1. In an undirected graph, the *degree* of a node u, $\texttt{deg}(u)$, is the number of neighbors u has, or equivalently, the number of edges incident upon it. Show that, in any simple, undirected graph with at least two vertices, there must exist nodes u, v such that $\texttt{deg}(u) = \texttt{deg}(v)$.

   Solution: For a node u in a simple graph, $\texttt{deg}(u)$ can take only take on values in the range $0, 1, \ldots, n-1$ where $n = |V|$. Note that a node with degree $n-1$ must have an edge with every other node in the graph. This means that if any node has degree 0, then there cannot be any node with degree $n-1$ and vice versa.

   As a result, there are only $n-1$ possible values for $\texttt{deg}(u)$ to take on and there are n total nodes. Thus, there must be at least two nodes with the same degree value. This is an application of the *pigeonhole principle*.

2. (Dasgupta et al. textbook 3.10) Rewrite the `explore` procedure (Figure 3.3) so that it is non-recursive (that is, explicitly use a stack). The calls to `previsit` and `postvisit` should be positioned so that they have the same effect as in the recursive procedure.

   Solution: At each node, we mark it as visited, previsit it, and then push all of its unvisited children onto the stack. When we encounter a node that has already been visited, we know we have already processed all of its children so we can pop it and postvisit it.

```
procedure explore(G, s)

S = {}
S.push(s)
while S is not empty:
  v = S.top()
  if not visited(v):
    visited(v) = true
    previsit(v)
    for each edge (v, u) ∈ E:
      if not visited(u):  S.push(u)
  else:
    S.pop()
    postvisit(v)
```

3. (Dasgupta et al. textbook 3.11, modified) Without modifying DFS, design a linear-time algorithm which, given an undirected graph G and a particular edge e in it, determines whether G has a cycle containing e.

   Solution: To determine if a cycle exists, we simply build another graph G' which is identical to G only with edge e removed. We then call `explore` on either of the nodes incident to e. If, after the `explore`, the node on the other end of the edge was visited, we know there must exist another path to that node, so with edge e this constitutes a cycle.

```
procedure cycle(G, e)

for all v ∈ V:
  visited(v) = false

G' = (V, E \ e)
(v, u) = e
explore(G', v)
return visited(u)
```

Building G' can be done in $O(|V| + |E|)$ time which is the same as an `explore` call so the total runtime will be linear.

4. (Dasgupta et al. textbook 3.23) Give an efficient algorithm that takes as input a directed acyclic graph $G = (V, E)$, and two vertices $s, t \in V$ and outputs the number of different directed paths from s to t in G.

Solution: For any node $v \in V$, let `numpaths`$[v]$ denote the number of distinct paths from s to v. Observe that for any node v, the number of ways s can reach v is the sum total of all the ways s can reach any of v's parents. That is, `numpaths`$[v]$ will be equal to the sum over the `numpaths` of each of v's parent nodes.

The key then is to process our graph in topological order using this fact. Note that any nodes occurring before s in the topological ordering is, by definition, not reachable from s and so will have `numpaths` equal to 0. Topological sorting takes linear time and then we only need to do one pass through the entire graph so the total runtime will be $O(|V| + |E|)$.

```
procedure countpaths(G, s, t)

for all v ∈ V:
  numpaths[v] = 0
numpaths[s] = 1

for all u ∈ V in decreasing post value:
  for each edge (u, v) ∈ E:
    numpaths[v] = numpaths[u] + numpaths[v]

return numpaths[t]
```

One subtlety to be aware of is that processing the nodes in topological order implicitly means we are running DFS and building the ordering simultaneously as we search. Just getting the `post` numbers is insufficient because it would require sorting which would take longer than linear time.

5. A gossip network is a directed graph where each node represents a person, and each edge (u, v) means that "if u hears some gossip, he/she will immediately pass it on to v". When person $p \in V$ hears some gossip, let $\#_p$ denote the number of people who are guaranteed to hear it too (including p).

   (a) Give an efficient algorithm that takes as input a graph $G = (V, E)$ as well as a specific node p, and computes $\#_p$. What is the running time of your algorithm, as a function of $|V|$ and $|E|$?

Solution: This value is just equal to the number of nodes reachable from p. We can get this easily just from the `pre` and `post` values.

```
procedure nump(G, p)

for all v ∈ V:
  visited(v) = false

explore(G, p)

return (post(p) − pre(p) + 1)/2
```

DFS is linear time so this runs in $O(|V| + |E|)$. A perhaps more obvious alternative would be to run `explore` and manually count the number of visited nodes. In either case the runtime is dominated by DFS so will be $O(|V| + |E|)$.

(b) Given a gossip network G, suppose you want to find a node p with the *smallest* $\#_p$ value. Consider how you might write an algorithm to do this. State concisely what the goal of that algorithm would be using the language of Chapter 3.

Solution: The answer here depends on a few facts.

- All nodes in the same SCC will have the same $\#_p$ value, which is at least the size of the SCC.
- The SCC metagraph is a DAG so will have at least one sink.
- For any non-sink SCC, the $\#_p$ values will be strictly greater than that of its descendants.

Putting all of the above together and stated concisely, this task in graph terms is equivalent to *finding any node in a minimal-size sink SCC.*

(c) At a high level, sketch the steps an algorithm would need to take to find the p from part (b). What is the running time in terms of $|V|$ and $|E|$?

Solution: To actually do this is fairly non-trivial using just the tools of Chapter 3 out of the box. At a high level, we need to do the following:

- Run the SCC algorithm from the text. Note that this only returns an array of size $|V|$ assigning each node a number corresponding to a strongly connected component.
- Walk the edge list. For each edge originating in one SCC and leading to a different SCC, mark the originating SCC as not a sink. At the end, any unmarked SCC will be a sink SCC.
- Count and store the number of nodes in each SCC.
- Of the SCC's which are sinks, find one of minimal size.
- Return any node in that SCC.

None of the steps take longer than linear time if correctly implemented so the total runtime is $O(|V| + |E|)$.