CS-6190: Homework 0

James Brissette

September 6, 2019

1. (a) YES. I have signed up for the course on Piazza.
   (b) YES. We know that if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} \leq \infty$ where $f(n) = n^2$ and $g(n) = n^3$, then $f(n) \in O(g(n))$. $O(n^2)$ is just a better upper-bound.
   (c) NO. We cannot assume to know the little-oh upper bound based solely on the Omega lower bound.
   (d) YES. We know that $n^{logn} << 2^n$ for n sufficiently large, and so using little-oh as a strict upper bound, if $\lim\limits_{n\to\infty} \frac{f(n)}{g(n)} = 0$ where $f(n) = n^{logn}$ and $g(n) = 2^n$, then $f(n) \in o(g(n))$.

2. (a) In order to have $\Theta(nk)$, we need to show $O(nk)$ and $\Omega(nk)$. So suppose we restrict the k largest elements of the array to be unsorted with respect to the rest of the array, and require the $n-k$ smallest elements to be in sorted order with respect to one another, you could achieve $\Theta(nk)$. In this scenario, after every iteration the largest element of n would propagate right until it reached it's final correct position, which would take n time to traverse the whole array. On the next pass through, you would take another n steps to make it to the end bringing along the next largest element until after k total iterations, your k displaced (and largest) elements would be sorted correctly, and the $n-k$ smallest elements would be grouped together as they were already sorted with respect to one another. In this contrived example you could not do any better or worse than $\Theta(nk)$ time.
   (b) If we make two comparisons and then recurse into a subdivision of size $\frac{n}{3}$ our recurrence becomes:

$$\begin{aligned} T(n) &= 2 + T(\frac{n}{3}) \\ &= 2 + (2 + T(\frac{n}{9})) \\ &= 2 + (2 + (2 + T(\frac{n}{3}))) \\ &\dots \\ &= 2r + T(\frac{n}{3^r}) \end{aligned}$$

Solving for r we get

$$\begin{aligned} \frac{n}{3^r} &= 1 \\ n &= 3^r \\ log_3(n) &= r \end{aligned}$$

and plugging back in we get

$$\begin{aligned} 2(log_3(n)) + T(1) &= 2log_3(n) + 1 \\ &= O(log_3 n^2) \end{aligned}$$

And this is worse than $O(log_2 n)$.

3. For any integer in the set N where each element of N is in the range N^3, you would make two comparisons at every step k, where each step would be the comparison of the k^{th} bit in the binary string representation of the integer. As N grows large, the number of bits needed to represent the largest possible binary representation of N^3 grows as well. If each bit can take two values, then r bits can take on up to 2^r possible numbers meaning you would need $log_2(M)$ bits to represent any number M. Given that for any N there are N^3 possible numbers, you would need at most $log_2(N^3)$ bits to represent the largest number possible given N.

   (a) In the worst case, in order to add an integer to the set in the same range, we would need to perform a comparison at every step and add a node that didn't exist before. This is two operations for every bit, so would take $O(2L)$ where $L = log_2(N^3)$.

   (b) In the worst case, to verify if some string belonged to the set, we would need to perform one comparison for every bit in the string, and then evaluate the boolean at the end. This would take $O(L)$ time where $L = log_2(N^3)$.

Bonus: This is considerably worse than a binary search tree when comparing numbers like this, in that a binary search tree, regardless of the size of the number, would make comparisons proportional to the size of the tree, not the length of the string.

4. (a)

   (b)

   (c)

5. One alternative algorithm to consider is first starting with the smallest set s where $\min_{1\leq i\leq t} s_i$. We know that this has the smallest number of elements to consider, and since we want the union across all sets, if it is not in this set, then we do not consider it. We make the assumption here that we are not taking into account pre-processing time and as a result assume that all the sets are already in sorted order. If this is the case then we can perform a binary search across every set, looking to see which elements in s are also present elsewhere. If we do this, we get that our search step takes $s * log(s_i)$ for each s_i. If we sum these across, we see that

$$\begin{aligned}\sum_{i=1}^{t} s(s_i) &= s\sum_{i=1}^{t} s_i \\ &= s(s_1 + s_2 + s_3 + \cdots + s_t) \\ &= O(s(s_1 + s_2 + s_3 + \cdots + s_t))\end{aligned}$$

6. As n grows large, the amortized complexity to add a single operation becomes n. Although you gain 100 new slots in the array every time you reach the capacity of your array, you still take n total operations to copy the old elements over into the new array. As n becomes HUGE, the benefit you gain from not having to create a new array for the next 100 consecutive adds becomes eclipsed by the number of copy operations needed to create the new array, essentially eliminating any benefit that was orignally present. The averaged run time for each operation approaches n and for N consecutive add operations, your run time becomes bounded by $O(n^2)$