1. (a) My algorithm start by looping through the columns of the adjacency matrix until it finds a column of all ones, it then checks the corresponding row to see if it is all zeros except the one on the main diagonal.

```
FindMovieStar(A, n)
 1  potStar gets0
 2  for j in 1..n
 3        do
 4           if potStar = 0
 5           isStar ← True
 6           for i in 1..n
 7                 do
 8                    if A[i, j] = 0
 9                      then
10                             isStar ← False
11           if isStar = True
12             then
13                    potStar ← j
14  if potStar ≠ 0
15     then
16           for j in 1..n
17                 do
18                    if A[potStar, j] = 1 ⋀ potStar ≠ j
19                      then
20                             return 0
21  return potStar
```

In order to check for a movie star the algorithm must see if everyone knows the movie star and the movie star knows nobody else. The first for loops check if there is a patron that everybody knows. The second loop inside the if statement takes that person and sees if they know anyone other than themselves. This algorithm relies on there only being one movie star which is true because of how a movie star is defined. If a movie star knows only themselves they cannot know another movie star in the bar, hence there is no other movie star.

(b) The worst case for my algorithm is that the last person we check is the movie star. In this case my algorithm takes $n^2 + n$ matrix accesses. n^2 for finding someone everyone knows and n for checking that the person doesn't know anybody else.

2. (a) See Figure 1.

(b) There are 10 cross edges, 1 forward edge and no back edges

(c) There are no back edges and therefore no cycles. This means that no course can require itself and that there is an order that the courses can be taken in.

(d) 5,6,3,1,4,2,7 using the topological sort from pg 613 in CLRS and the tutorial.

(e) See Figure 2

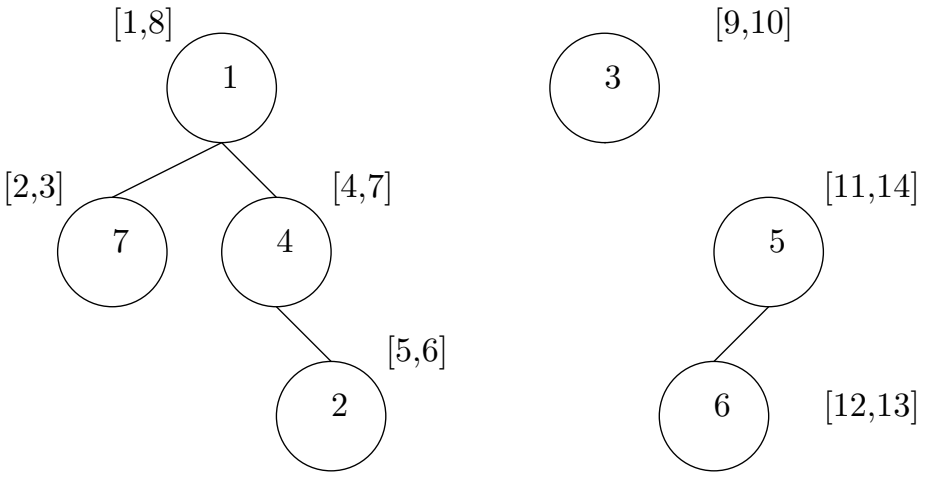

Figure 1:

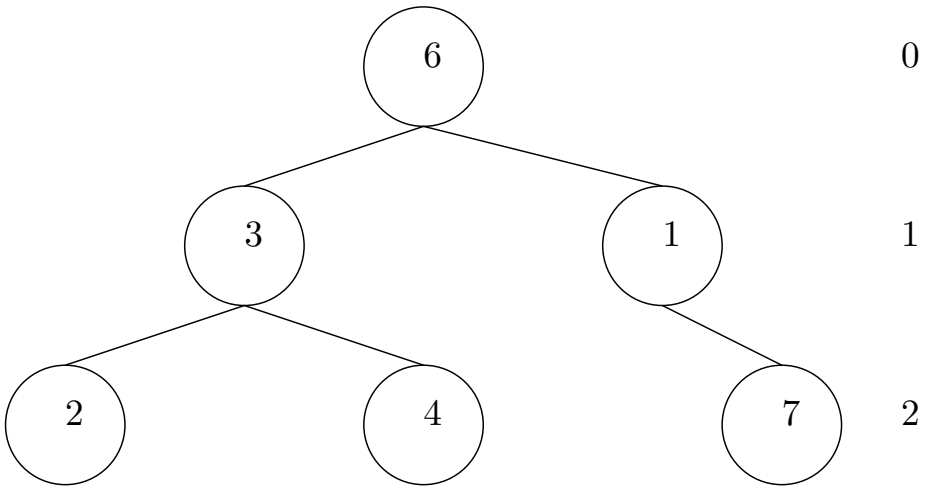

Figure 2:

3. (a) Suppose that a graph G has no sources. Then every node in G has at least one other node connected to it. Consider the resulting forest from a DFS algorithm, in particular consider the root of each discovery tree. Every root must have either a back edge or a cross edge leading to it. If there is a back edge we are done there is a cycle. If there are no back edges and only cross edges we start at one root and follow down the discovery tree until we reach a cross edge to another discovery tree. We continue until we end up back at that root, and have therefore found the path of the cycle.

   I have shown that if G has no sources then it has a cycle. Therefore if G is acyclic, then it has at least one source.

   (b) Note if G is acyclic then the tree resulting from deleting a node from G is also acyclic.

   If G is acyclic then G has at least one source by the previous question. By the note above deleting a node also results in an acyclic graph which must also have at least one source. This will continue on until the graph is empty.

   If G has a cycle, then if we isolate that cycle from the graph there are no sources within that cycle. Removing the nodes outside of the cycle will not result in creating sources within the cycle. So therefore source-deletion will delete everything from the graph except that cycle. Which means that at the end the graph will not be empty.

   Therefore G is acyclic if and only if source-deletion results in an empty graph.

   (c) My algorithm starts by finding all sources in G ($O(m+n)$) keeping track of all the sources in a linked list as well as a size variable. I then traverse the list, deleting all edges from each source and if appropriate add more sources to the end of the list and increment the size variable. ($O(m)$). When I reach the end of the list I compare the size variable to n, if they are equal there are no cycles. There for my algorithm runs in $O(m+n)$

   In the following code I assume that the linked lists I implement are iterable and therefore I can use a for loop.

```
hasCycles(n, E)
 1  S ←Empty iterable linked list
 2  size ← 0
 3  for i ∈ 1..n
 4        do
 5           for e ∈ A[i]
 6                 do
 7                    I(e) ← I(e) + 1
 8  for i ∈ 1 . . n
 9        do
10           if I(i) = 0
11              then
12                    size ← size +1
13                    Append(S, i)
14  for i ∈ S.value
15        do
16           for e ∈ A[i].value
17                 do
18                    I(e) = I(e) − 1
19                    if I(e) = 0
20                       then
21                             size ← size +1
22                             Append(S, e)
23  if size = n
24     then
25           return False
26  return True
```