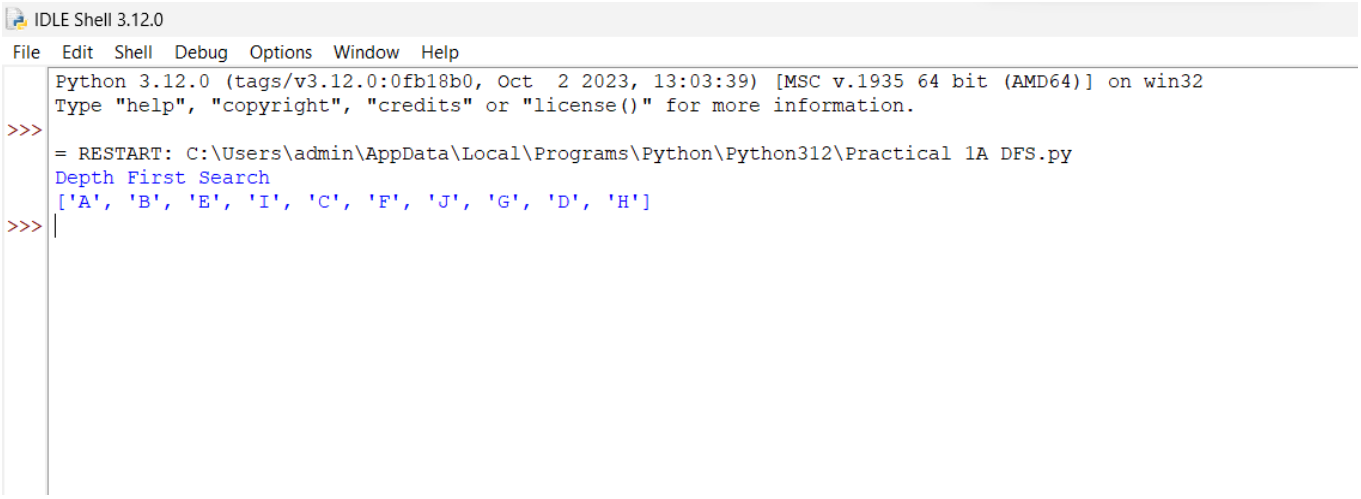
Practical 1A

Depth First Search.

CODE:-

```
graph={
   "A":['D','C','B'],
   "B":['E'],
   "C":['G','F'],
   "D":['H'],
   "E":['I'],
   "F":['J']
   }
path=[]
stack=['A']
while(len(stack)!=0):
   s=stack.pop()
   if s not in path:
      path.append(s)
   if s not in graph:
      continue
   for neighbour in graph[s]:
      stack.append(neighbour)
print(path)
```

OUTPUT:-


IDLE Shell 3.12.0
File Edit Shell Debug Options Window Help
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\admin\AppData\Local\Programs\Python\Python312\Practical 1A DFS.py
Depth First Search
['A', 'B', 'E', 'I', 'C', 'F', 'J', 'G', 'D', 'H']
>>>

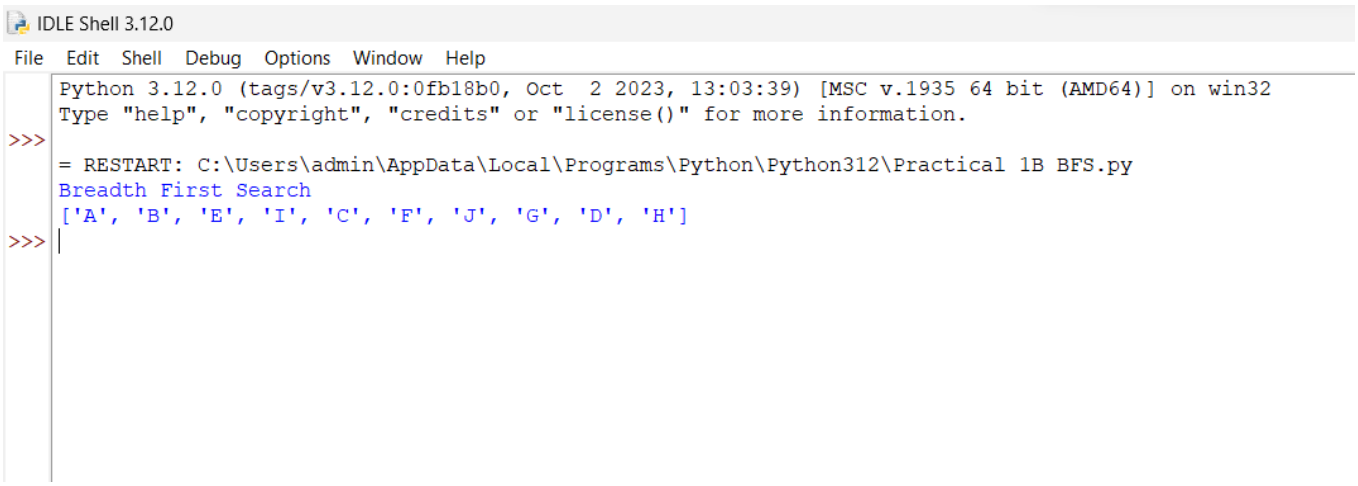
PRACTICAL 1B

Breadth First Search.

CODE:-

```
graph={
   "A":['D','C','B'],
   "B":['E'],
   "C":['G','F'],
   "D":['H'],
   "E":['I'],
   "F":['J']
   }
print("Aryan Ankushrao-01")
path=[]
stack=['A']
while(len(stack)!=0):
   s=stack.pop()
   if s not in path:
      path.append(s)
   if s not in graph:
      continue
   for neighbour in graph[s]:
      stack.append(neighbour)
print(path)
```

OUTPUT:-


IDLE Shell 3.12.0
File Edit Shell Debug Options Window Help
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct 2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\admin\AppData\Local\Programs\Python\Python312\Practical 1B BFS.py
Breadth First Search
['A', 'B', 'E', 'I', 'C', 'F', 'J', 'G', 'D', 'H']
>>>

PRACTICAL 2A

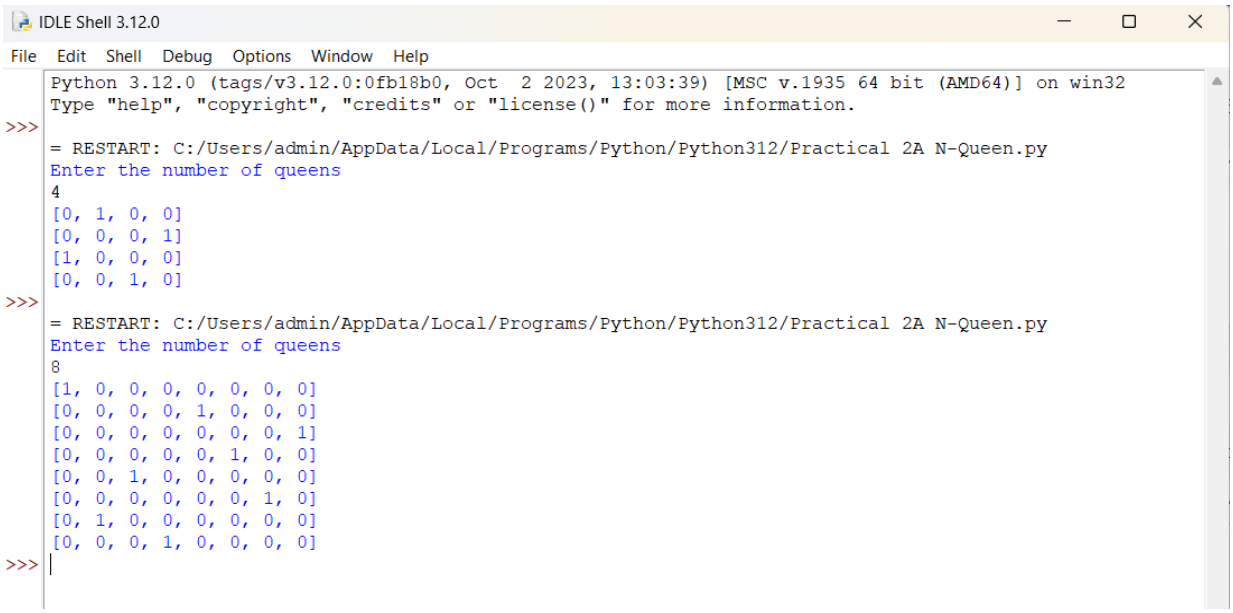
N-Queen Problem.

CODE:-

```
#Number of queens
print ("Enter the number of queens")
N = int(input())
#chessboard
#NxN matrix with all elements 0
board = [[0]*N for _ in range(N)]
def is_attack(i, j):
#checking if there is a queen in row or column
   for k in range(0,N):
      if board[i][k]==1 or board[k][j]==1:
         return True
#checking diagonals
   for k in range(0,N):
      for l in range(0,N):
         if (k+l==i+j) or (k-l==i-j):
            if board[k][l]==1:
               return True
   return False

def N_queen(n):
#if n is 0, solution found
   if n==0:
```

```
        return True
    for i in range(0,N):
        for j in range(0,N):
            '''checking if we can place a queen here or not
            queen will not be placed if the place is being attacked
            or already occupied'''
            if (not(is_attack(i,j))) and (board[i][j]!=1):
                board[i][j] = 1
#recursion
#wether we can put the next queen with this arrangment or not
                if N_queen(n-1)==True:
                    return True
                board[i][j] = 0
    return False
N_queen(N)
for i in board:
    print (i)
```

OUTPUT:-



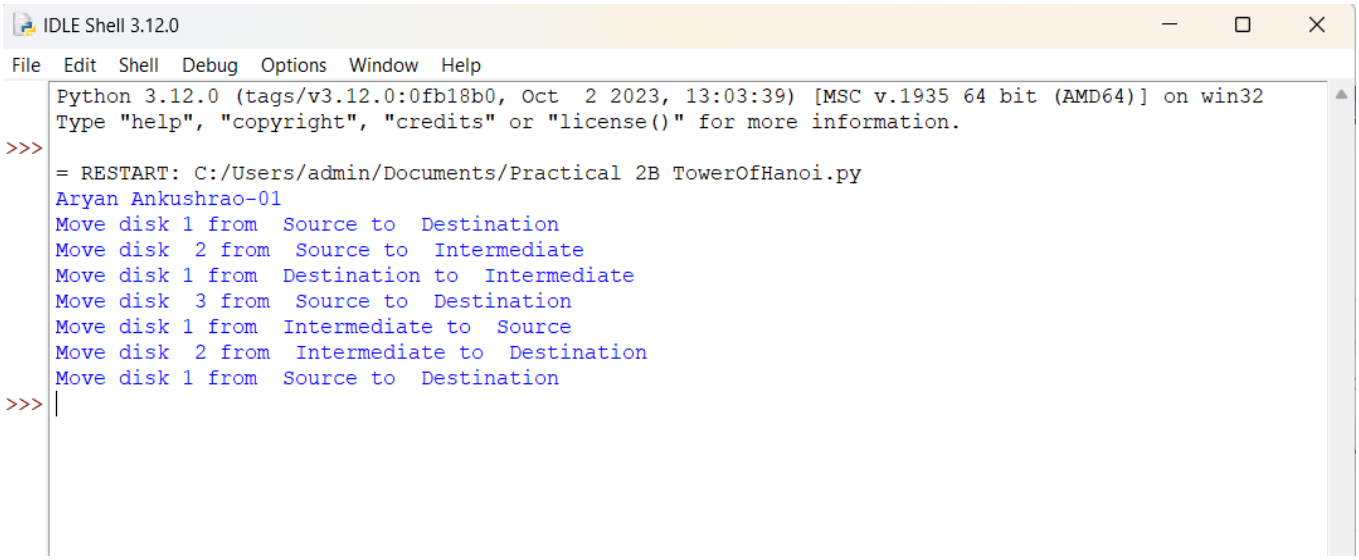
PRACTICAL 2B

Tower of Hanoi Problem.

CODE:-

```
def TowerOfHanoi(n , source, destination, Intermediate):
    if n==1:
        print("Move disk 1 from ",source,"to ",destination)
        return
    TowerOfHanoi(n-1, source, Intermediate, destination)
    print("Move disk ",n, "from ",source,"to ",destination)
    TowerOfHanoi(n-1, Intermediate, destination, source)
n=3
TowerOfHanoi(n,'Source','Destination','Intermediate')
```

OUTPUT:-



PRACTICAL 3A

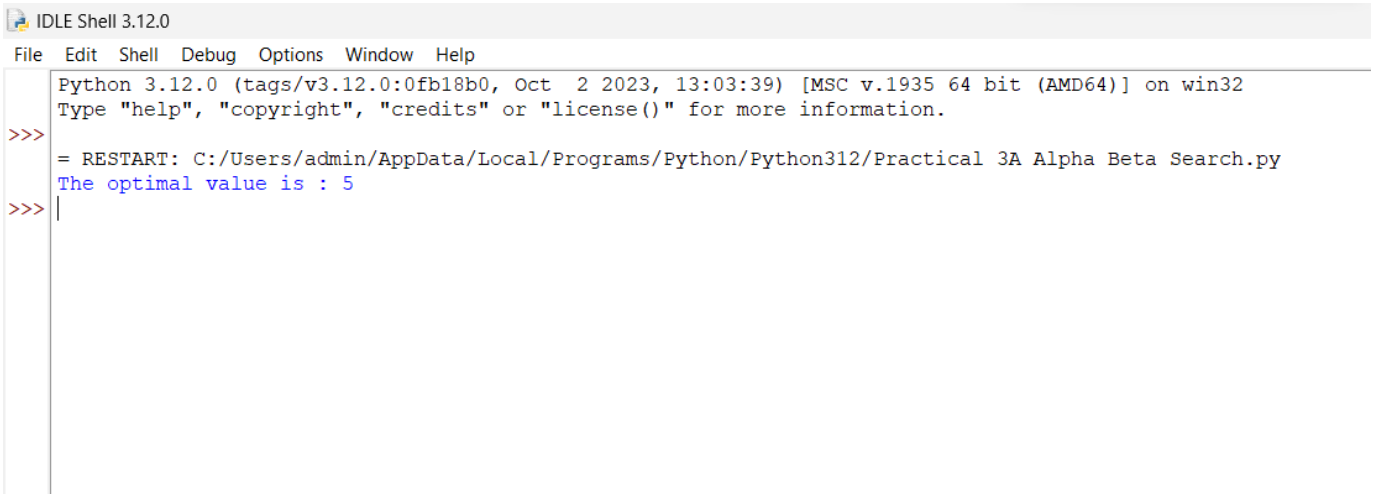
Alpha Beta search

Code:-

```
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta):

if depth == 3:

return values[nodeIndex]


if maximizingPlayer:

best = MIN

for i in range(0, 2):

val = minimax(depth + 1, nodeIndex * 2 + i,

False, values, alpha, beta)

best = max(best, val)
```

```
alpha = max(alpha, best)

# Alpha Beta Pruning
if beta <= alpha:
break
return best
else:
best = MAX

for i in range(0, 2):
val = minimax(depth + 1, nodeIndex * 2 + i,
True, values, alpha, beta)
best = min(best, val)
beta = min(beta, best)

# Alpha Beta Pruning
if beta <= alpha:
break
return best
if __name__ == "__main__":

values = [3, 5, 6, 9, 1, 2, 0, -1]
print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:-



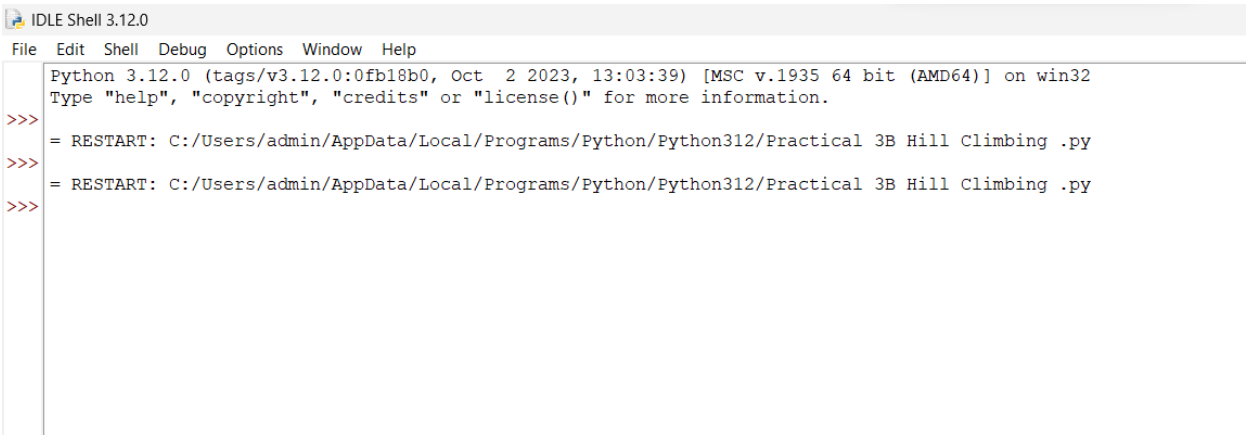
Practical 3B

Hill Climbing

Code:-

```
def hill_climbing(f, x0):
x = x0 # initial solution
while True:
neighbors = generate_neighbors(x) # generate neighbors of x
# find the neighbor with the highest function value
best_neighbor = max(neighbors, key=f)
if f(best_neighbor) <= f(x): # if the best neighbor is not better than x, stop
return x
x = best_neighbor # otherwise, continue with the best neighbor
```

Output:-



Practical 4 A

A * Algorithm

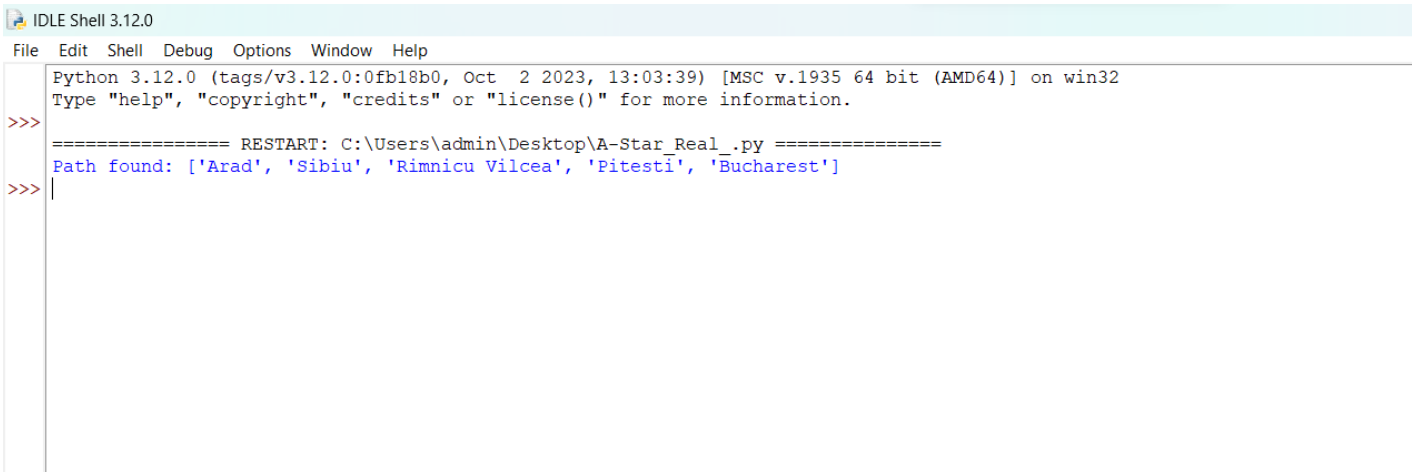
CODE:-

```
import heapq

romania_map = {

'Arad': {'Zerind': 75, 'Timisoara': 118, 'Sibiu': 140},

'Zerind': {'Arad': 75, 'Oradea': 71},

'Timisoara': {'Arad': 118, 'Lugoj': 111},

'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea': 80},

'Oradea': {'Zerind': 71, 'Sibiu': 151},

'Lugoj': {'Timisoara': 111, 'Mehadia': 70},

'Fagaras': {'Sibiu': 99, 'Bucharest': 211},

'Rimnicu Vilcea': {'Sibiu': 80, 'Pitesti': 97, 'Craiova': 146},

'Mehadia': {'Lugoj': 70, 'Drobeta': 75},

'Drobeta': {'Mehadia': 75, 'Craiova': 120},

'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},

'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
```

```
'Bucharest': {'Fagaras': 211, 'Pitesti': 101, 'Giurgiu': 90, 'Urziceni': 85},
'Giurgiu': {'Bucharest': 90},
'Urziceni': {'Bucharest': 85, 'Hirsova': 98, 'Vaslui': 142},
'Hirsova': {'Urziceni': 98, 'Eforie': 86},
'Eforie': {'Hirsova': 86},
'Vaslui': {'Urziceni': 142, 'Iasi': 92},
'Iasi': {'Vaslui': 92, 'Neamt': 87},
'Neamt': {'Iasi': 87}
}
class Node:
    def __init__(self, city, cost, parent=None):
        self.city = city
        self.cost = cost
        self.parent = parent
    def __lt__(self, other):
        return self.cost < other.cost
def astar_search(graph, start, goal):
    open_list = []
    closed_set = set()
    heapq.heappush(open_list, start)
    while open_list:
        current_node = heapq.heappop(open_list)
        if current_node.city == goal.city:
            path = []
```

```
            while current_node:
                path.append(current_node.city)
                current_node = current_node.parent
            return path[::-1]  # Reverse the path to get it from start to goal
        closed_set.add(current_node.city)
        for neighbor, distance in graph[current_node.city].items():
            if neighbor not in closed_set:
                new_cost = current_node.cost + distance
                new_node = Node(neighbor, new_cost, current_node)
                heapq.heappush(open_list, new_node)
    return None

start_city = 'Arad'
goal_city = 'Bucharest'
start_node = Node(start_city, 0)
goal_node = Node(goal_city, 0)
path = astar_search(romania_map, start_node, goal_node)
if path:
    print("Path found:", path)
else:
    print("No path found")
```

OUTPUT:-



Practical 5 A

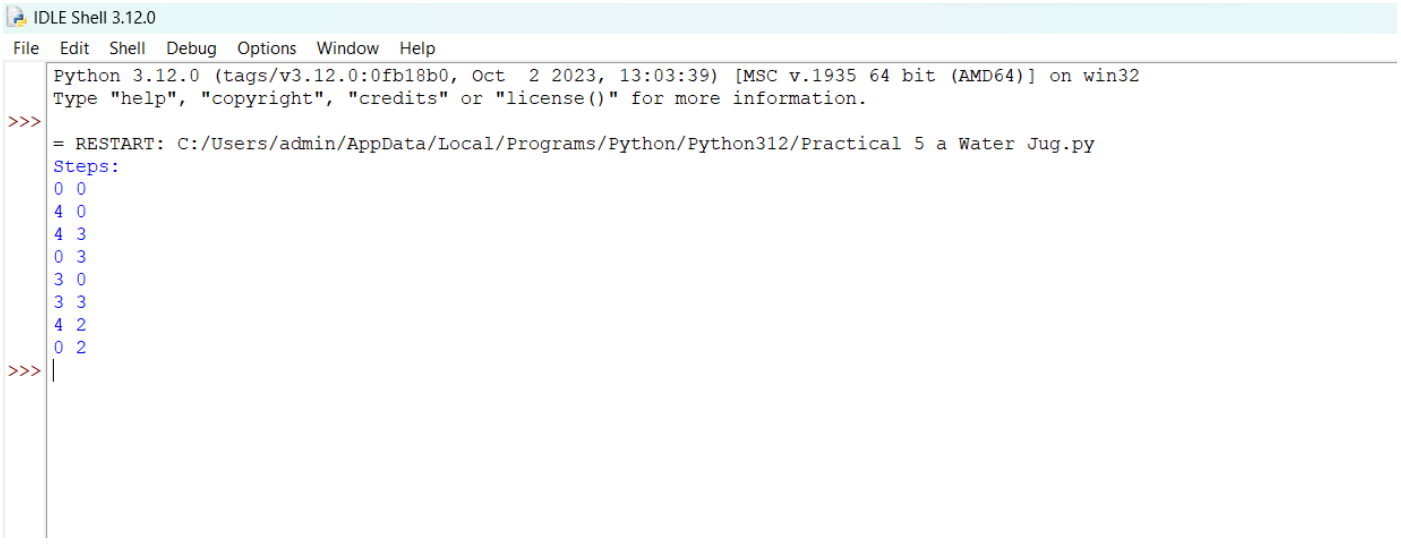
Water Jug Problem

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:-



Practical 5 B

Tic Tac Toe.

CODE:-

```
from math import inf as infinity
from random import choice
import platform
import time
from os import system

HUMAN = -1
COMP = +1
board = [
    [0, 0, 0],
    [0, 0, 0],
```

```
    [0, 0, 0],
]
def evaluate(state):
    """
    Function to heuristic evaluation of state.
    :param state: the state of the current board
    :return: +1 if the computer wins; -1 if the human wins; 0 draw
    """
    if wins(state, COMP):
        score = +1
    elif wins(state, HUMAN):
        score = -1
    else:
        score = 0
    return score
def wins(state, player):
    """
    This function tests if a specific player wins. Possibilities:
    * Three rows    [X X X] or [O O O]
    * Three cols    [X X X] or [O O O]
    * Two diagonals [X X X] or [O O O]
    :param state: the state of the current board
    :param player: a human or a computer
    :return: True if the player wins
```

```
    """
    win_state = [
        [state[0][0], state[0][1], state[0][2]],
        [state[1][0], state[1][1], state[1][2]],
        [state[2][0], state[2][1], state[2][2]],
        [state[0][0], state[1][0], state[2][0]],
        [state[0][1], state[1][1], state[2][1]],
        [state[0][2], state[1][2], state[2][2]],
        [state[0][0], state[1][1], state[2][2]],
        [state[2][0], state[1][1], state[0][2]],
    ]
    if [player, player, player] in win_state:
        return True
    else:
        return False
def game_over(state):
    """
    This function test if the human or computer wins
    :param state: the state of the current board
    :return: True if the human or computer wins
    """
    return wins(state, HUMAN) or wins(state, COMP)
def empty_cells(state):
    """
```

```
    Each empty cell will be added into cells' list
    :param state: the state of the current board
    :return: a list of empty cells
    """
    cells = []
    for x, row in enumerate(state):
        for y, cell in enumerate(row):
            if cell == 0:
                cells.append([x, y])
    return cells
def valid_move(x, y):
    """
    A move is valid if the chosen cell is empty
    :param x: X coordinate
    :param y: Y coordinate
    :return: True if the board[x][y] is empty
    """
    if [x, y] in empty_cells(board):
        return True
    else:
        return False
def set_move(x, y, player):
    """
    Set the move on board, if the coordinates are valid
```

```
    :param x: X coordinate
    :param y: Y coordinate
    :param player: the current player
    """
    if valid_move(x, y):
        board[x][y] = player
        return True
    else:
        return False
def minimax(state, depth, player):
    """
    AI function that choice the best move
    :param state: current state of the board
    :param depth: node index in the tree (0 <= depth <= 9),
    but never nine in this case (see iaturn() function)
    :param player: an human or a computer
    :return: a list with [the best row, best col, best score]
    """
    if player == COMP:
        best = [-1, -1, -infinity]
    else:
        best = [-1, -1, +infinity]
    if depth == 0 or game_over(state):
        score = evaluate(state)
```

```
        return [-1, -1, score]

    for cell in empty_cells(state):
        x, y = cell[0], cell[1]
        state[x][y] = player
        score = minimax(state, depth - 1, -player)
        state[x][y] = 0
        score[0], score[1] = x, y
        if player == COMP:
            if score[2] > best[2]:
                best = score  # max value
        else:
            if score[2] < best[2]:
                best = score  # min value
    return best
def clean():
    """
    Clears the console
    """
    os_name = platform.system().lower()
    if 'windows' in os_name:
        system('cls')
    else:
        system('clear')
```

```
def render(state, c_choice, h_choice):
    """
    Print the board on console
    :param state: current state of the board
    """
    chars = {
        -1: h_choice,
        +1: c_choice,
        0: ' '
    }
    str_line = '---------------'
    print('\n' + str_line)
    for row in state:
        for cell in row:
            symbol = chars[cell]
            print(f'| {symbol} |', end='')
        print('\n' + str_line)
def ai_turn(c_choice, h_choice):
    """
    It calls the minimax function if the depth < 9,
    else it choices a random coordinate.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
```

```
    """
    depth = len(empty_cells(board))
    if depth == 0 or game_over(board):
        return
    clean()
    print(f'Computer turn [{c_choice}]')
    render(board, c_choice, h_choice)
    if depth == 9:
        x = choice([0, 1, 2])
        y = choice([0, 1, 2])
    else:
        move = minimax(board, depth, COMP)
        x, y = move[0], move[1]
    set_move(x, y, COMP)
    time.sleep(1)


def human_turn(c_choice, h_choice):
    """
    The Human plays choosing a valid move.
    :param c_choice: computer's choice X or O
    :param h_choice: human's choice X or O
    :return:
    """
```

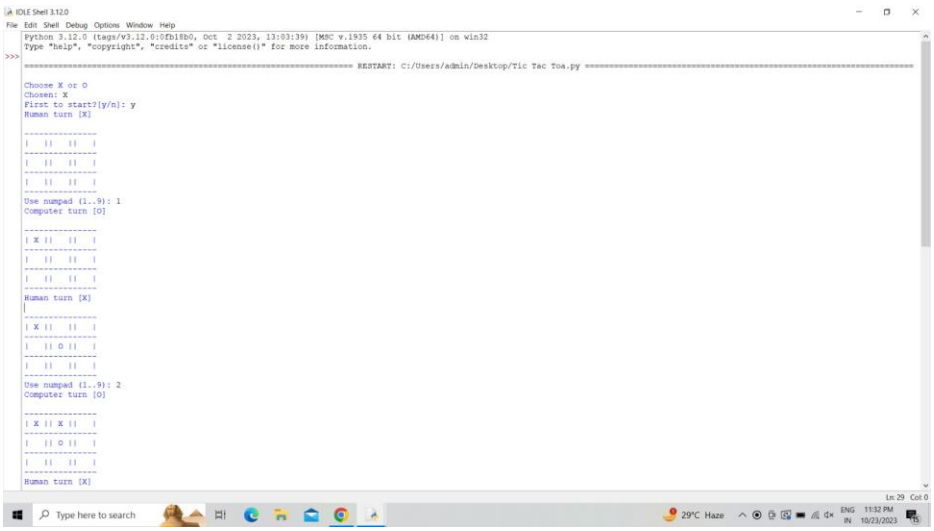
```
depth = len(empty_cells(board))
if depth == 0 or game_over(board):
    return
# Dictionary of valid moves
move = -1
moves = {
    1: [0, 0], 2: [0, 1], 3: [0, 2],
    4: [1, 0], 5: [1, 1], 6: [1, 2],
    7: [2, 0], 8: [2, 1], 9: [2, 2],
}
clean()
print(f'Human turn [{h_choice}]')
render(board, c_choice, h_choice)
while move < 1 or move > 9:
    try:
        move = int(input('Use numpad (1..9): '))
        coord = moves[move]
        can_move = set_move(coord[0], coord[1], HUMAN)

        if not can_move:
            print('Bad move')
            move = -1
    except (EOFError, KeyboardInterrupt):
        print('Bye')
```

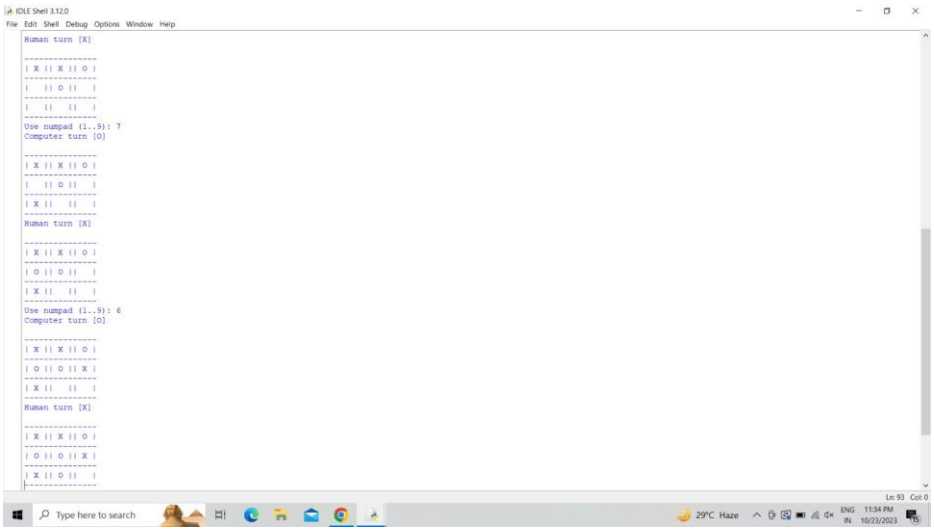
```
            exit()
        except (KeyError, ValueError):
            print('Bad choice')



def main():
    """
    Main function that calls all functions
    """
    clean()
    h_choice = ''  # X or O
    c_choice = ''  # X or O
    first = ''  # if human is the first
    # Human chooses X or O to play
    while h_choice != 'O' and h_choice != 'X':
        try:
            print('')
            h_choice = input('Choose X or O\nChosen: ').upper()
        except (EOFError, KeyboardInterrupt):
            print('Bye')
            exit()
        except (KeyError, ValueError):
            print('Bad choice')
    # Setting computer's choice
```

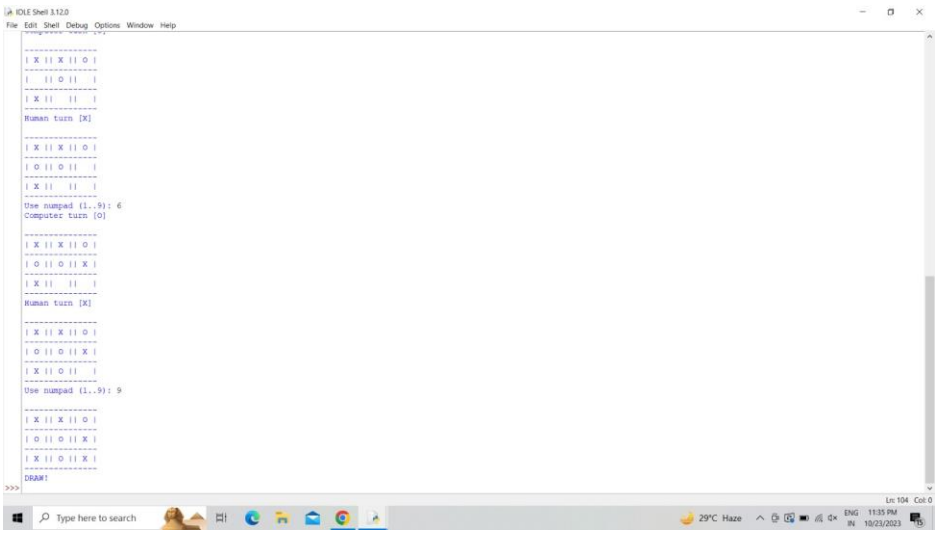
```
if h_choice == 'X':
    c_choice = 'O'
else:
    c_choice = 'X'
# Human may starts first
clean()
while first != 'Y' and first != 'N':
    try:
        first = input('First to start?[y/n]: ').upper()
    except (EOFError, KeyboardInterrupt):
        print('Bye')
        exit()
    except (KeyError, ValueError):
        print('Bad choice')
# Main loop of this game
while len(empty_cells(board)) > 0 and not game_over(board):
    if first == 'N':
        ai_turn(c_choice, h_choice)
        first = ''
    human_turn(c_choice, h_choice)
    ai_turn(c_choice, h_choice)
# Game over message
if wins(board, HUMAN):
    clean()
```

```
        print(f'Human turn [{h_choice}]')
        render(board, c_choice, h_choice)
        print('YOU WIN!')
    elif wins(board, COMP):
        clean()
        print(f'Computer turn [{c_choice}]')
        render(board, c_choice, h_choice)
        print('YOU LOSE!')
    else:
        clean()
        render(board, c_choice, h_choice)
        print('DRAW!')
    exit()
if __name__ == '__main__':
    main()
```

OUTPUT:-







Practical 6 A

Missionaries and Cannibal

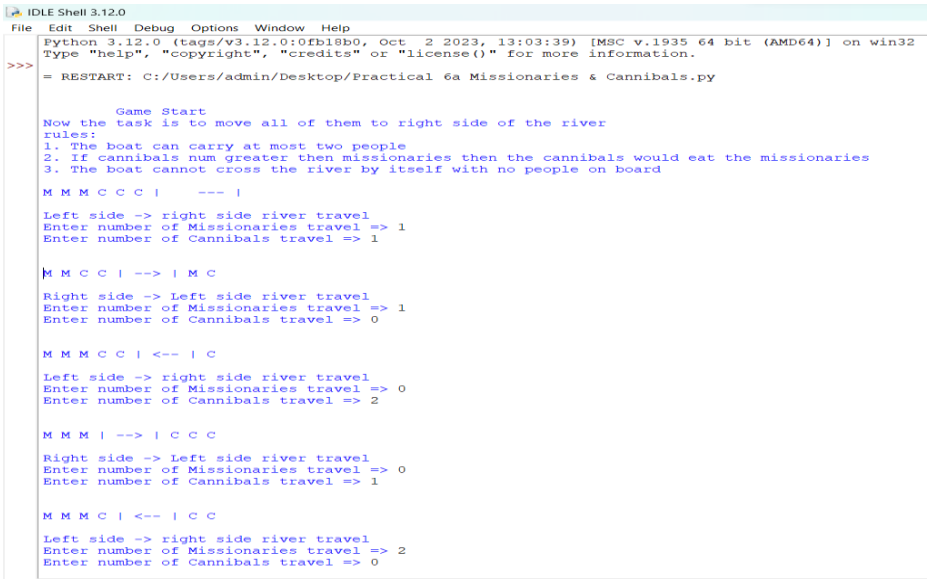
CODE:-

```
print("\n")
print("\tGame Start\nNow the task is to move all of them to right side of the river")
print("rules:\n1. The boat can carry at most two people\n2. If cannibals num greater then missionaries then the cannibals would eat the missionaries\n3. The boat cannot cross the river by itself with no people on board")
lM = 3
lC = 3
rM=0
rC=0
userM = 0
userC = 0
k = 0
print("\nM M M C C C |    --- | \n")
try:
        while(True):
                while(True):
                        print("Left side -> right side river travel")

                        uM = int(input("Enter number of Missionaries travel => "))
                        uC = int(input("Enter number of Cannibals travel => "))

                        if((uM==0)and(uC==0)):
                                print("Empty travel not possible")
                                print("Re-enter : ")
                        elif(((uM+uC) <= 2)and((lM-uM)>=0)and((lC-uC)>=0)):
                                break
                        else:
                                print("Wrong input re-enter : ")
                lM = (lM-uM)
                lC = (lC-uC)
                rM += uM
```

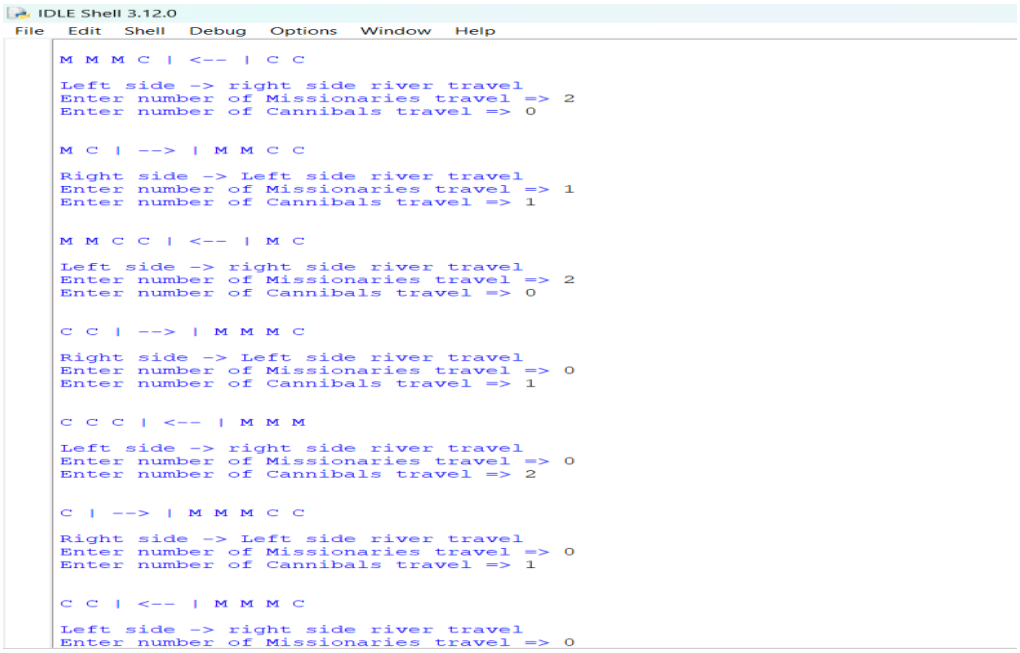
```
            rC += uC

            print("\n")
            for i in range(0,lM):
                print("M ",end="")
            for i in range(0,lC):
                print("C ",end="")
            print("| --> | ",end="")
            for i in range(0,rM):
                print("M ",end="")
            for i in range(0,rC):
                print("C ",end="")
            print("\n")

            k +=1

            if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                print("Cannibals eat missionaries:\nYou lost the game")

                break

            if((rM+rC) == 6):
                print("You won the game : \n\tCongrats")
                print("Total attempt")
                print(k)
                break
            while(True):
                print("Right side -> Left side river travel")
                userM = int(input("Enter number of Missionaries travel => "))
                userC = int(input("Enter number of Cannibals travel => "))

                if((userM==0)and(userC==0)):
                            print("Empty travel not possible")
                            print("Re-enter : ")
```

```
                elif(((userM+userC) <= 2)and((rM-userM)>=0)and((rC-
userC)>=0)):
                        break
                else:
                        print("Wrong input re-enter : ")
        lM += userM
        lC += userC
        rM -= userM
        rC -= userC

        k +=1
        print("\n")
        for i in range(0,lM):
                print("M ",end="")
        for i in range(0,lC):
                print("C ",end="")
        print("| <-- | ",end="")
        for i in range(0,rM):
                print("M ",end="")
        for i in range(0,rC):
                print("C ",end="")
        print("\n")



        if(((lC==3)and (lM ==
1))or((lC==3)and(lM==2))or((lC==2)and(lM==1))or((rC==3)and (rM ==
1))or((rC==3)and(rM==2))or((rC==2)and(rM==1))):
                print("Cannibals eat missionaries:\nYou lost the game")
                break
except EOFError as e:
        print("\nInvalid input please retry !!")
```

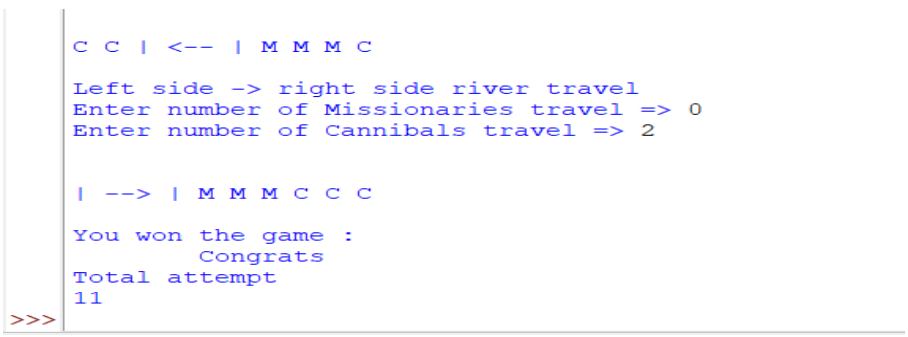
OUTPUT:-



```
IDLE Shell 3.12.0
File  Edit  Shell  Debug  Options  Window  Help
Python 3.12.0 (tags/v3.12.0:0fb18b0, Oct  2 2023, 13:03:39) [MSC v.1935 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/admin/Desktop/Practical 6a Missionaries & Cannibals.py


        Game Start
Now the task is to move all of them to right side of the river
rules:
1. The boat can carry at most two people
2. If cannibals num greater then missionaries then the cannibals would eat the missionaries
3. The boat cannot cross the river by itself with no people on board

M M M C C C |    --- |

Left side -> right side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 1


M M C C | --> | M C

Right side -> Left side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 0


M M M C C | <-- | C

Left side -> right side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 2


M M M | --> | C C C

Right side -> Left side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 1


M M M C | <-- | C C

Left side -> right side river travel
Enter number of Missionaries travel => 2
Enter number of Cannibals travel => 0
```



```
IDLE Shell 3.12.0
File  Edit  Shell  Debug  Options  Window  Help
M M M C | <-- | C C

Left side -> right side river travel
Enter number of Missionaries travel => 2
Enter number of Cannibals travel => 0


M C | --> | M M C C

Right side -> Left side river travel
Enter number of Missionaries travel => 1
Enter number of Cannibals travel => 1


M M C C | <-- | M C

Left side -> right side river travel
Enter number of Missionaries travel => 2
Enter number of Cannibals travel => 0


C C | --> | M M M C

Right side -> Left side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 1


C C C | <-- | M M M

Left side -> right side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 2


C | --> | M M M C C

Right side -> Left side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 1


C C | <-- | M M M C

Left side -> right side river travel
Enter number of Missionaries travel => 0
```



```
C C | <-- | M M M C

Left side -> right side river travel
Enter number of Missionaries travel => 0
Enter number of Cannibals travel => 2


| --> | M M M C C C

You won the game :
        Congrats
Total attempt
11
>>>
```

Practical 6 B

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]):

# Create a child node
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],

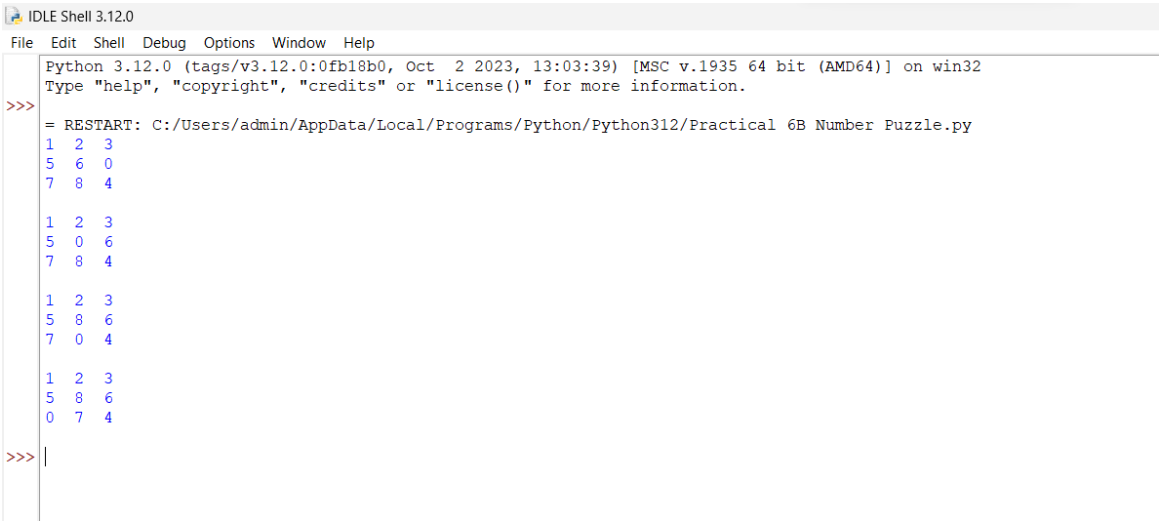
[5, 8, 6],

[0, 7, 4]]

empty_tile_pos = [1, 2]

solve(initial, empty_tile_pos, final)

Output:-



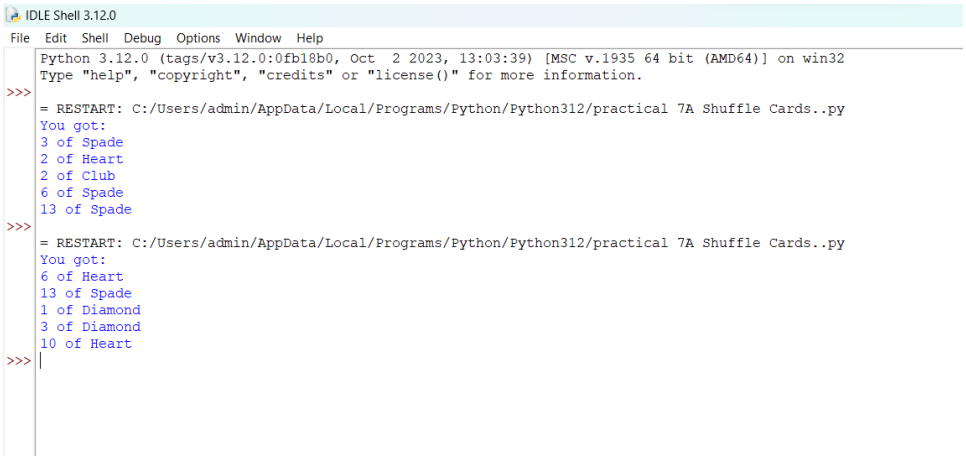
Practical 7 A

Shuffle deck of Cards.

CODE:-

```
# Python program to shuffle a deck of card

# importing modules
import itertools, random

# make a deck of cards
deck = list(itertools.product(range(1,14),['Spade','Heart','Diamond','Club']))

# shuffle the cards
random.shuffle(deck)

# draw five cards
print("You got:")
for i in range(5):
   print(deck[i][0], "of", deck[i][1])
```

OUTPUT:-



Practical 7 B

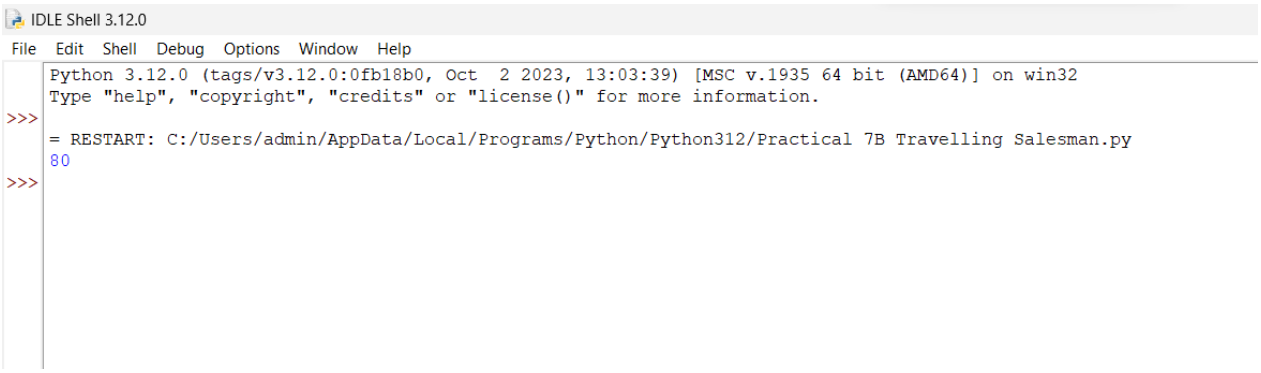
Travelling Salesman Problem

Code:-

```
from sys import maxsize
from itertools import permutations
V = 4
def travellingSalesmanProblem(graph, s):
vertex = []
for i in range(V):
if i != s:
vertex.append(i)
min_path = maxsize
next_permutation=permutations(vertex)
for i in next_permutation:
current_pathweight = 0
                k = s
for j in i:
current_pathweight += graph[k][j]
k = j
current_pathweight += graph[k][s]
                min_path = min(min_path, current_pathweight)
```

```
return min_path

if __name__ == "__main__":
        graph = [[0, 10, 15, 20], [10, 0, 35, 25],
[15, 35, 0, 30], [20, 25, 30, 0]]
s = 0
print(travellingSalesmanProblem(graph, s))
```

Output:-



Practical 8 B

Constraint Satisfaction Problem.

CODE:-

```
coloring(WA,SA,NT,QU,NSW,VI) :-
different(WA,SA),
different(WA,NT),
different(NT,SA),
different(NT,QU),
different(SA,QU),
different(SA,NSW),
different(SA,VI),
different(QU,E),
different(NSW,VI).


different(red,blue).
different(blue,red).
different(blue,green).
different(red, green).
different(green, blue,).
different(green,red).
```

OUTPUT:-

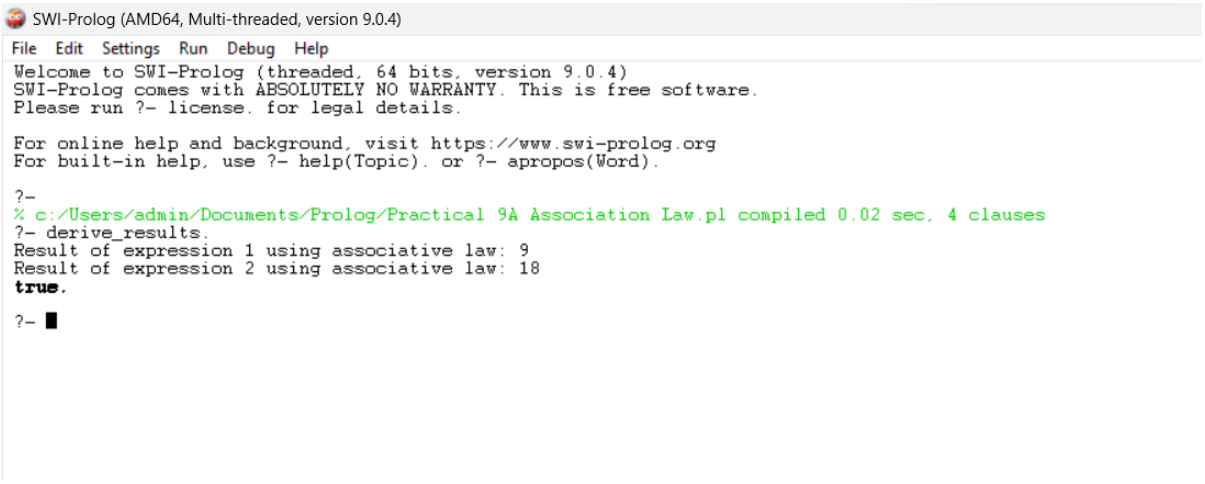
Practical 9 A

Expression based on Association Law.

Code:-

```
%Define the associative law for addition
associative_law(A, B, C, Result) :-
    Result is A+ (B + C).

%Define some example expressions

expressionl(2, 3, 4).
expression2(5, 6, 7).

%Derive the results using the associative law
derive_results :-
    expressionl(A, B, C),
    associative_law(A, B, C, Resultl),
    expression2(X, Y, Z),
    associative_law(X, Y, Z, Result2),
    write('Result of expression 1 using associative law: '), write(Resultl), nl,
    write('Result of expression 2 using associative law: '), write(Result2), nl.
```

Output:-


SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)
File Edit Settings Run Debug Help
Welcome to SWI-Prolog (threaded, 64 bits, version 9.0.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.
For online help and background, visit https://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
?-
% c:/Users/admin/Documents/Prolog/Practical 9A Association Law.pl compiled 0.02 sec, 4 clauses
?- derive_results.
Result of expression 1 using associative law: 9
Result of expression 2 using associative law: 18
true.
?-

Practical No 9B

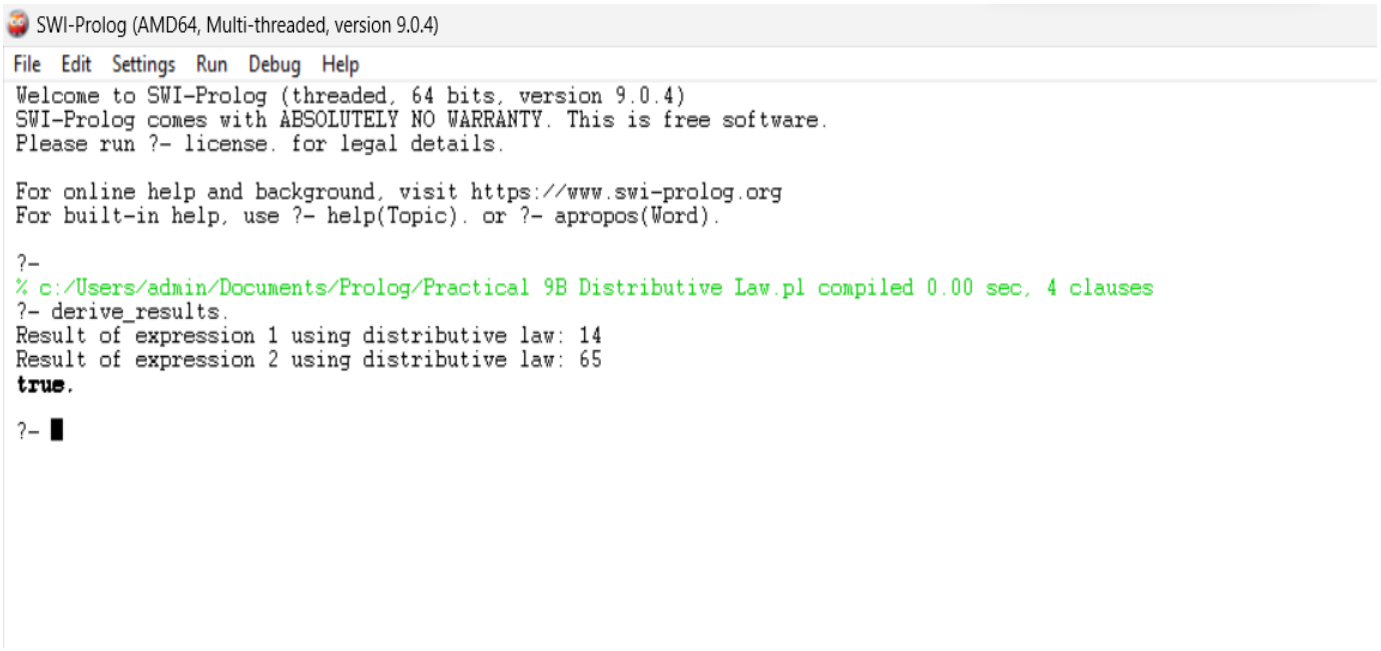
Distributive Law

Code:-

```
%Define the distributive law for multiplication over addition

distributive_law(A, B, C, Result) :-

    Result is A * (B + C) .


% Define some example expressions

expression1(2, 3, 4) .

expression2(5, 6, 7) .


% Derive the results using the distributive law distributive law

derive_results :-

    expression1(A, B, C),

    distributive_law(A, B, C, Result1),

    expression2(X, Y, Z),

    distributive_law(X, Y, Z, Result2),

    write('Result of expression 1 using distributive law: '), write(Result1),
nl,
```

write('Result of expression 2 using distributive law: '), write(Result2), nl.

Output:-



PRACTICAL 10 A

Derive the Predicate.

CODE:-

```
batsman(virat).

bowler(shami).

keeper(dhoni).
```

```
cricketer(X):-batsman(X);bowler(X);keeper(X).
```

OUTPUT:-

Practical No:-10 B

Family Tree

Code:-

```
male(adam).
male(george).
male(steve).
male(john).
```

```
female(alice).
female(anna).
female(eve).
female(steffi).
parent(adam,george).
parent(eve,george).
parent(adam,steffi).
parent(eve,steffi).
parent(george,john).
parent(anna,john).
parent(steffi,alice).
parent(steve,alice).
mother(X,Y) :- parent(X,Y), female(X).
father(X,Y) :- parent(X,Y), male(X).
sister(X,Y) :- parent(Z,X), parent(Z,Y), female(X),X\==Y.
brother(X,Y) :- parent(Z,X), parent(Z,Y), male(X),X\==Y.
grandfather(X,Z) :- father(X,Y), parent(Y,Z).
grandmother(X,Z) :- mother(X,Y), parent(Y,Z).
siblings(X,Y) :- (brother(X,Y);sister(X,Y)),X\==Y.
uncle(X,Y) :- parent(Z,Y),brother(X,Z).
aunty(X,Y) :- parent(Z,Y),sister(X,Z).
cousin(X,Y) :- parent(A,X),parent(B,Y),siblings(A,B).
```

Output:-

SWI-Prolog (AMD64, Multi-threaded, version 9.0.4)

File Edit Settings Run Debug Help

```
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/admin/Documents/Prolog/Practical 10B Family Tree.pl compiled 0.00 sec, 26 clauses
?- grandfather(X,Y).
X = adam,
Y = john .

?- grandmother(X,Y).
X = eve,
Y = john .

?- father(X,Y).
X = adam,
Y = george ;
X = adam,
Y = steffi ;
X = george,
Y = john ;
X = steve,
Y = alice.

?- cousin(X,Y).
X = john,
Y = alice ;
X = john,
Y = alice ;
X = alice,
Y = john ;
X = alice,
Y = john ;
false.

?- cousin(X,Y).
X = john,
Y = alice ;
X = john,
Y = alice ;
X = alice,
Y = john ;
X = alice,
Y = john ;
false.

?- uncle(X,Y).
X = george,
Y = alice ;
X = george,
Y = alice ;
false.

?- father(adam,_).
true .

?- father(adam,X).
X = george ;
X = steffi.

?-
```