Problem 1: You are given as input a (directed, weighted) graph $G$ and one of his minimum spanning trees $T$. Show how to update the MST if a new vertex with some adjacent edges are added to $G$. Analyze the correctness and the running time of your algorithm. Better credit will be given to faster algorithms.

Solution: Let $T$ be the MST of $G$. Note that $T$ has exactly $n$ nodes (where $n$ is the number of nodes in $G$) and $n - 1$ edges. When you add a new node $v$ and some edges, you are adding at most $n$ edges. Consider the graph $G'$ which is $T$ together with $v$ and its edges. $G'$ has $n + 1$ nodes and at most $2n - 1$ edges. Run any MST algorithm (e.g. Prim’s algorithm) and get a new MST $T'$ for $G'$. If you use Prim’s algorithm (with a heap implementation for the priority queue) this will cost $O(E \log V)$, but since in this case $E = O(n)$ we have that the update costs $O(n \log n)$.

If the number of edges attached to $v$ is small (say $k$) then we can do this in $O(k \log k + kn)$ which is $O(n)$ for constant $k$. All you have to do is: first sort the the new edges by weight ($O(k \log k)$ cost). Then add each edge smallest weight first and remove the heaviest edge from any cycle you create. To follow the cycle takes $O(n)$ hence the running time.

Problem 2: What happens when you run Johnson’s algorithm on a graph that has all positive weights on its edges? What are the values of the functions $h(v)$ and $\hat{w}(e)$ for any node $v$ and edge $e$?

Solution: Nothing happens! Recall that Johnson’s algorithm adds a node $s$ to the graph and edges $(s, v)$ for all nodes $v \in V$. Those edges will be given weight 0. Then the function $h(v)$ is defined as the shortest distance from $s$ to $v$ in the new graph. But since all other edges are positive, we have that the shortest path is the new edge that we added with weight 0, so $h(v) = 0$. Remember then that if $e = (u, v)$ we have that

$$\hat{w}(e) = w(e) + h(u) - h(v)$$

and therefore $\hat{w}(e) = w(e)$ in this case.

Problem 3: As described in the book and in class, the Floyd-Warshall algorithm requires $O(n^3)$ space since it stores the matrices $d^k_{ij}$ for all $i, j, k \in [1..n]$. Show a way to reduce the space requirement of the algorithm to $O(n^2)$.

Solution: Look carefully at the code for the Floyd-Warshall algorithm: you will see that to compute $d^k_{ij}$ you only need $d^{k-1}_{ij}$ and $d^{k-2}_{ij}$ and therefore it is enough too only keep the last two copies of the matrices $d_{ij}$ (i.e. once you compute $d^2_{ij}$, you can erase $d^0_{ij}$ and so on ... )