1. (10 pt) A binary tree is a rooted tree in which each node has at most two children. Prove that in any binary tree the number of nodes with two children is exactly one less than the number of leaves. (Hint: prove by induction.)

Prove by induction: Let n be the number of nodes in the binary tree T. Let n_i be the number of leaves and let n_j be the number of nodes with two children

Base Case: n = 1

Suppose a tree with one root node, there will be three possible situations:

- if this node has no children, the tree has 1 leaf and it has no nodes with 2 children. (Condition satisfied)
- If the root has a leaf, then similarly the tree has 1 leaf and no nodes with 2 children. (Condition satisfied)
- If the root node has two children that are leaves, then the tree has 2 leaves and 1 node with 2 children. (Condition satisfied)

Induction Hypothesis: In an arbitrary binary tree T with n nodes that the number of nodes with two children is exactly one less than the number of leaves. $(n_i - n_j = 1)$

Induction Step: Adding a new node to the binary tree T, then we get a binary tree T' which has n+1 nodes. Let the adding node be v and let u be v's parent.

Case 1: The parent u has only 1 child.

Number of leaves in T': $n_i' = n_i + 1$

Number of nodes with two children in T': $n_j' = n_j + 1$

We have $n_i' - n_j' = (n_i + 1) - (n_j + 1) = n_i - n_j = 1$, it satisfies for T' with n+1 nodes.

Case 2: The parent u has no child, it is a leaf.

Number of leaves in T': $n_i' = n_i$

Number of nodes with two children in T': $n_j' = n_j$

So we have $n_i' - n_j' = 1$, the condition satisfies for T' with n+1 nodes.

Proved.

2. (25 pt) For an undirected and unweighted graph, the BFS algorithm introduced in the class and textbook will output the shortest path from source (node s) to other nodes. Modify the BFS algorithm to also output number of shortest paths from s to other node. Write down the pseudo code and explain why it's correct. We assume the graph is connected and your algorithm should take $O(m)$ time.

```
BFS (G, s):
      Initialize visited[i] = false for all i≠s, visited[s] = true
      let Q be queue.
      Q.push(s)
      dist[s] = 0   //The shortest distance from src to other node
      count[s] = 1  // number of shortest path from src to other node

      while (Q is not empty)
           u  =  Q.pop( )
          for all neighbours v of u in Graph G
               if visited[v] = false   //When v is not visited
                        Q.push(v)     //Stores v in Q to further visit its neighbour
                        dist[v] = dist[u]+ 1
                        count[v] = count[u]
                        visited[v] = true  //Mark it is visited

               else if visited[v] = true  //When v has been visited before
                      if dist[v] == dist[u] + 1   //New shortest path is found
                        count[v] = count[v] + count[u]
                      else if dist[v] > dist[u] + 1 //If there is a shorter path
                        dist[v] = dist[u] + 1
                        count[v] = count[u]
                      Endif
               Endif
          Endfor
      Endwhile
```

Time complexity is O(m) where m is the number of edges in the graph. The reason is this algorithm goes over every edge exatly O(1) time.

Explanation:

Apply two arrays in BFS, dist[] represents the shorest distance from source vertex, count[] represent the number of different shortest paths from source to other nodes. When visit the neighbour node v first time, set the value dist[v] and count[v]. When visit the neighbour node which has been visited before, there will be three possible cases:

Case1: A new shortest path is found. Change the value of count[v].

Case2: The current path from src to the visiting node has distance shorter than previous path. It means a shorter path is found. Change the value of dist[v].

Case3: The current path from src to the visiting node has distance longer than previous path. This current path does not need to be recored, so do nothing in this case.

Eventually, the dist[goal] contains the shortest distance from source vertex to the goal node and count[goal] contains the number of paths from source vertex to the goal node. The algorithm is correct.

3. (30 pt) We define the distance between two nodes u, v in a graph as the minimum number of edges in a path joining them, denoted by dist(u, v). The diameter of graph $G = (V, E)$ is the maximum distance between any pair of nodes, i.e.,

$$\text{diameter}(G) = \max_{u,v \in V} \text{dist}(u, v)$$

(a) Design an algorithm that runs in time $O(nm)$ and finds the diameter of G.

```
BFS (G, s):
      diameter = 0
      for each node s in Graph G  //Run BFS on each node
        Let Q be an empty queue
        Initialize visited[i] = false for all i≠s, visited[s] = true, maximum = 0
        Q.push(s)
        dist[s] = 0
        while (Q is not empty)
             curr = Q.pop( )
            //processing all the neighbours of curr
            for all neighbours v of curr in Graph G
                if visited[v] = false   //When node v is not visited
                        Q.push(v)     //Stores v in Q to further visit its neighbour
                        dist[v] = dist[curr]+ 1
                        visited[v] = true  //Mark it is visited
                        maximum = dist[v]
                        //Everytime it visit the node in next level, the maximum update
                Endif
            Endfor
            if maximum > diameter    //If a new maximum is found, change diameter
                diameter = maximum
            Endif
      Endwhile
      Return diameter
```

Since the algorithm runs BFS on each node, time complexity of each BFS is O(m), hence the time Complexity of this algorithm is O(nm), when implemented using an adjacency list.

n: the number of nodes, m: the number of edges.

(b) If the graph is a tree, denoted as $T = (V, E)$, the diameter can be computed efficiently. Please give an $O(n)$ time algorithm [1] to find the diameter of T. Write down the pseudo code and explain why your algorithm is correct.

Suggestion: consider modifying the DFS so that it can compute the following two numbers for each node v:

- d_v: the diameter of the subtree of T rooted at v,
- ℓ_v: the longest path from v to a leaf in the subtree of T rooted at v.

Think about how to compute these numbers in DFS recursion and how to compute the diameter of T using these numbers.

```
diameter(root):
      Initialize diameter = 0   //[0] -> height, [1] -> diamter
      if root == NULL
          diameter = 0
      else
          left_rs[2] = diameter(root.left)  //[0] is the height, [1] is the diameter
          right_rs[2] = diameter(root.right)


          left_diameter= left_rs[1]   //the diameter of left subtree
          right_diameter = right_rs[1]  //The diameter of right subtree
          left_height = left_rs[0]  //The height of left subtree
          right_height = right_rs[0]  //The height of right subtee
          height = max(left_rs[0], right_rs[0]) +1
          root_diameter = left_height + right_height + 1 //The gteatest path between leaves that goes
through the root of T
          diameter = max(left_diameter, right_diameter, root_diameter) //get the maximum
      Endif
      return diameter
```

The time complexity is O(n), where n is the number of nodes. Since this algorithm access each node O(1) time.

Correctness Proof:

Basic idea is to use recursion function to get the height of the left and right subtree to calculate the diameter of the tree T. To make sure it access each node o(1) time, use an array to return the height and diameter in a single iteration. And by the data structure of tree we can know taht the diameter of a tree T is the largest of the following three quantities:

- The diameter of left subtree in T
- The diameter of right subtree in T
- The gteatest path between leaves that goes through the root of T

Above all, this algorithm is correct.

4. In this problem, we consider another notion of "semi-connectivity". A directed graph is semi-connected if, for all pairs of $u, v \in V$, there is a path from u to v or from v to u. We will develop an algorithm to test if a directed graph $G = (V, E)$ is semi-connected.

   (a) (15 pt) Consider a simplified case where G is a DAG. Design an $O(m)$ time algorithm to test whether a DAG is semi-connected.

Claim: Given a DAG, assume topological sort is running on this graph. This DAG would be semi-connected iff there is an edge between every consecutiv1e pair of vertices when topological sort.

Correctness Proof:

A semi-connected graph has a single path that goes through all vertices, which means there is a path between an consecutive pair of vertices when topological sort the DAG. Given such a DAG that with n vertices in linearized order $v_1, v_2 \dots v_n$. By the property of T-order, there cannot be a path v_{i+1} -> v_i. To make the graph to be semi-connected, the path between v_i and v_{i+1} is the edge between them ($v_i -> v_{i+1}$) for every i. So there are edges between all consecutive pairs of vertices in the topological order, it also indicates that there is a path go through all vertices in the graph, so this DAG is semi-connected.

Algorithm: Assume topological sort conducts on the given DAG. Then check if there is a pair of verties between every pair of consecutive vertices. If yes, this DAG is semi-connected, otherwise it is not semi-connected.

The **time complexity** is O(m). Reason: Counting degree takes O(m) time and running toplogical sort takes O(m) time. Hence, O(m)+O(m)=O(m).

(b) (5 pt) For a directed graph G, we can construct a pseudo-graph G' by the following way:

- Decompose G into a set of Strongly Connected Components (SCCs). This can be done using the Kosaraju's algorithm described in the class. The output of Kosaraju's algorithm will be k node sets $S_1, \ldots, S_k$, where each S_i contains nodes in the i-th SCC. The sets will be maximal in the sense that for each S_i, adding any other node will break its strong connectivity (we didn't officially prove this in the class but you can directly use this property).
- Take each SCC as a node to create the pseudo-graph G'. More specifically, $G' = (V', E')$ where V' has K nodes $v'_1, \ldots, v'_k$, each of them corresponds to a SCC; and $(v'_i, v'_j) \in E'$ if and only if there is an edge pointing from a node in S_i to a node in S_j in the original graph.

Prove that for any directed graph G, its pseudo-graph G' constructed this way will be a DAG.

Proof by contradiction. Suppose the pseudo-graph G' constructed this way is not DAG, since G' is directed, if it is not DAG, there would be cycles in G'.

Assume a cycle C in G' such that $(v'_1, v'_2, \ldots v'_n, v'_1)$ where n < k. Then this implies there exist nodes that $u_1 \in v'_1$ and $u_2 \in v'_2$ such that $(u_1, u'_2) \in E$. It indicates that there is an edge pointing from a node in S_1 to S_2.

(c) (15 pt) Design an algorithm (based on (a) and (b)) to test semi-connectivity of a directed graph in $O(m)$ time. Note that you can assume functions are given to conduct topological sort and to decompose a graph into SCCs (Kosaraju's algorithm). Prove the correctness of your algorithm.

Algorithm:

Given a graph G = (V,E), assume functions are given to conduct topological sort and to decompose a graph into SCCs. Replace each SCC with a vertex to get a graph G', which is a DAG. If there is an edge between each pair of consecutive vertices (v_i, v_{i+1}) for any i, then G' is semi-connected. It also indicates that G is semi-connected.

The **time complexity** of this algorithm is O(m) where m is the number of edges.

Connectness Proof:

Component grah is always a DAG, because it has no cycles, since a cycle containing SCCs would merge them all to a single SCC. So topologically sort the component graph is possible. And all vertices in each SCC are mutually reachable from each other. Therefore, by part(a), as long as there exist an edge between every consecutiv1e pair of vertices when topolologically sort the component graph, the graph is semi-connected.