
            break

        else:
            print("\nInvalid choice! Please enter 1, 2, or 3.")

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

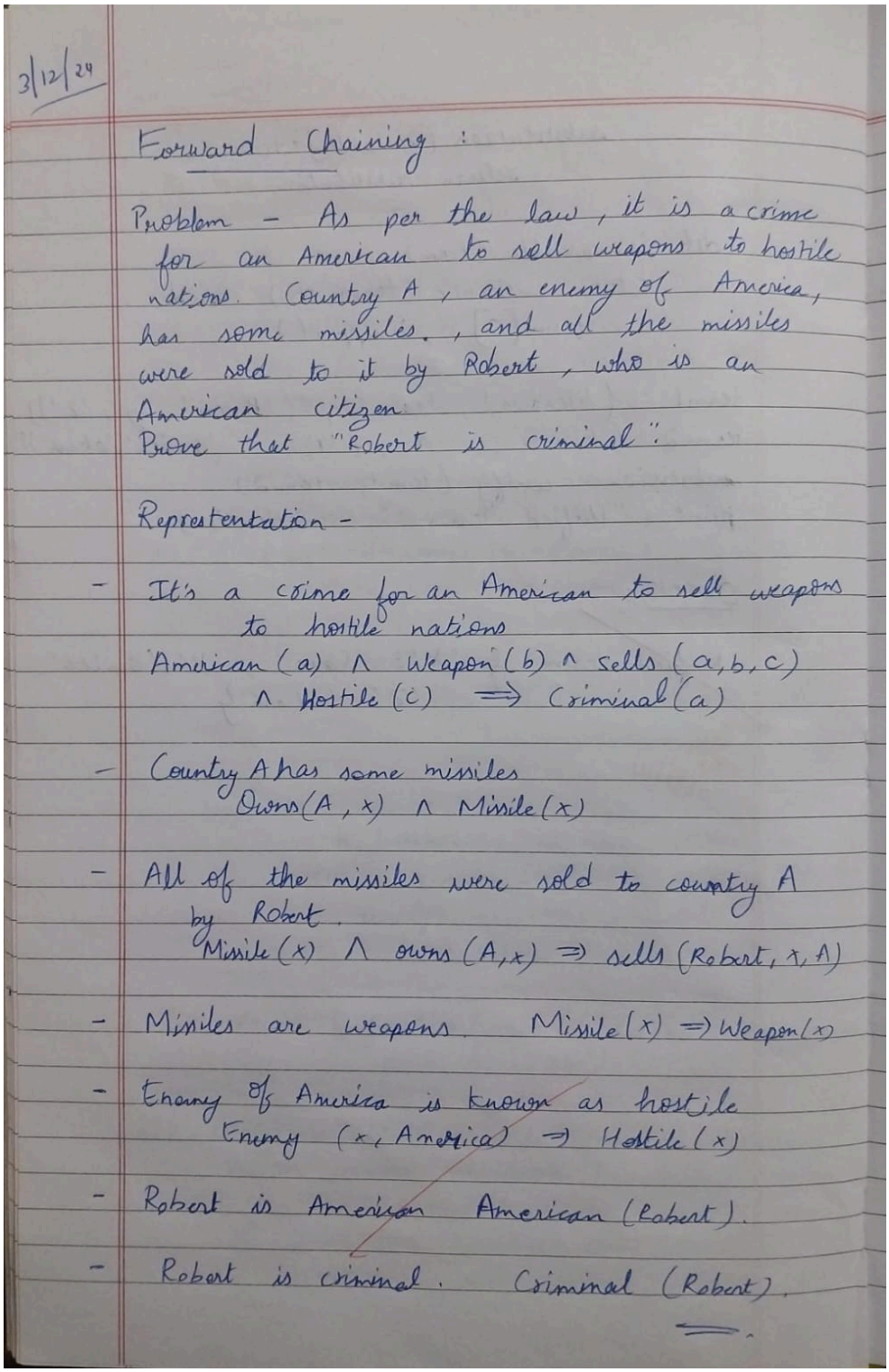
**Output:**

```
Enter the first expression (e.g., p(x, f(y))): p(b,x,f(g(z)))
Enter the second expression (e.g., p(a, f(z))): p(z,f(y),f(y))
Expression 1: ['p', '(b', 'x', ['f', '(g(z)))']]
Expression 2: ['p', '(z', ['f', '(y)'], ['f', '(y))']]
Result: Unification Successful
Substitutions: {'(b': '(z', 'x': ['f', '(y)'], '(g(z)))': '(y))'}
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(x,h(y))
Enter the second expression (e.g., p(a, f(z))): p(a,f(z))
Expression 1: ['p', '(x', ['h', '(y))']]
Expression 2: ['p', '(a', ['f', '(z))']]
Result: Unification Failed
Do you want to test another pair of expressions? (yes/no): yes
Enter the first expression (e.g., p(x, f(y))): p(f(a),g(y))
Enter the second expression (e.g., p(a, f(z))): p(x,x)
Expression 1: ['p', '(f(a)', ['g', '(y))']]
Expression 2: ['p', '(x', 'x)']
Result: Unification Successful
Substitutions: {'(f(a)': '(x', 'x)': ['g', '(y))']}
Do you want to test another pair of expressions? (yes/no): no
```

**Program 8**

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

**Algorithm:**



3/12/24

Forward Chaining :

Problem - As per the law, it is a crime for an American to sell weapons to hostile nations. Country A, an enemy of America, has some missiles., and all the missiles were sold to it by Robert, who is an American citizen
Prove that "Robert is criminal".

Representation -

- It's a crime for an American to sell weapons to hostile nations
  American(a) ∧ Weapon(b) ∧ sells(a,b,c) ∧ Hostile(c) ⟹ Criminal(a)

- Country A has some missiles
  Owns(A, x) ∧ Missile(x)

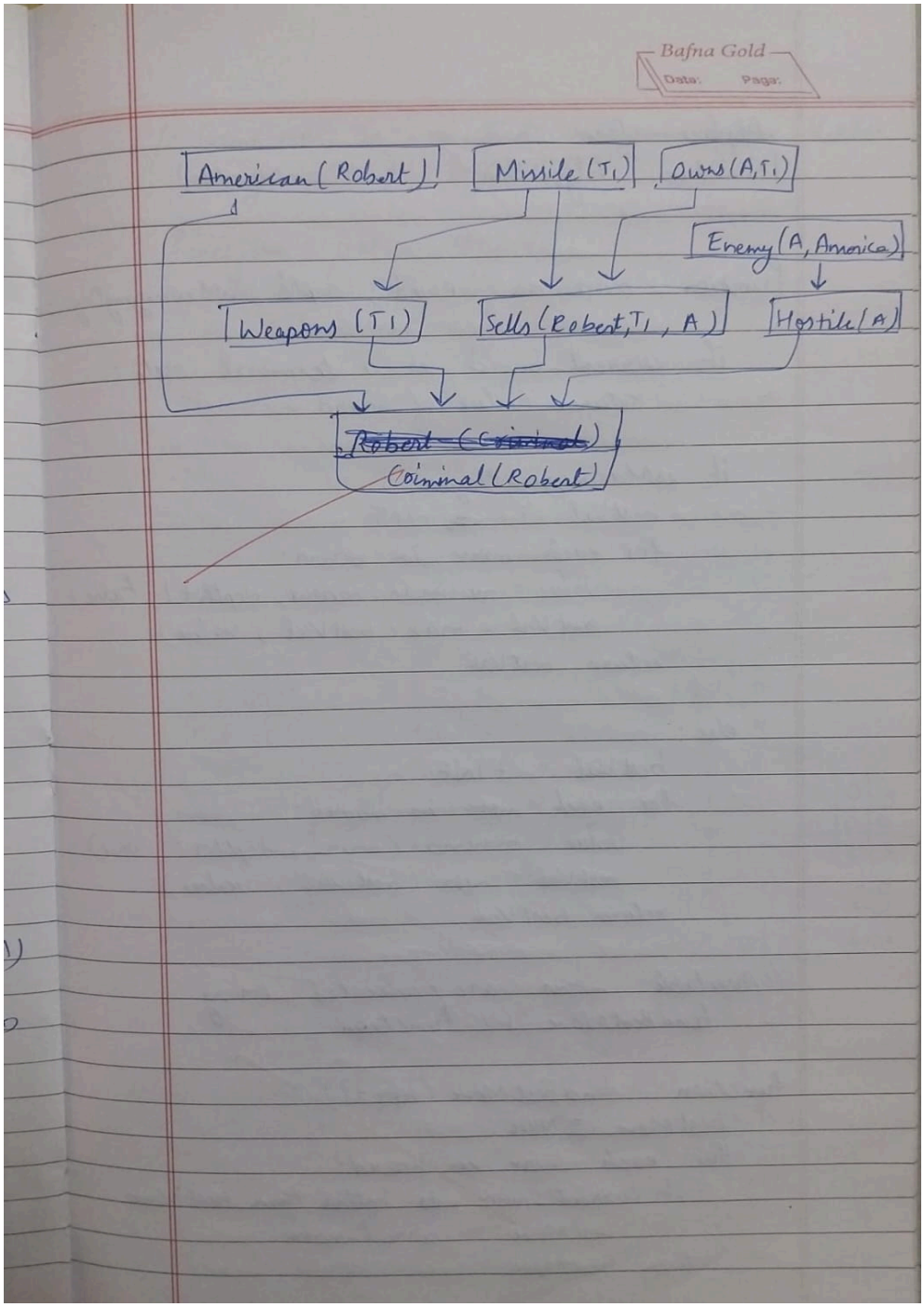
- All of the missiles were sold to country A by Robert.
  Missile(x) ∧ owns(A,x) ⇒ sells(Robert, x, A)

- Missiles are weapons. Missile(x) ⇒ Weapon(x)

- Enemy of America is known as hostile
  Enemy(x, America) ⇒ Hostile(x)

- Robert is American. American(Robert).

- Robert is criminal. Criminal(Robert).




**Code:**

```
# Define facts and rules as dictionaries and lists
facts = {
    "American(Robert)": True,
    "Missile(T1)": True,
    "Owns(A, T1)": True,
    "Enemy(A, America)": True,
```

```
}
rules = [
    # Rule 1: All missiles are weapons
    {"if": ["Missile(x)"], "then": ["Weapon(x)"]},

    # Rule 2: A criminal is made if someone sells weapons to a hostile country
    {"if": ["American(p)", "Weapon(q)", "Sells(p, q, r)", "Hostile(r)"], "then": ["Criminal(p)"]},

    # Rule 3: Robert sells all missiles to country A
    {"if": ["Missile(x)", "Owns(A, x)"], "then": ["Sells(Robert, x, A)"]},

    # Rule 4: If someone is an enemy of America, they are hostile
    {"if": ["Enemy(x, America)"], "then": ["Hostile(x)"]},
]

# Query to be proved
query = "Criminal(Robert)"

# Helper function to unify facts with conditions
def unify_condition_with_fact(condition, fact):
    predicate_condition, args_condition = condition.split("(")
    predicate_fact, args_fact = fact.split("(")
    if predicate_condition != predicate_fact:
        return None  # Predicates must match
    args_condition = args_condition[:-1].split(",")
    args_fact = args_fact[:-1].split(",")
    if len(args_condition) != len(args_fact):
        return None
    substitutions = {}
    for var, constant in zip(args_condition, args_fact):
        if var.islower():  # If it's a variable, substitute it
            substitutions[var] = constant
        elif var != constant:  # If constants don't match, return None
            return None
    return substitutions

def substitute_statement(statement, substitutions):
    predicate, args = statement.split("(")
    args = args[:-1].split(",")
    new_args = [substitutions.get(arg, arg) for arg in args]
    return f"{predicate}({', '.join(new_args)})"

# Forward chaining algorithm to derive facts
def perform_forward_chaining(facts, rules, query):
    inferred_facts = set(facts.keys())  # Start with the given facts
    while True:
        new_inferences = set()
        for rule in rules:
```

```
        # Process the "if" part of the rule
        conditions = rule["if"]
        possible_substitutions = [{}]
        for condition in conditions:
            temp_substitutions = []
            for fact in inferred_facts:
                subs = unify_condition_with_fact(condition, fact)
                if subs is not None:
                    # Combine existing substitutions with new ones
                    for existing_subs in possible_substitutions:
                        combined_subs = existing_subs.copy()
                        combined_subs.update(subs)
                        temp_substitutions.append(combined_subs)
            if not temp_substitutions:
                break
            possible_substitutions = temp_substitutions
        # If conditions are satisfied, infer the "then" part
        if possible_substitutions:
            for subs in possible_substitutions:
                for conclusion in rule["then"]:
                    new_fact = substitute_statement(conclusion, subs)
                    new_inferences.add(new_fact)
    # Add the newly inferred facts to the set of facts
    if query in new_inferences:
        return True  # The query has been proven
    # If no new facts were inferred, stop the process
    if not new_inferences - inferred_facts:
        return False  # The query cannot be proven
    inferred_facts.update(new_inferences)
result = perform_forward_chaining(facts, rules, query)
if result:
    print(f"It is proven that'{query}' is true")
else:
    print(f"It is proven that '{query}' is false")
```

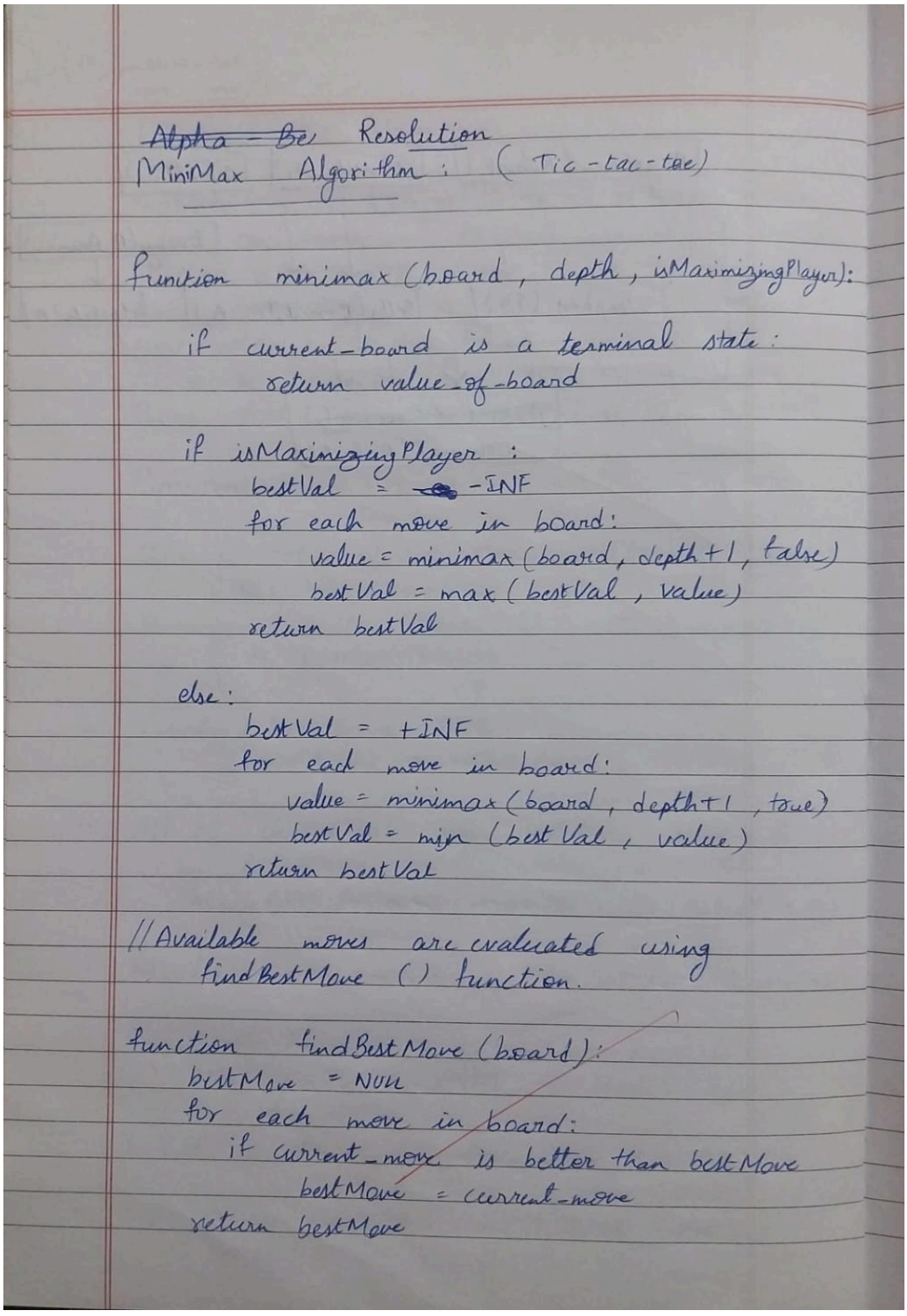
Output:

```
It is proven that'Criminal(Robert)' is true
```

**Program 9**

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

**Algorithm:**



Alpha - Beta Resolution

MiniMax Algorithm : ( Tic-tac-toe)

```
function minimax(board, depth, isMaximizingPlayer):

    if current_board is a terminal state:
        return value_of_board

    if isMaximizingPlayer :
        bestVal = -INF
        for each move in board:
            value = minimax(board, depth+1, false)
            bestVal = max(bestVal, value)
        return bestVal

    else:
        bestVal = +INF
        for each move in board:
            value = minimax(board, depth+1, true)
            bestVal = min(bestVal, value)
        return bestVal

//Available moves are evaluated using
    findBestMove() function.

function findBestMove(board):
    bestMove = NULL
    for each move in board:
        if current_move is better than bestMove
            bestMove = current_move
    return bestMove
```

**Code:**

```
# Knowledge Base (KB)
facts = {
    "Eats(Anil, Peanuts)": True,
    "not Killed(Anil)": True,
    "Food(Apple)": True,
    "Food(Vegetables)": True,
}

rules = [
    # Rule: Food(X) :- Eats(Y, X) and not Killed(Y)
    {"conditions": ["Eats(Y, X)", "not Killed(Y)"], "conclusion": "Food(X)"},
    # Rule: Likes(John, X) :- Food(X)
    {"conditions": ["Food(X)"], "conclusion": "Likes(John, X)"},
]

# Query
query = "Likes(John, Peanuts)"

# Helper function to substitute variables in a rule
def substitute(rule_part, substitutions):
    for var, value in substitutions.items():
        rule_part = rule_part.replace(var, value)
    return rule_part

# Function to resolve the query
def resolve_query(facts, rules, query):
    working_facts = facts.copy()
    while True:
        new_facts_added = False
        for rule in rules:
            conditions = rule["conditions"]
            conclusion = rule["conclusion"]

            # Try all substitutions for variables (X, Y) in the rules
            for entity in ["Apple", "Vegetables", "Peanuts", "Anil", "John"]:
                substitutions = {"X": "Peanuts", "Y": "Anil"}  # Fixed for this problem
                resolved_conditions = [substitute(cond, substitutions) for cond in conditions]
                resolved_conclusion = substitute(conclusion, substitutions)

                # Check if all conditions are true
                if all(working_facts.get(cond, False) for cond in resolved_conditions):
                    if resolved_conclusion not in working_facts:
                        working_facts[resolved_conclusion] = True
                        new_facts_added = True
                        print(f"Derived Fact: {resolved_conclusion}")

        if not new_facts_added:
```

```
            break

    # Check if the query is resolved
    return working_facts.get(query, False)

# Run the resolution process
if resolve_query(facts, rules, query):
    print(f"Proven: {query}")
else:
    print(f"Not Proven: {query}")
```

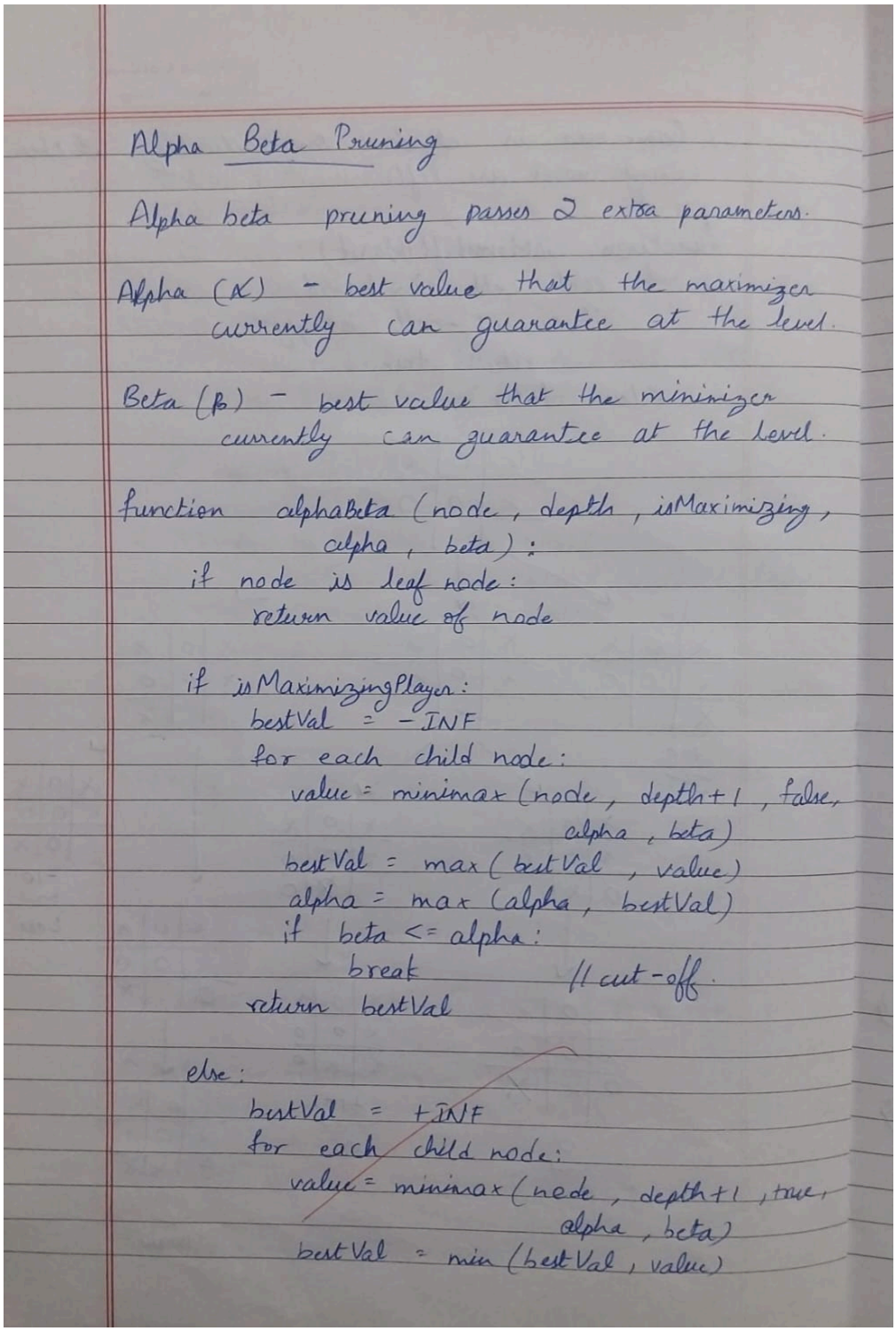
**Output:**

```
Derived Fact: Food(Peanuts)
Derived Fact: Likes(John, Peanuts)
Proven: Likes(John, Peanuts)
```

**Program 10**

**Implement Alpha-Beta Pruning.**

**Algorithm:**



Alpha Beta Pruning

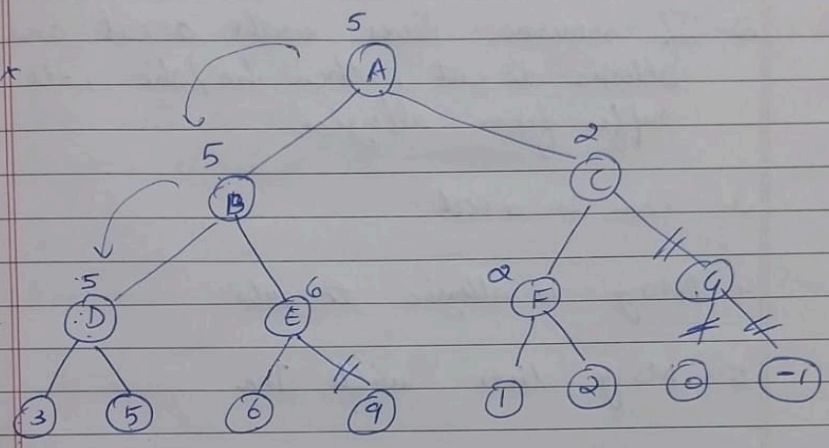
Alpha beta pruning passes 2 extra parameters.

Alpha (α) - best value that the maximizer currently can guarantee at the level.

Beta (β) - best value that the minimizer currently can guarantee at the level.

```
function alphaBeta (node, depth, isMaximizing,
                    alpha, beta):
    if node is leaf node:
        return value of node

    if isMaximizingPlayer:
        bestVal = -INF
        for each child node:
            value = minimax (node, depth+1, false,
                             alpha, beta)
            bestVal = max (bestVal, value)
            alpha = max (alpha, bestVal)
            if beta <= alpha:
                break                    // cut-off.
        return bestVal

    else:
        bestVal = +INF
        for each child node:
            value = minimax (node, depth+1, true,
                             alpha, beta)
            bestVal = min (bestVal, value)
```

alpha = max (alpha, ~~beta~~ bestVal)
if beta <= alpha
break
return bestVal




Optimal value is 5.

$\alpha = -\infty, \beta = +\infty$

$\alpha = 3, \beta = +\infty$

$\alpha = \max(3,5) = 5$

~~At E~~

3/12/W

**Code:**

```
import math

def minimax(depth, index, maximizing_player, values, alpha, beta):
    # Base case: when we've reached the leaf nodes
    if depth == 0:
        return values[index]

    if maximizing_player:
        max_eval = float('-inf')
        for i in range(2):  # 2 children per node
            eval = minimax(depth - 1, index * 2 + i, False, values, alpha, beta)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:  # Beta cutoff
                break
        return max_eval
    else:
        min_eval = float('inf')
        for i in range(2):  # 2 children per node
            eval = minimax(depth - 1, index * 2 + i, True, values, alpha, beta)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:  # Alpha cutoff
                break
        return min_eval


# Accept values from the user
leaf_values = list(map(int, input("Enter the leaf node values separated by spaces: ").split()))

# Check if the number of values is a power of 2
if math.log2(len(leaf_values)) % 1 != 0:
    print("Error: The number of leaf nodes must be a power of 2 (e.g., 2, 4, 8, 16).")
else:
    # Calculate depth of the tree
    tree_depth = int(math.log2(len(leaf_values)))
    optimal_value = minimax(depth=tree_depth, index=0, maximizing_player=True, values=leaf_values,
alpha=float('-inf'), beta=float('inf'))
    print("Optimal value calculated using Minimax:", optimal_value)
```

**Output:**

```
Enter the leaf node values separated by spaces: 10 9 14 18 5 4 50 3
Optimal value calculated using Minimax: 10
```