This is similar to the solved exercise, although the graph is a bit larger. Node a must come first, and node f must come last. Among the remaining four nodes, b must precede c, and d must precede e, but otherwise we can place them however we want.

To figure out how many orderings of these four middle nodes are possible, we note that either b or d must come first among them.

- If b comes first, then either c or d comes next.
    - If c comes next, the ordering must be b, c, d, e.
    - If d comes next, then the ordering can be either b, d, c, e or b, d, e, c.
- If d comes first, we have three orderings obtained simply by reversing the roles of b, c and d, e in the above reasoning.

Thus the total number of orders for these four nodes is 6, and this is the total number of topological orderings.

[1] ex580.660.846

We assume the graph G is connected; otherwise we work with the connected components separately (after computing them in $O(m+n)$ time).

We run BFS starting from an arbitrary node s, obtaining a BFS tree T. Now, if every edge of G appears in the BFS tree, then $G = T$, so G is a tree and contains no cycles. Otherwise, there is some edge $e = (v, w)$ that belongs to G but not to T. Consider the least common ancestor u of v and w in T; we obtain a cycle from the edge e, together with the u-v and u-w paths in T.

[1] ex858.281.666

We run the same algorithm as in the text, with the following small modification. If in every iteration we find a node with no incoming edges, then we will succeed in producing a topological ordering. If in some iteration, it transpires that every node has at least one incoming edge, then we saw in the text a proof that G must contain a cycle. We can turn this proof into an algorithm with running time $O(n)$ to actually find a cycle: we repeatedly follow an edge into the node we're currently at (choosing the first one on the adjacency list of incoming edges, so that this can run in constant time per node). Since every node has an incoming edge, we can do this repeatedly until we re-visit a node v for the first time. The set of nodes encountered between these two successive visits is a cycle C (traversed in the reverse direction).

[1] ex947.5.72

Given a set of n specimens and m judgments, we need to determine if the set of judgments are consistent. To be able to label each specimen either A or B, we construct an undirected graph $G = (V, E)$ as follows: Each specimen is a vertex. There is an edge between v_i and v_j if there is a judgment involving the corresponding specimens.

Once the graph is constructed, arbitrarily designate some vertex as the starting node s. Note that the graph G need not be connected in which case we will need starting nodes for each component. For each component G_i (with starting node s_i) of G label the specimen associated with s_i A. Now perform Breadth-First Search on G_i starting at s_i. For each node v_k that is visited from v_j, consider the judgment made on the specimens corresponding to (v_j, v_k). If the judgment was "same," label v_k the same as v_j and if the judgment was "different," label v_k the opposite of v_j. Note that there may be some specimens that are not associated with any judgments. These specimens maybe labeled arbitrarily, but we shall label them A. Once the labeling is complete we may go through the list of judgments to check for consistency. More precisely (Refer to pg. 39 of the text)

```
For each component C of G
   designate a starting node s and label it A
   Mark s as ``Visited.''
   Initialize R=s.
   Define layer L(0)=s.
   For i=0,1,2,...
      For each node u in L(i)
         Consider each edge (u,v) incident to v
         If v is not marked ``Visited'' (then v is also not labeled)
            Mark v ``Visited''
            If the judgment (u,v) was ``same'' then
               label v the same as u
            else (the judgment was ``different'')
               label v the opposite of u
            Endif
            Add v to the set R and to layer L(i+1)
         Endif
      Endfor
   Endfor
Endfor
For each edge (u,v) (for each judgment (u,v))
   If the judgment was ``same''
      If u and v have different labels
         there is an inconsistency
      Endif
   Else (the judgment was ``different'')
```

[1]ex926.803.309

```
        If u and v have the same labels
           there is an inconsistency
        Endif
     Endif
  Endfor
```

First note that the running time of this algorithm is $O(m+n)$: Constructing G takes $O(m+n)$ since it has n vertices and m edges. Performing BFS on G takes $O(m+n)$ and going through the list of judgments to check consistency takes $O(m)$. Thus the running time is $O(m+n)$.

It is easily shown that if the labeling produced by the BFS is inconsistent, then the set of judgments is inconsistent. Note that this BFS labeling uses a subset of the judgments (the edges of the resulting BFS tree). Further the BFS labeling is the only possible labeling with the exception of inverting the labeling in each component of G, i.e. switching A and B. Thus if an inconsistency is found in this labeling then surely the entire set of m judgments cannot be consistent. On the other hand if the labeling is consistent with respect to the m judgments, we are done.

We prove this by induction on the number of nodes in T. Let $n_0(T)$ denote the number of leaves of a binary tree T, and let $n_2(T)$ denote the number of ndoes with two children.

The basis of the induction is a tree with a single node. This node is the only leaf, and there are no nodes with two children.

Now, let T be an arbitrary binary tree on more than one node, and let v be a leaf. Since T has more than one node, v is not the root, so it has a parent u. Let T' be the tree obtained by deleting v.

If u had no other child in T, then it becomes a leaf in T', so we have $n_0(T') = n_0(T)$ and $n_2(T') = n_2(T')$. Applying the induction hypothesis to T' completes the induction step in this case. On the other hand, if u had another child in T, then it does not become a leaf after the deletion; but it used to have two children and now it doesn't. Thus we have $n_0(T') = n_0(T) - 1$ and $n_2(T') = n_2(T') - 1$. Again, applying the induction hypothesis to T' completes the induction step in this case.

[1]ex113.833.893

Suppose that G has an edge $e = \{a, b\}$ that does not belong to T. Since T is a depth-first search tree, one of the two ends must be an ancestor of the other — say a is an ancestor of b. Since T is a breadth-first search tree, the distance of the two nodes from u in T can differ by at most one.

But if a is an ancestor of b, and the distance from u to b in T is at most one greater than the distance from u to a, then a must in fact be the direct parent of b in T. From this it follows that $\{a, b\}$ is an edge of T, contradicting our initial assumption that $\{a, b\}$ did not belong to T.

[1] ex374.652.223

The claim is true; here is a proof. Let G be a graph with the given properties, and suppose by way of contradiction that it is not connected. Let S be the nodes in its smallest connected component. Since there are at least two connected components, we have $|S| \leq n/2$. Now, consider any node $u \in S$. Its neighbors must all lie in S, so its degree can be at most $|S| - 1 \leq n/2 - 1 < n/2$. This contradicts our assumption that every node has degree at least $n/2$.

[1]ex722.926.396

The claim is false; we show that for every natural number c, there exists a graph G so that $diam(G)/apd(G) > c$. First, we fix a number k (the relation to c will be determined later), and consider the following graph. We take a path on $k-1$ nodes $u_1, u_2, \ldots, u_{k-1}$ in this order. We then attach $n-k+1$ additional nodes $v_1, v_2, \ldots, v_{n-k+1}$, each by a single edge, to u_1; the number n will also be chosen below.

The diameter of G is equal to $dist(v_1, u_{k-1}) = k$. It is not difficult to work out the exact value of $apd(G)$; but we can get a simple upper bound as follows. There are at most kn 2-element sets with at least one element from $\{u_1, u_2, \ldots, u_{k-1}\}$. Each of these pairs is at most distance $\le k$. The remaining pairs are all distance at most 2. Thus

$$apd(G) \le \frac{2\binom{n}{2} + k^2 n}{\binom{n}{2}} \le 2 + \frac{2k^2}{n-1}.$$

Now, if we choose $n-1 > 2k^2$, then we have $apd(G) < 3$. Finally, choosing $k > 3c$, we have $diam(G)/apd(G) > 3c/3 = c$.

[1]ex85.422.171

We run BFS starting from node s. Let d be the layer in which node t is encountered; by assumption, we have $d > n/2$. We claim first that one of the layers $L_1, L_2, \ldots, L_{d-1}$ consists of a single node. Indeed, if each of these layers had size at least two, then this would account for at least $2(n/2) = n$ nodes; but G has only n nodes, and neither s nor t appears in these layers.

Thus, there is some layer L_i consisting of just the node v. We claim next that deleting v destroys all s-t paths. To see this, consider the set of nodes $X = \{s\} \cup L_1 \cup L_2 \cup \cdots \cup L_{i-1}$. Node s belongs to X but node t does not; and any edge out of X must lie in layer L_i, by the properties of BFS. Since any path from s to t must leave X at some point, it must contain a node in L_i; but v is the only node in L_i.

[1] ex758.356.752

We will solve the more general problem of computing the number of shortest paths from v to every other node.

We perform BFS from v, obtaining a set of layers $L_0, L_1, L_2, \ldots$, where $L_0 = \{v\}$. By the definition of BFS, a path from v to a node x is a shortest v-x path if and only if the layer numbers of the nodes on the path increase by exactly one in each step.

We use this observation to compute the number of shortest paths from v to each other node x. Let $S(x)$ denote this number for a node x. For each node x in L_1, we have $S(x) = 1$, since the only shortest-path consists of the single edge from v to x. Now consider a node y in layer L_j, for $j > 1$. The shortest v-y paths all have the following form; they are a shortest path to some node x in layer L_{j-1}, and then they take one more step to get to y. Thus, $S(y)$ is the sum of $S(x)$ over all nodes x in layer L_{j-1} with an edge to y.

After performing BFS, we can thus compute all these values in order of the layers; the time spent to compute a given $S(y)$ is at most the degree of y (since at most this many terms figure in the sum from the previous paragraph). Since we have seen that the sum of the degrees in a graph is $O(m)$, this gives an overall running time of $O(m + n)$.

[1] ex427.691.966

The key here is that, while this kind of connectivity includes a notion of time, it can be converted into a graph connectivity problem of a more standard sort.

We construct the following directed graph G. We scan through the ordered triples in the trace data, maintaining an array pointing to linked lists associated with each computer C_a. (Each list is initalized to null.)

For each triple (C_i, C_j, t_k) we see in our scan, we create nodes (C_i, t_k) and (C_j, t_k), and we create directed edges joining these two nodes in both directions. Also, we append these nodes to the lists for C_i and C_j respectively. If this is not the first triple involving C_i, then we include a directed edge from (C_i, t) to (C_i, t_k), where t is the timestamp in the preceding element (the previously last one) in the list for C_i. We do the analogous thing for C_j. By explicitly maintaining these lists for each node, we are thus able to construct all these new nodes and edges in constant time per triple.

Now, given a collection of triples, we want to decide whether a virus introduced at computer C_a at time x could have infected computer C_b by time y. We walk through the list for C_a until we get to the last node (C_a, x') for which $x' \leq x$. We now run directed BFS from (C_a, x') to determine all nodes that are reachable from it. If a node of the form (C_b, y') with $y' \leq y$ is reachable, then we declare that C_b could have been infected by time y; otherwise we declare it could not have.

Let's argue first about the correctness of the algorithm, then its running time. First, we claim that if there is a path from (C_a, x') to (C_b, y') as in the previous paragraph, then C_b could have been infected by time y. To see this, we simply have the virus move between computers C_i and C_j at time t_k, whenever an edge from (C_i, t_k) to (C_j, t_k) is traversed by the BFS. This is a feasible sequence of virus transmissions that results in the virus first leaving C_a at time x or later (by the definition of x') and arriving at C_b by time y.

Conversely, suppose there were a sequence of virus transmissions that results in the virus first leaving C_a at time x or later and arriving at C_b by time y. Then we can build a path in our graph as follows. We start at node (C_a, x') and follow edges to (C_a, x''), for the x'' when the virus first leaves C_a. (Note that there are such edges since $x' \leq x \leq x''$; or else $x' = x''$.) In general, for each time that the virus moves from C_i to C_j at time t_k, we add the edge from (C_i, t_k) to (C_j, t_k) to the path; if it next moves out of C_j at time $t \geq t_k$, we add the the sequence of edges from (C_j, t_k) to (C_j, t). When the virus first arrives at node C_b, we will have just added a node (C_b, y'') to the path; since $y'' \leq y$ and y' is the largest y involving C_b in the trace data with this property, there is a sequence of edges from (C_b, y'') to (C_b, y'), completing the path.

Finally, we consider the running time. Each triple in the trace data causes us to add a constant number of nodes and edges to the graph, so the graph has $O(m)$ nodes and edges, and since we build it in constant time per node and edge, this takes time $O(m)$. Running BFS takes time linear in the size of the graph, so this too takes time $O(m)$.

[1] ex207.316.912

We construct a directed graph G; the test for consistency will turn out to be equivalent to testing whether G is acyclic.

For each person P_i, we define nodes b_i and d_i, representing their (unknown) birth and death dates respectively. Edges will correspond to one event preceding another. So to start, we include edges (b_i, d_i) for each i. When we are told that

- for some i and j, person P_i died before person P_j was born,

we include an edge (d_i, b_j). When we are told that

- for some i and j, the lifespans of P_i and P_j overlapped at least partially,

we include edges (b_i, d_j) and (b_j, d_i). This completes the construction of G.

Now suppose G has a cycle. Then each of the events corresponding to nodes in this cycle must precede the next; but this means that there is no event among these that can be put first in time, consistent with the given information. Hence the information is not internally consistent.

On the other hand, suppose G has no cycle. Then it has a topological ordering. If we use this as the order of the birth and death dates of all people, then we have an ordering that is consistent with all the given pieces of information.

[1] ex92.401.128