Lab 4: Algorithms Analysis

Sam Harder

1. Declaring that an algorithm is at least $O(n^2)$ only establishes a lower bound for how the run time grows as the input grows. Without an upper bound on run time growth we could have an algorithm that was $O(n^3)$ or, even worse, $O(4^n)$. Lacking an upper bound limits our ability to apply the algorithm, since we can't make any reasonable assumptions about how long the algorithm will take. It would be much more useful to have a statement such as "an algorithm A is at most $O(n^2)$".

2. (a)

$$2^{n+1} \leq c2^n$$
$$2 \leq c$$

So we may choose $c = 2$ and $n_0 = 0$.

(b)

$$2^{2n} \leq c2^n$$
$$2^n \leq c$$

Since c is a constant, and 2^n grows as n grows, there is no c large enough such that this inequality will always hold. Thus there is no c and n_0 to satisfy the big-O requirement.

3.

$$100^{100^{100}} < \lg n < \sqrt{n} < n, n - n^2 + 5n^3, 943n + 52 < n \lg n < n^2 < n^2 \lg n < n^3 < 2^n < 4^n$$

4. (a) $O(n)$
   (b) $O(n)$
   (c) $O(n^2)$
   (d) $O(n^2)$
   (e) $O(n^4)$

5. The big-O notation reflects how an algorithm's performance grows when dealing with very large data sets. Al is wrong in his statement. An $O(n \lg n)$-time algorithm is not *always* faster. The correct statement is that it is always faster beyond some n_0.

   To demonstrate this let's say we have two algorithms, one of which has n^2 steps and the other which has $n+30$. The first one is $O(n^2)$-time, the second is $O(n)$-time. By Al's logic, the $O(n)$ algorithm should always be faster than the $O(n^2)$ algorithm.

   Yet for inputs less than $n_0 = 6$ it is clear that

   $$n + 30 \geq n^2 \quad \text{where} \quad (0 \leq n \leq 6).$$

   Although for $n > 6$ the algorithm of $O(n^2)$ will always take longer than the $O(n)$ algorithm. This is an example of how algorithms with worse big-O can out perform function with better big-O with small inputs.