7.1 Breadth first decomposition

a) All edges in BFS connect either nodes on the same layer, or nodes that are 1 level apart. This is because under the definition of BFS, connected edges are either ancestors and descendants, or peers on the same level. It is not possible for a node to connect with another node that is more than 1 layer apart.

b) As the nodes are being traversed under the BFS algorith, for each node it is tested whether it is visited or not. For the ones that are seen (visited before), it will not be printed. Thus, only parent-child edges will be rendered as tree edges.

7.3 Depth first traversal

If the edge e = (a, b), and there is a path from b to c to a, e is a back edge (a is a descendant of b). However, since b is already traversed, the edge e can also be a cross edge.

7.5 DFS and why you mark you territory: to prevent explosions

a)

```
F(k) = {
    1, k = 1;
    1 + F(k-1) + 2 + 2F(k-2) + 4 + 4F(k-3) + ..., otherwise;
}
```

b)

```
F(k-1) = {
    1 + F(k-2) + 2 + 2F(k-3) + 4 + 4F(k-4)+ ...
} (general case)

F(k) - F(k-1) = 2 + F(k-2) + 2 + 2F(k-3) + 4 + 4F(k-4)...
              = 1 + F(k-1)

F(k) = 1 + 2F(k-1)

F(k) = 1 + 2 + 4 + ... + 2^(k-1) = 2^k - 1
```

c)

```
global seen [];
procedure DFS(k;;);
    seen.append(k);
    if k > 0 then
        for h <- k - 1 downto 0 do
            if h not in seen then
                print(k, h);
                DFS(h);
            end if;
        end for;
    end if;
end_DFS;
```

d)
Exactly once for every edge:
n - 1 + n - 2 + n - 3 + ... + 1 = n * (n + 1) / 2

e)
For the WWKWW problem, having a lookup table saves time when the same calculation is required. Instead of repeating the calculation, the algorithm simply looks for the value that is stored from the last calculation. In this case, it is similar that the seen array stores nodes that have been traversed, so the subsequence recursive calls can skip these nodes and avoid extra printing.

7.7 Topoligical sort

When a Tarjan algorithm processes through the first repeated node in a cycle, the node would be marked "started" but not yet "done". This would be a sign of a cycle, so the algorithm can execute its cycle processing subsequence.

7.8 Topoligical sort

When a BFS algorithm runs in a graph that has a cycle, the cycle part will not have a "handle" which can allow every node within the cycle to enter the "ready" queue to be sorted. For that reason, the BFS algorithm will fail when there is a cycle in the graph.

7.10 Preorder and postorder vertex numberings

a)
All nodes from the root to u's children.

b)
If u.pre > v.pre and u.post > v.post, u is a descendant of v.

c)
Since a parent-child relationship is reversed in preorder and postorder, the two nodes will have a reversed positioning in these two orderings.

7.Xa

```
global G = (V,E);
global adj[1...n] stores all egdes starting from 1 to n;
global Ready q;

// prepare every node
for every node k, add its number of edges to its k.val;
push nodes with its val = 0 to q;

procedure BFS(G, k);
    // start from the ready q;
    while q is not empty,
        k <- q.pop;
        for all vertices m connected to k,
            m.val - 1;
            if m.val = 0 then q.append(m);
            recursively call BFS(m);
        end for;
    end while;
    // after the while loop, q should be empty.
    print(k);
end_BFS;
```

7.Xb

```
// High-level Tarjan's Reverse Topoligical Sort
// prepare all nodes
mark all nodes as not done and not started

global G = (V,E);
procedure TarjanDFS(G;;w);
    mark w as started;
    for all neighbors m of w do
        if m is unstarted then TarjanDFS(m);
        else if m is undone then // there is a cycle
            start cycle processing;
        end if;
    end for;
    print(w);
    mark w as done;
end_TarjanDFS;
```

```
7.Y
// preprocessing
// assumption: singleton nodes are centers according to the
definition.
global T = (V, E);
global adj[1...n];
global leaf [];
global center [];
global leftover [];
procedure preprocess(T);
    for v in T do
        v.val <- adj[v].length;
        if v.val = 1 then leaf.append(v);
        else if v.val = 0 then center.append(v);
        else leftover.append(v);
        end if;
    end for;
end_preprocess;

procedure leafBiting(T);
    while leftover ≥ 2 do
        v <- leaf.pop;
        for k in adj[v] do
            leaf.append(k);
            leftover.remove(k);
        end for;
    end while;
    // at this point, leftover should have 1 or 2 nodes that are the
centers
    for v in leftover do
        center.append(v);
    end for;

    for v in center do
        print(v);
    end for;
end_leafBiting;
```

7.Z

a)

```
// driver and preprocessing
global G = (V,E);
global Adj[1...n];
global root[];
procedure driver(Adj[1...n]);
    for k in Adj[1...n] do
        if k is null then
            // k is a leaf
            k.longestoutof <- 0;
            mark k as done;
        else
            mark k as undone;
        end if;
    end for;

    if r in Adj[1...n] is not in other vertices' Adj then
        // r is a root
        root.append(r);
    end if;

    while root is not empty do
        m <- root.pop();
        longest(m);
    end while;
end_driver;


// DFS

procedure longest(vertex m);
    if m is undone then
        for k in Adj[m] do
            m.longestoutof <- max(k.longestoutof) + Ecost(m, k);
        end for;
        mark m as done;
    end if;
end_longest;
```

b)

```
global G = (V,E);
global Adj[1...n];

// prepare all nodes
```

```
mark all nodes as not done and not started;
mark roots as done;
mark all roots' longestinto as 0;
for roots x do
    TarjanDFS(G,x);
end for;

procedure TarjanDFS(G;;w);
    mark w as started;
    for all neighbors m of w do
        if m is unstarted then
            m.longestinto <- max(w.longestinto + Ecost(w,m),
m.longestinto)
            TarjanDFS(m);
        else if m is undone then // there is a cycle
            start cycle processing;
        end if;
    end for;
    print(w);
    mark w as done;
end_TarjanDFS;
```

c)
Use part b's answer as a's input.

d)
Use part a's answer as b's input.