CS 180 Midterm

1.
    a. True
    b. True
    c. True
    d. False
    e. False

2.

Algorithm:

Initialize Unreached Vector
Run BFS on the complement graph.
Check all vertices that have not been reached yet.
For each of these vertices:
Check if there is an between the current node from BFS and this node
If there is an edge in the original graph then:
There will be at most O(X) of these things
OR
There exists no such edge and you can reach the vertex in the complement graph in also O(X)

Correctness: We want to initialize an unreached vector to contain all the unreached nodes as well as to see if there is an edge set that could possibly be used. Then, we run BFS on the complement graph. The Idea is that we want to check if the complement of this undirected graph is connected since the current non-adjacency list is giving us pairs of nodes that do NOT have an edge. So, we want the opposite and this would be through a complement graph. For convenience sake, let X be the number of vertices + number of edges

Time Complexity: The time complexity of this algorithm is O(m) since m represents the number of elements in the non adjacency list. Because this is a connected graph, m represents the number of total non-edges*2. So the real number of non edges in m/2. So, we can say time complexity is O(X). This is linear time complexity since m is the number of elements represented in the non adjacency-lists times two and X represents edges + vertices.

3.

Algorithm:

Use Topological Sort in order to get the topological order of the Directed acyclic Graph in an array X
For all elements in Array X, except the last element:
    If ∃ no edge from the current node to the next one then:
        ∃ no such path
    Endif
Endfor
∃ a path

Correctness: Every single graph that is directed and acyclic or a directed acyclic graph (DAG) has a topological ordering. If we want to visit each node or vertex only once, ∃ an edges between every consecutive node in this topological ordering. Otherwise, this means a node or a vertex can only be reached from a node that has already been visited.

Time Complexity: For topological sort, the runtime is O(|E| + |V|). How we get here is that we first know that checking all the nodes will be linear and therefore O(|V|), which means that the algorithm runs in O(m) times, as m >= n.

4.

```
//If we get an array of n integers, this returns a set of partitions that has indices where the partitions have the same values
getPartition(Array):
    Initialize Partition = {}
    Initialize addedElements = {}
    For each element in Array:
        Query addedElements
        Query (element)U(addedElements)
        If Queries match:
            searchForPartition(Partition, 1, element)
            Store element in found partition set that is a part of Partition
        Endif
        Else:
            Make new partition set with element
            Partition += new partition set
        Store element in addedElements
        Endelse
    Endfor
    Return Partition

//Given some partition Set, the current partition's indice, and the element that needs to be placed, return an index of the partition that we are looking for
searchForPartition(Set, Indice, element):
    Return index if only one partition set is left

    Query(Left Half of Set)U(element) //left side
    Return index if query is 1
    Query the left half of the set

    Query(Right Half of Set)U(element) //right
    Return index if query is 1
    Query the right half of the set

    If Queries on the Left are the same or match:
        Return searchForPartition(Left Half, Indice, element)
    Else:
        Mid = Integer(Set size / 2)
        Return searchForPartition(Right Half, Indice+mid, element)
```

Justification:

For the first function or part of this algorithm, Partition was for storing the groups of values and addedElements was used to do the same thing except store elements that were already in Partition. Then, for each element, we want to check if queries show that the element is already in a partition, so we can call the second function to find where to put it. If the element is not in a partition, however, then it is ideal to create a new partition for element and place it in the Partition. After this whole part, Partition would contain the groupings that we want.

If there is 1 partition left in the set, then element has to belong to it.
If the Query(Left Half of Set)U(element) is 1, then we have found the partition!
If the Query(Right Half of Set)U(element) is 1, then we have also found the partition!
If we have not found it, then we can make the search field the left half if left queries match. This means element's partition will belong to the left half.
Else, it would be the right half that would be the search field for similar but opposite reasons as above.
We keep going until we find a partition set that element belongs to.

Time Complexity:

Since we use a loop to go through each element of the array, this would be O(n), a standard linear time complexity. Then when we are searching for the partition, this is kind of like binary search in that we narrow the searching girls to half of the original set. This means that with each recursive call on itself, we would make the search smaller by half which would be O(logn) times. Therefore, this algorithm partitions elements in O(nlogn) queries.