1. An undirected graph $G = (V, E)$ is said to be *bipartite* if the vertex set V can be partitioned into disjoint subsets V_L and V_R (the "left" and "right" vertices, respectively) so every vertex is in exactly one of V_L or V_R, and every edge goes between some vertex in V_L and one in V_R.

   Give an $O(V + E)$-time algorithm (and prove it correct) that determines whether a graph is bipartite, and if so, outputs valid sets V_L and V_R. Make sure your algorithm works even if the graph is unconnected! (Recall that an undirected graph is *connected* if it is possible to reach any vertex from any other vertex by following edges in the graph.)

   Solution: The main observation is that in a bipartite graph with partition V_L, V_R, node distances (from an arbitrary source s) correspond to which side of the partition the node belongs to: assuming (with no loss of generality) that $s \in V_L$, all nodes in V_L are at even distances from s, and nodes in V_R are at odd distances from s. This is because *any* path (and in particular, any shortest path) from s to a vertex v must cross between V_L and V_R for each edge on the path. (If the graph is unconnected, then the same applies for each connected component using an arbitrary new source vertex.)

   Conversely, if the BFS node distances (from some s) are such that every edge in the graph is between a node with odd distance and one with even distance, then the graph is bipartite: simply put all nodes with even distances in V_L, and those with odd distances in V_R. In other words, a graph is bipartite *if and only if* all edges are between nodes with odd and even distances, respectively.

   We can use this observation in an algorithm in various ways. The simplest way is simply to run BFS from an arbitrary vertex (for each connected component), and then check that every edge goes between nodes with even/odd distances; each of the two phases takes $O(V + E)$ time. Another way is to modify BFS itself so that whenever it re-discovers a node (i.e., when the node is already black), it checks the constraint on the node distances.

2. Do CLRS Exercise 22.4-3. Pay close attention to the $O(V)$ runtime requirement.

   Solution: We use the fact that if an undirected graph has $\geq |V|$ edges, then it must have a cycle. (Some graphs with fewer than $|V|$ edges can have cycles as well.) Therefore, we can run DFS (the one that eventually visits all connected components of the graph), keeping track of the total number of edges seen so far. If the DFS ever encounters a back edge, it outputs "cycle;" if the DFS ever sees a total of $|V|$ total edges, it also outputs "cycle;" if the DFS completes without encountering either of these conditions, it outputs "no cycle." The algorithm is correct: if it outputs "cycle" then there definitely is a cycle in the graph (as already argued), whereas if it outputs "no cycle" then it completed a full DFS without encountering a back edge, so the graph has no cycle. The runtime is only $O(V)$ because we inspect at most $|V|$ vertices and a total of at most $|V|$ edges before the algorithm finishes.

3. Consider the following 2-player game played on a directed acyclic graph (DAG) G with some starting vertex s. First, player 1 chooses an edge from s to some node v_1. Then, player 2 picks an edge from v_1

to some node v_2. Player 1 then picks an edge from v_2 to some node v_3, and so on. A player wins by putting its opponent at a node with no out-edges (i.e., the opponent cannot make a legal move).

Give an efficient algorithm that, given G and s, determines which player (if any) has a winning strategy (i.e., a way to guarantee a win no matter what the opponent does). If a winning strategy exists for a player, the algorithm should also output a table telling what move that player should make at each vertex in order to guarantee a win.

Solution: Our solution will follow from two main observations. The first is that since we are dealing with a DAG, we might as well topologically sort it (in $O(V + E)$ time) to order the vertices $1, 2, 3, \ldots, n = |V|$, so that every edge $(i, j) \in E$ goes from "left to right," i.e., $i < j$.

The second observation is a recursive formulation of whether a player has a winning strategy, based on the vertex from which she is about to move. Let's define $W[i]$ to be a boolean value which is true if there is a winning strategy for the player who is about to move from node i. Note that it doesn't matter whether player 1 or 2 is about to move, since the same rules apply to both players. All that matters is whether the player who is about to move from node i can guarantee a win against her opponent.

Let's first consider the base case(s). Clearly, if some node i has no outgoing edges, then $W[i] =$ false, because a player at node i has no legal moves and hence immediately loses. If node i *does* have an outgoing edge, then we have the following recursive reasoning: the player at node i must choose an out-neighbor j of i, at which point the opponent will have to move from node j. Of course, if the opponent has a winning strategy from node j (i.e., if $W[j] =$ true), then moving to j is clearly not a winning strategy for the player, because the opponent could then guarantee a win for itself. Whereas if $W[j] =$ false, the moving to j *is* is winning strategy for the player, because the opponent cannot guarantee a win (and somebody has to eventually win the game). Therefore we have two cases:

- If $W[j] =$ true for *every* out-neighbor j of i, then the player does *not* have a winning strategy, and $W[i] =$ false.
- Otherwise, there is *some* out-neighbor j of i for which $W[j] =$ false, so $W[i] =$ true.

In conclusion, we have the following rule, which applies in all cases: $W[i] =$ true if and only if there is some edge (i, j) for which $W[j] =$ false. We can write this more compactly as the logical formula $W[i] = \neg \bigwedge_{(i,j)\in E} W[j]$.

To turn this into a dynamic programming algorithm, we just topologically sort the graph and then fill in the values $W[n], W[n-1], W[n-2], \ldots, W[1]$. We go in this order because in order to set the value $W[i]$, we need to know the values $W[j]$ for $j > i$, due to the topological ordering. To set $W[i]$ according to the above rule, we just need to traverse the adjacency list of node i and do a constant amount of work for each edge (i, j), so altogether we do only $O(V + E)$ work.

Once that work is done, we can say that player 1 has a winning strategy if and only if $W[s] =$ true (where s is the starting vertex), otherwise player 2 has a winning strategy. To generate a table of moves that guarantee a win, we do the following when filling in the table W: for each vertex i for which $W[i] =$ true we just remember one of i's out-neighbors j for which $W[j] =$ false (such a j must exist by the recurrence for W). Any player who follows this table will guarantee a win for herself, because she will always keep her opponent on nodes j for which $W[j] =$ false.

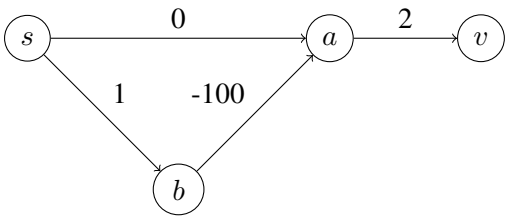
4. Do CLRS Exercise 24.1-3.

Solution: In Bellman-Ford, introduce a boolean flag to keep track of whether a pass over the edges change the value of any $v.d$. At the beginning of each pass, set the flag to false, and if at any point during the pass some value of $v.d$ changes (due to a relaxation), set the flag to true. At the end of the pass over all the edges, if the flag is still false, terminate the algorithm.

This works for the following reasons. Recall that for every $v \in V$ there is a shortest path from s to v having at most m edges. Then by the path relaxation property, after m passes over the edges, every $v.d = \delta(s, v)$. (See the correctness proof of Bellman-Ford for more details.) Following this, the values of $v.d$ never change, because they can never increase and they never go below $\delta(s, v)$. So, the flag remains false during the $(m + 1)$st pass, and the algorithm terminates after this pass.

5. Do CLRS Exercise 24.3-2.

Solution: Here is a simple example graph that serves as a counterexample.



Consider Dijkstra's algorithm run from s in the above graph. After relaxing the edges out of s, we have $a.d = 0, b.d = 1$, so the next node taken from the priority queue is a. After relaxing the edge out of a, we have $v.d = 2$, so the next node taken from the priority queue is b. After relaxing the edge out of b, we have $a.d = -99$. But since a was already removed from the priority queue, the only node left to remove is v (and no edged are relaxed). So at the end, $v.d = 2$ even though the true shortest path distance is $\delta(s, v) = -97$. Dijkstra fails on this example because the edges of the true shortest path $s \to b \to a \to v$ are *not* relaxed in order, so we cannot invoke the path-relaxation property.

Note that we needed the node v to make this a true counterexample. Without it, even though $a.d = 0$ is incorrect when a is removed from the queue, we eventually correctly set $a.d = -99 = \delta(s, a)$ when the edges out of b are relaxed.

6. Suppose you are given a weighted graph $G = (V, E)$ having no negative-weight edges, where the attributes $v.d$ and $v.\pi$ for all $v \in V$ have already been set. Give an $O(V + E)$-time algorithm (and prove it correct) that determines whether these attributes are consistent with a shortest-paths tree from a given source vector $s \in V$.

Solution: We can solve this problem by making the following observations about the $v.d$ and $v.\pi$ values of a valid shortest-paths tree. First, we have $s.d = 0$ and $s.\pi =$ nil, and the edges $(v.\pi, v) \in E$ define a tree rooted at s, where s is the source. Also, for each edge $(u, v) \in E$, we must have $v.d \leq u.d + w(u, v)$ with equality when $u = v.\pi$. This is because $v.d = \delta(s, v)$, the true shortest path distance from s to v. (By convention, we let $w(u, v) = \infty$ if $(u, v) \notin E$.)

It turns out that the above conditions are also *sufficient*, i.e., if they all hold, then the values $v.d$ and $v.\pi$ correspond to a valid shortest-paths tree (rooted as s). To prove this, first note that since all the conditions hold, the $v.d$ and $v.\pi$ values give us a tree rooted at s, and the path in the tree from s to each v has weight exactly $v.d$. Now suppose for contradiction that there is some vertex v such that a true shortest path (in the whole graph) from s to v has length $\delta(s, v) < v.d$. Let v be such a vertex having a shortest path of the *fewest* total edges, say T. Since $\delta(s, s) = 0 = s.d$, we must have $v \neq s$, and so we can let u be v's predecessor on a shortest path from s to v having T edges. Then we must have $u.d = \delta(s, u)$, because there is a shortest path from s to u of only $T - 1$ edges. But now because all the above tests pass, we have $v.d \leq u.d + w(u, v) = \delta(s, u) + w(u, v) = \delta(s, v)$, which contradicts our assumption that $v.d > \delta(s, v)$. This proves the claim. (Note that we did not even need to assume that edge weights are nonnegative, only that shortest paths are well defined – so we only need to assume that there are no negative-weight cycles.)

Therefore, to test that the $v.d$ and $v.\pi$ values actually correspond to a valid shortest-paths tree, we just need to test all of the above conditions. This can be done in $O(V + E)$ time, using DFS to check the tree constraint and by doing constant work for each edge.