MATH299M/CMSC389W
Spring 2019 – Dan Zou, Devan Tamot, Vlad Dobrin
Model H1: Asymptotic Complexity Exposition
Assigned: Friday August 30th
Due: Monday September 9th, 11:59PM

Welcome to your first homework assignment! Like I said in class, I'm going to try to leave the problem statement fairly open-ended because I want to see your different styles, I guarantee some of you will use functions that I've never seen or in ways I've never thought of before! First, here's an overview of the material in case you're not familiar with it:

Asymptotic Complexity refers to how many calculations a computer has to do to run a certain algorithm depending on the size of the input, that is, how much the computation time is going to scale with input size. For sorting algorithms, the state of the art is $nlogn$ time, meaning that if the size of the list you would like to sort is n, then it's going to take on the order of $nlogn$ computations, basic additions and multiplications, to get the job done.

Now, what do we mean by "on the order of"? Intuitively we mean up to scaling by constants, that is, if it takes n^2 time or $2n^2$ time we don't really care to make that distinction, we're interested in the *asymptotic behavior*. To quantitatively analyze this asymptotic behavior, naturally we introduce limits. We're going to say that two algorithms running in $f(n)$ and $g(n)$ computations satisfy f being *Big-O* of g, the notation being $f(n) = O(g(n))$, if in the limit their behavior is the same, formally:

$$f(n) = O(g(n)) \leftrightarrow \frac{f(n)}{g(n)} \in R^{\geq 0}$$

The Big-O expression $f = O(g)$ can be interpreted as "f is at least as fast as g", meaning that f might run asymptotically faster than g, or the same. Notice how if $f(n) = n^3$ and $g(n) = n^2$, for example, then the limit will diverge, meaning that $f \neq O(g)$, that is, f is asymptotically worse than g. Another way of saying this than with limits is that f is Big-O of g if and only if there exists some constant c such that for some N, for all n after that N, g(n) is bigger than c times f(n):

$$f(n) = O(g(n)) \leftrightarrow \exists\, c \in R^{\geq 0} \;\; \exists\, N \in Z^{+} \;\; \forall\, n \in Z^{>N} \;\; g(n) \geq cf(n)$$

So in other words, g(n) can be a better algorithm for small n, but in the limit as the size of the input goes to infinity, f(n) will be a better or comparable (off by a scalar multiplication) algorithm. Also note that the addition of asymptotically smaller terms doesn't effect anything as is obvious with the limit definition, that is, the behavior of a limit of a sum of terms in terms of n as n goes to infinity is only dictated by the behavior of the largest term.

Problem: I want a visualization of how these functions run compared to each other. How much worse is an n^2 algorithm than n (linear time) algorithm? n^3 vs. n^2? What if you compare a quadratic time algorithm to a linear time algorithm that's really steep, like f(n)=100n? What if

the linear time algorithm has huge overhead, that is, a huge constant term added to it – how big input does it take for the linear algorithm to actually be better than the quadratic one?

Ideas:

Summit in the U.S. is the fastest supercomputer in the world (https://www.theverge.com/circuitbreaker/2018/6/12/17453918/ibm-summit-worlds-fastest-supercomputer-america-department-of-energy), operating at 122.3 petaflops, meaning it can perform 122.3 x 10^{15} (that's quadrillions) floating point calculations (additions and multiplications) per *second*, if we're dealing with this kind of power, how big of input actually scares us, that is, maybe sometimes polynomial or even exponential time algorithms are fine if the input is small enough.

Brown University professor John Hughes (the guy who literally wrote the book on computer graphics) told me "Logarithmic time is basically constant", his argument went like this:

- The diameter of the observable universe is 93 billion light years, and the Planck volume is 4 x 10^{-105} cubic meters, meaning there are 1.1 x 10^{179} Planck volumes in the observable universe.
- So, by the laws of physics, 10^{179} is the maximum amount of data that could ever be encoded in our reality, and that's if you somehow managed to encode the smallest bit allowed by quantum physics over the entire observable universe.
- If n = 10^{179}, the log(n) time using base 10 is 179, and log(n) time using base 2 is still only 594.77; 595 computations could be done BY HAND if you wanted to!
- The logarithm function grows so incredibly slowly that if you get a log(n) time algorithm you don't need to look any further.

You could show me this!

A *polynomial time algorithm* is one that runs in time f(n) where f(n) is a polynomial in n. A *non-deterministic Polynomial-time algorithm* is one that runs in something worse than polynomial time, like exponential time, or worse. These are the *P* and *NP* in the famous problem *P vs. NP*.
Why is P vs. NP such a big deal? Is exponential time really that much worse than polynomial time? The best P vs. NP video by the way: https://www.youtube.com/watch?v=YX40hbAHx3s

Note: You can use the second week's notes as well! You may find the Map and Table functions particularly useful, maybe wait for in class to tackle the implicit function stuff though.