Charles Bolton
4/11/19

Problem 1
(a): Prove or disprove that the function $\log_k n$ is $O(\lg\ n)$ for any $k > 1$:

We can actually prove that $\log_k n \in \Theta\ (\lg\ n)$. Once we have proven that, it follows somewhat trivially that $\log_k n$ is also $O(\lg\ n)$.

Proof:

1) Let $\mathrm{f}(n) = \log_k n$
2) Let $\mathrm{g}(n) = \lg\ n$

3) $\forall n \geq n_0, \quad \exists\, c_1, c_2 > 0 \mid \ c_1 \mathrm{g}(n) \leq \mathrm{f}(n) \leq c_2\ \mathrm{g}(n)$ (Defn. Θ)

So,

4) $c_1 \log_2 n \leq \log_k n \leq c_2 \log_2 n$

Using the change-of-base logarithm property to get a common term for all three sides, we can obtain the following equality:

$$\log_k n = \frac{\log_2 n}{\log_2 k} = \log_2 n \cdot \frac{1}{\log_2 k}$$

This means that we can divide all sides of the inequality in line 4 by $\log_2 n$, thus obtaining our bounds:

$$c_1 \leq \frac{1}{\log_2 k} \leq c_2.$$

Ignoring the left side of the inequality, we see that if we choose $c_2 \geq \frac{1}{\log_2 k}$ then $\mathrm{g}(n) \in O(\lg n)$. Therefore, choose $\mathrm{c} \geq \log_k 2$ and we have $\log_k n \in O \log_2 n$.

(b): The following recurrence relation solves to $O(n \lg^2 n)$. Prove this by substition:

$$T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$$
$$T(1) = 0$$

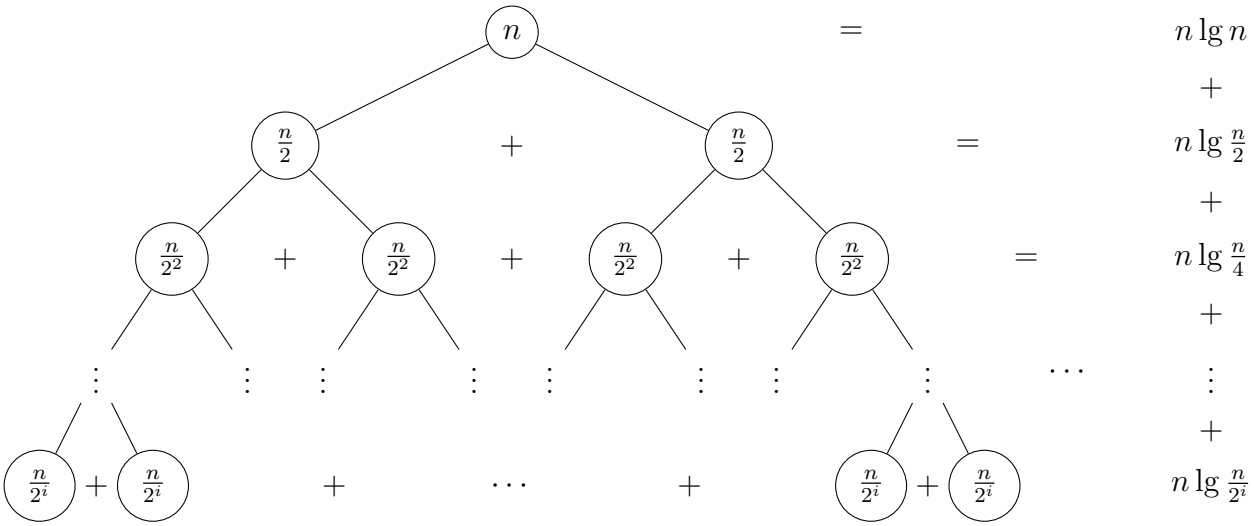
We can first analyze the recurrence mathematically to determine a general case:

$$
\begin{aligned}
T(\frac{n}{2}) &= 2T\left(\frac{n}{4}\right) + \frac{n}{2}\lg\frac{n}{2} \\
\therefore T(n) &= 2\cdot(2T(\frac{n}{4}) + \frac{n}{2}\lg\frac{n}{2}) + n\lg n = 4T(\frac{n}{4}) + n\lg\frac{n}{2} + n\lg n \\
T(\frac{n}{4}) &= 2T\left(\frac{n}{8}\right) + \frac{n}{4}\lg\frac{n}{4} \\
\therefore T(n) &= 4\cdot(2T(\frac{n}{4}) + \frac{n}{4}\lg\frac{n}{4}) + n\lg\frac{n}{2} + n\lg n = 8T(\frac{n}{8}) + n\lg\frac{n}{4} + n\lg\frac{n}{2} + n\lg n
\end{aligned}
$$

We can now see the general formula for T(n):

$$
T(n) = 2^i T(\frac{n}{2^i}) + n\lg n + n\lg\frac{n}{2} + n\lg\frac{n}{4} + n\lg\frac{n}{8} + ... + n\lg\frac{n}{2^{i-1}}
$$

The following recursion tree illustrates the arguments to T (the size of each subproblem) and how many subproblems are computed at each level of recursion:



Latex modified from http://www.texample.net/tikz/examples/merge-sort-recursion-tree/

The right hand column of the graphic shows the costs at each level of the tree.
We know from the base case of this recurrence that

$$
T(1) = 0
$$

Setting our general argument to T equal to our base case and solving for i, we get

$$
T(\frac{n}{2^i}) = T(1)
$$

Therefore, $i = \lg n$ and we know that there are $\lg\ n$ levels in this recursion tree. Furthermore, we know that at each level of the tree there are 2^i nodes. The cost at the bottom level, where $i = \lg n$ is 0, so we do not factor this into the total cost. Therefore, we will add costs from levels $i = 0$ to $i = \lg n - 1$

The following summation defines this recursion:

$$T(n) = \sum_{i=0}^{\lg n - 1} n \lg \frac{n}{2^i}$$

$$= n \sum_{i=0}^{\lg n - 1} \lg \frac{n}{2^i}$$

$$= n \sum_{i=0}^{\lg n - 1} \lg n - \lg 2^i$$

$$= (n \sum_{i=0}^{\lg n - 1} \lg n) - (n \sum_{i=0}^{\lg n - 1} \lg 2^i)$$

$$= (n \sum_{i=0}^{\lg n - 1} \lg n) - (n \sum_{i=0}^{\lg n - 1} i)$$

The left sum simplifies to $n \cdot \lg n \cdot \lg n$ because we are adding $(\lg\ n)\ \lg\ n$ times. The right sum simplifies to

$$n \cdot \frac{(\lg n - 1)(\lg n - 1 + 1)}{2} = \frac{n \cdot \lg n \cdot \lg n - \lg n}{2}.$$

doing the subtraction and tossing out the constants we wind up with

$$T(n) \in O(n \cdot \lg n \cdot \lg n)$$

(c) Suppose that $f(n)$ and $g(n)$ are non-negative functions. Prove or disprove the following: if $f(n) \in O(g(n))$ then $2^{f(n)} \in O(2^{g(n)})$.

We know that if $f(n) \in O(g(n))$ then $f(n) \leq c(g(n))$ where $c > 0$. We might be tempted to think that we could simply raise an arbitrary constant to both functions and the asymptotic complexity would remain the same, much as if we multiplied by a scalar or added/subtracted from both sides. The mistake one might make in making this assumption would be:

$$2^{f(n)} = 2^{c \cdot g(n)}$$

The mistake is in thinking that you can carry the constant up in to the exponent, but actually, the equation would look like this:

$$2^{f(n)} = c \cdot 2^{\cdot g(n)}$$

The example solution from CLRS is to

$$\begin{aligned} Let\ f(n) &= 2n \\ Let\ g(n) &= n \\ f(n) &\in O(g(n)) \\ because\ 2n &\leq cn \qquad \forall c \geq 2 \end{aligned}$$

Raising 2 to the power of both functions, though, means that $g(n) = 2^n$ while $f(n) = 4^n$. It's certainly possible that c could be equal to 2^n, but we should be able to solve for a c without bothering to include this caveat (ie, c should solve to a realistic constant value rather than an exponential function). Therefore, we have a contradiction and $f(n) > g(n)$.

For fun, here's another proof that begins from a similarly contrived supposition:

$$\begin{aligned} f(n) &= \sqrt{n} \\ g(n) &= \frac{\sqrt{n}}{2} \\ \sqrt{n} &\stackrel{?}{\leq} c \cdot \frac{\sqrt{n}}{2} \end{aligned}$$

Choose $c \geq 2$ and $f(n) \in O(g(n))$

$$\begin{aligned} f(n) &= 2^{\sqrt{n}} \\ g(n) &= 2^{\frac{\sqrt{n}}{2}} = \sqrt{2}^{\sqrt{n}} \\ 2^{\sqrt{n}} &\stackrel{?}{\leq} c \cdot \sqrt{2}^{\sqrt{n}} \end{aligned}$$

It seems that we would have to choose $c \geq \sqrt{2}^{\sqrt{n}}$ for this to work, but again, this is not a constant, but an exponential function.

Problem 2 Prove that $\lg(n!) \in \Theta(n \lg n)$:

Proving the upper bound is pretty simple, because we know that $n \lg n$ is $(\lg n + \lg n + ... + \lg n)$ (with n additions) which is just $\lg n$ multiplied n times. On the other hand, we know that $\lg(n!) = \lg n + lg(n-1) + ... + lg(1)$. From this, it is obvious that the only term that is equivalent to any term in the first summation is the first term, namely, $\lg n$. Every other term after the first in this summation is less than $\lg n$.

So, simply choose $c = 1$ and $n_0 \geq 1$ and

$$\lg(n!) \in O(n \lg n)$$

The lower bound is trickier to tease out; this proof involves thinking about a subset of the terms in lg $(n!)$.

$$\lg(n!) = \lg n + \lg(n-1) + ... + \lg(\frac{n}{k}) + ... + \lg(1) \tag{1}$$

Chopping the factorial sum at some subset of just $\lg(\frac{n}{k})$terms, we can set up the inequality

$$\lg(n!) = \lg n + \lg(n-1) + ... + \lg(\frac{n}{k}) + ... + \lg(1) \geq \lg(\frac{n}{k}) + ... + \lg(n) \tag{2}$$

Wherever we chop this upper subset of the factorial summation, we can choose the first or lowest term of this sum and multiply it $\frac{n}{k}$ times

$$\lg(\frac{n}{k}) + \lg(\frac{n}{k}) + \lg(\frac{n}{k}) + ... = (\frac{n}{k}) \lg \frac{n}{k}) \tag{3}$$

Since each term is the lowest term, we also know

$$\lg(\frac{n}{k}) + ... + \lg(n) \geq (\frac{n}{k}) \lg \frac{n}{k}) \tag{4}$$

$$\lg(n!) = \lg n + \lg(n-1) + ... + \lg(\frac{n}{k}) + ... + \lg(1) \geq \lg(\frac{n}{k}) + ... + \lg(n) \geq (\frac{n}{k}) \lg \frac{n}{k}) \tag{5}$$

Then, if we choose a factor of 2 for k, this becomes

$$\frac{n}{2^i} \lg \frac{n}{2^i} = \frac{n}{2^i}(\lg n - \lg 2^i) = \frac{n}{2^i}(\lg n - i) \tag{6}$$

Well, at this point, the i subtracted on the right is just a constant and we can ignore it (or we could multiply the $\frac{n}{2^i}$ through and then ignore it. Choose $c \geq \frac{1}{2^i}$ for some $i \geq 2$, now we have

$$\lg(n!) = \lg n + \lg(n-1) + ... + \lg(\frac{n}{k}) + ... + \lg(1) \geq \lg(\frac{n}{k}) + ... + \lg(n) = c \cdot n \lg n \tag{7}$$

$$\lg(n!) \in \Omega(n \lg n) \tag{8}$$

It follows from the above that $\lg(n!) \in \Theta\, n \lg n$

Problem 3: Peak-finding
(a) Linear Time algorithm
Note: Array A uses 0-origin indexing.

Input: Array A, $n = A.length$

```
peakFinder(A, n)
1       k = 0
2       for i = 0 → n:
3           k = A[i]
4           if (i == 0 ∧ k ≥ A[1]) ∨ (i == n − 1 ∧ k ≥ A[i − 1]):
5           return i
6           elif k ≥ A[i − 1] ∧ k ≥ A[i + 1] :
7           return i
```

This brute-force algorithm, as one would expect, walks through the array, checking every element to see if it is greater than both of its neighbors.

Loop invariant analysis:

This loop invariant is:
The array A or subarray at each iteration contains one or more peaks and that the subarray that has already been traversed doesn't contain any peaks.

Initialization: The initialization is sound because the algorithm is called with the length of the array, and k is initialized to 0 which is the beginning of the array. Since nothing in the array has been checked, the invariant holds.

Maintenance: If the peak is not the first element in the array, the algorithm ignores this element and increments k until the values adjacent to k on either side are $\leq k$ (lines 6-7). This means that before the loop starts, the invariant holds. If a peak isn't found, k is incremented, so the invariant holds after the loop as well.

Termination: Termination happens on lines 5 and 7. If the first element k is a peak ($\leq A[i+1]$), $i = 0$, the algorithm immediately halts and returns i (line 5). On the other hand, if the loop runs through the whole array

and finds a peak at the last element, it returns $i = n$ (line 5). Otherwise index of the peak is returned when k is a peak, in lines 6-7. An index is returned only if the adjacent elements are less than or equal than the element at the index. This shows that the algorithm is correct.

(b) The log $O(n)$ solution:
Note: This pseudo-code assumes that division of two integers resulting in a non-integer quotient returns the floor of the quotient.
Input: Array A, start, end

$fasterPeakFinder(A, start, end)$
1 $mid = (\frac{start+end}{2})$
2 if $(A[mid] \geq A[mid-1]) \wedge (A[mid] \geq A[mid+1])$:
3 return mid
4 elif $A[mid+1] \geq A[mid]$:
5 return $fasterPeakFinder(A, mid+1, end)$
6 elif $A[mid-1] \geq A[mid]$:
7 return $fasterPeakFinder(A, start, mid-1)$

This algorithm works similarly to binary search; it starts at the midpoint of the array and returns the index of the array once a peak is found. It recursively calls itself by breaking the problem down into a single subproblem half the size of itself based on whether or not the elements in the array adjacent to the midpoint (of each subproblem) are greater than the midpoint, with the logic being that of a person navigating in the dark reaching out and moving forward based on the space in front of them. If they reach a wall, they move toward space. In the same way, this algorithm gropes toward the incline, knowing that, at least it's not going downhill. Instead of just moving brute-force in the direction of incline, though, like a person in the dark, the algorithm knows the size of the array, and so it teleports itself to the midpoint of that array and gropes forth for the incline. In this way it may accidentally skip over a nearby peak in it's haste to break the problem down, but because of the division, it solves the general problem more quickly than brute-force.

Loop invariant:
Same as above except there exists a peak in the subarray at each recursive call.

Initialization: The initial call to the function takes the whole array as the input to the problem, because the initial call has $start = 0$ and $end = n$, where $n = A.length$. This is sound as long as end is not out of bounds of the array, which we could write an exception handler for, but will assume it's been asserted here.

Maintenance: Unlike the brute force *peakFinder*, this version is maintained via a recursive function call. The recursive calls (lines 4-7) divide the array in half by finding the element adjacent to the midpoint and calling the function with a new start or end, depending arbitrarily on the order of the flow control in this section (it doesn't matter if we go left or right first). Because we're either truncating up or down and taking the array's upper or lower half, we maintain the division of the array upon each recursive call. The recursion only happens if a peak hasn't been found yet, so the invariant holds.

Termination: If the middle element of the subproblem is a peak, the algorithm halts and returns the mid (this is the index of the array). Otherwise, the problem is divided as above and the midpoint recalculated, and returned if it is a peak. In the worst case, we will divide the array down to one element, which will also be the midpoint, which will also be a peak. Note that both this and the above algorithm will return a peak no matter what, because we are not using a strict $<$ or $>$ definition of a peak (in other words, it also may return a plateau). Otherwise, we could terminate each with a return of some sentinel error value (such as -1).

Recurrence Analysis:

$$T(n) = T(\frac{n}{2}) + 1$$
$$T(\frac{n}{2}) = T(\frac{n}{4}) + 1$$
$$\therefore T(n) = T(\frac{n}{4}) + 2$$
$$T(\frac{n}{4}) = T(\frac{n}{8}) + 1$$
$$\therefore T(n) = T(\frac{n}{8}) + 3$$
$$\implies T(n) = T(\frac{n}{2^i}) + i$$

As stated above (Termination) $T(1) = 1$ so

$$T(n) = T(\frac{n}{2^i})$$
$$i = lgn$$

Ignore the the constant and

$$T(n) \in O(\lg\ n)$$

(c) Finding a Hill

Input: 2D nxm Array A, n, m where n = number of rows and m = number of columns, col = indexing column

```
hillFinder(A, n, m, col)
1   j = col
2   max = −∞
3   for i = 0 → n:
4       if A[i][j] ≥ max:
5           max = A[i][j]
6           maxindex = A[i][j]
7           row = i
8     if j == 0 ∨ j == n − 1:
9       return maxindex
11   elif (A[row][j + 1] ≥ max) ∧ (A[row][j − 1] ≥ max) :
12           return maxindex
13   elif: (A[row][j + 1] ≥ max)
14       hillFinder(A, n, m, (m + j)/2)
15    elif: (A[row][j − 1] ≥ max)
16       hillFinder(A, n, m, (j − 1)/2)
```

Loop Invariant: The matrix contains a hill at each recursive call, and that prior to each recursion, a hill has not been discovered. Also, each recursive call breaks the problem down into a submatrix with fewer columns.

Initialization: Note that the initial call to this algorithm is $hillFinder(A, n, m, m/2)$ This is important as it chooses the middle column of the matrix. At this stage, a hill exists in the matrix and has not been found, so the invariant holds.

Maintenance: The recursive calls only happen if a hill hasn't been found. After the maximum of each column j is found, there are checks at lines 11-12 to see if the adjacent columns are less than this element. If not, a peak hasn't been found and the recursion happens, choosing a column to the left or right for the new value of j depending

order-arbitrarily (left or right) on whether that element is greater. Thus, the invariant holds.

Termination: The loop halts when a max element in a column is found and the adjacent elements to the left and right are less than the max element. If we are at the edge of the matrix, this is a base case and we have found an edge hill, which is returned. This makes sense because we only end up on the edge at the $(lgn)^{th}$ recurrence (the last recursive call).

Recurrence Analysis: At each recurrence, we divide the columns m in half and traverse the n rows, giving us:

$$\begin{aligned}
T(1) = n = 1T(m) &= T(\frac{m}{2}) + n \\
T(\frac{m}{2}) &= T(\frac{m}{4}) + n \\
\therefore T(m) &= T(\frac{m}{4}) + 2n \\
T(\frac{m}{4}) &= T(\frac{m}{8}) + n \\
\therefore T(m) &= T(\frac{m}{8}) + 3n \\
\implies T(m) &= T(\frac{m}{2^i}) + in \\
T(m) &= n \lg m
\end{aligned}$$

Since we are talking about a square nxn matrix (as stated in the problem), we have $T(n) = n \lg n$.