1. First, let's note that we have a risk free exchange $r_{i_1,i_2} \cdot r_{i_2,i_3} \cdot r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$ then if we take the log of each term and turn the product in to a sum we can see that $\log(r_{i_1,i_2}) + \log(r_{i_2,i_3}) + \log(r_{i_{k-1},i_k}) + \log(r_{i_k,i_1}) > 0$. Therefore if a risk free exchange existed and we took the negative log of each term in that sequence the sum would be negative. This assumes that all the exchange rates are greater than 0.

   Construct a graph where every vertex $c_i \in C$ is equivalent to one of these currencies and every edge between two vertices (c_i, c_j) is the negative logarithm of $r_{i,j}$, the exchange from currency i to currency j. Let C be the set of currencies (vertices) and let $|R|$ be the set of exchange rates (edges). Having a risk free exchange in this graph is equivalent to having a cycle in which the sum of the edges in the cycle is negative per the argument above. Now run the Bellman-Ford algorithm described in class that detects negative cycles in $O(|V||E|)$ time after perform $|V|-1$ rounds of updates over the edges. Once it terminates do one more round of updates over the edges and distances calculated by the algorithm and if any changes occur during this final round of updates there must've been a negative cycle because if there were no negative cycles, $|V|-1$ rounds would've been enough because the shortest paths can contain no more than $|V|$ - 1 edges unless they form a negative cycle. If a negative cycle is detected, there a risk free exchange. This whole process runs in $O(|C||R|)$ time - constructing the graph take $O(|C|+|R|)$ time and the negative cycle detection takes $O(|C||R|)$.

2. Let the updated edge be (a, b) and have weight $w(a, b)$ and remove (a, b) from T. Let $T' = T - (a, b)$. T', because T was a tree, now is two disconnected components one of which contains a and the other containing b. Let component[] be an array that will keep track if what component a vertex is in. Let DFS_update(v, 0) be a function that performs DFS and updates component[v] and all vertices that come after v with a 0.

   GetNewEdge($V, E, T', (a, b)$):

```
min_edge = (a, b)
min_weight = w(a, b)
for v in V:
        component[v] = 0
component[b] = 1
DFS_update(b,1)
for (u, v) in E:
        if component[u] != component[v] and w(u, v) < min_weight:
                min_weight = w(u, v)
                min_edge = (u, v)
return T' ∪ min_edge
```

   This runs in $O(|V| + |E|)$ time because the Depth First Search runs in $O(|V| + |E|)$ and then the for-loop runs in $O(|E|)$ time so at worst, it's $O(|V| + |E|)$ time. Now, here's why this algorithm will find us the new best tree. When we construct T' we create two disconnected components since T was a tree and in order to construct the new best minimum spanning tree we have to find an edge one of whose vertices is in a's component and the other's in b's component. This edge also needs to have a weight that is less than the updated weight of (a, b). If there is no such edge, then we have to put (a, b) back in the graph and return that tree. Importantly we know that the among the tree that comes off at a is the minimum spanning tree for all of the vertices in a's component and the same holds true for b, because if they weren't T would not have been a minimum spanning tree in the first place. This implies that we need only look at the edges that could connect components a and b to find our new minimum spanning tree. The algorithm basically runs like Kruskal's algorithm but there are only 2 sets (a's set and b's set) and we don't have a list of sorted edges so we have to go through them once to find the smallest edge that can connect a and b.

3. Consider the set $X = \{1, 2, ..., 3^{b+1} - 3\}$ for any $b \geq 1$ and subsets of X of the form $S_1, S_2, ...S_b$ where S_i contains $2 \cdot 3^i$ elements from of X and are all pairwise disjoint. Additionally consider the subsets T_1, T_2, T_3 which are all pairwise disjoint and contain one third of the elements from all the S_i. So you

could think of T_1 as having the first third of the elements in $S_1, S_2, ...$ if we say that the sets have some kind of order. Each T_i contains $3^b - 1$ elements. The greedy algorithm will select S_b first, because it's the largest, $2 \cdot 3^b > 3^b - 1$. Then considering the next sets we have either S_{b-1} or the T_i's. No element in S_{b-1} has been covered yet because all S_i are pairwise disjoint. Each T_i has $3^b - 1 - 2 \cdot 3^{b-1} = 3^{b-1} - 1$ elements that have not been covered. So S_{b-1} is the next set selected. We can see that the greedy algorithm will select the sets $S_b, S_{b-1}, ...S_2, S_1$ in that order and never select T_0, T_1, or T_2. Therefore the greedy algorithm on this set uses b sets for $n = 3^{b+1} - 3$, which is $\Omega(logn)$ whereas the optimal algorithm would just use T_1, T_2, T_3 and cover X with 3 sets. Now here is a proof to show that the greedy algorithm will always draw from the sets of the form S_k and never T_1, T_2, or T_3. We have already shown that at the first stage, the greedy algorithm will pick set S_b. Now assume that for all stages less than k, the greedy algorithm will have chosen a set from the set $\{S_i\}_{i=1}^{b}$. So at stage i, the greedy algorithm will have chosen S_{b-i}. Therefore up until stage $k-1$, we have covered $2 \cdot 3^b + 2 \cdot 3^{b-1} ... 2 \cdot 3^{b-k+1}$ elements which by summing the geometric series can be shown to equal $3^{b+1}(1 - 3^{-k})$ elements in total. Because T_1, T_2 and T_3 contain one-third of the elements in each of the S_i, after $k-1$ stages, $3^b(1 - 3^{-k})$ elements in each one has been covered and so $3^b - 1 - 3^b(1 - 3^{-k}) = 3^{b-k} - 1$ elements remain uncovered in T_1, T_2 and T_3. Therefore at stage k, set S_{b-k} is selected because it is guaranteed to contain $2 \cdot 3^{b-k}$ uncovered elements which is more than any of the remaining sets.

If given another value of k, construct $X = \{1, 2, ...k^{b+1} - k\}$ for any $b \geq 1$ and consider disjoint subsets $S_1, S_2 ... S_b$ where S_i contains $(k-1)k^b$ elements and consider the sets $T_1, T_2 ..., T_k$ that contain $\frac{1}{k}$ elements from each of the S_i and are pairwise disjoint. We can go through the same inductive argument to show that the sets T_i are never chosen by the greedy algorithm. Per the same reasoning, the set cover returned by the greedy algorithm is of size b.

4. Let G be the maximum time for our greedy algorithm for some set of tasks. Per the hints, we know that $r_{max} \leq T_{OPT}$ and that $\frac{1}{2}\sum_i r_i \leq T_{OPT}$ (alternatively that $\sum_i r_i \leq 2T_{OPT}$) where T_{OPT} is the optimal amount of time it would take to finish these jobs on two machines. Let t_l be the time at which one of the two machines begins processing the job that will finish last and will take r_l time to finish. Note that $G = t_l + r_l$. We know that $2t_l + r_l \leq \sum_i r_i \leq 2T_{OPT}$ because the amount of work that has to be done up until t_l is $2t_l$ since all the machines have been busy up until now because if they weren't the final job would have started at a time before t_l and we have at least r_l more work that has to be done and so we can conclude that $2t_l + r_l \leq 2T_{OPT}$. This implies that $t_l \leq (1 - \frac{1}{2})T_{OPT}$.
Using the fact that $G = t_l + r_l$, we see that $G \leq (1 - \frac{1}{2})T_{OPT} + T_{OPT} = \frac{3}{2}T_{OPT}$, which is what we wanted to show. If the sequence of the jobs had lengths 4,4,8 minutes the optimal run time will be 8 minutes but the greedy algorithm will generate a job schedule that will take 12 minutes. Now suppose there are m machines and via a similar argument I'll show that $G \leq (2 - \frac{1}{m})T_{OPT}$.

Again, because $r_{max} \leq T_{OPT}$ and $\frac{1}{m}\sum_i r_i \leq T_{OPT}$ we can conclude that $mt_l + r_l \leq \sum_i r_i \leq mT_{OPT}$ and so $t_l \leq (1 - \frac{1}{m})T_{OPT}$. So $G = t_l + r_l \leq (1 - \frac{1}{m})T_{OPT} + T_{OPT} = (2 - \frac{1}{m})T_{OPT}$. Here's a general sequence of jobs that will take the Greedy algorithm 2 - $\frac{1}{m}$ times the optimal amount of time. The first $m(m-1)$ jobs are of length $\frac{1}{m^2}$ and the last job is length $\frac{1}{m}$. Optimally, this will take $\frac{1}{m}$ time because we place m jobs of $\frac{1}{m^2}$ on $m-1$ machines and then place the job of length $\frac{1}{m}$ on the m^{th} machine. The greedy algorithm will put $m-1$ tasks on the m machines and then the last one will be put on one of the machines and so it will take $\frac{m-1}{m^2} + \frac{1}{m} = \frac{2m-1}{m^2} = \frac{1}{m} \cdot \frac{2m-1}{m} = T_{OPT} \cdot (2 - \frac{1}{m})$ time.

5. (a) Let x and y be our n digit numbers and B be the base x and y are written in.
$x = x_2 B^{2n/3} + x_1 B^{n/3} + x_0$
$y = y_2 B^{2n/3} + y_1 B^{n/3} + y_0$
$xy = (x_2 B^{2n/3} + x_1 B^{n/3} + x_0)(y_2 B^{2n/3} + y_1 B^{n/3} + y_0)$
$= x_2 y_2 B^{4n/3} + (x_2 y_1 + x_1 y_2) B^n + (x_2 y_0 + x_1 y_1 + x_0 y_2) B^{2n/3} + (x_1 y_0 + x_0 y_1) B^{n/3} + x_0 y_0$

$c_1 = x_2 y_2$
$c_2 = x_2 y_1 + x_1 y_2$
$c_3 = x_2 y_0 + x_1 y_1 + x_0 y_2$
$c_4 = x_1 y_0 + x_0 y_1$

$c_5 = x_0 y_0$

So in order to compute these values, we can the following 6 multiplications:
$m_1 = x_0 y_0$
$m_2 = x_1 y_1$
$m_3 = x_2 y_2$
$m_4 = (x_1 + x_2)(y_1 + y_2)$
$m_5 = (x_0 + x_1)(y_0 + y_1)$
$m_6 = (x_0 + x_2)(y_0 + y_2)$

Now using some algebra we can rewrite all of coefficients c_i in terms of the m_i
$c_1 = m_3$
$c_2 = m_4 - m_3 - m_2$
$c_3 = m_6 - m_3 - m_1 + m_2$
$c_4 = m_5 - m_2 - m_1$
$c_5 = m_1$

(b) A recurrence relation that will describe this multiplication algorithm is $T(n) = 6T(\lceil n/3 \rceil) + c \cdot n$ because if there are two n digit numbers, 6 multiplications with numbers roughly $n/3$ long have to be done and the numbers themselves always have to be read and also bitshifted (multiplied by B^i), which takes $c \cdot n$ time. Using the master theorem, we see that $T(n) = \Theta(n^{\log_3(6)}) = \Theta(n^{1.63})$. The algorithm in class that only split the numbers in two parts ran in $\Theta(n^{\log_2(3)}) = \Theta(n^{1.58})$ time and so the algorithm that splits into two parts is the better algorithm for sufficiently large n.

(c) If we only used five multiplications in the algorithm in part a, then the recurrence relation describing it would be $T(n) = 5T(\lceil n/3 \rceil) + c \cdot n$ and so it would run in $\Theta(n^{\log_3(5)}) = \Theta(n^{1.46})$ time which is better than the two part algorithm. So if would get down to only 5 multiplications we'd have a much better algorithm and we should use the 5 multiplication algorithm that splits the numbers into 3 parts.