**1.Write a program to demonstrate Water jug problem**

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 (amt 1,

(jug2-amt2)))

else:

return False

Print ("Steps: ")

waterJugSolver(0,0)

**2.given an undirected graph, the task is to print all the connected components line by line (DFS)**

class Graph:

def \_\_init\_\_ (self, vertices):

Self, V= vertice S

Self graph defaultdict (list)

def add Edge (self, u.v):

self. graph [u].append (v)

def is Reachable (self, sid):

visited [False] \*(self.v)

queue = []

queue.append (s)

visited [s]=True

while queue:

n = queue.pop()

if n==d:

retuen True

for i in self. graph [n]:

if visited [i] == False:

queue.append (1)

visited [i] = True

return false

9=Graph (4)

g.addEdge [u:0, 0:1)

g.add Edge [v:0.0:2)

g.addEdge [u:1:2)

g.addEdge [u: 2,0:0)

g.add Edge [u: 2,v:3)

goodd Edge [v:s. v.3)

U= 1

V=3

If g.is Reachable (u.v):

Print ("There is apath from %d to 1.8" % (0.0))

else:

Print ("There is no path from "%d to %d" % (uv))

**3.check if all nodes of undirected graph can be visited from given Node**

def Canvisit all Nodes (arr, x,n):

q=[]

visited = [False]\*n

q.append (x)

visited [x] = True

Count=0

while (len (q)>0):

size=len(q)

for i in range (size):

curr= q.pop(0)

Count = count + 1

for i in arr[curr]:

If (visited [j] == False):

q.append c)

visited [j] = True

If (count == n):

return false

arr = [[1,2], [0,3,2], [0,1],[1]]

N=5

X=0

If (can visit All Nodes (arr, X, N)):

Print ("YES")

Else:

print("No")

**4. find the level of given node in an undirected graph,(BFS)**

def find\_level(edges, x):

max\_vertex = 0

for it in edges:

max\_vertex = max(max\_vertex, max(it[0], it[1]))

adj = [[] for \_ in range(max\_vertex + 1)]

for i in range(len(edges)):

adj[edges[i][0]].append(edges[i][1])

adj[edges[i][1]].append(edges[i][0])

if x > max\_vertex or len(adj[x]) == 0:

return -1

q = deque()

q.append(0)

level = 0

visited = [0] \* (max\_vertex + 1)

visited[0] = 1

while len(q) > 0:

sz = len(q)

while sz > 0:

current\_node = q.pop(0)

if current\_node == x:

return level

for it in adj[current\_node]:

if not visited[it]:

q.append(it)

visited[it] = 1

sz -= 1

level += 1

return -1

v=5

edge=[[0,1],[0,2],[1,3],[2,4]]

x=3

level=find level(v,edge,x)

print(level)

**5. Q-queen code**

Print ("enter the number of queeng :")

N=int (input())

board = [[OJXN for-in range crosy

def. is attack (I,j):

for k in range (o,n):

if board [][K]==1 or board [K][j]==1:

return True

for k in range coin)

If board [!][K] == 1 or boord [K][J]==1:

return True

for k in range (o,N):

for i in range (O,N):

if(k+1==i+j) or (K-1==i-j):

If board [K][l]==1:

return True.

return False

def N-queen (n):

if n==0:

return true

for i in range (o,N):

if (not (is-attack (i,j))and (board [i][j]!=1):

board [i][j]=1

If N-queen (n-1)== True?

return True

board [i][j]=0

return false

N-queen (n)

for I in board:

Print (i)