CS 124 Problem Set 3
HARVARD ID: 10939860

1. The *risk-free currency exchange problem* offers a risk-free way to make money. Suppose we have currencies $c_1, ..., c_n$. (For example, c_1 might be dollars, c_2 rubles, c_3 yen, etc.) Some pairs of currencies (not necessarily all of them) can be exchanged. If two currencies c_i and c_j have an exchange rate of $r_{i,j}$, then you can exchange one unit of c_i for $r_{i,j}$ units of c_j. Note that if $r_{i,j} \cdot r_{j,i} > 1$, then you can make money simply by trading nits of currency i into units of currency j and back again. 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 s sequence of currencies $c_{i_1}, c_{i_2}, ..., c_{i_k}$, such that $r_{i_1,i_2} \cdot r_{i_2,i_3} ... r_{i_{k-1},i_k} \cdot r_{i_k,i_1} > 1$, then trading one unit of c_{i_1} into c_{i_2} and trading that into c_{i_3} and so on will yield a profit. Design an efficient algorithm to detect if a risk-free currency exchange exists. (You need not actually find it.)

   Solution.
   To solve this problem, we can use the Bellman-Ford algorithm. First, we construct a directed graph from the given information, letting $c_{i_1}, c_{i_2}, ..., c_k$ be the vertices of graph G, and $w_{i_1,i_2}, w_{i_2,i_3}, ..., w_{i_k,i_1}$ be the edges, where $w_{i,j} = -log(r_{i,j})$. Taking the negative logarithm of the exchange rates gives a convenient setup for the Bellman-Ford algorithm. Because the log of any x such that $0 < x < 1$ is a negative value, we can negate the log value of each exchange rate, and if a negative cycle exists (originally a positive cycle), then there must be a risk-free currency exchange. We know that if there is a cycle under these conditions, then there is a way to get back to a currency from itself such that its rates multiply to a value greater than 1.

   The Bellman-Ford algorithm can determine whether or not G has a negative cycle. We run the algorithm, and instead of looping $|V| - 1$ times as in the typical algorithm, we loop one more time for a total of $|V|$ times, and if any updates are made to the *dist* array in the algorithm on the last iteration, there must be a negative cycle, because a shorter path can be reached. If no updates are made, then no negative cycle exists, and there is no existing risk-free currency exchange. This algorithm works because the Bellman-Ford algorithm checks every vertex and every edge, ensuring that if a directed negative cycle exists, meaning that there must be a risk-free currency exchange for the above stated reasons.

   As for the runtime, we must consider the time of making the graph, and the time of running Bellman-Ford. The time to make the graph will be done in linear time, due to a constant amount of work for both the vertices and the edges, and the time of Bellman-Ford is $O(|V||E|)$, as we can see from the double for-loop in the algorithm. Thus, the total time is dominated by the Bellman-Ford algorithm, and this algorithm's total runtime is $O(|V||E|)$.

2. 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.

   Solution.

   We assume that we know which edge e in T was increased at the start of the problem. Then, delete e from T, and we assume that edge connected two sub-graphs, so we now consider the two sub-graphs. These two sub-graphs must also be MSTs, which we can prove by contradiction:

Assume that the two graphs, call them T_1 and T_2, are not MSTs. Then, we know that there must be some edge within T_1, say, or across the two graphs from T_1 to T_2 that is not the shortest weight to travel to that specific vertex. If the edge is in T_1 but not in the subgraph's actual MST, then it would not be in the original MST of the entire graph before deleting e. But we know that the original graph was an MST, so we reach a contradiction, and both T_1 and T_2 must be MSTs by the same logic.

From this fact, we can now connect the two sub-graphs by the shortest edge that travels between T_1 and T_2. In order to do this, simply search through every edge in the graph, determine whether or not it goes from T_1 to T_2 or T_2 to T_1, and then return the shortest length edge that satisfies this condition. We can see why this process works because of the cut property of MSTs. For the sake of convenience, make the "cut" be the set of vertices in T_1. We know that the shortest edge that crosses the cut, T_1, to the set of vertices not in the cut, T_2, must be in the MST of the graph. So, use that found edge as the final edge of the new MST. Connecting these two subgraphs with the new edge will create the new MST.

Now, we consider the runtime. The only work done in this algorithm is in searching through the set of edges to find the shortest edge between the two-subgraphs, and connecting them, which happens in $O(|E|)$ time because it involves checking every edge once in the worst case. This is linear time.

3. 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 $\Omega(\log n)$. 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 $b \geq 10$, 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 $b = \Omega(\log n)$ (Your actual wording may differ substantially, of course, but this is the sort of thing were 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 $k = 3$.)

Solution.

Consider the set $X = \{1, 2, ..., 3^b\}$ for any $b \geq 10$, and consider subsets of 2 forms:

In the first form, let the first subset be $\{3^{b-1}, ..., 3^b\}$, the next subset be $\{3^{b-2}, ..., 3^{b-1}\}$, and so on until $\{3^{b-b}\} = \{1\}$. The first subset will be three times the size of the second subset, the second subset will be three times the size of the third, and so on. This ensures that this organization will cover all elements of X and result in $log(n)$ subsets.

In the second form, we will split the original set into three subsets using the modulo function to modulo every member in X by 3. The first subset will be every element of X that outputs 0 when every element in X is modulo-ed by 3. The second subset will be all original members that output 1 after the modulo, and the third subset will be all original members that output 2 after the modulo.

With these two forms of subsets, the minimum set cover is of size 3, and the greedy algorithm will always take a set cover of size $\Omega(log(n))$. This is because the greedy algorithm will always first take the largest subset available that covers part of X, so it will take the first subset from the first described form of subsets ($\{3^{b-1}, ..., 3^b\}$), as it is much bigger than any of the three subsets in the

second form. After taking that subset, it will take the next largest, which is $\{3^{b-2}, ..., 3^{b-1}\}$. It will continue to $\{3^{b-3}, ..., 3^{b-2}\}$, as this is the majority. It will continue all the way down this chain until it takes $\{1\}$, the final subset. This algorithm, due to taking the locally best option at every step, will take a complete $\Omega(log(n))$-sized set cover. Though, this method is inefficient, and the second form of subsets is much better for completing a set cover. With the three subsets all roughly the same size, they will stay consistent no matter the value of b, and will always contain every member of X. Therefore, it creates a minimum set cover of size 3.

To generalize this construction for other values of k, we consider the set $X = \{1, 2, ..., k^b\}$. For the first family of subsets, we simply split the subsets into $\{k^{b-1}, ..., k^b\}$, $\{k^{b-2}, ..., k^{b-1}\}$, and so on. For the second family, we modulo by k instead of by 3. This generalization will always give a minimum set cover of size k and a set cover size of $\Omega(log(n))$ with the greedy algorithm.

4. 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 $\frac{3}{2}$ 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 $\frac{3}{2}$ 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.

Solution.

We will consider the best and worst case scenarios in comparison to each other in minimizing the completion time. Logically, the best case will occur when the two machines have as close to the same total load as possible. As such, if one machine gets a job with a large runtime, successive jobs will consecutively be given to the other machine until the two machines even out, minimizing the difference and thus the total completion time. The greedy algorithm may succeed in this mission sometimes, but will fail if a large job comes at the end, since the greedy algorithm does not consider the order of the jobs, so there may not be a chance to even out the two machines.

In the best case, the total load's running time will be the total sum of running times for all jobs divided by two if the largest job's running time is smaller than the sum of all other jobs, or will be the running time of the largest job's running time if that time is larger than the sum of all other jobs' running times. Let s be the sum of all running times, and b be the biggest running time of any r_i. Then, the best case can be represented as b if $b \geq s - b$ or $\frac{s}{2}$ if $b \leq s - b$. The worst case is the scenario where the greedy algorithm takes b as the final job and adds it to one of the machines after splitting the other jobs among the machines evenly. This can be represented as $b + \frac{s-b}{2} = \frac{b}{2} + \frac{s}{2}$.

To prove this, algebraically reduce the equations for the best case to show the factor the strategy is within of the best possible placement of jobs:

For the case of $b \geq s - b$:
$2b \geq s$
$b \geq \frac{s}{2}$
$\frac{3b}{2} - \frac{b}{2} \geq \frac{s}{2}$
$\frac{3b}{2} \geq \frac{s}{2} + \frac{b}{2}$
For the case of $b \leq s - b$:
$s \geq 2b$
$\frac{s}{2} \geq b$
$\frac{s}{4} \geq \frac{b}{2}$
$\frac{3s}{4} - \frac{2s}{4} \geq \frac{b}{2}$
$\frac{3s}{4} \geq \frac{b}{2} + \frac{s}{2}$
$\frac{3}{2} * \frac{s}{2} \geq \frac{s}{2} + \frac{b}{2}$
Thus, this algebra shows that in both cases of the best case, the greedy algorithm yields a completion time within a factor of $\frac{3}{2}$ of the best possible completion time.

For an example where a factor of $\frac{3}{2}$ is achieved, consider the case where $b = \frac{s}{2}$. Then, plugging $\frac{s}{2}$ into our formula for the worst case running time, we get $\frac{s}{4} + \frac{s}{2} = \frac{3s}{4} = \frac{3}{2} * \frac{s}{2}$. Thus, this example gives us a factor of exactly $\frac{3}{2}$.

Lastly, we generalize to m machines, which uses very similar algebra, but divides everything among m machines instead of 2. We see that the best case with the greedy algorithm is b if $b \geq \frac{s}{m}$ and $\frac{s}{m}$ if $x \leq \frac{s}{m}$, while the worst case is $b + \frac{s-b}{m} = b - \frac{b}{m} + \frac{s}{m}$.

Using the same algebra methodology as above:
For the case of $b \geq \frac{s}{m}$
$2b \geq \frac{s}{m} + b$
$2b - \frac{b}{m} \geq \frac{s}{m} + b - \frac{b}{m}$
$b(2 - \frac{1}{m}) \geq \frac{s}{m} + b - \frac{b}{m}$
For the case of $b \leq \frac{s}{m}$:
$\frac{s}{m} \geq b$
$\frac{2s}{m} \geq b + \frac{s}{m}$
$\frac{2s}{m} - \frac{b}{m} \geq b + \frac{s}{m} - \frac{b}{m}$
$\frac{s}{m}(2 - \frac{b}{s}) \geq b + \frac{s}{m} - \frac{b}{m}$
And continuing the logic of this case:
$mb \leq s$
$m \leq \frac{s}{b}$
$\frac{1}{m} \leq \frac{b}{s}$
$-\frac{1}{m} \geq -\frac{b}{s}$
$2 - \frac{1}{m} \geq 2 - \frac{b}{s}$
Thus, we can conclude that in all cases of the best case, the greedy algorithm yields a completion time within a factor of $2 - \frac{1}{m}$ of the best possible completion time with m machines. The factor between the greedy algorithm's solution and the optimal solution will always be as large as possible when $b \geq \frac{s}{m}$ and $b = r_n$. We can see a family of examples where the factor between the two solutions is as large as you can make it when we have one job of size m, and $m(m-1)$ jobs of size 1. With this setup, we have $s = m^2$ and $b = m$. Plugging into our derived equation for the worst case scenario, we have $m - \frac{m}{m} + \frac{m^2}{m} = 2m - 1$, and from the equation for the best case scenario, we have m. The

ratio between the worst and best case is $\frac{2m-1}{m} = 2 - \frac{1}{m}$, thus showing that this example gives the largest possible factor between the optimal and greedy solutions.

5. Consider an algorithm for integer multiplication of two n-digit numbers where each number is split into three parts, each with $\frac{n}{3}$ 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).

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

   (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?

   (d) Challenge problem—this is optional, and not worth any points. Solving it will simply impress the instructors. Find a way to use only five multiplications on the smaller parts. Can you generalize to when the two initial n-digit numbers are split into k parts, each with $\frac{n}{k}$ digits? Hint: also consider multiplication by a constant, such as 2; note that multiplying by 2 does not count as one of the five multiplications. You may need to use some linear algebra.

Solution.

(a) The setup for this algorithm will be very similar to the integer multiplication algorithm presented in class, using a divide and conquer strategy with slight modifications. Call the first number a, and split it into three parts: x, y, and z. Call the second number b, and split it into three parts: q, r, and s. In both numbers, each of the three segments will have size $\frac{n}{3}$. Thus, we can represent the two numbers as:
$a = x * 10^{2n/3} + y * 10^{n/3} + z$,
$b = q * 10^{2n/3} + r * 10^{n/3} + s$.
Multiplying these two equations together gives the following expression:
$a * b = xq * 10^{4n/3} + (qy + xr) * 10^n + (yr + sx + qz) * 10^{2n/3} + (rz + sy) * 10^{n/3} + sz$.
This equation gives us nine multiplications needed to compute to compute the overall equation: $xq, qy, xr, yr, sx, qz, rz, sy$, and sz. Thus, we have the following recurrence: $T(n) = 9T(n/3) + O(n)$, which by the master theorem gives a bound of $\Theta(n^2)$.

To do better, we can come up with a series of six equations that will give us all nine necessary multiplications, by subtracting and adding certain equations:
$m_1 = xq$
$m_2 = sz$
$m_3 = yr$
$m_4 = (x + y)(q + r) - m_1 - m_3$
$m_5 = (x + z)(s + q) - m_1 - m_2 + m_3$
$m_6 = (y + z)(r + s) - m_2 - m_3$
These six equations will successfully give all the nine multiplications needed to solve the original multiplication: $xq, qy, xr, yr, sx, qz, rz, sy$, and sz, therefore giving a solution to the given problem.

(b) With the improved solution, our new recurrence equation will be $T(n) = 6T(n/3)+O(n)$. Using the master theorem for recurrences, with $a = 6, b = 3, k = 1$, we get that the asymptotic running time of the above algorithm is $\Theta(n^{log_3 6}) = \Theta(n^{1.631})$.

Splitting each number into two parts instead would give the recurrence $T(n) = 3T(n/2)+O(n)$, giving a faster running time of $\Theta(n^{1.585})$, so splitting the number into two parts is better.

(c) If we could split the multiplication into five equations, we'd have the recurrence $T(n) = 5T(n/3) + O(n)$ for splitting each number into three parts, which using the master theorem with $a = 5, b = 3, k = 1$, gives the running time as $\Theta(n^{log_3 5}) = \Theta(n^{1.465})$.

Meanwhile, splitting each number into two parts with five equations gives the recurrence $T(n) = 5T(n/2) + O(n)$, with a solution by master theorem as $\Theta(n^{2.322})$, showing that splitting each number into three segments is a much better in this case.

I collaborated on this problem set with Kate Schember and Sam Bieler.