#1

Input: directed graph G

Result:

If there is node v with no incoming edges
  Topological sort run
Else, if there is no node v with no incoming edges we have a cycle
  Choose arbitrary vertex j
  Create empty set A
  Until we re-visit a node k for first time
    Traverse from incoming edges starting with first edge in j's adjacency list of incoming edges
    Record traversed nodes in A.
  Output Set A as cycle that shows G is not a DAG.

Analysis: If there is always a node with no incoming edges, a topological ordering will be given. However, the second case, which there is a node with at least one incoming edge. There will be a cycle in this case, and the algorithm just goes to each incoming edge until eventually re-visiting a node.

Runtime: The runtime here will go through all edges and vertices at worst case, as topological sort normally is. So, it is $O(m+n)$. Traversing through the graph backward takes $O(n)$, this is when we are trying to find the cycle. Essentially, we are reversing the graph when finding the cycle which is $O(m)$. Together, the runtime is $O(m+n)$, when trying to find the cycle.

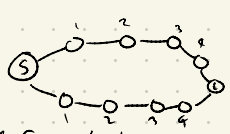
#2 Exercise 4

Construct a graph G with n nodes, the butterflies, and m edges, the judgements.
Have each node have ability to be coloured blue or red.
Pick arbitrary node j and color it blue
For each judgement between j and adjacent node k
  If C(j,k) is marked "same"
    Mark them the same color
  If C(j,k) is marked "different"
    Mark k a different color than j
  If j and k are already colored
    If C(j,k) is marked "same" and j and k are colored different
      Labeling inconsistent → end
    Else if C(j,k) is marked "different" and j and k are colored same
      Labeling inconsistant → end
  Have j = k
  Repeat until every node is colored
Every node has been checked and there are no discrepancies, so labeled

Analysis: First, the graph is created with the butterflies and judgements. Then, we run the 2 color algorithm to determine if the graph is consistent (bipartite) or not. It is very similar to the BFS algorithm.

Runtime: We must go through every node and graph so: $O(m+n)$, it is close to the BFS algorithm. Constructing the graph of butterflies and judgements will take the same runtime as we are creating all the vertices and edges. Overall it is $O(m+n)$.

#3 Exercise 9



n=10

s to t = 4 ≯ 5

a) Proof that v exists

If we say that v does not exist, and there are two different paths from s to t that are greater than n/2. This is not true as shown above. Also, you would not have enough nodes for both paths if you had 2 paths greater than n/2. Proven by contradiction.

b. Algorithm.

Given graph G
Run BFS of G, starting with node s.
For each level i, construct the list L[i] of nodes in level i.
Find a level j, where $1 \leq j \leq n/2$, L[j] only has 1 node, label it k.
Output k.

Analysis: As it was proven before, there must be at least one single node at a layer. There might be multiple, but there is at least one. This one node will be our node v, as there cannot be a path from s to t.

Run Time: BFS is run first, which is O(m+n), Finding the level j should take O(n). So, the run time ends up being O(m+n).

#4 Exercise 11

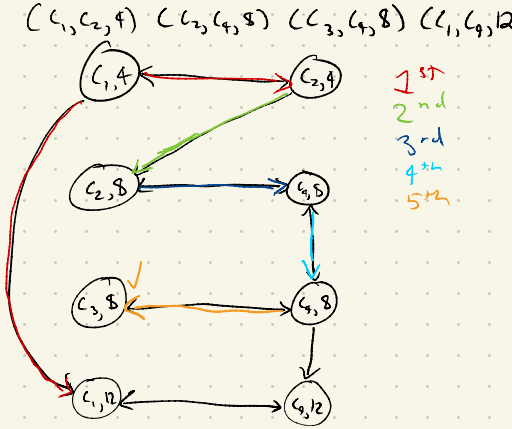
We must find an algorithm that given trace data, will decide whether a virus introduced at computer C_a at time x could have infected computer C_B at time y

Inputs: Computer C_a, Computer C_B, set of trace data S, time x, time y
Result:
Ignore triples with time earlier than x and later than y
For a triple (C_i, C_j, t)
Create node (C_i, t) and (C_j, t) with edges between them in both directions
Let previous node with latest time of C_i point to (C_i, t)
if times are the same, (C_i, t) points back
Let previous node with latest time of C_j point to (C_j, t)
if times are the same (C_j, t) points back

Run BFS from earliest node of C_A to check if it reaches C_B.
if C_B is reached, it means C_A could have infected C_B by time y.
else C_B not reached, C_B is not infected.

Analysis: First, a directed graph is created, with each node having two values, the computer and time accessed. The directions indicate a connection between computers. So, when BFS is run from the first infected computer, we can see if C_B is eventually reached and infected at the time specified.

Runtime: $O(m+n)$ as BFS runs in $O(m+n)$. The worst case is going through every single node and edge, as C_B is the last computer connected.



Running BFS from $(C_1, 2)$ gets to $(C_3, 8)$.

#5 Exercise 12

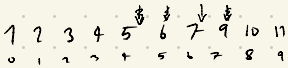
Construct a graph G:
  For each person P_i
    Let node b_i equal birth date
    Let node d_i equal death date
  If P_i died before P_j was born
    Include edge (d_i, b_j)
  If lifespans of P_i and P_j overlap
    Include edges (b_i, d_j) and (b_j, d_i)

If we can run a topological sort on graph G
  We take order of birth and death dates of all people,
  then we have an ordering consistent with provided info. ⟶ end
Else we cannot, G has a cycle
  Info not consistent ⟶ end

Analysis: First, we construct the graph with two nodes for each person, their birth date and death date. The two facts we are given are implemented into creating where each edge will direct to. After the graph is created, we see if a topological sort can be run through the graph, which means there is no cycle, and if it can, then we have the ordering we want and the graph is consistent. Otherwise, there is a cycle and the graph is inconsistent.

Runtime: Topological Sort takes $O(m+n)$. Creating the graph will also be $O(m+n)$, so in total, the algorithm runs in $O(m+n)$

#6a)



Given array A with n elements
Let k = 1, j = n
While j - k > 1 do
  Set l = $\lfloor \frac{j+k}{2} \rfloor$
  if A[l]-1 != A[l-1]
    Output A[l]-1 ——> end
  else if A[l]-1 = l
    set j = l
  else
    set k = l
end
if A[k]-1 != A[k-1]
  Output A[k]-1
else if A[j]-1 != A[j-1]
  Output A[j]-1
else
  Output "Error"

Analysis: If A[l]-1 = l, we search the right
if A[l]-2 = l, we search the left
It is binary search with a small difference.

Runtime: O(log N), binary search runtime, not much else to it.

b)

Given array A with n elements
let sum = $\frac{(n+1)(n+2)}{2}$
let newsum = 0, i = 0
For n elements in A while i != A.length
  newsum += A[i]
return newsum - sum

Analysis: This calculates the sum from 1 to n+1, then finds the sum of the given array. The difference of this is the missing number.

Runtime: O(n). Calculating the sum takes O(1), but going through the array will take O(n).