1. **DFS mechanics.** For each graph, show the “DFS forest” resulting from an execution of DFS. Whenever there is a choice of vertices, choose the one that is alphabetically first. Identify the cross, forward, and back edges, and label each vertex with its discovery and finishing time.

![Graph](image1)

(i)

(ii)

2. **DFS mechanics.** Run the DFS-based topological sort algorithm on the following graph. Whenever there is a choice of vertices, choose the one that is alphabetically first. Show the “DFS forest”, including discovery and finishing times. Give the resulting topological ordering of the vertices.

![Graph](image2)

3. **DFS mechanics.** Run the strongly-connected components algorithm on this graph. Show the “DFS forest”, including discovery and finishing times, for both runs of the DFS algorithm. Draw the resulting component graph. As usual, when faced with a choice among vertices, pick the one that is alphabetically first.

![Graph](image3)

4. **Dijkstra mechanics.** Run Dijkstra’s algorithm on the following graph, starting from vertex A. Whenever there is a choice of vertices, choose the one that is alphabetically first. Show the state of the algorithm just after each vertex is removed from the queue.

![Graph](image4)
5. **Carrying stones.** You are given a directed acyclic graph $G = (V, E)$ with $n$ vertices and $m$ edges, along with two distinguished vertices $s, t \in V$. At each vertex $v \in V$, there are a number of stones $q(v)$ (so $q(v)$ is a nonnegative integer). Your goal is to find a path from $s$ to $t$, picking up stones along the way, and arrive at $t$ carrying as many stones as possible. At each node $v$ along the path, you are allowed to pick up at most $q(v)$ stones (and for simplicity, assume that $q(t) = 0$). However, there is a complication: each edge $e \in E$ has a capacity $c(e)$ (which is a positive integer), and you are not allowed to carry more than $c(e)$ stones across that edge. If necessary, you can drop stones before crossing an edge.

Show how to solve this problem in time $O(n + m)$. The output should be an optimal path from $s$ to $t$, and for every vertex along the path, your output should include the number of stones to be picked up or dropped at that vertex. Assume the the graph is given in adjacency list form, and that you can fetch the $q$ and $c$ values in constant time.

Feel free to use any standard algorithms as subroutines.

6. **Gotham City.** The mayor of Gotham City has made all the streets of the city one way. We can model the street layout as a directed graph $G = (V, E)$, where the vertices $V$ of the graph correspond to street intersections (locations), and the edges $E$ correspond the the roads connecting intersections. So for every pair of distinct vertices $u, v \in V$, if there is an edge from $u$ to $v$, then there is no edge from $v$ to $u$. Assume a sparse representation for $G$.

(a) The mayor claims that one can legally drive from any one location in the city to any other. Show how to verify the mayor’s claim using a linear-time algorithm that takes as input the graph $G$.

(b) Suppose the mayor’s claim is not necessarily true: that it may not be possible to legally drive from any one location to any other.

Let us call a location $v$ safe if wherever you legally drive starting at $v$, you can always legally drive back to $v$. Said another way, $v$ is safe if the following property holds: for every location $w$, if one can legally drive from $v$ to $w$, then one can legally drive back from $w$ to $v$.

Give a linear-time algorithm that outputs all of the safe locations, given the graph $G$ as input.

(c) Suppose again that the mayor’s claim is not necessarily true. Nevertheless, an emergency arises, and you must drive from location $s$ to location $t$, even though there may not be a legal route from $s$ to $t$. You want to plan a route that traverses some edges in the graph in the reverse (i.e., illegal) direction. In order to minimize the chance of getting caught by the police, your planned route should minimize the number of illegal edges. The number of legal edges in the route is irrelevant.

Give a linear-time algorithm that computes a route from $s$ to $t$ that minimizes the number of illegal edges (or reports that no route exists), given the graph $G$ and locations $s$ and $t$ as input.

Feel free to use any standard algorithms as subroutines.

7. **Help!** You are given a directed graph $G = (V, E)$, along with vertices $s, t \in V$. Some edges are labeled with a letter ‘a’–’z’, while other edges have no label. Your goal is to find a path from $s$ to $t$ that spells out the word “help”. So the path should consist of edges labeled ‘h’, ‘e’, ‘l’, ‘p’, in that order, along with any number of other edges. These other edges may be either labeled or unlabeled, but unlabeled edges are preferred. So among all such paths, you should find one with the minimum number of labeled edges.

Design a linear time algorithm to solve this problem.

Feel free to use any standard algorithms as subroutines.

8. **Invasion.** Two armies simultaneously invade a country. Let’s call them the “red army,” and the “blue army.” The red army starts out occupying city $a$, and the blue army starts out occupying city $b$. Both armies fan out simultaneously in all directions, and whichever army arrives at a city first, occupies that city, and blocks the other army from either occupying or transiting through that city. The occupying army leaves a small occupation force at that city, but the remainder of the army continues to fan out to all neighboring cities. In case of a tie, the city is occupied by neither army, and neither army may transit the city. The army that occupies the most cities wins the war. The question is: which army wins? Let’s model this problem as a directed graph with positive edge weights. The nodes in the graph represent the cities, and the edges represent roads between cities. The weight of an edge $(u, v)$ represents the amount of time required for either army to travel from city $u$ to city $v$. Design and analyze an efficient algorithm to solve this problem. The input is a directed, weighted graph, along with distinct nodes $a$ and $b$. The output is “red wins,” “blue wins,” or “tie.”
Observations and hints: You might think that you could just run Dijkstra twice, once starting at a and once starting at b. However, the tie-breaking rule means that this won’t work. Why? You might want to come up with an example graph that shows why this does not work.

To solve this, you will have to modify Dijkstra’s algorithm. The idea is to associate with each city two pieces of information: a running estimate for red’s best time to reach that city, a running estimate for blue’s best time to reach that city. In every step of the algorithm, we greedily choose one city whose status is “undecided” and move it into the “decided” category, assigning it to either red, blue, or neither, and then update the estimates for the other undecided cities accordingly.

Fill in the details of the above idea, and try to carefully prove the correctness of it using a loop invariant similar to what was done in class for Dijkstra. In your proof, you should identify where we used the fact that the edge weights are positive. You can use a priority queue in your algorithm, without worrying about how that is implemented.

9. Subway madness. The city of Brombus has n subway stations \( v_1, \ldots, v_n \) and k subway systems \( B_1, \ldots, B_k \). Each system \( B_s \) has the following structure: there is a fixed entrance fee \( p_s \) to enter \( B_s \) (from any station \( v_i \)), and there is a set of fares \( w_{ts}(v_i, v_j) \) to go from station \( v_i \) to station \( v_j \) on system \( B_s \). We assume that each entrance fee \( p_s \) is a positive number, and that each fare \( w_{ts}(v_i, v_j) \) is a positive number or \( \infty \).

When traveling from \( v_i \) to \( v_j \) from the street, one can enter any subway system \( B_s \), pay admission \( p_s \), travel to some intermediate city \( v_\ell \) by paying the fare \( w_{ts}(v_i, v_\ell) \), and then (if necessary) repeat the same process, using either a different subway system (and paying its entrance fee), or using the same subway system (without needing to pay the entrance fee again), until one reaches the desired destination \( v_j \).

Design an algorithm that takes as input the entrance fee and fare data as above, and outputs a table of values \( M_{ij} \) for \( i, j = 1, \ldots, n \), where \( M_{ij} \) is the minimum cost required to get from \( v_i \) to \( v_j \). Your algorithm does not need to compute the corresponding routes. State its complexity as a function of \( n \) and \( k \). For full credit, your algorithm should run in time \( O(n^3k) \).

Feel free to use any standard algorithms as subroutines.

Hint: First, run an all-pairs shortest path algorithm on each subway system.