TheoryPrep Readings

May 2020

Selected from Mathematics for Computer Science (Lehman, Leighton, Meyer 2017), Building Blocks for Computer Science (Fleck 2013), Introduction to Theoretical Computer Science (Barak, 2020), CS 121 materials, and CS 125 materials.

2 Asymptotics

Pages 2 and 3 contain a summary of all the material for this week. The rest of the reading explains these concepts in more depth. We recommend reading it all if you have time (especially because the rest of the reading may be less dense)!

2.1 Asymptotic Definitions

It is important to be able to analyze the runtime of algorithms in a clean, generalizable way, and this is where asymptotics come in. The exact runtime of an algorithm may be very convoluted and case-dependent, and we want to be able to make a clear statement about how fast or slow an algorithm runs.

Let's say we have an algorithm that takes in strings of length n, and we want to see if it runs in fewer than n^2 steps. It might have some constant precalculations that it needs to do, so it might be slower than n^2 originally, and it might eventually take somewhere between n^2 and $2n^2$ to run, but neither of those things are that important to us as long as eventually as n increases, the algorithm runs faster than some constant c times n^2 (we don't care that much about the constant since eventually the n will far outpace it). Additionally, we'd like to know if it runs about at the speed of n^2 or if it runs much much faster. This is where asymptotics come in.

We use the asymptotics $O, \Omega, \Theta, o, \omega$ to denote how functions compare as their inputs go to ∞ (in particular, in CS 121 we will mostly use them to compare how algorithms' runtimes–which are functions themselves mapping the size of the input to runtime–compare to bounding functions like $n^2, 2^n$). A good way to remember $O, \Omega, \Theta, o, \omega$ is to relate them to $\leq, \geq, =, <<, >>$ respectively. If we say $f = O(g)$ this means that f is smaller than or equal to g as their inputs go to infinity. Similarly, $f = \Omega(g)$ means that f is larger than or equal to g as their inputs go to infinity. $f = \Theta(g)$ means that f and g are approximately equal. The small-oh and small-omega represent $<<, >>$, indicating that some functions are much much larger than others. $f = o(g)$ is similar to saying $f << g$ and $f = \omega(g)$ is similar to saying $f >> g$. For example, $x = o(x^2)$ because as x goes to ∞, x^2 increases far faster than x. This intuition motivates us to give more rigorous mathematical definitions.

We can thus define *Big-Oh* as

Definition 4. Let functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Then $f(n) = O(g(n))$ if $\exists_{c,N \in \mathbb{N}} \forall_{n \in \mathbb{N} | n > N} f(n) \leq c \cdot g(n)$.

Big-Omega is defined below, but before reading its definition, attempt to write it out for yourself

Definition 5. Let functions $f, g : \mathbb{N} \to \mathbb{R}^+$. Then $f(n) = \Omega(g(n))$ if $\exists_{c,N \in \mathbb{N}} \forall_{n \in \mathbb{N} | n > N} f(n) \geq c \cdot g(n)$.

Alternatively, we can say that $f(n) = \Omega(g(n))$ if $g(n) = O(f(n))$ (do you see how using the above definitions, we can prove the equivalencey of these two definitions of Big-Omega?).

We define *little-oh* and *little-omega* similarly,

Definition 6. For functions $f, g : \mathbb{N} \to \mathbb{R}^+$ we say $f(n) = o(g(n))$ if $\forall_{\epsilon > 0} \exists_{c,N \in \mathbb{N}} \forall_{n \in \mathbb{N} | n > N} f(n) < \epsilon g(n)$.

Delete only after doing 1.b

It is important to note that no asymptotic relationship may exist between two functions. For example, consider $f(x) = sin(x), g(x) = cos(x)$. Intuitively, we would expect neither function to be greater than the other, and we see this is true because for any N, there will always be an $n > N$ such that $f(n) = 0, g(n) \neq 0$ (this means that $g(n)$ is infinitely times more than $f(n)$) and another $n' > N$ such that $f(n') = 0, g(n') \neq 0$, so neither can be established to be greater than the other.

Another way that asymptotics are defined is with limits (which may not be so surprising given the name asymptotics). If these definitions seem even more obscure to you, please feel free to skip them, but for those to whom limits make more sense, these may provide more intuition. We can consider the limit below

$$\lim_{n \to \infty} \frac{f(n)}{g(n)}$$

If this evaluates to zero, then $f(n) = o(g(n))$ (since $g(n)$ is much larger). If this evaluates to a constant, this implies that $f(n) = \Theta(g(n))$. Note here that limits can be undefined, and this corresponds to functions not having an asymptotic relationship (consider the above example and its implied limit of $\lim_{n \to \infty} \frac{sin(n)}{cos(n)}$).

Chapter 14

Big-O

This chapter covers asymptotic analysis of function growth and big-O notation.

14.1 Running times of programs

An important aspect of designing a computer programs is figuring out how well it runs, in a range of likely situations. Designers need to estimate how fast it will run, how much memory it will require, how reliable it will be, and so forth. In this class, we'll concentrate on speed issues.

Designers for certain small platforms sometimes develop very detailed models of running time, because this is critical for making complex applications work with limited resources. E.g. making God of War run on your Iphone. However, such programming design is increasingly rare, because computers are getting fast enough to run most programs without hand optimization.

More typically, the designer has to analyze the behavior of a large C or Java program. It's not feasible to figure out exactly how long such a program will take. The transformation from standard programming languages to machine code is way too complicated. Only rare programmers have a clear grasp of what happens within the C or Java compiler. Moreover, a very detailed analysis for one computer system won't translate to another pro-

gramming language, another hardware platform, or a computer purchased a couple years in the future. It's more useful to develop an analysis that abstracts away from unimportant details, so that it will be portable and durable.

This abstraction process has two key components:

- Ignore behavior on small inputs, concentrating on how well programs handle large inputs. (This is often called asymptotic analysis.)
- Ignore multiplicative constants

Multiplicative constants are ignored because they are extremely sensitive to details of the implementation, hardware platform, etc. Behavior on small inputs is ignored, because programs typically run fast enough on small test cases. Difficult practical problems typically arise when a program's use expands to larger examples. For example, a small database program developed for a community college might have trouble coping if deployed to handle (say) all registration records for U. Illinois.

14.2 Asymptotic relationships

So, suppose that you model the running time of a program as a function $f(n)$, where n is some measure of the size of the input problem. E.g. n might be the number of entries in a database application. Your competitor is offering a program that takes $g(n)$ on an input of size n. Is $f(n)$ faster than $g(n)$, slower than $g(n)$, or comparable to $g(n)$? To answer this question, we need formal tools to compare the growth rates of two functions.

Suppose that $f(n) = n$ and $g(n) = n^2$. For small positive inputs, n^2 is smaller. For the input 1, they have the same value, and then g gets bigger and rapidly diverges to become much larger than f. We'd like to say that g is "bigger," because it has bigger outputs for large inputs.

When a function is the sum of faster and slower-growing terms, we'll only be interested in the faster-growing term. For example, $n^2 + 7n + 105$ will be treated as equivalent to n^2. As the input n gets large, the behavior of the

function is dominated by the term with the fastest growth (the first term in this case).

Formally, we'll compare two functions $f(n)$ and $g(n)$ whose inputs and outputs are real numbers by looking at their ratio $\frac{f(n)}{g(n)}$. Because we are only interested in the running times of algorithms, most of our functions produce positive outputs. The exceptions, e.g. the log function, produce positive outputs once the inputs are large enough. So this ratio exists as long as we agree to ignore a few smaller input values.

We then look at what happens to this ratio as the input goes to infinity. Specifically, we'll define

(asymptotically equivalent) $f(n) \sim g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 1$

(asymptotically smaller) $f(n) \ll g(n)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Recall from calculus that $\lim_{n\to\infty} h(n)$ is the value that output values $h(n)$ approach more and more closely as the input n gets larger and larger.[1] So two asymptotically equivalent functions become more and more similar as the input values get larger. If one function is asymptotically smaller, there is a gap between the output values that widens as the input values get larger.

So, for example $\lim_{n\to\infty} \frac{n^2+n}{n^2} = 1$ so $n^2 + n \sim n^2$. $\lim_{n\to\infty} \frac{n^2+17n}{n^3} = 0$ so $n^2 + 17n \ll n^3$. And $\lim_{n\to\infty} \frac{3n^2+17n}{n^2} = 3$ so $3n^2 + 17n$ and n^2 aren't related in either direction by $\sim$ or $\ll$.

If we pick two random functions f and g, $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ does not always exist because $\frac{f(n)}{g(n)}$ might oscillate rather than converging towards some particular number. In practice, we'll apply the above definitions only to selected well-behaved primitive functions and use a second, looser definition to handle both multiplicative constants and functions that might be badly behaved.

[1] Don't panic if you can't recall, or never saw, the formal definition of a limit. An informal understanding should be sufficient.

14.3 Ordering primitive functions

Let's look at some primitive functions and try to put them into growth order. It should be obvious that higher-order polynomials grow faster than lower-order ones. For example $n^2 \ll n^5$ because

$\lim_{n\to\infty} \frac{n^2}{n^5} = \lim_{n\to\infty} \frac{1}{n^3} = 0$

You're probably familiar with how fast exponentials grow. There's a famous story about a judge imposing a doubling-fine on a borough of New York, for ignoring the judge's orders. Since the borough officials didn't have much engineering training, it took them a few days to realize that this was serious bad news and that they needed to negotiate a settlement. So $n^2 \ll 2^n$. And, in general, for any exponent k, you can show that $n^k \ll 2^n$.

The base of the exponent matters. Consider 2^n and 3^n. $\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} (\frac{2}{3})^n = 0$ So $2^n \ll 3^n$.

Less obviously, $2^n \ll n!$. This is true because 2^n and $n!$ are each the product of n terms. For 2^n, they are all 2. For $n!$ they are the first n integers, and all but the first two of these are bigger than 2. So as n grows larger, the difference between 2^n and $n!$ widens. We'll do a formal proof of this result in Section 14.7 below.

We can summarize these facts as:

$$n \ll n^2 \ll n^3 \ldots \ll 2^n \ll 3^n \ll n!$$

For the purpose of designing computer programs, only the first three of these running times are actually good news. Third-order polynomials already grow too fast for most applications, if you expect inputs of non-trivial size. Exponential algorithms are only worth running on extremely tiny inputs, and are frequently replaced by faster algorithms (e.g. using statistical sampling) that return approximate results.

Now, let's look at slow-growing functions, i.e. functions that might be the running times of efficient programs. We'll see that algorithms for finding entries in large datasets often have running times proportional to $\log n$. If you draw the log function and ignore its strange values for inputs smaller than 1, you'll see that it grows, but much more slowly than n. And the

constant function 1 grows even more slowly: it doesn't grow at all!

Algorithms for sorting a list of numbers have running times that grow like $n \log n$. We know from the above that $1 \ll \log n \ll n$. We can multiply this equation by n because factors common to f and g cancel out in our definition of $\ll$. So we have $n \ll n \log n \ll n^2$.

We can summarize these primitive function relationships as:

$$1 \ll \log n \ll n \ll n \log n \ll n^2$$

It's well worth memorizing the relative orderings of these basic functions, since you'll see them again and again in this and future CS classes.

14.4 The dominant term method

The beauty of these asymptotic relationships is that it is rarely necessary to go back to the original limit definition. Having established the relationships among a useful set of primitive functions, we can usually manipulate expressions at a high level. And we can typically look only at the terms that dominant each expression, ignoring other terms that grow more slowly.

The first point to notice is that $\sim$ and $\ll$ relations for a sum are determined entirely by the dominant term, i.e. the one that grows fastest. If $f(n) \ll g(n)$, then $g(n) \sim g(n) \pm f(n)$. For example, suppose we are confronted with the question of whether

$$47n + 2n! + 17 \ll 3^n + 102n^3$$

The dominant term on the lefthand side is $2n!$. On the righthand side, it is 3^n. So our question reduces to whether

$$2n! \ll 3^n$$

We know from the previous section that this is false, because

$$3^n \ll n!$$

.

The asymptotic relationships also interact well with normal rules of algebra. For example, if $f(n) \ll g(n)$ and $h(n)$ is any non-zero function, then $f(n)h(n) \ll g(n)h(n)$.

14.5 Big-O

Asymptotic equivalence is a great way to compare well-behaved reference functions, but it is less good for comparing program running times to these reference functions. There are two issues. We'd like to ignore constant multipliers, even on the dominant term, because they are typically unknown and prone to change. Moreover, running times of programs may be somewhat messy functions, which may involve operations like floor or ceiling. Some programs[2] may run dramatically faster on "good" input sizes than on "bad" sizes, creating oscillation that may cause our limit definition not to work.

Therefore, we typically use a more relaxed relationship, called **big-O** to compare messier functions to one another or two reference functions. Suppose that f and g are functions whose domain and co-domain are subsets of the real numbers. Then $f(n)$ is $O(g(n))$ (read "big-O of g) if and only if

> There are positive real numbers c and k such that $0 \leq f(n) \leq cg(n)$ for every $n \geq k$.

The factor c in the equation models the fact that we don't care about multiplicative constants, even on the dominant term. The function $f(n)$ is allowed to wiggle around as much as it likes, compared to $cg(n)$, as long as it remains smaller. And we're entirely ignoring what f does on inputs smaller than k.

If $f(n)$ is asymptotically smaller than $g(n)$, then $f(n)$ is $O(g(n))$ but $g(n)$ is not $O(f(n))$ but So, for example, $3n^2 + 17n$ is $O(2^n)$. But $3n^2$ is not $O(n + 132)$. The big-O relationship also holds when the two functions have the same dominant term, with or without a difference in constant multiplier. E.g. $3n^2$ is $O(n^2 - 17n)$ because the dominant term for both functions has

[2]Naive methods of testing for primality, for example.

the form cn^2. So the big-O relationship is a non-strict partial order like $\leq$ on real numbers, whereas $\ll$ is a strict partial order like $<$.

When $g(n)$ is $O(f(n))$ and $f(n)$ is $O(g(n))$, then $f(n)$ and $g(n)$ are forced to remain close together as n goes to infinity. In this case, we say that $f(n)$ is $\Theta(g(n))$ (and also $g(n)$ is $\Theta(f(n))$). The Θ relationship is an equivalence relation on this same set of functions. So, for example, the equivalence class $[n^2]$ contains functions such as n^2, $57n^2 - 301$, $2n^2 + n + 2$, and so forth.

Notice that logs with different bases differ only by a constant multiplier, i.e. a multiplier that doesn't depend on the input n. E.g. $\log_2(n) = \log_2(3)\log_3(n)$. This means that $\log_p(n)$ is $\Theta(\log_q(n))$ for any choice of p and q. Because the base of the log doesn't change the final big-O answer, computer scientists often omit the base of the log function, assuming you'll understand that it does not matter.

14.6 Applying the definition of big-O

To show that a big-O relationship holds, we need to produce suitable values for c and k. For any particular big-O relationship, there are a wide range of possible choices. First, how you pick the multiplier c affects where the functions will cross each other and, therefore, what your lower bound k can be. Second, there is no need to minimize c and k. Since you are just demonstrating existence of suitable c and k, it's entirely appropriate to use overkill values.

For example, to show that $3n$ is $O(n^2)$, we can pick $c = 3$ and $k = 1$. Then $3n \leq cn^2$ for every $n \geq k$ translates into $3n \leq 3n^2$ for every $n \geq 1$, which is clearly true. But we could have also picked $c = 100$ and $k = 100$.

Overkill seems less elegant, but it's easier to confirm that your chosen values work properly, especially in situations like exams. Moreover, slightly overlarge values are often more convincing to the reader, because the reader can more easily see that they do work.

To take a more complex example, let's show that $3n^2 + 7n + 2$ is $O(n^2)$. If we pick $c = 3$, then our equation would look like $3n^2 + 7n + 2 \leq 3n^2$. This clearly won't work for large n.

So let's try $c = 4$. Then we need to find a lower bound on n that makes $3n^2 + 7n + 2 \leq 4n^2$ true. To do this, we need to force $7n + 2 \leq n^2$. This will be true if n is big, e.g. ≥ 100. So we can choose $k = 100$.

14.7 Proving a primitive function relationship

Let's see how to pin down the formal details of one ordering between primitive functions, using induction. I claimed above that $2^n \ll n!$. To show this relationship, notice that increasing n to $n+1$ multiplies the lefthand side by 2 and the righthand side by $n + 1$. So the ratio changes by $\frac{2}{n+1}$. Once n is large enough $\frac{2}{n+1}$ is at least $\frac{1}{2}$. Firming up the details, we find that

Claim 50 *For every positive integer $n \geq 4$, $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$.*

If we can prove this relationship, then since it's well-known from calculus that $(\frac{1}{2})^n$ goes to zero as n increases, $\frac{2^n}{n!}$ must do so as well.

To prove our claim by induction, we outline the proof as follow:

> Proof: Suppose that n is an integer and $n \geq 4$. We'll prove that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ using induction on n.
>
> Base: $n = 4$. [show that the formula works for $n = 4$]
>
> Induction: Suppose that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ holds for $n = 4, 5, \ldots, k$. And, in particular, $\frac{2^k}{k!} < (\frac{1}{2})^{k-4}$
>
> We need to show that the claim holds for $n = k + 1$, i.e. that $\frac{2^{k+1}}{(k+1)!} < (\frac{1}{2})^{k-3}$

When working with inequalities, the required algebra is often far from obvious. So, it's especially important to write down your inductive hypothesis (perhaps explicitly writing out the claim for the last one or two values covered by the hypothesis) and the conclusion of your inductive step. You can then work from both ends to fill in the gap in the middle of the proof.

Proof: Suppose that n is an integer and $n \geq 4$. We'll prove that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ using induction on n.

Base: $n = 4$. Then $\frac{2^n}{n!} = \frac{16}{24} < 1 = (\frac{1}{2})^0 = (\frac{1}{2})^{n-4}$.

Induction: Suppose that $\frac{2^n}{n!} < (\frac{1}{2})^{n-4}$ holds for $n = 4, 5, \ldots, k$.

Since $k \geq 4$, $\frac{2}{k+1} < \frac{1}{2}$. So $\frac{2^{k+1}}{(k+1)!} < \frac{1}{2} \cdot \frac{2^k}{k!}$.

By our inductive hypothesis $\frac{2^k}{k!} < (\frac{1}{2})^{k-4}$. So $\frac{1}{2} \cdot \frac{2^k}{k!} < (\frac{1}{2})^{k-3}$.

So then we have $\frac{2^{k+1}}{(k+1)!} < \frac{1}{2} \cdot \frac{2^k}{k!} < (\frac{1}{2})^{k-3}$ which is what we needed to show.

Induction can be used to establish similar relationships between other pairs of primitive functions, e.g. n^2 and 2^n.

14.8 Variation in notation

Although the concepts in this area are reasonable constant across authors, the use of shorthand symbols is not. The symbol $\sim$ is used for many diverse purposes in mathematics. Authors frequently write "f(n) is o(g(n))" (with a small o rather than a big one) to mean $f(n) \ll g(n)$. And $\ll$ may be used to mean big-O. Fortunately, the usage of big-O seems to be standard.

In computer science, we typically look at the behavior of functions as input values get large. Areas (e.g. perturbation theory) use similar definitions and notation, but with limits that tend towards some specific finite value (e.g. zero).

In the definition of big-O, some authors replace $0 \leq f(n) \leq cg(n)$ with $|f(n)| \leq c|g(n)|$. This version of the definition can compare functions with negative output values but is correspondingly harder for beginners to work with. Some authors state the definition only for functions f and g with positive output values. This is awkward because the logarithm function produces negative output values for very small inputs.

Outside theory classes, computer scientists often say that $f(n)$ is $O(g(n))$ when they actually mean the stronger statement that $f(n)$ is $\Theta(g(n))$. When you do know that the bound is tight, i.e. that the functions definitely grow