Bryan Arnold

CSE 3500

2/1/17

Homework 2

**Asymptotic Notations**

1. First, let $f(n) = 8n^3logn + 14n^2$ and $g(n) = n^3logn$. To find if $8n^3logn + 14n^2 = \Theta(n^3logn)$, it must be shown that $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$. To start, I'll show that $f(n) = \Omega(g(n)$.

   First, we need two constants c and $n_0$ such that $f(n) \geq c(g(n))$ for all $n \geq n_0$. By giving values to c and $n_0$, we can prove that $f(n) \geq c(g(n))$ for all $n \geq n_0$, and this is the method for proving that $f(n) = \Omega(g(n)$. We can assign the value of 1 to each of the constants, $c = 1$ and $n_0 = 1$, thus giving the following by substitution:

   $8n^3logn + 14n^2 \geq n^3logn$ for all $n \geq 1$. This statement is true, thus proving that $f(n) = \Omega(g(n)$. Next, I'll prove that $f(n) = O(g(n)$.

   To begin, we again need two constants c' and $n_0{}'$ such that $f(n) \leq c'(g(n))$ for all $n \geq n_0{}'$. We need to give values to c' and $n_0{}'$ to prove that following statement, which in turn proves that $f(n) = O(g(n)$. To make this statement true, we can assign the values of $c' = 22$ and $n_0{}' = 1$. Through substitution, we get the following:

   $8n^3logn + 14n^2 \leq 22n^3logn$ for all $n \geq 1$. This statement is true, thus proving that $f(n) = O(g(n)$. Now that the two components necessary to prove that $8n^3logn + 14n^2 = \Theta(n^3logn)$ have been found to be true, it can be said that $8n^3logn + 14n^2 = \Theta(n^3logn)$ is also true.

2. This statement is false. An example disproving this statement would be let $f(n) = 3n$ and $g(n) = n$. To prove that $f(n) = O(g(n)$, we must show that $f(n) \leq c(g(n))$ for all $n \geq n_0$. Simply assigning $c = 3$ and $n_0 = 1$, it can be show that:

   $3n \leq 3n$ for all $n \geq 1$, thus proving that $f(n) = O(g(n)$. Per the given statement, we should now be able to assume $2^{f(n)} = O(2^{g(n)})$, but when plugging in the same $f(n)$ and $g(n)$, this will be seen to be false: $2^{f(n)} = 6^n$ and $2^{g(n)} = 2^n$. $6^n$ grows faster than $2^n$, and $2^{f(n)} = O(2^{g(n)})$ states that $f(n) \leq c(g(n))$ for all $n \geq n_0$, so $f(n)$ will

eventually overtake $g(n)$ and cannot be bounded by a constant c. So, the statement if $f(n) = O(g(n)$, then $2^{f(n)} = O(2^{g(n)})$, is false.

3. First, let $f(n) = (n + b)^a$ and $g(n) = n^a$. To find if $(n + b)^a = \Theta(n^a)$, it must be shown that $f(n) = O(g(n)$ and $f(n) = \Omega(g(n))$. To start, I'll show that $f(n) = \Omega(g(n)$.

   First, we need two constants c and $n_0$ such that $f(n) \geq c(g(n))$ for all $n \geq n_0$. By giving values to c and $n_0$, we can prove that $f(n) \geq c(g(n))$ for all $n \geq n_0$, and this is the method for proving that $f(n) = \Omega(g(n)$. To begin, set up the problem as the following:

   $(\mathrm{n} + \mathrm{b})^{\mathrm{a}} \leq c(\mathrm{n} + \mathrm{b})^{\mathrm{a}}$, where $\mathrm{n} > 0$, where $n_0 = 0$ and c is not known yet. Next, since b is a real constant, the following can be done:

   $(\mathrm{n} + \mathrm{b})^{\mathrm{a}} \leq \mathrm{c}(\mathrm{n} + \mathrm{n})^{\mathrm{a}}$, where $\mathrm{n} > \mathrm{b}$. Simplifying this problem from this stage gives us:

   $(\mathrm{n} + \mathrm{b})^{\mathrm{a}} \leq c(2\mathrm{n})^{\mathrm{a}}$, where $\mathrm{n} > \mathrm{a}$. Since we wanted to find $\Omega(n^a)$, the constant c has been found (c = 2^a), meaning that f(n) is bounded, making the statement above true that $f(n) = \Omega(g(n)$, or $(n + b)^a = \Omega(n^a)$. Now, it must be shown that $f(n) = O(g(n)$ is also true to make $(n + b)^a = \Theta(n^a)$ true.

   To begin, we again need two constants c' and $n_0'$ such that $f(n) \leq c'(g(n))$ for all $n \geq n_0'$. We need to give values to c' and $n_0'$ to prove that following statement, which in turn proves that $f(n) = O(g(n)$. To begin, set up the problem as the following:

   $(\mathrm{n} + \mathrm{b})^{\mathrm{a}} \geq c'(\mathrm{n} - \mathrm{b})^{\mathrm{a}}$, where $\mathrm{n} > 0$, where $n_0 = 0$ and c is not known yet. Next, since b is a real constant, the following can be done:

   $(\mathrm{n} + \mathrm{b})^{\mathrm{a}} \geq c'(\mathrm{n})^{\mathrm{a}}$, where $\mathrm{n} \geq 2\mathrm{b}$ for $\mathrm{c}' = 1/2$ where $\mathrm{n} \geq 2\mathrm{b}$ as $\mathrm{n}/2 \leq \mathrm{n} - \mathrm{b}$, for $\mathrm{n} \geq 2\mathrm{b}$. The constant of c has been found for the following statement, meaning that the function is bounded making it true to state that $f(n) = O(g(n)$. Since it has been proven that $f(n) = O(g(n)$ and $f(n) = \Omega(g(n)$, it can be concluded that $(n + b)^a = \Theta(n^a)$ is also true when $c = 2^a, c' = 2^{-b}, and\ n_0 \geq 2b$.

**Asymptotic Order**

1. The order of the functions given in asymptotically ascending order by growth rate is as follows:

   1, lnn, $\lg^2 n$, n, nlogn, $n^2$, $n^3$, $2^{logn}$, $\frac{3^n}{2}$, $2^n$, $e^n$, $n * 2^n$, $then\ n!$

**Sum of two numbers**

1. For this algorithm, one could simply go through the algorithm, select an integer x, then check if an integer y exists that equals u – x. The only issue with this is, you'd have to check every integer across every other integer in the array, so you'd check each integer twice making the runtime O(n^2). To improve upon this runtime, you can first sort the algorithm using some sort of efficient sort. Heap sort or merge sort, maybe quick sort, would both work as they have efficient runtimes of O(nlogn). Now that the array is sorted from least to greatest, you can every element x in the array for an element y to see if there exists a y = u – x. A good searching algorithm to find this would be a binary search, as it eliminates half the array until the desired element is found, making it run at O(logn) time. You'd then display in some manner if elements exist in the array that satisfy x + y = u, then maybe display x and y. The two runtimes in this algorithm are O(logn) and O(nlogn). Using common sense rules, we can say the overall runtime for the algorithm is O(nlogn). The overall design of the algorithm would be as follows:

   **Algorithm:** twoIntegersMakeOneInteger
   **Input:** array A[1….n] of arbitrary integers and integer u.

   **Sort:** merge/heap sort of A[1…n] in increasing order.
   **End sort**

   **BinarySearch:** search for all integers x in array A that satisfy u – x = y, y is some other element in array A.
   **When/If found:** display the two integers that add to u.
   **End BinarySearch**

   **Display:** whether u can be made given the array (not needed but can add to the program).

   **End algorithm**

   **A problem on Graph**

1. Claim: Let G be a graph on n nodes, where n is an even number. Prove that if every node of G has degree at least n/2, then G is connected.

   I believe that the following claim is **true.**

   Proof:

   First, suppose that the above claim is false. So, this means that G is not connected which in turn means that there exist two nodes such that there isn't a path between them, but they are connected to at least n/2 other nodes.

Now, let node a and node b be the two nodes without a path between them. These two nodes total to be 2, meaning that out of the remaining total node n, only n-2 are remaining. An adjacent node must exist about nodes a and b, meaning that a path is connecting a and b together through the adjacent node. This denotes a contradiction of the supposition and makes the claim true as the supposition was a negation of the claim.