**AI INTERNAL LAB**

1. **TIC-TAC-TOE  
     
   Program:**

import random

def printBoard(board):

for row in board:

print("|".join(row))

print("-" \* 5)

def checkWin(board, player):

for row in board:

if all([cell == player for cell in row]):

return True

for col in range(3):

if all([board[row][col] == player for row in range(3)]):

return True

if all([board[i][i] == player for i in range(3)]) or all([board[i][2 - i] == player for i in range(3)]):

return True

return False

def checkDraw(board):

return all([cell != ' ' for row in board for cell in row])

def getAvailableMoves(board):

moves = []

for i in range(3):

for j in range(3):

if board[i][j] == ' ':

moves.append((i, j))

return moves

def computerMove(board):

available\_moves = getAvailableMoves(board)

return random.choice(available\_moves)

def ticTacToe():

board = [[' ' for \_ in range(3)] for \_ in range(3)]

human = 'X'

computer = 'O'

print("Welcome to Tic-Tac-Toe! You're 'X', the computer is 'O'.")

while True:

printBoard(board)

while True:

try:

row = int(input("Enter your move (row 0-2): "))

col = int(input("Enter your move (column 0-2): "))

if board[row][col] == ' ':

board[row][col] = human

break

else:

print("Invalid move, try again.")

except:

print("Invalid input, please enter row and column between 0 and 2.")

if checkWin(board, human):

printBoard(board)

print("Congratulations, you win!")

break

if checkDraw(board):

printBoard(board)

print("It's a draw!")

break

row, col = computerMove(board)

board[row][col] = computer

print(f"Computer moves to ({row}, {col})")

if checkWin(board, computer):

printBoard(board)

print("Computer wins! Better luck next time.")

break

if checkDraw(board):

printBoard(board)

print("It's a draw!")

break

ticTacToe()

**OUTPUT**

Welcome to Tic-Tac-Toe! You're 'X', the computer is 'O'.

| |

-----

| |

-----

| |

-----

Enter your move (row 0-2): 1

Enter your move (column 0-2): 1

Computer moves to (2, 1)

| |

-----

|X|

-----

|O|

-----

Enter your move (row 0-2): 2

Enter your move (column 0-2): 0

Computer moves to (1, 2)

| |

-----

|X|O

-----

X|O|

-----

Enter your move (row 0-2): 0

Enter your move (column 0-2): 2

| |X

-----

|X|O

-----

X|O|

-----

Congratulations, you win!

**2.WATER-JUG PROBLEM**

**Program:**

x=5

y=3

x1=int(input("Enter the initial amount of water in 5 liter jug : "))

y1=int(input("Enter the initial amount of water in 3 liter jug : "))

s=4

while True:

rule=int(input("Enter the rule no : "))

if(rule==1):

if x1<x:x1=x

if rule==2:

if x1>0:x1=0

if rule==3:

if y1<y:y1=y

if rule==4:

if y1>0:y1=0

if rule==5:

if x1+y1<=3 and y1>0:x1,y1=x1+y1,0

if rule==6:

if x1+y1<=5 and x1>0:x1,y1=0,x1+y1

if rule==7:

if x1+y1>=3:x1,y1=5,x1+y1-5

if rule==8:

if x1+y1>=5:x1,y1=x1+y1-3,3

print(f"X is : {x1}\n Y is : {y1}")

if x1==s:

print("The goal state is reached")

break

**OUTPUT**

Enter the initial amount of water in 5 liter jug : 0

Enter the initial amount of water in 3 liter jug : 0

Enter the rule no : 1

X is : 5

Y is : 0

Enter the rule no : 8

X is : 2

Y is : 3

Enter the rule no : 4

X is : 2

Y is : 0

Enter the rule no : 6

X is : 0

Y is : 2

Enter the rule no : 1

X is : 5

Y is : 2

Enter the rule no : 8

X is : 4

Y is : 3

The goal state is reached

**3.BFS  
  
Program**dict1={}

res=[]

queue=[]

visited=[0 for i in range(10)]

def addEdges(dict1,key,val):

if key in dict1:

dict1[key].append(val)

else:

dict1[key]=[val]

start=1

visited[start]=1

queue.append(start)

addEdges(dict1,1,2)

addEdges(dict1,2,1)

addEdges(dict1,1,3)

addEdges(dict1,3,1)

addEdges(dict1,2,9)

addEdges(dict1,9,2)

addEdges(dict1,3,4)

addEdges(dict1,4,3)

addEdges(dict1,3,5)

addEdges(dict1,5,3)

addEdges(dict1,4,6)

addEdges(dict1,6,4)

addEdges(dict1,4,8)

addEdges(dict1,8,4)

addEdges(dict1,5,7)

addEdges(dict1,7,5)

while(len(queue)!=0):

node=queue.pop(0)

res.append(node)

if node in dict1:

for neighbour in dict1[node]:

if(visited[neighbour]==0):

queue.append(neighbour)

visited[neighbour]=1

print(res)

**OUTPUT  
  
[1, 2, 3, 9, 4, 5, 6, 8, 7]**

**4.DFS  
  
Program :**visited=[0 for i in range(10)]

dict1={}

res=[]

def addEdges(dict1,key,value):

if key in dict1:

dict1[key].append(value)

else:

dict1[key]=[value]

def dfs(node):

res.append(node)

visited[node]=1

if node in dict1:

for neighbour in dict1[node]:

if(visited[neighbour]==0):

dfs(neighbour)

addEdges(dict1,1,2)

addEdges(dict1,2,1)

addEdges(dict1,1,3)

addEdges(dict1,3,1)

addEdges(dict1,2,9)

addEdges(dict1,9,2)

addEdges(dict1,3,4)

addEdges(dict1,4,3)

addEdges(dict1,3,5)

addEdges(dict1,5,3)

addEdges(dict1,4,6)

addEdges(dict1,6,4)

addEdges(dict1,4,8)

addEdges(dict1,8,4)

addEdges(dict1,5,7)

addEdges(dict1,7,5)

dfs(1)

print(res)  
  
**OUTPUT :   
  
[1, 2, 9, 3, 4, 6, 8, 5, 7]**5. **8-PUZZLE-PROBLEM  
  
Program :**def puzzle():

row = []

for i in range(3):

col = []

for j in range(3):

val = int(input("Enter the values for row {}, column {}: ".format(i+1, j+1)))

col.append(val)

row.append(col)

return row

def checkGoal(initial\_state, final\_state):

return initial\_state == final\_state

def findBlank(initial\_state):

for i in range(3):

for j in range(3):

if initial\_state[i][j] == 0:

return i, j

def checkMoves(i, j):

moves = []

# Up

if i > 0:

moves.append((i - 1, j))

# Down

if i < 2: # Since it's a 3x3 board, index 2 is the last row

moves.append((i + 1, j))

# Left

if j > 0:

moves.append((i, j - 1))

# Right

if j < 2:

moves.append((i, j + 1))

return moves

def makeMove(initial\_state, blank\_pos, move):

new\_state = [row[:] for row in initial\_state]

i, j = blank\_pos

new\_i, new\_j = move

new\_state[i][j], new\_state[new\_i][new\_j] = new\_state[new\_i][new\_j], new\_state[i][j]

return new\_state

from collections import deque

def solvePuzzle(initial\_state, final\_state):

queue = deque([(initial\_state, findBlank(initial\_state), [])])

visited = set()

while queue:

current\_state, blank\_pos, path = queue.popleft()

if checkGoal(current\_state, final\_state):

return path

visited.add(tuple(tuple(row) for row in current\_state))

for move in checkMoves(\*blank\_pos):

new\_state = makeMove(current\_state, blank\_pos, move)

if tuple(tuple(row) for row in new\_state) not in visited:

queue.append((new\_state, move, path + [move]))

return None

final\_state = [[1, 2, 3], [4, 0, 5], [6, 7, 8]]

initial\_state = puzzle()

solution = solvePuzzle(initial\_state, final\_state)

if solution:

print("Solution found:", solution)

else:

print("No solution found.")

**OUTPUT :**Enter the values for row 1, column 1: 1

Enter the values for row 1, column 2: 2

Enter the values for row 1, column 3: 3

Enter the values for row 2, column 1: 4

Enter the values for row 2, column 2: 5

Enter the values for row 2, column 3: 6

Enter the values for row 3, column 1: 7

Enter the values for row 3, column 2: 8

Enter the values for row 3, column 3: 0

Solution found: [(2, 1), (2, 0), (1, 0), (1, 1), (1, 2), (2, 2), (2, 1), (1, 1), (1, 0), (2, 0), (2, 1), (2, 2), (1, 2), (1, 1)]

6.**TOWERS OF HANOI  
  
Program :**n=3

def hanoi(n,from1,to,aux):

if n==0:

return -1

hanoi(n-1,from1,aux,to)

print("Move to disk",n,"from rod",from1,"to rod",to,"using Auxiliary rod",aux)

hanoi(n-1,aux,to,from1)

hanoi(n,'A','B','C')  
  
**OUTPUT :**  
Move to disk 1 from rod A to rod B using Auxiliary rod C

Move to disk 2 from rod A to rod C using Auxiliary rod B

Move to disk 1 from rod B to rod C using Auxiliary rod A

Move to disk 3 from rod A to rod B using Auxiliary rod C

Move to disk 1 from rod C to rod A using Auxiliary rod B

Move to disk 2 from rod C to rod B using Auxiliary rod A

Move to disk 1 from rod A to rod B using Auxiliary rod C

**7**. **MISSIONARIES AND CANNIBALS  
  
Program :**nlm = 3 # Number of missionaries on the left side

nlc = 3 # Number of cannibals on the left side

nrm = 0 # Number of missionaries on the right side

nrc = 0 # Number of cannibals on the right side

direction = "left"

print("Left to Right : ")

print(nlm, "M", "|", nlc, "C", "---->", nrm, "M", "|", nrc, "C")

k=0

def checkSafe(lm, rm, lc, rc):

# Check if the number of missionaries or cannibals is negative

if lm < 0 or lc < 0 or rm < 0 or rc < 0:

return False

# Check if cannibals outnumber missionaries on either side

if (lm > 0 and lc > lm) or (rm > 0 and rc > rm):

return False

return True

while True:

# Left side to right side

if direction == "left":

um = int(input("Enter no of missionaries to travel from left to right: "))

uc = int(input("Enter no of cannibals to travel from left to right: "))

if um + uc > 2 or um + uc == 0 or um < 0 or uc < 0:

print("Invalid input")

continue

# Update the numbers after the move

new\_nlm = nlm - um

new\_nlc = nlc - uc

new\_nrm = nrm + um

new\_nrc = nrc + uc

if not checkSafe(new\_nlm, new\_nrm, new\_nlc, new\_nrc):

print("Move is not safe, try again.")

continue

# Apply the changes if the move is safe

nlm, nlc, nrm, nrc = new\_nlm, new\_nlc, new\_nrm, new\_nrc

k+=1

print("Left Side to Right Side:")

print(nlm, "M", "|", nlc, "C", "---->", nrm, "M", "|", nrc, "C")

direction = "right"

# Right side to left side

else:

um = int(input("Enter no of missionaries to travel from right to left: "))

uc = int(input("Enter no of cannibals to travel from right to left: "))

if um + uc > 2 or um + uc == 0 or um < 0 or uc < 0:

print("Invalid input")

continue

# Update the numbers after the move

new\_nlm = nlm + um

new\_nlc = nlc + uc

new\_nrm = nrm - um

new\_nrc = nrc - uc

if not checkSafe(new\_nlm, new\_nrm, new\_nlc, new\_nrc):

print("Move is not safe, try again.")

continue

# Apply the changes if the move is safe

nlm, nlc, nrm, nrc = new\_nlm, new\_nlc, new\_nrm, new\_nrc

k+=1

print("Right Side to Left Side:")

print(nlm, "M", "|", nlc, "C", "---->", nrm, "M", "|", nrc, "C")

direction = "left"

# Check if goal state is reached

if nrm == 3 and nrc == 3:

print("Goal State is reached")

print(nlm, "M", "|", nlc, "C", "---->", nrm, "M", "|", nrc, "C")

print("No of Steps : ",k)

break

**OUTPUT :**Left to Right :

3 M | 3 C ----> 0 M | 0 C

Enter no of missionaries to travel from left to right: 1

Enter no of cannibals to travel from left to right: 1

Left Side to Right Side:

2 M | 2 C ----> 1 M | 1 C

Enter no of missionaries to travel from right to left: 1

Enter no of cannibals to travel from right to left: 0

Right Side to Left Side:

3 M | 2 C ----> 0 M | 1 C

Enter no of missionaries to travel from left to right: 0

Enter no of cannibals to travel from left to right: 2

Left Side to Right Side:

3 M | 0 C ----> 0 M | 3 C

Enter no of missionaries to travel from right to left: 0

Enter no of cannibals to travel from right to left: 1

Right Side to Left Side:

3 M | 1 C ----> 0 M | 2 C

Enter no of missionaries to travel from left to right: 2

Enter no of cannibals to travel from left to right: 0

Left Side to Right Side:

1 M | 1 C ----> 2 M | 2 C

Enter no of missionaries to travel from right to left: 1

Enter no of cannibals to travel from right to left: 1

Right Side to Left Side:

2 M | 2 C ----> 1 M | 1 C

Enter no of missionaries to travel from left to right: 2

Enter no of cannibals to travel from left to right: 0

Left Side to Right Side:

0 M | 2 C ----> 3 M | 1 C

Enter no of missionaries to travel from right to left: 0

Enter no of cannibals to travel from right to left: 1

Right Side to Left Side:

0 M | 3 C ----> 3 M | 0 C

Enter no of missionaries to travel from left to right: 0

Enter no of cannibals to travel from left to right: 2

Left Side to Right Side:

0 M | 1 C ----> 3 M | 2 C

Enter no of missionaries to travel from right to left: 0

Enter no of cannibals to travel from right to left: 1

Right Side to Left Side:

0 M | 2 C ----> 3 M | 1 C

Enter no of missionaries to travel from left to right: 0

Enter no of cannibals to travel from left to right: 2

Left Side to Right Side:

0 M | 0 C ----> 3 M | 3 C

Goal State is reached

0 M | 0 C ----> 3 M | 3 C

No of Steps : 11  
  
  
  
8.**TRAVELLING SALESMAN PROBLEM   
  
Program :**from itertools import permutations

def calculateDistance(graph, route):

total\_distance = 0

num\_cities = len(route)

for i in range(num\_cities - 1):

total\_distance += graph[route[i]][route[i + 1]]

total\_distance += graph[route[-1]][route[0]]

return total\_distance

def travelingSalesman(graph):

num\_cities = len(graph)

cities = [i for i in range(num\_cities)]

all\_routes = permutations(cities)

min\_cost = float('inf')

best\_route = None

for route in all\_routes:

current\_cost = calculateDistance(graph, route)

if current\_cost < min\_cost:

min\_cost = current\_cost

best\_route = route

return min\_cost, best\_route

graph = [

[0, 10, 15, 20],

[10, 0, 35, 25],

[15, 35, 0, 30],

[20, 25, 30, 0]

]

min\_cost, best\_route = travelingSalesman(graph)

print("Minimum cost:", min\_cost)

print("Best route:", best\_route)

**OUTPUT**Minimum cost: 80

Best route: (0, 1, 3, 2) #A-0,B-1,C-2,D-3