**Depth First Search:**

DFS1:

#DFS non recursive

def dfs\_non\_recursive(graph, source):

    if source is None or source not in graph:

        return 'Invalid Input'

    path = []

    stack = [source]

    while (len(stack) != 0):

        s = stack.pop()

        if s not in path:

            path.append(s)

        if s not in graph:

            continue

        for neighbor in graph[s]:

            stack.append(neighbor)

    return ' '.join(path)

def main():

    graph = {

        'A': ['D', 'C', 'B'],

        'B': ['E'],

        'C': ['F', 'G'], #The example has G, F but that seems wrong.

        'D': ['H'],

        'E': ['I'],

        'F': ['J']

    }

    path = dfs\_non\_recursive(graph, 'A')

    print(path)

main()



DFS2:

#DFS Recursive

def dfs\_recursive(graph, source, path = []):

    if source not in path:

        path.append(source)

        if source not in graph:

            return path

        for neighbor in graph[source]:

            path = dfs\_recursive(graph, neighbor, path)

    return path

def main():

    graph = {

        'A': ['D', 'C', 'B'],

        'B': ['E'],

        'C': ['F', 'G'], #The example has G, F but that seems wrong.

        'D': ['H'],

        'E': ['I'],

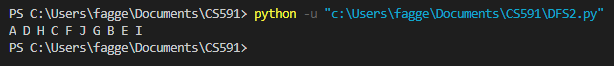
        'F': ['J']

    }

    path = dfs\_recursive(graph, 'A')

    print(\*path)

main()



DFS3:

#Binary Tree DFS

class Node:

    def \_\_init\_\_(self, value):

        self.value = value

        self.left = None

        self.right = None

class Tree:

    def \_\_init\_\_(self, value):

        self.root = Node(value)

    def insert(self, value):

        current = self.root

        while True:

            if value > current.value:

                if current.right is None:

                    current.right = Node(value)

                    break

                else:

                    current = current.right

            elif value < current.value:

                if current.left is None:

                    current.left = Node(value)

                    break

                else:

                    current = current.left

            else:

                current.value = value

                break

    def DFSTree(self, node: Node):

        if node is None:

            return

        else:

            print(node.value, end = ' ')

        self.DFSTree(node.left)

        self.DFSTree(node.right)

    def PrintDFSTree(self):

        self.DFSTree(self.root)

def main():

    root = Tree(7)

    root.insert(2)

    root.insert(25)

    root.insert(9)

    root.insert(80)

    root.insert(0)

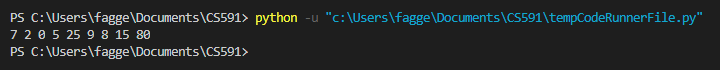
    root.insert(5)

    root.insert(15)

    root.insert(8)

    root.PrintDFSTree()

main()



DFS4:

#DFS networkx and matplot lib

import networkx as nx

import matplotlib.pyplot as plt

G = nx.Graph()

G.add\_node(1)

G.add\_node(2)

G.add\_node(3)

G.add\_node(4)

G.add\_node(5)

G.add\_nodes\_from([6, 7, 8, 9])

G.add\_edge(5, 8)

G.add\_edge(5, 4)

G.add\_edge(5, 7)

G.add\_edge(8, 2)

G.add\_edge(4, 3)

G.add\_edge(4, 1)

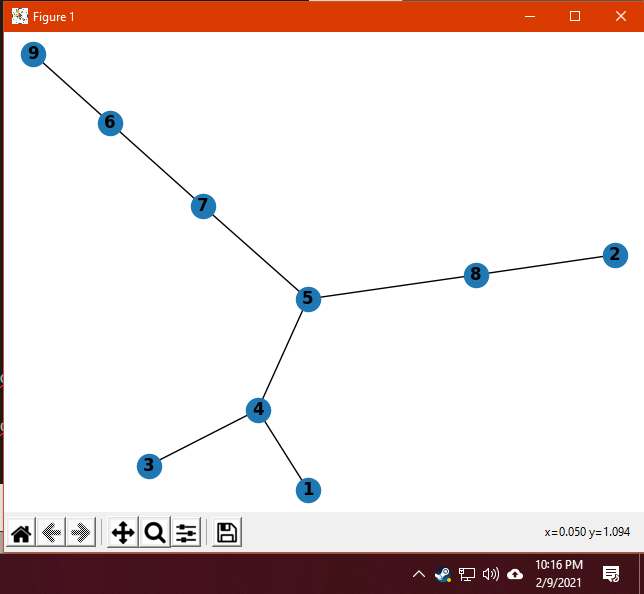
G.add\_edge(7, 6)

G.add\_edge(6, 9)

nx.draw(G, with\_labels = True, font\_weight = 'bold')

output = list(nx.dfs\_preorder\_nodes(G, source = 5))

plt.show()



DFS5:

#networkx digraph

import networkx as nx

import matplotlib.pyplot as plt

def dfs(dag, start, visited, stack):

       if start in visited:

           # node and all its branches have been visited

           return stack, visited

       if dag.out\_degree(start) == 0:

           # if leaf node, push and backtrack

           stack.append(start)

           visited.append(start)

           return stack, visited

       #traverse all the branches

       for node in dag.neighbors(start):

           if node in visited:

               continue

           stack, visited = dfs(dag, node, visited, stack)

       #now, push the node if not already visited

       if start not in visited:

           print("pushing %s"%start)

           stack.append(start)

           visited.append(start)

       return stack, visited

def topological\_sort\_using\_dfs(dag):

       visited = []

       stack=[]

       start\_nodes = [i for i in dag.nodes if dag.in\_degree(i)==0]

   #     print(start\_nodes)

       for s in start\_nodes:

           stack, visited = dfs(dag, s, visited, stack)

       print("Topological sorted:")

       while(len(stack)!=0):

           print(stack.pop(), end=" ")

dag = nx.digraph.DiGraph()

dag.add\_nodes\_from(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'])

dag.add\_edges\_from([('A', 'B'), ('A', 'E'), ('B', 'D'), ('E', 'C'),

                    ('D', 'G'), ('C', 'G'), ('C', 'I'), ('F', 'I')])

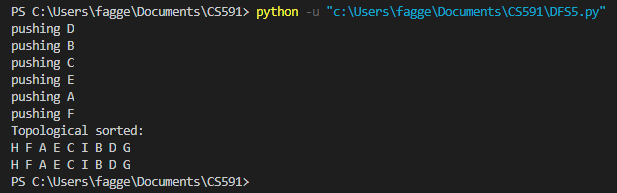
topological\_sort\_using\_dfs(dag)

topological\_sorting = nx.topological\_sort(dag)

print()

for n in topological\_sorting:

    print(n, end=' ')



DFS6:

#Connected components DFS

import networkx as nx

import matplotlib.pyplot as plt

def dfs\_traversal(graph, start, visited, path):

        if start in visited:

            return visited, path

        visited.append(start)

        path.append(start)

        for node in graph.neighbors(start):

            visited, path = dfs\_traversal(graph, node, visited, path)

        return visited, path

def find\_connected\_components(graph):

    visited = []

    connected\_components = []

    for node in graph.nodes:

        if node not in visited:

            cc = []

            visited, cc = dfs\_traversal(graph, node, visited, cc)

            connected\_components.append(cc)

    return connected\_components

def main():

    graph = nx.Graph()

    graph.add\_nodes\_from(['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I'])

    graph.add\_edges\_from([('A', 'B'), ('B', 'E'), ('A', 'E')])

    graph.add\_edges\_from([('C', 'D'), ('D', 'H'), ('H', 'F'), ('F', 'C')])

    graph.add\_edge('G', 'I')

    nx.draw(graph, with\_labels =  True, font\_weight='bold')

    plt.show()

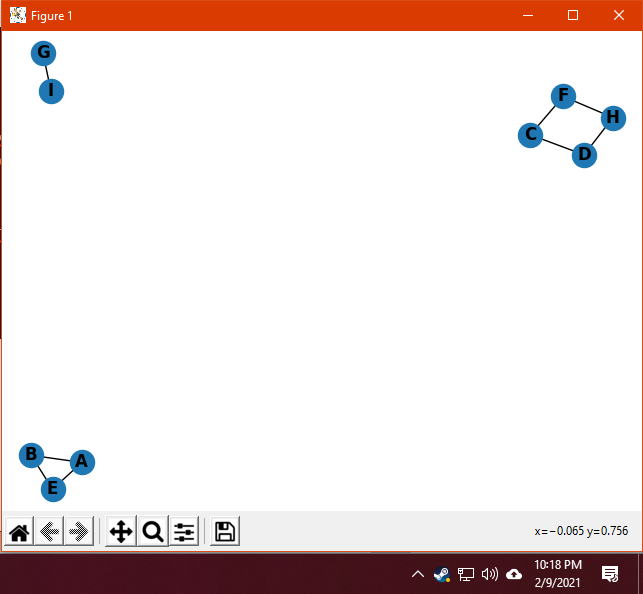
    connected\_components = find\_connected\_components(graph)

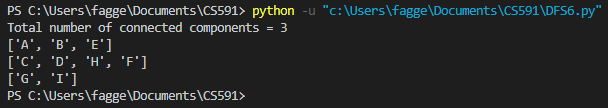
    print(f'Total number of connected components = {len(connected\_components)}')

    for cc in connected\_components:

        print(cc)

main()





**Breadth First Search:**

Iterative BFS:

#Iterative BFS

from collections import deque

class Graph:

    def \_\_init\_\_(self, edges, N):

        #A list of lists that represents an adjacency list

        self.adjList = [[] for \_ in range(N)]

        #This adds edges to the undirected graph

        for (src, dest) in edges:

            self.adjList[src].append(dest)

            self.adjList[dest].append(src)

#Perform BFS on the graph with starting point 'v'

def BFS(graph, v, discovered):

    q = deque() #queue for BFS

    discovered[v] = True #source vertex marked discovered

    #enqueue source vertex

    q.append(v)

    #loop till queue is empty

    while q:

        #dequeue front node and print it

        v = q.popleft()

        print(v, end=' ')

        #do for every edge `v -> `u

        for u in graph.adjList[v]:

            if not discovered[u]:

                #mark it as discovered and enqueue it

                discovered[u] = True

                q.append(u)

if \_\_name\_\_ == '\_\_main\_\_':

    edges = [

        (1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (5, 9),

        (5, 10), (4, 7), (4, 8), (7, 11), (7, 12)

        #vertexes 0, 13, and 14 are single nodes

    ]

    N = 15

    graph = Graph(edges, N)

    discovered = [False] \* N

    for i in range(N):

        if not discovered[i]:

            BFS(graph, i, discovered)



Recursive BFS:

#Recursive BFS Implementation

from collections import deque

class Graph:

    def \_\_init\_\_(self, edges, N):

        self.adjList = [[] for \_ in range(N)]

        #add edges to the undirected graph

        for (src, dest) in edges:

            self.adjList[src].append(dest)

            self.adjList[dest].append(src)

#Perform BFS recursively on the graph

def recursiveBFS(graph, q, discovered):

    if not q: return

    #dequeue front node and print it

    v = q.popleft()

    print(v, end=' ')

    #do for every edge `v -> `u

    for u in graph.adjList[v]:

        if not discovered[u]:

            #mark it as discovered and enqueue it

            discovered[u] = True

            q.append(u)

    recursiveBFS(graph, q, discovered)

if \_\_name\_\_ == '\_\_main\_\_':

    #List of Graph edges:

    edges = [

        (1, 2), (1, 3), (1, 4), (2, 5), (2, 6), (5, 9),

        (5, 10), (4, 7), (4, 8), (7, 11), (7, 12)

    ]

    N = 15

    #build a graph from edges

    graph = Graph(edges, N)

    discovered = [False] \* N

    q = deque()

    for i in range(N):

        if not discovered[i]:

            discovered[i] = True

            q.append(i)

            recursiveBFS(graph, q, discovered)

