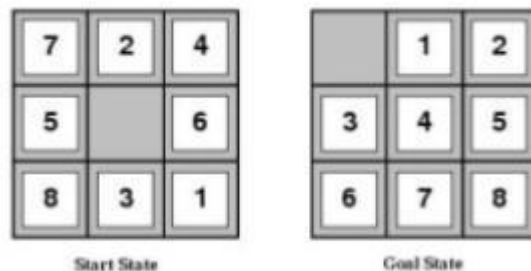## PRELAB

1. Define Cost Function and idea cost function in 8- puzzle problem.
   Take the below picture as an example and calculate the heuristic functions for it (using the most two commonly used techniques).

Start State   Goal State

Cost Function:
Each node X in the search tree is associated with a cost. The cost function is useful for determining the next E-node. The next E-node is the one with the least cost. The cost function is defined as

$$C(x) = h(x)+g(x)$$
where $g(x)$ is the cost of reaching the current node from root
$h(x)$ is the cost of reaching an answer node from x.

Ideal Cost function for 8-puzzle Algorithm:
We assume that moving one tile in any direction will have 1 unit cost. Keeping that in mind, we define a cost function for the 8-puzzle algorithm as below: $C(x) = f(x)+h(x)$
$f(x)$ is the length of the path from root to x (the number of moves so far) and $h(x)$ is the number of nonblank tiles not in their goal position (the number of misplaced tiles).There are at least $h(x)$ moves to transform state x to a goal state. $H(n)$ is the heuristic function
$H1(n)$ is number of misplaced tiles
$H2(n)$ is total Manhattan Distance
$H(n) = H1(n)+H2(n)$
Above problem
$H1(n) = 8$
$H2(n) = 3+1+2+2+2+3+3+2=18$
$H(n)=H1(n)+H2(n)$
$H(n)=8+18=26$
It takes 26 steps to sovle the 8 puzzle problem.

2. Consider a game which has 4 final states and paths to reach the final state are from root to 4 leaves of a perfect binary tree as shown below. Assume you are the maximizing player and you get the first chance to move, i.e., you are at the root and your opponent at the next level. Which move you would make as a maximizing player considering that your opponent also plays optimally?
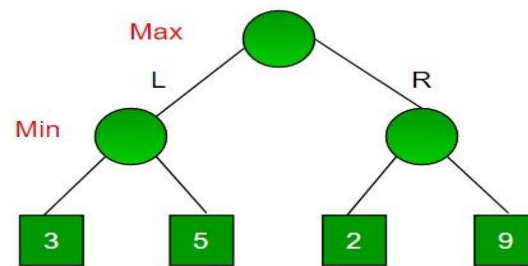
```python
import math
def minimax (curDepth, nodeIndex,maxTurn, scores,targetDepth):
    # base case : targetDepth reached
    if (curDepth == targetDepth):
     return scores[nodeIndex]
    if (maxTurn):
     return max(minimax(curDepth + 1,nodeIndex*2,False,scores,targetDepth),minimax(curDepth + 1, nodeIndex * 2 + 1,False, scores, targetDepth))
    else:
     return min(minimax(curDepth + 1,nodeIndex * 2,True, scores, targetDepth),minimax(curDepth + 1, nodeIndex * 2 + 1,True, scores, targetDepth))

# Driver code
scores = [3, 5, 2, 9, 12, 5, 23, 23]
treeDepth = math.log(len(scores),2)
print("The optimal value is : ", end = "")
print(minimax(0, 0, True, scores, treeDepth))
```

INLAB

1. Implementation of Tic-Tac-Toe game
Rules of the Game

● The game is to be played between two people (in this program between HUMAN and COMPUTER).
● One of the players chooses 'O' and the other 'X' to mark their respective cells.
● The game starts with one of the players and the game ends when one of the players has one whole row/ column/ diagonal filled with his/her respective character ('O' or 'X').
If no one wins, then the game is said to be drawn.

Implementation
In our program the moves taken by the computer and the human are chosen randomly.
We use the rand() function for this.
What more can be done in the program?
The program is not played optimally by both sides because the moves are chosen randomly.
The program can be easily modified so that both players play optimally (which will fall under the category of Artificial Intelligence). Also the program can be modified such that the user himself gives the input (using scanf() or cin).
The above changes are left as an exercise to the readers.
Winning Strategy – An Interesting Fact
If both the players play optimally then it is destined that you will never lose ("although the match can still be drawn"). It doesn't matter whether you play first or second. In other ways – " Two expert players will always draw ". isn't this interesting

```python
import numpy as np
import random
from time import sleep
# Creates an empty board
def create_board():
  return(np.array([[0, 0, 0],[0, 0, 0],[0, 0, 0]]))
# Check for empty places on board
def possibilities(board):
  l = []
  for i in range(len(board)):
    for j in range(len(board)):
      if board[i][j] == 0:
        l.append((i, j))
        return(l)


# Select a random place for the player
```

```python
def random_place(board, player):
 selection = possibilities(board)
 current_loc = random.choice(selection)
 board[current_loc] = player
 return(board)

# Checks whether the player has three # of their marks in a horizontal row
def row_win(board, player):
 for x in range(len(board)):
  win = True
 for y in range(len(board)):
  if board[x, y] != player:
   win = False
   continue
  if win == True:
   return(win)
 return(win)

# Checks whether the player has three # of their marks in a vertical row
def col_win(board, player):
 for x in range(len(board)):
  win = True
  for y in range(len(board)):
   if board[y][x] != player:
    win = False
    continue
  if win == True:
   return(win)
 return(win)

# Checks whether the player has three # of their marks in a diagonal row
def diag_win(board, player):
 win = True
 y = 0
 for x in range(len(board)):
  if board[x, x] != player:
   win = False
 if win:
  return win
 win = True
 if win:
  for x in range(len(board)):
   y = len(board) - 1 - x
   if board[x, y] != player:
    win = False
 return win
```

```python
# Evaluates whether there is # a winner or a tie
def evaluate(board):
  winner = 0
  for player in [1, 2]:
    if (row_win(board, player) or col_win(board,player) or diag_win(board,player)):
      winner = player
  if np.all(board != 0) and winner == 0:
    winner = -1
  return winner
# Main function to start the game
def play_game():
  board, winner, counter = create_board(), 0, 1
  print(board)
  sleep(2)

  while winner == 0:
    for player in [1, 2]:
      board = random_place(board, player)
      print("Board after " + str(counter) + " move")
      print(board)
      sleep(2)
      counter += 1
      winner = evaluate(board)
      if winner != 0:
        break
  return(winner)


# Driver Code
print("Winner is: " + str(play_game()))
```

OUTPUT

POSTLAB

1.Sudoku | Backtracking-7

Given a partially filled 9×9 2D array 'grid[9][9]', the goal is to assign digits (from 1 to 9) to the empty cells so that every row, column, and subgrid of size 3×3 contains exactly one instance of the digits from 1 to 9.

3		6	5		8	4		
5	2							
	8	7					3	1
		3		1			8	
9			8	6	3			5
	5			9		6		
1	3					2	5	
							7	4
		5	2		6	3		

Example:
Input:
grid = {{3, 0, 6, 5, 0, 8, 4, 0, 0},
        {5, 2, 0, 0, 0, 0, 0, 0, 0},
        {0, 8, 7, 0, 0, 0, 0, 3, 1},
        {0, 0, 3, 0, 1, 0, 0, 8, 0},
        {9, 0, 0, 8, 6, 3, 0, 0, 5},
        {0, 5, 0, 0, 9, 0, 6, 0, 0},
        {1, 3, 0, 0, 0, 0, 2, 5, 0},
        {0, 0, 0, 0, 0, 0, 0, 7, 4},
        {0, 0, 5, 2, 0, 6, 3, 0, 0}}
Output:
        3 1 6 5 7 8 4 9 2
        5 2 9 1 3 4 7 6 8
        4 8 7 6 2 9 5 3 1
        2 6 3 4 1 5 9 8 7
        9 7 4 8 6 3 1 2 5
        8 5 1 7 9 2 6 4 3
        1 3 8 9 4 7 2 5 6
        6 9 2 3 5 1 8 7 4
        7 4 5 2 8 6 3 1 9
Explanation: Each row, column and 3*3 box of the output matrix contains unique numbers.

Method : Backtracking.
Approach:
Like all other Backtracking problems, Sudoku can be solved by one by one assigning numbers to empty cells. Before assigning a number, check whether it is safe to assign. Check that the same number is not present in the current row, current column and current 3X3 subgrid. After checking for safety, assign the number, and recursively check whether this assignment leads to a solution or not. If the assignment doesn't lead to a solution, then try the next number for the current empty cell. And if none of the numbar (1 to 9) leads to a solution, return false and print no solution exists.

```python
SIZE=9
matrix = [
 [6,5,0,8,7,3,0,9,0],
 [0,0,3,2,5,0,0,0,8],
 [9,8,0,1,0,4,3,5,7],
 [1,0,5,0,0,0,0,0,0],
 [4,0,0,0,0,0,0,0,2],
 [0,0,0,0,0,0,5,0,3],
 [5,7,8,3,0,1,0,2,6],
 [2,0,0,0,4,8,9,0,0],
 [0,9,0,6,2,5,0,8,1]]

#function to print sudoku
def print_sudoku():
  for i in matrix:
    print (i)

#function to check if all cells are assigned or not
#if there is any unassigned cell
#then this function will change the values of
#row and col accordingly
def number_unassigned(row, col):
  num_unassign = 0
  for i in range(0,SIZE):
    for j in range (0,SIZE):
      #cell is unassigned
      if matrix[i][j] == 0:
        row = i
        col = j
        num_unassign = 1
        a = [row, col, num_unassign]
        return a
  a = [-1, -1, num_unassign]
  return a
#function to check if we can put a #value in a paticular cell or not
def is_safe(n, r, c):
  #checking in row
  for i in range(0,SIZE):
    #there is a cell with same value
    if matrix[r][i] == n:
      return False
    #checking in column
  for i in range(0,SIZE):
    #there is a cell with same value
    if matrix[i][c] == n:
      return False
  row_start = (r//3)*3
```
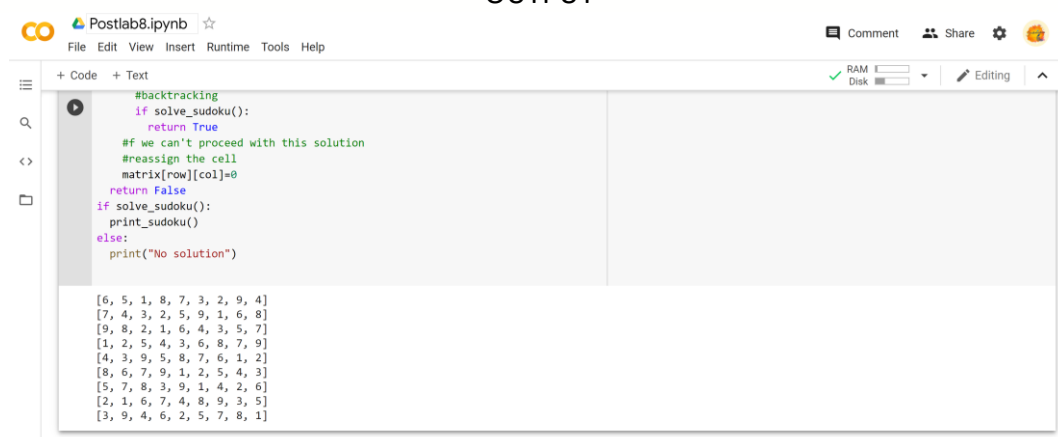
```python
  col_start = (c//3)*3;
   #checking submatrix
  for i in range(row_start,row_start+3):
   for j in range(col_start,col_start+3):
    if matrix[i][j]==n:
      return False
  return True
#function to check if we can put a #value in a paticular cell or not
def solve_sudoku():
 row = 0
 col = 0
 #if all cells are assigned then the sudoku is already solved  #pass by reference because number_unassigned will change
 # the values of row and col
 a = number_unassigned(row, col)
 if a[2] == 0:
   return True
 row = a[0]
 col = a[1]  #number between 1 to 9
 for i in range(1,10):
 #if we can assign i to the cell or not   #the cell is
   matrix[row][col]
   if is_safe(i, row, col):
    matrix[row][col] = i
    #backtracking
    if solve_sudoku():
      return True
   #f we can't proceed with this solution
   #reassign the cell
   matrix[row][col]=0
 return False
if solve_sudoku():
 print_sudoku()
else:
 print("No solution")
```

                              OUTPUT

190031187   **NERELLA VENKATA RADHAKRISHNA**