**EXPERIMENT NO.1**

**Q.1]Write a Program to Implement Breadth First Search**.

graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : ['F'],

'D' : [],

'E' : ['F'], 'F' : []

}

visited = [] # List to keep track of visited nodes. queue = [] #Initialize a queue

def bfs(visited, graph, node):

visited.append(node) queue.append(node) while queue:

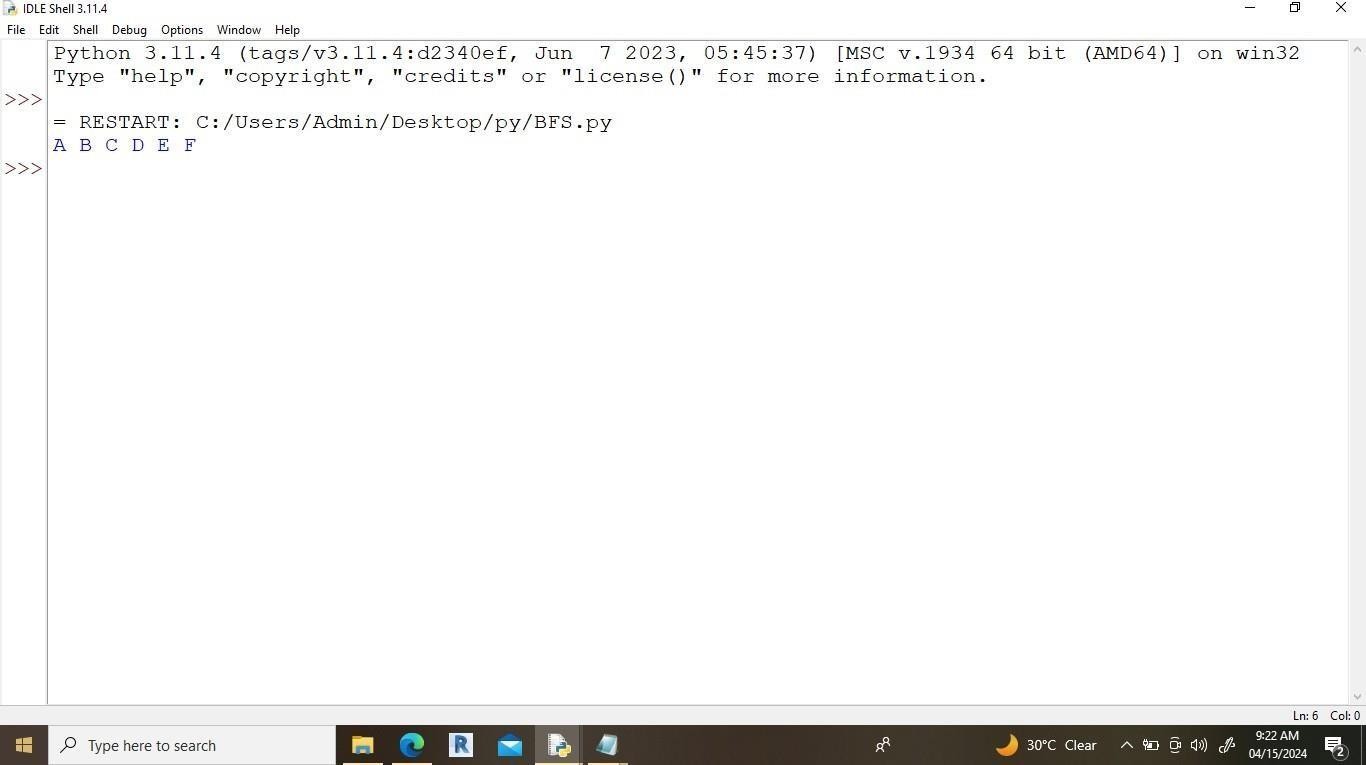
s = queue.pop(0) print (s, end = " ")

for neighbour in graph[s]:

if neighbour not in visited: visited.append(neighbour) queue.append(neighbour)

# Driver Code bfs(visited, graph, 'A')

## Output-



**EXPERIMENT NO.2**

**Q.2]Write a Program to Implement Depth First Search using Python**.

# Using a Python dictionary to act as an adjacency list graph = {

'A' : ['B','C'],

'B' : ['D', 'E'],

'C' : ['F'],

'D' : [],

'E' : ['F'], 'F' : []

}

visited = set() # Set to keep track of visited nodes. def dfs(visited, graph, node):

if node not in visited: print (node) visited.add(node)

for neighbour in graph[node]: dfs(visited, graph, neighbour)

# Driver Code dfs(visited, graph, 'A')

## Output-



**EXPERIMENT NO.3**

**Q.3]Write a Program to Implement Tic-Tac-Toe game using Python.**

# Tic-Tac-Toe Program using # random number in Python

# importing all necessary libraries

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 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:

win = True if win:

return win

for x in range(len(board)): y = len(board) - 1 - x if board[x, y] != player:

win = False

return win

# 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

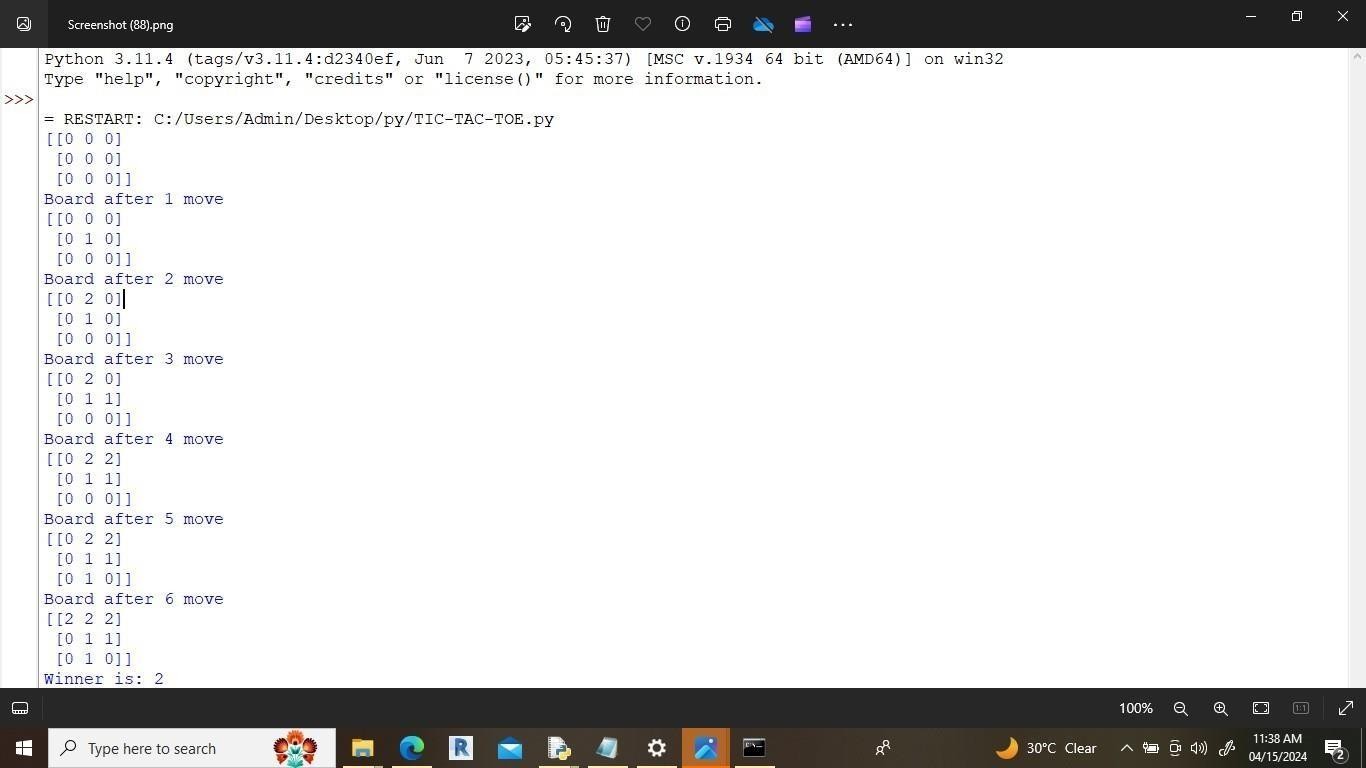
winner = evaluate(board) if winner != 0:

break return(winner)

# Driver Code

print("Winner is: " + str(play\_game()))

# Output-



**EXPERIMENT NO.4**

**Q.4] Write a Program to Implement 8-Puzzle problem using Python.**

class Solution:

def solve(self, board): dict = {}

flatten = []

for i in range(len(board)):

flatten += board[i] flatten = tuple(flatten) dict[flatten] = 0

if flatten == (0, 1, 2, 3, 4, 5, 6, 7, 8):

return 0

return self.get\_paths(dict) def get\_paths(self, dict):

cnt = 0 while True:

current\_nodes = [x for x in dict if dict[x] == cnt] if len(current\_nodes) == 0:

return -1

for node in current\_nodes: next\_moves = self.find\_next(node) for move in next\_moves:

if move not in dict: dict[move] = cnt + 1

if move == (0, 1, 2, 3, 4, 5, 6, 7, 8):

return cnt + 1

cnt += 1

def find\_next(self, node): moves = {

|  |  |  |  |
| --- | --- | --- | --- |
| 0: | [1, | 3], | |
| 1: | [0, | 2, | 4], |
| 2: | [1, | 5], | |
| 3: | [0, | 4, | 6], |
| 4: | [1, | 3, | 5, 7], |
| 5: | [2, | 4, | 8], |
| 6: | [3, | 7], | |
| 7: | [4, | 6, | 8], |
| 8: | [5, | 7], |  |

}

results = []

pos\_0 = node.index(0) for move in moves[pos\_0]:

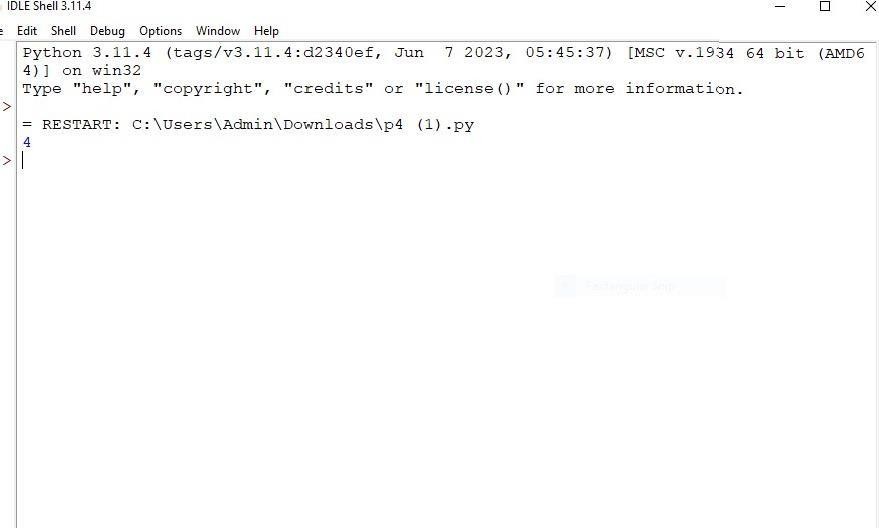
new\_node = list(node)

new\_node[move], new\_node[pos\_0] = new\_node[pos\_0], new\_node[move] results.append(tuple(new\_node))

return results

ob = Solution() matrix = [

### Class: MCA II Year



**Subject: AI & Applications Lab**

|  |  |  |
| --- | --- | --- |
| [3, | 1, | 2], |
| [4, | 7, | 5], |
| [6,  ] | 8, | 0] |

print(ob.solve(matrix))

# Output-

### Name: Piusha Uttam Magdum Div: A Roll No. 96

**EXPERIMENT NO.5**

### Q.5] Write a Program to Implement Water-Jug problem using Python.

# This function is used to initialize the # dictionary elements with a default value.

from collections import defaultdict

# jug1 and jug2 contain the value jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with # default value as false.

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

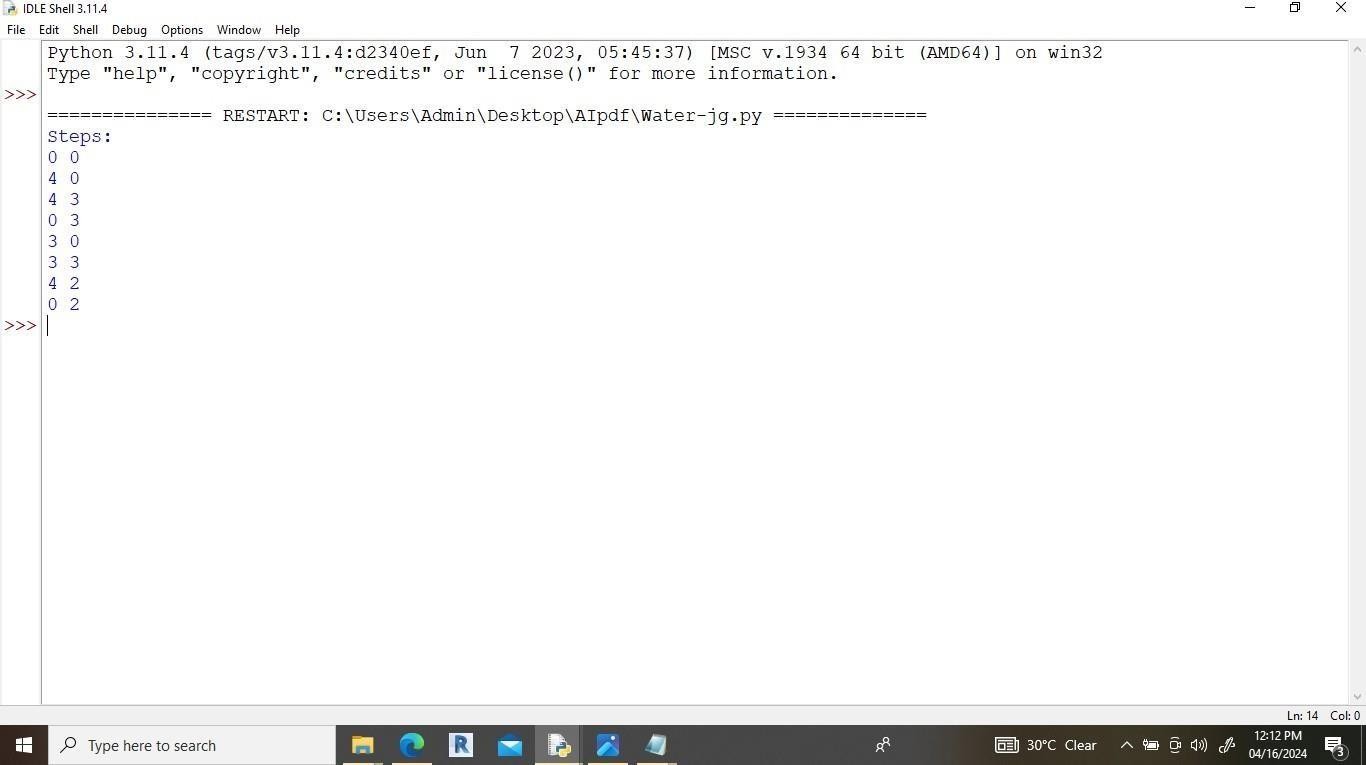
waterJugSolver(amt1, 0) or waterJugSolver(jug1, amt2) or waterJugSolver(amt1, jug2) or waterJugSolver(amt1 + min(amt2, (jug1-amt1)), amt2 - min(amt2, (jug1-amt1))) or waterJugSolver(amt1 - min(amt1, (jug2-amt2)), amt2 + min(amt1, (jug2-amt2))))

else:

return False

print("Steps: ") waterJugSolver(0, 0)

# Output-



**EXPERIMENT NO.6**

**Q.6] Write a Program to Implement Travelling Salesman Problem using Python.**

# Python3 implementation of the approach V = 4

answer = []

# Function to find the minimum weight # Hamiltonian Cycle

def tsp(graph, v, currPos, n, count, cost): # If last node is reached and it has

# a link to the starting node i.e

# the source then keep the minimum # value out of the total cost of # traversal and "ans"

# Finally return to check for # more possible values

if (count == n and graph[currPos][0]): answer.append(cost + graph[currPos][0]) return

# BACKTRACKING STEP

# Loop to traverse the adjacency list

# of currPos node and increasing the count # by 1 and cost by graph[currPos][i] value for i in range(n):

if (v[i] == False and graph[currPos][i]): # Mark as visited

v[i] = True

tsp(graph, v, i, n, count + 1, cost + graph[currPos][i])

# Mark ith node as unvisited v[i] = False

# Driver code

# n is the number of nodes i.e. V if name == ' main ':

n = 4

graph= [[0,10,15,20],

|  |  |  |  |
| --- | --- | --- | --- |
| [ | 10, | 0, 35, 25 | ], |
| [ | 15, | 35, 0, 30 | ], |
| [ | 20, | 25, 30, 0 | ]] |

# Boolean array to check if a node # has been visited or not

v = [False for i in range(n)] # Mark 0th node as visited

v[0] = True

# Find the minimum weight Hamiltonian Cycle tsp(graph, v, 0, n, 1, 0)

# ans is the minimum weight Hamiltonian Cycle print(min(answer))

# Output-



#### EXPERIMENT N0.7

Q.7] Write a Program to Implement Tower of Hanoi using Python. # Recursive Python function to solve the tower of hanoi

def TowerOfHanoi(n , source, destination, auxiliary): if n==1:

print ("Move disk 1 from source",source,"to destination",destination) return

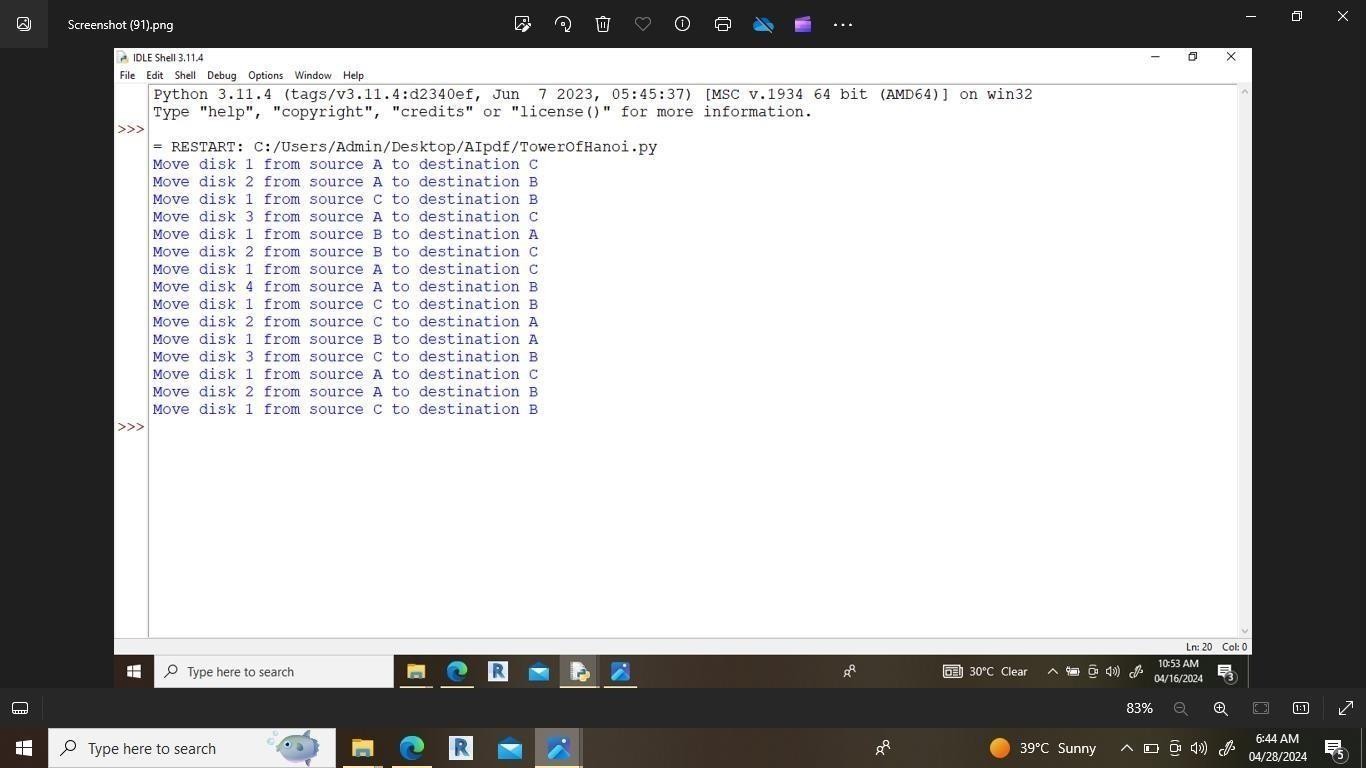
TowerOfHanoi(n-1, source, auxiliary, destination)

print ("Move disk",n,"from source",source,"to destination",destination) TowerOfHanoi(n-1, auxiliary, destination, source)

# Driver code n = 4

TowerOfHanoi(n,'A','B','C') #A, C, B are the name of rods

# Output-



**EXPERIMENT N0.8**

**Q.8] Write a Program to Implement Monkey Banana Problem using Python.'''**

i += 1

print('Step:', i, 'Monkey moves from', x, 'to', y)

def Monkey\_move\_box(x, y): global i

i += 1

print('Step:', i, 'Monkey takes the box from', x, 'to', y)

def Monkey\_on\_box(): global i

i += 1

print('Step:', i, 'Monkey climbs onto the box')

def Monkey\_get\_banana(): global i

i += 1

print('Step:', i, 'Monkey picks the banana')

# Read the input operating parameters

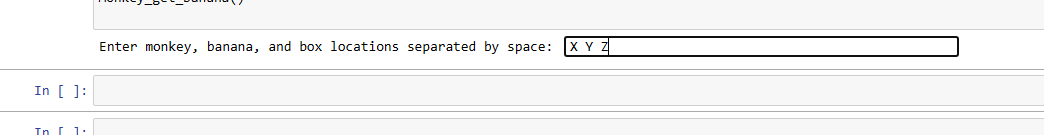
codeIn = input('Enter monkey, banana, and box locations separated by space: ') codeInList = codeIn.split()

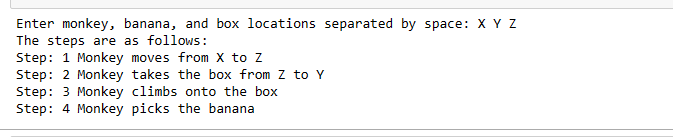
monkey, banana, box = codeInList[0], codeInList[1], codeInList[2] print('The steps are as follows:')

# Perform the monkey picking banana task Monkey\_go\_box(monkey, box) Monkey\_move\_box(box, banana) Monkey\_on\_box()

Monkey\_get\_banana()

# Output-





#### EXPERIMENT NO.9

Q.9] Write a Program to Implement Alpha-Beta Pruning using Python.

# working of Alpha-Beta Pruning

# Initial values of Alpha and Beta MAX, MIN = 1000, -1000

# Returns optimal value for current player #(Initially called for root and maximizer)

def minimax(depth, nodeIndex, maximizingPlayer,

values, alpha, beta): # Terminating condition. i.e # leaf node is reached

if depth == 3:

return values[nodeIndex] if maximizingPlayer:

best = MIN

# Recur for left and right children 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

# Recur for left and # right children

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

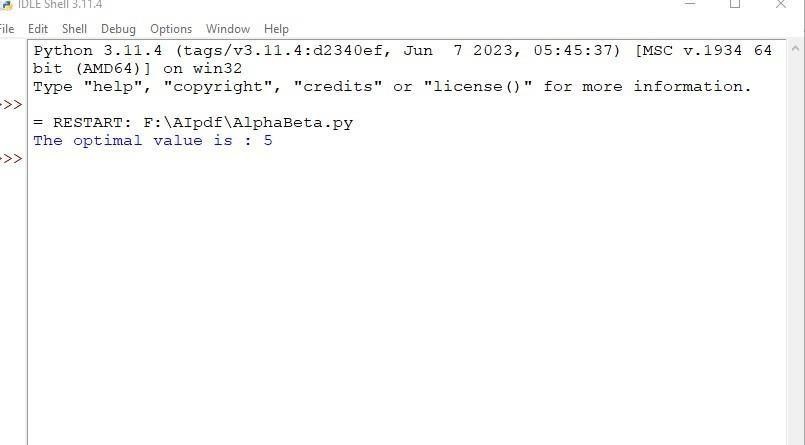
return best # Driver Code

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-



**EXPERIMENT NO.10**

Q.10] **Write a Program to Implement 8-Queens Problem using Python.**

# Python program to solve N Queen problem global N

N = 4

def printSolution(board): for i in range(N):

for j in range(N): print (board[i][j]),

print

def isSafe(board, row, col):

# Check this row on left side for i in range(col):

if board[row][i] == 1: return False

# Check upper diagonal on left side

for i, j in zip(range(row, -1, -1), range(col, -1, -1)): if board[i][j] == 1:

return False

# Check lower diagonal on left side

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):

# base case: If all queens are placed # then return true

if col >= N:

return True for i in range(N):

if isSafe(board, i, col):

#Place this queen in board[i][col] board[i][col] = 1

# recur to place rest of the queens

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

# driver program to test above function solveNQ()

# Output-

