1.

```
Def findDiameter (TreeNode root):
   if root == None:
        return 0

   ans = 0
   dfs(root, ans)
   return ans

def dfs (TreeNode root, int ans):
     if root is None: return 0

     res1 = dfs(root.left, ans)
     res2 = dfs(root.right, ans)
     ans = max(ans, res1 + res2)
     return max(res1, res2) + 1
```

time complexity is O(n), n means the number of nodes in the tree

2.
G is a directed acyclic graph

```
topoSort (G):
  declare T to be an empty list
  declare Z to be an empty queue
  declare A to be an array, which stores the current indegree of each vertex
  for each vertex in G.V:
        for each u adjacent to v:
            A[v] ++
  For each v in G.V:
        If A[v] == 0: add v to Z
  While Z is not empty:
        node = Z.removefront()
        T.append(node)
        For each u adjacent to node:
            Decrement A[u]
            If A[u] == 0:
                Add u to Z

  Return T
```

Time complexity is O (V + E), space complexity is O (V)

Explanation: T stores the sorted result of the graph. Z stores the vertices whose indegree is 0. Array A stores the indegree of the current vertex. Then loop through the vertices in the graph and get the indegree for each vertex. If there are any vertices whose indegree in 0, add it to the Z. then loop through Z, remove the first node in the queue and append it to the T. For each adjacent node, decrement the indegree of the removed node and if the indegree of the removed node becomes 0, add this removed node to Z. eventually return T.

3.

```
buildNetwork(N, M, D):
    choose a city as the root
    for each u in N:
        u.key = infinity
        u.parent = NIL

   root.key = 0
   Q = N
   While Q is not empty:
        u = EXTRACT-MIN(Q)
        for each city in u’s adjacent cities:
                If city belongs to Q and D (u, city) is less than v.key:
                        City.parent = u
                        City.key = D (city, u)
```

the time complexity is O (N + M), the space complexity is O(N);

Explanation: this algorithm is called prim’s algorithm. The first for loop set the key of each vertex to be infinity and set the parent of each vertex to be NIL. And initialize the min-priority queue Q to contain all the vertices. In the while loop, first extract a city u from the Q, this city is incident on a light edge that cross the cut (N – Q, Q). the extracted city is being added to the minimum spanning tree. Then iterate through all the adjacent cities of the extracted city, if the adjacent city belongs to the Q and its distance to the city u is less than adjacent city’s key, then update this adjacent city’s key to be its distance to the city u.

4.
I assume that G is the undirected connected graph

```
SPFailure (G):
  E = G.E              # E means all the edges in the graph
  Declare a List object called 'graph'
  For each edge in E:
        There are two nodes for each edge, node1, node2
        Graph[node1].append(node2)
        Graph[node2].append(node1)

  Declare a Boolean array called 'visited'
  Declare an int array called 'discover'
  Declare an int array called 'low'
  Declare an int array called 'parent'
  Declare a boolean array called 'ap'
  For each vertex in G.V:                # G.V means all the vertex in the graph
        If visited[vertex] == false:
            Dfs(graph, vertex, visited, discover, parent, low, ap, 0)

 Declare a list called answer
 For each node in ap:
        If ap[node] == true: answer.append(node)

 Return answer

Dfs (graph, vertex, visited, discover, parent, low, ap, time):
        Children = 0
        Visited[vertex] = true
        Discover[vertex] = time
        Low[vertex] = time
        time += 1
        For v in graph[vertex]:
            If visited[v] = false:
                 Parent[v] = u
                 Children += 1
                 Dfs (graph, v, visited, discover, parent, low, ap, time)
                 Low[vertex] = min (low[vertex], low[v])
                 If parent[vertex] == -1 and children > 1:
                     Ap[vertex] = true

                 If parent[vertex] != -1 and low[v] >= discover[vertex]:
                     Ap[vertex] = true

               Else if v != parent[vertex]: low[vertex] = min (low[vertex], discover[v])
```

Time complexity is O (V + E), the space complexity is O (V)

Explanation: I used articulation points graph algorithm in this problem. Articulation point is the single point of failure in the question. The first step is to build an undirected graph using the list object called 'graph'. The boolean array called 'visited', which shows whether a node has been visited or not. The int array discover is used to store the time that a vertex is visited. Int array low is used to store the minimum discover time of all the adjacent vertex of a given vertex. The int array parent is used to store the parent of a given vertex. The array ap is used to store the articulation points. In the for loop, the function loop through all the vertices in the graph. The dfs function calculates the number of children a vertex has, mark this vertex as visited, and record the discover time of this vertex. Store the low value of this vertex as the discover time. Then use the recursion to iterate through all the adjacent vertices of this vertex, then update the value of low[vertex]. If a given vertex has no parent and it has more than 1 children, which means this vertex is an articulation point, if a given vertex has a parent and its low value is greater than the vertex's discover time, which means this vertex is an articulation point. At the end of the function, return the articulation points of the whole graph.