In [1]:

```python
# Breadth First Traversal for a Graph
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def BFS(self, s):
        visited = [False] * (max(self.graph) + 1)
        queue = []
        queue.append(s)
        visited[s] = True

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

            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print ("Following is Breadth First Traversal" " (starting from vertex 2)")
g.BFS(2)
```

```
Following is Breadth First Traversal (starting from vertex 2)
2 0 3 1
```

In [2]:

```python
# Depth First Traversal for a Graph
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

```
Following is DFS from (starting from vertex 2)
2 0 1 3
```

In [3]:

```python
# Count the number of nodes at given level in a tree using BFS
from collections import deque

adj = [[] for i in range(1001)]

def addEdge(v, w):
    adj[v].append(w)
    adj[w].append(v)

def BFS(s, l):
    V = 100
    visited = [False] * V
    level = [0] * V

    for i in range(V):
        visited[i] = False
        level[i] = 0

    queue = deque()
    visited[s] = True
    queue.append(s)
    level[s] = 0

    while (len(queue) > 0):
        s = queue.popleft()
        for i in adj[s]:
            if (not visited[i]):
                level[i] = level[s] + 1
                visited[i] = True
                queue.append(i)

    count = 0
    for i in range(V):
        if (level[i] == l):
            count += 1
    return count
if __name__ == '__main__':

    addEdge(0, 1)
    addEdge(0, 2)
    addEdge(1, 3)
    addEdge(2, 4)
    addEdge(2, 5)

    level = 2
    print(BFS(0, level))
```

```
3
```

In [4]:

```python
# Count number of trees in a forest
def addEdge(adj, u, v):
    adj[u].append(v)
    adj[v].append(u)

def DFSUtil(u, adj, visited):
    visited[u] = True
    for i in range(len(adj[u])):
        if (visited[adj[u][i]] == False):
            DFSUtil(adj[u][i], adj, visited)

def countTrees(adj, V):
    visited = [False] * V
    res = 0
    for u in range(V):
        if (visited[u] == False):
            DFSUtil(u, adj, visited)
            res += 1
    return res
if __name__ == '__main__':

    V = 5
    adj = [[] for i in range(V)]
    addEdge(adj, 0, 1)
    addEdge(adj, 0, 2)
    addEdge(adj, 3, 4)
    print(countTrees(adj, V))
```

```
2
```

In [6]:

```python
# Detect Cycle in a Directed Graph
from collections import defaultdict

class Graph():
    def __init__(self,vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self,u,v):
        self.graph[u].append(v)

    def isCyclicUtil(self, v, visited, recStack):
        visited[v] = True
        recStack[v] = True

        for neighbour in self.graph[v]:
            if visited[neighbour] == False:
                if self.isCyclicUtil(neighbour, visited, recStack) == True:
                    return True
            elif recStack[neighbour] == True:
                return True

        recStack[v] = False
        return False

    def isCyclic(self):
        visited = [False] * (self.V + 1)
        recStack = [False] * (self.V + 1)
        for node in range(self.V):
            if visited[node] == False:
                if self.isCyclicUtil(node,visited,recStack) == True:
                    return True
        return False
g = Graph(4)
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)

if g.isCyclic() == 1:
    print("Graph has  cycle")
else:
    print("Graph has no cycle")
```

```
Graph has  cycle
```

In [7]:

```python
# Implement n-Queen's Problem
global N
N = 4

def printSolution(board):
    for i in range(N):
        for j in range(N):
            print (board[i][j], 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
solveNQ()
```

```
0 0 1 0
1 0 0 0
0 0 0 1
0 1 0 0
```

Out[7]:

```
True
```

In [ ]: