Algorithms -1
Tutorial 6
Solution Sketch

1. Prove or disprove: BFS and DFS algorithms on a undirected, connected graph G = (V,E) produce the same tree if and only if G is a tree.

   First we show that G is a tree when both BFS-tree and DFS-tree are same. If G and T are not same, then there should exist an edge e(u, v) in G, that does not belong to T. In such a case:

   - in the DFS-tree, one of u or v, should be an ancestor of the other.
   - in the BFS-tree, u and v can differer by only one level.

   Since, both DFS-tree and BFS-tree are same tree T, it follows that one of u and v should be an ancestor of the other and they can differ by only one level. This implies that the edge connecting them must be in T. So, there cannot be any edges in G which are not in T. In the second part of the proof: Since G is a tree, each node has a unique path from the root. So, both BFS and DFS produce the same tree, and the tree is same as G.

2. There are N cities in Magicland. Some of the cities are connected by some roads. A road connects two cities and is bi-directional i.e., we can go either way through a road. There is a path from a city i to every other city j and there is no cycle among the cities. You need to find a pair of cities (i, j) such that the length of the path between i and j is maximum among all such pairs. The length of a path is the number of edges on the path. Your algorithm should run in O(V + E) time.

   The first thing to observe is that the cities form a tree. The maximum of the shortest distance between any pair of vertices in a graph is known as diameter. The simplest algorithm to find diameter is to do a BFS starting from each node and find the distance of the farthest node. It takes O(V (V +E)) time using adjacency list representation. However, there is an algorithm with better complexity for a tree. Start BFS from any node say u. Let v be a farthest node from u. Now, do a BFS from v. Let w be a farthest node from v. The distance between v and w is the diameter. Note that this algorithm is only for tree and not for general graphs.

3. Consider a directed unweighted graph *G*. Suppose that G has *k* strongly connected components S_1, S_2, …,S_k. We form another directed graph G_1 with *k* nodes, numbered from 1 to *k*, with an edge from node *i* to node *j* if and only if there is an edge from some node in S_i to some node in S_j in G. What is the maximum possible number of cycles in G_1? Justify your answer clearly.

   No cycle is possible. Because if there is a cycle, the corresponding strongly connected components are actually one strongly connected component (can go from any node in one SCC to any node in another following paths inside the SCC and then to the other SCCs) which is false.

4. You are given a DAG (Directed Acyclic Graph) G(V,E). You need to find the longest path in G. The length of a path is the number of edges on the path. Your algorithm should run in O(V + E) time.

   Let u_1, u_2, ..., u_N be a topological order of the vertices where N = |V|. Let us define dp[i] as the length of the longest path starting at i. Now, dp[i] = max(1, dp[j] + 1) for all (i, j) in E. If we start at a vertex i, either we finish the path at i or take one of its neighbour j as the next vertex in the path. The maximum over dp[i] is the length of the longest path.

5. Give an O(V + E) time algorithm that takes as input a directed acyclic graph G = (V,E) and two nodes s and t, and returns the number of paths from s to t in G.

   Let v_1, v_2, ..., v_n be a topological sorting of G where n = |V |. Let dp[v_i] denote the number of paths from s to v_i. Let v_x = u.

   dp[v_i] = 0 if i < x
   = 1 if i = x
   = $\sum_{(vj,\, vi) \text{ in } E}$ dp[v_j] otherwise

   By definition, dp[t] gives the number of paths from s to t

6. There are N variables x_1, x_2, ..., x_N and M relations of the form $x_i < x_j$ where $i \neq j$. A subset S of relations is called inconsistent if there does not exist any assignment of variables that satisfies all the relations in S. e.g. $\{x_1 < x_2, x_2 < x_1\}$ is inconsistent. You need to find if there's an inconsistent subset of M. Your algorithm should run in O(V + E) time.

   Let us define a graph G_S(V,E) such that V = x_1, x_2, ..., x_N and E = {(x_i, x_j) iff ($x_i < x_j$) in S} where S is some subset of M. A subset S is inconsistent if and only if there is a cycle in G_S. Also if S is inconsistent then so is M. So, we just need to check whether there is a cycle in G_M.

7. Let $G = (V, E)$ be an undirected graph represented as an adjacency list. A subset of edges F is called a ***feedback edge set*** if F contains at least one edge of every cycle of G. The size of a feedback edge set is the number of edges in it. A minimum size feedback edge set is a feedback edge set with minimum size among all possible feedback edge sets of G. Design an $O(|V|+|E|)$ time algorithm to find a minimum size feedback edge set of G.

   Do a DFS. Everytime a back edge is encountered, add the edge to the feedback edge set.

8. The eccentricity of a vertex v in a graph is the maximum distance from v to any other vertex in the graph. The center of a graph is the set of all vertices with minimum eccentricity. Your task is to find the center of a tree in linear time.

Steps to find the center:

1. If G has at most 2 vertices, output V(G) as the center.
2. Remove all leaves (vertices with degree 1) from G and corresponding edges
3. Go to 1

9. You are given a graph G = (V, E) with positive edge weights, and a minimum spanning tree T = (V, E_1) with respect to these weights. Now suppose the weight of a particular edge e in E is modifed from w(e) to a new value w'(e). You wish to quickly update the minimum spanning tree T to reflect this change, without recomputing the entire tree from scratch. There are four cases. In each case give a linear-time algorithm for updating the tree, if needed.
   (a) e not in E_1 and w'(e) > w(e).
   (b) e not in E_1 and w'(e) < w(e).
   (c) e in E_1 and w'(e) < w(e).
   (d) e in E_1 and w'(e) > w(e).

(a) e not in E_1 and w'(e) > w(e): T is still MST

(b) e not in E_1 and w'(e) < w(e): We add this edge to T. Now we've got exactly 1 cycle. Based on cycle property in MST, we need to find and remove edge with highest value on that cycle. We can do it using DFS or BFS.

(c) e in E_1 and w'(e) < w(e): T is still MST

(d) e in E_1 and w'(e) > w(e): We remove this edge from T. Now we have 2 connected components that should be connected. We can find both components using BFS or DFS. Now, we need to find an edge with smallest value that connects these components which we can find by iterating over the edges.