Lecture 37: Graph cycle detection- Union Find and Path Compression Algo, Path Exists in Graph, Invert Binary Tree

1. Find if the path Exists in Graph

Solution Approach: Breadth First Traversal
Time Complexity: O(V+E)

```
from collections import defaultdict
class Solution:
    def validPath(self, n: int, edges: List[List[int]], source: int, destination: int) -> bool:
        ## Breadth First Traversal
        graph = defaultdict(list)
        for a, b in edges:
            graph[a].append(b)
            graph[b].append(a)


        ## Store all the nodes to be visited in the queue
        visited = [False] * n
        visited[source] = True
        queue = collections.deque([source])

        while queue:
            node = queue.popleft()
            if node == destination:
                return True

            for adjacent_node in graph[node]:
                if not visited[adjacent_node]:
                    visited[adjacent_node] = True
                    queue.append(adjacent_node)


        return False
```

2. Invert Binary Tree

Solution Approach: Recursion

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if root is not None:
            rightSubtree = self.invertTree(root.right)
            leftSubtree = self.invertTree(root.left)

            root.left = rightSubtree
            root.right = leftSubtree

            return root
```

3. Graph cycle detection algorithm: Union by rank and Path Compression algo

```
class Graph:

        def __init__(self, num_of_v):
                self.num_of_v = num_of_v
                self.edges = defaultdict(list)

        def add_edge(self, u, v):
                self.edges[u].append(v)


class Subset:
        def __init__(self, parent, rank):
                self.parent = parent
                self.rank = rank
## Path compression algorithm
def find(subsets, node):
        if subsets[node].parent != node:
                subsets[node].parent = find(subsets, subsets[node].parent)
        return subsets[node].parent

# A function that does the union of two sets
# of u and v(uses union by rank)


def union(subsets, u, v):

        # Attach smaller rank tree under root
        # of high rank tree(Union by Rank)
        if subsets[u].rank > subsets[v].rank:
                subsets[v].parent = u
        elif subsets[v].rank > subsets[u].rank:
                subsets[u].parent = v

        # If ranks are the same, then make one as
        # root and increment its rank by one
        else:
                subsets[v].parent = u
                subsets[u].rank += 1

# The main function is to check whether a given
# graph contains cycle or not
def isCycle(graph):
```

```
        # Allocate memory for creating sets
        subsets = []

        for u in range(graph.num_of_v):
                subsets.append(Subset(u, 0))

        # Iterate through all edges of graph,
        # find sets of both vertices of every
        # edge, if sets are same, then there
        # is cycle in graph.
        for u in graph.edges:
                u_rep = find(subsets, u)

                for v in graph.edges[u]:
                        v_rep = find(subsets, v)

                        if u_rep == v_rep:
                                return True
                        else:
                                union(subsets, u_rep, v_rep)



# Driver Code
g = Graph(3)

# add edge 0-1
g.add_edge(0, 1)

# add edges 1-2
g.add_edge(1, 2)

# add edge 0-2
g.add_edge(0, 2)

if isCycle(g):
        print('Graph contains cycle')
else:
        print('Graph does not contain cycle')
```