1. Suppose that you have just finished computing the array \( \text{dist}[1..V, 1..V] \) of shortest-path distances between all pairs of vertices in an edge-weighted directed graph \( G \). Unfortunately, you discover that you incorrectly entered the weight of a single edge \( u \to v \), so all that precious CPU time was wasted. Or was it? Maybe your distances are correct after all!

In each of the following problems, let \( w(u \to v) \) denote the weight that you used in your distance computation, and let \( w'(u \to v) \) denote the correct weight of \( u \to v \).

(a) Suppose \( w(u \to v) > w'(u \to v) \); that is, the weight you used for \( u \to v \) was larger than its true weight. Describe an algorithm that repairs the distance array in \( \mathcal{O}(V^2) \) time under this assumption. [Hint: For every pair of vertices \( x \) and \( y \), either \( u \to v \) is on the shortest path from \( x \) to \( y \) or it isn’t.]

**Solution:** The algorithm is similar to problem 3 on the previous homework.

```plaintext
REPAIRDISTANCES(dist, w'(u \to v)):
    for every vertex \( s \)
        for every vertex \( t \)
            if dist[s, t] > dist[s, u] + w'(u \to v) + dist[v, t]
                dist[s, t] ← dist[s, u] + w'(u \to v) + dist[v, t]
```

(b) Maybe even that was too much work. Describe an algorithm that determines whether your original distance array is actually correct in \( \mathcal{O}(1) \) time, again assuming that \( w(u \to v) > w'(u \to v) \). [Hint: Either \( u \to v \) is the shortest path from \( u \) to \( v \) or it isn’t.]

**Solution:** The edge \( u \to v \) appears in any shortest path if and only if \( u \to v \) itself is a shortest path from \( u \) to \( v \). Thus, if \( u \to v \) is not the unique shortest path from \( u \) to \( v \) after fixing its weight, then all shortest paths can avoid \( u \to v \), which means all the old distances are correct. On the other hand, if \( \text{dist}[u, v] > w'(u \to v) \), then at least \( \text{dist}[u, v] \) is incorrect.

```plaintext
CHECKDISTANCES(dist, w'(u \to v)):
    if dist[u, v] ≤ w'(u \to v)
        return True
    else
        return False
```

(c) **To think about later:** Describe an algorithm that determines in \( \mathcal{O}(VE) \) time whether your distance array is actually correct, even if \( w(u \to v) < w'(u \to v) \).

**Solution:** If \( w(u \to v) < w'(u \to v) \), we need to compute the correct shortest-path distance from \( u \) to \( v \). If this new distance is equal to the old value \( \text{dist}[u, v] \), then \( u \to v \) was not a shortest path under the old weights, so all (old and new) shortest paths avoid \( u \to v \), so all the old distances are correct. Otherwise, at least the distance \( \text{dist}[u, v] \) is incorrect.

```plaintext
CHECKDISTANCES(dist, w'(u \to v)):
    compute \( \text{dist}'[u, v] \) via Bellman-Ford
    if \( \text{dist}'[u, v] = \text{dist}[u, v] \)
        return True
    else
        return False
```
(d) **To think about later:** Argue that when \( w(u \rightarrow v) < w'(u \rightarrow v) \), repairing the distance array requires recomputing shortest paths from scratch, at least in the worst case.

**Solution:** Let \( G \) be an arbitrary edge-weighted directed graph. Construct a new graph from \( H \) by adding two vertices \( u \) and \( v \), edges \( x \rightarrow u \) and \( v \rightarrow x \) with length 0 for every vertex \( x \) in \( G \), and an edge \( u \rightarrow v \) with weight \( -\infty \). Then every shortest path in \( H \) has length \( -\infty \), because it contains the edge \( u \rightarrow v \) and at most two other edges \( x \rightarrow u \) and \( v \rightarrow y \). In particular, the lengths of the edges in \( G \) are utterly irrelevant.

Now if we set \( w'(u \rightarrow v) = \infty \), then the new shortest path in \( H \) between two nodes of \( G \) is just their shortest path in \( G \). But we have absolutely no information about shortest paths in \( G \); all we have is a distance array full of incorrect \( -\infty \)s! We have no choice but to recompute all shortest paths in \( G \) from scratch.  ■
2. You—yes, you—can cause a major economic collapse with the power of graph algorithms! The arbitrage business is a money-making scheme that takes advantage of differences in currency exchange. In particular, suppose that 1 US dollar buys 120 Japanese yen; 1 yen buys 0.01 euros; and 1 euro buys 1.2 US dollars. Then, a trader starting with $1 can convert their money from dollars to yen, then from yen to euros, and finally from euros back to dollars, ending with $1.44! The cycle of currencies $\rightarrow ¥ \rightarrow € \rightarrow$ is called an arbitrage cycle. Of course, finding and exploiting arbitrage cycles before the prices are corrected requires extremely fast algorithms.

Suppose $n$ different currencies are traded in your currency market. You are given the matrix $R[1..n]$ of exchange rates between every pair of currencies; for each $i$ and $j$, one unit of currency $i$ can be traded for $R[i,j]$ units of currency $j$. (Do not assume that $R[i,j] \cdot R[j,i] = 1$.)

(a) Describe an algorithm that returns an array $V[1..n]$, where $V[i]$ is the maximum amount of currency $i$ that you can obtain by trading, starting with one unit of currency 1, assuming there are no arbitrage cycles.

**Solution:** Construct a complete graph $G$ on $n$ vertices with edge weights $w(i\rightarrow j) := -\log R[i,j]$.

Any sequence of trades that starts with one unit of currency $i$ and ends with $M$ units of currency $j$ corresponds to a path in $G$ from vertex $i$ to vertex $j$ with length $-\log M$. Conversely, any path of length $\ell$ from vertex $i$ to vertex $j$ corresponds to a sequence of trades that starts with one unit of currency $i$ and ends with $2^{-\ell}$ units of currency $j$. In particular, a negative cycle in $G$ would correspond to an arbitrage cycle; thus, $G$ has no negative cycles.

Compute the shortest paths from vertex 1 to every other vertex in $G$, using Shimbel’s algorithm, because some edge weights may be negative. Shimbel’s algorithm runs in $O(VE) = O(n^3)$ time. Finally, for each $j$, let $V[j] = 2^{-\text{dist}(j)}$, where $\text{dist}(j)$ is the shortest-path distance from vertex 1 to vertex $j$ computed by Shimbel. Computing the output array $V[1..n]$ requires only $O(n)$ additional time.

Alternatively, if we don’t like logs and exponents, we can modify Shimbel to multiply edge lengths instead of adding them, and to reverse the direction of all comparisons. Here is the resulting algorithm, which clearly runs in $O(n^3)$ time:

```
SHIMBELTRADING(R[1..n,1..n])
V[1] <- 1  
for i <- 2 to n
  V[i] <- 0
for k <- 1 to n - 1
  for i <- 1 to n
    for j <- 1 to n
      if V[j] \leq V[i] \cdot R[i,j]
        V[j] <- V[i] \cdot R[i,j]

return V[1..n]
```

(b) Describe an algorithm to determine whether the given array of currency exchange rates creates an arbitrage cycle.

Solution: One more iteration of Shimbel's algorithm detects negative cycles, so we can use almost the same algorithm as in part (a).

```
SHIMBEL_ARBITRAGE(R[1..n, 1..n])
V[1] ← 1
for i ← 2 to n
    V[i] ← 0
for k ← 1 to n − 1
    for i ← 1 to n
        for j ← 1 to n
            if V[j] ≤ V[i] · R[i, j]
                V[j] ← V[i] · R[i, j]
    for i ← 1 to n
        for j ← 1 to n
            if V[j] ≤ V[i] · R[i, j]
                return True
return False
```

*(c) To think about later: * Modify your algorithm from part (b) to actually return an arbitrage cycle, if such a cycle exists.

Solution: We further modify Shimbel's algorithm to maintain predecessor edges, exactly as described in the lecture notes. Then if there is a negative cycle in the graph, at least one such cycle is described by the predecessor edges; conversely, if the predecessor edges induce a cycle, the total weight of that cycle must be negative. Thus, we can find a negative weight cycle in \(O(V + E) = O(n^2)\) additional time using an obvious modification the ISACYCLIC algorithm in the notes.

Of course, the two underlying claims require proof.

Claim 1. If there is a negative cycle in the graph, then after \(n\) iterations of Shimbel's algorithm, there is a cycle in the graph of predecessor edges.

Proof: To simplify discussion, assume that every other vertex is reachable from \(s\) (as in the case in our arbitrage problems). If there is no negative cycles in the graph, then every vertex except \(s\) itself has an incoming predecessor edge when Shimbel's algorithm halts. If there is a negative cycle containing \(s\), the algorithm will relax some edge into \(s\), giving \(s\) an incoming predecessor edge. At that point, every vertex has an incoming predecessor edge. Thus, if we walking backward along those edges we will never get stuck; we must eventually repeat a vertex.

More generally, let \(N\) be the set of vertices reachable from \(s\) that lie on a negative cycle; obviously every vertex in such a negative cycle must lie in \(N\). After \(n\) iterations of Shimbel's algorithm, the predecessor of each vertex in \(N\) is also in \(N\), which implies that there must be a cycle among the predecessor edges in \(N\).

Claim 2. If there is a cycle in the graph of predecessor edges after Shimbel's algorithm halts, the total weight of that cycle is negative.
Proof: Consider a predecessor cycle \( C = \nu_0 \rightarrow \nu_1 \rightarrow \cdots \rightarrow \nu_{\ell-1} \rightarrow \nu_0 \), where for each index \( i \), we have \( \text{pred}(\nu_i) = (\nu_{i-1} \mod \ell) \). (I'll omit the “mod \ell” from now on.) For each index \( i \), define

\[
\tilde{w}(\nu_{i-1} \rightarrow \nu_i) = w(\nu_{i-1} \rightarrow \nu_i) - \text{dist}(\nu_i) + \text{dist}(\nu_{i-1})
\]

Just after the last time \( \nu_{i-1} \rightarrow \nu_i \) was relaxed, we had \( \tilde{w}(\nu_{i-1} \rightarrow \nu_i) = 0 \); since that time, \( \text{dist}(\nu_i) \) has not changed and \( \text{dist}(\nu_{i-1}) \) has not increased. It follows that \( \tilde{w}(\nu_{i-1} \rightarrow \nu_i) \leq 0 \) for all \( i \).

Suppose \( \nu_{i-1} \rightarrow \nu_i \) was the last edge in \( C \) to be relaxed. That relaxation decreased \( \text{dist}(\nu_i) \), and therefore decreased \( \tilde{w}(\nu_i \rightarrow \nu_{i+1}) \), so we must have \( \tilde{w}(\nu_{i-1} \rightarrow \nu_i) < 0 \). (Equivalently, \( \nu_i \rightarrow \nu_{i+1} \) must be tense!)

Finally, we can express the total length of \( C \) in terms of the adjusted weights \( \tilde{w} \) as follows:

\[
\ell - 1 \sum_{i=0}^{\ell-1} w(\nu_{i-1} \rightarrow \nu_i) = \ell - 1 \sum_{i=0}^{\ell-1} (\tilde{w}(\nu_{i-1} \rightarrow \nu_i) + \text{dist}(\nu_i) - \text{dist}(\nu_{i-1}))
\]

\[
= \sum_{i=0}^{\ell-1} \tilde{w}(\nu_{i-1} \rightarrow \nu_i) + \ell - 1 \sum_{i=0}^{\ell-1} \text{dist}(\nu_i) - \ell - 1 \sum_{i=0}^{\ell-1} \text{dist}(\nu_{i-1})
\]

\[
= \sum_{i=0}^{\ell-1} \tilde{w}(\nu_{i-1} \rightarrow \nu_i)
\]

Every term in the final sum is non-positive, and at least one term is negative. We conclude that \( C \) is a negative cycle. \( \square \)