Practice problems (don't turn in):

1. **Dijkstra's algorithm**: This is not covered in the lectures because it's the sort of thing many of you have seen before, possibly multiple times. If you haven't seen it, or need a refresher, look at Chapter 4.4 of [DPV]. We will assume you know the answer to the following questions:
   (a) What is the input and output for Dijkstra's algorithm?
   (b) What is the running time of Dijkstra's algorithm using min-heap (aka priority queue) data structure? (Don't worry about the Fibonacci heap implementation or running time using it.)
   (c) What is the main idea for Dijkstra's algorithm?
2. [DPV] Problem 3.3 (Topological ordering example)
3. [DPV] Problem 3.4 (SCC algorithm example)
4. [DPV] Problem 3.5 (Reverse of graph)
5. [DPV] Problem 3.8 (Pouring water), if your book has a part (c) you can skip it... or not... this is just practice!
6. [DPV] Problem 4.13 (cities and highways).

Instructions.

For the graded problems, you are allowed to use the algorithms from class as black-boxes without further explanation. These include

- DFS (outputs connected components, a path between two vertices, topological sort on a DAG. You also have access to the pre and post arrays!), BFS and the Explore subroutine.
- Dijkstra's algorithm to find the shortest distance from a source vertex to all other vertices and a path can be recovered backtracking over the pre labels.
- Bellman-Ford and Floyd-Warshall to compute the shortest path when weights are allowed to be negative.
- SCCs which outputs the strongly connected components, and the metagraph of connected components.
- Kruskal's and Prim's algorithms to find MST.
- Ford-Fulkerson and Edmonds-Karp to find max flow on networks.

When using a black-box, make sure you clearly describe which input you are passing to it and how you use the output from or take advantage of the data structures created by the algorithm. To receive full credit, your solution must:

- Include the description of your algorithm in words (no pseudocode!).
- Explain the correctness of your design.
- State and analyse the running time of your design (you can cite and use the running time of black-boxes without further explanations).

Unless otherwise indicated, black-box graph algorithms should be used without modification.

Example: I take the input graph G, I first find the vertex with largest degree, call it v^*. I take the complement of the graph G, call it $\overline{G}$. Run Dijkstra's algorithm on $\overline{G}$ with $s = v^*$ and then I get the array $dist[v]$ of the shortest path lengths from s to every other vertex in the graph $\overline{G}$. I square each of these distances and return this new array.

We don't want you to go into the details of these algorithms and tinker with it, just use it as a black-box as showed with Dijkstra's algorithm above. Make sure to explain your algorithm in words, no pseudocode.

Problem 1 [DPV] Problem 3.15 (Computopia)

Disclaimer: Your solution should describe your algorithm with words (no pseudocode!) and explain why it is correct. You must state and justify the runtime.

- Assume that the Town Hall sits at 1 single intersection.
- The mayor's claim in (b) is the following: If you can get to another intersection, call it u, from the Town Hall intersection, then you can get back to the Town Hall intersection from u.
- Note, linear time means $O(n + m)$ where $n = |V|$ and $m = |E|$.

Part (a):

Part (b):

Problem 2 (All edges in shortest paths)

Given an undirected graph $G = (V, E)$ with positive edge weights and two nodes $s, t \in V$, design an efficient algorithm to determine the set of all edges that lie on at least one shortest path from s to t. Describe your algorithm with words (no pseudocode!) and explain why it is correct. You must state and justify the runtime.