CS180 Solution 3

1. A node v in a directed rooted tree $T = (V, E)$ is an *ancestor* of node u if the path from the root to u passes through v. The *lowest common ancestor(LCA)* of two nodes u and v in T is the deepest node w that is an ancestor of both u and v, where the depth of node is defined as the number of edges on the path from the root to that node. Given two nodes u and v in an directed rooted tree T, design an algorithm that finds the lowest common ancestor in $O(h)$ time, where h is the distance between u and v in the underlying undirected version of T. You are allowed to do $O(|E|)$ preprocessing on the tree before various pairs u and v are given to you, and the cost of preprocessing will not be counted in your LCA algorithm.

   Solution:

   The idea is we can construct the path from u and v to the root, save thm into an array with length compare them level by level from to top.

   Assume the tree is represented in a standard data structure that every node have serval pointers that point to its children

   ```
   PREPROCESSING($T, r$)
     for each child $c$ of $r$
         parent($c$) $\leftarrow r$
         depth($c$) $\leftarrow$depth($r$) + 1
         PREPROCESSING($T, c$)
   ```

   The preprocessing takes $O(|E|)$ time since for each node except root we add a pointer and there are $|E| + 1$ nodes.

   ```
   ANNCESTOR($T$)
     w.l.o.g., assume depth($u$) > depth($v$) and let $d$ = depth($u$) − depth($v$)
     $p \leftarrow u,\ q \leftarrow v$
     for $i \leftarrow 1$ to $d$
         $p \leftarrow$ parent($p$)
     while ($p \neq q$)
         $p \leftarrow$ parent($p$), $q \leftarrow$ parent($q$)
     return $p$
   ```

   The calculation of LCA takes $O(h)$ time because the total number of marching toward the LCA is h.

2. In some country, there are n flights among n cities. Each city has an exactly one outgoing flight but may have no incoming flight or multiple incoming flights. Design an algorithm to find the largest subset S of cities such that each city in S has at least one incoming flight from another city in S. (hint: recall the in-degree array we used in developing the topological sort algorithm in class)

   Solution:

If we consider the cities are represented by $1..n$ and flights are given by the function f, the problem is equivlaent to be the following

Given such a function f in the array representation $f[1..n]$, design an algorithm that find a subset $S \subseteq \{1 \dots n\}$ with the maximum number of elements, such that

(a) the function f maps every element of S to another element of S, i.e., f restrict on S is a bijection(one-to-one mapping.

(b) no two elements of S are mapped to the same element (i.e., f is one-to-one when restricted to S)

This is because every element in S maps to another element and every element in S has at least one element mapped from)

We solve the the problem inductively. If there is only one element in the set , then the subset is just itself since the function must be a one-to-one function. Assume we know how to solve the problem for sets of $n-1$. To reduct a problem with set of n, we claim that any element i such that no other element mapped to it cannot belong to S. Otherwise, the function restricted to S will map from $|S|$ elements to $|S|-1$ elements, and so the function will not be one-to-one. Hence, if there exists such an element that no element maps to it, we can remove them and solve the subproblem by our induction hypothesis. We design an interactive algorithm using the previous idea: repeatedly find an element that has no element maps to, and remove the element.

```
ONE-TO-ONE($f$)
  $S \leftarrow \{1 \dots n\}$
  for $i \leftarrow 1$ to $n$
      counter[$i$] $\leftarrow 0$    \\initialized a counter to count how many elements map to $i$
  for $i \leftarrow 1$ to $n$
      increment counter[$f[i]$]
  for $i \leftarrow 1$ to $n$
      if counter[$i$] $= 0$ then add $i$ in Queue
  while Queue is not empty    \\repeatedly remove the empty element and update
      remove $k$ from the top of the Queue
      $S \leftarrow S - \{k\}$
      decrement counter[$f[k]$]
      if counter[$f[k]$] $= 0$
          add $f[k]$ in Queue
  return $S$
```

The above algorithm is very similar to topological sorting that we repeatedly remove a vertex with indegree 0.

3. A directed graph $G = (V, E)$ is acyclic if there is no cycle in G. Given a directed graph G, design an $O(|E|)$ algorithm to determine whether it is acyclic or not.

Solution:

Depth First Traversal can be used to detect a cycle in a Graph. DFS for a connected graph produces a tree. There is a cycle in a graph only if there is a back edge present in the graph. A back edge is an edge that is from a node to itself (self-loop) or one of its ancestor in the tree produced by DFS.

To detect a back edge, we can keep track of vertices currently in recursion stack of function for DFS traversal. If we reach a vertex that is already in the recursion stack, then there is a cycle in the tree. The edge that connects current vertex to the vertex in the recursion stack is a back edge. DFS visits each edge once. Therefore, the time complexity is $O(|E|)$.

```
isCyclic(i, visited, recStack)
    if resStack[i]
        return True
    if visited[i]
        return False
    visited[i] = True
    recStack[i] = True
    for each neighbor j of i
        if isCyclic(j, visited, recStack)
            return True
    recStack[i] = False
    return False
```

```
Main()
    for $i \leftarrow 1$ to $n$
        if isCyclic(i, visited, recStack) then Return True
    return False
```

4. Given a directed acyclic graph(DAG) $G = (V, E)$ which is just a directed simple path and $|V| = n$. The topological sorting can be solved on G by doing DFS from the source node and numbering nodes from n down to 1 when the DFS colors a node *black*, i.e., reverse post-ordering on the DFS tree. If we are given another DAG G' which are two node-disjoint simple paths, we can extend G' by creating a super source node and two edges from the super source to the two sources of the paths. The topological sorting on extended G' can be solved if we start a DFS from the super source node and allocate numbers from n down to 0 in the manner described above. The super source node will get the number 0, and all the other nodes will be topologically sorted from 1 to n. Given an arbitrary DAG $G = (V, E)$, extend this idea to give an $O(|E|)$ algorithm to solve topological sorting on G by using DFS. Argue the correctness of your algorithm and its $O(|E|)$ time complexity.

   Solution:

   In topological sorting, we can use a temporary stack. We don't print the vertex immediately, we first recursively call topological sorting for all its adjacent vertices, then push it to a stack. Finally, print contents of stack. Note that a vertex is pushed to stack only when all of its adjacent vertices (and their adjacent vertices and so on) are already in stack.

```
TS(v, visited, stack)
    visited[v] = True
    for each outgoing neighbor w for node v:
        if visited[w] == False:
            TS(w, visited, stack)
    stack.push(v)
```

```
TopologicalSort()
  for i in 0 to n-1:
      visited[i] = False
  for i in 0 to n-1:
      if visited[i] == False:
          TS(i, visited, stack)
  while stack is not empty:
      print stack.pop()
```