VISVESVARAYA TECHNOLOGICAL UNIVERSITY

"JnanaSangama", Belgaum -590014, Karnataka.

LAB REPORT
on

Artificial Intelligence (23CS5PCAIN)

Submitted by

Pradeep P T (1BM22CS197)

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

<u>CERTIFICATE</u>

This is to certify that the Lab work entitled “Artificial Intelligence (23CS5PCAIN)” carried out by **Pradeep P T (1BM22CS197),** who is bonafide student of **B.M.S. College of Engineering.** It is in partial fulfillment for the award of **Bachelor of Engineering in Computer Science and Engineering** of the Visvesvaraya Technological University, Belgaum. The Lab report has been approved as it satisfies the academic requirements in respect of an Artificial Intelligence(23CS5PCAIN) work prescribed for the said degree.

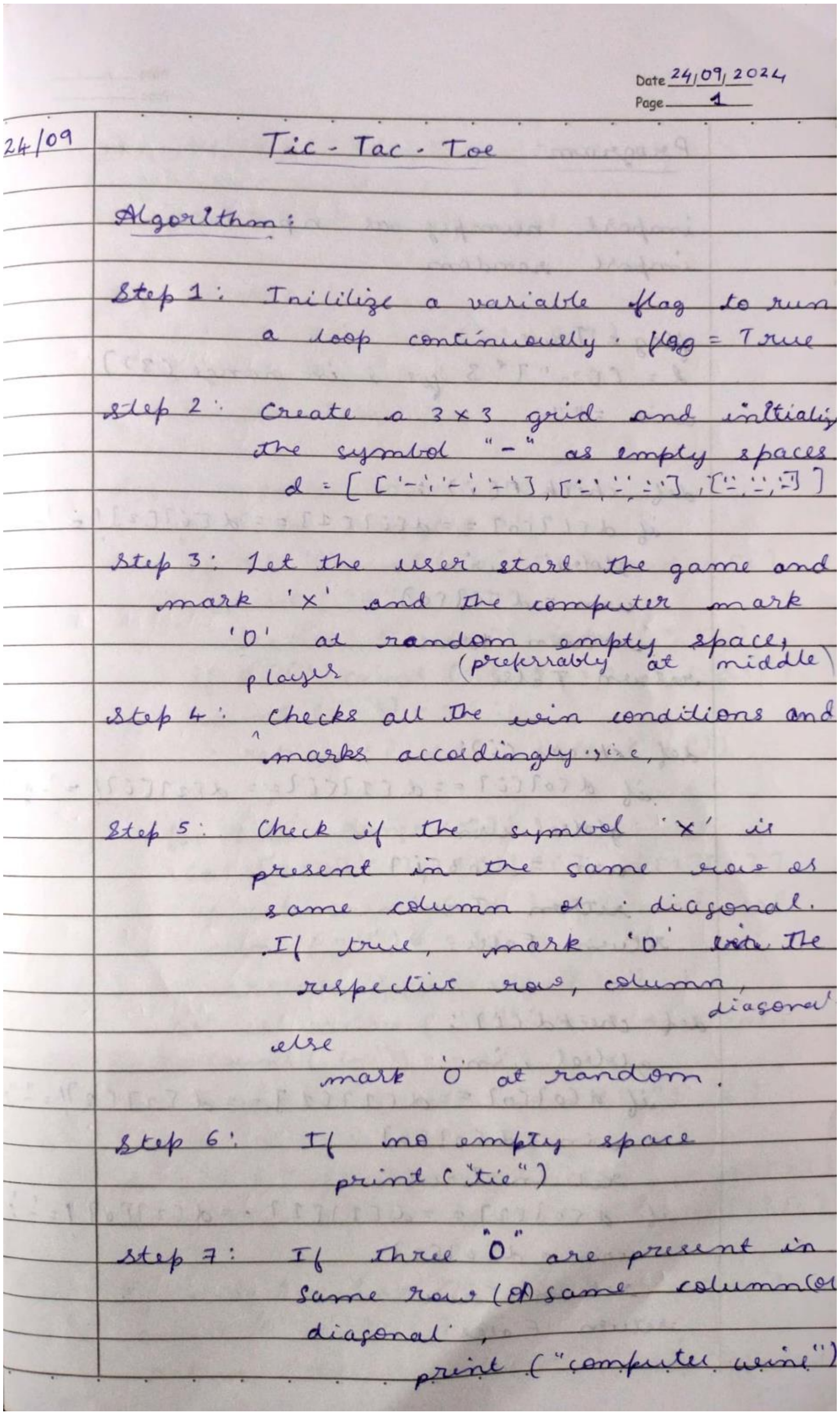
Prof. Sneha S Bagalkot	Dr. Kavitha Sooda
Assistant Professor	Professor & HOD
Department of CSE, BMSCE	Department of CSE, BMSCE

Index

Github Link:
AI Lab Github Link

Implement Tic-Tac-Toe Game.

Algorithm:



Date 24/09/2024
Page 1

24/09

Tic - Tac - Toe

Algorithm:

Step 1: Inililize a variable flag to run
a loop continuously. flag = True

Step 2: Create a 3×3 grid and initializ
the symbol "–" as empty spaces
d = [['-','-','-'],['-','-','-'],['-','-','-']]

Step 3: Let the user start the game and
mark 'x' and the computer mark
'O' at random empty space,
player (preferrably at middle)

Step 4: checks all the win conditions and
marks accordingly, i.e,

Step 5: Check if the symbol 'x' is
present in the same row or
same column or diagonal.
If true, mark 'O' into the
respective row, column,
diagonal
else
mark 'O' at random.

Step 6: If no empty space
print ("tie")

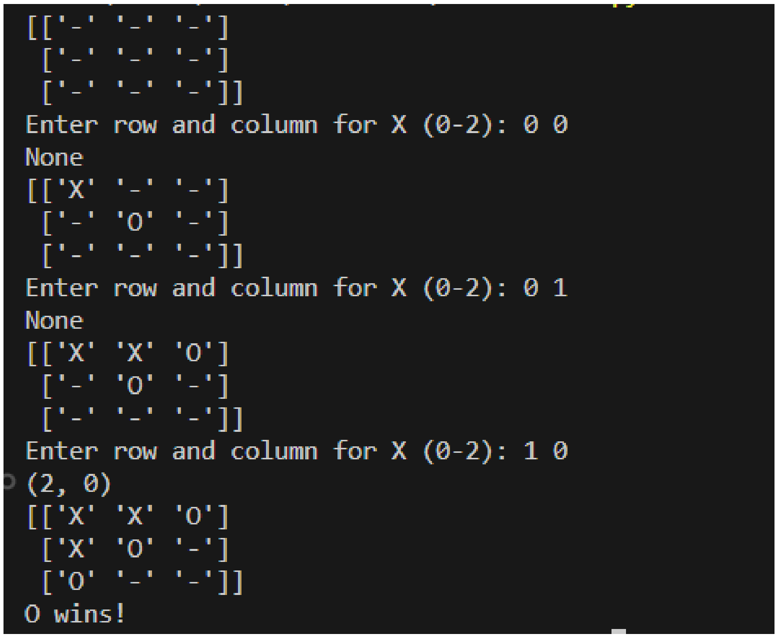
Step 7: If three "O" are present in
same row (or) same column(or
diagonal,
print ("computer wins")

Code:

```
import random
import numpy as np

board = [["-"] * 3 for _ in range(3)]

def check_win():
    for i in range(3):
        if board[i][0] == board[i][1] == board[i][2] != "-":
            return True
        if board[0][i] == board[1][i] == board[2][i] != "-":
            return True

    if board[0][0] == board[1][1] == board[2][2] != "-":
        return True
    if board[0][2] == board[1][1] == board[2][0] != "-":
        return True

    return False

def full():
    return all(cell != "-" for row in board for cell in row)

def can_win(m):
    for i in range(3):
        row = board[i]
        if row.count(m) == 2 and row.count("-") == 1:
            return (i, row.index("-"))

    for i in range(3):
        col = [board[j][i] for j in range(3)]
        if col.count(m) == 2 and col.count("-") == 1:
            return (col.index("-"), i)

    diag1 = [board[i][i] for i in range(3)]
    if diag1.count(m) == 2 and diag1.count("-") == 1:
        return (diag1.index("-"), diag1.index("-"))

    diag2 = [board[i][2 - i] for i in range(3)]
    if diag2.count(m) == 2 and diag2.count("-") == 1:
        return (diag2.index("-"), 2 - diag2.index("-"))

    return None

def display():
    print(np.array(board))

while True:
    display()
    u = tuple(map(int, input("Enter row and column for X (0-2): ").strip().split()))
    if board[u[0]][u[1]] != "-":
        print("Invalid move, try again.")
        continue
```

```
board[u[0]][u[1]] = "X"

if check_win():
    display()
    print("X wins!")
    break

if full():
    display()
    print("It's a tie!")
    break

move = can_win("O")
print(move)
if move is None:
    move = can_win("X")
    if move is None:
        empty = [(i, j) for i in range(3) for j in range(3) if board[i][j] == "-"]
        move = random.choice(empty)
if board[1][1]=="-":
    move=(1 ,1)
board[move[0]][move[1]] = "O"

if check_win():
    display()
    print("O wins!")
    break
```

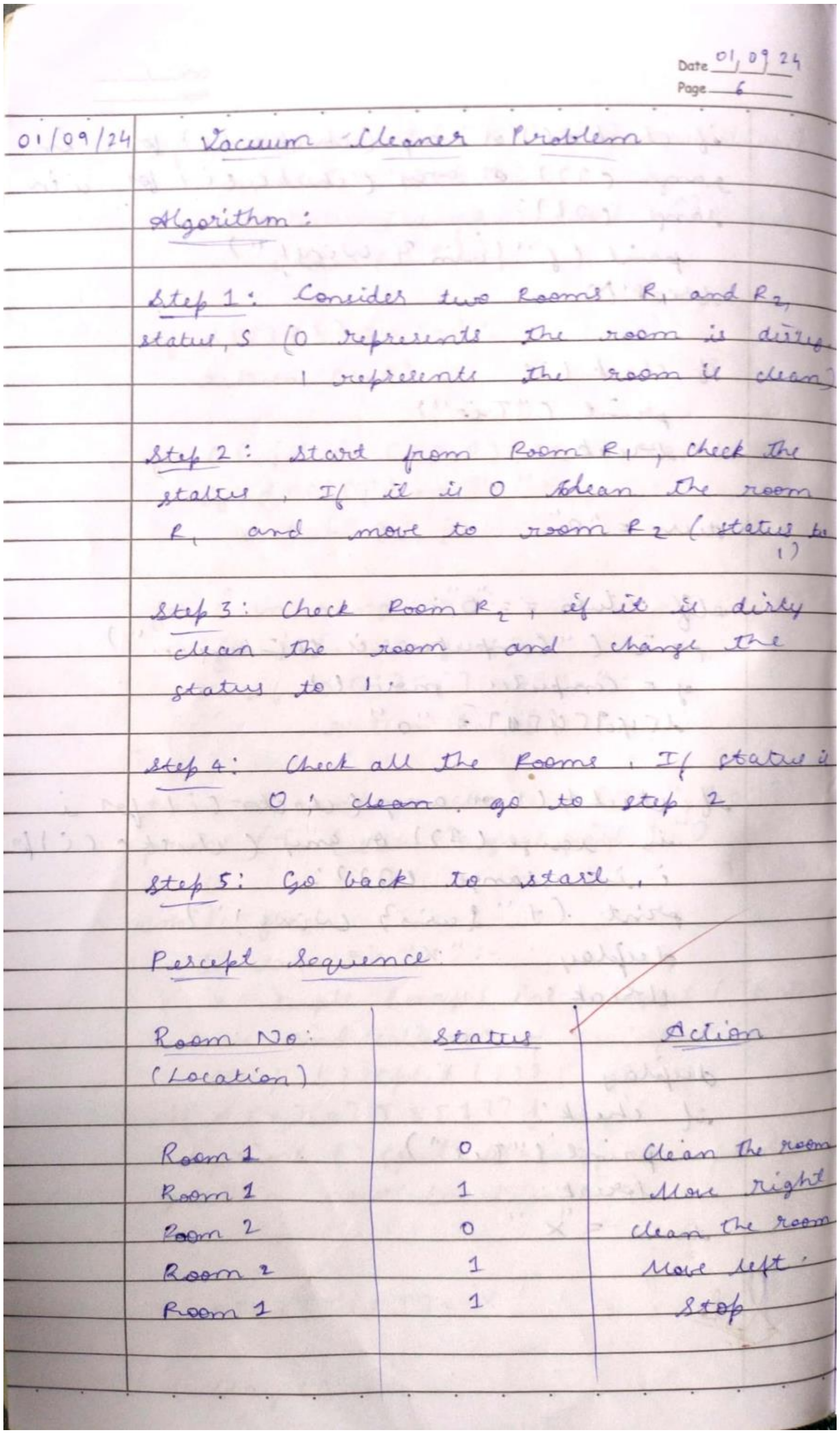
Output:



```
[['-' '-' '-']
 ['-' '-' '-']
 ['-' '-' '-']]
Enter row and column for X (0-2): 0 0
None
[['X' '-' '-']
 ['-' 'O' '-']
 ['-' '-' '-']]
Enter row and column for X (0-2): 0 1
None
[['X' 'X' 'O']
 ['-' 'O' '-']
 ['-' '-' '-']]
Enter row and column for X (0-2): 1 0
(2, 0)
[['X' 'X' 'O']
 ['X' 'O' '-']
 ['O' '-' '-']]
O wins!
```

Implement Vacuum Cleaner Agent.

Algorithm:



Date 01, 09 24
Page 6

01/09/24

Vacuum Cleaner Problem

Algorithm :

Step 1 : Consider two Rooms R_1 and R_2, status, S (0 represents the room is dirty. 1 represents the room is clean)

Step 2 : Start from Room R_1, check the status, If it is 0 clean the room R_1 and move to room R_2 (status to 1)

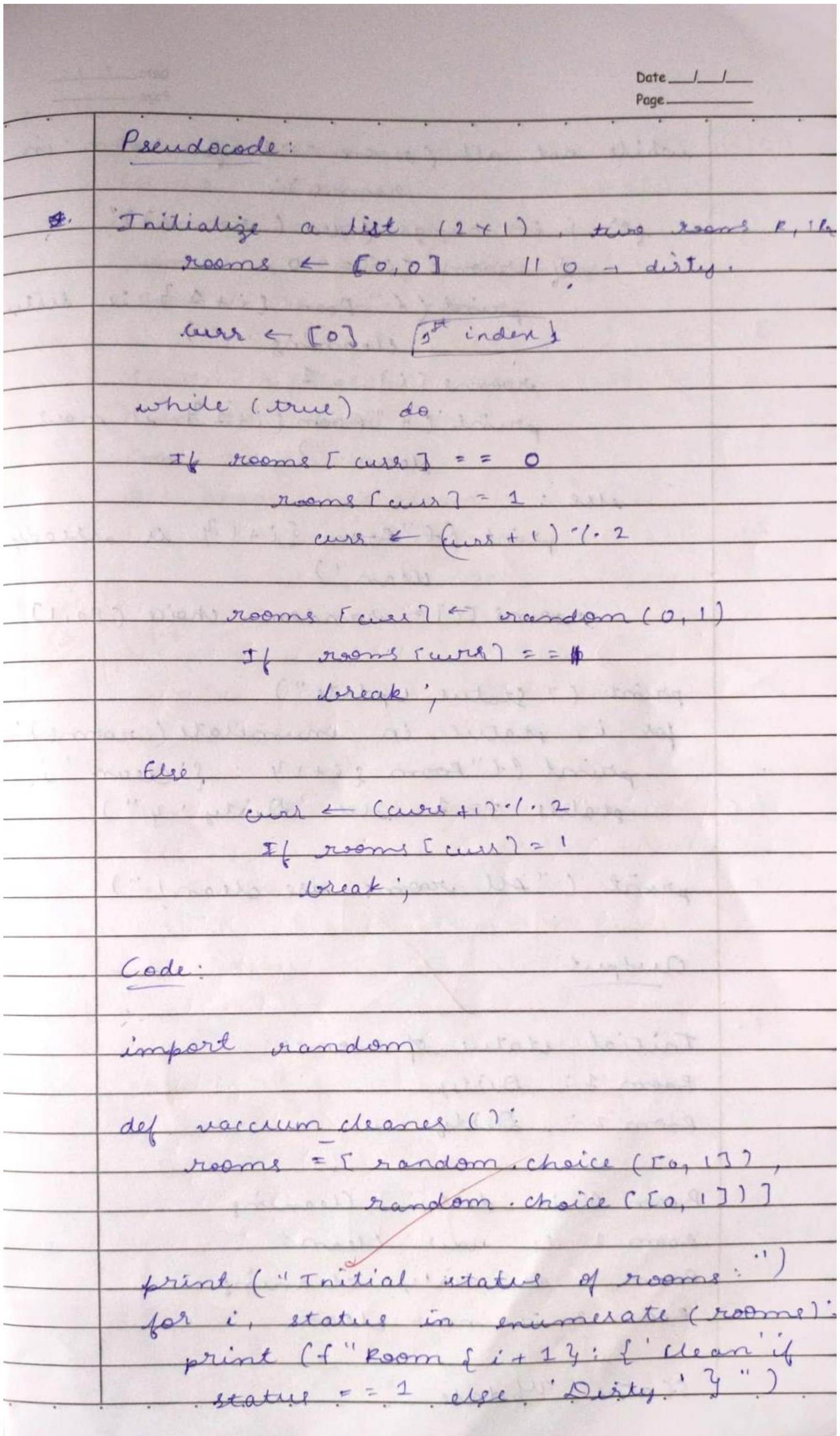
Step 3 : Check Room R_2, if it is dirty clean the room and change the status to 1 :

Step 4 : Check all the Rooms, If status is 0 ; clean . go to step 2

Step 5 : Go back to start .

Percept Sequence :

Room No : (Location)	Status	Action
Room 1	0	Clean the room
Room 1	1	Move right
Room 2	0	clean the room
Room 2	1	Move left .
Room 1	1	Stop



Date __/__/__

Page ____

Pseudocode:

```
Initialize a list (2×1), two rooms R, 1R
    rooms ← [0,0]      // 0 → dirty.

    curr ← [0]   [1st index]

while (true) do

    If rooms [curr] == 0
        rooms [curr] = 1 ;
            curr ← (curr + 1) % 2

        rooms [curr] ← random (0, 1)
        If rooms [curr] == 1
            break ;

    Else,
        curr ← (curr + 1) % 2
        If rooms [curr] = 1
            break ;
```

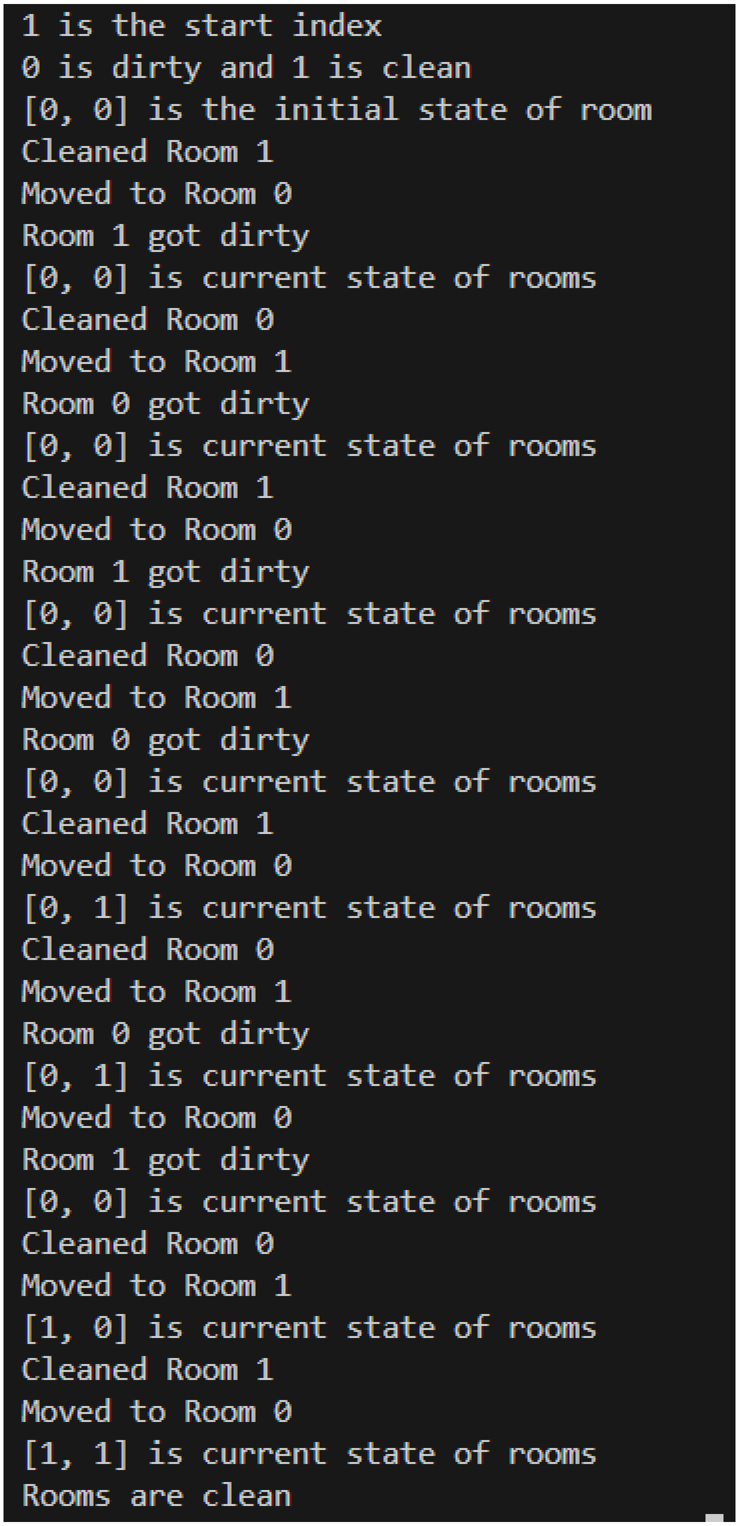
Code:

```
import random

def vaccum_cleaner ():
    rooms = [ random.choice ([0, 1]) ,
              random.choice ([0, 1]) ]

print ("Initial status of rooms:")
for i, status in enumerate (rooms):
    print (f"Room {i+1}: {'clean' if
    status == 1 else 'Dirty'}")
```

Code:

```
import random
l=[random.choice([0,1]),random.choice([0,1])]
def check(i):
    if l[i]==0:
        l[i]=1
        print(f"Cleaned Room {i}")
    print(f"Moved to Room {(i+1)%2}")
    return (i+1)%2
i=random.choice([0,1])

print(f"{i} is the start index")
print("0 is dirty and 1 is clean")
print(f"{l} is the initial state of room")

while sum(l)!=2:
    i=check(i)
    if l[(i+1)%2]==1:
        l[(i+1)%2]=random.choice([0,1])
        if l[(i+1)%2]==0:
            print(f"Room {(i+1)%2} got dirty")
    print(f"{l} is current state of rooms")
print("Rooms are clean")
```

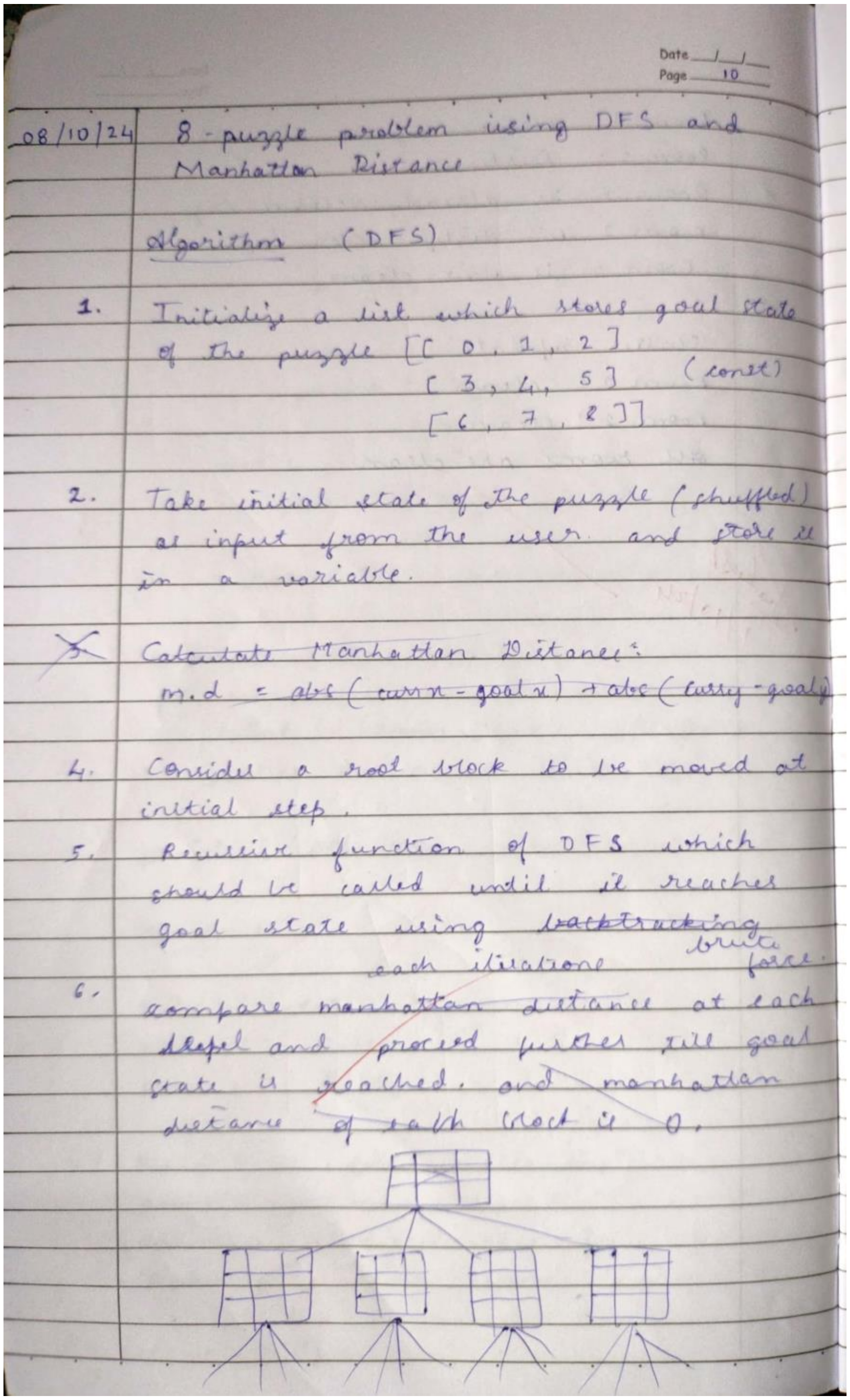
Output:



```
1 is the start index
0 is dirty and 1 is clean
[0, 0] is the initial state of room
Cleaned Room 1
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
[0, 1] is current state of rooms
Cleaned Room 0
Moved to Room 1
Room 0 got dirty
[0, 1] is current state of rooms
Moved to Room 0
Room 1 got dirty
[0, 0] is current state of rooms
Cleaned Room 0
Moved to Room 1
[1, 0] is current state of rooms
Cleaned Room 1
Moved to Room 0
[1, 1] is current state of rooms
Rooms are clean
```

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

Algorithm:



Date / /
Page 10

08/10/24 8-puzzle problem using DFS and Manhattan Distance

Algorithm (DFS)

1. Initialize a list which stores goal state of the puzzle [[0, 1, 2]
[3, 4, 5] (const)
[6, 7, 8]]

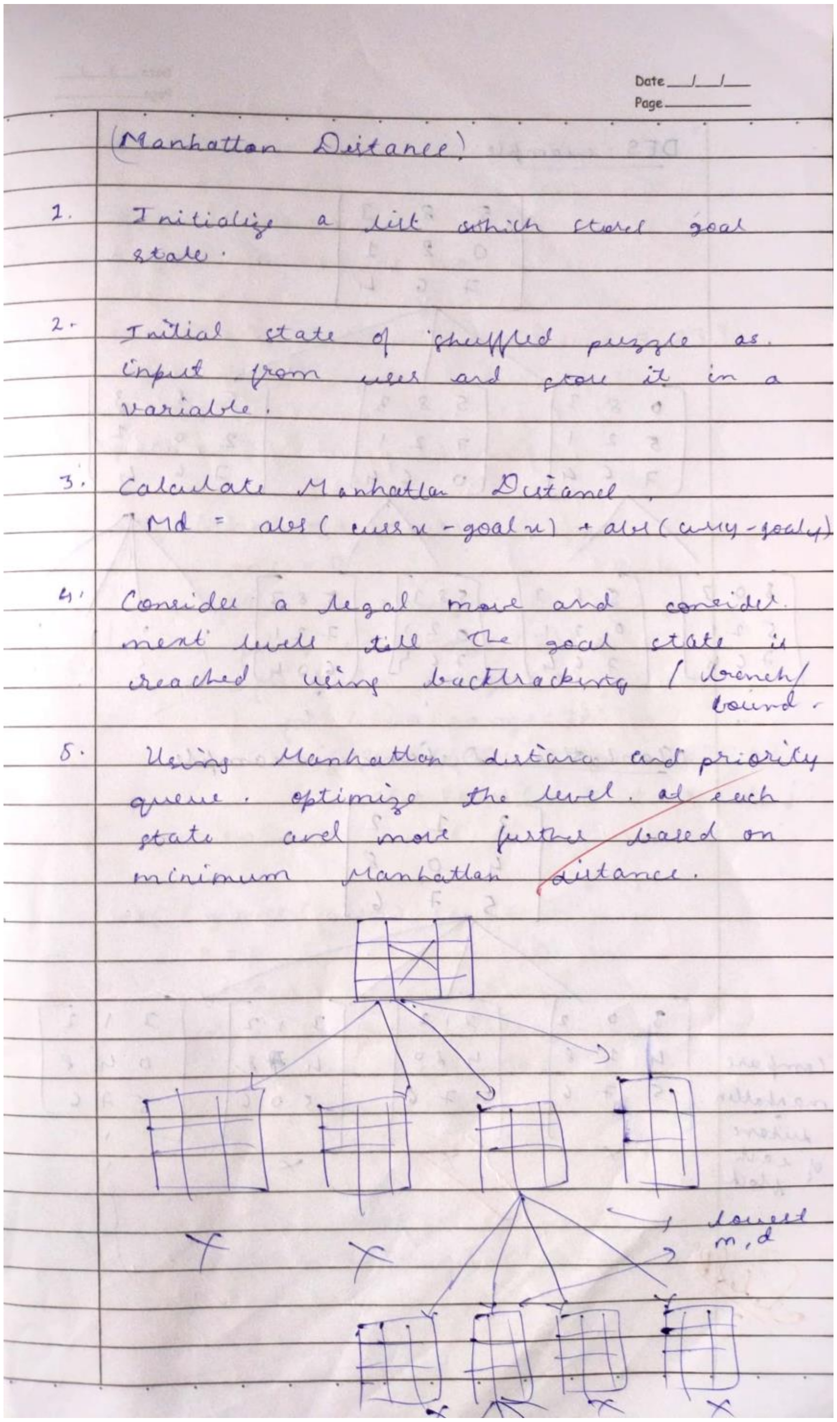
2. Take initial state of the puzzle (shuffled) as input from the user. and store it in a variable.

~~3. Calculate Manhattan Distance:~~
~~m.d = abs(currx - goalx) + abs(curry - goaly)~~

4. Consider a root block to be moved at initial step.

5. Recursive function of DFS which should be called until it reaches goal state using ~~backtracking~~ brute force.

6. compare manhattan distance at each each iterations steps and proceed further till goal state is reached. and manhattan distance of each block is 0.



Date __/__/__
Page ______

(Manhattan Distance)

1. Initialize a list which stores goal state.

2. Initial state of shuffled puzzle as input from user and store it in a variable.

3. Calculate Manhattan Distance.
   Md = abs (currx - goalx) + abs (curry - goaly)

4. Consider a legal move and consider next level till the goal state is reached using backtracking / branch/ bound.

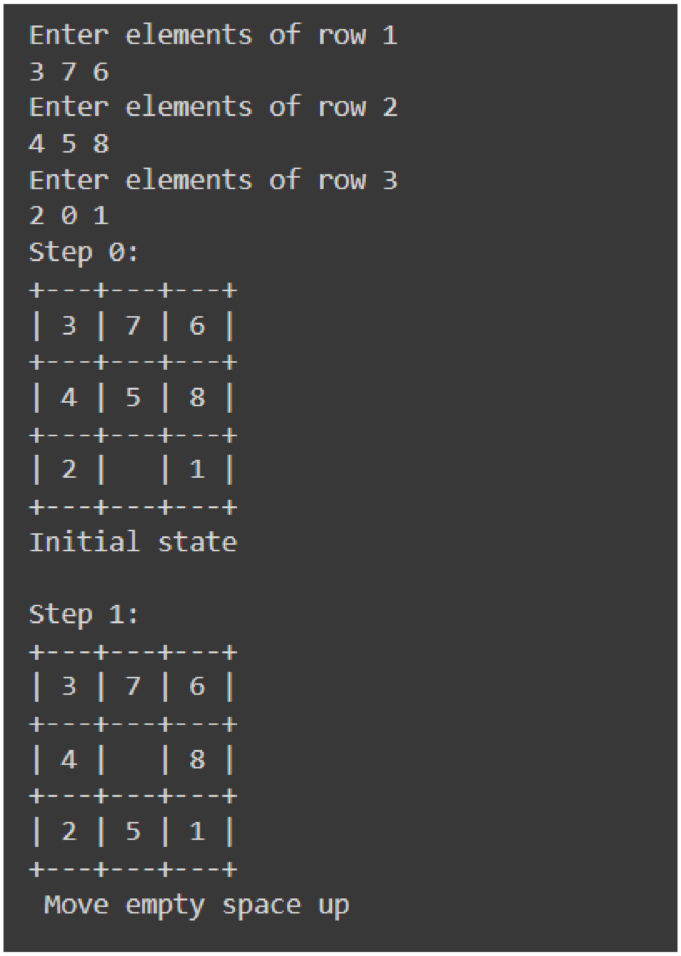
5. Using Manhattan distance and priority queue. optimize the level. at each state and move further based on minimum Manhattan distance.

lowest m,d

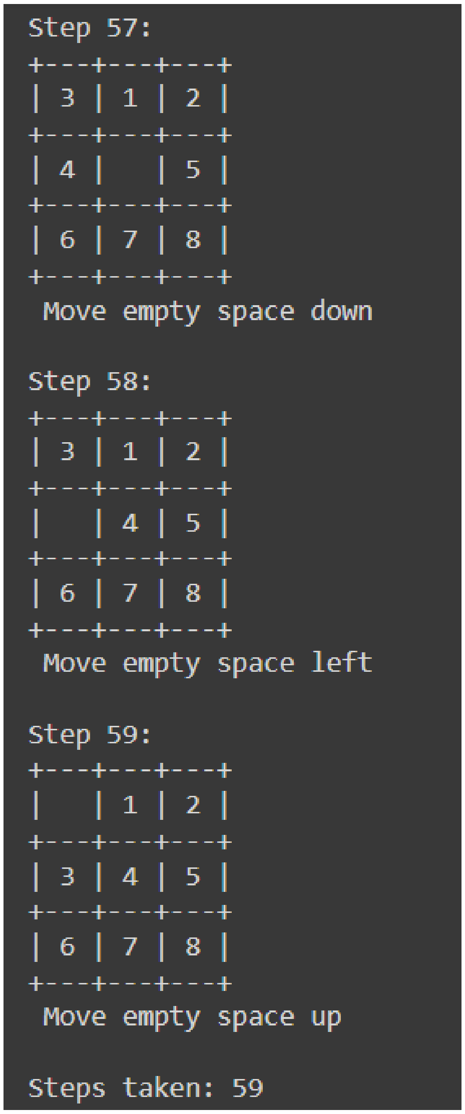
Code:

```
import heapq

import numpy as np


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

vis = set()

q = []

parent_map = {}

move_map = {}


def manhattan(curr):

    ans = 0

    pos = {goal[i][j]: (i, j) for i in range(3) for j in range(3)}

    for i in range(3):

        for j in range(3):

            x, y = pos[curr[i][j]]

            ans += abs(i - x) + abs(j - y)

    return ans


def moves(curr):

    x, y = [(i, j) for i in range(3) for j in range(3) if curr[i][j] == 0][0]

    poss = [[0, -1, 'left'], [-1, 0, 'up'], [1, 0, 'down'], [0, 1, 'right']]

    for dx, dy, direction in poss:

        nx, ny = x + dx, y + dy

        if 0 <= nx < 3 and 0 <= ny < 3:

            curr1 = [row[:] for row in curr]

            curr1[x][y], curr1[nx][ny] = curr1[nx][ny], curr1[x][y]
            tuple_curr1 = tuple(map(tuple, curr1))
            if tuple_curr1 not in vis:
                heapq.heappush(q, (manhattan(curr1), curr1))
                vis.add(tuple_curr1)
                parent_map[tuple(map(tuple, curr1))] = curr
                move_map[tuple(map(tuple, curr1))] = direction

def dfs(curr):
    vis.add(tuple(map(tuple, curr)))
    if curr == goal:
        return True
    moves(curr)
    if q:
        curr = heapq.heappop(q)[1]
        if dfs(curr):
            return True
    return False

def display_board(board):
    print("+---+---+---+")
    for row in board:
        print("| " + " | ".join(str(x) if x != 0 else ' ' for x in row) + " |")
        print("+---+---+---+")

c =[[] for i in range(3)]
for i in range(3):
  print(f"Enter elements of row {i+1}")
  c[i]=list(map(int,input().split()))
dfs(c)


result_path = []
directions = []
state = goal
while state:
    result_path.append(state)
    directions.append(move_map.get(tuple(map(tuple, state)), None))
    state = parent_map.get(tuple(map(tuple, state)))


for ind, (state, direction) in enumerate(reversed(list(zip(result_path, directions)))):
    print(f"Step {ind}:")
    display_board(state)
    if ind==0:
      print("Initial state")
    if direction:
        print(f" Move empty space {direction}")
    print()

print(f"Steps taken: {len(result_path) - 1}")
```

Output:



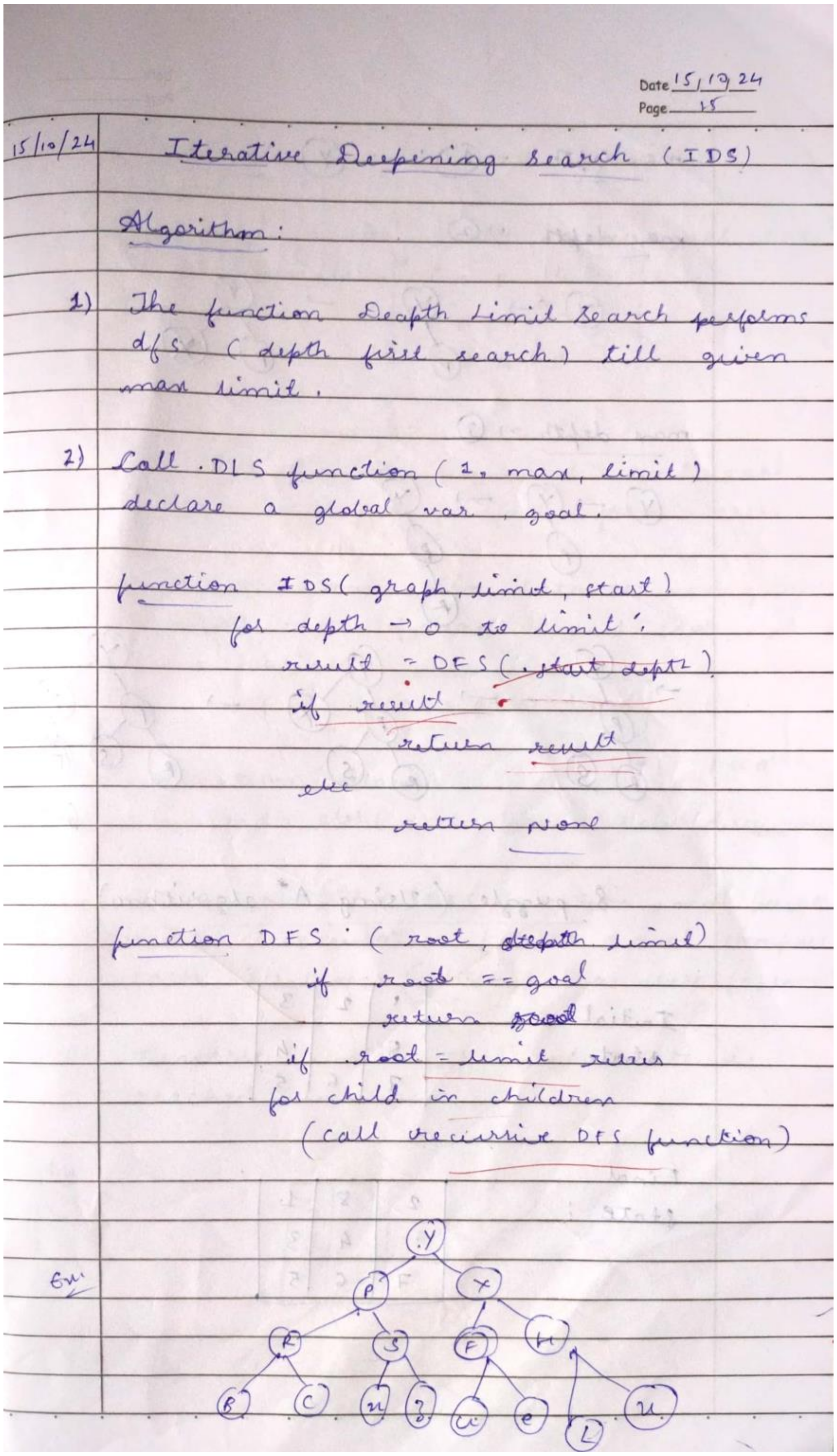
```
Enter elements of row 1
3 7 6
Enter elements of row 2
4 5 8
Enter elements of row 3
2 0 1
Step 0:
+---+---+---+
| 3 | 7 | 6 |
+---+---+---+
| 4 | 5 | 8 |
+---+---+---+
| 2 |   | 1 |
+---+---+---+
Initial state

Step 1:
+---+---+---+
| 3 | 7 | 6 |
+---+---+---+
| 4 |   | 8 |
+---+---+---+
| 2 | 5 | 1 |
+---+---+---+
 Move empty space up
```



```
Step 57:
+---+---+---+
| 3 | 1 | 2 |
+---+---+---+
| 4 |   | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space down

Step 58:
+---+---+---+
| 3 | 1 | 2 |
+---+---+---+
|   | 4 | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space left

Step 59:
+---+---+---+
|   | 1 | 2 |
+---+---+---+
| 3 | 4 | 5 |
+---+---+---+
| 6 | 7 | 8 |
+---+---+---+
 Move empty space up

Steps taken: 59
```

Implement Iterative deepening search algorithm.

Algorithm:



Date 15/10/24
Page 15

15/10/24

Iterative Deepening search (IDS)

Algorithm:

1) The function Depth Limit Search performs dfs (depth first search) till given max limit.

2) Call DLS function (1, max, limit)
declare a global var, goal.

```
function IDS( graph, limit, start)
        for depth -> 0 to limit:
            result = DFS( ~~, start~~ ~~depth~~ )
            if result
                return result
            else
                return None
```

```
function DFS : ( root, ~~depth~~ limit)
            if root == goal
                return ~~goal~~
        if root = limit return
        for child in children
            (call recursive DFS function)
```

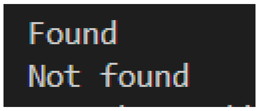
Ex:

Code:

```
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []  # List to hold children nodes

    def add_child(self, child_node):
        self.children.append(child_node)

def iddfs(root, goal):
    for i in range(0,100000):
        res=dls(root,goal,i)
        if res:
            print("Found")
            return
    print("Not found")

def dls(root,goal,depth):
    if depth==0:
        if root.value==goal:
            return True
        return False
    for child in root.children:
        if dls(child,goal,depth-1):
            return True
    return False

root=TreeNode("Y")
node1=TreeNode("P")
node2=TreeNode("X")
node3=TreeNode("R")
node4=TreeNode("S")
node5=TreeNode("F")
node6=TreeNode("H")
node7=TreeNode("B")
node8=TreeNode("C")
```

```
node9=TreeNode("S")

root.add_child(node1)
root.add_child(node2)

node1.add_child(node3)
node1.add_child(node4)

node2.add_child(node5)
node2.add_child(node6)

node3.add_child(node7)
node3.add_child(node8)

node4.add_child(node9)

iddfs(root, "F")
iddfs(root, "A")
```

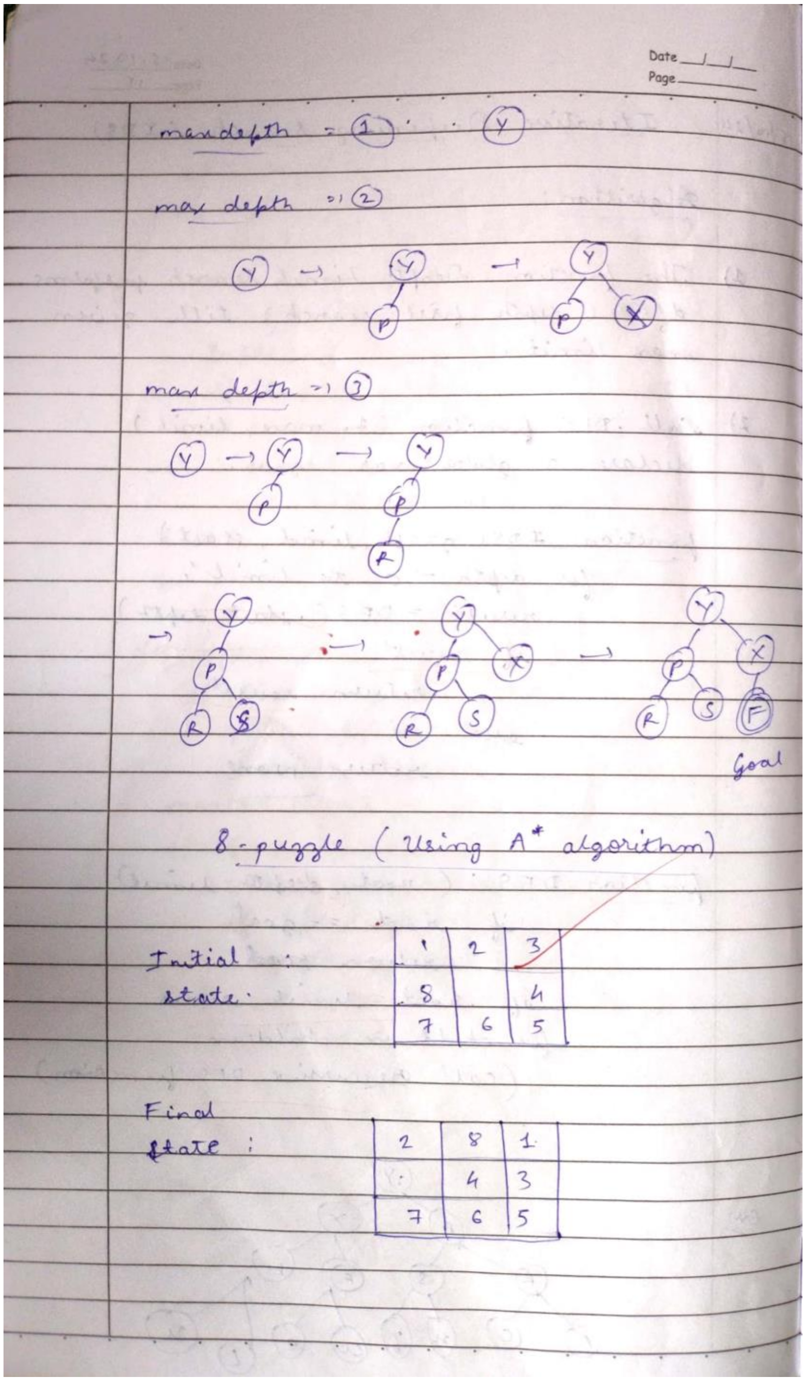
Output:



```
Found
Not found
```

Implement A* search algorithm.

Algorithm:



Date __/__/__

Page ____

maxdepth = 1

max depth => 2

max depth => 3

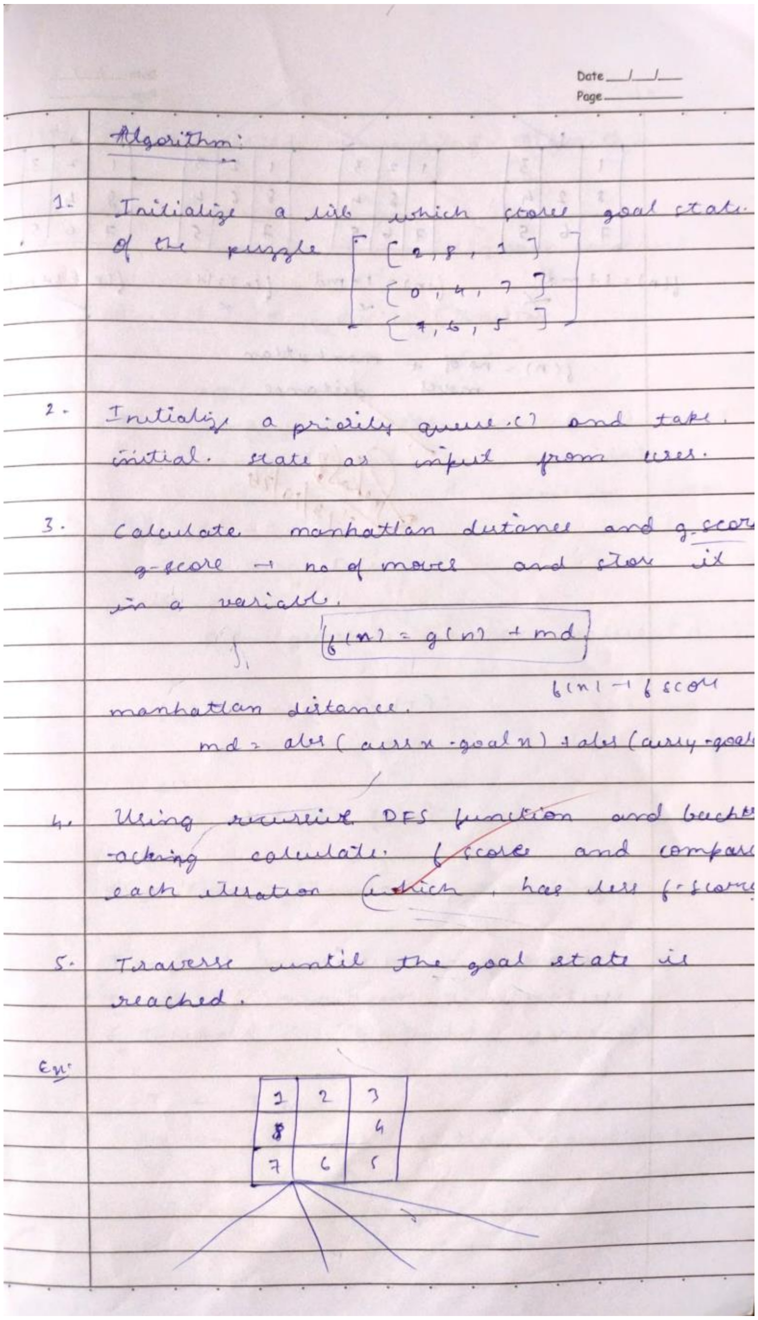
8-puzzle (Using A* algorithm)

Initial state:

1	2	3
8		4
7	6	5

Final state :

2	8	1
	4	3
7	6	5



Date __/__/__

Page ______

Algorithm:

1. Initialize a list which stores goal state of the puzzle $\begin{bmatrix} [2, 8, 1] \\ [0, 4, 7] \\ [7, 6, 5] \end{bmatrix}$

2. Initialize a priority queue () and take initial state as input from user.

3. Calculate manhattan distance and g-score
g-score → no of moves and store it in a variable.

$$\boxed{f(n) = g(n) + md}$$

f(n) → f score

manhattan distance.

md = abs (currx - goalx) + abs (curry - goal

4. Using recursive DFS function and backtr -acking calculate f score and compare each iteration (which has less f-score

5. Traverse until the goal state is reached.

Ex:

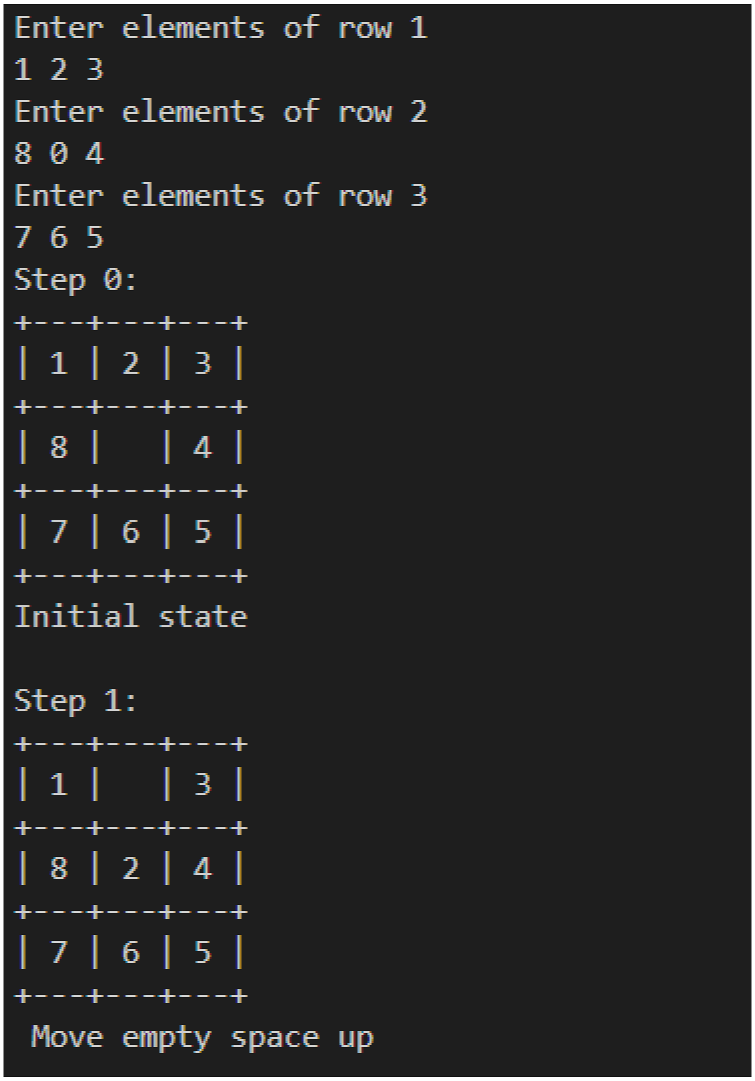
1	2	3
8		4
7	6	5

Code:

```
import heapq
import numpy as np

goal = [[2,8,1], [0,4,3], [7,6,5]]
vis = set()
q = []
parent_map = {}
move_map = {}

def manhattan(curr):
    ans = 0
    pos = {goal[i][j]: (i, j) for i in range(3) for j in range(3)}
    for i in range(3):
        for j in range(3):
            x, y = pos[curr[i][j]]
            ans += abs(i - x) + abs(j - y)
    return ans

def moves(curr,g):
    x, y = [(i, j) for i in range(3) for j in range(3) if curr[i][j] == 0][0]
    poss = [[0, -1, 'left'], [-1, 0, 'up'], [1, 0, 'down'], [0, 1, 'right']]
    for dx, dy, direction in poss:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:
            curr1 = [row[:] for row in curr]
            curr1[x][y], curr1[nx][ny] = curr1[nx][ny], curr1[x][y]
            tuple_curr1 = tuple(map(tuple, curr1))
            if tuple_curr1 not in vis:
                f=g+1+manhattan(curr1)
                heapq.heappush(q, (f, curr1,g+1))
                vis.add(tuple_curr1)
                parent_map[tuple(map(tuple, curr1))] = curr
                move_map[tuple(map(tuple, curr1))] = direction

def a_star(curr,g):
    vis.add(tuple(map(tuple, curr)))
    if curr == goal:
        return True
    moves(curr,g)
    if q:
        curr = heapq.heappop(q)
        if a_star(curr[1],curr[2]):
            return True
    return False
```

```
def display_board(board):
    print("+---+---+---+")
    for row in board:
        print("| " + " | ".join(str(x) if x != 0 else ' ' for x in row) + " |")
        print("+---+---+---+")

c =[[] for i in range(3)]
for i in range(3):
  print(f"Enter elements of row {i+1}")
  c[i]=list(map(int,input().split()))
a_star(c,0)

result_path = []
directions = []
state = goal
while state:
    result_path.append(state)
    directions.append(move_map.get(tuple(map(tuple, state)), None))
    state = parent_map.get(tuple(map(tuple, state)))

for ind, (state, direction) in enumerate(reversed(list(zip(result_path, directions)))):
    print(f"Step {ind}:")
    display_board(state)
    if ind==0:
      print("Initial state")
    if direction:
        print(f" Move empty space {direction}")
    print()

print(f"Steps taken: {len(result_path) - 1}")
```

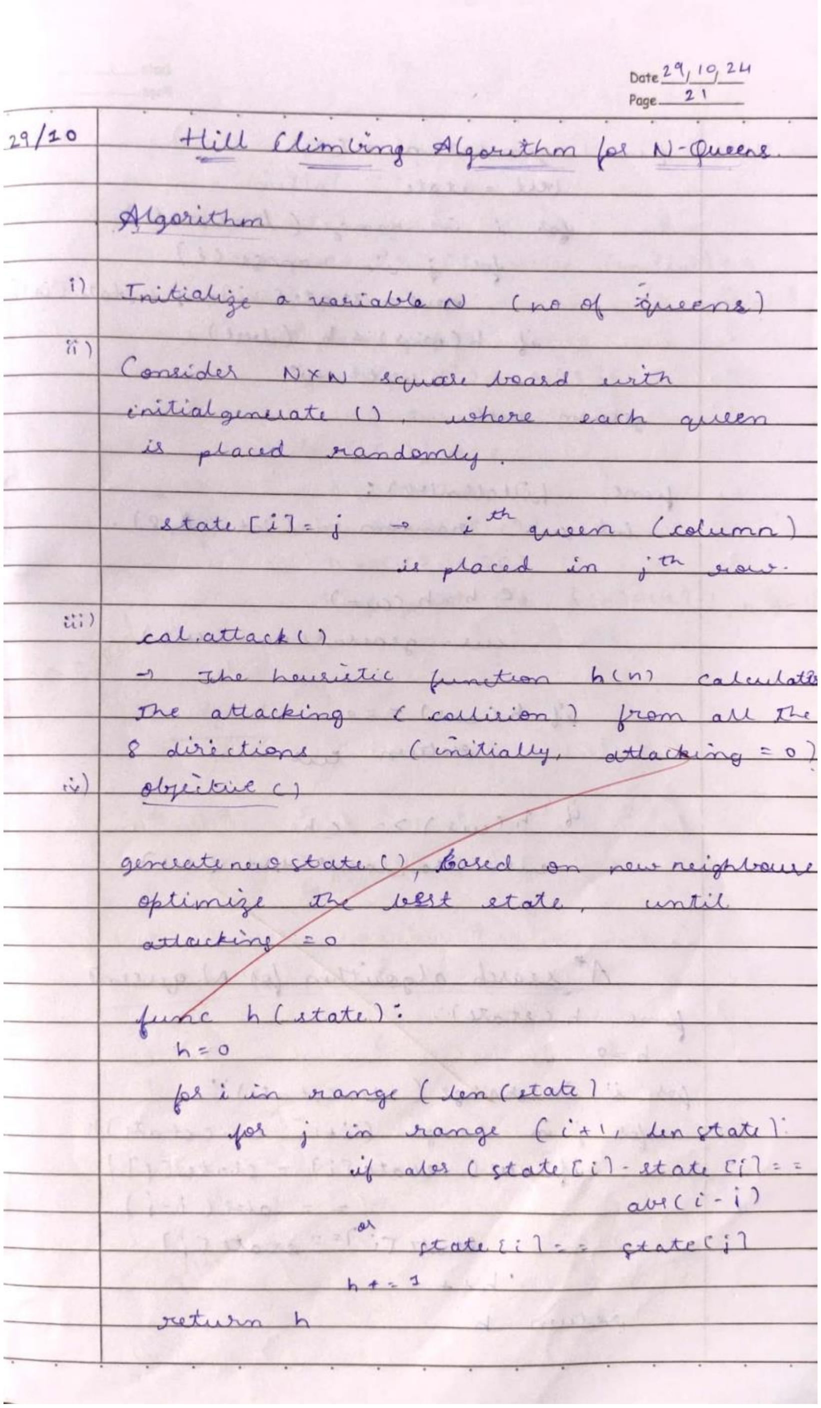
Output:



```
Enter elements of row 1
1 2 3
Enter elements of row 2
8 0 4
Enter elements of row 3
7 6 5
Step 0:
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
Initial state

Step 1:
+---+---+---+
| 1 |   | 3 |
+---+---+---+
| 8 | 2 | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
 Move empty space up
```

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

Algorithm:



Date 29, 10, 24
Page 21

29/10

Hill Climbing Algorithm for N-Queens.

Algorithm

i) Initialize a variable N (no of queens)

ii) Consider NXN square board with initialgenerate (), where each queen is placed randomly.

state [i] = j → i th queen (column) is placed in j th row.

iii) cal.attack ()
→ The heuristic function h(n) calculates The attacking (collision) from all the 8 directions (initially, attacking = 0)

iv) objective ()

generatenewstate (), based on new neighbours optimize the best state, until attacking = 0

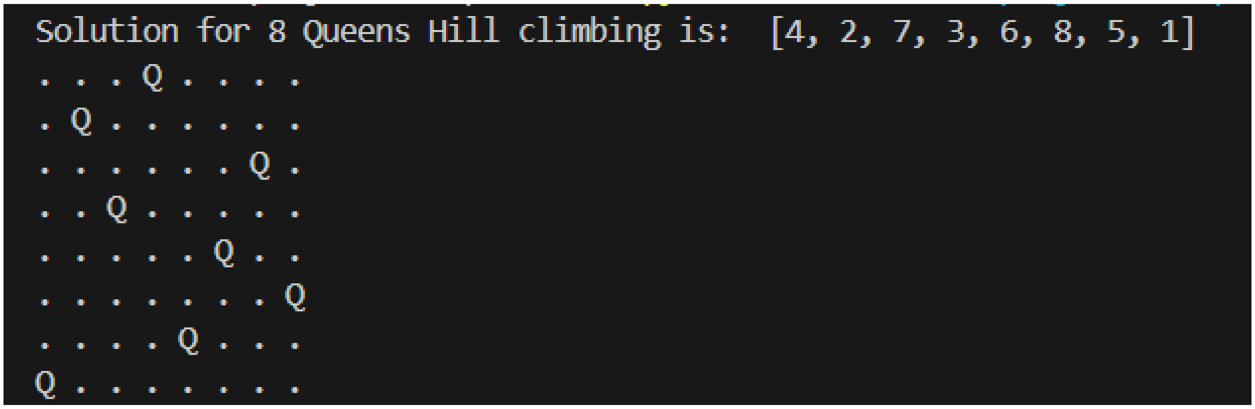
```
func h (state):
    h = 0
    for i in range ( len (state)
        for j in range ( i+1, len state):
            if abs ( state [i] - state [j] ==
                                    abs( i - j )
            or state [i] == state [j]
            h += 1
    return h
```

Code:

```
import random
def h(s):
    h = 0
    n = len(s)
    for i in range(n):
        for j in range(i + 1, n):
            if s[i] == s[j] or abs(s[i] - s[j]) == abs(i - j):
                h += 1
    return h

def new(s):
    best=s
    for i in range(len(s)):
        for j in range(1,9):
            if j!=s[i]:
                n=s[:i]+[j]+s[i+1:]
                if h(n)<h(best):
                    best=n
    return best

def hc():
    curr=[random.randint(1,8) for i in range(8)]
    while True:
        ch=h(curr)
        curr=new(curr)
        if h(curr)==0:
            return curr
        if h(curr)>=ch:
            curr=[random.randint(1,8) for i in range(8)]

def print_board(solution):
    print("Solution for 8 Queens Hill climbing is: ",solution)
    if solution is None:
        print("No solution found.")
        return

    board = [['.' for _ in range(8)] for _ in range(8)]

    for row in range(len(solution)):
        col = solution[row] - 1
        board[row][col] = 'Q'

    for row in board:
        print(' '.join(row))
```

print_board(hc())

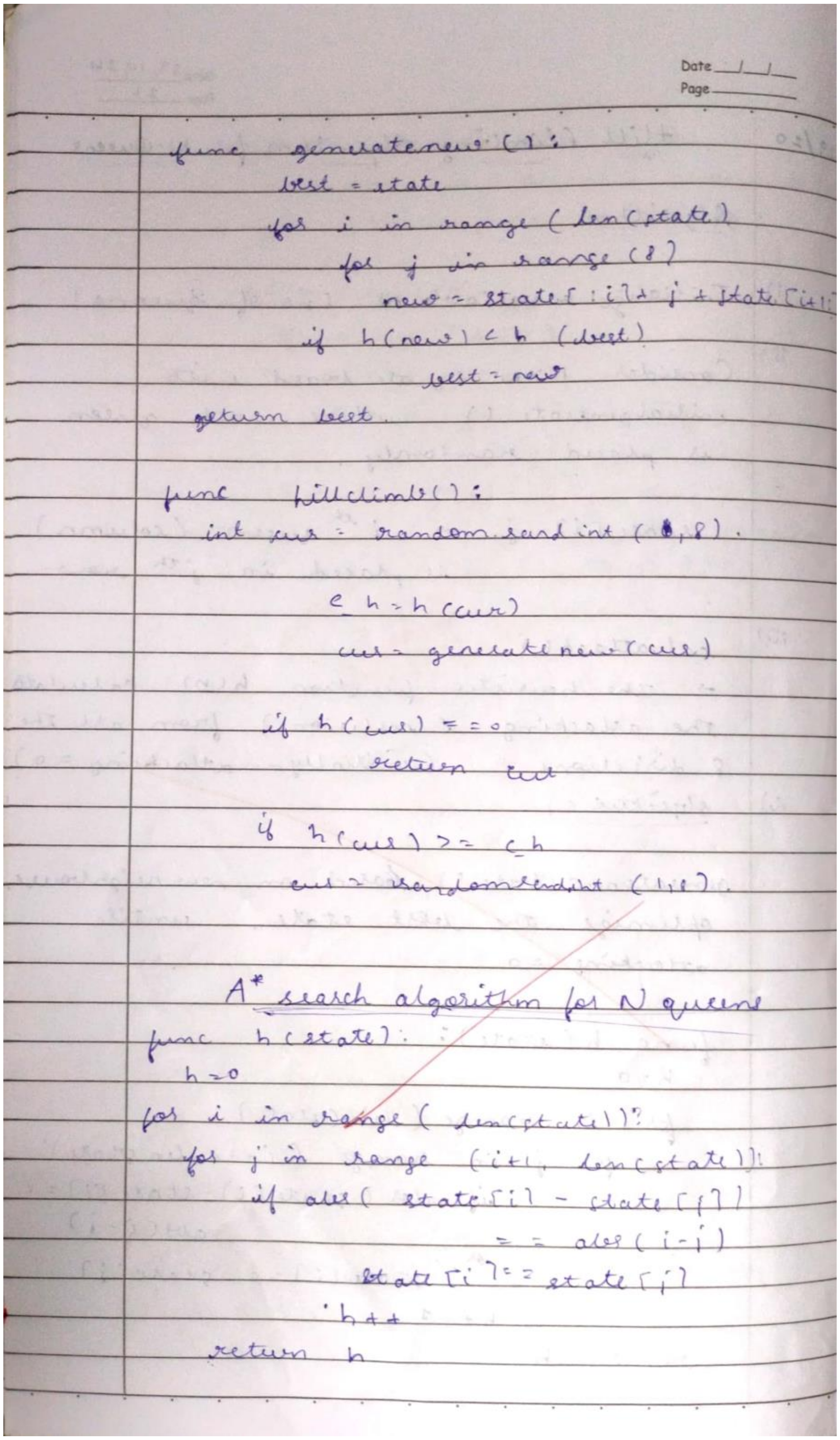
Output:



```
Solution for 8 Queens Hill climbing is:  [4, 2, 7, 3, 6, 8, 5, 1]
. . . Q . . . .
. Q . . . . . .
. . . . . . Q .
. . Q . . . . .
. . . . . Q . .
. . . . . . . Q
. . . . Q . . .
Q . . . . . . .
```

Implement A star algorithm to solve N-Queens problem.

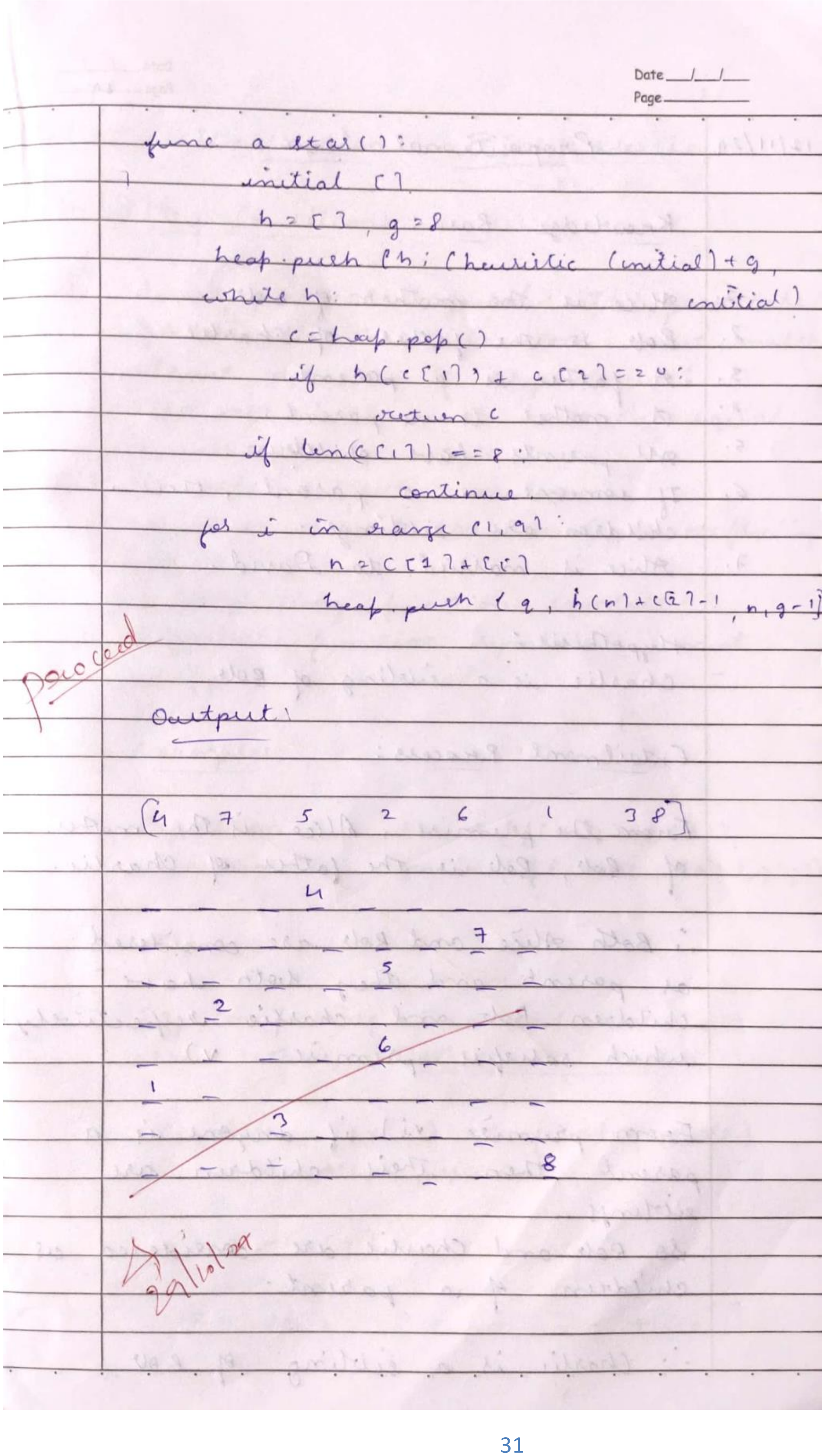
Algorithm:



Date__/__/__

Page____

```
func generatenew ():
        best = state
        for i in range ( len(state)
            for j in range (8)
                new = state[ :i]+ j + state [i+1:
            if h(new) < h (best)
                    best = new
    return best.

func    hillclimb():
    int cur = random.rand int ( 0,8).

        c_h = h (cur)
        cur = generatenew (cur)

    if h(cur) = = 0
            return cur

    if h (cur) >= c_h
        cur = randomrandint (1,8).


        A* search algorithm for N queens
func  h (state):
   h =0
for i in range ( len(state))?
   for j in range (i+1, len(state)):
       if abs( state[i] - state[j])
                       = = abs( i-j)
             state[i]= = state[j]
          h++
   return h
```



Date __/__/__
Page ______

```
func a star():
        initial []
          h = [], g = 8
      heap.push (h; (heuristic (initial) + g,
      while h:                          initial)
              c = heap.pop()
                if h(c[1]) + c[2] == 0:
                     return c
             if len(c[1]) == 8:
                            continue.
      for i in range (1, 9):
                n = c[1] + [i]
                  heap.push (q, h(n) + c[2] - 1, n, g-1)
```

Proceed

Output:

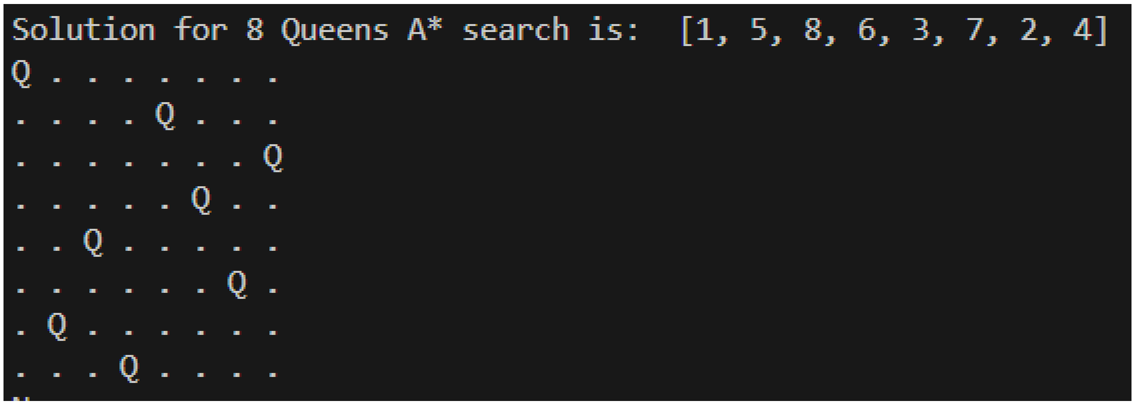
```
[4   7   5   2   6   1   3 8]

_   _   _   4   _   _   _   _
_   _   _   _   _   _   7   _
_   _   _   _   5   _   _   _
_   2   _   _   _   _   _   _
_   _   _   _   _   6   _   _
1   _   _   _   _   _   _   _
_   _   3   _   _   _   _   _
_   _   _   _   _   _   _   8
```

29/10/24

Code:

```
import heapq

def h(s):

    h = 0
    n = len(s)
    for i in range(n):
        for j in range(i + 1, n):
            if s[i] == s[j] or abs(s[i] - s[j]) == abs(i - j):
                h += 1
    return h

def a_star():
    initial_state = []
    q = []
    g = 8
    heapq.heappush(q, (h(initial_state), initial_state, g))

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

        if len(state) == 8 and h(state) == 0:
            return state

        for i in range(1, 9):
            if i not in state:
                new_state = state + [i]

                heapq.heappush(q, (h(new_state) + g, new_state, g - 1))


    return None


solution = a_star()

print("Solution:", solution)
```

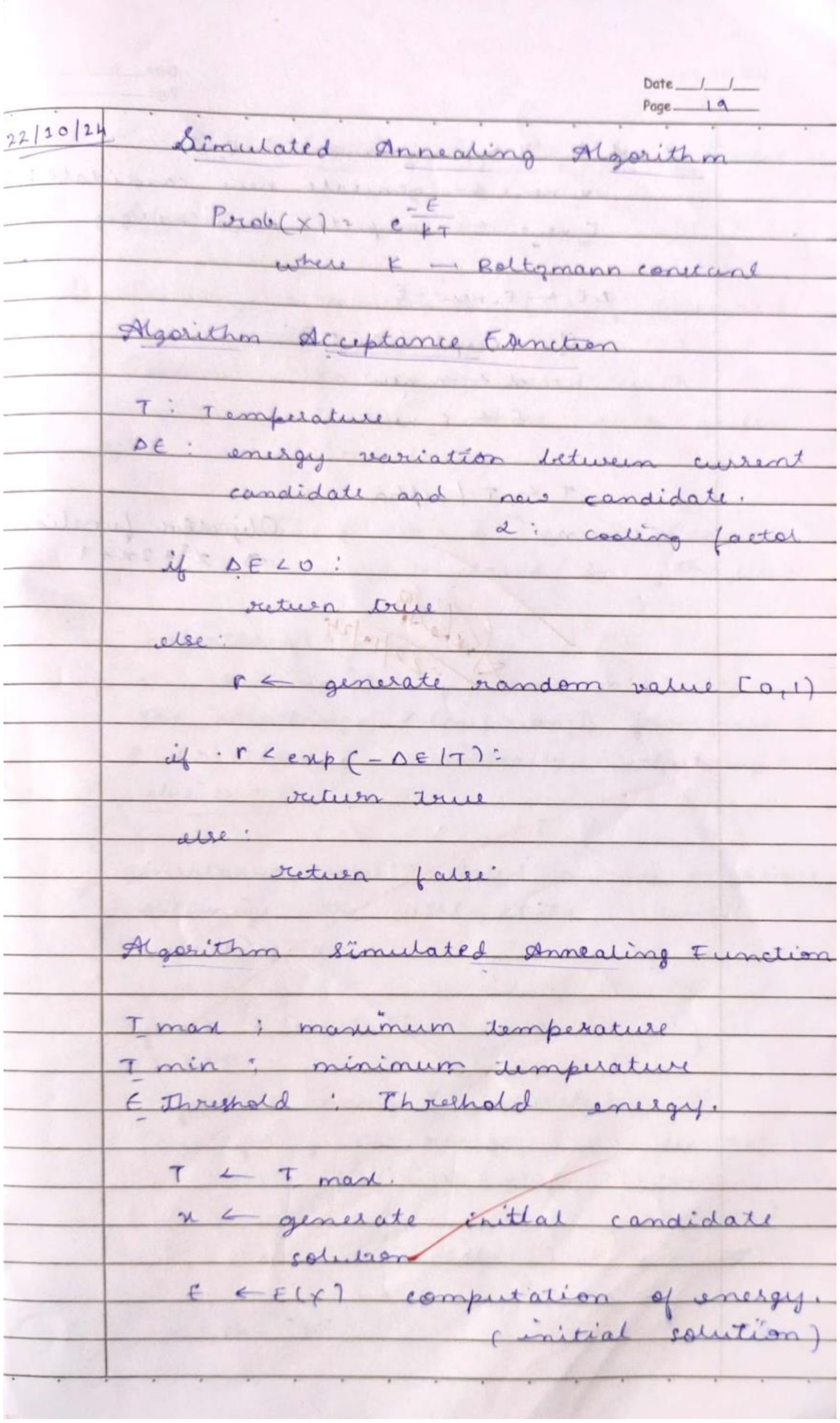
Output:



```
Solution for 8 Queens A* search is:  [1, 5, 8, 6, 3, 7, 2, 4]
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .
```

Implement Simulated Annealing:

Algorithm:



Date __/__/__
Page 19

22/10/24

Simulated Annealing Algorithm

$$Prob(x) = e^{\frac{-E}{kT}}$$

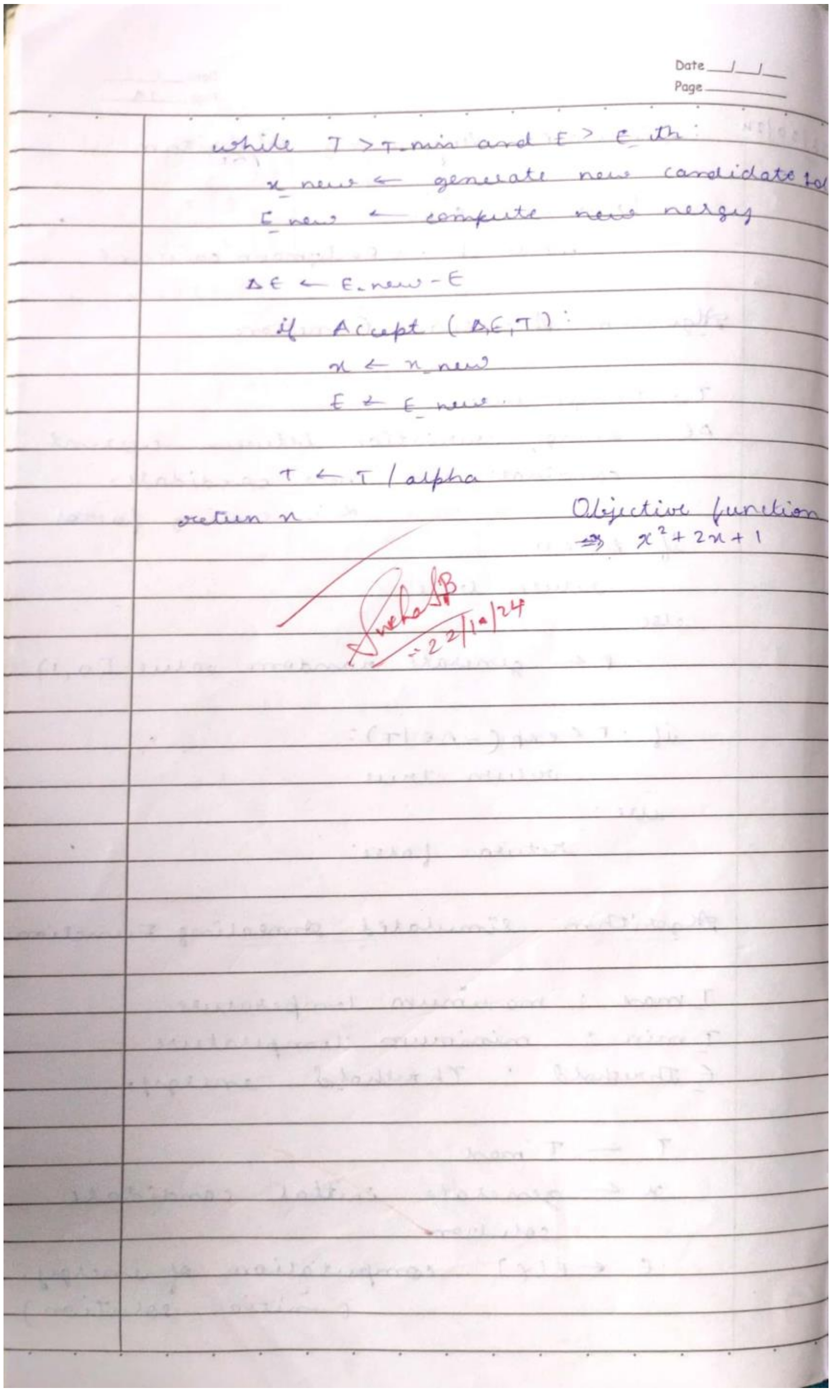
where k → Boltzmann constant

Algorithm Acceptance Function

```
T : Temperature
ΔE : energy variation between current
     candidate and new candidate.
                    α : cooling factor
  if ΔE < 0 :
       return true
  else :
       r ← generate random value [0,1)

  if r < exp(-ΔE/T) :
       return true
  else :
       return false
```

Algorithm Simulated Annealing Function

```
T_max : maximum temperature
T_min : minimum temperature
E_Threshold : Threshold energy.

   T ← T max.
   x ← generate initial candidate
         solution
   E ← E(x)  computation of energy.
                  (initial solution)
```



Date __/__/__

Page ______

```
while T > T_min and E > E_th :
    x_new ← generate new candidate sol
    E_new ← compute new nergy

    ΔE ← E_new - E

    if Accept (ΔE, T) :
        x ← x_new
        E ← E_new.

    T ← T / alpha
return x
```

Objective function

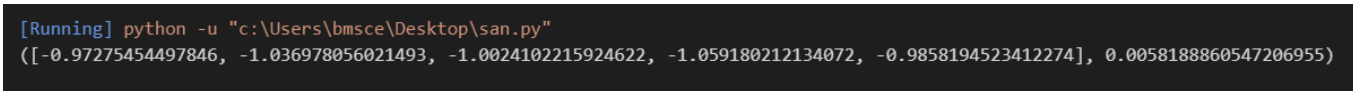
⇒ $x^2 + 2x + 1$

Sneha SB
22/10/24

Code:

```
import random
import math

def sim_anneal(ini, in_temp, max_iter, cool):
    # Initialize current state and best state
    curr_s = ini
    best_s = curr_s
    best_c = obj(best_s)
    temp = in_temp  # Set the initial temperature

    # While the temperature is above a threshold
    while temp > 1:
        for i in range(max_iter):
            new_s = neig(curr_s)
            curr_c = obj(curr_s)
            new_c = obj(new_s)

            if ap(curr_c, new_c, temp) > random.random():
                curr_s = new_s  # Move to the new state
            if new_c < best_c:
                best_s = new_s
                best_c = new_c
        temp *= cool

    return best_s, best_c

def neig(state):
    new_s = state.copy()
    ind = random.randint(0, len(state) - 1)
    new_s[ind] += random.uniform(-1, 1)
    return new_s

def obj(state):
    c = 1
    for i in state:
        c += i**2 + 2*i + 1
    return c

def ap(curr_c, new_c, temp):
    if new_c < curr_c:
        return 1
    else:
        return math.exp((curr_c - new_c) / temp)

print(sim_anneal([1, 2, 3, 4, 5], 1000, 1000, 0.99))
```

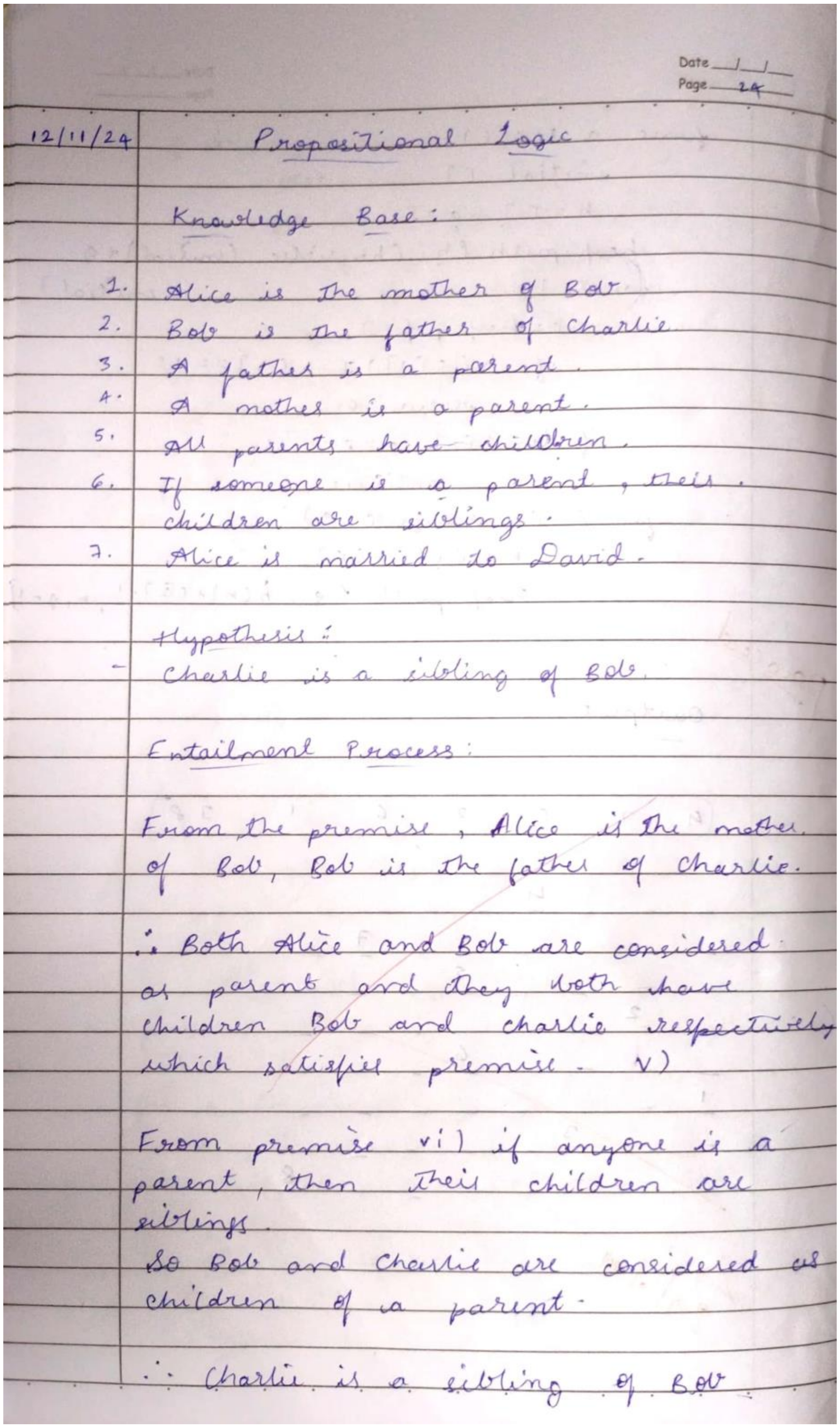
Output:



```
[Running] python -u "c:\Users\bmsce\Desktop\san.py"
([-0.97275454497846, -1.036978056021493, -1.0024102215924622, -1.059180212134072, -0.9858194523412274], 0.0058188860547206955)
```

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

Algorithm:



Date __/__/__
Page 24

12/11/24 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.

Entailment Process:

From the premise, Alice is the mother of Bob, Bob is the father of Charlie.

∴ Both Alice and Bob are considered as parent and they both have children Bob and charlie respectively which satisfies premise. v)

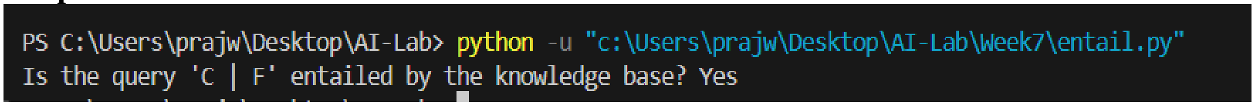
From premise vi) if anyone is a parent, then their children are siblings.
So Bob and Charlie are considered as children of a parent.

∴ Charlie is a sibling of Bob.

Code:

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

def is_entailment(kb, query):

   # Negate the query
   negated_query = Not(query)

   # Add negated query to the knowledge base
   kb_with_negated_query = And(*kb, negated_query)

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

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


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

# Check entailment
result = is_entailment(kb, query)
print(f"Is the query '{query}' entailed by the knowledge base? {'Yes' if result else 'No'}")
```

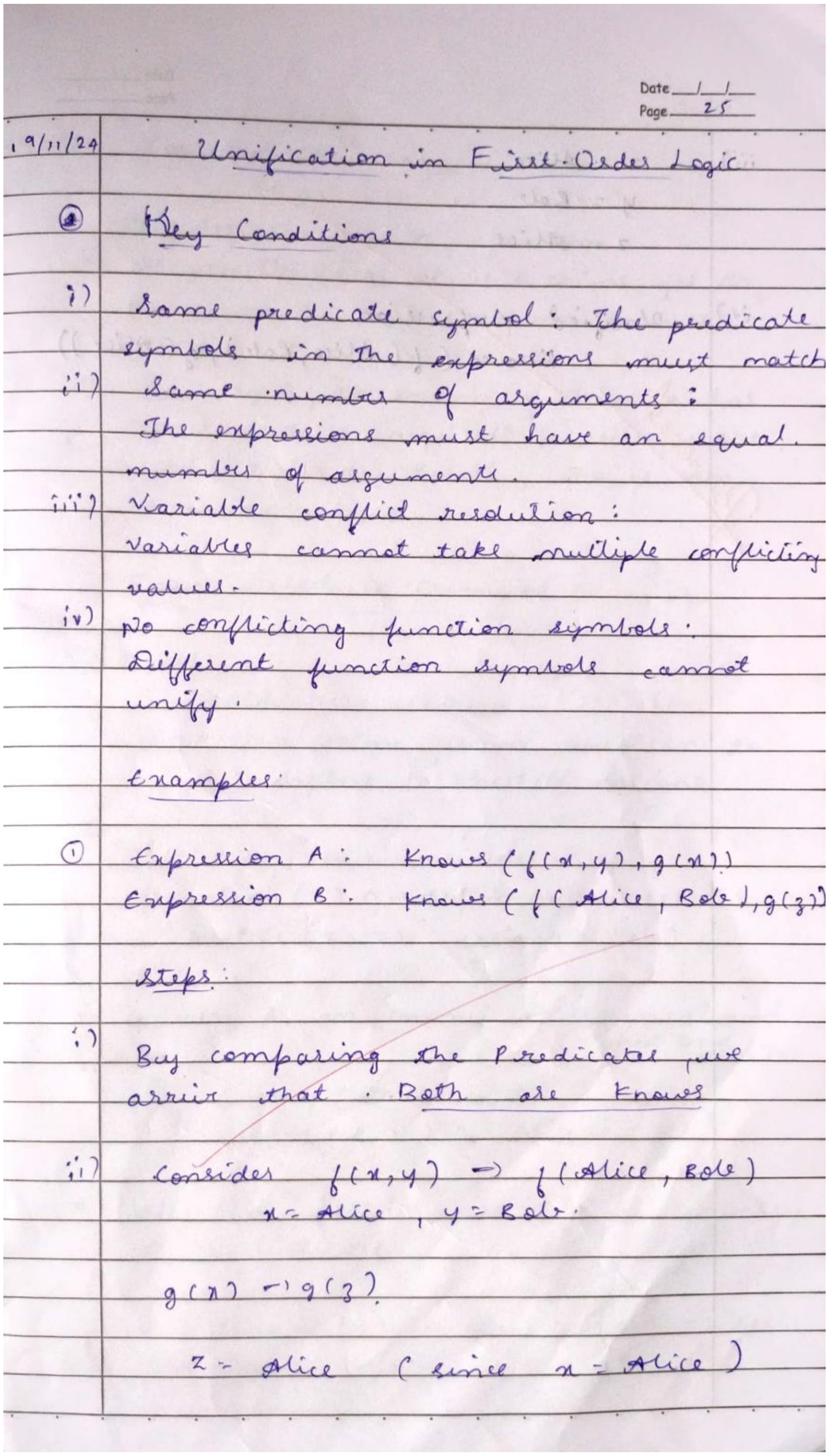
Output:



```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week7\entail.py"
Is the query 'C | F' entailed by the knowledge base? Yes
```

Implement unification in first order logic.

Algorithm:



Date __/__/__
Page 25

19/11/24

Unification in First-Order Logic

(a) Key Conditions

i) Same predicate symbol : The predicate symbols in the expressions must match

ii) Same number of arguments : The expressions must have an equal. number of arguments.

iii) Variable conflict resolution : variables cannot take multiple conflicting values.

iv) No conflicting function symbols : Different function symbols cannot unify.

Examples:

① Expression A : Knows (f(x, y), g(x))
Expression B : Knows (f(Alice , Bob), g(z))

Steps:

i) By comparing the Predicates , we arrive that : Both are Knows

ii) Consider f(x, y) → f(Alice , Bob)
x = Alice , y = Bob.

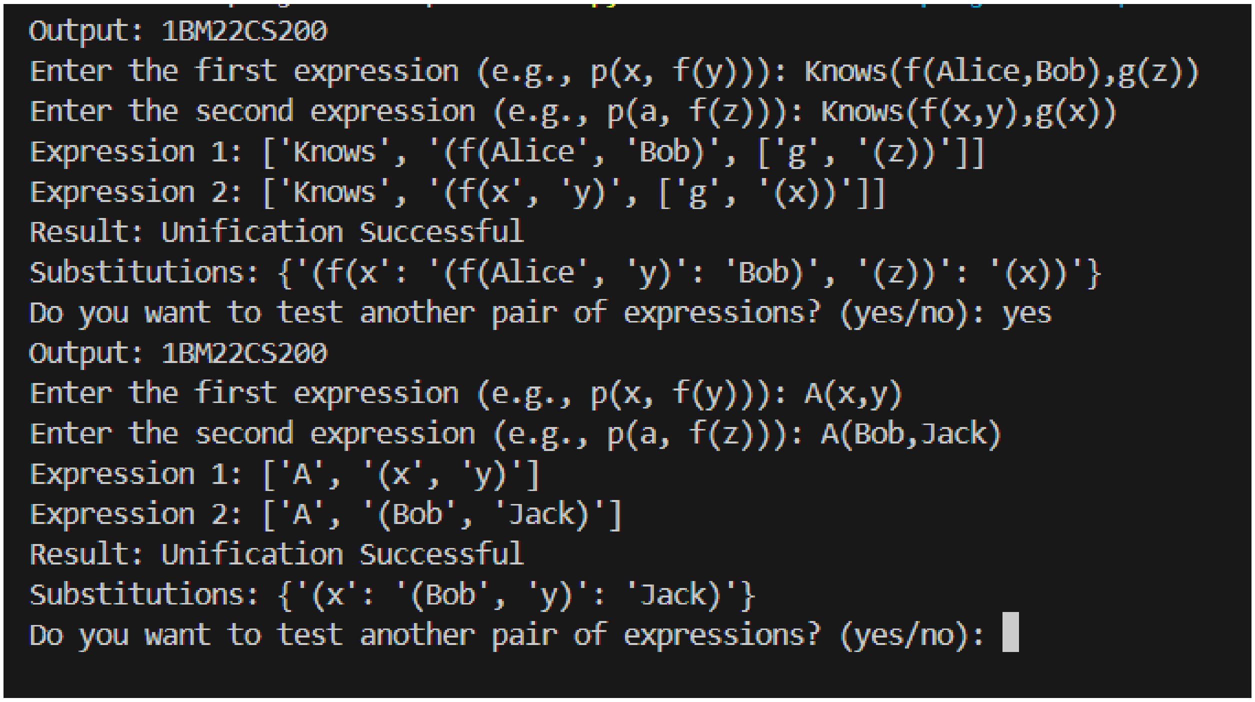
g(x) → g(z).

z = Alice (since x = Alice)

Code:

```
import re

def occurs_check(var, x):
    if var == x:
        return True
    elif isinstance(x, list):
        return any(occurs_check(var, xi) for xi in x)
    return False

def unify_var(var, x, subst):
    if var in subst:
        return unify(subst[var], x, subst)
    elif isinstance(x, (list, tuple)) and tuple(x) in subst:
        return unify(var, subst[tuple(x)], subst)
    elif occurs_check(var, x):
        return "FAILURE"
    else:
        subst[var] = tuple(x) if isinstance(x, list) else x
        return subst

def unify(x, y, subst=None):
    if subst is None:
        subst = {}
    if x == y:
        return subst
    elif isinstance(x, str) and x.islower():
        return unify_var(x, y, subst)
    elif isinstance(y, str) and y.islower():
        return unify_var(y, x, subst)
    elif isinstance(x, list) and isinstance(y, list):
        if len(x) != len(y):
            return "FAILURE"
        if x[0] != y[0]:
            return "FAILURE"
        for xi, yi in zip(x[1:], y[1:]):
            subst = unify(xi, yi, subst)
            if subst == "FAILURE":
                return "FAILURE"
        return subst
    else:
        return "FAILURE"

def unify_and_check(expr1, expr2):
    result = unify(expr1, expr2)
    if result == "FAILURE":
        return False, None
    return True, result

def display_result(expr1, expr2, is_unified, subst):
    print("Expression 1:", expr1)
    print("Expression 2:", expr2)
    if not is_unified:
        print("Result: Unification Failed")
    else:
        print("Result: Unification Successful")
        print("Substitutions:", {k: list(v) if isinstance(v, tuple) else v for k, v in subst.items()})

def parse_input(input_str):
    input_str = input_str.replace(" ", "")
    def parse_term(term):
        if '(' in term:
            match = re.match(r'([a-zA-Z0-9_]+)(.*)', term)
            if match:
                predicate = match.group(1)
                arguments_str = match.group(2)
                arguments = [parse_term(arg.strip()) for arg in arguments_str.split(',')]
                return [predicate] + arguments
        return term
    return parse_term(input_str)

def main():
    while True:
        expr1_input = input("Enter the first expression (e.g., p(x, f(y))): ")
        expr2_input = input("Enter the second expression (e.g., p(a, f(z))): ")
        expr1 = parse_input(expr1_input)
        expr2 = parse_input(expr2_input)
        is_unified, result = unify_and_check(expr1, expr2)
        display_result(expr1, expr2, is_unified, result)
        another_test = input("Do you want to test another pair of expressions? (yes/no): ").strip().lower()
        if another_test != 'yes':
            break

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

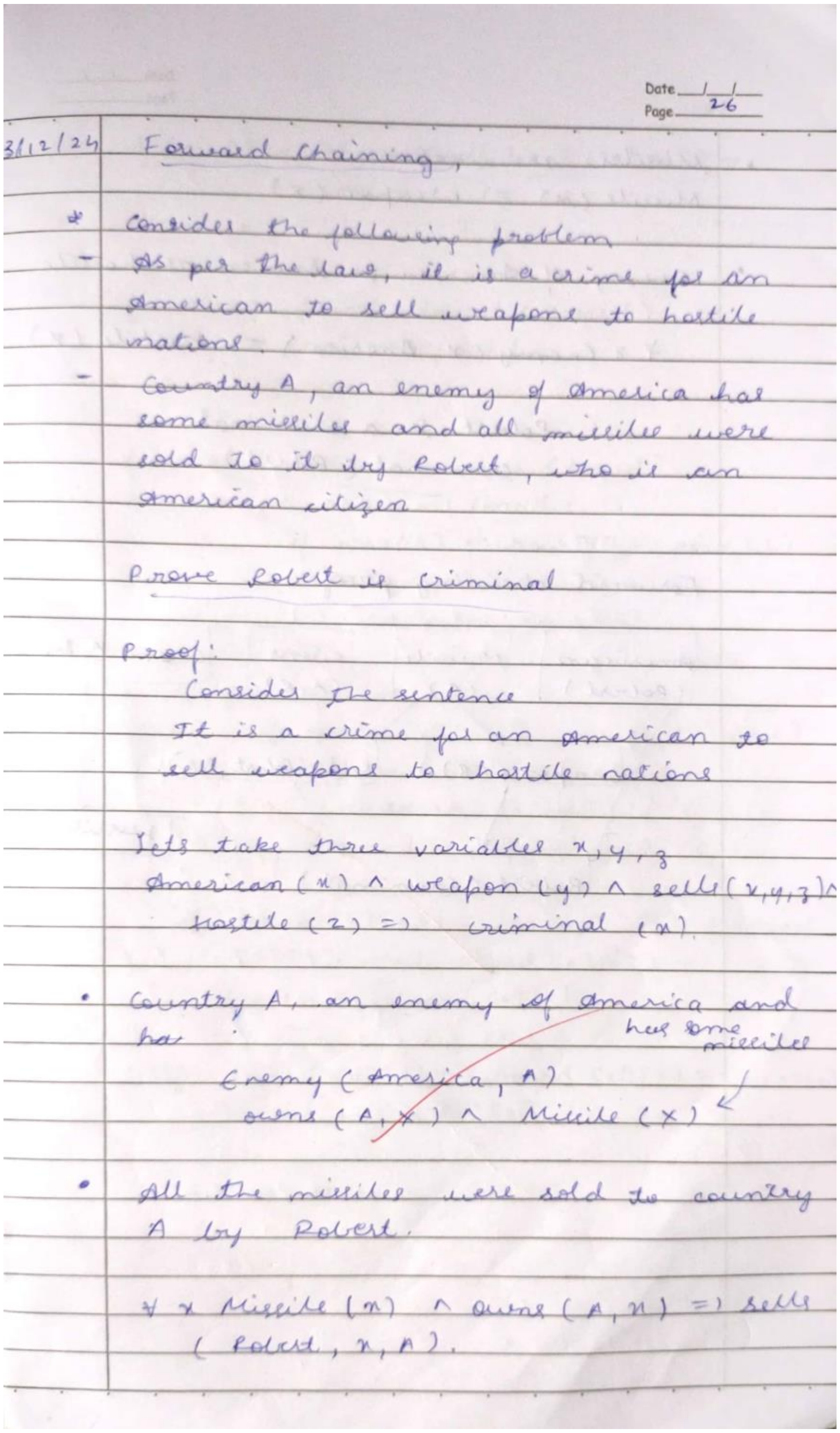
Output:



```
Output: 1BM22CS200
Enter the first expression (e.g., p(x, f(y))): Knows(f(Alice,Bob),g(z))
Enter the second expression (e.g., p(a, f(z))): Knows(f(x,y),g(x))
Expression 1: ['Knows', '(f(Alice', 'Bob)', ['g', '(z))']]
Expression 2: ['Knows', '(f(x', 'y)', ['g', '(x))']]
Result: Unification Successful
Substitutions: {'(f(x': '(f(Alice', 'y)': 'Bob)', '(z))': '(x))'}
Do you want to test another pair of expressions? (yes/no): yes
Output: 1BM22CS200
Enter the first expression (e.g., p(x, f(y))): A(x,y)
Enter the second expression (e.g., p(a, f(z))): A(Bob,Jack)
Expression 1: ['A', '(x', 'y)']
Expression 2: ['A', '(Bob', 'Jack)']
Result: Unification Successful
Substitutions: {'(x': '(Bob', 'y)': 'Jack)'}
Do you want to test another pair of expressions? (yes/no):
```

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

Algorithm:



Date __/__/__
Page 26

3/12/24 Forward chaining,

* Consider the following 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 missiles were sold to it by Robert, who is an American citizen

Prove Robert is criminal

Proof:

Consider the sentence

It is a crime for an American to sell weapons to hostile nations

Lets take three variables x, y, z

$American(x) \wedge weapon(y) \wedge sells(x,y,z) \wedge Hostile(z) \Rightarrow criminal(x).$

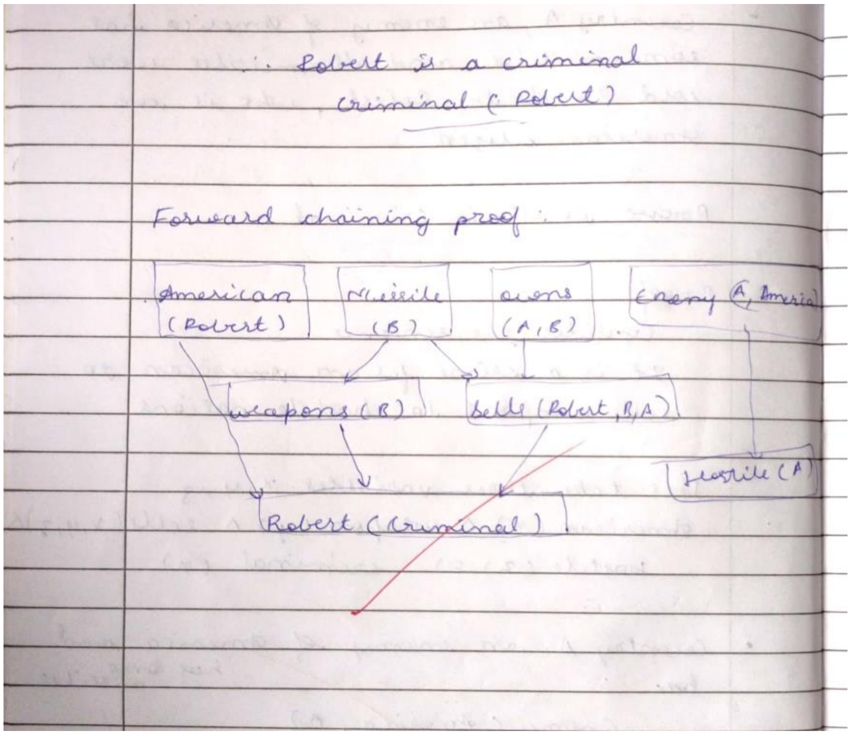
- Country A, an enemy of America and has : has some missiles

$Enemy(America, A)$

$owns(A, X) \wedge Missile(X)$

- All the missiles were sold to country A by Robert.

$\forall x \; Missile(x) \wedge owns(A, x) \Rightarrow sells(Robert, x, A).$



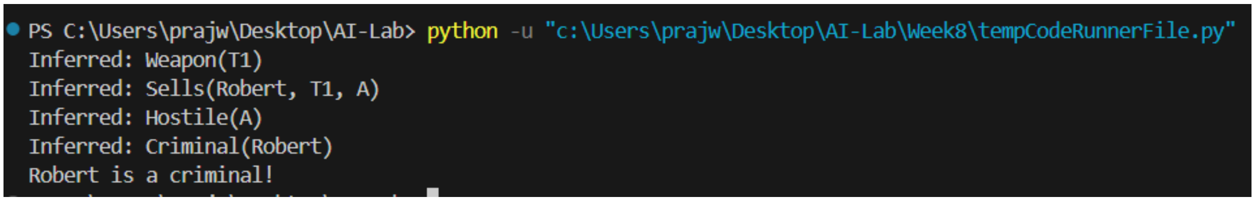
∴ Robert is a criminal
criminal (Robert)

Forward chaining proof :

Code:

```
KB = set()

KB.add('American(Robert)')
KB.add('Enemy(America, A)')
KB.add('Missile(T1)')
KB.add('Owns(A, T1)')

def modus_ponens(fact1, fact2, conclusion):
    if fact1 in KB and fact2 in KB:
        KB.add(conclusion)
        print(f"Inferred: {conclusion}")

def forward_chaining():
    if 'Missile(T1)' in KB:
        KB.add('Weapon(T1)')
        print(f"Inferred: Weapon(T1)")

    if 'Owns(A, T1)' in KB and 'Weapon(T1)' in KB:
        KB.add('Sells(Robert, T1, A)')
        print(f"Inferred: Sells(Robert, T1, A)")

    if 'Enemy(America, A)' in KB:
        KB.add('Hostile(A)')
        print(f"Inferred: Hostile(A)")
    if 'American(Robert)' in KB and 'Weapon(T1)' in KB and 'Sells(Robert, T1, A)' in KB and 'Hostile(A)' in KB:
        KB.add('Criminal(Robert)')
        print("Inferred: Criminal(Robert)")

    if 'Criminal(Robert)' in KB:
        print("Robert is a criminal!")
    else:
        print("No more inferences can be made.")

forward_chaining()
```

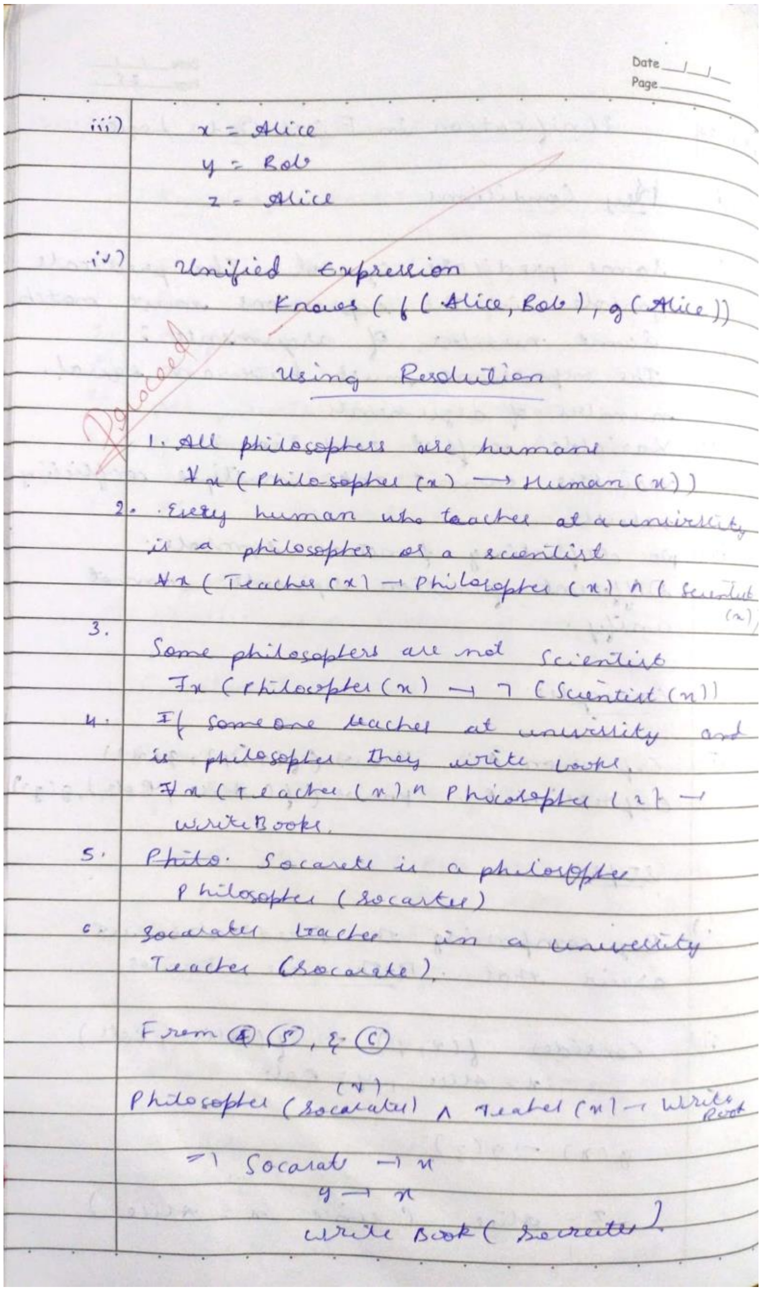
Output:



```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week8\tempCodeRunnerFile.py"
  Inferred: Weapon(T1)
  Inferred: Sells(Robert, T1, A)
  Inferred: Hostile(A)
  Inferred: Criminal(Robert)
  Robert is a criminal!
```

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

Algorithm:



Date __/__/__
Page ____

iii) x = Alice
y = Bob
z = Alice

iv) Unified Expression
Knows (f (Alice, Bob), g (Alice))

Using Resolution

1. All philosophers are humans
$\forall x$ (Philo-sopher (x) $\longrightarrow$ Human (x))

2. Every human who teaches at a university is a philosopher or a scientist
$\forall x$ (Teacher (x) $\rightarrow$ Philosopher (x) $\wedge$ (Scientist (x)

3. Some philosophers are not Scientist
$\exists x$ (Philosopher (x) $\rightarrow$ $\neg$ (Scientist (x))

4. If someone teaches at university and is philosopher they write books.
$\forall x$ (Teacher (x) $\wedge$ Philosopher (x) $\rightarrow$ writeBooks.

5. ~~Philo.~~ Socarete is a philosopher
Philosopher (Socartes)

6. Socarates teaches in a university
Teacher (Socarate).

From ④ ⑤, & ⓒ

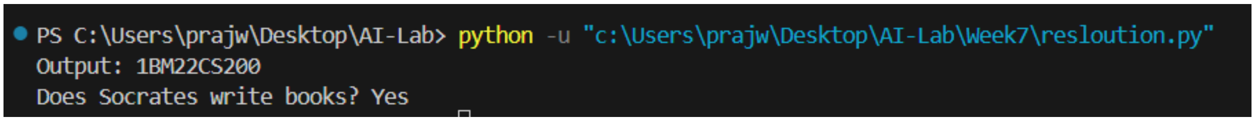
Philosopher (Socarates (x)) $\wedge$ Teacher (x) $\rightarrow$ Write Book

$\Rightarrow$ Socarate $\rightarrow x$
$y \rightarrow x$
write Book (Socrates).

Code:

```
# Define the knowledge base (KB)
KB = {
    # Rules and facts
    "philosopher(X)": "human(X)",  # Rule 1: All philosophers are humans
    "human(Socrates)": True,  # Socrates is human (deduced from philosopher)
    "teachesAtUniversity(X)": "philosopher(X) or scientist(X)",  # Rule 2
    "some(philosopher, not scientist)": True,  # Rule 3: Some philosophers are not scientists
    "writesBooks(X)": "teachesAtUniversity(X) and philosopher(X)",  # Rule 4
    "philosopher(Socrates)": True,  # Fact: Socrates is a philosopher
    "teachesAtUniversity(Socrates)": True,  # Fact: Socrates teaches at university
}

# Function to evaluate a predicate based on the KB
def resolve(predicate):
    # If it's a direct fact in KB
    if predicate in KB and isinstance(KB[predicate], bool):
        return KB[predicate]

    # If it's a derived rule
    if predicate in KB:
        rule = KB[predicate]

        if " and " in rule:  # Handle conjunction
            sub_preds = rule.split(" and ")
            return all(resolve(sub.strip()) for sub in sub_preds)
        elif " or " in rule:  # Handle disjunction
            sub_preds = rule.split(" or ")
            return any(resolve(sub.strip()) for sub in sub_preds)
        elif "not " in rule:  # Handle negation
            sub_pred = rule[4:]  # Remove "not "
            return not resolve(sub_pred.strip())
        else:  # Handle single predicate
            return resolve(rule.strip())

    # If the predicate contains variables
    if "(" in predicate:
        func, args = predicate.split("(")
        args = args.strip(")").split(", ")
        # Handle philosopher and human link
        if func == "philosopher":
            return resolve(f"human({args[0]})")
        # Handle writesBooks rule explicitly
        if func == "writesBooks":
            return resolve(f"teachesAtUniversity({args[0]})") and resolve(f"philosopher({args[0]})")
    # Default to False if no rule or fact applies
    return False

# Query to check if Socrates writes books
query = "writesBooks(Socrates)"
result = resolve(query)

# Print the result
print("Output: 1BM22CS200")
print(f"Does Socrates write books? {'Yes' if result else 'No'}")
```

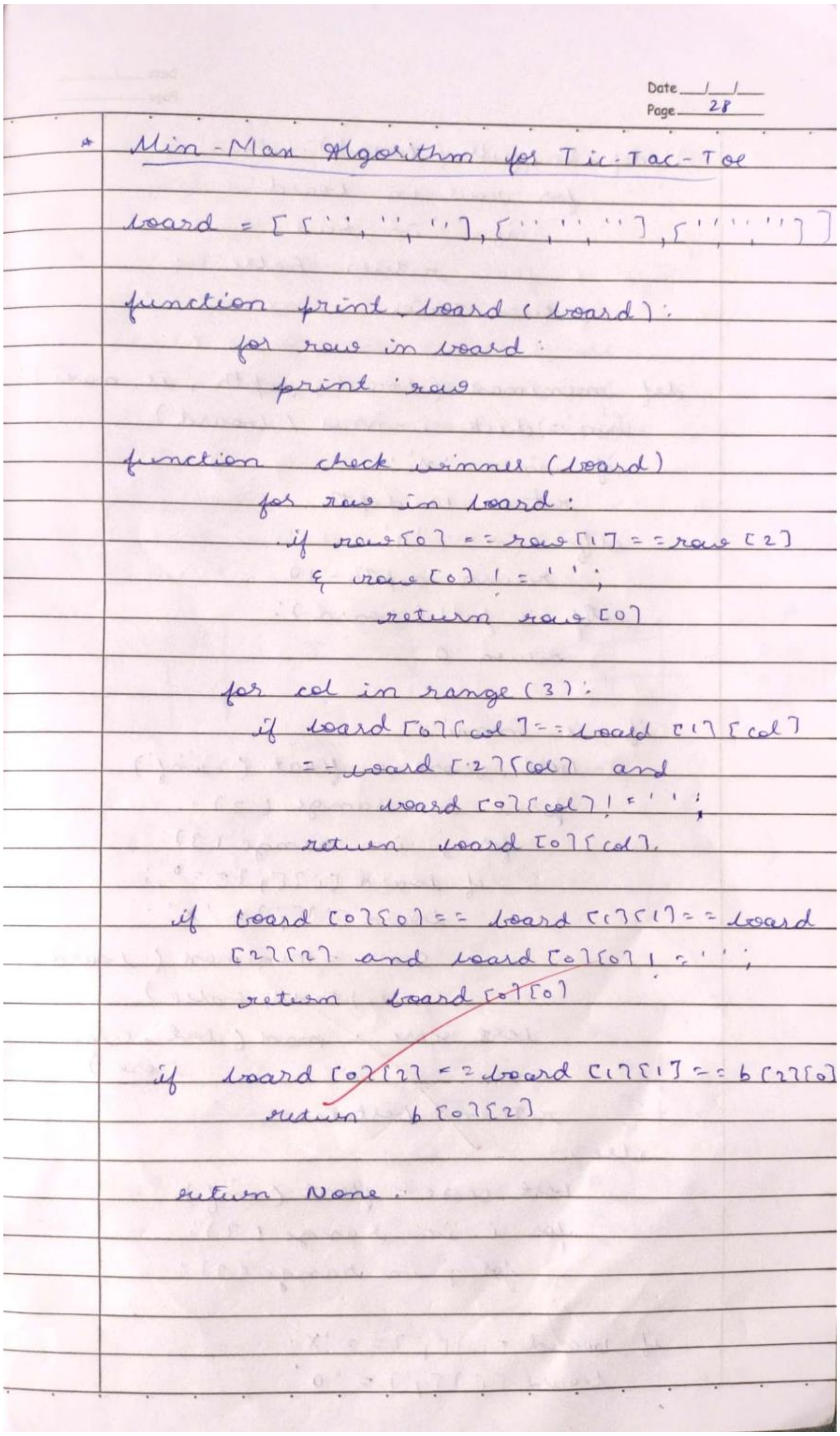
Output:



```
PS C:\Users\prajw\Desktop\AI-Lab> python -u "c:\Users\prajw\Desktop\AI-Lab\Week7\resloution.py"
Output: 1BM22CS200
Does Socrates write books? Yes
```

Implement MinMax Algorithm for TicTacToe.

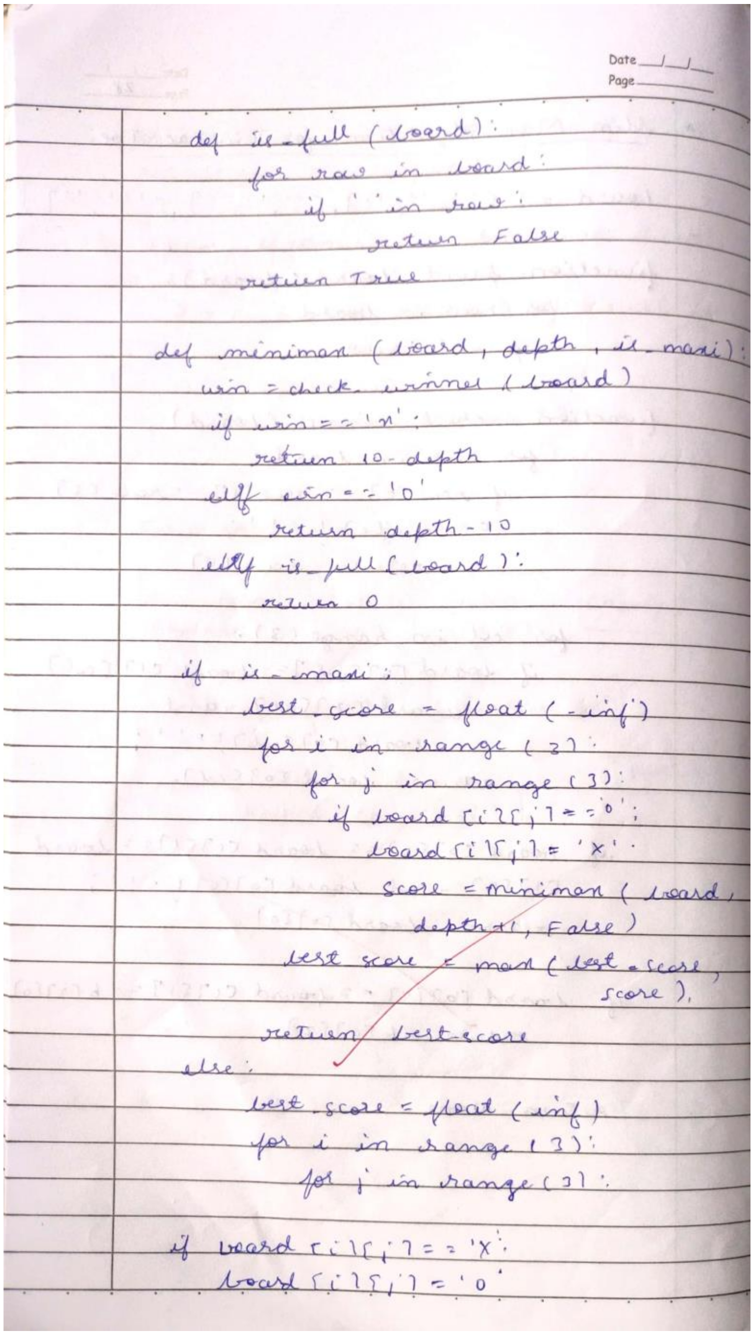
Algorithm:



Date__/__/__
Page 28

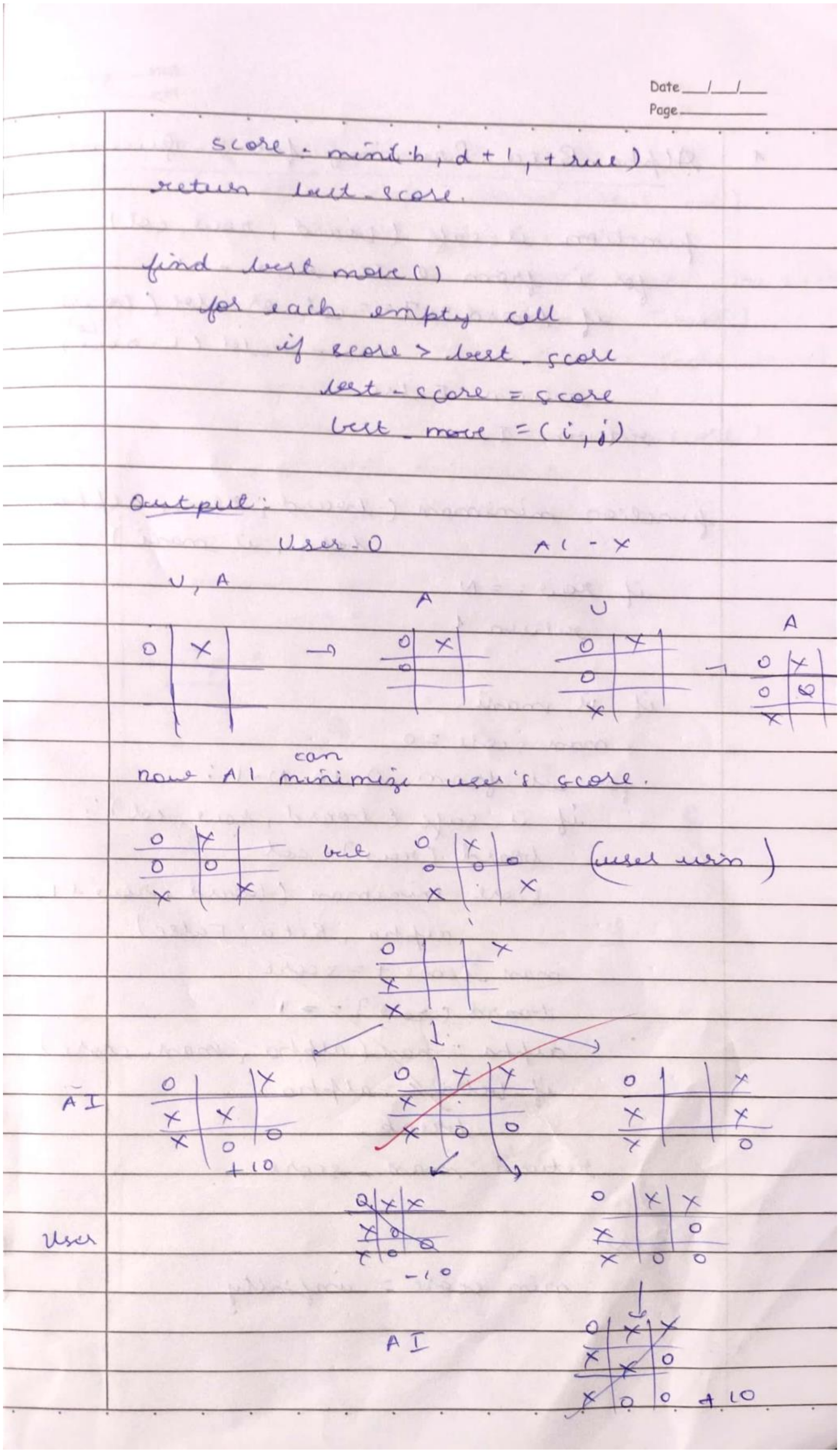
* Min-Max Algorithm for Tic-Tac-Toe

```
board = [['', '', ''], ['', '', ''], ['', '', '']]

function print_board (board):
        for row in board:
            print row

function   check winner (board)
           for row in board:
               if row[0] == row[1] == row[2]
                  & row[0] != '';
                      return row[0]

        for col in range(3):
            if board[0][col] == board [1][col]
               == board [2][col]   and
                     board [0][col] != '';
               return   board [0][col].

     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] == b[2][0]
           return   b[0][2]

     return None.
```



Date___/___/___
Page________

```
def is_full (board):
        for raw in board:
            if ' ' in raw:
                return False
    return True

def minimax (board, depth, is_maxi):
    win = check_winner (board)
    if win == 'x':
        return 10-depth
    elif win == '0'
        return depth-10
    elif is_full (board):
        return 0

  if is_maxi:
        best_score = float (-inf')
        for i in range (3):
            for j in range (3):
                if board [i][j] == '0':
                    board [i][j] = 'x'.
                    score = minimax (board,
                            depth+1, False)
            best score = max (best_score,
                                score),
        return best_score
  else:
        best_score = float (inf)
        for i in range (3):
            for j in range (3):

 if board [i][j] == 'X':
    board [i][j] = '0'
```



Date__/__/__
Page____

```
    score = mini(b, d + 1, true)
return best_score.

find best move()
    for each empty cell
        if score > best_score
            best_score = score
            best_move = (i, j)
```

Output:

User - O   AI - X

U, A

```
                 A              U               A
O | X |          O | X |        O | X |         O | X |
  |   |    ->    O |   |        O |   |    ->   O | O |
  |   |            |   |        X |   |         X |   |
```

now AI can minimize user's score.

```
O | X |             O | X |
O | O |     but     O | O | O     (user win)
X |   | X           X |   | X
```

```
                         O |   | X
                         X |   |
                         X |   |

AI      O |   | X        O | X | X        O |   | X
        X | X |          X |   |          X |   | X
        X | O | O        X | O | O        X |   | O
           +10

User                     O | X | X        O | X | X
                         X | O | O        X |   | O
                         X | O |          X | O | O
                            -10

                         AI               O | X | X
                                          X | X | O
                                          X | O | O   +10
```

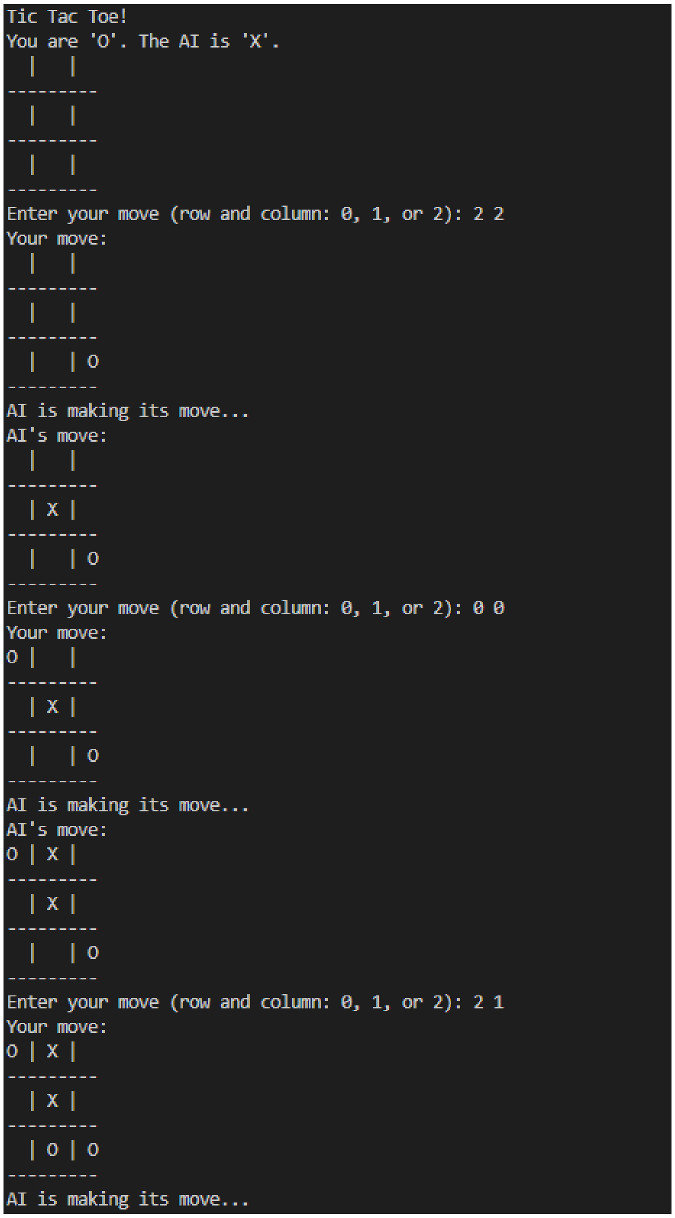
Code:

```
import math


def printBoard(board):
    for row in board:
        print(" | ".join(cell if cell != "" else " " for cell in row))
        print("-" * 9)


def evaluateBoard(board):
    for row in board:
        if row[0] == row[1] == row[2] and row[0] != "":
            return 10 if row[0] == 'X' else -10
    for col in range(3):
        if board[0][col] == board[1][col] == board[2][col] and board[0][col] != "":
            return 10 if board[0][col] == 'X' else -10
    if board[0][0] == board[1][1] == board[2][2] and board[0][0] != "":
        return 10 if board[0][0] == 'X' else -10
    if board[0][2] == board[1][1] == board[2][0] and board[0][2] != "":
        return 10 if board[0][2] == 'X' else -10
    return 0


def isDraw(board):
    for row in board:
        if "" in row:
            return False
    return True
```

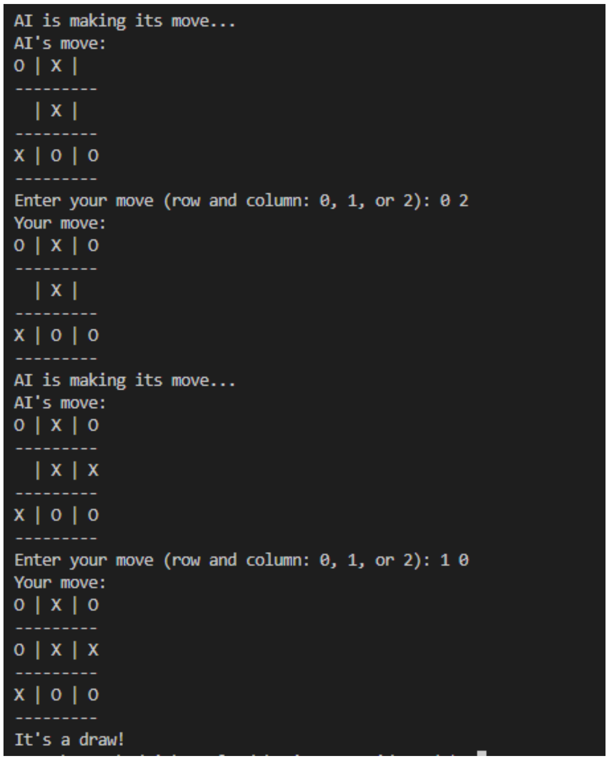
```
def minimax(board, depth, isMaximizing):
    score = evaluateBoard(board)
    if score == 10 or score == -10:
        return score
    if isDraw(board):
        return 0

    if isMaximizing:
        bestScore = -math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == "":
                    board[i][j] = 'X'
                    score = minimax(board, depth + 1, False)
                    board[i][j] = ""
                    bestScore = max(bestScore, score)
        return bestScore
    else:
        bestScore = math.inf
        for i in range(3):
            for j in range(3):
                if board[i][j] == "":
                    board[i][j] = 'O'
                    score = minimax(board, depth + 1, True)
                    board[i][j] = ""
                    bestScore = min(bestScore, score)
        return bestScore
```

```
def findBestMove(board):
    bestValue = -math.inf
    bestMove = (-1, -1)
    for i in range(3):
        for j in range(3):
            if board[i][j] == "":
                board[i][j] = 'X'
                moveValue = minimax(board, 0, False)
                board[i][j] = ""
                if moveValue > bestValue:
                    bestMove = (i, j)
                    bestValue = moveValue
    return bestMove


def playGame():
    board = [["" for _ in range(3)] for _ in range(3)]
    print("Tic Tac Toe!")
    print("You are 'O'. The AI is 'X'.")
    printBoard(board)

    while True:
        while True:
            try:
                row, col = map(int, input("Enter your move (row and column: 0, 1, or 2): ").split())
                if board[row][col] == "":
                    board[row][col] = 'O'
                    break
                else:
                    print("Cell is already taken. Choose another.")
            except (ValueError, IndexError):
                print("Invalid input. Enter row and column as two numbers between 0 and 2.")

        print("Your move:")
        printBoard(board)

        if evaluateBoard(board) == -10:
            print("You win!")
            break
        if isDraw(board):
            print("It's a draw!")
            break

        print("AI is making its move...")
        bestMove = findBestMove(board)
        board[bestMove[0]][bestMove[1]] = 'X'

        print("AI's move:")
        printBoard(board)

        if evaluateBoard(board) == 10:
            print("AI wins!")
            break
        if isDraw(board):
            print("It's a draw!")
            break

    playGame()
```

Output:



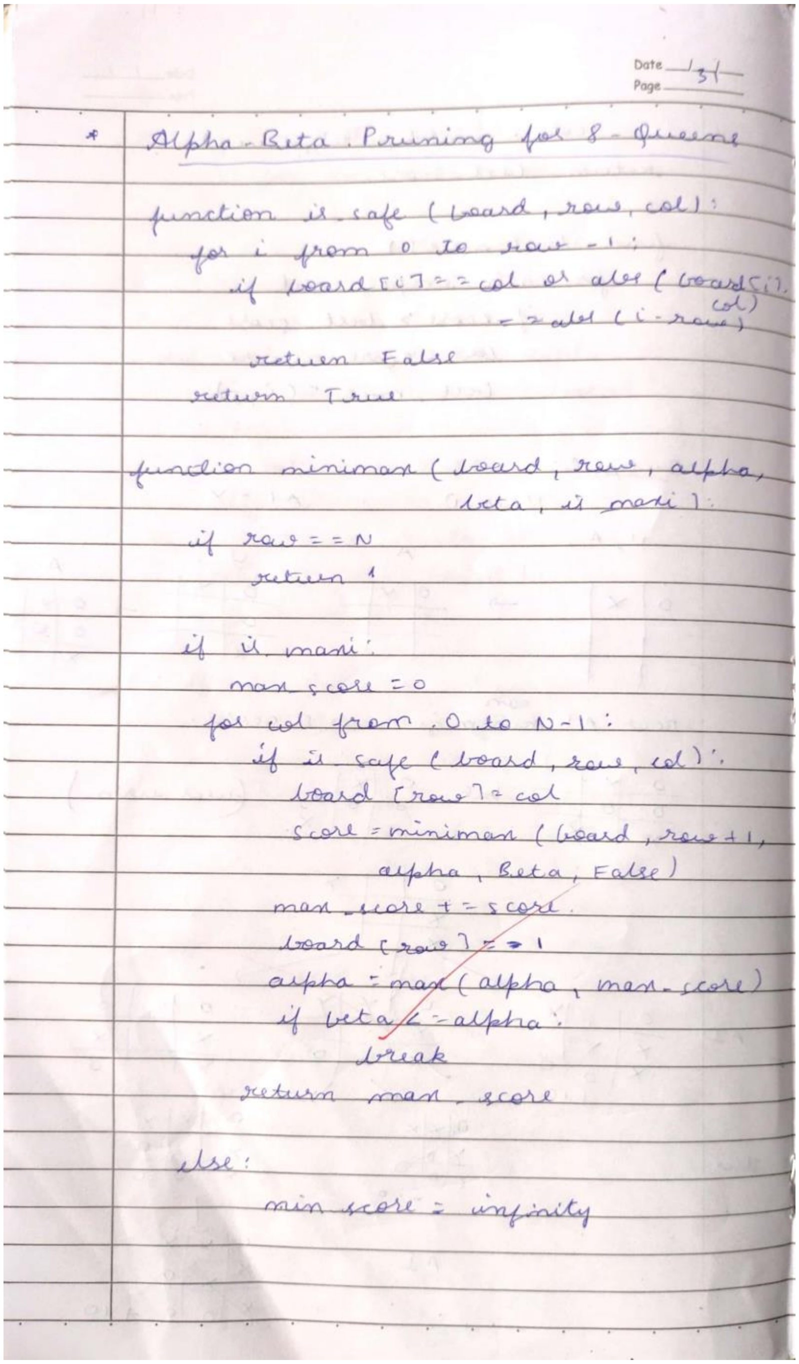
```
Tic Tac Toe!
You are 'O'. The AI is 'X'.
  |   |
---------
  |   |
---------
  |   |
---------
Enter your move (row and column: 0, 1, or 2): 2 2
Your move:
  |   |
---------
  |   |
---------
  |   | O
---------
AI is making its move...
AI's move:
  |   |
---------
  | X |
---------
  |   | O
---------
Enter your move (row and column: 0, 1, or 2): 0 0
Your move:
O |   |
---------
  | X |
---------
  |   | O
---------
AI is making its move...
AI's move:
O | X |
---------
  | X |
---------
  |   | O
---------
Enter your move (row and column: 0, 1, or 2): 2 1
Your move:
O | X |
---------
  | X |
---------
  | O | O
---------
AI is making its move...
```



```
AI is making its move...
AI's move:
O | X |
---------
  | X |
---------
X | O | O
---------
Enter your move (row and column: 0, 1, or 2): 0 2
Your move:
O | X | O
---------
  | X |
---------
X | O | O
---------
AI is making its move...
AI's move:
O | X | O
---------
  | X | X
---------
X | O | O
---------
Enter your move (row and column: 0, 1, or 2): 1 0
Your move:
O | X | O
---------
O | X | X
---------
X | O | O
---------
It's a draw!
```

Implement Alpha-Beta Pruning for 8Queens.

Algorithm:

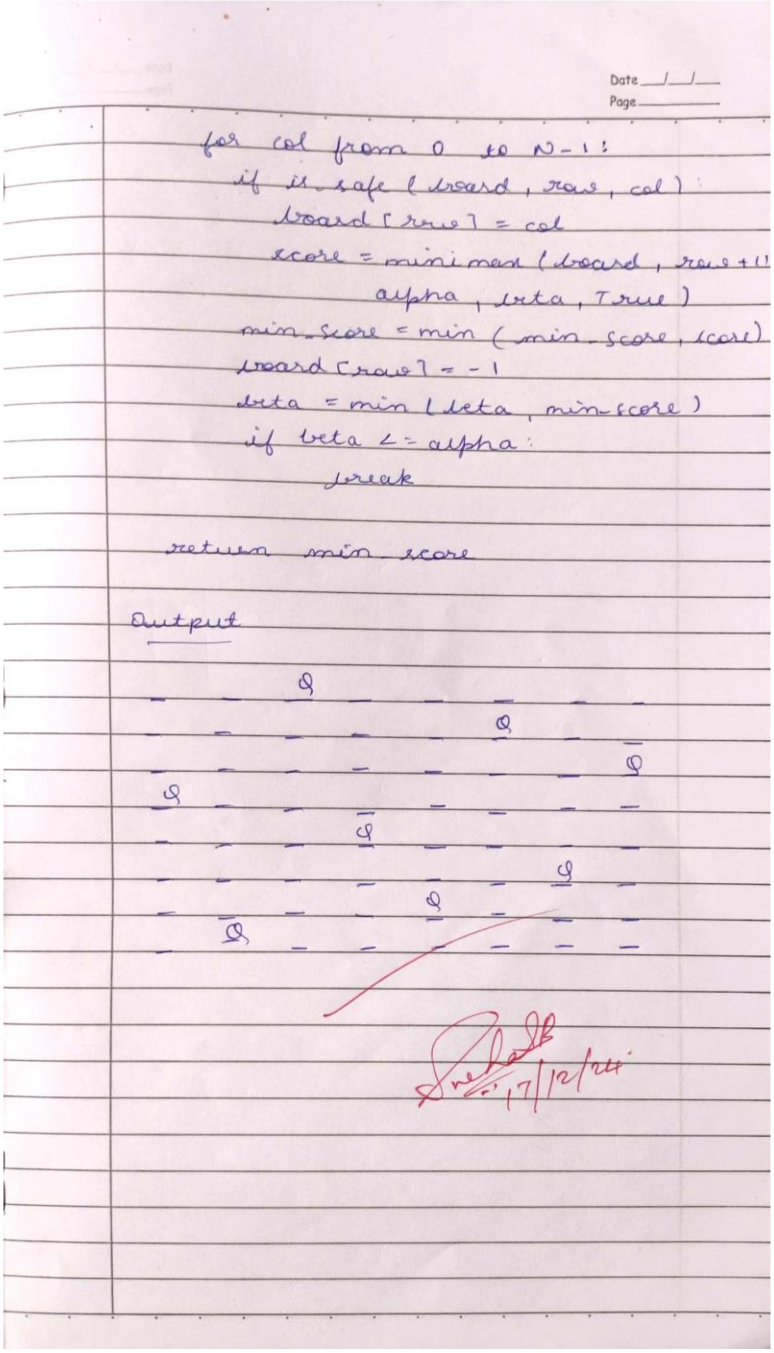


Date /3/

Page

* Alpha-Beta Pruning for 8-Queens

```
function is_safe (board, row, col):
    for i from 0 to row - 1:
        if board [i] == col or abs (board[i] - col)
                                  == abs (i - row)
            return False
    return True

function minimax (board, row, alpha,
                          beta, is_maxi):
    if row == N
        return 1

    if is_maxi:
        max_score = 0
        for col from 0 to N-1:
            if is_safe (board, row, col):
                board [row] = col
                score = minimax (board, row+1,
                            alpha, Beta, False)
                max_score += score
                board [row] = -1
                alpha = max (alpha, max_score)
                if beta <= alpha:
                    break
        return max_score

    else:
        min score = infinity
```



Date __/__/__
Page ______

```
for col from 0 to N-1:
  if is_safe (board, row, col):
     board [row] = col
     score = minimax (board, row+1,
                alpha, beta, True)
  min_score = min (min_score, score)
  board [row] = -1
  beta = min (beta, min_score)
  if beta <= alpha:
       break

 return min_score
```

Output

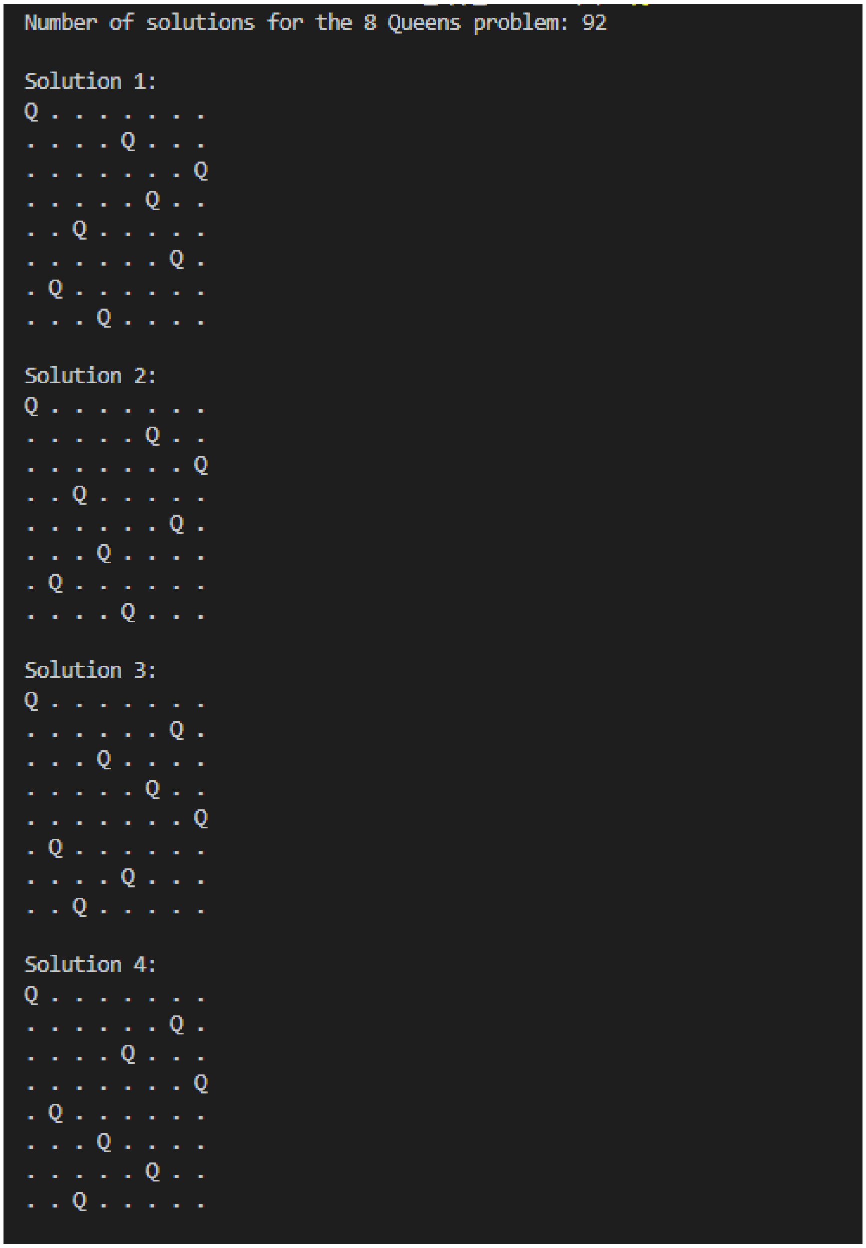
```
_ _ Q _ _ _ _ _
_ _ _ _ _ Q _ _
_ _ _ _ _ _ _ Q
Q _ _ _ _ _ _ _
_ _ _ Q _ _ _ _
_ _ _ _ _ _ Q _
_ _ _ _ Q _ _ _
_ Q _ _ _ _ _ _
```

17/12/24

Code:

```
def is_valid(board, row, col):

    for i in range(row):
        if board[i] == col or \
           abs(board[i] - col) == abs(i - row):
            return False
    return True

def alpha_beta(board, row, alpha, beta, isMaximizing):

    if row == len(board):
        return 1

    if isMaximizing:
        max_score = 0
        for col in range(len(board)):
            if is_valid(board, row, col):
                board[row] = col
                max_score += alpha_beta(board, row + 1, alpha, beta, False)
                board[row] = -1
                alpha = max(alpha, max_score)
                if beta <= alpha:
                    break
        return max_score
    else:
        min_score = float('inf')
        for col in range(len(board)):
            if is_valid(board, row, col):
                board[row] = col
                min_score = min(min_score, alpha_beta(board, row + 1, alpha, beta, True))
                board[row] = -1
                beta = min(beta, min_score)
                if beta <= alpha:
                    break
        return min_score

def solve_8_queens():

    board = [-1] * 8
    alpha = -float('inf')
    beta = float('inf')
    return alpha_beta(board, 0, alpha, beta, True)

solutions = solve_8_queens()
print(f"Number of solutions for the 8 Queens problem: {solutions}")
```

Output:



```
Number of solutions for the 8 Queens problem: 92

Solution 1:
Q . . . . . . .
. . . . Q . . .
. . . . . . . Q
. . . . . Q . .
. . Q . . . . .
. . . . . . Q .
. Q . . . . . .
. . . Q . . . .

Solution 2:
Q . . . . . . .
. . . . . Q . .
. . . . . . . Q
. . Q . . . . .
. . . . . . Q .
. . . Q . . . .
. Q . . . . . .
. . . . Q . . .

Solution 3:
Q . . . . . . .
. . . . . . Q .
. . . Q . . . .
. . . . . Q . .
. . . . . . . Q
. Q . . . . . .
. . . . Q . . .
. . Q . . . . .

Solution 4:
Q . . . . . . .
. . . . . . Q .
. . . . Q . . .
. . . . . . . Q
. Q . . . . . .
. . . Q . . . .
. . . . . Q . .
. . Q . . . . .
```