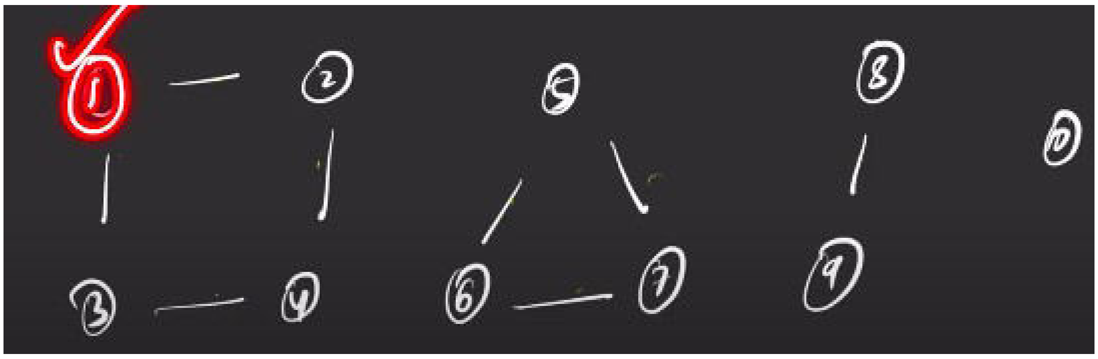
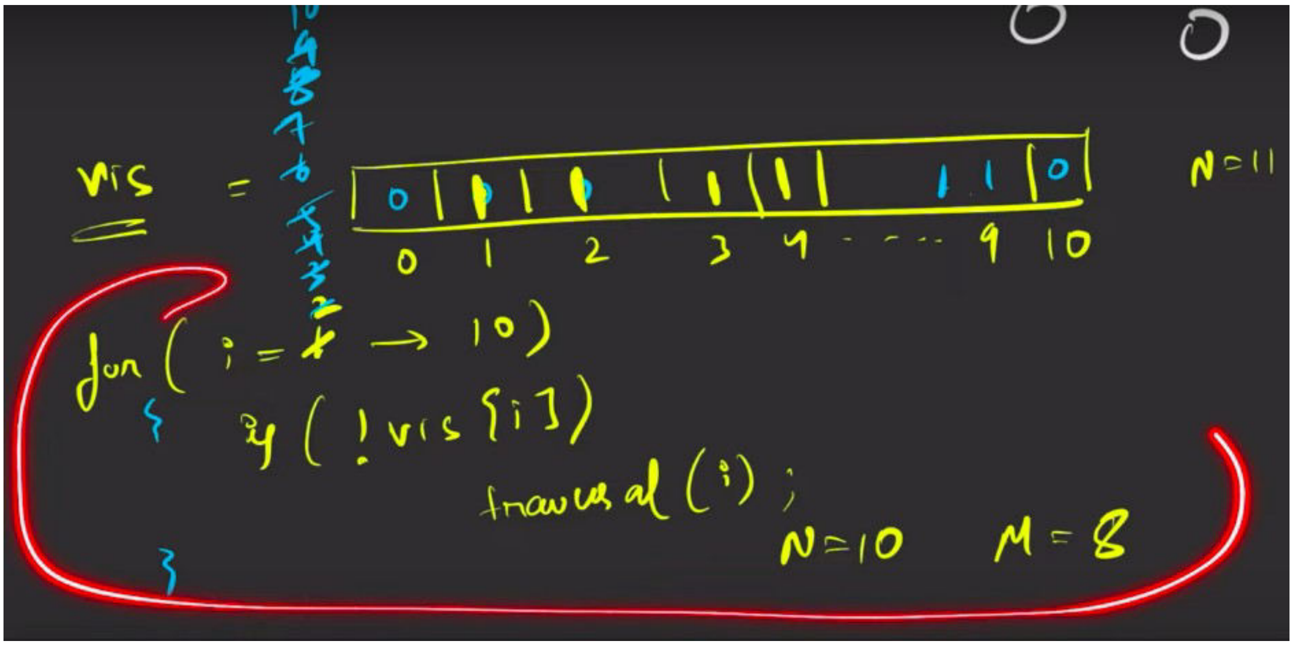
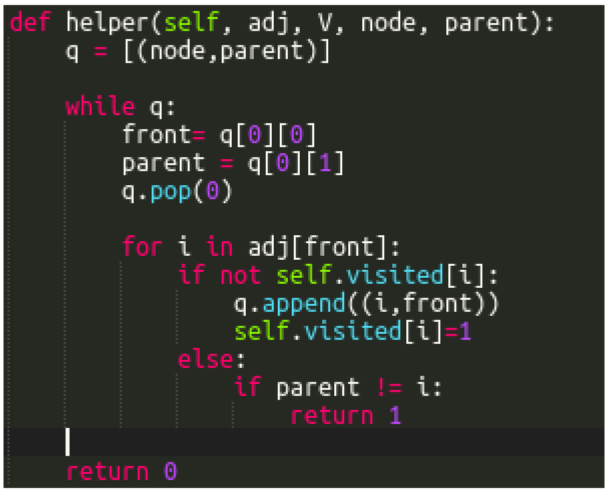
-> **Components of graph:** There can be multiple disconnected components in a graph.
-> Maintain a visited array, everytime start from an unvisited node and traverse that component.





Questions to practice:
1. flood fill
2. no. Of islands
3. rotten oranges

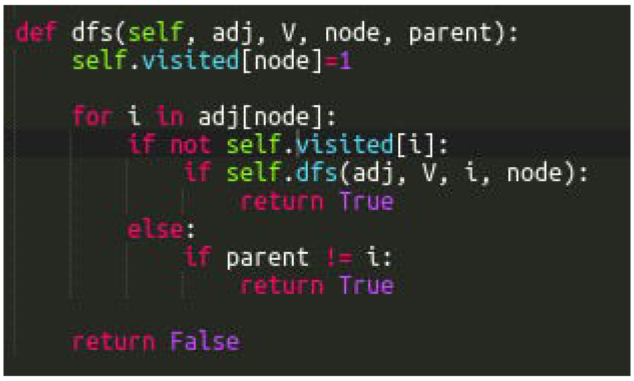
-> **Cycle detection in undirected graph:**
1. using bfs:
-> maintain a visited list, to track which node is visited already.
-> maintain a queue, in which element will be (node, parent_of_node).



```
def helper(self, adj, V, node, parent):
    q = [(node,parent)]

    while q:
        front= q[0][0]
        parent = q[0][1]
        q.pop(0)

        for i in adj[front]:
            if not self.visited[i]:
                q.append((i,front))
                self.visited[i]=1
            else:
                if parent != i:
                    return 1

    return 0
```

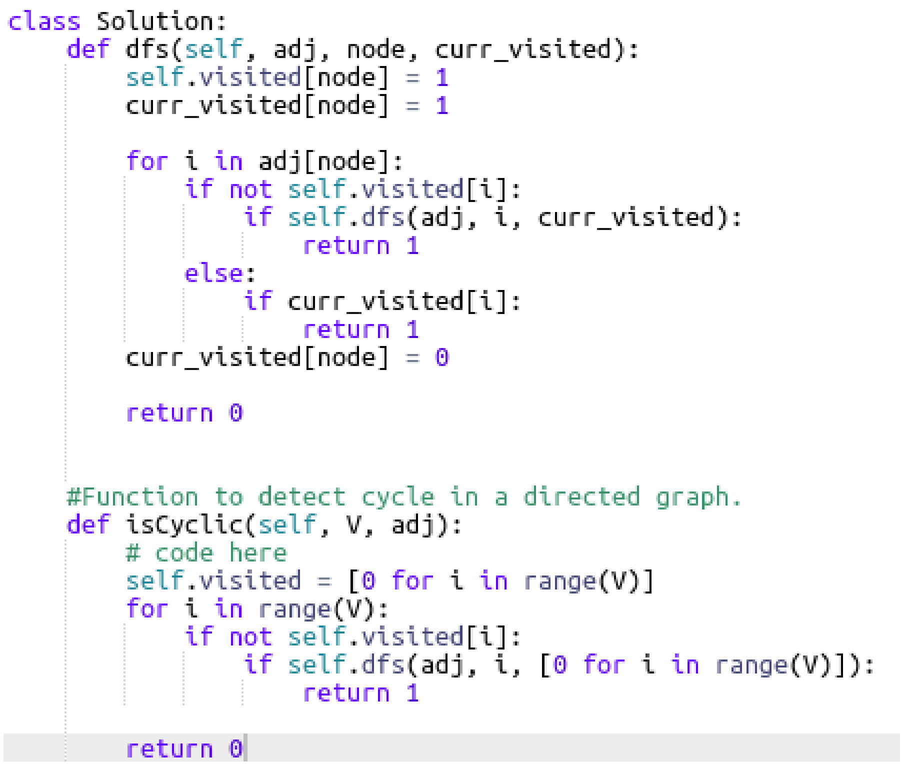
-> while bfs traversal, if we find any node is already visited, check parent_of_current_node is equal to visited node or not. If not, then cycle is present.

2. **Using dfs:**
-> maintain a visited list, to track which node is visited already.
-> in each dfs call pass (node, parent_of_node), if any adjacent node is already visited and this visited node isnot equal to parent of current node, it means there is a cycle, so return true.



```python
def dfs(self, adj, V, node, parent):
    self.visited[node]=1

    for i in adj[node]:
        if not self.visited[i]:
            if self.dfs(adj, V, i, node):
                return True
        else:
            if parent != i:
                return True

    return False
```

-> Cycle detection in directed graph:
1. Using dfs:
-> maintain a visited list, to track which node has been visited. Also, maintain a list current_path to store nodes which are being visited in dfs traversal (current path).
-> start dfs from any unvisited node, mark that node visited and check its neighbour nodes, if any neighbour node is visited and also present in current_path, that means this neighbour node is a part of some cycle.



```python
class Solution:
    def dfs(self, adj, node, curr_visited):
        self.visited[node] = 1
        curr_visited[node] = 1

        for i in adj[node]:
            if not self.visited[i]:
                if self.dfs(adj, i, curr_visited):
                    return 1
            else:
                if curr_visited[i]:
                    return 1
        curr_visited[node] = 0

        return 0


    #Function to detect cycle in a directed graph.
    def isCyclic(self, V, adj):
        # code here
        self.visited = [0 for i in range(V)]
        for i in range(V):
            if not self.visited[i]:
                if self.dfs(adj, i, [0 for i in range(V)]):
                    return 1

        return 0
```

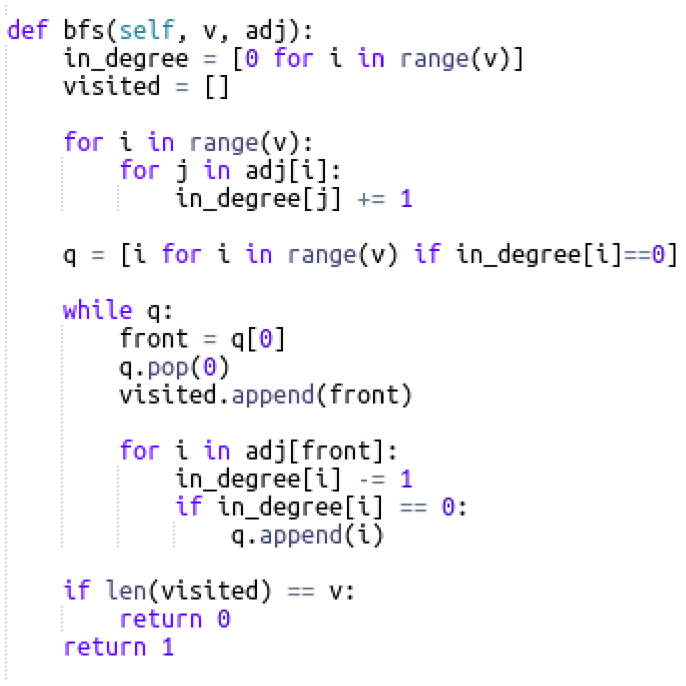
2. Using bfs:

NOTE: understand topo sort from below, then try to detect cycle using topo sort.

-> maintain an in-degree node list, and bfs traversal from that node whose in-degree is 0.

-> maintain a counter or visited list to store topo sort. Whenever a node is popped from queue, append it in visited list.

-> after all possible traversal, check if length of visited list is equal to number of nodes, if yes then no cycle, otherwise cycle is present.



```python
def bfs(self, v, adj):
    in_degree = [0 for i in range(v)]
    visited = []

    for i in range(v):
        for j in adj[i]:
            in_degree[j] += 1

    q = [i for i in range(v) if in_degree[i]==0]

    while q:
        front = q[0]
        q.pop(0)
        visited.append(front)

        for i in adj[front]:
            in_degree[i] -= 1
            if in_degree[i] == 0:
                q.append(i)

    if len(visited) == v:
        return 0
    return 1
```

-> **Topological sort**: (linear ordering of nodes such that if there is an edge from a->b, then a should appear before b in linear ordering), only applicable for graph not having cycle.

1. Using dfs:
   -> maintain a stack to store topo sort.
   -> maintain a visited array which contains whether node is visited or not (o or 1).Initially all nodes are unvisited.
   -> start traversing from any node, then visit neighbour nodes (if not visited already), when dfs call for any node gets completed, push it in stack.
   -> pop elements from stack, it is the resultant topo sort.

```
def dfs(self, v, adj, node):
    self.visited[node] = 1

    for i in adj[node]:
        if not self.visited[i]:
            self.dfs(v, adj, i)
            self.stk.append(i)

#Function to return list containing vertices
def topoSort(self, V, adj):
    # Code here
    # return self.bfs(V, adj)

    self.stk = []
    self.visited = [0 for i in range(V)]

    for i in range(V):
        if not self.visited[i]:
            self.dfs(V, adj, i)
            self.stk.append(i)

    return self.stk[::-1]
```

2. Using bfs:

   -> maintain a list specifying in-degree of each node in graph.

   -> maintain a queue which initially had nodes with in-degree 0.

   -> start bfs traversal from node taken out of queue, decrease in-degree count by 1, as in-degree of any node becomes 0, place that node in queue.

   -> take all elements from queue in order, represents topo sort.

```
def bfs(self, v, adj):
    in_degree = [0 for i in range(v)]
    topo_sort = []
    for i in range(v):
        for j in adj[i]:
            in_degree[j] += 1

    q = [i for i in range(v) if in_degree[i]==0]

    while q:
        front = q[0]
        q.pop(0)
        topo_sort.append(front)

        for i in adj[front]:
            in_degree[i] -= 1
            if in_degree[i] == 0:
                q.append(i)

    return topo_sort
```

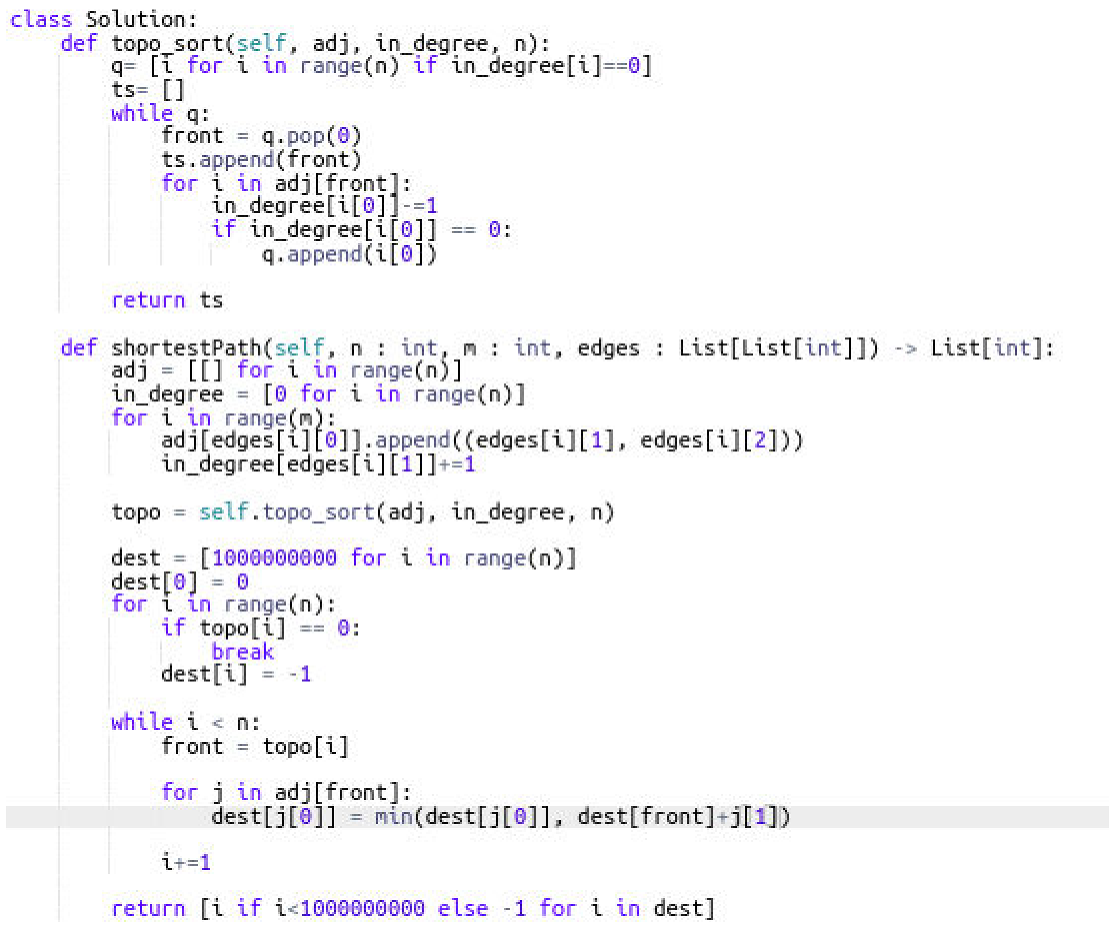
***** *NOTE: Practice*

-> alien dictionary (try to create DAG from given words using the letters and then use topo sort on it)

-> find eventual safe states (reverse given graph for bfs or use dfs maintain state of every node)

-> Shortest path in DAG using topo sort:

1. find the topo sort of the given DAG.
2. maintain a list *dest,* which stores minimum distance to reach ith node from any source node.
3. all the nodes before source node in topo sort will be set to -1 in dest as they can never be reached from source node.
4. start taking one node from topo sort (begin with src node) and update minimum distance of neighbour nodes in dest and so on.

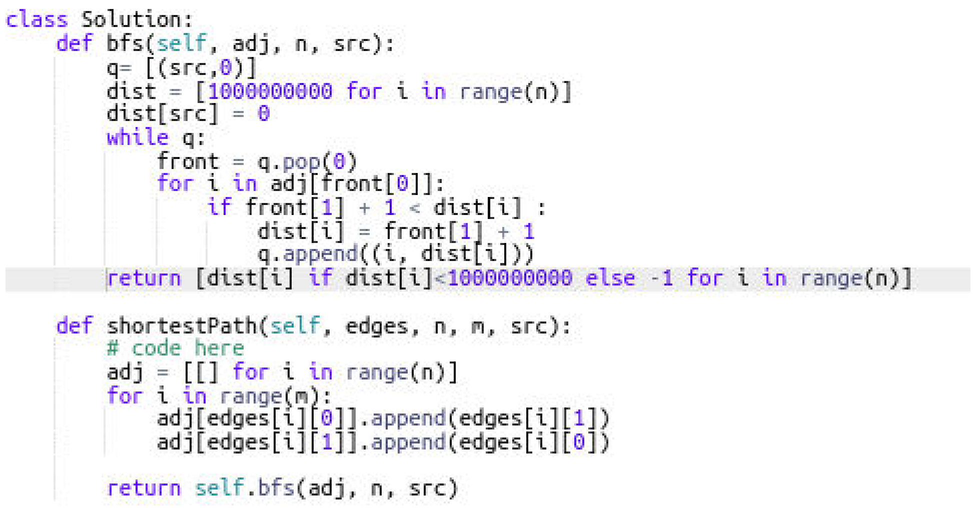


```python
class Solution:
    def topo_sort(self, adj, in_degree, n):
        q= [i for i in range(n) if in_degree[i]==0]
        ts= []
        while q:
            front = q.pop(0)
            ts.append(front)
            for i in adj[front]:
                in_degree[i[0]]-=1
                if in_degree[i[0]] == 0:
                    q.append(i[0])

        return ts

    def shortestPath(self, n : int, m : int, edges : List[List[int]]) -> List[int]:
        adj = [[] for i in range(n)]
        in_degree = [0 for i in range(n)]
        for i in range(m):
            adj[edges[i][0]].append((edges[i][1], edges[i][2]))
            in_degree[edges[i][1]]+=1

        topo = self.topo_sort(adj, in_degree, n)

        dest = [1000000000 for i in range(n)]
        dest[0] = 0
        for i in range(n):
            if topo[i] == 0:
                break
            dest[i] = -1

        while i < n:
            front = topo[i]

            for j in adj[front]:
                dest[j[0]] = min(dest[j[0]], dest[front]+j[1])

            i+=1

        return [i if i<1000000000 else -1 for i in dest]
```

-> Shortest path in undirected graph:

-> maintain a distance list and a queue having tuple (node, distance_upto_node).

-> execute bfs and if distance to neighbour is less than distance in list, append that neighbour nodes. Do this until queue is empty.



```
class Solution:
    def bfs(self, adj, n, src):
        q= [(src,0)]
        dist = [1000000000 for i in range(n)]
        dist[src] = 0
        while q:
            front = q.pop(0)
            for i in adj[front[0]]:
                if front[1] + 1 < dist[i] :
                    dist[i] = front[1] + 1
                    q.append((i, dist[i]))
        return [dist[i] if dist[i]<1000000000 else -1 for i in range(n)]

    def shortestPath(self, edges, n, m, src):
        # code here
        adj = [[] for i in range(n)]
        for i in range(m):
            adj[edges[i][0]].append(edges[i][1])
            adj[edges[i][1]].append(edges[i][0])

        return self.bfs(adj, n, src)
```

******* NOTE: must do word ladder 1 and 2***

HINT: in word ladder 1 --> start bfs with beginWord, check all possible words that can be generated from current word and if that word is present in dictionary, append that word in queue.

In word ladder 2 --> either maintain whole sequence/list in queue or use word ladder 1 logic and back track.

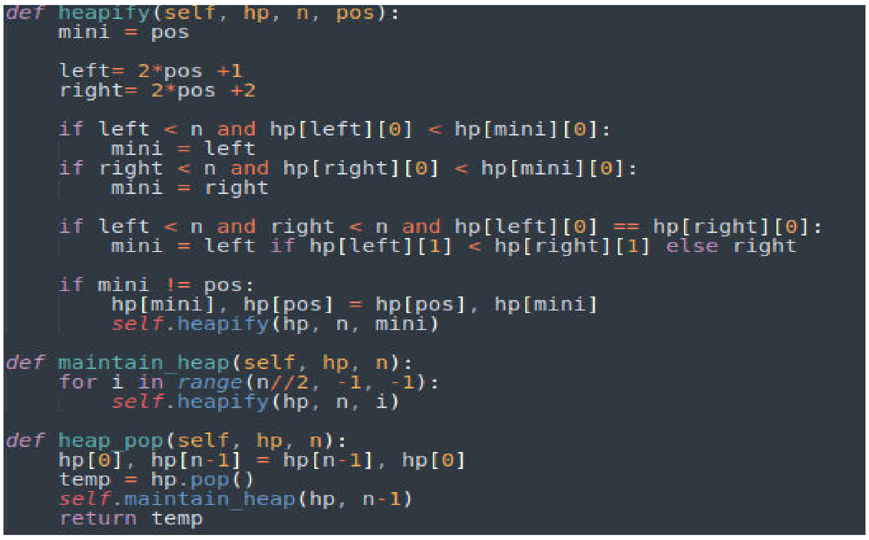
-> Dijkstra algorithm:

-> this algorithm is used to find shortest path in any graph (directed or undirected) but only having edges with non negative weights.

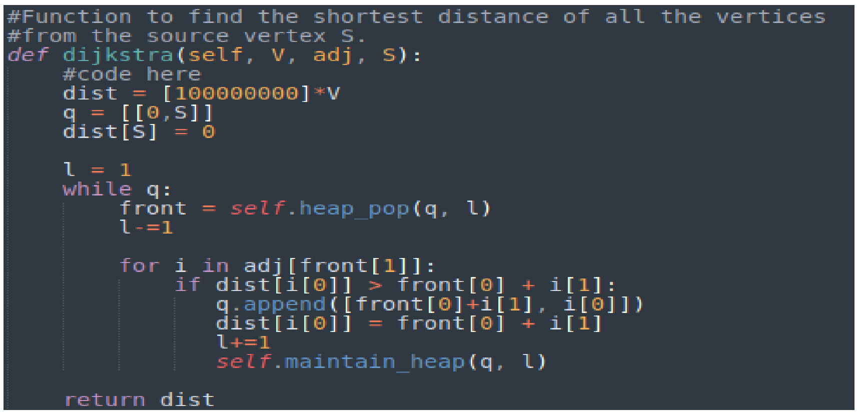
-> use priority queue to store (weight, node) *weight* to reach *node.*

-> cannot detect negative cycle in graph

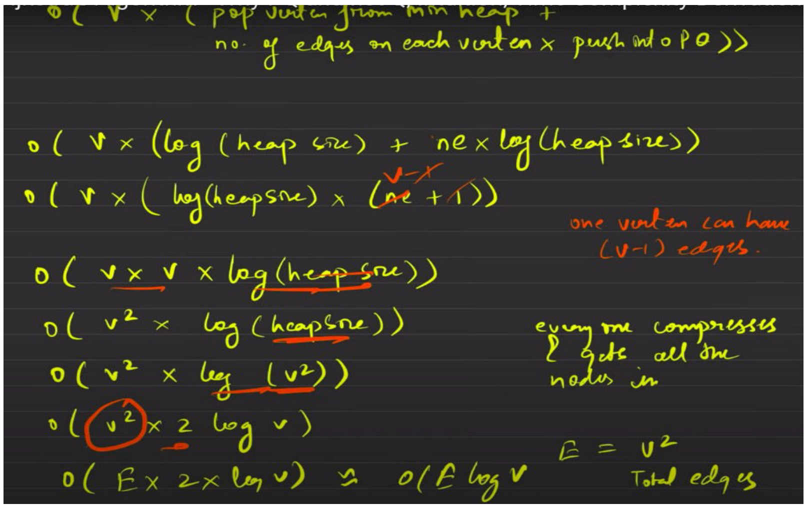
-> do bfs until q is empty, while bfs, if distance to reach any node is less than current reach distance of node, then update that distance in a list and place this node with weight on priority queue. (***TC: ElogV , see derivation below***)



```python
def heapify(self, hp, n, pos):
    mini = pos

    left= 2*pos +1
    right= 2*pos +2

    if left < n and hp[left][0] < hp[mini][0]:
        mini = left
    if right < n and hp[right][0] < hp[mini][0]:
        mini = right

    if left < n and right < n and hp[left][0] == hp[right][0]:
        mini = left if hp[left][1] < hp[right][1] else right

    if mini != pos:
        hp[mini], hp[pos] = hp[pos], hp[mini]
        self.heapify(hp, n, mini)

def maintain_heap(self, hp, n):
    for i in range(n//2, -1, -1):
        self.heapify(hp, n, i)

def heap_pop(self, hp, n):
    hp[0], hp[n-1] = hp[n-1], hp[0]
    temp = hp.pop()
    self.maintain_heap(hp, n-1)
    return temp
```



```python
#Function to find the shortest distance of all the vertices
#from the source vertex S.
def dijkstra(self, V, adj, S):
    #code here
    dist = [100000000]*V
    q = [[0,S]]
    dist[S] = 0

    l = 1
    while q:
        front = self.heap_pop(q, l)
        l-=1

        for i in adj[front[1]]:
            if dist[i[0]] > front[0] + i[1]:
                q.append([front[0]+i[1], i[0]])
                dist[i[0]] = front[0] + i[1]
                l+=1
                self.maintain_heap(q, l)

    return dist
```



O(V × (pop vertex from min heap +
no. of edges on each vertex × push into PQ))

$O(V \times (\log(\text{heap size}) + ne \times \log(\text{heap size}))$

$O(V \times (\log(\text{heapsize}) \times (\overset{V-1}{\cancel{ne}} + \cancel{1}))$

one vertex can have (V-1) edges.

$O(\underline{V \times V} \times \log(\underline{\text{heap size}}))$

$O(V^2 \times \log(\underline{\text{heapsize}}))$

every one compresses & gets all the nodes in

$O(V^2 \times \underline{\log (V^2)})$

$O(\boxed{V^2} \times \underline{2} \log V)$

$E = V^2$

Total edges

$O(E \times 2 \times \log V) \simeq O(E \log V$

-> **Bellman-Ford algo:**

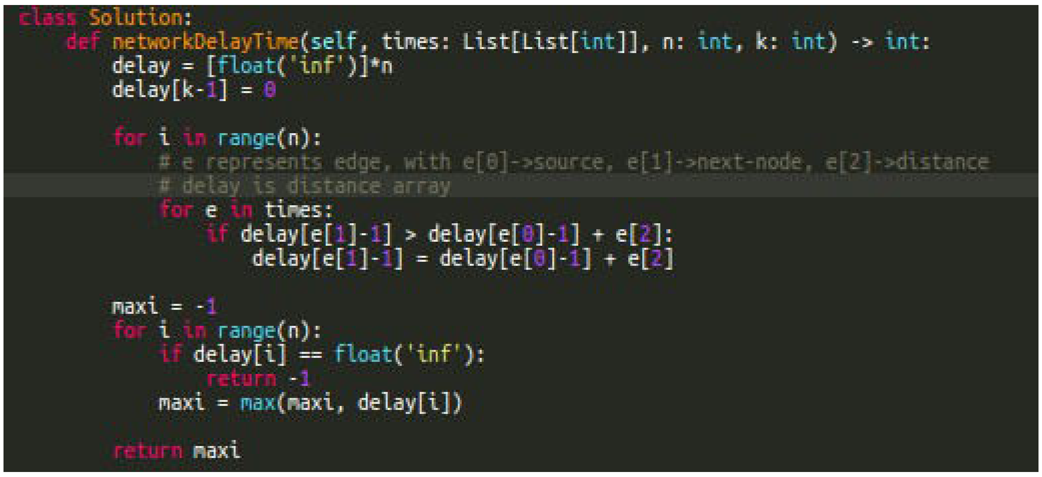
-> to overcome issues with dijkstra, this algo. Can be used. On the basis of observation, it can be said that if there are n nodes in a graph, this graph can be traversed in at most n-1 iterations of all edges.

-> maintain a distance array, to store least weight to reach a specific node.

-> explore all edges n-1 times, updating dist array accordingly.

** to detect if negative cycle is present in graph:

-> iterate one more time nth iteration, if there is relaxation in dist array, that means negative is present.

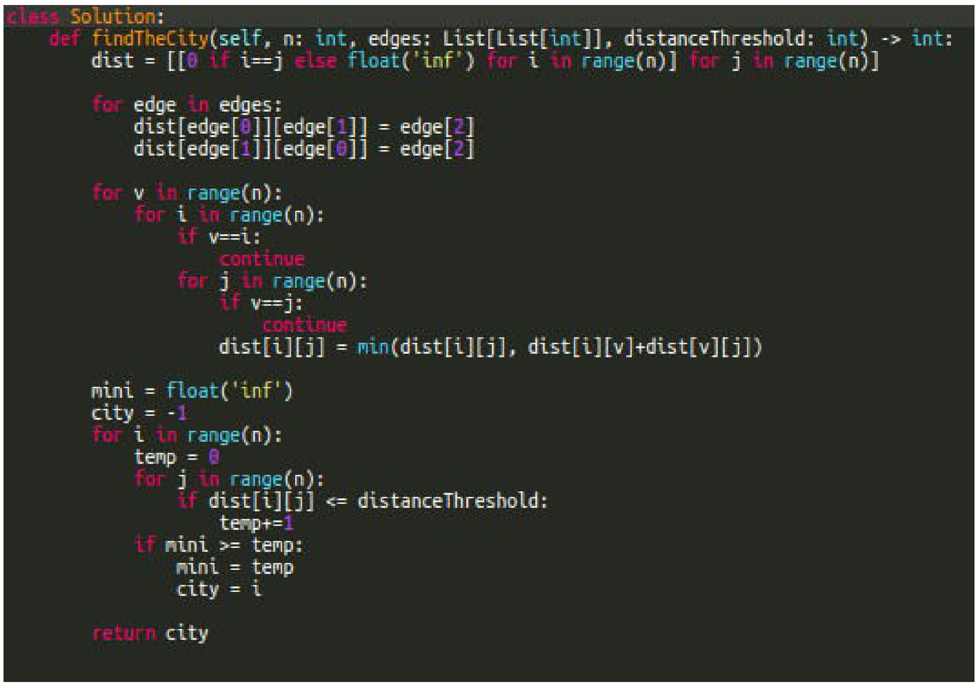


```python
class Solution:
    def networkDelayTime(self, times: List[List[int]], n: int, k: int) -> int:
        delay = [float('inf')]*n
        delay[k-1] = 0

        for i in range(n):
            # e represents edge, with e[0]->source, e[1]->next-node, e[2]->distance
            # delay is distance array
            for e in times:
                if delay[e[1]-1] > delay[e[0]-1] + e[2]:
                    delay[e[1]-1] = delay[e[0]-1] + e[2]

        maxi = -1
        for i in range(n):
            if delay[i] == float('inf'):
                return -1
            maxi = max(maxi, delay[i])

        return maxi
```

-> **Floyd-Warshall algo:**

-> this algorithm can be used to find multiple source shortest path.

-> can detect negative cycle if distance to same node is negative.



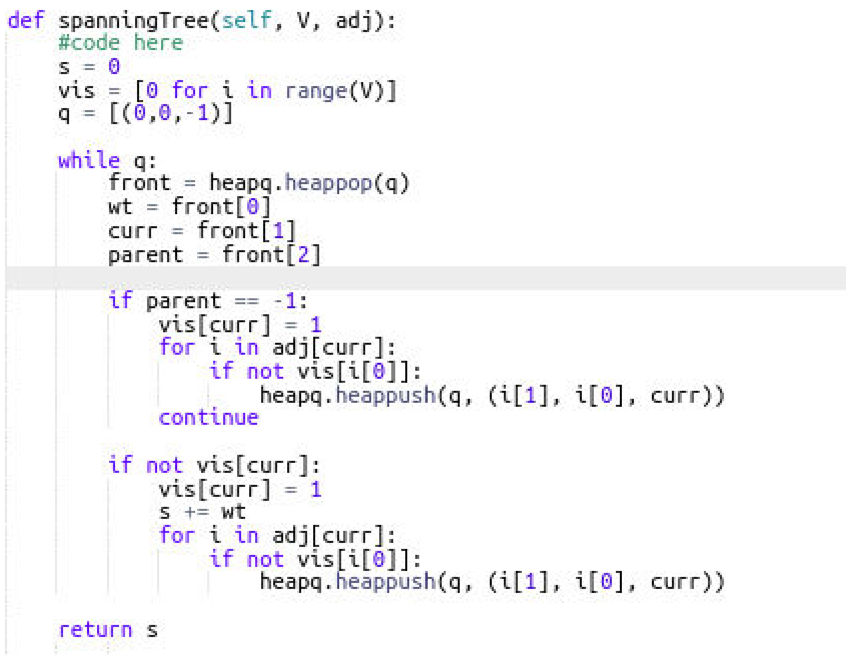
```python
class Solution:
    def findTheCity(self, n: int, edges: List[List[int]], distanceThreshold: int) -> int:
        dist = [[0 if i==j else float('inf') for i in range(n)] for j in range(n)]

        for edge in edges:
            dist[edge[0]][edge[1]] = edge[2]
            dist[edge[1]][edge[0]] = edge[2]

        for v in range(n):
            for i in range(n):
                if v==i:
                    continue
                for j in range(n):
                    if v==j:
                        continue
                    dist[i][j] = min(dist[i][j], dist[i][v]+dist[v][j])

        mini = float('inf')
        city = -1
        for i in range(n):
            temp = 0
            for j in range(n):
                if dist[i][j] <= distanceThreshold:
                    temp+=1
            if mini >= temp:
                mini = temp
                city = i

        return city
```

--> **MST (minimum spanning tree):**

-> Using Prims Algo:

a mst is a tree in which every node is reachable to every other node and there is no cycle. Also, there are multiple MST possible for a given graph and sum of all edges in MST is the least as compared to other spanning tree. A graph having n nodes and n-1 edges with no cycles.

-> maintain a PQ (priority queue), in which elements are (weight, curr_node, parent_of_node).
-> also maintain a visited array to check if a node is already visited or not.
-> start from any node with parent=-1 (signifying that we are starting).
-> if parent is -1, do not consider this node in MST, only mark it as visited and start traversing its neighbour nodes, if neighbour nodes are not in visited list, than insert them in PQ.
-> iterate until PQ is empty, if node popped is not in visited than only consider this node, and start traversing its neighbour nodes, if neighbour nodes are not in visited list, than insert them in PQ.



```python
def spanningTree(self, V, adj):
    #code here
    s = 0
    vis = [0 for i in range(V)]
    q = [(0,0,-1)]

    while q:
        front = heapq.heappop(q)
        wt = front[0]
        curr = front[1]
        parent = front[2]

        if parent == -1:
            vis[curr] = 1
            for i in adj[curr]:
                if not vis[i[0]]:
                    heapq.heappush(q, (i[1], i[0], curr))
            continue

        if not vis[curr]:
            vis[curr] = 1
            s += wt
            for i in adj[curr]:
                if not vis[i[0]]:
                    heapq.heappush(q, (i[1], i[0], curr))

    return s
```

-> Disjoint set union:

-> disjoint set union is a data structure which can be used to find if any node belongs to some specific component or not.

-> it includes two important methods:

1. find_parent :- this method gives the ultimate parent of a node.
2. union_by_size / union_by_rank :- combines two nodes or components on the basis of rank or size.

-> find_parent:

-> searches for parent of a particular node recursively until a node is found whose parent is the node itself (parent[node] = node).

-> this method can be optimized using path compression. As we find parent of a node, along with that update parent of all nodes in the path.

-> union_by_size :-

-> adds one component to another, comp. Whose size is smaller gets attached to another.

-> if same size, then attach anyone to anyone.

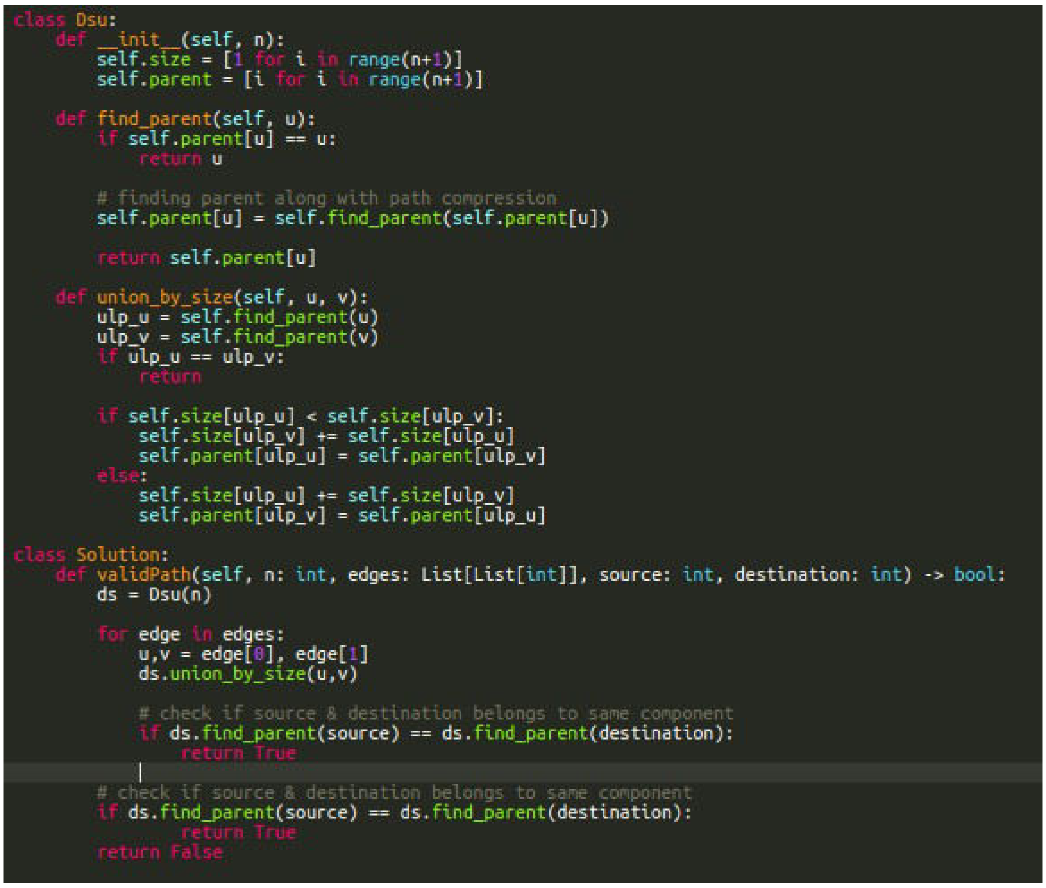
-> update ultimate parent of the attached node to node with whom it is attached and increase the size component accordingly.

-> union_by_rank :-

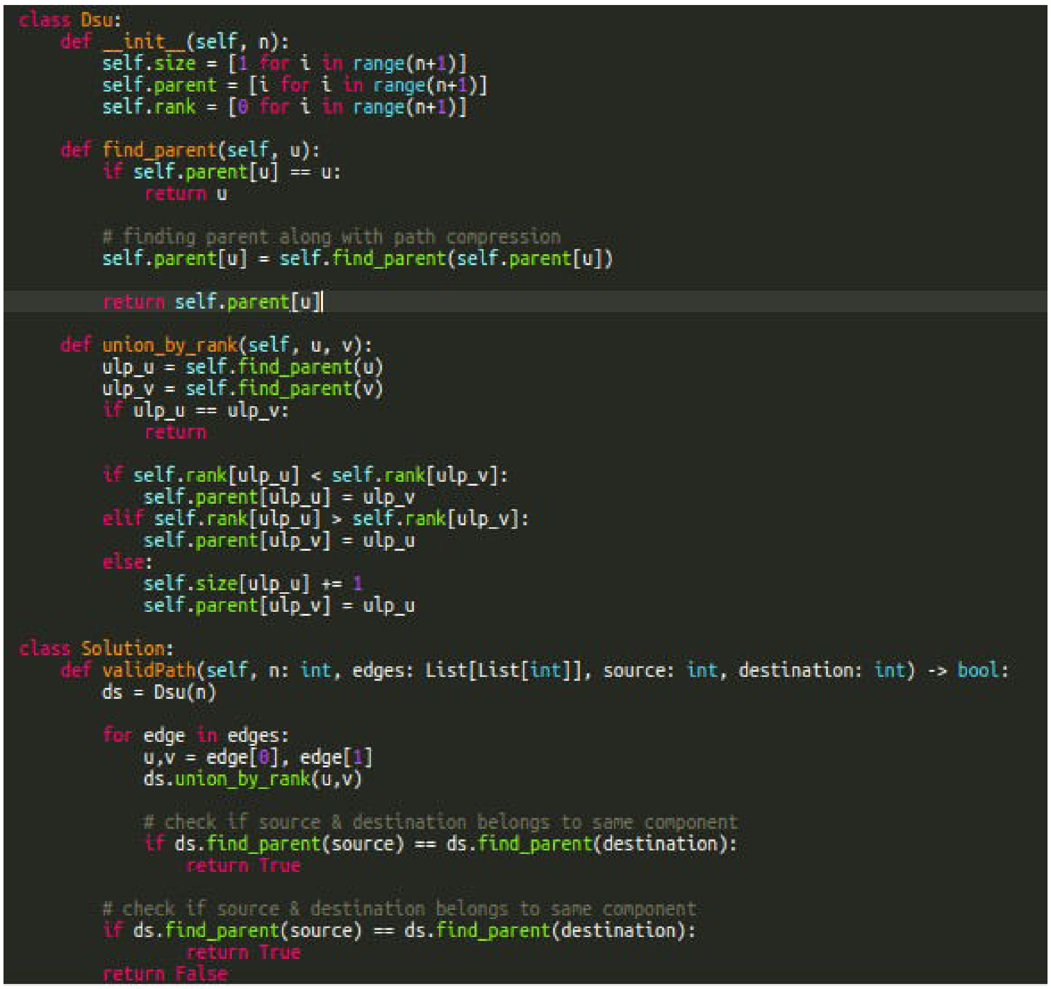
-> adds one component to another, comp. Whose rank is smaller gets attached to another.

-> if same rank, then attach anyone to anyone and increase the rank.

-> update ultimate parent of the attached node to node with whom it is attached.



```python
class Dsu:
    def __init__(self, n):
        self.size = [1 for i in range(n+1)]
        self.parent = [i for i in range(n+1)]

    def find_parent(self, u):
        if self.parent[u] == u:
            return u

        # finding parent along with path compression
        self.parent[u] = self.find_parent(self.parent[u])

        return self.parent[u]

    def union_by_size(self, u, v):
        ulp_u = self.find_parent(u)
        ulp_v = self.find_parent(v)
        if ulp_u == ulp_v:
            return

        if self.size[ulp_u] < self.size[ulp_v]:
            self.size[ulp_v] += self.size[ulp_u]
            self.parent[ulp_u] = self.parent[ulp_v]
        else:
            self.size[ulp_u] += self.size[ulp_v]
            self.parent[ulp_v] = self.parent[ulp_u]

class Solution:
    def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:
        ds = Dsu(n)

        for edge in edges:
            u,v = edge[0], edge[1]
            ds.union_by_size(u,v)

            # check if source & destination belongs to same component
            if ds.find_parent(source) == ds.find_parent(destination):
                return True

        # check if source & destination belongs to same component
        if ds.find_parent(source) == ds.find_parent(destination):
                return True
        return False
```



```python
class Dsu:
    def __init__(self, n):
        self.size = [1 for i in range(n+1)]
        self.parent = [i for i in range(n+1)]
        self.rank = [0 for i in range(n+1)]

    def find_parent(self, u):
        if self.parent[u] == u:
            return u

        # finding parent along with path compression
        self.parent[u] = self.find_parent(self.parent[u])

        return self.parent[u]

    def union_by_rank(self, u, v):
        ulp_u = self.find_parent(u)
        ulp_v = self.find_parent(v)
        if ulp_u == ulp_v:
            return

        if self.rank[ulp_u] < self.rank[ulp_v]:
            self.parent[ulp_u] = ulp_v
        elif self.rank[ulp_u] > self.rank[ulp_v]:
            self.parent[ulp_v] = ulp_u
        else:
            self.size[ulp_u] += 1
            self.parent[ulp_v] = ulp_u

class Solution:
    def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:
        ds = Dsu(n)

        for edge in edges:
            u,v = edge[0], edge[1]
            ds.union_by_rank(u,v)

            # check if source & destination belongs to same component
            if ds.find_parent(source) == ds.find_parent(destination):
                return True

        # check if source & destination belongs to same component
        if ds.find_parent(source) == ds.find_parent(destination):
                return True
        return False
```

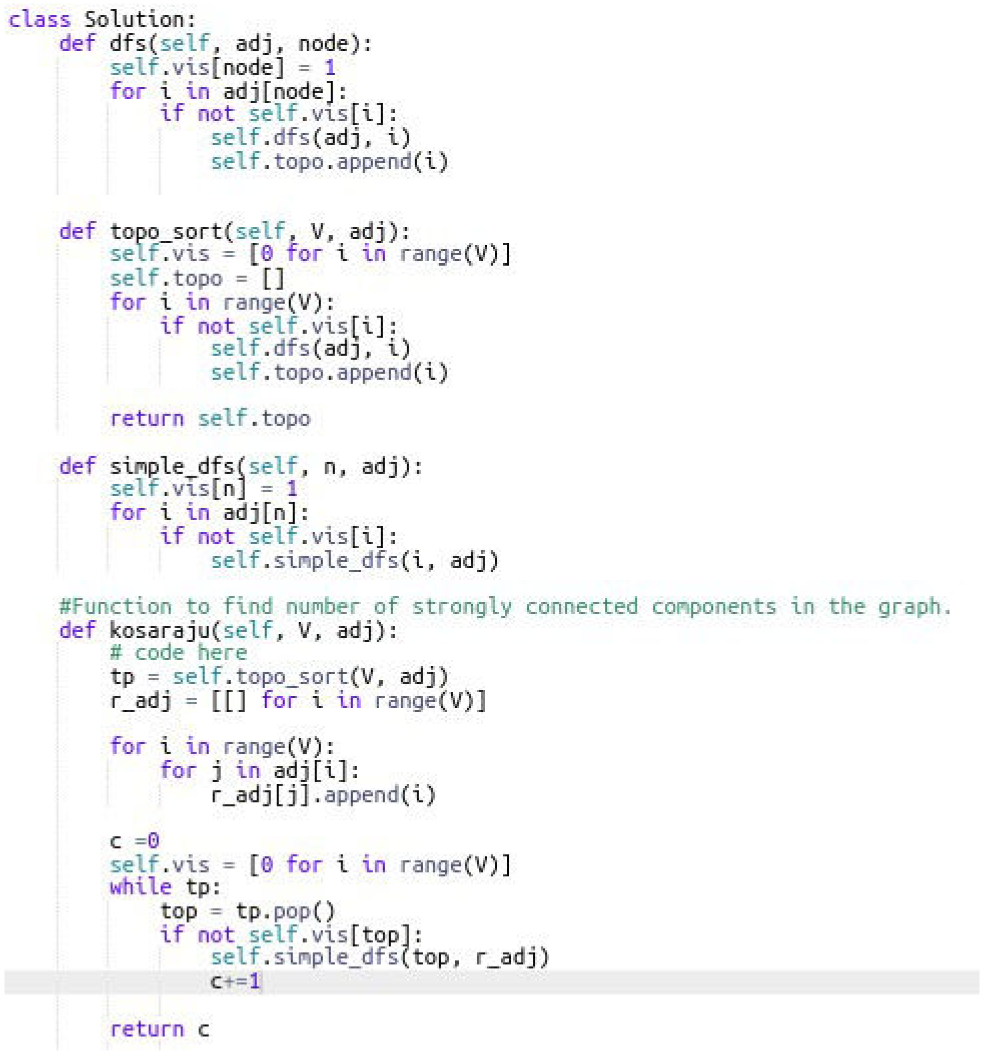
-> Strongly connected components (SCCs) Kosaraju's Algo:

-> only applicable to directed graphs.

-> A SCC is a component of graph in which every node of component is reachable to every other node.

-> reverse the entire graph.

-> first figure out the topo sort of original graph, because we need the first node of which dfs will start. This node will be part of first SCC. Sort acc. To finish time.



```python
class Solution:
    def dfs(self, adj, node):
        self.vis[node] = 1
        for i in adj[node]:
            if not self.vis[i]:
                self.dfs(adj, i)
                self.topo.append(i)


    def topo_sort(self, V, adj):
        self.vis = [0 for i in range(V)]
        self.topo = []
        for i in range(V):
            if not self.vis[i]:
                self.dfs(adj, i)
                self.topo.append(i)

        return self.topo

    def simple_dfs(self, n, adj):
        self.vis[n] = 1
        for i in adj[n]:
            if not self.vis[i]:
                self.simple_dfs(i, adj)

    #Function to find number of strongly connected components in the graph.
    def kosaraju(self, V, adj):
        # code here
        tp = self.topo_sort(V, adj)
        r_adj = [[] for i in range(V)]

        for i in range(V):
            for j in adj[i]:
                r_adj[j].append(i)

        c =0
        self.vis = [0 for i in range(V)]
        while tp:
            top = tp.pop()
            if not self.vis[top]:
                self.simple_dfs(top, r_adj)
                c+=1

        return c
```

-> Bridges in Graph:

-> to get bridge (edge) in graph, maintain two list:

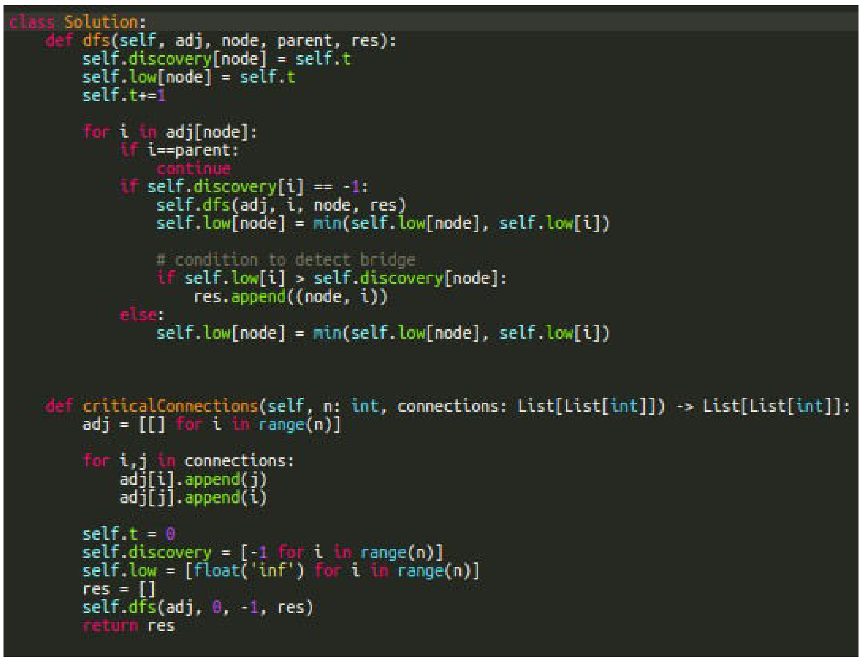
1. discovery_time: time when node was discovered by dfs.
2. low : it's element represents the lowest time of a node after which this node can be found. (minimum discovery time of neighbour node)

-> initially discovery time has all -1 as elements representing that no node is visited yet.

-> start dfs traversal from any node and store its discovery time and low time (both are same until dfs backtracks)

-> if a node is not visited, call dfs for that and after completion of dfs, compare node's low time with parent's low time and update parent's low time if it's greater than node's low time. IF DISCOVERY TIME OF PARENT NODE IS LESSER THAN LOW TIME OF CURRENT NODE, THEN THERE IS A BRIDGE. (it represents that there is no way to reach node without parent node.)

-> if a node's already visited, just compare low time of node and it's parent.



```python
class Solution:
    def dfs(self, adj, node, parent, res):
        self.discovery[node] = self.t
        self.low[node] = self.t
        self.t+=1

        for i in adj[node]:
            if i==parent:
                continue
            if self.discovery[i] == -1:
                self.dfs(adj, i, node, res)
                self.low[node] = min(self.low[node], self.low[i])

                # condition to detect bridge
                if self.low[i] > self.discovery[node]:
                    res.append((node, i))
            else:
                self.low[node] = min(self.low[node], self.low[i])


    def criticalConnections(self, n: int, connections: List[List[int]]) -> List[List[int]]:
        adj = [[] for i in range(n)]

        for i,j in connections:
            adj[i].append(j)
            adj[j].append(i)

        self.t = 0
        self.discovery = [-1 for i in range(n)]
        self.low = [float('inf') for i in range(n)]
        res = []
        self.dfs(adj, 0, -1, res)
        return res
```

-> Articulation points:

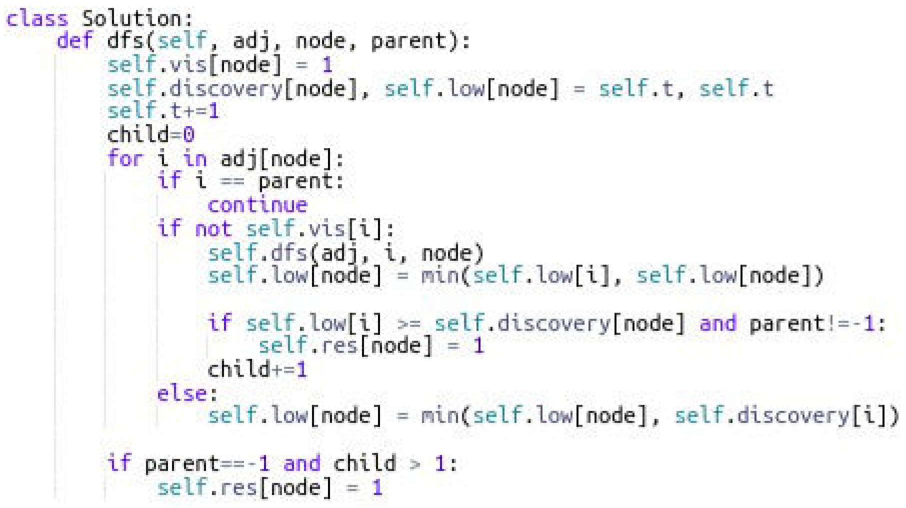
-> those nodes in graph on whose removal graph will be divided into two or more components.

-> logic will be same as bridge in graph, only difference is that in bridge detection we were working on removal of edges, here, node will be removed.

-> if adjacent node is visited, instead of updating low time of node with low time of neighbour node, discovery time of adjacent node will be used, as it is possible that this neighbout node might get removed.

-> if adjacent node not visited, do dfs, after that, update low time of node with this adjacent node and after that compare discovery time of node with low time of adjacent node with equality (this represents that were looking beyond this adjacent node) and parent should not be -1.

-> for starting node, we can count unvisited childs. If there more than 1 unvisited childs than this node is also articulation node.



```python
class Solution:
    def dfs(self, adj, node, parent):
        self.vis[node] = 1
        self.discovery[node], self.low[node] = self.t, self.t
        self.t+=1
        child=0
        for i in adj[node]:
            if i == parent:
                continue
            if not self.vis[i]:
                self.dfs(adj, i, node)
                self.low[node] = min(self.low[i], self.low[node])

                if self.low[i] >= self.discovery[node] and parent!=-1:
                    self.res[node] = 1
                child+=1
            else:
                self.low[node] = min(self.low[node], self.discovery[i])

        if parent==-1 and child > 1:
            self.res[node] = 1
```