1.

a) We express the arbitrage problem in terms of a graph with weights on the edges. Each currency is represented by a vertex in the graph. Since we can convert from any currency to any other, our graph contains all possible edges between two currencies. We want to find a profitable conversion sequence by some how assigning weights to the edges so that a profitable conversion sequence shows up as a negative-weight cycle.

The length of a shortest path in a graph is expressed as the sum of the edge weights in the path, while the rate of profit on a cycle in our graph is expressed as a product of conversion rates. But we know that

$$R[i_1, i_2]R[i_2, i_3] \cdots R[i_{k-1}, i_k] > 1 \text{ iff } \lg R[i_1, i_2] + \lg R[i_2, i_3] + \ldots + \lg R[i_{k-1}, i_k] > \lg 1 = 0$$

In other words, if we assign a weight of $\lg R[i, j]$ to each edge, a profitable cycle is a *positive-weight cycle*. If instead we assign a weight $w(i, j) = -\lg R[i, j]$ to each edge (i, j), a profitable cycle appears as a negative-weight cycle in the graph. To find out if the graph contains a negative-weight cycle, we run the Bellman-Ford algorithm, and then test each edge (i, j) to see if $d(j) > d(i) + w(i, j)$. If this is true for any edge, then we know the graph contains a negative-weight cycle.

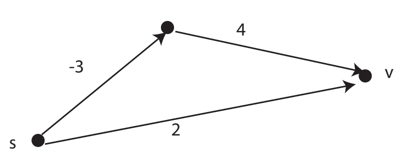
b) To actually print out the sequence of vertices in some negative-weight cycle, we need to store extra information with the shortest-paths, just as we did in some of the dynamic programming algorithms. We keep an array of *predecessor* pointers $p(i)$, along with the path-weight matrix $d(i)$. Every $p(i)$ is initialized to NULL. When running the Bellman-Ford algorithm, each time we reduce the path length to vertex v_j by setting $d(j) \leftarrow d(i) + w(i, j)$, we set $p(j) \leftarrow i$, to indicate that the shortest path found so far to v_j ends with edge (i, j). The predecessor pointers define a graph on the vertices.

Claim: After we have run the Bellman-Ford algorithm, any cycle in the graph defined by the predecessor poitners is a negative-weight cycle.
Proof: Consider the last edge (i, j) in the cycle for which $p(j)$ was set by RELAXing edge (i, j). Let (j, k) be the next edge in the cycle. Since the value $d(j)$ was reduced since $d(k)$ was set, we could reduce $d(k)$by setting it equal to $d(j) + w(j, k)$. Similarly we can reduce the $d()$ value for every vertex in the cycle, finally reducing $d(i)$ and demonstrating that this is a negative-weight cycle.

To find a cycle in the graph of predecessor pointers, we can use depth-first search.

2.

No, a shortest path with respect to W' is not necessarily a shortest path with respect to W. For instance, in the graph below, the shortest path with respect to W has length one, while the shortest path with respect to W' is a different path and has length five.



3.

We can find a maximum-weight independent set using dynamic programming. We consider the subproblems of finding a maximum-weight independent set in graphs $\{(v_1, v_2), (v_2, v_3), \ldots, (v_{k-1}, v_k)\}$. Call the solution to such a problem $m(k)$. We have

$$\begin{aligned} m(0) &= 0 \\ m(1) &= \max\{w_1, 0\} \end{aligned}$$

since just taking v_1 gives a maximum independent set if the weight of v_1 is positive. For any k, we can express $m(k)$ recursively as

$$m(k) = \max\{m(k-1), m(k-2) + w_k\}$$

Notice that if $w_k < 0$, then $m(k) = m(k-1)$. Using this recursive formulation, we can fill in the values of $m(0), \ldots, m(n)$ in $O(n)$ time.

To actually produce the independent set, we include a value $x(k)$ at each k, indicating whether v_k is used in the solution for $m(k)$. If we find that $m(k+2) + w_k > m(k-1)$, we set $x(k)$ =TRUE, and otherwise we set $x(k)$ to be FALSE. To reconstruct the independent set, we call the following recursive procedure with parameter n:

OutputSet(k) If $k = 0$ return
If $x(k) =$ TRUE
    print v_k
    OutputSet($k-2$)
else OutputSet($k-1$)