Algorithms Supo Work

9 i) Yes since at each stage of Prim's algorithm we are looking for the edge with the lowest weight to connect a vertice not in the tree to the tree. All of these edges will have had c added so their ordering is preserved i.e. the lowest weight will still be the lowest.

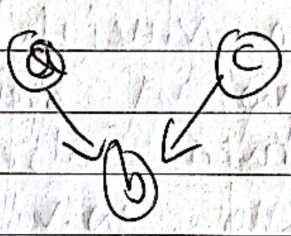
ii) It would still work to find a minimum spanning tree. This is because at each stage we only care about the differences in edge weight to connect a new vertice to the tree, not the distance from the start vertex. Therefore Prim's will not suffer from negative edge weights in the same way as Dijkstra's.

10. Base case:
Prim's chooses an arbitrary vertex to start from. Since all vertices must be in the MST for the graph, the chosen vertex is in the MST.

Inductive step:
Assume that Prim's has constructed a forest F of k-1 vertices that is part of the MST for the given graph.
On the k^{th} iteration we will have a frontier of all vertices that aren't in F but could be connected by a single edge. These edges form a cut C such that no edges in F cross C. Prim's will then add the minimum weight edge from this cut. Then by the theorem given the new forest will still be part of the MST.

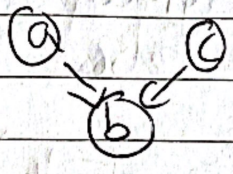
11. Yes we always get another DAG. Assume we have a DAG g and it's reversed version r. If r has a cycle then by reversing all the edges in the cycle we must get another cycle. However the reversed edges would be in g and we know g has no cycles. Therefore r has no cycles. Also it's obvious that r is still directed.

12. i) Take the graph



If we run dfs-rcurse starting from a, which has no incoming edges, then we'd never reach e so we couldn't get a total order.

ii) Take the same graph



Start from a we would append a, b, c in that order so our total order would be [a, b, c]. However $c \to b$ so c should come before b in our order.

13.
```
def isDag(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        cycle = False
        for v in g.vertices:
            if not v.visited:
                stack = []
                cycle = visit(v, stack)
                if cycle:
                    return cycle
    return cycle

def visit(v, stack):
    v.visited = True
    s = stack.copy()
    s.append(v)
    for w in v.neighbours:
        if w in s:
            return True
        if not w.visited:
            return visit(w, s)
    return False
```