Exercise 1

I will only show steps at which we append to the list, for brevity. The state of the DFS tree is shown in parenthesis notation. We append to a list exactly when we cannot expand any further <-> we close a parenthesis:

1. `[]`, `(A (C (D (F (G G)`
2. `[G]`, `(A (C (D (F (GG) (HH)`
3. `[H,G]`, `(A (C (D (F (GG) (HH) F)`
4. `[F,H,G]`, `(A (C (D (F (GG) (HH) F) D)`
5. `[D,F,H,G]`, `(A (C (D (F (GG) (HH) F) D) (EE)`
6. `[E,D,F,H,G]`, `(A (C (D (F (GG) (HH) F) D) (EE) C)`
7. `[C,E,D,F,H,G]`, `(A (C (D (F (GG) (HH) F) D) (EE) C) (BB)`
8. `[B,C,E,D,F,H,G]`, `(A (C (D (F (GG) (HH) F) D) (EE) C) (BB) A)`
9. `[A,B,C,E,D,F,H,G]`, `(A (C (D (F (GG) (HH) F) D) (EE) C) (BB) A)`

How many t.o.'s are possible? All valid topological orders should respect the following partial ordering relations:

`[A,B] > C > [D,E] > F > [G,H]`

So we can only swap orders within `[A,B]`, `[D,E]` and `[G,H]`, which we can do 3 times independently, so by the product rule there are 2 * 2 * 2 = 8 possible topological orderings.

Exercise 2

```
DFS-Visit(G, u):
    color[u] = GRAY
    time = time + 1
    d[u] = time

    for each v in Adj[G][u]:
        do if color[v] == WHITE then:
            p[v] = u
            edge_type[ (u,v) ] = TREE_EDGE
            DFS-Visit(G, v)

        else if color[v] == GRAY then:
            edge_type[ (u,v) ] = BACK_EDGE

        else if color[v] == BLACK then:
            if d[u] < d[v] then:
                edge_type[ (u,v) ] = FORWARD_EDGE
            else:
                edge_type[ (u,v) ] = CROSS_EDGE


    color[u] = BLACK
    time = time + 1
    f[u] = time
```

Because this is precisely how we recognize FORWARD, BACK, CROSS and TREE edges. Since DFS encounters all edges once, we will have determined the type of edge for every edge in the graph.

Exercise 3

We can identify the DAG with a partial order, where u < v iff. there is a path from v to u (using directed edges). This defines a partial order because this relation is transitive, reflexive if we set u <= v iff. u == v or u < v, and symmetric because a DAG contains no cycles (suppose u < v and v < u, then there is a path from u to v and from v to u, but these are along different edges because these are directed edges. So that means `u -> v -> u` is not a backtrack but a proper circuit, and hence the DAG must contain a cycle, contradiction).

A path that visits all vertices of G is simply a topological order on G where there is an edge u -> v for every pair of consecutive elements {.. u, v ..}. Thus, we can only have a path that visits all vertices of G iff. G has only one unique possible topological order if such a path exists, because a topological order has to extend all order constraints imposed by the connections in the graph, and thus any topological order has to satisfy the total order imposed by the path v1 -> ... -> vn that visits all edges.

This means that we only have to find a topological order with the topological ordering algorithm (which runs in linear time O(V + E)) and then check whether the resulting list is indeed a path throught the DAG.

Checking whether something is a path means that for every of the `|V|-1` pairs of consecutive vertices v{i}, v{i+1} in the list, we have to check whether there is an edge from v{i} to v{i+1}. We assume that we can do this in O(1) time (which is true if we store the graph using an adjacency matrix, but for the adjacency list/set representation, it depends on the datastructure in which we store the adjacent neighbours of a vertex (could be O (lg n) if we store them in some kind of search tree)).

In total , we have an O(V+E) step for the topological sort and an O(V-1) step for checking the path, which in total gives an O(V+E) complexity of the algorithm. If there is a path covering all edges, then we not only know but we also at the same time found it using this algorithm.

Exercise 4

If we knew what the last courses were that we could complete, then we knew that every course that is a **direct** predecessor of one of these courses (i.e. there is an **edge** u -> v) can be completed at last one semester before we complete the last course.

We will keep for every node a new variable named LTC, or Last Time to Complete. This is the least number of semesters that is needed from the beginning of this course before we can complete the final semester of final course(s) in the dependency graph. So for example, in the graph of exercise 1, LTC[G] = LTC[H] = 1 and LTC[F] = 2.

We see that, for every v in V[G]:

if v is an end node of a DFS-branch, i.e. f[v] = d[v] + 1, then f[v] is a final course and LTC[v] = 0

Otherwise:

LTC[v] = max {LTC[child], child in Adj[v]} + 1

where we define max EMPTY_SET = 0 (empty sets are encountered precisely when there are no edges exiting a node).

This is even well-defined recursion because we always arrive at a base case: eventually {LTC[child], child in Adj[v]} will become empty because otherwise we would either have an infinite graph or a cycle in our graph.

Let's implement it:

```
LTC(G)
    for each vertex u in V[G] do:
        color[u] = WHITE
        ltc[u] = 0

    time = 0

    for each vertex u in V[G] do:
        if color[u] == WHITE then:
            LTC-Visit(G, u)


LTC-Visit(G, u):

    for each v in Adj[G][u] do:
        if color[v] == WHITE then:
            ltc[u] = max ( ltc[u], LTC-Visit(G, v) )

        else if color[v] == BLACK then:
            if d[u] < d[v] then:
                // forward edges can never create a critical path,
                // that is, a path with a longer chain of dependencies
                // than the path between u and v that was already explored.
                // this is because forward edges are "shortcuts"
                // along the dfs arm.
                nothing
            else:
                // cross-edges may increase the LTC because they indicate
                // dependencies with another branch of the BFS tree.
                ltc[u] = max ( ltc[u], LTC-Visit(G, v) )

    if Adj[G][u] == EMPTY then:
        ltc[u] = 1

    color[u] = BLACK
```

We don't care about GRAY nodes because if we ever see a gray node during expansion then there is a cycle, and we are working with a DAG which has no cycles.

If we expand a node u and see a BLACK node v, then we have to check whether the current depth + 1 is not larger than LTC[v], and if it is, we change LTC[v] to the current depth + 1. Also see that this can only happen if (u,v) is a cross edge. Forward egdes are "shortcuts" down the DFS tree. But this as an aside.

If we expand a node u and see a WHITE node v, then we determine the LTC of v first, and this recursion terminates because we are in a DAG so eventually the DFS-arm will end, and we then know by definition that it only takes one semester to finish the curriculum.

When we start somewhere in the middle of a path, say at u with predecessor v, this is not an issue because we will expand, paint u black and the next expansion that visits u from v will give v the appropriate LTC.

When we have determined the LTC of every white node in the graph, we take the maximum of that, which is an at most O(V) operation, if we have created $|V|$ disjoint search trees (this happens exactly when there are no dependencies between the courses). Because DFS (which LTC is an instance of) takes O(V+E) visits, this makes LTC O(V+E) + O(V) = O(V+E).

Exercise 5

If DFS uses DFS-visit in the following form:

```
DFS-Visit(G, u):
    color[u] = GRAY
    time = time + 1
    d[u] = time

    for each v in Adj[G][u]:
        do if color[v] == WHITE then:
            p[v] = u
            edge_type[ (u,v) ] = TREE_EDGE
            DFS-Visit(G, v)

        else if color[v] == GRAY then:
            edge_type[ (u,v) ] = BACK_EDGE

        else if color[v] == BLACK then:
            if d[u] < d[v] then:
                edge_type[ (u,v) ] = FORWARD_EDGE
            else:
                edge_type[ (u,v) ] = CROSS_EDGE


    color[u] = BLACK
    time = time + 1
    f[u] = time
```

Then if we encounter a GRAY node during expansion, this indicates a loop because the DFS-arm loops back on itself. There is a loop iff. one of the edges is a back edge, because if there is a loop, then this will produce a back edge somewhere in the loop.

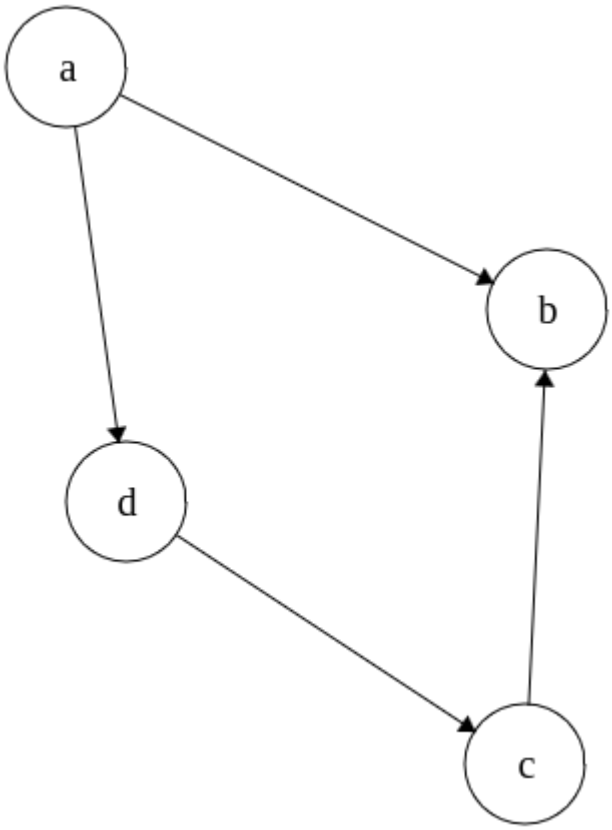
Since DFS explores all edges, we can be certain that we find all back-edges if they exist and thus there is a loop iff. there is a back edge.

Now look at the student's program:

The things that are wrong with the student's program:

1. We only expand one randomly chosen element of the graph. But if our graph consists of two disconnected subcomponents, one of which is acyclic while the other has a loop, then this algorithm will sometimes discover the loop and sometimes not discover the loop.

2. Instead of using GRAY to indicate that a node has branches that are not completely expanded yet, and BLACK to indicate that the node is completely expanded, the student's algorithm either MARKs a node or does not mark. But then we cannot differentiate between back edges, and cross/forward edges. And this is exactly what cc(G,x) does: if it finds a marked node while expanding, it assumes that this node is GREY and thus that if found a back edge, hence a cycle. But this does not have to be the case, for example:



Here, we could pick a randomly, then explore a with cc and get the following dfs tree:

```
(a (bb) (d (cc) d) a)
```

At node c, we expand and discover b, which is MARKed hence the student assumes a cycle. But b would have been BLACK in the original DFS, and thus we don't have a back edge and hence no cycle. So the algorithm would incorrectly return true and print the not-cycle

```
b
c
d
a
```

We will now correct the mistakes:

```
procedure findCycle(G)
    for all x ∈ V do
        mark_expanded[x] = false
        mark_expanding[x] = false
        parent[x] ← nil

    loop = false

    for all x in V do
        loop = cc(G, x)

    if loop = false then
        WriteLn(”No loop”)
end procedure



procedure cc(G, x)
    mark[x] ← true

    for all y ∈ neighbourhood(x, G) do
        if (not mark_expanding[y])
        and (not mark_expanded[y]) then
            // WHITE node
            parent[y] ← x
            if cc(G, y) = true then
                return True

        else if mark_expanding[y]
            // GREY node
            display(y)
            z←x
            while z ≠ y do
                display(z)
                z ← parent[z]
            return true

    return false
end procedure
```