**MLA0101- ARTIFICIAL INTELLIGENCE AND EXPERT SYSTEMS FOR ENGINEERING APPLICATIONS**

**EX.NO:**1 **TITLE**: Python program to solve Water Jug Problem

**INPUT:**

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))))

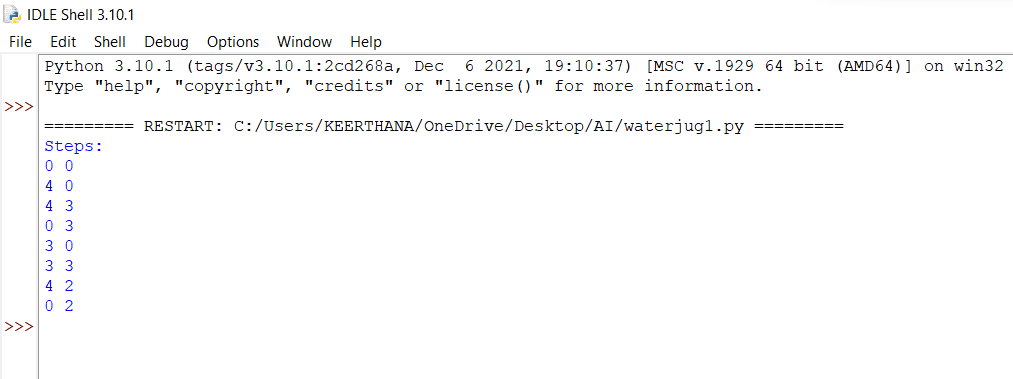
else:

return False

print("Steps: ")

waterJugSolver(0, 0)

**OUTPUT:**



**EX NO:** 2 **TITLE:** Python program for Breadth First Search

**INPUT:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = [] # List for visited nodes.

queue = [] #Initialize a queue

def bfs(visited, graph, node): #function for BFS

visited.append(node)

queue.append(node)

while queue: # Creating loop to visit each node

m = queue.pop(0)

print (m, end = " ")

for neighbour in graph[m]:

if neighbour not in visited:

visited.append(neighbour)

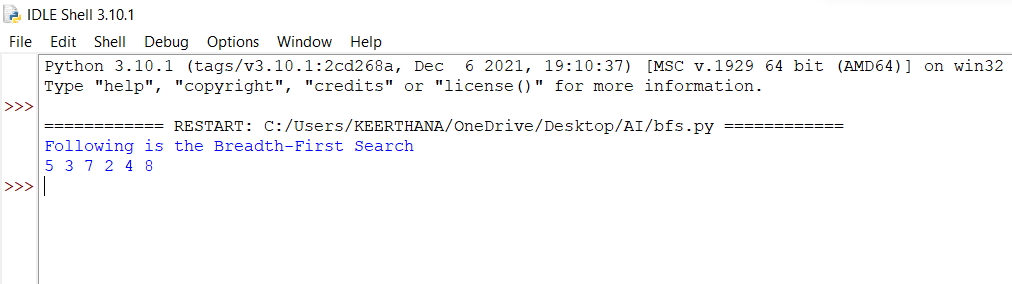
queue.append(neighbour)

# Driver Code

print("Following is the Breadth-First Search")

bfs(visited, graph, '5') # function calling

**OUTPUT:**



**EX NO:**3 **TITLE:** Python program for Depth First Search

**INPUT:**

graph = {

'5' : ['3','7'],

'3' : ['2', '4'],

'7' : ['8'],

'2' : [],

'4' : ['8'],

'8' : []

}

visited = set() # Set to keep track of visited nodes of graph.

def dfs(visited, graph, node): #function for dfs

if node not in visited:

print (node)

visited.add(node)

for neighbour in graph[node]:

dfs(visited, graph, neighbour)

# Driver Code

print("Following is the Depth-First Search")

dfs(visited, graph, '5')

**OUTPUT:**

****

**EX NO:** 4 **TITLE:** Python program to solve Cryptarithmetic Puzzles

**INPUT:**

def isCryptSolution(crypt, solution):

newsol = list(zip(\*reversed(solution)))

newstring1 = ''

total = 0

for word in range(len(crypt)-1):

subtotal, sol\_total = 0, 0

newstring = ''

for char in crypt[word]:

idx = newsol[0].index(char)

newstring = newstring + newsol[1][idx]

subtotal = int(newstring)

# if newstring[0] == '0':

# return False

total = total + subtotal

for char1 in crypt[-1]:

nidx = newsol[0].index(char1)

newstring1 = newstring1 + newsol[1][nidx]

sol\_total = int(newstring1)

if total == sol\_total and newstring[0] != '0':

return print('True')

elif total == 0 and newstring[0] == '0' and len(newstring) == 1:

return print('True')

else:

return print('False')

crypt = ["SEND", "MORE", "MONEY"]

solution = [['O', '0'],

['M', '1'],

['Y', '2'],

['E', '5'],

['N', '6'],

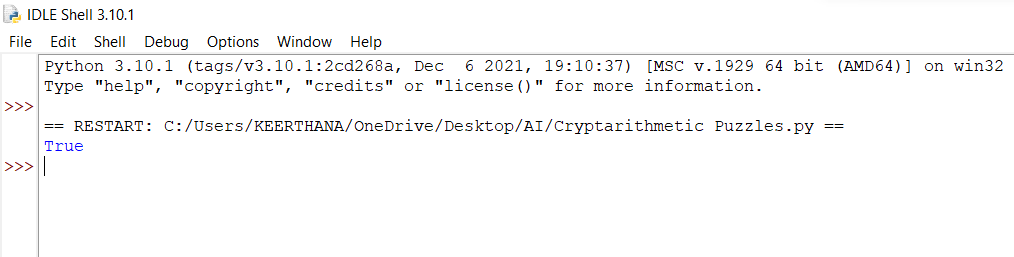
['D', '7'],

['R', '8'],

['S', '9']]

isCryptSolution(crypt, solution)

**OUTPUT:**

****

**EX NO**:5 **TITLE:** Python program to implement Minimax Algorithm

**INPUT:**

import math

def minimax (curDepth, nodeIndex,

maxTurn, scores,

targetDepth):

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]

treeDepth = math.log(len(scores), 2)

print("The optimal value is : ", end = "")

print(minimax(0, 0, True, scores, treeDepth))

**OUTPUT:**

