1. Arbitrage is the use of discrepancies in currency exchange rates to make a profit. For example, there may be a small window of time during which 1 U.S. dollar buys 0.75 British pounds, 1 British pound buys 2 Australian dollars, and 1 Australian dollar buys 0.70 U.S. dollars. At such a time, a smart trader can trade one U.S. dollar and end up with $0.75 \times 2 \times 0.7 = 1.05$ U.S. dollars, which is a profit of 5%. Suppose that there are n currencies $c_1, \ldots, c_n$ and an $n \times n$ table R of exchange rates, such that one unit of currency c_i buys $R[i, j]$ units of currency c_j. Devise and analyze an algorithm to determine the maximum value of
$R[c_1, c_{i_1}] \cdot R[c_{i_1}, c_{i_2}] \cdots R[c_{i_{k-1}}, c_{i_k}] \cdot R[c_{i_k}, c_1]$
(**10 points**)
Solution:
Construct a graph with vertices as currency denominations, and exchange rates (positive values) as edge weights. We are asked to find a cycle of length $k+1$ containing vertex c_1 such that the product of edge weights is maximized. A "brute force" approach to solving this problem is to enumerate all possible cycles of length $k+1$ containing vertex c_1 and determine the one with the maximum edge weight product value. This algorithm requires $O(n^k)$ time.

   Now, consider transforming the problem so that we can use a known shortest paths or shortest cycle-finding algorithm. One possible edge weight transformation we looked at previously was to take the negative logarithm of the exchange rates. Since all exchange rates are positive, we can apply this transformation. Maximizing the product of the original weights is equivalent to minimizing the sum of the transformed weights. Since edge weights can be negative, we want to find the minimum weight negative cycle of length k containing vertex c_1. We can apply the modified Bellman-Ford algorithm to detect if there's a negative weight cycle in the graph, but we cannot get the minimum weight cycle. So we need to fall back on the previously-described brute force approach.

   Grading scheme: 2 points for attempt + 3 points for saying we need to find a cycle where product of edge weights is maximized and suggesting the negative log transformation + 3 points for explaining why Bellman-Ford won't work + 2 points for giving brute-force approach.

2. Design and analyze an algorithm that takes as input an undirected graph $G = (V, E)$ and determines whether G contains a simple cycle of length four. Its running time should be at most $O(|V|^3)$. You may assume that the input graph is represented either as an adjacency matrix or with adjacency lists, whichever makes your algorithm simpler.
(**8 points**)
Solution:
Let us assume the graph G uses an adjacency list representation.

Two vertices u and v form a square if and only if they have at least two common neighbors, w and x. That is, w and x appear in both u's and v's adjacency lists.

We can perform the intersection of the adjacency lists of two vertices in $O(n)$ time. Since we need to consider $n(n-1)/2$ pairs of vertices, the running time of this square-detection algorithm will be $O(n^3)$.

Grading scheme: 2 points for attempt + 3 points for observation + 3 points for running time analysis.

3. You are given a strongly connected directed graph $G = (V, E)$ with positive edge weights along with a particular node $v_0 \in V$. Give an efficient algorithm for finding shortest paths between all pairs of nodes, with the one restriction that these paths must all pass through v_0.
(**8 points**)
Solution:
Consider a shortest path P from u to v passing through v_0. Then, we must have that $d(u, v) = d(u, v_0) + d(v_0, v)$.

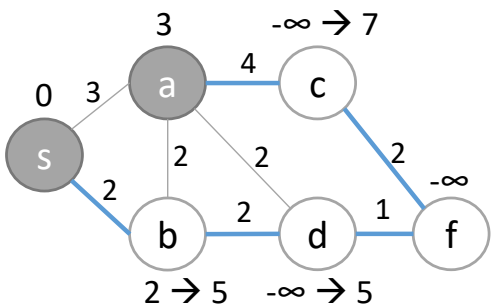
We can determine $d(v_0, v)$ for all vertices $v \in V$ by executing Dijkstra's algorithm once with v_0 as the source vertex. Because the graph is strongly connected, all vertices will get finite distance labels.

Consider a graph G^R with edge directions reversed. This graph is also strongly connected. Further, a shortest path from a vertex u to v_0 manifests as a shortest path from v_0 to u in the reversed graph. Thus, we can execute Dijkstra's algorithm in the reversed graph with v_0 as the source vertex to get path lengths $d(u, v_0)$ in the original graph.

The running time is $O(|V|^2)$, since we are executing Dijkstra's algorithm only twice. The size of the distance matrix is also $O(|V|^2)$.

Grading scheme: 2 points for distance observation + 2 points for Dijkstra's from v_0 + 3 points for reversed graph construction and executing Dijkstra's algorithm + 1 point on a comment why this algorithm is efficient (for instance, by giving running time).

4. Can we modify Dijkstra's algorithm to solve the single-source longest path problem by changing minimum to maximum? If so, then prove that the algorithm is correct. If not, provide a counterexample.
(**7 points**)
Solution:
No, we cannot modify Dijkstra's algorithm to solve the single-source *longest* path problem, even with seemingly reasonable changes to initialization and the priority queue. Dijkstra's algorithm (for SSSP) maintains the invariant that once a vertex is settled, its distance label is no longer updated. The greedy choice in Dijkstra's algorithm is made by picking the unsettled vertex with the smallest distance label at every iteration, and then relaxing edges out of this vertex. Once this vertex is processed, it is considered settled. Here is a example for longest paths where this property does not hold true:



In the above graph, let us assume s is the source, and initialize all distance labels to $-\infty$. After s is settled, a has the next highest distance label, and its adjacencies are relaxed. a will be then incorrectly marked as settled (if we apply Dijkstra's algorithm), but the longest (simple) path from s to a is through b, d, f, and c. Note that for a graph with positive weight cycles, the single source longest path problem is not well-defined.

Grading scheme: 1 point for No + 3 points for a brief explanation why not + 3 points for an example and steps of Dijkstra's algorithm.

5. Argue that in a breadth-first search, the distance label assigned to a vertex v is independent of the order in which the vertices appear in each adjacency list. Also, show using an example that the breadth-first tree computed by BFS can depend on the ordering of vertices in the adjacency list representation of the graph.
(**7 points**)
Solution:
We can have two cases: v is reachable from the source, or v is unreachable. If v is unreachable, the distance label stays at ∞. If v is reachable, then it must have a predecessor u in the BFS tree, and $d(v) = d(u) + 1$. Let us assume the predecessor changes from one execution to another, depending on the order of the vertices. The new predecessor must also be at the same distance from the root in the BFS tree (source), as the previous one. This follows from recursively applying the algorithm and going all the way to the root.

The second part is straightforward: any valid example showing two alternate adjacency list representations and valid BFS trees will do.

Grading scheme: 4 points for first part (with partial credit as appropriate, depending on clarity of solution) + 3 points for second part (any valid example with clearly-marked BFS trees).