1. Exercise 3 Page 107

We can extend the topological ordering algorithm and check if a node is already visited every time we visit the node. If a node is already visited and we are visiting again, this means there's a cycle and the set of nodes visited between the node is the cycle.

The algorithm is the following:

```
To compute a topological ordering of G:
Find a node v with no incoming edges
If v is already visited:
    Return nodes visited from end of ordering list to v (i.e. cycle C)
Place v in the ordering list
Delete v from G
Recursively compute a topological ordering of G - {v} and append this order after v
```

2. Exercise 4 Page 107

Construct a graph *G* with *n* nodes and *m* edges, where a node represents a specimen and an edge represents a judgement. Then we will use BFS to traverse the graph. We want to find an instance of one judgement contradicting the other judgement. For example, we have the judgements "*v* is same as *u*" and "*u* is same as *w*". Then the judgement "*v* is different from *w*" contradicts the previous judgements. The idea is to follow each judgement/edge and label specimen/node accordingly. If there is inconsistency, then somewhere during the edge traversal, we will encounter, for example, a case where we have a judgement "*v* is same as *u*" but *v* is already labeled as A and *u* is already labeled as B. If the judgements are consistent, then we will successfully traverse all the judgement without encountering any inconsistency.

The algorithm is the following:

```
Construct a graph G with n nodes and m edges, where a node represents a specimen and an
edge represents a judgement.
While there exists a node that hasn't been visited yet:
  Arbitrary pick such a node v
  Arbitrary pick a label for v
  For each edge e of v (that connects to a node u): … (*BFS)
    If u hasn't been visited yet:
      If e represents "same" judgement:
        Set the label of u be same as the label of v (which we arbitrarily picked earlier)
      Else:
        Set the label of u be different from the label of v
      Recursively traverse the node u (Go back to (*BFS) with v = u)
    Else:
      If e represents "same" judgement:
        If the label of u is different from the label of v:
          Return "Inconsistent"
      Else:
        Set the label of u be different from the label of v
        If the label of u is same as the label of v:
          Return "Inconsistent"
Return "Consistent"
```

3. Exercise 9 Page 110

Let *d* be the distance between *s* and *t*. Then, if we perform BFS on *G* from node *s*, node *t* will be found in *d*-th layer. If any layer between 1st layer and (*d* - 1)-th layer has only one node *v*, then it is a node such that deleting it from *G* will destroy all *s*-*t* path because every *s*-*t* path will contain exactly one node from each layer.

Thus, we want to show there exists a layer between 1st layer and (*d* - 1)-th layer that has only one node. Suppose on the contrary, each layer between 1st layer and (*d* - 1)-th layer has at least two nodes. Then there are at least 2(*d* - 1) + 2 = 2*d* nodes in *G* (There are 2(d - 1) between 1st layer and (*d* - 1)th layer, node *s*, and node *t*). Since *d* > *n*/2, 2*d* > *n*. This contradicts the fact that there are *n* nodes in *G*. Therefore, there exists a layer between 1st layer and (*d* - 1)-th layer that has only one node.

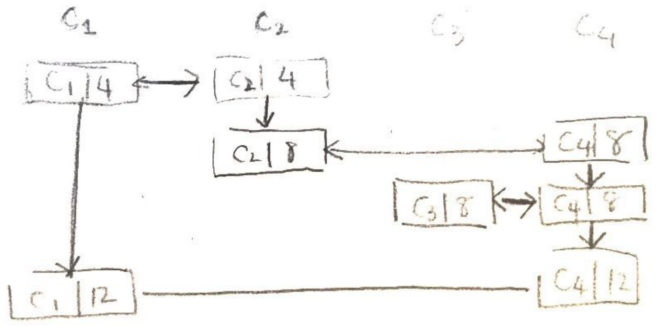
Q.E.D.

The algorithm is the following:

```
Let Q be a queue
Let L be a dictionary that maps from a vertex to a level
Add s to Q
L[s]=0
Mark s as visited
While Q is not empty:
    Dequeue a vertex v from Q
    For each edge e of v:
        If the destination vertex v' is not visited:
            Add v' to Q
            L[v' ] = L[v] + 1
            If v' is t :
                d=L[v' ]
            Mark v' as visited
Check L to find out which node is the only node in a layer, and return such a node.
```

4. Exercise 11 Page 111

We will construct a directed graph *G* with the following way. First, we iterate over the triples such that for each triple (C_i, C_j, t_k), we create nodes (C_i, t_k) and (C_j, t_k) and connect with undirected edge (directed edge from both sides). Then we add directed edges from previous node that involves C_i to (C_i, t_k) and from C_j to (C_j, t_k) respectively. For example, suppose *n* = 4, the trace data consists of the triples $(C_1, C_2, 4)$, $(C_2, C_4, 8)$, $(C_3, C_4, 8)$, $(C_1, C_4, 12)$. Then the graph *G* would look as follows:



Then we perform BFS from node (C_a, t_k) where t_k is when C_a communicated with someone for the first time after time *x*. If we can reach to node (C_b, t_k) where $t_k \leq y$, then a virus introduced at computer C_a at time *x* could have infected computer C_b by time *y*. Otherwise it could not have.

The algorithm is the following:

```
For each triple $(C_i, C_j, t_k)$:
    Create a node $(C_i, t_k)$
    Create a node $(C_j, t_k)$
    Add an edge from $(C_i, t_k)$ to $(C_j, t_k)$
    Add an edge from $(C_j, t_k)$ to $(C_i, t_k)$
    Add $(C_i, t_k)$ to list of nodes of $C_i$
    Add an edge from last node in $C_i$, if there exists one, to $(C_i, t_k)$
    Add $(C_j, t_k)$ to list of nodes of $C_j$
    Add an edge from last node in $C_j$, if there exists one, to $(C_j, t_k)$
Let Q be a queue
```

```
Find ($C_a$, $t_k$) where $t_k$ is when $C_a$ communicated with someone for the first time after time x
from the list of nodes of $C_a$
Mark ($C_a$, $t_k$) as visited
While Q is not empty:
   Dequeue a vertex ($C$, $t$) from Q
   For each edge e of ($C$, $t$):
      If the destination vertex ($C'$, $t'$) is not visited:
         Add ($C'$, $t'$) to Q
         If $C'$ is $C_b$ and $t' \leq y$:
            Return "$C_b$ could have been infected by time y"
         Mark ($C'$, $t'$) as visited
Return "$C_b$ could NOT have been infected by time y"
```

5. Exercise 12 Page 112

We will construct a directed graph *G* with the following way. For each person P_i, we create two nodes b_i and d_i. b_i denotes the birth date. d_i denotes the date of death. Then we add a directed edge from b_i to d_i. For each fact, we will add directed edges as the following. If person P_i died before person P_j was born, then add a directed edge from d_i to b_j. If the life spans of P_i and P_j overlapped at least partially, then the order of the nodes are either (b_i, b_j, d_i, d_j) or (b_j, b_i, d_j, d_i). Thus, we add a directed edge from b_i to d_j and another directed edge from b_j to d_i.

Now the graph *G* is constructed. If the graph has no cycle, then it has a topological ordering, and thus, the facts collected were consistent. Otherwise, if the graph has a cycle, each event (born or death) in the cycle is waiting for another event in the loop to happen. Thus, no event can happen first, so the facts collected were inconsistent.

The algorithm is the following:

```
Let L be the topological ordered list
For person $P_i$:
    Create nodes $b_i$ and $d_i$
For each fact:
    If person $P_i$ died before person $P_j$ was born:
        Add an edge from $d_i$ to $b_j$
    If the life spans of $P_i$ and $P_j$ overlapped at least partially:
        Add an edge from $b_i$ to $d_j$
        Add an edge from $b_j$ to $d_i$
Find a node $b$ with no incoming edges
If $b$ is already visited:
    Return "Inconsistency"
Place $b$ in L
Delete $b$ from G
Recursively compute a topological ordering of G - $\{b\}$ and append this order after $b$
Propose dates in the order of L
Return proposed dates
```

6. (a)

Assume the array is zero indexed. Then we want to find the smallest index i such that it's element is $i+2$. Then $i+1$ is the missing number. We can find this index i with binary search. First we check the element at index $2/n$. If its element is $2/n+1$, then we recursively search the right hand side of the array. Otherwise, if its element is $2/n+2$, then we recursively search the left hand side of the array. Since we are continuously reducing the number of elements to search by half, this algorithm runs in $O(logn)$.

The algorithm is the following:

```
Find index $i$ in the middle of given array
If the size of the array is 1: (Base Case)
   Return $i+1$, which is the missing element
If the element at index $i$ is $i+1$:
   Recursively search the right hand side of the array until the size of array is 1
Else:
   Recursively search the left hand side of the array until the size of array is 1
```

6 (b)

Assume the array is zero indexed. We can create an auxiliary array of size $n+1$. Initialize every element in this array to be 0. Then, we iterate over the original array and use its element - 1 as an index of the auxiliary array, and set its value to be 1. Now, every element in the auxiliary array is 1 except for the missing element. Finally, we iterate over the auxiliary array and find the index i whose element is 0. $i+1$ is the missing number. Since we are iterating over arrays of size n and $n+1$, the time complexity of this algorithm is $O(n)$.

The algorithm is the following:

```
Let original array be $A$
Create an auxiliary array $B$ of size $n+1$
Initialize all elements in $B$ to be $0$
For each element $e$ in $A$:
    $B[e-1]=1$
For each element $e$ in $B$:
    If $e$ is $0$:
        Let $i$ be the index of $e$ in $B$
        Return $i+1$ as the missing number
```