1. Suppose we have an undirected, connected graph G = (V, E), and a specific vertex u ∈ V . Suppose we compute a depth-first search tree rooted at u and obtain a tree T that includes all nodes of G. Suppose we then compute a breadth-first search tree rooted at u and obtain the same tree T. Prove that G = T. (In other words, if T is both a depth-first search tree and a breadth-first search tree rooted at u, then G cannot contain any edges that do not belong to T.)

It is known (and stated in the problem statement) that if the graph T produced by doing both a BFS and a DFS on graph G = (V, E) from vertex u ∈ V is equal for both the BFS implementation and the DFS implementation, then T is a tree.

Suppose the input graph G is not a tree (the graph has a cycle). This would indicate that G has vertices {v1 -> v2 -> … ->vn -> v1} ∈ V. In the DFS tree created, vertices v1, v2, … , vn will all be on the same path moving from the root to the leaf. However, for the BFS tree created from the undirected graph G, the vertices v1, v2, … , vn will create at least two branches, since the first node visited in {v1, v2, … , vn} will have an edge to at least two other nodes in {v1, v2, … , vn}. The problem statement indicates this is not the case, since tree produced by both BFS and DFS are identical. This proves by contradiction that G = (V, E) is a tree, and that u ∈ V is the root of the tree.

By definition, a given tree with n vertices has n-1 edges. Both BFS and DFS require traversing n-1 edges to reach n vertices in a connected graph. Given that the graph is connected, all vertices will be visited by both BFS and DFS. Given that each vertex is visited and visiting n vertices requires traveling on at least n-1 edges, and that G = (V, E) has been proven to be a tree with n-1 edges, both BFS and DFS must have traveled on each edge in E. If T has all the same vertices and edges as G, then G = T.

2. Suppose you and your friend Alanis live together with n − 2 other people at a popular "cooperative" apartment. Over the next n nights, each of you is supposed to cook dinner for the co-op exactly once, so some one cooks on each of the nights. To make things interesting, everyone has scheduling conflicts on some of the nights (e.g., exams, deadlines at work, basketball games, etc.), so deciding who should cook on which night becomes a tricky task. For concreteness, let's label the people {p1, ..., pn} and the nights {d1, ..., dn}. Then for person pi , associate a set of nights Si ⊂ {d1, ..., dn} when they are not available to cook. A feasible dinner schedule is defined to be an assignment of each person in the co-op to a different night such that each person cooks on exactly one night, then there is someone to cook on each night, and if pi cooks on night dj , then dj 6∈ Si .
   a) [10 points] Describe a bipartite graph G such that G has a perfect matching if and only if there is a feasible dinner schedule for the co-op.

A bipartite graph is a graph whose vertices can be divided into two independent sets L and R such that every edge either connects from L to R or from R to L (so no edges go L to L or R to R).

Our graph will be a bipartite graph G(V, E) with each person {p1, …, pn} making up the set L and each day {d1, …, dn} making up the set R. Since there are an equal number of people and days on the schedule, it follows that there will be an equal number of nodes in L and R.

For each person p_i in L we can define edges from p_i to each day d_i that the person is available. This will populate our bipartite graph with edges between the nodes such that each edge has one vertex in L and one vertex in R.

Let perfect matching be defined by the set of edges $E_p \in E$ such that each vertex in L and each vertex in R has exactly one attached edge in E_p. Perfect matching, then, is a special case of maximum bipartite matching where the cardinality of the matching is equal to the number of nodes in L and R. In other words, perfect matching is the set of edges E_p such that each person in L and each day in R has a single edge attached.

If we can find a set of edges E_p such that there is perfect matching between L and R, then we know we can create a feasible dinner schedule. Each edge will match each person in L with a day in R. And we know that the edges of the bipartite graph represent a day that the person can cook, thus we have a bipartite graph G such that there is perfect matching if there is a feasible dinner schedule for the co-op.

b) [10 points] Your friend Alanis takes on the task of trying to construct a feasible dinner schedule. After great effort, she constructs what she claims is a feasible schedule and then heads off to work for the day. Unfortunately, when you look at the schedule she created, you notice a big problem– n − 2 of the people at the co-op are assigned to different nights on which they are available (no problem there), but for the other two people pi and pj , and the other two days dk and dl , you discover she has accidentally assigned both pi and pj to cook on night dk and no one to cook on night dl . You want to fix this schedule without having to recompute everything from scratch. Show that it is possible, using her "almost correct" schedule, to decide in only O(n^2) time whether there exists a feasible dinner schedule for the co-op. If one exists, your algorithm should also provide that schedule.

The algorithm below is based on the Ford-Fulkerson algorithm for finding the max flow in a bipartite graph. Alanis has done good work to determine a matching of n-1 in the bipartite graph. Now we need to just do one iteration of the Ford-Fulkerson method to determine if we can increase the matching by 1, which would give us a matching of n, or perfect matching in a bipartite graph.

```
# algorithm is provided with:
# G: The complete, directed, bipartite graph described in part a
# g_alanis: the directed, bipartite graph determined by Alanis, sans the incorrect edges
# p_mismark: the two people Alanis mismarked on her graph
# n_mismark: the two nights Alanis mismarked on her graph (the second night has no person assigned)
def correct_schedule(G, g_alanis, p_mismark, n_mismark):
  for edge in g_alanis:
    G.reverse(edge)                                # we reverse the direction of edges Alanis found
  # add the source and sink nodes
  G.add_vertex(source)
  G.add_vertex(sink)

  # check if Alanis mismarked person 1 (at index 0 in p_mismark)
  G1 = G
  G1.add_edge(source, p_mismark[0])
  G1.add_edge(n_mismark[1], sink)
  path = BFS(G1, source, sink)                    # do BFS from source to sink on G1 and return path
  if path:
    return get_schedule(g_alanis, path)           # return the schedule based on Alanis graph and path

  # check if Alanis mismarked person 2 (at index 1 in p_mismark)
  G2 = G
  G2.add_edge(source, p_mismark[0])
  G2.add_edge(n_mismark[1], sink)
  path = BFS(G2, source, sink)                    # do BFS from source to sink on G2 and return path
  if path:
    return get_schedule(g_alanis, path)           # return the schedule based on Alanis graph and path
  return false                                    # if there is no possible schedule, return false

def get_schedule(g_alanis, path):                 # get the schedule from Alanis schedule and path
  for edge in path:
    if edge.has(source) or edge.has(sink):        # if we have a source or sink node attached ignore edge
      # do nothing
    elif edge in g_alanis:
      g_alanis.remove_edge(edge)                  # if edge in path was in the original schedule, remove
    else:
      g_alanis.add_edge(edge)                     # if edge in path was not in original schedule, add it
  return g_alanis
```

As we explained above, this algorithm is executing one cycle of the Ford-Fulkerson method to see if we can increment the flow of the bipartite graph. If we can, then we will have perfect matching of the bipartite graph and the schedule will be complete. If not, we simply return false, which indicates that there is no perfect matching that exists.

The first part of the algorithm basically just preps the initial schedule graph G for the bfs. We take G as a directed graph where each of the edges point from L (the bipartite half denoting the people) to R (the bipartite half denoting the days). First, for each correct edge in Alanis' schedule, we reverse the edge, meaning the edge is pointing from R to L. This will represent our residual capacity for Ford-Fulkerson. Since there are at most O(n-2) edges in Alanis' initial schedule, we can do this in O(n) time. Next we add the source and sink vertexes to the graph, each operation taking O(1) time, for a total of O(n) + O(1) + O(1) = O(n) time.

Since there are two people scheduled on one night, we must consider that the solution may require that either of the two people must be scheduled for that night. Thus for both people we must assume the person will cook on the night originally assigned and we use BFS to check whether we can increment the flow by one. If the BFS is successful along the bipartite graph with the residual capacity edges from the source to the sink then we know that there exists a set of perfect matching in the bipartite graph, and we know we can present a valid schedule. If this is not the case for either person, then we know we cannot increment the flow anymore, and we simply return false. Adding the source and sink nodes takes O(1) time, however BFS typically takes O(V + E) time. In a given bipartite graph with V vertexes, there will be V/2 vertexes on each side. With just V/2 vertexes on each side, there can be no more than V^2/4 edges in the graph. Now that we know how many edges there are, a BFS will take no more than O(V + V^2/4) = O(V^2) time. Thus for n people/nights, we can perform BFS in O(n^2) time.

The problem statement designates that if there exists a schedule we should return it. Since Alanis' schedule is submitted to the algorithm as a bipartite graph, we will return the schedule as a bipartite graph. Part a of the problem states how the bipartite graph with perfect matching will indicate a daily cooking schedule. To create the bipartite graph with perfect matching, we start with Alanis' given schedule and the path traveled by the BFS. If either node on an edge is the source or sink, we ignore those. If the edge on the path already exists in Alanis' graph, we remove it from the schedule. This would be the case if the edge was marked as having residual capacity and BFS traversed it, indicating that the attached person is no longer assigned to that particular day. if the edge does not already exist in Alanis' graph, we add it. This would indicate that the person is being rescheduled to another day they are available. Doing this will produce a bipartite graph with perfect matching between the people and days, thus representing a schedule for the cooperative apartment. Since there are just n-2 paths with residual capacity, the BFS will produce no more than 2n-2 edges (counting the edge to go to the source and to the sink). Thus we will loop over no more than O(n) edges. And since there are no more than n edges in Alanis' original graph, we can check if an edge exists in Alanis' graph in O(n) time. Since we have to do this O(n) times for each edge in the BFS, this will take O(n^2) time.

Since each consecutive step in the algorithm takes no more than O(n^2) and each step is executed in series, the total runtime of the algorithm is O(n^2).

We note that this is simply one iteration of the Ford-Fulkerson algorithm to see if we can increment the flow between L and R by 1. The correctness of Ford-Fulkerson is proven in the textbook in chapter 26, and if we can use that to increment our flow by 1 then we will have perfect matching and a viable schedule. If we cannot increment the flow there is no perfect matching and there is no perfect schedule. Thus, the algorithm given is correct.

3. You are helping some security analysts monitor a collection of networked computers, tracking the spread of an online virus. There are n computers in the system, labeled C1, ..., Cn, and as

input, you are given a collection of trace data indicating the times at which pairs of computers communicated. Thus the data is a sequence of ordered triples (Ci , Cj , tk). Such a triple indicates that Ci and Cj communicated at time tk. Assume there are m triples total.

Now let us assume that the triples are presented to you sorted by time of communication. For purposes of simplicity, we will assume that each pair of computers communicates at most once during the interval you are observing. The security analysts you are working with would like to be able to answer the following question: If the virus was inserted into computer Ca at time x, could it possibly have infected computer Cb by time y? The mechanics of infection are simple–if an infected computer Ci communicates with an uninfected computer Cj at time tk, (in other words, if one of the triples (Ci , Cj , tk) or (Cj , Ci , tk) appears in the trace data), then 1 the computer Cj becomes infected as well, starting at time tk. Infection can thus spread from one machine to another across a sequence of communications, provided that no step in this sequence involves a move backwards in time. Thus, for example, if Ci is infected by time tk and the trace data contains triples (Ci , Cj , tk) and (Cj , Cq, tr), where tk ≤ tr, then Cq will become infected via Cj . (Note that it is okay for tk to be equal to tr. This would mean that Cj had open connections to both Ci and Cq at the same time, so a virus could move from Ci from Cq.)

For example, suppose n = 4, and the trace data consists of the triples

(C1, C2, 4),(C2, C4, 8),(C3, C4, 8),(C1, C4, 12),

and the virus was inserted into computer C1 at time 2. Then C3 would be infected at time 8 by a sequence of three steps–first C2 becomes infected at time 4, then C4 gets the virus from C2 at time 8, and then C3 gets the virus from C4 at time 8. On the other hand, if the trace data were

(C2, C3, 8),(C1, C4, 12),(C1, C2, 14),

and again the virus was inserted into computer C1 at time 2, then C3 would not become infected during the period of observation. Observe, however, that although C2 becomes infected at time 14, C3 only communicates with C2 before C2 becomes infected. There is no sequence of communications moving forward in time by which the virus could get from C1 to C3 in this second example.

Design an algorithm that answers questions of this type: given a collection of trace data, the algorithm should decide whether a virus introduced at computer Ca at time x could have infected computer Cb by time y. Prove that the algorithm runs in time O(m). Also, prove the correctness of your algorithm.

We will have to create a data structure that we can use to check whether a computer in the network has been infected yet. A list would seem to be a reasonable data structure. However, to check if an item is in a list will take O(n) time if there are n elements in the list. That is a lot of time to be used checking if a computer is infected if we only have O(m) time to process the m triples.

Perhaps it would make more sense to use a hash table. Each key in the hash table could be a computer id (C_1, C_2, … , C_n), and each value of the hash table could indicate whether the computer was infected (true or false). This would save us lots of time checking whether a computer is infected, since each lookup of the hash table would take O(1) time. Python implements its dictionary object as a hash table with O(1) lookup time, so perhaps we can use that in our pseudocode. If a computer does not show up in the hash table, python's .get() method will return null (which evaluates to false for if statements), and we can use this to assume that the computer is not infected.

```
def is_infected(Ca, timex, Cb, timey, trace):        # inputs are Ca, time x, Cb, time y and the trace triples
  infected = {Ca: false, Cb: false}                  # initialize Ca and Cb as not infected in dictionary
  com_index = 0                                      # initialize a counter to track the communication index
  com = trace[com_index]                             # initialize the first communication in trace
  time = com[2]                                      # initialize the time to be the time for the first triple

  while time < timey                                 # iterate over each trace before time y
    comp1 = com[0]                                   # comp1 is the first computer in the triple
    comp2 = com[1]                                   # comp2 is the second computer in the triple
    time = com[2]                                    # track the time of the trace
    if time >= timex:                                # check if we are past the time Ca got infected
      infected[Ca: true]                             # indicate that Ca has been infected
    if infected.get(comp1):                          # check whether comp1 is infected
      infected[comp2] = true                         # if comp1 was infected, comp2 is also infected
    if infected.get(comp2):                          # check whether comp2 is infected
      infected[comp1] = true                         # if comp2 was infected, comp1 is also infected
    com_index += 1                                   # iterate communication index
    com = trace[com_index]                           # move on to the next communication in trace
    time = com[2]                                    # iterate time to check against while() loop condition
  # end while() loop

  return infected.get(Cb)                            # return whether Cb was infected
```

This algorithm is relatively simple. First it initializes some useful things for our while() loop, which happens in O(1) time. Next the algorithm iterates over each of the m triples in the trace that occur before time y. Within each iteration of the while() loop, it checks whether the time variable has passed time x, the time which Ca was infected. If so, it marks Ca as infected in the infected dictionary. Next it checks whether either comp1 or comp2 in the triple were infected (this occurs in O(1) time since it is a hash table lookup, as described previously), and if so marks the other as infected. Finally it increments some variables used in the while() loop. Since all of these operations occur in O(1) time, and there are a finite series of instructions in each iteration of the while() loop, each iteration of the while loop runs in O(1) time. And since there are O(m) iterations of the while loop before the time passes time y, the while while() loop runs in O(m) time. Finally we check the hash table to see if Cb was infected, and return the result (O(1) time). O(1) + O(m) + O(1) time is a total of O(m) time, thus our algorithm runs in O(m) time.

At each communication in the trace we check whether one of the computers in the communication were infected, and if so mark its counterpart as also infected. Thus if one computer was infected before the communication, both computers are infected after the communication. If neither computer was infected before the communication, then neither will be after the communication. This follows the infection by time logic laid out in the problem statement. Thus, if Cb communicates with an infected computer before time y, it will be marked as infected when we get to the return statement. Likewise, if Cb did not communicate with an infected computer before time y, it will not be marked as infected when we get to the return statement. Thus, our algorithm is correct.

4. We define the Escape Problem as follows. We are given a directed graph G = (V, E) (picture a network of roads.) A certain collection of vertices X ⊂ V are designated as populated vertices, and a certain other collection S ⊂ V are designated as safe vertices. (Assume that X and S are disjoint.) In case of an emergency, we want evacuation routes from the populated vertices to the safe vertices. A set of evacuation routes is defined as a set of paths in G such that (i) each vertex in X is the tail of one path, (ii) the last vertex on each path lies in S, and (iii) the paths do not share any edges. Such a set of paths gives a way for the occupants of the populated vertices to "escape" to S without overly congesting any edge in G.

(a) [20 points] Given G,X, and S, show how to decide in polynomial time whether a set of evacuation routes exists.

For this problem we can utilize Ford-Fulkerson to determine if there exists a set of evacuation routes. If the flow determined by Ford-Fulkerson is equal to the number of populated vertexes, then the set of evacuation routes exist. If the max flow is less than that value, the evacuation routes do not exist.

First, we note that Ford-Fulkerson assumes one source and one sink, while in this example we have multiple sources and multiple sinks. One way to get around this is to add a super-source and connect that to each of the source vertexes with an edge of capacity 1. Then we can add a super-sink node and connect that to each of the sink vertexes with infinite capacity. Thus the flow will travel from our super-source node at most once for each source to any of our sink nodes. Note that each other edge in G is assumed to have capacity 1 to enforce that we do not use the same path twice.

Now that we have a graph set up, we can just run Ford-Fulkerson from the super-source to the super-sink and determine the max flow. If the max flow is equal to the number of nodes in V, then there is a set of evacuation routes. If not, then there are not evacuation routes for each of the vertexes in X. This is correct because Ford-Fulkerson gives the max flow, and if there is a flow equal to the number of evacuation areas, then there are enough evacuation routes, and if not, there are not. Now that we have a process we can write out some pseudo code:

```
def are_paths(G, X, S)                      # inputs include graph G, populated X, safe S
  super_source = vertex()                   # create super_source node
  super_sink = vertex()                     # create super_sink node
  for node in X:
    G.add_edge(super_source, node, 1)       # add edge of capacity 1 from source to super_source
  for node in S:
    G.add_edge(super_sink, node, inf)       # add edge of capacity inf from sink to super_sink
```

```
 paths = FordFulkerson(G, super_source, super_sink       # calculate the number of available paths
 if paths == len(X):                                     # check if there are enough paths
  return true                                            # there are enough paths, return true
 else:
   return false                                          # there are not enough paths, return false
```

Adding the super_source and super_sink nodes and their edges will take no more than O(n) time, where n is the number of vertexes in the graph, as adding the edge to the graph will take O(1) time and we will have to add no more than n edges. Ford-Fulkerson has been shown to run in polynomial time in the textbook, thus this algorithm runs in polynomial time.

(b) [20 points] Suppose we have exactly the same problem as in (a), but we want to enforce an even stronger version of the "no congestion" condition (iii). Thus, we change (iii) to say, "the paths do not share any vertices." With this new condition, show how to decide in polynomial time whether such a set of evacuation routes exists. Also provide an example with the same G, X, and S in which the answer is "yes" to the question in (a) but "no" to the question in (b).

To ensure the paths do not share any vertexes, this is a more strict implementation of the same problem presented in a. The process will be mostly the same in terms of using Ford-Fulkerson to determine the number of escape routes, however we will have to make a slight change. Each time a vertex is visited when creating an augmenting path using Ford-Fulkerson, we will update the flow attribute for each of the edges connected to each of the vertexes visited. This is a simple way using Ford-Fulkerson to ensure that we do not travel to the same vertex again, and will not block any paths in the graph that do not go through the vertex in question, thus will not incorrectly eliminate any correct answers.

The algorithm for Ford-Fulkerson from the book is shown below:

```
for each edge (u,v)  ∈ G.E
   (u,v).f = 0
while there exists a path p from s to t in the residual network $G_f$
   $c_f(p)$ = min{$c_f(u,v)$ : (u,v) is in p}
   for each edge(u,v)  in p
      if(u,v)  ∈ G.E
         (u,v).f = (u,v).f + $c_f(p)$
      else (v,u).f = (v,u).f - cf(p)
```

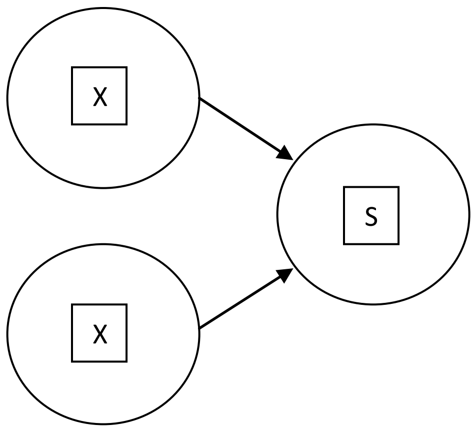
We can make a slight change to update the flow attribute for each edge connected to each visited vertex. See the updated version below:

```
for each edge (u,v)  ∈ G.E
   (u,v).f = 0
while there exists a path p from s to t in the residual network $G_f$
   $c_f(p)$ = min{$c_f(u,v)$ : (u,v) is in p}
   for each edge(u,v)  in p
for each edge(v,d) in v.get_edges()
    if(v,d) $\in$ G.E
        (v,d).f = (v,d).f + $c_f$(p)
    else (v,d).f = (v,d).f - cf(p)
```

Thus, for each vertex visited, each edge in the connected vertex will be marked as used. This is the behavior that we wanted to avoid reusing any vertexes as stated before, so this is the correct method to accomplish this. We can use the same algorithm as given in part a, except we can use the modified version of Ford-Fulkerson shown above.

The original version of Ford-Fulkerson runs in O(E*|f|) time, where |f| is the number of times the while loop will execute. In the updated version, for each edge in the while loop we are looking at all connected edges, of which there can be at most V. Thus, the running time of the altered Ford-Fulkerson is no more than O(V*E*|f|). Fortunately, that is still a polynomial runtime, and thus meets the problem statement request of polynomial runtime.

One example of a graph G which would meet the requirements of part a but not part b is the following:



This would pass part a because the upper populated node could use the upper path to sink node S, while the lower populated node could use the lower path to sink node S. However, In part b, it is determined that the two paths cannot reuse vertexes, and since S is a vertex used in both paths, this would return True to the question in (a) but False to the question in (b).