UNIVERSAL

VIKAS VIDYA EDUCATION TRUST'S

Lords Universal College

Department of Information Technology

CERTIFICATE

This is to certify that Mr./Ms. ____ of ____________________ Uni. Exam No. ______________ (____ Semester) has satisfactorily completed Practical, in the subject of ____ as a part of B.Sc. in Information Technology Program during the academic year 20______-20______.

Place:

Date:

Subject In-charge

Co-Ordinator,
Department of Information Technology

Signature of External Examiner

INDEX

Practical No. 1A
Aim: Implement depth first search algorithm

Code:

```
graph = {
 '5': ['3','7'], '3': ['2', '4'], '7': ['8'], '2': [],  '4': ['8'],  '8': []
}
visited = set()
def dfs(visited, graph, node):
   if node not in visited:
      print (node)
      visited.add(node)
      for neighbour in graph[node]:
         dfs(visited, graph, neighbour)
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

Output:

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    JUPYTER

\1a.py'
Following is the Depth-First Search
5
3
2
4
8
7
```

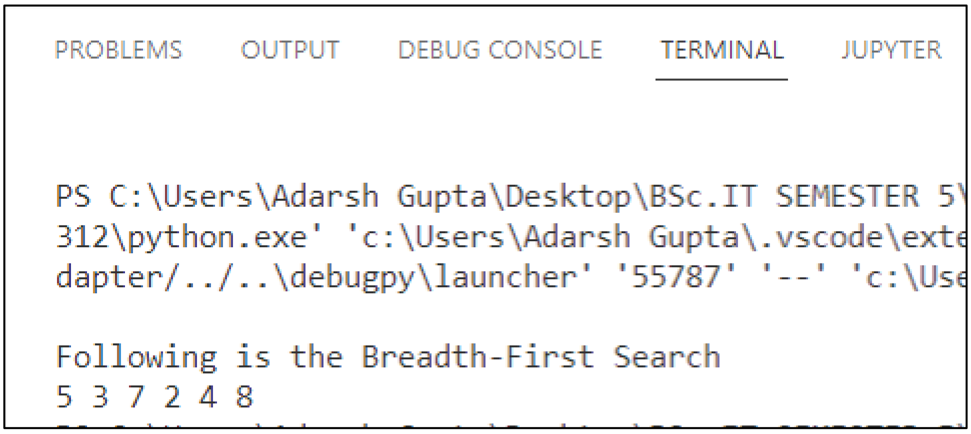
Practical No. 1B

Aim: Implement breadth first search algorithm

Code:

```
graph = {
  '5' : ['3','7'],'3' : ['2', '4'],'7' : ['8'],'2' : [],'4' : ['8'], '8' : []
}
visited = []
queue = []
def bfs(visited, graph, node):
  visited.append(node)
  queue.append(node)
  while queue:
    m = queue.pop(0)
    print (m, end = " ")
    for neighbour in graph[m]:
        if neighbour not in visited:
            visited.append(neighbour)
            queue.append(neighbour)
print("Following is the Breadth-First Search")
bfs(visited, graph, '5')
```

Output:



Practical No. 2A

Aim: Simulate 4-Queen / N-Queen problem.

Code:

```
global N
N = 4

def printSolution(board):
   for i in range(N):
      for j in range(N):
         if board[i][j] == 1:
            print("Q",end=" ")
         else:
            print(".",end=" ")
      print()

def isSafe(board, row, col):
   for i in range(col):
      if board[row][i] == 1:
         return False
   for i, j in zip(range(row, -1, -1),
               range(col, -1, -1)):
      if board[i][j] == 1:
         return False
   for i, j in zip(range(row, N, 1),
               range(col, -1, -1)):
      if board[i][j] == 1:
         return False
   return True
```

```
def solveNQUtil(board, col):
    if col >= N:
        return True;
    for i in range(N):
        if isSafe(board, i, col):
            board[i][col] = 1
            if solveNQUtil(board, col + 1) == True:
                return True
            board[i][col] = 0
    return False

def solveNQ():
    board = [[0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0],
             [0, 0, 0, 0]]

    if solveNQUtil(board, 0) == False:
        print("Solution does not exist")
        return False
    printSolution(board)
    return True

if __name__ == '__main__':
    solveNQ()
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6
AMD64)] on win32
Type "help", "copyright", "credits" or "lic

================ RESTART: C:/Users/Adarsh G
. . Q .
Q . . .
. . . Q
. Q . .
```

Practical No. 2B
Aim: Solve tower of Hanoi problem.

Code:

```
def TowerOfHanoi(n, from_rod, to_rod, aux_rod):
    if n == 0:
        return
    TowerOfHanoi(n-1, from_rod, aux_rod, to_rod)
    print("Move disk", n, "from rod", from_rod, "to rod", to_rod)
    TowerOfHanoi(n-1, aux_rod, to_rod, from_rod)


N = 4


TowerOfHanoi(N, 'A', 'C', 'B')
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16)
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more

= RESTART: C:/Users/Adarsh Gupta/Desktop/hanoi.py
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 3 from rod A to rod B
Move disk 1 from rod C to rod A
Move disk 2 from rod C to rod B
Move disk 1 from rod A to rod B
Move disk 4 from rod A to rod C
Move disk 1 from rod B to rod C
Move disk 2 from rod B to rod A
Move disk 1 from rod C to rod A
Move disk 3 from rod B to rod C
Move disk 1 from rod A to rod B
Move disk 2 from rod A to rod C
Move disk 1 from rod B to rod C
```

Practical No. 3A
Aim: Implement alpha beta search.

Code:

```
class Node:
    def __init__(self, name, children=None, value=None):
        self.name = name
        self.children = children if children is not None else []
        self.value = value

def evaluate(node):
    return node.value

def is_terminal(node):
    return node.value is not None

def get_children(node):
    return node.children

def alpha_beta_pruning(node, depth, alpha, beta, maximizing_player):
    if depth == 0 or is_terminal(node):
        return evaluate(node)

    if maximizing_player:
        max_eval = float('-inf')
        for child in get_children(node):
            eval = alpha_beta_pruning(child, depth-1, alpha, beta, False)
            max_eval = max(max_eval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha:
```

```
                break
        return max_eval
    else:
        min_eval = float('inf')
        for child in get_children(node):
            eval = alpha_beta_pruning(child, depth-1, alpha, beta, True)
            min_eval = min(min_eval, eval)
            beta = min(beta, eval)
            if beta <= alpha:
                break
        return min_eval

D = Node('D', value=3)
E = Node('E', value=5)
F = Node('F', value=6)
G = Node('G', value=9)
H = Node('H', value=1)
I = Node('I', value=2)

B = Node('B', children=[D, E, F])
C = Node('C', children=[G, H, I])
A = Node('A', children=[B, C])

maximizing_player = True
initial_alpha = float('-inf')
initial_beta = float('inf')
depth = 3
optimal_value = alpha_beta_pruning(A, depth, initial_alpha, initial_beta, maximizing_player)
print(f"The optimal value is: {optimal_value}")
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" fo

= RESTART: C:/Users/Adarsh Gupta/Desktop/hanoi.py
The optimal value is: 3
```

Practical No. 3B

Aim: Implement hill climbing problem.

Code:

```
import random

import numpy as np


coordinate = np.array([[1,2], [30,21], [56,23], [8,18], [20,50], [3,4], [11,6], [6,7], [15,20],
[10,9], [12,12]])


def generate_matrix(coordinate):
    matrix = []
    for i in range(len(coordinate)):
        for j in range(len(coordinate)) :
            p = np.linalg.norm(coordinate[i] - coordinate[j])
            matrix.append(p)
    matrix = np.reshape(matrix, (len(coordinate),len(coordinate)))
    return matrix


def solution(matrix):
    points = list(range(0, len(matrix)))
    solution = []
    for i in range(0, len(matrix)):
        random_point = points[random.randint(0, len(points) - 1)]
        solution.append(random_point)
        points.remove(random_point)
    return solution


def path_length(matrix, solution):
    cycle_length = 0
    for i in range(0, len(solution)):
```

```
        cycle_length += matrix[solution[i]][solution[i - 1]]
    return cycle_length

def neighbors(matrix, solution):
    neighbors = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbor = solution.copy()
            neighbor[i] = solution[j]
            neighbor[j] = solution[i]
            neighbors.append(neighbor)

    best_neighbor = neighbors[0]
    best_path = path_length(matrix, best_neighbor)

    for neighbor in neighbors:
        current_path = path_length(matrix, neighbor)
        if current_path < best_path:
            best_path = current_path
            best_neighbor = neighbor
    return best_neighbor, best_path



def hill_climbing(coordinate):
    matrix = generate_matrix(coordinate)

    current_solution = solution(matrix)
    current_path = path_length(matrix, current_solution)
    neighbor = neighbors(matrix,current_solution)[0]
```

```
    best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)

    while best_neighbor_path < current_path:
        current_solution = best_neighbor
        current_path = best_neighbor_path
        neighbor = neighbors(matrix, current_solution)[0]
        best_neighbor, best_neighbor_path = neighbors(matrix, neighbor)

    return current_path, current_solution
final_solution = hill_climbing(coordinate)
print("The solution is \n", final_solution[1])
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" fo

= RESTART: C:/Users/Adarsh Gupta/Desktop/hanoi.py
The solution is
 [4, 2, 1, 8, 10, 6, 0, 5, 7, 9, 3]
```

Practical No. 4A

Aim: Implement A* algorithm.

Code:

```
import math

import heapq


class Cell:

   def __init__(self):

      self.parent_i = 0

      self.parent_j = 0

      self.f = float('inf')

      self.g = float('inf')

      self.h = 0

ROW = 9

COL = 10


def is_valid(row, col):

   return (row >= 0) and (row < ROW) and (col >= 0) and (col < COL)

def is_unblocked(grid, row, col):

   return grid[row][col] == 1

def is_destination(row, col, dest):

   return row == dest[0] and col == dest[1]

def calculate_h_value(row, col, dest):

   return ((row - dest[0]) ** 2 + (col - dest[1]) ** 2) ** 0.5


def trace_path(cell_details, dest):

   print("The Path is ")

   path = []

   row = dest[0]

   col = dest[1]
```

```
    while not (cell_details[row][col].parent_i == row and cell_details[row][col].parent_j ==
col):
        path.append((row, col))
        temp_row = cell_details[row][col].parent_i
        temp_col = cell_details[row][col].parent_j
        row = temp_row
        col = temp_col
    path.append((row, col))
    path.reverse()

    for i in path:
        print("->", i, end=" ")
    print()

def a_star_search(grid, src, dest):
    if not is_valid(src[0], src[1]) or not is_valid(dest[0], dest[1]):
        print("Source or destination is invalid")
        return
    if not is_unblocked(grid, src[0], src[1]) or not is_unblocked(grid, dest[0], dest[1]):
        print("Source or the destination is blocked")
        return
    if is_destination(src[0], src[1], dest):
        print("We are already at the destination")
        return

    closed_list = [[False for _ in range(COL)] for _ in range(ROW)]
    cell_details = [[Cell() for _ in range(COL)] for _ in range(ROW)]

    i = src[0]
    j = src[1]
    cell_details[i][j].f = 0
```

```
    cell_details[i][j].g = 0
    cell_details[i][j].h = 0
    cell_details[i][j].parent_i = i
    cell_details[i][j].parent_j = j

    open_list = []
    heapq.heappush(open_list, (0.0, i, j))
    found_dest = False

    while len(open_list) > 0:
        p = heapq.heappop(open_list)
        i = p[1]
        j = p[2]
        closed_list[i][j] = True
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0),
                      (1, 1), (1, -1), (-1, 1), (-1, -1)]
        for dir in directions:
            new_i = i + dir[0]
            new_j = j + dir[1]

            if is_valid(new_i, new_j) and is_unblocked(grid, new_i, new_j) and not
closed_list[new_i][new_j]:
                if is_destination(new_i, new_j, dest):
                    cell_details[new_i][new_j].parent_i = i
                    cell_details[new_i][new_j].parent_j = j
                    print("The destination cell is found")
                    trace_path(cell_details, dest)
                    found_dest = True
                    return
                else:
                    g_new = cell_details[i][j].g + 1.0
```

```
                h_new = calculate_h_value(new_i, new_j, dest)
                f_new = g_new + h_new

                if cell_details[new_i][new_j].f == float('inf') or cell_details[new_i][new_j].f >
f_new:
                    heapq.heappush(open_list, (f_new, new_i, new_j))
                    cell_details[new_i][new_j].f = f_new
                    cell_details[new_i][new_j].g = g_new
                    cell_details[new_i][new_j].h = h_new
                    cell_details[new_i][new_j].parent_i = i
                    cell_details[new_i][new_j].parent_j = j

    if not found_dest:
        print("Failed to find the destination cell")

def main():
    grid = [
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 1],
        [1, 1, 1, 0, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 0, 0, 0, 0, 1],
        [1, 1, 1, 0, 1, 1, 1, 0, 1, 0],
        [1, 0, 1, 1, 1, 1, 0, 1, 0, 0],
        [1, 0, 0, 0, 0, 1, 0, 0, 0, 1],
        [1, 0, 1, 1, 1, 1, 0, 1, 1, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]
    ]

    src = [8, 0]
    dest = [0, 0]
    a_star_search(grid, src, dest)
```

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

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [MSC v.1940 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.

= RESTART: C:/Users/Adarsh Gupta/Desktop/hanoi.py
The destination cell is found
The Path is
-> (8, 0) -> (7, 0) -> (6, 0) -> (5, 0) -> (4, 1) -> (3, 2) -> (2, 1) -> (1, 0)
-> (0, 0)
```

Practical No. 4B
Aim: Solve water jug problem

Code:

```
from collections import defaultdict
jug1, jug2, aim = 4, 3, 2
visited = defaultdict(lambda: False)

def waterJugSolver(amt1, amt2):
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
        print(amt1, amt2)
        return True
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)
        visited[(amt1, amt2)] = True
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))
    else:
        return False

print("Steps: ")

waterJugSolver(0, 0)
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for mor

= RESTART: C:/Users/Adarsh Gupta/Desktop/water jug 4b.py
Steps:
0 0
4 0
4 3
0 3
3 0
3 3
4 2
0 2
```

Practical No. 5A

Aim: Simulate tic – tac – toe game using min-max algorithm.

Code:

```
player, opponent = 'x', 'o'
def isMovesLeft(board) :
        for i in range(3) :
                for j in range(3) :
                        if (board[i][j] == '_') :
                                return True
        return False
def evaluate(b) :
        for row in range(3) :
                if (b[row][0] == b[row][1] and b[row][1] == b[row][2]) :
                        if (b[row][0] == player) :
                                return 10
                        elif (b[row][0] == opponent) :
                                return -10
        for col in range(3) :
                        if (b[0][col] == b[1][col] and b[1][col] == b[2][col]) :

                        if (b[0][col] == player) :
                                return 10
                        elif (b[0][col] == opponent) :
                                return -10
        if (b[0][0] == b[1][1] and b[1][1] == b[2][2]) :
                if (b[0][0] == player) :
                        return 10
                elif (b[0][0] == opponent) :
                        return -10
        if (b[0][2] == b[1][1] and b[1][1] == b[2][0]) :
```

```
        if (b[0][2] == player) :
            return 10
        elif (b[0][2] == opponent) :
            return -10
    return 0
def minimax(board, depth, isMax) :
    score = evaluate(board)
    if (score == 10) :
        return score
    if (score == -10) :
        return score
    if (isMovesLeft(board) == False) :
        return 0
    if (isMax) :
        best = -1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j]=='_') :
                    board[i][j] = player
                    best = max( best, minimax(board, depth + 1, not isMax)
)
                    board[i][j] = '_'
        return best
    else :
        best = 1000
        for i in range(3) :
            for j in range(3) :
                if (board[i][j] == '_') :
                    board[i][j] = opponent
                    best = min(best, minimax(board, depth + 1, not isMax))
                    board[i][j] = '_'
```

```
        return best
def findBestMove(board) :
    bestVal = -1000
    bestMove = (-1, -1)
    for i in range(3) :
        for j in range(3) :
            if (board[i][j] == '_') :
                board[i][j] = player
                moveVal = minimax(board, 0, False)
                board[i][j] = '_'
                if (moveVal > bestVal) :
                    bestMove = (i, j)
                    bestVal = moveVal
    print("The value of the best Move is :", bestVal)
    print()
    return bestMove
board = [
    [ 'x', 'o', 'x' ], [ 'o', 'o', 'x' ], [ '_', '_', '_' ]
]
bestMove = findBestMove(board)
print("The Optimal Move is :")
print("ROW:", bestMove[0], " COL:", bestMove[1])
```

Output:

```
============= RESTART: C:/Users/Adarsh Gupta/Desktop/tic toe  5a.py ============
The value of the best Move is : 10

The Optimal Move is :
ROW: 2  COL: 2
```

Practical No. 5B
Aim: Shuffle deck of cards.

Code:

```
Suits = ["\u2663", "\u2665",
         "\u2666", "\u2660"]

Ranks = ['A', '2', '3', '4', '5',
         '6', '7', '8', '9', '10',
         'J', 'Q', 'K']

for rank in Ranks:
    for suit in Suits:
        print(f'{rank} of {suit}'.ljust(10), end='')
    print()
```

Output:

```
============ RESTART: C:/Users/Adarsh Gupta/Deskt
A of ♣    A of ♥    A of ♦    A of ♠
2 of ♣    2 of ♥    2 of ♦    2 of ♠
3 of ♣    3 of ♥    3 of ♦    3 of ♠
4 of ♣    4 of ♥    4 of ♦    4 of ♠
5 of ♣    5 of ♥    5 of ♦    5 of ♠
6 of ♣    6 of ♥    6 of ♦    6 of ♠
7 of ♣    7 of ♥    7 of ♦    7 of ♠
8 of ♣    8 of ♥    8 of ♦    8 of ♠
9 of ♣    9 of ♥    9 of ♦    9 of ♠
10 of ♣   10 of ♥   10 of ♦   10 of ♠
J of ♣    J of ♥    J of ♦    J of ♠
Q of ♣    Q of ♥    Q of ♦    Q of ♠
K of ♣    K of ♥    K of ♦    K of ♠
```

Practical No. 6A

Aim: Design an application to simulate number puzzle problem.

Code:

```
import copy
from heapq import heappush, heappop


n = 3
row = [ 1, 0, -1, 0 ]
col = [ 0, -1, 0, 1 ]


class priorityQueue:
  def __init__(self):
    self.heap = []
  def push(self, k):
    heappush(self.heap, k)
  def pop(self):
    return heappop(self.heap)
  def empty(self):
    if not self.heap:
      return True
    else:
      return False


class node:
  def __init__(self, parent, mat, empty_tile_pos,
          cost, level):
    self.parent = parent
    self.mat = mat
    self.empty_tile_pos = empty_tile_pos
    self.cost = cost
```

```
        self.level = level
    def __lt__(self, nxt):
        return self.cost < nxt.cost

def calculateCost(mat, final) -> int:
    count = 0
    for i in range(n):
        for j in range(n):
            if ((mat[i][j]) and
                (mat[i][j] != final[i][j])):
                count += 1
    return count

def newNode(mat, empty_tile_pos, new_empty_tile_pos,
            level, parent, final) -> node:
    new_mat = copy.deepcopy(mat)

    x1 = empty_tile_pos[0]
    y1 = empty_tile_pos[1]
    x2 = new_empty_tile_pos[0]
    y2 = new_empty_tile_pos[1]
    new_mat[x1][y1], new_mat[x2][y2] = new_mat[x2][y2], new_mat[x1][y1]

    cost = calculateCost(new_mat, final)
    new_node = node(parent, new_mat, new_empty_tile_pos,
                    cost, level)
    return new_node

def printMatrix(mat):
    for i in range(n):
        for j in range(n):
```

```
            print("%d " % (mat[i][j]), end = " ")
        print()

def isSafe(x, y):
    return x >= 0 and x < n and y >= 0 and y < n

def printPath(root):
    if root == None:
        return
    printPath(root.parent)
    printMatrix(root.mat)
    print()

def solve(initial, empty_tile_pos, final):
    pq = priorityQueue()
    cost = calculateCost(initial, final)
    root = node(None, initial,
                empty_tile_pos, cost, 0)
    pq.push(root)

    while not pq.empty():
        minimum = pq.pop()
        if minimum.cost == 0:
            printPath(minimum)
            return
        for i in range(4):
            new_tile_pos = [
                minimum.empty_tile_pos[0] + row[i],
                minimum.empty_tile_pos[1] + col[i], ]
            if isSafe(new_tile_pos[0], new_tile_pos[1]):
                child = newNode(minimum.mat,
```

```
                        minimum.empty_tile_pos,
                        new_tile_pos,
                        minimum.level + 1,
                        minimum, final,)
                pq.push(child)

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

empty_tile_pos = [ 1, 2 ]
solve(initial, empty_tile_pos, final)
```

Output:

```
=========== RESTART: C:/Users/Adarsh Gupta/Desk
1   2   3
5   6   0
7   8   4

1   2   3
5   0   6
7   8   4

1   2   3
5   8   6
7   0   4

1   2   3
5   8   6
0   7   4
```

Practical No. 7A

Aim: Solve constraint satisfaction problem.

Code:

```
class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]
    def isSafe(self, v, colour, c):
        for i in range(self.V):
            if self.graph[v][i] == 1 and colour[i] == c:
                return False
        return True
    def graphColourUtil(self, m, colour, v):
        if v == self.V:
            return True
        for c in range(1, m + 1):
            if self.isSafe(v, colour, c) == True:
                colour[v] = c
                if self.graphColourUtil(m, colour, v + 1) == True:
                    return True
                colour[v] = 0
    def graphColouring(self, m):
        colour = [0] * self.V
        if self.graphColourUtil(m, colour, 0) == None:
            return False
        print("Solution exist and Following are the assigned colours:")
        for c in colour:
            print(c, end=' ')
        return True
```

```
if __name__ == '__main__':
   g = Graph(4)
   g.graph = [[0, 1, 1, 1], [1, 0, 1, 0], [1, 1, 0, 1], [1, 0, 1, 0]]
   m = 3

   g.graphColouring(m)
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16)
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more

= RESTART: C:/Users/Adarsh Gupta/Desktop/constraint 7a.py
Solution exist and Following are the assigned colours:
1 2 3 2
```

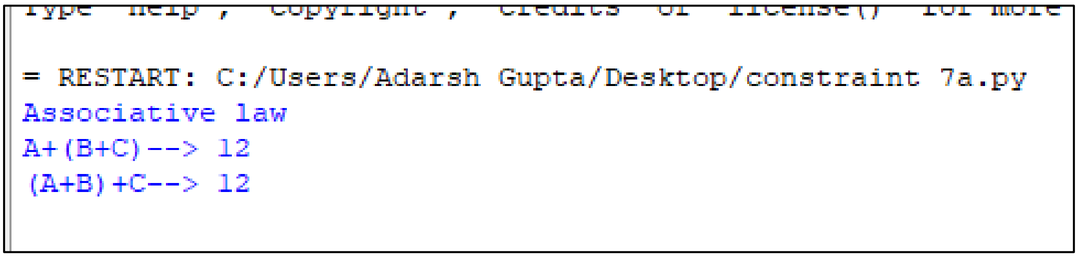
Practical No. 8A

Aim: Derive the expressions based on Associative Law.

Code:

```
import random


a=3

b=2

c=7


print("Associative law")

print("A+(B+C)-->",(a+(b+c)))

print("(A+B)+C-->",((a+b)+c))
```

Output:



```
= RESTART: C:/Users/Adarsh Gupta/Desktop/constraint 7a.py
Associative law
A+(B+C)--> 12
(A+B)+C--> 12
```

Practical No. 8B
Aim: Derive the expressions based on Distributive Law.

Code:

```
import random


a=random.randint(0,99)

b=random.randint(0,99)

c=random.randint(0,99)


print("Distributive law")


print("A(B+C)-->",(a*(b+c)))

print("(A*B)+(A*C)-->",((a*b)+(a*c)))
```

Output:

```
Type "help", "copyright", "credits" or "license()" for more

= RESTART: C:/Users/Adarsh Gupta/Desktop/constraint 7a.py
Distributive law
A(B+C)--> 1200
(A*B)+(A*C)--> 1200
```

Practical No. 9A

Aim: Derive the predicate. (for e.g.: Sachin is batsman, batsman is cricketer)

- > Sachin is Cricketer

Code:

```
class PredicateDeriver:
    def __init__(self):
        self.relationships = {}

    def add_relationship(self, subject, relation, obj):
        self.relationships[(subject, relation)] = obj

    def derive(self, subject, relation):
        if (subject, relation) in self.relationships:
            obj = self.relationships[(subject, relation)]
            if relation in self.relationships:
                return self.derive(obj, self.relationships[(obj, relation)])
            return obj
        return None

deriver = PredicateDeriver()
deriver.add_relationship('Sachin', 'is', 'batsman')
deriver.add_relationship('batsman', 'is', 'cricketer')

result = deriver.derive('Sachin', 'is')
print(f"Sachin is Cricketer: {result}")
```

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [I
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more i

= RESTART: C:/Users/Adarsh Gupta/Desktop/constraint 7a.py
Sachin is Cricketer: batsman
```

Practical No. 10A

Aim: Write a program which contains three predicates: male, female, parent. Make rules for following family relations: father, mother, grandfather, grandmother, brother, sister, uncle, aunt, nephew and niece, cousin. Question: i. Draw Family Tree. ii. Define: Clauses, Facts, Predicates and Rules with conjunction and disjunction

Code:

```
class FamilyRelations:
    def __init__(self):
        self.facts = {
            'male': ['John', 'Mike', 'Tom'],
            'female': ['Jane', 'Lisa', 'Anna'],
            'parent': {
                'John': ['Mike', 'Anna'],
                'Jane': ['Mike', 'Anna'],
                'Mike': ['Tom'],
                'Lisa': ['Tom']
            }
        }
    def is_father(self, child):
        for father in self.facts['male']:
            if child in self.facts['parent'].get(father, []):
                return father
        return None

    def is_mother(self, child):
        for mother in self.facts['female']:
            if child in self.facts['parent'].get(mother, []):
                return mother
```

```
        return None
    def is_grandfather(self, grandchild):
        father = self.is_father(grandchild)
        if father:
            return self.is_father(father)
        return None

    def is_grandmother(self, grandchild):
        mother = self.is_mother(grandchild)
        if mother:
            return self.is_mother(mother)
        return None

    def is_brother(self, sibling):
        father = self.is_father(sibling)
        siblings = self.facts['parent'].get(father, [])
        return [bro for bro in siblings if bro != sibling and bro in self.facts['male']]

    def is_sister(self, sibling):
        father = self.is_father(sibling)
        siblings = self.facts['parent'].get(father, [])
        return [sis for sis in siblings if sis != sibling and sis in self.facts['female']]

family = FamilyRelations()
print("Father of Tom:", family.is_father('Tom'))
print("Mother of Tom:", family.is_mother('Tom'))
print("Grandfather of Tom:", family.is_grandfather('Tom'))
print("Grandmother of Tom:", family.is_grandmother('Tom'))
print("Brothers of Anna:", family.is_brother('Anna'))
```

print("Sisters of Mike:", family.is_sister('Mike'))

Output:

```
Python 3.12.4 (tags/v3.12.4:8e8a4ba, Jun  6 2024, 19:30:16) [M
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more in

= RESTART: C:/Users/Adarsh Gupta/Desktop/constraint 7a.py
Father of Tom: Mike
Mother of Tom: Lisa
Grandfather of Tom: John
Grandmother of Tom: None
Brothers of Anna: ['Mike']
Sisters of Mike: ['Anna']
```