1. **Dijkstra’s algorithm with bounded edge weights.** Suppose the input to Dijkstra’s algorithm is a weighted graph \( G = (V, E) \), where each edge \( e \in E \) has a weight \( w(e) \in \{0, 1, \ldots, B\} \). Show how to implement Dijkstra’s algorithm so that it runs in time \( O(|V|B + |E|) \).

*Hint:* first, argue that if Dijkstra’s algorithm removes elements from \( Q \) in order \( v_0, v_1, v_2, \ldots \), then \( \delta(s, v_i) \leq \delta(s, v_{i+1}) \) for \( i = 0, 1, \ldots \).

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

*Hint:* reduce to an ordinary shortest path problem; you may also use the result of Exercise 1.

3. **Running on empty.** In this problem, we want to determine how to drive a car from point \( s \) to point \( t \) without running out of gas. Our car has a gas tank that is initially filled up to its capacity \( c \). We may have to stop to refuel along the way: we can never allow the amount of gas in the tank to become negative, and we can never fill our tank beyond its capacity. Let us model this problem using a directed graph \( G = (V, E) \) with non-negative edge weights \( w : E \to \mathbb{R}_{\geq 0} \), along with nodes \( s, t \in V \). For an edge \( (u, v) \in E \), \( w(u, v) \) represents the amount of gas required to drive from \( u \) to \( v \). Also, a subset \( F \subset V \) represents those places where we may stop to refuel (assume each node \( v \) is marked with a bit \( f(v) \) indicating whether there is a gas station located at \( v \)). Give an efficient algorithm to solve this problem. Your algorithm should determine if there is a viable path from \( s \) to \( t \), and if so, output such a path. State the running time of your algorithm as a function of \( |V|, |E|, \) and \( |F| \).

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

4. **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.

5. **Arbitrage.** In class, we studied the Bellman-Ford algorithm. Suppose the input is a directed graph \( G = (V, E) \) with edge weights \( w : E \to \mathbb{R} \), along with a start node \( s \). Recall the “predecessor graph” \( G_\pi \) defined by the array \( \pi \) in that algorithm: if \( \pi[v] = u \), then \( (u, v) \) is an edge in \( G_\pi \). We proved that if the graph \( G \) has no negative weight cycles reachable from \( s \), then when the algorithm terminates, the graph \( G_\pi \) is acyclic, and in fact, is a tree that encodes shortest paths from \( s \) to every node reachable from \( s \).

The goal of this exercise is to prove the converse: if the graph \( G \) does contain a negative weight cycle that is reachable from \( s \), then the graph \( G_\pi \) contains a cycle \( C \). This will allow us to efficiently find a negative weight cycle in \( G \), if one exists.
(a) For \( u, v \in V \), define \( \delta'(u, v) \) to be the minimum weight of any simple path from \( u \) to \( v \) (recall that a simple path is one in which no vertices repeat). Prove that after \( |V| - 1 \) iterations of the main loop of Bellman-Ford, we have \( d[v] \leq \delta'(s, v) \) for all \( v \in V \).

*Hint:* use relaxation property R3.

(b) Assume \( G \) has a negative weight cycle reachable from \( s \) and that we have run Bellman-Ford to termination. Prove that \( d[v^*] < \delta'(s, v^*) \) for some \( v^* \in V \).

*Hint:* use part (a) and properties that we proved in class.

(c) Assume \( G \) has a negative weight cycle reachable from \( s \) and that we have run Bellman-Ford to termination. Let \( v^* \) be as in part (b), so that \( d[v^*] < \delta'(s, v^*) \). Further, suppose that \( d[s] = 0 \). Consider the following algorithm:

\[
v \leftarrow v^*
\]

while \( v \neq s \) do

\[
v \leftarrow \pi[v]
\]

Prove that this algorithm cannot terminate. From this, conclude that there is a cycle in \( G_{\pi} \).

*Hint:* suppose the algorithm does terminate; use relaxation property R4 and the assumption that \( d[s] = 0 \) to show that \( d[v^*] \geq \delta'(s, v^*) \), which is a contradiction.

(d) Suppose that when Bellman-Ford terminates, we have \( d[s] < 0 \). Show that \( G_{\pi} \) must contain a cycle.

*Hint:* if \( d[s] < 0 \) then \( \pi[s] \neq \text{Nil} \).

*Note:* Parts (c) and (d) together show that if \( G \) contains a negative weight cycle reachable from \( s \), then \( G_{\pi} \) must contain a negative weight cycle when Bellman-Ford terminates. Using DFS, or by other means, we can efficiently find a cycle in \( G_{\pi} \). Results from class show that any cycle in \( G_{\pi} \) is a negative weight cycle in \( G \).

(e) Now suppose we are given a weighted graph \( G \), and we want to determine if it contains a negative weight cycle, and if so, to find such a cycle. Show how to do this efficiently using Bellman-Ford. The only difference here is that we are looking for any negative weight cycle, not just one reachable from a given node \( s \).

*Hint:* you could do this by running Bellman-Ford for every possible \( s \in V \), but there is a better way.

6. **Shortest path mechanics.** Consider the following graph.

![Graph](image)

(a) Run Bellman-Ford starting from \( s = 2 \), using the following edge ordering:

\[
(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (2, 5), (4, 3), (5, 1), (5, 3).
\]

Show the evolution of the \( d \) and \( \pi \) arrays.

(b) Demonstrate that if you ran Dijkstra from \( s = 2 \), the output would be incorrect. What went wrong?

(c) Run Floyd-Warshall. Show the \( D \) array at the beginning and after each iteration of the main loop.

7. **MST mechanics.** Run Prim’s MST algorithm on the following graph. Whenever there is a choice of nodes, always use alphabetic ordering (e.g., start from node A). Show the evolution of the \( d \) and \( \pi \) arrays.

![Graph](image)