**Graphs**

Depth first search

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

    # 1. Visit one vertex, go to one of its adjacent vertex and then repeat

    def DFS(self,s):

        visited = set()

        self.DFSUtil(s, visited)

    def DFSUtil(self, v, visited):

        if v not in visited:

            visited.add(v)

        print(v, end=" ")

        for i in self.graph[v]:

            if i not in visited:

                visited.add(i)

                self.DFSUtil(i, 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)

g.DFS(1)

Bread first search

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.list = defaultdict(list)

        # print(self.list)

    def addEdge(self,u,v):

        self.list[u].append(v)

    def BFS(self,start\_node):

        # initialize visited list

        visited = set()

        # For BFS, queue is used

        queue = []

        # 1. Mark the node as visited and add to queue

        queue.append(start\_node)

        visited.add(start\_node)

        while queue:

            s = queue.pop(0) # pop 2

            print(s, end=" ")

            # add vertices which are edges to start node(2)

            for i in self.list[s]:

                if i not in visited:

                    queue.append(i)

                    visited.add(i)

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)

g.BFS(2)

Detect cycle in directed graph using DFS

Detect cycle in directed graph using BFS

Detect cycle in undirected graph using DFS

from collections import defaultdict

class Graph:

    def \_\_init\_\_(self):

        self.graph = defaultdict(list)

    def addEdge(self,u,v):

        self.graph[u].append(v)

        self.graph[v].append(u)

    def hasCycle(self, node):

        visited = set()

        print(self.dfs(node, visited, -1))

    def dfs(self,node, visited, parent):

        visited.add(node)

        for neighbor in self.graph[node]:

            if neighbor not in visited:

                # visited.add(node)

                if self.dfs(neighbor, visited, node):

                    return True

            elif parent != neighbor:

                return True

        return False

        # print(visited)

g = Graph()

g.addEdge(1, 2)

g.addEdge(2, 3)

g.addEdge(3, 4)

g.addEdge(4, 5)

g.addEdge(5, 2)

g.hasCycle(1)

Topological sort

'''

1. a vertex is pushed to stack only when all of its adjacent vertices

(and their adjacent vertices and so on) are already in stack.

Time complexity: O(V+E)

Space complexity: O(V)

Algorithm:

1. Visit one vertex and then recursively visit its neighbors

2. Push the neighbor which has no more neighbors into the stack

3. Do it for all vertices

'''

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 topologicalUtil(self,v,visited,result):

        # set vertex as visited

        visited[v] = True

        # look for its adjacent vertices

        for i in self.graph[v]:

            if visited[i] == False:

                self.topologicalUtil(i,visited,result)

        result.insert(0,v)

    def topological\_sort(self):

        visited = [False] \* self.V

        result = []

        for i in range(self.V):

            if visited[i] == False:

                self.topologicalUtil(i,visited, result)

        print(result)

g= Graph(6)

g.addEdge(5, 2);

g.addEdge(5, 0);

g.addEdge(4, 0);

g.addEdge(4, 1);

g.addEdge(2, 3);

g.addEdge(3, 1);

g.topological\_sort()

Course schedule

Course schedule 2

332 Reconstruct itinerary

'''

Approach:

Difficulties faced:

Steps to resolve Difficulties:

Time complexity: O(E/2 \* log E/2)

Space complexity: O(V + 2E)

Algorithm:

1. Sort the neighbors of the graph in reverse order

2. Pop the neighbor and do DFS

3. Add the node to the result when its neighbor list is empty

4. Reverse the result list

'''

class Solution:

    def findItinerary(self, tickets: List[List[str]]) -> List[str]:

        from collections import defaultdict

        self.flightMap = defaultdict(list)

        for ticket in tickets:

            origin, dest = ticket[0], ticket[1]

            self.flightMap[origin].append(dest)

        for origin, itinerary in self.flightMap.items():

        # Note that we could have multiple identical flights, i.e. same origin and destination.

            itinerary.sort(reverse=True)

        self.result = []

        self.DFS('JFK')

        return self.result[::-1]

    def DFS(self, origin):

        destlist = self.flightMap[origin]

        while destlist:

            nextDest = destlist.pop()

            self.DFS(nextDest)

        self.result.append(origin)