3.6 **Answer**: Proof by contradiction; suppose G contains an edge (x, y) that T does not. Since T is a BFS tree, x and y must belong to adjacent layers. Since T is also a DFS tree, one of x or y is the ancestor of the other. We can take x as the ancestor of y upon renaming. Then since x is the ancestor of y and they belong to adjacent layers, x is the parent of y. This would mean that (x, y) is an edge in T which contradicts with our initial assumption, therefore $G = T$.

3.9 **Algorithm**:

1. Run BFS from s.
2. Traverse through the layers of the BFS tree, find and remove the layer with exactly one node v.

Proof of correctness: We first want to show that such v always exists: since the distance between s and t is strictly greater than $\frac{n}{2}$, there are at least $\frac{n}{2}$ layers in our BFS tree. If every layer had 2 or more nodes, we would have strictly more than n nodes in total which contradicts G having n nodes. So there is at least one layer with exactly one node. By construction of BFS, removing this node breaks all paths from s to t. Since the algorithm performs exactly as stated, it will always remove v.
Complexity: $O(m + n)$ since it is the complexity of BFS.

3.11 **Algorithm**:

1. Initialize an empty array for each computer $C_1, \ldots, C_n$.
2. Traverse through the sequence and perform the following for each triple (C_i, C_j, t_k):
   - Create nodes (C_i, t_k) and (C_j, t_k) and edges in both directions.
   - Store (C_i, t_k) and (C_j, t_k) in the corresponding C_i and C_j arrays.
   - If C_i has appeared before (if its array is nonempty), create an edge to the previous C_i node.
   - Repeat the previous step for C_j.
3. Let x' be the first time (C_a, x') appears in the list corresponding to C_a such that $x' \geq x$; similarly, let $y' \leq y$ be the last time (C_b, y') appears in the list corresponding to C_b such that $y' \leq y$.
4. Run BFS from (C_a, x'), check if (C_b, y') is reachable. Return true if so and false if not.

Proof of correctness: By contradiction; suppose the computer virus infects C_b by time y but (C_b, y') is not reachable by BFS. However, for the virus to travel from C_a to C_b there must have been a path between them; i.e. take arbitrary C_i and C_j on the path, (C_i, t_k) and (C_j, t_k) must be connected on the graph. Since BFS traverses through all nodes with time before y, it must have traversed through the edge $((C_i, t_k), (C_j, t_k))$. Then our algorithm will always return true if the virus reaches C_b.
Now suppose the virus does not infect C_b by time y but the algorithm reaches (C_b, y'). Since the virus does not travel to C_b, (C_a, x') and (C_b, y') must be in different components on the graph. Then there is no way for BFS to traverse from (C_a, x') to (C_b, y'). Therefore our algorithm will always return false if the virus does not reach C_b.
Complexity: $O(m + n)$; traversing the sequence of tuples takes $O(m)$ time and BFS takes $O(m + n)$ time.

4.2 (a) **Answer**: True, we can simply run Kruskal's algorithm for both graphs; since the ordering of the costs stay the same, Kruskal's algorithm will return the same tree.

(b) **Answer**: False, by counter example: let $G = (V, E) = (s, v, t, (s, v, 2), (v, t, 2), (s, t, 3))$, then $G' = (V', E') = (s, v, t, (s, v, 4), (v, t, 4), (s, t, 9))$. The shortest path in G is (s, t) with cost 3 whereas the shortest path in G' is $(s, v), (v, t)$ with cost 8.

P1 You have been commissioned to write a program for the next version of electronic voting software. The input will be the number of candidates, d, and an array votes of size v holding the votes in the order they were cast where each vote is an integer from 1 to d. The goal is to determine if there is a candidate with a majority of the votes. You can only use a constant number of extra storage (note that v and d are not constants). Prove the correctness of your algorithm and analyze its complexity.
Algorithm:

1. Set variable $max = 0$ (max. number of votes).
2. For each candidate i, do the following:
   - Traverse through all votes and sum up the total number of votes for the candidate
   - Set max to the number of votes found in the previous step if the number of votes is greater than max.
3. Check if $\frac{max}{v} > 0.5$. Return true if so and false if not.

Proof of correctness: By contradiction. Suppose a candidate c has majority votes but the algorithm returns false, which means that $\frac{max}{v} \leq 0.5$. However, since we are always replacing max with candidates with more votes, max stores the number of votes for c which is > 0.5. Therefore our algorithm will return true when a candidate has majority votes.
Now suppose no candidate has majority votes but the algorithm returns true, which means that $\frac{max}{v} > 0.5$. However, since our algorithm traverses through every candidate, there is at least one candidate with majority votes, which contradicts with our assumption. Then by contradiction our algorithm will return false when there is not a candidate with majority votes.
Complexity: $O(dv)$ since we're traversing d candidates for the outer loop and v votes for the inner loop.

P2 (a) Can you design an algorithm that finds the longest path in a directed graph (you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.
Algorithm:

1. Initialize an empty array that will be used to stores path.
2. Traverse through every node in the graph, for each node n, perform the following:
   - Run DFS from n, when DFS reaches a dead end, record the path in the array.
3. Return the longest path from the array.

Proof of correctness: Suppose the longest path starts from node u and ends at node v. Since we are traversing through every node, at some point we must have encountered u as the starting node. Then since DFS always finds all simple paths, the longest path from u to v will always be

found, which then will be stored into the array and eventually returned.

Complexity: $O(n^2+mn)$; traversing through every node is $O(n)$ and DFS is $O(n+m)$, therefore our algorithm is $O(n(n+m)) = O(n^2+mn)$.

(b) Can you design an algorithm that finds the longest path in a directed acyclic graph (you can use an edge at most once)? If yes, describe the algorithm and analyze its time complexity.

Algorithm:

1. Initialize an array *lengths* that stores path lengths.
2. Create a toplogical order of the nodes.
3. Traverse through the nodes in toplogical order, do the following for each node i:
   - For all adjacent nodes j, store $lengths[j] = \max(lengths[j], lengths[i]+1)$.
4. Find the node of *lengths* with the highest value, say max with value $length[max]$, then find the next node in the path by finding the node with value $length[max]-1$, until a node with value 0 is reached. Return the list of nodes as the longest path.

Proof of correctness: Suppose that the algorithm returns a path that is not the longest path. Then there must exists another path with a longer length; let i and f be the initial and final nodes of this path, respectively. Since we are traversing through every node, i must have been traversed to at some point. Then from there the intermediate nodes would record the length between i and that node. Eventually $lengths[n]$ would record the length of that path, which since we assumed to be the longest path, would be the max value of the *lengths* array, which would be returned by the algorithm, contradicting our assumption. Therefore our algorithm always returns the longest path.

Complexity: $O(m+n)$; toplogical sorting takes $O(m+n)$ time, then the traversal takes another $O(m+n)$, so overall time complexity is $O(m+n)$.