(1) DPV Problem 4.11.

Give an algorithm that takes as input a directed graph with positive edge lengths, and returns the length of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most $O(\|V\|^3)$.

First, for each node in the graph, use Dijkstra's algorithm to find the shortest path to form a cycle. Implement a list and store the lengths returned from each node following Dijkstra's algorithm, storing infinity if there is no cycle. After exiting this loop, implement a method to find the smallest value in the list and store this value. The final step would be to implement a conditional to check if this value is equal to infinity, in which case it would be acyclic, or just return the value directly. The algorithm would be $O(\|V\|^3)$ because Dijkstra's is $O(\|V\|^2)$ while it is implemented inside a loop, giving $O(\|V\|^3)$. Additionally, the implementation to find the smallest value is less than $O(\|V\|^3)$.

(2) DPV Problem 4.13.

You are given a set of cities, along with the pattern of highways between them, in the form of an undirected graph $G = (V, E)$. Each stretch of highway $e \in E$ connects two of the cities, and you know its length in miles, l_e. You want to get from city s to city t. Theres one problem: your car can only hold enough gas to cover L miles. There are gas stations in each city, but not between cities. Therefore, you can only take a route if every one of its edges has length $l_e \leq L$.

(a) Given the limitation on your cars fuel tank capacity, show how to determine in linear time whether there is a feasible route from s to t.

We can determine this through the use of an algorithm that consists of two steps. The first is to go through every edge in the graph and remove edges where $l_e > L$. Following this, the second step would just be a BFS implementation to find if there is a path from s to t.

(b) You are now planning to buy a new car, and you want to know the minimum fuel tank capacity that is needed to travel from s to t. Give an $O((\|V\| + \|E\|)log\|V\|)$ algorithm to determine this.

This algorithm can be implemented by a modified version Dijkstra's algorithm. This change is implemented because, rather than finding the shortest total path, our goal is to find the path which has the shortest maximum edge length for a single edge. Inside the innermost loop of Dijkstra's, if the two lines are modified from:
if $dist(v) > dist(u) + l(u, v)$:
$dist(v) = dist(u) + l(u, v)$
to be:
if $dist(v) > max(dist(u), l(u, v))$:
$dist(v) = max(dist(u), l(u, v))$
Changing the method by which distances are updated also changes the order notation to be $O((\|V\|+\|E\|)log\|V\|)$. Following this, it just requires to return $dist[t]$.

(3) DPV Problem 4.19. Your algorithm should run in time at most $O(\|V\|^2)$.

Generalized shortest-paths problem. In Internet routing, there are delays on lines but also, more significantly, delays at routers. This motivates a generalized shortest-paths problem.
Suppose that in addition to having edge lengths $\{l_e : e \in E\}$, a graph also has vertex costs $c_v : v \in V$. Now define the cost of a path to be the sum of its edge lengths, plus the costs of all vertices on the path (including the endpoints). Give an efficient algorithm for the following problem.

Input: A directed graph $G = (V, E)$; positive edge lengths l_e and positive vertex costs c_v; a starting vertex $s \in V$.

Output: An array $cost[\cdot]$ such that for every vertex u, $cost[u]$ is the least cost of any path from s to u (i.e., the cost of the cheapest path), under the definition above.
Notice that $cost[s] = c_s$.

We can implement this algorithm in two steps. In the first, loop through every edge length $\{l_e : e \in E\}$. For each l_e value, it is directed from one component to another. If we assume it is moving from vertex u to vertex v, for every edge, we will add c_v to the length.
The second step would be to implement Dijkstra's using the modified edge lengths.

(5) Give a linear-time algorithm that takes as input a tree and determines whether it has a perfect matching: a set of edges that touches each node exactly once.

The first step is to look if there are an odd number of nodes and if there are, return false.
Loop through each node in the tree and find the number of edges connected to each node, saving them as degrees for each node.
Following that, loop through each node in the tree to find any leaf nodes by checking for nodes where the degree is 1. For every time a leaf is found, save the leaf to a queue object.
Loop through the queue, using pop to pull out each leaf. A leaf will have a single edge. This edge will connect to a second node that will be referred as n_2. For each node connected to n_2 other than the leaf node, decrease the degree by one and remove the edges connected to n_2 other than that to the leaf. Additionally, check the new node degrees and if the degree has changed to 1, push the node back into the queue, at the same time, if the new node degrees are 0, return false.
If the algorithm is able to go through every leaf in the queue, it will find a a perfect matching and return true.

(4) DPV Problem 5.5.

Consider an undirected graph $G = (V, E)$ with nonnegative edge weights $w_e \geq 0$. Suppose that you have computed a minimum spanning tree of G, and that you have also computed shortest paths to all nodes from a particular node $s \in V$.
Now suppose each edge weight is increased by 1: the new weights are $w_e = w_e + 1$.

(a) Does the minimum spanning tree change? Give an example where it changes or prove it cannot change.

No, it will not change. To prove this, we will use Kruskal's algorithm to find the minimum spanning tree. Since all the edges will remain distinct even after increasing by one, the graph structure will remain unchanged and there will still be a single unchanged minimum spanning tree.

(b) Do the shortest paths change? Give an example where they change or prove they cannot change.

Yes, the shortest paths can change. In order to five an example of this, imagine a figure with two paths to a node. One consists of two edges which we will call p_1. The other consists of six edges will be called p_2. Imagine that, before changing the weights, p_1 had a path length of 8, while p_2 had a path length of 6. This would make the previous shortest path p_2. However, after adding weights, the path lengths will change, with p_1 becoming 10 due to having two edges while p_2 becomes 12 due to having six edges. Thus the new shortest path would be p_1.