For all homework problems where you are asked to give an algorithm, you must prove the correctness of your algorithm and establish the best upper bound that you can give for the running time. You should always write a clear informal description of your algorithm in English. You may also write pseudocode if you feel your informal explanation requires more precision and detail. As always, try to make your answers as clear and concise as possible.

For this assignment, I collaborated with Noah Epstein, Fred Kelly, Rodrigo Daboin Sanchez, and all the lovely people of Sunday/Tuesday night office hours. The collaboration was conversation centralized such as talking through the construction of data structures.

1 Currency Exchange Problem

The risk-free currency exchange problem offers a risk-free way to make money. Suppose we have currencies c1, . . . , cn. (For example, c1 might be dollars, c2 rubles, c3 yen, etc.) Some pairs of currencies (not necessarily all of them) can be exchanged. If two currencies ci and cj have an exchange rate ri,j , then you can exchange one unit of ci for ri,j units of cj . Note that if ri,j · rj,i ¿ 1, then you can make money simply by trading units of currency i into units of currency j and back again. (Note that it may not be possible to exchange any specific pair of currencies.) This almost never happens, but occasionally (because the updates for exchange rates do not happen quickly enough) for very short periods of time exchange traders can find a sequence of trades that can make risk-free money. That is, if there is a sequence of currencies ci1, ci2, . . . , cik such that ri1,i2 · ri2,i3 . . . · rik1,ik · rik,i1 > 1, then trading one unit of ci1 into ci2 and trading that into ci3 and so on will yield a profit. Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually describe how to find it, but it is recommended that you try to determine how you would find the exchange.)

We can solve this problem by manipulating Bellman Ford's Negative Cycle Detection. We will manipulate the algorithm to detect when an exchange exists within a graph that will result in profit.

It is given that to currency exchange can be determined using multiplication of the corresponding exchange rate. When all current currency exchange values do not yield profit, there are no patterns of r value multiplications that will continue to raise the initial value of c_1. The existence of a pattern of multiplications is separate from the existence of an optimal exchange of currencies. Thinking back to shortest paths, this is the difference between the existence of a negative cycle and the existence of an optimal shortest path connecting vertices.

For terminology sake, we will refer to a pattern of multiplications that results in the continuous growth of the initial value (the equivalent of a negative cycle in a shortest path) as an **increasing multiplication cycle**.

Because Bellman Ford's algorithm can detect Negative Cycles, we can manipulate the algorithm so that it will detect Increasing Multiplication Cycles.

First, we will put the Currency Exchange System in terms of a Graph G (V, E). The vertices of G represent types of currency c_i for every i value between 0 and n-1. The edges of this graph are equal to the rates of exchange. Because the problem tells us that $c_i = r_{i,j}{}^* \, c_j$, the edge from c_j to c_i or (c_j, c_i) = $r_{i,j}$ for every i and j between 0 and n-1. Thus, G(Currencies, Exchange Rates) can be represented as a Graph G (V, E). G is a reflexive graph and every vertex has an edge value connecting to itself of $r_{i,i} = 1$ for i values

between 0 and n-1.

To manipulate Bellman Ford towards multiplication and increasing multiplication cycle detection, we will take advantage of the properties of logarithms. We input all edge weight values into log functions so that the values stored of an edge weight $r_{i,j}$ is represented as $\log(r_{i,j})$. This utilizes the logarithmic property:

$$\log a + \log b = \log ab$$

$$OR$$

$$\log c_1 + \log r_{2,1} = \log c_2$$

We will manipulate the edges to be inputted to log functions in two places in the Bellman Ford algorithm. Below listed is a numbered description of the algorithm with *** at the steps that involve the log function.

Bool SHORTEST PATH BF G (V, E) :
1) Make an array called Exchange of size V and set each value equal to negative infinity.
2) Topologically sort V
3) ***Set the 0th element of sorted V to have an Exchange value equal to log(1): Exchange[V[0]] = log(1)
4) For each vertex in sorted V, and for each edge (u, v) in the graph, run the UPDATEfunction below.
5) Now, to detect whether there exists an increasing multiplication cycle, for each edge in the graph, we will run the conditional statement of the UPDATE function (step 1). Instead of updating (like Step 2), we will return True. Saying that, yes, you can make a profit on this graph!!
6) If step 5 doesn't return True, then the function will return False. No profit can be made right now based on exchange rates.

UPDATE Function for edge (u, v)
1) ***Check if the value for v stored in Exchange is less than the value of u stored in Exchange plus $\log(r_{u,v})$.
2) ***If step one is True (if the value for v is less), than set the value of v in Exchange to be equal to the value of u stored in Exchange plus $\log(r_{u,v})$. edge (w,v) Every time an exchange results in a larger value, it will update the Exchange Array.

RUNTIME:
Looking at the three lines with ***, we can analyze the differences between this algorithm and Bellman Ford to see if it affects the runtime. 3) from the SHORTEST PATH function sets an array index to equal a different value (that's not 0). This doesn't affect runtime. 1) from the UPDATE function is performing a check very similar (except for $>$ and $<$ swap) to Bellman Ford's UPDATE. 2) from UPDATE is setting a value of an array (this doesn't affect runtime either). Because the differences in this algorithm and Bellman Ford's algorithm do not affect this algorithm's runtime, then just like Bellman Ford's algorithm, this has a runtime of $\mathbf{O}(|V| * |E|)$.

Now that we have the algorithm, we need to prove its correctness. We need to prove that every time the function makes an update, it is a correct update. Bellman Ford has a correct update function, and when this update function runs an extra time on all of the edges, it correctly detects the presence of a negative cycle. Thus, if the manipulated algorithm's UPDATE function is always correct, then this algorithm will correctly detect an increasing multiplication cycle on its extra run of the update function (SHORTEST PATH's step 5).

Proof. We aim to prove the correctness of the UPDATE function. We will use proof by strong induction on the number of edges we have updated.

Base case 0: We have not evaluated the edges of any vertex, or we have evaluated 0 edges of the graph. Because we have not evaluated any edges, we have not run the UPDATE function. The first Vertex of the topologically sorted array's Exchange's value (or Exchange[V[0]]) is equal to log(1) and the values of all other vertices are $-\infty$.

Base case 1: This is the case of the edge 1 of the first vertex of the topologically sorted array. Let's call this edge 1 (a, b) with an exchange rate r_b, a. The evaluation and update based upon edge (a,b) will be correct. Vertex b will have a value stored in Exchange of $-\infty$. Vertex a will have a value stored in Exchange of log(1). Assuming exchange rates cannot be equal to $-\infty$, $-\infty$ will be less than $\log(r_{b,a}) + \log(1)$. Thus, the algorithm must update Exchange[b] with the correct value of $\log(r_{b,a})$.

Inductive Hypothesis: All vertices with edges will update the Exchange array correctly for edges 0-n.

Inductive Step: We are proving that the n+1th edge will update the Exchange array correctly.

There exists an n+1th edge (x, y) with an edge weight of $r_{y,x}$. All values in Exchange are correct because of the Inductive Hypothesis. Evaluating the n+1th edge, we will check to see if the following statement is true:

$$Exchange[y] < Exchange[x] + \log(r_{y,x})$$

Because the values in exchange have been updated correctly and the third term is numerical information derived from the graph itself, this operation must be correct.

Thus, every time the graph updates, the update must be correct. As mentioned previously, Bellman Ford's Negative Cycle Detection is reliant upon the correctness of the UPDATE function. Thus, because this manipulated algorithm's UPDATE function is proven to be correct, then this algorithm correctly detects the presence of an Increasing Multiplication Cycle and solves the problem presented. ■

2 MST with an Increased Edge

You are given a graph G = (V, E) and a minimum spanning tree T. The weight of one of the edges in T is increased. Give a linear time algorithm that determines the new MST.

A piazza post told me that I am "given the graph, a list of edges in the MST, and told edge (u,v) has changed weight." So, let edge (u, v) be the edge with the increased weight d. Let T have a weight of W.

I have outlined an algorithm below that determines the new MST T'. The steps are numbered for clarity purposes.
1) We can remove (u, v) from graph T to create MST subtrees T_1 and T_2.
2) We go through all vertices and associate each one with a value 1 or 2, to indicate which subtree each vertex belongs to.
3) We will create a minimum value A_{min} and set it . We will create an edge A_{edge} and set it to (NULL, NULL).
4) We will go through each edge, check to see if it has a vertex belonging to 1 and and a vertex belonging 2. If it has both 1 and 2, we will compare it's weight value to A_{min}. If it is smaller than min, we will set A_{min} equal to the edge weight and A_{edge} equal to the new smallest edge.
5) Once we have gone through every edge, we will combine T_1 and T_2 using edge A_{edge} to create new MST T'. T' has a weight of W - d +

$$A_{min}$$

Step 2 goes through all vertices and step 4 goes through all edges. The rest of the steps are constant time. Thus, this algorithm has a runtime of O(V + E).

We will prove the correctness of this algorithm with the use of the cut property. First, in step 1, the increased edge (u, v) is removed from otherwise correct T to yield two Subtrees. Because of the properties of MST, we know that every edge of an MST is connects two series of edges that all satisfy the cut property thus makeup two distinct MSTs. Thus, removing (u,v), will yield two MSTs trees, and T_1 and T_2 must be MSTs. If T_1 and T_2 are MSTs, then when we examined all edges connecting the two trees in step 4 and chose the smallest one, we were executing the cut property (choosing the smallest edge of this particular cut set). By executing the cut property across two MSTs and correctly choosing the new smallest edge, we created a new and correct MST, T'.

3 Cover Problem

Give a family of set cover problems where the set to be covered has n elements, the minimum set cover is size k = 3, and the greedy algorithm returns a cover of size (log n). That is, you should give a description of a set cover problem that works for a set of values of n that grows to infinity – you might begin, for example, by saying, "Consider the set X = 1, 2, . . . , 2 b for any b 10, and consider subsets of X of the form...", and finish by saying "We have shown that for the example above, the set cover returned by the greedy algorithm is of size b = (log n)." (Your actual wording may differ substantially, of course, but this is the sort of thing we're looking for.) Explain briefly how to generalize your construction for other (constant) values of k. (You need not give a complete proof of your generalization, but explain the types of changes needed from the case of k = 3.)

Consider the set X = 1, ... n for any $n \geq 2^3$. Consider two types subsets of X, T_1 and T_2. T_1 subsets of X contains 3 subsets. It contains subsets of the form

$$[1 + \frac{(a-1)n}{3}, \frac{a*n}{3}]$$

in which $1 \leq a \leq 3$.

T_2 subsets of X contains subsets of the form

$$\bigcup_{a=1}^{3} [1 + \frac{n}{2^{b-1}} + \frac{(a-1)n}{3}, \frac{n}{3*2^b} + \frac{(a-1)n}{3}]$$

in which $1 \leq b \leq \log n$.

The optimal solution for set cover is equal to 3. There are three sets in T_1 that together fully cover X.

However, the Greedy algorithm will end up only choosing sets from T_2. For clarity sake, let us refer to the three subsets of T_1 as A, B, and C. A is $[1, \frac{n}{3}]$. B is$[\frac{n}{3}+1, \frac{2n}{3}]$. C is $[\frac{2n}{3}+1, \frac{3n}{3}]$. $|A| = |B| = |C| = \frac{n}{3}$

The first set that the Greedy algorithm will choose will be T_2's b = 1 value where that subset is equal to $[1, \frac{n}{6}] \cup [1+\frac{n}{3}, \frac{n}{6}+\frac{n}{3}] \cup [1+\frac{2n}{3}, \frac{n}{6}+\frac{2n}{3}]$. This has a size equal to $\frac{n}{2}$. After this set is chosen, sets A, B, and C, only contain $\frac{n}{3*2^b}$ or $\frac{n}{6}$ elements that we "care" about in order to cover the rest of X.

A pattern of this algorithm is that for every step, T_2 will always contain one half of the remaining (A + B + C) or that (A + B + C) that "we care about".

The second set that the Greedy algorithm will choose will be T_2's b = 2 value, where that subset is equal to $[1 + \frac{n}{6}, \frac{n}{12}] \cup [1 + \frac{n}{6} + \frac{n}{3}, \frac{n}{12} + \frac{n}{3}] \cup [1 + \frac{2n}{3} + \frac{n}{6}, \frac{n}{12} + \frac{2n}{3}]$. This has a size equal to $\frac{n}{4}$. Just like the step before, T_2 contains one half of the remaining (A + B + C). Now, sets A, B, and C only contain $\frac{n}{3*2^b}$ or $\frac{n}{12}$ elements that we are concerned about.
This will continue until T_2 contains a union of subsets that will each contain at most one element of X. Then, the algorithm will arbitrarily choose A, B, C, or a T_2 element. When T_2 has unions of single element subsets, it will be on the $\log n$ step, when b = $\log n$. Thus, the Greedy algorithm will complete on the $\log n$th step.
We can transform this problem into one no longer based on 3, but based on value k with simple replacement in the T_1 subsets and T_2 subsets.

Consider the set X = 1, ... n for any $n \geq 2^k$. Consider again our two types of subsets of X, T_1 and T_2. T_1:

$$[1 + \frac{(a-1)n}{k}, \frac{a*n}{k}]$$

in which $1 \leq a \leq k$.
T_2:

$$\bigcup_{a=1}^{k} [1 + \frac{n}{2^{b-1}} + \frac{(a-1)n}{k}, \frac{n}{k*2^b} + \frac{(a-1)n}{k}]$$

in which $1 \leq b \leq \log n$.

These two equations, just as they did for the example of k=3, populates T_1 with k sets and populates T_2 with sets of modified boundaries.

4 Machine Scheduling Problem

Consider the following scheduling problem: we have two machines, and a set of jobs $j_1, j_2, j_3, ..., j_n$ that we have to process one at a time. To process a job, we place it on a machine. Each job j_i has an associated running time r_i . The load on the machine is the sum of the running times of the jobs placed on it. The goal is to minimize the completion time, which is the maximum load over all machines. Suppose we adopt a greedy algorithm: each job j_i is put on the machine with the minimum load after the first i 1 jobs. (Ties can be broken arbitrarily.) Show that this strategy yields a completion time within a factor of 3/2 of the best possible placement of jobs. (Hint: Think of the best possible placement of jobs. Even for the best placement, the completion time is at least as big as the biggest job, and at least as big as half the sum of the jobs. You may want to use both of these facts.) Give an example where a factor of 3/2 is achieved. Suppose now instead of 2 machines we have m machines. What is the performance of the greedy solution, compared to the optimal, as a function of m? Give a family of examples (that is, one for each m – if they are very similar, it will be easier to write down!) where the factor separating the optimal and the greedy solutions is as large as you can make it.

We will first focus on solving 2 machines before expanding to m.

We will begin by describing the problem using distinguishing variables. We have two machines, M_1 and M_2. The sum of the load time all total jobs is T, or: $T = \sum_{i=1}^{n} r_i$. Each machine, at any point in time with k total jobs scheduled, has a particular **load time** which is equal to $\sum_{i=1}^{k} r_i$. At the end of the Greedy Algorithm, the algorithm will return the maximum of M_1 and M_2's final load times, or $Max(T_1, T_2) = T_G$. We will refer to the Optimal Solution's return time as T_O. We assume that T is greater than 0, and that n is also greater than 0.

We will prove that T_O is at most $\frac{3}{2}$ of T_G by comparing the three scenario's of T_O's lower bound with T_G's upper bound per each scenario.

We are able to generate a general upper bound (or worst case) scenario for T_G by first proving the that the worst case scenario for the Greedy algorithm occurs when the machines M_1 and M_2 are equal before the last job j_n comes around.

To prove that the worst case scenario is involved with the last job, we can use simple logic. If a large job comes along early in the Greedy algorithm's runtime, then the algorithm will put the large job on a machine, and won't load that machine again until the other machine has an equal load time. It can eventually compensate for that earlier, large job, and the load times of all machines will eventually even out. However, if a large job comes on the exact last job, then the Greedy algorithm won't be able to later compensate for this job. One of the machines will have a disproportionately high load time than the other.

Not only does the worst case scenario of the Greedy algorithm involve the last job, but it involves the last job when both machines have an equal load time. Before j_n, $T_{n-1} = T - r_n$. M_1 has a load time equal to $\frac{T-r_n}{2} + x$ and M_2 has a load time $\frac{T-r_n}{2} - x$ in which x is equal to a nonnegative number. Because the two machines are arbitrary, it does not matter whether M_1 or M_2 have + or - x load time. By the Greedy Algorithm, job j_n will go to the machine with a load time less than or equal to the other machine's load time. In this case, j_n must go towards either the least or less than or equal to the total job times than the other machine, in this case M_2. M_2 will then have a load time of $r_n + \frac{T-r_n}{2} - x$. The value of x that will maximize this load time will be 0. Thus, we will focus on the case in which M_1 and M_2 have equal load times just before the last job j_n.

Now that we have the worst case scenario of the Greedy Algorithm lightly mapped out, we can compare all three best case scenario of the Optimal Solution with the corresponding worst case scenario of the Greedy Algorithm. We will prove that the Greedy Algorithm will generate a T_G value that is at most a factor $\frac{3}{2}$ of the Optimal Solution's return value T_O.

It is given in the problem that for the Optimal solution, T_O is at least as big as $\frac{T}{2}$ or r_{max}. There are three cases for the Optimal Solution's T_O best case value:

Case I: $r_{max} = \frac{T}{2} = T_O$
Case II: $r_{max} > \frac{T}{2}$; $r_max = T_O$
Case III: $r_{max} < \frac{T}{2}$; $\frac{T}{2} = T_O$

For these three cases, let us compare the worst case T_G with the best case of T_O and prove that T_G is at most equal to $\frac{3}{2}$ T_0. As previously stated, the worst case of T_G is when all other jobs other than the maximum have been divided equally between M_1 and M_2 before r_{max} (also r_n) comes along as the last job.

Case I: $r_{max} = \frac{T}{2} = T_O$
In this case, all jobs other than the maximum job must sum to yield $\frac{T}{2}$. We will divide this equally between T_1 and T_2 (the loading times of machines M_1 and M_2) and add r_{max} to an arbitrary machine.

$$T_1 = \frac{T}{4}$$
$$T_2 = \frac{T}{4} + \frac{T}{2} = \frac{3T}{4}$$
$$T_G = Max(T_1, T_2) = Max(\frac{T}{4}, \frac{3T}{4}) = \frac{3T}{4}$$

We compare T_G's upper bound with T_O's lower bound.

$$T_G = \frac{3T}{4}$$
$$T_O = \frac{T}{2}$$
$$T_O * \frac{3}{2} = T_G$$

In this case, T_G is equal to $\frac{3}{2}$ of T_O.

Case II: $r_{max} > \frac{T}{2}$; $r_{max} = T_O$
In this case, the maximum job is larger than the sum of all other jobs. Thus, the equation below holds for a y that is a real number, greater than 0:

$$r_{max} = \frac{T}{2} + y$$

Thus, we know that the loadtime of all jobs other than r_{max} sum to:

$$T - (\frac{T}{2} + y) = \frac{T}{2} - y$$

We will divide this expression equally between T_1 and T_2 (the loading times of machines M_1 and M_2) and add r_{max} to an arbitrary machine.

$$T_1 = \frac{T}{4} - \frac{y}{2}$$

$$T_2 = \frac{T}{4} - \frac{y}{2} + \frac{T}{2} + y = \frac{3T}{4} + \frac{y}{2}$$

$$T_G = Max(T_1, T_2) = Max(\frac{T}{4} - \frac{y}{2}, \frac{3T}{4} + \frac{y}{2}) = \frac{3T}{4} + \frac{y}{2}$$

We compare T_G's upper bound with T_O's lower bound.

$$T_O = r_{max} = \frac{T}{2} + y$$

$$T_G = \frac{3T}{4} + \frac{y}{2}$$

$$T_O * \frac{3}{2} = (\frac{T}{2} + y) * \frac{3}{2} = \frac{3T}{4} + \frac{3y}{2}$$

Because y is larger than 0:

$$\frac{3T}{4} + \frac{3y}{2} > \frac{3T}{4} + \frac{y}{2}$$

$$\frac{3y}{2} > \frac{y}{2}$$

$$T_O * \frac{3}{2} > T_G$$

In this case, T_G is smaller than $\frac{3}{2}$ of T_O.

Case III: $r_{max} < \frac{T}{2}$; $\frac{T}{2} = T_O$ In this case, the maximum job is smaller than the sum of all other jobs. The equation below holds for a y that is a real number, greater than 0 and less than $\frac{T}{2}$:

$$r_{max} = \frac{T}{2} - y$$

We know that the loadtime of all jobs other than r_{max} sum to:

$$T - (\frac{T}{2} - y) = \frac{T}{2} + y$$

We will divide this expression equally between T_1 and T_2 (the loading times of machines M_1 and M_2) and add r_{max} to an arbitrary machine.

$$T_1 = \frac{T}{4} + \frac{y}{2}$$

$$T_2 = \frac{T}{4} + \frac{y}{2} + \frac{T}{2} - y = \frac{3T}{4} - \frac{y}{2}$$

$$T_G = Max(T_1, T_2) = Max(\frac{T}{4} + \frac{y}{2}, \frac{3T}{4} - \frac{y}{2})$$

Because $y < \frac{T}{2}$, $\frac{y}{2} < \frac{T}{4}$. Thus, $\frac{T}{4} + \frac{y}{2} < \frac{T}{2}$ and $\frac{3T}{4} - \frac{y}{2} > \frac{T}{2}$. So in this case, $T_G = \frac{3T}{4} - \frac{y}{2}$.

We compare T_G's upper bound with T_O's lower bound.

$$T_O = \frac{T}{2}$$

$$T_G = \frac{3T}{4} - \frac{y}{2}$$

$$T_O * \frac{3}{2} = \frac{T}{2} * \frac{3}{2} = \frac{3T}{4}$$

Because y is larger than 0:

$$\frac{3T}{4} > \frac{3T}{4} - \frac{y}{2}$$

$$0 > -\frac{y}{2}$$

$$T_O * \frac{3}{2} > T_G$$

In this case, T_G is smaller than $\frac{3}{2}$ of T_O.

Thus, we have proved that for all best case scenarios of T_O, the worst case scenario of T_G is at most $\frac{3}{2}$ of T_O.

Applying this to a problem with m machines, we will prove that T_G is at most $\frac{2m-1}{m}$ of T_O. We are able to yield this expression from Case I (this will be explained below).

It is given in the problem of 2 Machines, that for the Optimal solution, T_O is at least as big as $\frac{T}{2}$ or r_{max}. Now, to apply this to m machines, T_O is at least as big as $\frac{T}{m}$ or r_{max}. There are three cases for the Optimal Solution's T_O best case value for m machines:

Case I: $r_{max} = \frac{T}{m} = T_O$
Case II: $r_{max} > \frac{T}{m}$; $r_{max} = T_O$
Case III: $r_{max} < \frac{T}{m}$; $\frac{T}{m} = T_O$

For these three cases, let us again compare the worst case T_G with the best case of T_O, and prove that T_G is at most $\frac{2m-1}{m}$ of T_0. The worst case of T_G is when all other jobs other than the maximum have been divided equally between all machines before r_{max} (also r_n) comes along as the last job.

Case I: $r_{max} = \frac{T}{m} = T_O$
In this case, all jobs other than the maximum job must sum to yield $\frac{T}{m}$. We will divide the remaining load time equally between all machines and add r_{max} to an arbitrary machine T_K. We know that the loadtime of all jobs other than r_{max} sum to:

$$T - (\frac{T}{m}) = \frac{T(m-1)}{m}$$

Dividing this up over m machines yields:

$$\frac{T(m-1)}{m^2}$$

$$T_1 = \frac{T(m-1)}{m^2}$$

$$T_k = \frac{T}{m} + \frac{T(m-1)}{m^2} = \frac{T(2m-1)}{m^2}$$

$$T_G = Max(T_1, ...T_k, ...T_n) = Max(\frac{T(m-1)}{m^2}, \frac{T(2m-1)}{m^2}) = \frac{T(2m-1)}{m^2}$$

We compare T_G's upper bound with T_O's lower bound.

$$T_G = \frac{T(2m-1)}{m^2}$$

$$T_O = \frac{T}{m}$$

$$\frac{T}{m} * \frac{(2m-1)}{m} = \frac{T(2m-1)}{m^2}$$

$$T_O * \frac{(2m-1)}{m} = T_G$$

In this case, T_G is equal to $\frac{(2m-1)}{m}$ of T_O.

In the case of two machines, Case I was the only case in which T_G was equal to $T_O * \frac{(2m-1)}{m}$ (or $\frac{3}{2}$). If T_G is equal to or less than $T_O * \frac{(2m-1)}{m}$ for the next two cases, then we know that the maximum constant factor of $\frac{(2m-1)}{m}$ holds.

Case II: $r_{max} > \frac{T}{m}$; $r_{max} = T_O$
In this case, the maximum job is larger than the sum of all other jobs. Thus, the equation below holds for a y that is a real number, greater than 0:

$$r_{max} = \frac{T}{m} + y$$

We will divide the remaining load time equally between all machines and add r_{max} to an arbitrary machine T_K. We know that the loadtime of all jobs other than r_{max} sum to:

$$T - (\frac{T}{m} + y) = \frac{T(m-1)}{m} - y$$

Dividing this over m machines yields:

$$\frac{T(m-1)}{m^2} - \frac{y}{m}$$

$$T_1 = \frac{T(m-1)}{m^2} - \frac{y}{m}$$

$$T_k = \frac{T(m-1)}{m^2} - \frac{y}{m} + \frac{T}{m} + y = \frac{T(2m-1)}{m^2} + \frac{(m-1)y}{m}$$

$$T_G = Max(T_1, ...T_k, ...T_n) = Max(\frac{T(m-1)}{m^2} - \frac{y}{m}, \frac{T(2m-1)}{m^2} + \frac{(m-1)y}{m}) = \frac{T(2m-1)}{m^2} + \frac{(m-1)y}{m}$$

We compare T_G's upper bound with T_O's lower bound.

$$T_O = r_{max} = \frac{T}{m} + y$$

$$T_G = \frac{T(2m-1)}{m^2} + \frac{(m-1)y}{m}$$

$$T_O * \frac{(2m-1)y}{m} = (\frac{T}{m} + y) * \frac{(2m-1)y}{m} = \frac{(2m-1)T}{m} + \frac{(2m-1)y}{m}$$

Because y is larger than 0:

$$\frac{T(2m-1)}{m} + \frac{(2m-1)y}{m} > \frac{T(2m-1)}{m^2} + \frac{(m-1)y}{m}$$

$$\frac{(2m-1)y}{m} > \frac{(m-1)y}{m}$$

$$T_O * \frac{(2m-1)}{m} > T_G$$

In this case, T_G is smaller than $\frac{(2m-1)}{m}$ of T_G.

In all three cases of T_G's upper bound compared to T_O's lower bound for m machines, a maximum differing factor (of T_O) is equal to $\frac{(2m-1)}{m}$. Thus, we have proven that the Greedy solution, compared to the optimal, as a function of m has a performance of $\frac{(2m-1)}{m}$.

5 Problem 5

Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with n/3 digits.
(a) Design and explain such an algorithm, similar to the integer multiplication algorithm presented in class. Your algorithm should describe how to multiply the two integers using only six multiplications on the smaller parts (instead of the straightforward nine).

Let's design an algorithm very similar to the integer multiplication algorithm for showed in class, except we will split our numbers into three parts. After we have performed the split, we will be able to obtain all of our terms using only 6 multiplications instead of the regular 9.
We are trying to multiply N_1 by N_2. Let us say that

$$N_1 = L_1 * 10^{\frac{2n}{3}} + M_1 * 10^{\frac{n}{3}} + R_1$$

$$N_2 = L_2 * 10^{\frac{2n}{3}} + M_2 * 10^{\frac{n}{3}} + R_2$$

So far, this looks very similar to the split we performed in class. The two differences in this equation and the class's intial equation is that the number is split into three parts instead of two and that each power of 10 has an exponent in the form of thirds rather than halves.
Now, we can perform standard multiplication using these terms (this will give us 9 multiplications, as the splitting into two groups yields 4. However, we will be able to replace these 9 multiplications using only 6 in the following steps).

$$N_1 * N_2 = R_2 * R_1 + R_2 * M_1 * 10^{\frac{n}{3}} + R_2 * L_1 * 10^{\frac{2n}{3}} + M_2 * R_1 * 10^{\frac{n}{3}} + M_2 * M_1 * 10^{\frac{2n}{3}}$$
$$+M_2 * L_1 * 10^{\frac{3n}{3}} + L_2 * R_1 * 10^{\frac{2n}{3}} + L_2 * M_1 * 10^{\frac{3n}{3}} + L_2 * L_1 * 10^{\frac{4n}{3}}$$

Condensing this multiplication yields:

$$N_1 * N_2 = (L_2 * L_1)10^{\frac{4n}{3}} + ((M_2 * L1) + (L_2 * M_1))10^{\frac{3n}{3}} + ((R_2 * L_1) + (M_2 * M_1) + (L_2 * R_1))10^{\frac{2n}{3}}$$

$$+((M_2 * R_1) + (R_2 * M_1))10^{\frac{n}{3}} + R_2 * R_1$$

We are trying to get 5 terms from this expression above. Labelling these $T_1 - T_5$ yields:

$$N_1 * N_2 = T_1 * 10^{\frac{4n}{3}} + T_2 * 10^{\frac{3n}{3}} + T_3 * 10^{\frac{2n}{3}} + T_4 * 10^{\frac{n}{3}} + T_5$$

$$T_1 = L_1 * L_2$$

$$T_2 = (M_2 * L1) + (L_2 * M_1)$$

$$T_3 = (R_2 * L_1) + (M_2 * M_1) + (L_2 * R_1)$$

$$T_4 = (M_2 * R_1) + (R_2 * M_1)$$

$$T_5 = R_2 * R_1$$

From here, we will define our 6 multiplications that we can use solely to represent terms $T_1 - T_5$:

$$Mult_1 = L_1 * L_2$$

$$Mult_2 = M_1 * M_2$$

$$Mult_3 = R_1 * R_2$$

$Mult_4 = (L_1+M_1)(L2+M_2) = L_1*L_2+L_1*M_2+L_2*M_1+M_1*M_2 = L_1*M_2+L2*M_1+Mult_1+Mult_2$

$Mult_5 = (L_1+R_1)(L_2+R_2) = L_1*L_2+L_1*R_2+R_1*L_2+R_1*R_2 = L_1*R_2+R_1*L_2+Mult_1+Mult_3$

$Mult_6 = (M_1+R_1)(M_2+R_2) = M_1*M_2+M_1*R_2+R_1*M_2+R_1*R_2 = M_1*R_2+R_1*M_2+Mult_2+Mult_3$

Building $T_1 - T_5$ using $Mult_1 - Mult_6$:

$$T_1 = Mult_1$$
$$T_2 = Mult_4 - Mult_1 - Mult_2$$
$$T_3 = Mult_5 - Mult_1 - Mult_3 + Mult_2$$
$$T_4 = Mult_6 - Mult_2 - Mult_3$$
$$T_5 = Mult_3$$

Thus, we are able to represent our final equation using only these six multiplication:

$$N_1 * N_2 = Mult_1 * 10^{\frac{4n}{3}} + (Mult_4 - Mult_1 - Mult_2) * 10^{\frac{3n}{3}} + (Mult_5 - Mult_1 - Mult_3 + Mult_2) * 10^{\frac{2n}{3}}$$
$$+(Mult_6 - Mult_2 - Mult_3) * 10^{\frac{n}{3}} + Mult_3$$

which can also be written as:

$$N_1 * N_2 = (L_1 * L_2) * 10^{\frac{4n}{3}} + (((L_1 + M_1)(L2 + M_2)) - (L_1 * L_2) - (M_1 * M_2)) * 10^{\frac{3n}{3}}$$
$$+(((L_1 + R_1)(L_2 + R_2)) - (L_1 * L_2) - (R_1 * R_2) + Mult_2) * 10^{\frac{2n}{3}}$$
$$+(((M_1 + R_1)(M_2 + R_2)) - (M_1 * M_2) - (R_1 * R_2)) * 10^{\frac{n}{3}} + (R_1 * R_2)$$

(b) Determine the asymptotic running time of your algorithm. Would you rather split it into two parts or three parts?

The Recurrence equation of the algorithm split into two parts (given in class) is:

$$T(n) = 3T(\frac{n}{2}) + O(n)$$

This is because there were three multiplications performed, the numbers were always split into halves, and bit shifts as well as addition/subtraction are linear.

Using the master theorem, the asymptotic running time of the algorithm split into two parts is $n^{\log_2 3} \approx n^{1.585}$.

Applying exactly the same logic to this problem's algorithm, this asymptotic runtime is:

$$T(n) = 6T(\frac{n}{3}) + O(n)$$

The asymptotic running time is $n^{\log_3 6} \approx n^{2.862}$.

$n^{1.585} < n^{2.862}$ Thus, it would be better to split the multiplication into two parts in order to yield a better runtime.

(c) Suppose you could use only five multiplications instead of six. Then determine the asymptotic running time of such an algorithm. In this case, would you rather split it into two parts or three parts?

The Recurrence relation of the algorithm would be:

$$T(n) = 5T(\frac{n}{3}) + O(n)$$

The asymptotic runtime of the equation would be: $n^{\log_3 5} \approx n^{1.464}$

$n^{1.585} > n^{1.464}$ Because this is a smaller exponent than the two-part multiplication, splitting the numbers into three parts and performing 5 multiplications would be the optimal algorithm.