1. **DFS Mechanics.** For the following graph, show the “DFS forest” resulting from an execution of DFS. You should start with vertex $q$, and 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 times.

![Graph](image)

2. **Dijkstra Mechanics.** Suppose we run Dijkstra’s algorithm on the following graph, starting from the node labeled $A$. Show how the $d$ array evolves over the course of the algorithm’s execution. More precisely, show the contents of the $d$ array just before the first loop iteration and at the end of every loop iteration. As usual, whenever there is a choice, give preference to the vertex which comes alphabetically first.

![Graph](image)

3. **Complete paths.** In a graph, a *complete path* is a path that visits every node.

   (a) Give a linear time algorithm that takes as input an acyclic directed graph and determines whether it contains a complete path.

   (b) Now generalize your algorithm so that it also works in linear time on arbitrary directed graphs. Note that a complete path need not be simple (i.e., it may contain repeated nodes).

4. **Blackish nodes.** Consider a directed graph $G = (V,E)$, where each node has a color $c[v]$, which is either black or white. A node $v$ is called blackish if there is a path from $v$ that passes through all black nodes.

   (a) Give a linear time algorithm that takes as input an acyclic directed graph, along with the node colors, and finds all blackish nodes.

   (b) Now generalize your algorithm so that it also works in linear time on arbitrary directed graphs.

5. **Minimum cost spanning set.** Design a linear time algorithm for the following problem. Consider a directed graph $G = (V,E)$ where every vertex $v \in V$ has a positive weight $wt(v)$. A set $S \subseteq V$ is called a spanning set if every vertex in $V$ is reachable from some vertex in $S$ (that is, for every $v \in V$, there exists $s \in S$ such that $v$ is reachable from $s$). The weight of $S$ is defined to be $\sum_{v \in S} wt(v)$. Design a linear time algorithm that takes as input a graph $G$, along with the vertex weights, and finds a spanning set of minimum weight.
10. **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 \( wt_s(v_i, v_j) \) 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 \( wt_s(v_i, v_j) \) 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 \( u_l \) by paying the fare \( wt_s(v_i, u_l) \), and then (if necessary) repeat the same process, using

---

**Hint:** First consider the problem for acyclic graphs where every vertex has weight 1. Show that such graphs have a unique spanning set of minimum weight — what property characterizes the vertices in this set?

6. **Net income.** You are given a directed acyclic graph \( G = (V, E) \) with \( n \) vertices and \( m \) edges, along with two distinguished vertices \( s, t \in V \). Each edge \( e \in E \) has a cost \( c(e) \), which is a nonnegative integer. Each vertex \( v \in V \) has an income \( I(v) \in E \), which is also a nonnegative integer. For a path \( p = (v_0, v_1, \ldots, v_k) \), we define the net income of \( p \) to be the sum of the income derived from the vertices in \( p \), minus the sum of all the edges in \( p \).

Intuitively, you can think of the problem as follows: on each vertex there is some money, which you can pick up if you visit that vertex. In addition, every time you cross an edge, you pay a toll.

Your goal is to find a path from \( s \) to \( t \) in \( G \) that generates the most net income. For simplicity, you can just focus on the problem of calculating the maximum net income, rather than the path. Assume the the graph is given in adjacency list form, and that you can fetch \( I \) and \( c \) values in constant time.

Give an algorithm that solves this problem in time \( O(n + m) \).

**Hint:** use a standard algorithm as a subroutine.

**Note:** One can generalize this problem to arbitrary (possibly cyclic) graphs. In this generalization, you may visit nodes and cross edges several times; once you visit a node and pick up the money, it is gone, so visiting it again will not generate any additional income; however, you have to pay a toll each and every time you cross an edge. There is no known efficient algorithm for this generalization (in fact, it is NP complete).

7. **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.

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 at that vertex. Assume the the graph is given in adjacency list form, and that you can fetch \( q \) and \( c \) values in constant time.

**Hint:** use a standard algorithm as a subroutine.

**Note:** You may want to first solve a variant of the above problem in which at each vertex, in addition to picking up stones, you may also throw some stones away. It should be easy to modify your solution to this variant to obtain a solution to the original problem.

8. **Colorful paths (I).** You are given a directed graph \( G = (V, E) \), and nodes \( s, t \). Nodes are colored red, yellow, blue, or white. Let us say a path (not necessarily simple) is colorful if it runs from \( s \) to \( t \), and contains at least one red, one yellow, and one blue node. Design and analyze an algorithm that determines if there is a colorful path, and if so, finds one containing a minimum number of white nodes. Your algorithm should run in time \( O(|V| + |E|) \).

**Hint:** use a standard algorithm as a subroutine.

9. **Colorful paths (II).** You are given a directed graph \( G = (V, E) \), and nodes \( s, t \). Nodes are colored red, yellow, blue, or white. A path (not necessarily simple) is called \( k \)-colorful if it runs from \( s \) to \( t \) and contains at least \( k \) red nodes, \( k \) yellow nodes, and \( k \) blue nodes (counting any repetitions of nodes). \( G \) is called dazzling if there is a \( k \)-colorful path for every \( k > 0 \). Design and analyze an algorithm to determine if \( G \) is dazzling. Your algorithm should run in time \( O(|V| + |E|) \).

**Hint:** use a standard algorithm as a subroutine.


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.

11. **Challenge problem: 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. 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.

12. **Challenge problem: Min-max graphs.** Consider a directed graph \( G = (V, E) \). Each node is labeled as either a *min* node, a *max* node, or a *constant* node. A constant node \( u \) has no outgoing edges, and is also labeled with \( c(u) \in \mathbb{R} \). A *value assignment* for \( G \) is a function \( \alpha : V \rightarrow \mathbb{R} \cup \{-\infty, \infty\} \). For any given value assignment \( \alpha \), we define a new value assignment \( \alpha^G \) as follows:

(i) for all *min* nodes \( u \), \( \alpha^G(u) = \min \{ \alpha(v) : (u, v) \in E \} \cup \{\infty\} \);

(ii) for all *max* nodes \( u \), \( \alpha^G(u) = \max \{ \alpha(v) : (u, v) \in E \} \cup \{-\infty\} \);

(iii) for all *constant* nodes \( u \), \( \alpha^G(u) = c(u) \).

\( \alpha \) is called a *fixed point* of \( G \) if \( \alpha^G = \alpha \). Observe that for any fixed point, a *min* node with no outgoing edges is assigned \( \infty \), while a *max* node with no outgoing edges is assigned \( -\infty \).

There may be more than one fixed point for \( G \). A fixed point \( \alpha \) is called a *least fixed point* if for any other fixed point \( \beta \), we have \( \alpha(u) \leq \beta(u) \) for all \( u \in V \). It is clear that a least fixed point, if it exists, must be unique.

(a) Your first task is to prove that a least fixed point always exists. To this end, let \( \alpha_0 \) be the value assignment that assigns \( -\infty \) to all nodes. For \( i = 0, 1, 2, \ldots \), let \( \alpha_{i+1} := \alpha_i^G \). Show that there exists some \( k \geq 0 \) such that

\[
\alpha_k = \alpha_{k+1} = \alpha_{k+2} = \cdots,
\]

and moreover, for this \( k \), \( \alpha_k \) is the least fixed point of \( G \).
(b) Suppose that $G$ contains $N$ distinct values labeling its constant nodes. Show that the value $k$ in part (a) is at most $|V|(N + 1)$. Based on this, describe a simple algorithm to compute the least fixed point of $G$ in time $O(|E||V|(N + 1))$.

(c) Suppose that $G$ is acyclic. Show how to compute the least fixed point of $G$ in time $O(|V| + |E|)$.

(d) Suppose that $G$ contains no constant nodes. In this situation, we can naturally interpret $-\infty$ as false, $\infty$ as true, $\min$ as and, and $\max$ as or, and the least fixed point represents the minimal, consistent truth-value assignment. Show how to find the least fixed point of $G$ in this case in time $O(|V| + |E|)$.

Note: the solution to part (d) may be viewed as a special case of the solution to part (e), but it may be easier to tackle this simpler problem first; alternatively, you can first show how to solve part (e), and then adapt that solution to part (d).

(e) In the general setting, show how to find the least fixed point of $G$ in time $O((|V| + |E|) \log |V|)$.