Question 1: In LeLann’s leader election discussed in class, show that when
communication links are not FIFO and the system is asynchronous, a leader
is not necessarily elected.

Answer: Consider the following ring of three processors. Node 0 has identi-
fier 12 (i.e. C[0].id = 12), node 1 has identifier 10 and node 2 has identifier
11.

Now suppose all nodes send their identifiers at the same time. But then
node 1 immediately forwards the value 12 it receives from node 0, and this
value overtakes the value 10 node 1 sent in it’s first step on the link from
node 1 to node 2. In other words, node 2 first receives 12 and then receives
10. Node 2 therefore first sends its own value 11, then the value 12 and
finally the value 10. As a result, node 0 first receives the value 11 and then
it’s own identifier 12. This makes it stop, and decide it is not a leader.
But because it stopped, it never forwards the value 10 still in the message
queue! This means node 1 never receives back it’s own identifier, and never
discovers it should be the leader. Finally, because node 2 did receive the
value 10, which is smaller than it’s own identifier, node 2 will not decide to
be a leader either. We see none of the nodes becomes a leader.

Question 2: In Peterson’s leader election protocol, what happens to the other
nodes when a node becomes a leader?

What could you do about that?

Answer: When a node becomes a leader, then it received its own identity
from both its left an right hand neighbour on the ring. This means all other
nodes are passive (active nodes only send their own identity to their neigh-
bours). Once a node is leader it stops sending messages. Passive nodes only
send messages if they receive one. If we assume the links are FIFO, as soon
as a node becomes leader, there are no other messages in transit. This means
the passive nodes are forever waiting for a next message to arrive.

This could be solved by introducing a clean-up phase where the leader
sends a ’stop listening’ messages clockwise along the ring, that all passive
nodes forward to the next clockwise neighbour, after which they leave the while loop. The code would then look like this

```plaintext
C[i].active = true
C[i].leader = false
while true /* new round */
do if (C[i].leader == true)
    send right 'stop'
    break
else if (C[i].active == true)
    send left C[i].id
    send right C[i].id
    receive right rightid
    receive left leftid
    if ((C[i].id == leftid) /
         (C[i].id == rightid))
        C[i].leader = true
    else if ((C[i].id < leftid) \n           (C[i].id < rightid))
        C[i].active = false
else /* passive */
    receive left id ; send right id
    if id==stop then break
    receive right id ; send left id
```

**Question 3**: Design a leader election protocol that works on general undirected graphs, provided they are connected.

What is the message complexity?

**Answer**: (The idea was not to Google for exiting solutions but to try to think of a nice algorithm yourself.)

There are several approaches. One approach is to embed a virtual ring on top of the general graph you are given. This is the easiest way. But this is not very efficient. In the worst case you need a virtual ring of length $n^2$, where $n$ is the number of nodes in the original graph. (This will be shown in a late lecture.) Also, you could argue this is cheating, because if you have arrange the nodes of the graph in such a virtual ring, then you can also just appoint some node the leader...

Another approach is to create an algorithm that floods identifiers over the network in a similar style as LeLann’s algorithm. However, you need a way to determine that all nodes received all there is to know about the identities of all nodes in the graph.

The idea is to let each node create a spanning tree of the graph with itself as root. While building the trees (note: $n$ of them), the identifier of the root is pushed down the tree. This uses messages of the form $(id,v)$ where $id$ specifies the type of the messages, and where $v$ is the value of the identifier.
being sent. A node that receives a new identifier (and joins the spanning tree for that identifier) forwards it to all other neighbours (excluding the link it received it from). Instead, if it already is a member of the tree, it immediately says so. (It sends a \((\text{member},v)\) message back.) If all neighbours are already member, then the node is a leaf. Once a leaf is reached, a boolean flag is propagated back to the root indicating whether the root identifier is larger than all identifiers in the subtree. This uses a message \((\text{flag},v)\). The root node that receives true is elected leader.

The protocol for node \(i\) would look something like this (where \(\text{neighbours}[i]\)) denotes the nodes that are immediate neighbours of \(i\). We assume a bidirectional graph.

\[
I = \{ C[i].id \}
\]
\[
C[i].\text{leader} = \text{propagate}( C[i].id, \text{neighbours}[i] )
\]

while true
  do receive \((id,v)\) from \(j\)
    if \(v\) in \(I\)
      send \((\text{member},v)\) to \(j\)
    else
      (spawn) \(v = \text{propagate}( v, \text{neighbours}[i] - \{j\} )\)
      send \((\text{flag},v)\) to \(j\)

\text{propagate}(v,S):
  \[
  I = I + \{v\}
  \]
  \[
  \text{largest} = \text{true}
  \]
  do for all \(j\) in \(S\)
    do send \((id,v)\) to \(j\)
    receive \(m\) from \(j\)
    case \(m\) of
      \((\text{member},v)\): do nothing
      \((\text{flag},v)\) : \(\text{largest} = \text{largest} \text{ AND } v\)
  return \(\text{largest} \text{ AND } C[i].id < v\)

In terms of message complexity, each identifier for each node is sent exactly once over each edge. So \(|E|\) messages of type \((id,v)\) are sent for each identifier. In response, either a \((\text{member},v)\) or a \((\text{flag},v)\) messages is always sent (but never both). So in total this protocol sends \(|V| \ast |E| \ast 2\) messages.