

We define the graph $G = (V, E)$ for all three subproblems as follows: create n vertices, one for each currency, and there are N^2 directed edges such that the edge $v_i v_j = -\log(Exch[i][j])$ and $v_j v_i = -\log(Exch[j][i])$.

We see that taking the negative log of the exchange rates means we add rather than multiply the edge weights to find the amount of money earned. Let A be the initial amount of currency, A' be the amount after exchanging and $R_1 \dots R_k$ the exchange rates.

$$\begin{aligned} A' &= A_0 * \prod_{i=1}^k R_k \\ -\log(A') &= -\log(A_0 * \prod_{i=1}^k R_k) \\ -\log(A') &= -\log(A_0) + \sum_{i=1}^k -\log(R_k) \end{aligned}$$

Each edge is one element of the summation. Note that taking the negative log of an exchange rate >1 will result in a negative number, and <1 will result in a positive number.

3.a

Describe an algorithm that returns an array $MaxAmt[1..n]$, where $MaxAmt[i]$ 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: If we start an amount of currency A , and we trade to get more currency (i.e an exchange rate >1), this means we traverse a negative-length edge in our graph G . Thus, the problem of finding the maximum amount of currency you can obtain starting with one unit of currency i is equivalent to the single source shortest paths problem in G starting at v_i . From the equation above, we see that $-\log(A') = -\log(A) + \sum_{i=1}^k -\log(R_k)$. The last node in the shortest path is the node at which we have the most currency.

We start the path-finding problem with $-\log(A)$ units. Each edge we add the neative log of the exchange rate at that edge. Since we have negative edge lengths, we use the Bellman-Ford algorithm which runs in $O(VE)$ time. There are no arbitrage cycles which means there are no negative length cycles (see part b).

To get the array $MaxAmt[1..n]$, we solve the shortest path problem starting at index 1, then get the distance to each node. See Algorithm 1 bellow.

Time Complexity Constructing the graph takes $O(V + E) = O(N^2)$ time. Bellman Ford runs in $O(VE) = O(N^3)$ time, which dominates the total runtime of $O(N^3)$.

Algorithm 1 MaxAmt[1...n]

Run Bellman Ford SSSP on G from v_1 with initial value of $-\log(1)$.

for $i = 1$ to $i = n$ **do**

$L \leftarrow$ distance from v_1 to v_n from the Bellman-Ford above

$MaxAmt[i] = 10^{-L}$

end for

Space Complexity Space complexity is dominated by the exchange rates matrix of size $O(N^2)$ used to construct the graph.

■

3.b

Describe an algorithm to determine whether the given matrix of currency exchange rates creates an arbitrage cycle.

Solution: We will construct the same graph G as used in part (a).

We recognize that an arbitrage cycle is equivalent to the existence of a negative length cycle in G . We show this as follows: Suppose we start with A units of currency i . An arbitrage cycle is a cycle of exchange rates $R_1 \dots R_k$ ending back at currency i such that $A * \prod_{i=1}^k R_i > A$. We must end at the same currency we started with. In our graph G , multiplying by a positive exchange rate is equivalent to adding a negative value. A series of rates $R_1 * R_2 * \dots * R_k > 1$ that defines an arbitrage cycle, when we take the negative log of it, is a sequence of negative values $-\log(R_1) - \log(R_2) - \dots - \log(R_k) < 0$,

To detect arbitrage cycle, we simply run Bellman Ford on our graph G . If it detects a negative cycle, there is an arbitrage cycle; if it does not detect a negative cycle there is no arbitrage.

Time Complexity The runtime is dominated by the Bellman Ford runtime, which is $O(VE) = O(V^3)$.

Space Complexity The space is dominated by the graph, which requires an adjacency matrix of size $O(N^2)$.

■

3.c

Modify your algorithm from part (b) to actually return an arbitrage cycle, if it exists.

Solution: We will construct the same graph G as used in part (a).

We run Bellman Ford on our graph G , as in part (b), to detect an arbitrage cycle. However, we modify the cycle detection process to walk through the cycle and return the nodes in it. See Algorithm 2 below:

This algorithm modifies the negative cycle detection routine in the Bellman Ford algorithm. Before we run this routine, we must run the $O(VE)$ path distance routine. Since this part is unchanged from the standard Bellman-Ford algorithm, we do not reprint it here. Note that $d[v]$ is the Bellman-Ford distance at node v , and $p[v]$ is the parent node of the node v . Both of these are part of the standard implementation of Bellman-Ford.

Algorithm 2 Bellman Ford Negative Cycle Modification

```
cycle is a vector of vertices
for each edge  $e = (u,v)$  in  $G$  do
  if  $\text{dist}[u] + \text{weight}(e) < \text{dist}[v]$  then
    Negative cycle detected containing node  $v$ 
     $\text{curr} \leftarrow p[v]$ 
    while  $\text{curr} \neq v$  do
       $\text{cycle.push}(\text{curr})$ 
       $\text{cycle} \leftarrow p[\text{curr}]$ 
    end while
  end if
end for
 $\text{cycle.reverse}()$ 
return  $\text{cycle}$ 
```

Time Complexity The modified cycle detection routine runs in $O(VE)$ time. This is the same as the runtime for the standard Bellman-Ford routine, so the total runtime is $O(VE)$.

Space Complexity The space is dominated by the graph, which requires an adjacency matrix of size $O(N^2)$.

