CS 4349.003: Advanced Algorithm Design and Analysis **Fall 2019**

Lecture 3 — August 26

Asymptotic Analysis

Last week, we looked at the Tower of Hanoi puzzle where we had to move a stack of disks from one peg to another, moving only one disk at a time, and never letting a larger disk sit upon a smaller one. By the end of the lecture, we derived the following algorithm for solving the puzzle. HANOI(n, src, dst, tmp) describes how to transfer the smallest n disks of a collection where disks are numbered 1 through n from smallest to biggest. These n disks start at peg src, end at dst, and occasionally move to the temporary peg tmp.

```
HANOI(n, src, dst, tmp):
  if n > 0
      HANOI(n−1, src, tmp, dst) ⟨⟨Recurse!⟩⟩
      move disk n from src to dst
      HANOI(n−1, tmp, dst, src) ⟨⟨Recurse!⟩⟩
```

We then proved that HANOI(n, src, tmp, dst) will perform exactly $2^n - 1$ moves. In terms of real world time, the seven disk toy I brought would have taken me a little over two minutes to solve under the unrealistic assumption that I could move one disk every second. With 15 disks, it would take over nine hours to solve the puzzle. Moving 64 disks as described by the original story would take around 585 billion years! This result shouldn't be too surprising, though, because we should "intuitively" know that the function 2^n grows *very, very quickly*. Today, we will formalize this notion of how fast a function grows by reviewing ***asymptotic notation***.

3.1 Asymptotic Notation

3.1.1 "Big-oh" notation

You're likely already familiar with using O-notation (big-oh notation) to express algorithm running times. Let $g(n) : \mathbb{N} \to \mathbb{R}^+$ be a positive function over the natural numbers. We use O-notation to describe other functions that grow no quicker than g *up to constant factors*. Let $f(n) : \mathbb{N} \to \mathbb{R}^+$ be another positive function over the natural numbers; perhaps $f(n)$ represents the running time of an algorithm whose input size is n. We say $f(n)$ is *in* or *equals* $O(g(n))$ if there is some point at which $f(n)$ becomes smaller than $g(n)$ after scaling $g(n)$ up by a constant factor.

Formally,

$$O(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le f(n) \le cg(n) \text{ for all } n \ge n_0\}.$$

So, $f(n) \in O(g(n))$ means that there are some constants c and n_0 determined by our particular function $f(n)$. When n becomes large enough ($n \ge n_0$), then $f(n)$ becomes smaller than $cg(n)$.

For smaller values of n, we frankly don't care, because $f(n)$ must be "small" in those cases as well. Again, we often write $f(n) = O(g(n))$ and say $f(n)$ equals or is $O(g(n))$ even though *technically* $g(n)$ is a set containing $f(n)$. This abuse of notation can be formalized, though; see below.

O-notation provides a *loose* upper bound for how fast a function grows. And as we'll see later, it lets you ignore constant factors and "lower order" terms. In other words, it's perfectly valid to state that $n = O(n^2)$, that $25n^3 = O(n^3)$, and that $0.0001n^2 + 100000n + 123456789 = O(n^2)$. When expressing the running time of algorithms, you should state as simple a O bound as you can that is true even in the worst case. For example, the procedure HANOI(n, src, dst, tmp) uses $O(2^n)$ moves.

One nice feature of O-notation being a loose upper bound is that your can claim running times that aren't *exactly* tight, but are still accurate. Maybe you know an algorithm runs in $O(n^2 \log n)$ time, but you suspect it might really be $O(n^2)$; you just have no way to prove it. The $O(n^2 \log n)$ time bound is still correct, just loose. Of course, the more accurate a time bound you can prove, the more points you'll get in the homework and exams. Don't simply claim everything runs in $O(2^{2^{2^n}})$ time. That statement barely says anything except that your algorithm will eventually terminate.

3.1.2 Ω and Θ-notation

Sometimes, a loose upper bound isn't what you want. For example, HANOI$(n, \ldots)$ using $O(2^n)$ moves isn't that illuminating; most functions we'll encounter in this class are in $O(2^n)$ time, including running times for simple tasks like sorting! If you want a loose lower bound, use Ω-notation (big-omega notation).

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\}.$$

In contrast to O-notation, $f(n) \in \Omega(g(n))$ (or more simply, $f(n) = \Omega(g(n))$) means that $f(n)$ eventually grows *at least as large* as some constant factor times $g(n)$. HANOI$(n, \ldots)$ also uses $\Omega(2^n)$ moves, and this statement tells us we're guaranteed to use *a lot* of moves when n gets big enough. If you need to argue that something is *at least* so big, use Ω-notation.

Finally, you sometimes know exactly how fast a function grows, up to constant factors. For these situations, we have Θ-notation (theta-notation; hey no big this time).

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$

In other words, $f(n)$ is both smaller than a constant multiple of $g(n)$ and *larger* than a different constant multiple once n gets large enough. Equivalently, $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. In these cases, we say $g(n)$ is an ***asymptotically tight bound*** for $f(n)$. You should use Θ notation whenever you want to emphasize that you know exactly how quickly a function grows. We know HANOI$(n, \ldots)$ uses $\Theta(2^n)$ moves.

3.1.3 o and ω-notation

Finally, there are the rare occasions where a loose upper or lower bound aren't what you want, and you instead want something akin to a tight inequality. In other words, you'd like to express a function is O or $\Omega(g(n))$, but you also want to express that it is *not* $\Theta(g(n))$. For these cases, we use o-notation (little-oh notation) or ω-notation (little-omega notation).

$$o(g(n)) = \{f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq f(n) < cg(n) \text{ for all } n \geq n_0\}$$

$$\omega(g(n)) = \{f(n) : \text{for any constant } c > 0, \text{ there exists a constant } n_0 > 0 \text{ such that } 0 \leq cg(n) < f(n) \text{ for all } n \geq n_0\}$$

In other words, no matter which constant c you choose, $g(n)$ will eventually become bigger or smaller than $f(n)$ and they will continue to separate as n grows larger. We use these little notations whenever we want to emphasize that something is strictly bigger or smaller than something else. For example, we know HANOI$(n, \dots)$ takes a long time, because it uses $\omega(n^{10000})$ moves! But that's not so bad, right? After all, it only uses $o(4^n)$ moves!

3.2 Working with Asymptotics

There are a number of useful rules for working with asymptotics that I'll claim without proof. We'll use these rules extensively, often without explicitly mentioning them, almost every time we analyze an algorithm in this class. Unless specified otherwise, we'll assume throughout this section that any functions we mention are positive for sufficiently large n.

3.2.1 Comparing functions

Many of the properties are you used to seeing from comparison of real numbers also apply to asymptotics.

Transitivity $f(n) = x(g(n))$ and $g(n) = x(h(n))$ implies $f(n) = x(h(n))$ where x is any one of the five $\Theta, O, \Omega, o, \omega$ discussed above.

Reflexivity $f(n) = \Theta(f(n))$, $f(n) = O(f(n))$, and $f(n) = \Omega(f(n))$.

Symmetry and transpose symmetry $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. Similarly, $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$, and $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$.

3.2.2 "Algebra"

Suppose $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$. Then,

$$c \cdot f_1(n) = O(f_1(n)) \text{ for any positive constant } c, \tag{3.1}$$
$$f_1(n) + f_2(n) = O(g_1(n) + g_2(n)), \tag{3.2}$$
$$f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n)), \text{ and} \tag{3.3}$$
$$f_1(n) + f_2(n) = O(\max\{g_1(n), g_2(n)\}). \tag{3.4}$$

The first equation applies to Ω and O as well. The remaining equations apply to all five pieces of asymptotic notation. Observe that the first equation implies $c = O(1)$ for any positive constant c. Consequently, we'll often denote constants as $O(1)$ or $\Theta(1)$.

3.2.3 Example: Analyzing Fibonacci Multiplication

Let's try applying the above rules to analyze the running time of a non-trivial algorithm. Last week, we saw a procedure FIBONACCIMULTIPLY($X[0..m-1], Y[0..n-1]$) for multiplying two non-negative integers x and y such that

$$x = \sum_{i=0}^{m-1} X[i] \cdot 0^i \quad \text{and} \quad \sum_{j=0}^{n-1} Y[j] \cdot 10^j.$$

The output was an array $Z[0..m+n-1]$ representing the product

$$z = x \cdot y = \sum_{k=0}^{m+n-1} Z[k] \cdot 10^k.$$

Here is the psuedocode.

```
FIBONACCIMULTIPLY(X[0 .. m - 1], Y[0 .. n-1]):
  hold ← 0
  for k ← 0 to n + m − 1
      for all i and j such that i + j = k
          hold ← hold + X[i] · Y[j]
      Z[k] ← hold mod 10
      hold ← ⌊hold/10⌋
  return Z[0..m + n − 1]
```

We've only discussed asymptotic notation over one variable so far, so let's analyze the algorithm's running time in the simple case where $n = m$. We'll assume we can multiply two digits together, add a two-digit number to another number of arbitrary size, look up the least significant digit of a number (i.e. compute $hold \mod 10$), and remove the least significant digit of a number (i.e. compute $\lfloor hold/10 \rfloor$) in constant time per operation.

We'll begin by looking at the inner for loop. By (3.4), the operation $hold \leftarrow hold + X[i] \cdot Y[j]$ takes constant time. It occurs $k + 1 \leq 2n - 1 = O(n)$ times for any fixed value of k, so (3.3) implies the inner for loop takes $O(n)$ time for any fixed value of k. Applying (3.4) a couple

times, we see each iteration of the outer for loop takes $O(n)$ time as well. There are $2n-1$ iterations of the outer for loop, so (3.3) implies the outer loop takes $O(n^2)$ time total. Finally, the remaining operations take constant time, so (3.4) implies the whole algorithm takes $O(n^2)$ time total.

That explanation was more detailed than any I'll expect from you or even myself for the rest of the semester. If I were analyzing this algorithm as part of any other lecture, I might say, "It has a doubly nested for loop over $O(n)$ values per loop, and everything else is insignificant, so the running time is $O(n^2)$." You'll have to use your best judgment for how obvious such statements are and whether they need more explanation. We'll sometimes need to be more clever with how we count operations, especially when we start discussing graph algorithms.

3.3 Other considerations

3.3.1 Multiple variables

Sometimes, you'll see two or more variables appear in asymptotic notation. For example, it would be reasonable to say that FIBONACCIMULTIPLY$(X[0..m-1], Y[0..n-1])$ runs in $O(mn)$ time. Unfortunately, there is no firm agreement on the formal definition of these notations over multiple variables. CLRS (Exercise 3.1-8) would say the inequalities should hold when *at least one* variable is large enough. I personally find this definition a bit odd, because we would be claiming FIBONACCIMULTIPLY would take 0 time when $m = 0$ and n is sufficiently large. The algorithm would still be correct in this case; it would just iterate over all k from 0 to $n-1$ and return an array of 0s. This iteration would actually take $\Theta(n)$ time, which certainly grows faster than 0. (Double check the definition of x. If we define the summation over nothing as 0, then an empty array X is a perfectly valid way to represent $x = 0$.)

For this class, I prefer to say the inequalities hold when *all* the variables are sufficiently large. Surprisingly, the rules in this section for adding or multiplying don't necessarily hold for either formulation if you work with badly behaved functions. We won't encounter any of these examples in this class, however, so you should be able to trust that your intuition for how to add and multiply functions is correct.

3.3.2 Asymptotic notation in equations and inequalities

Similar to how we abuse notation by writing $f(n) = O(g(n))$ when we're really talking about set inclusion, we may also include asymptotic notation in the middle of more complicated equations or inequalities. For example, we might say $2n^2 + O(n) + O(1) = O(n^2)$ or $42n^3 + 439n^2 + 5 \leq 50n^3 + \Theta(n^2)$. The (informal) rule for these situations is that *for all* choices of functions on the left falling within the asymptotic sets, *there exists* a choice of functions on the right that make the equation or inequality true. In the first example, we would have to consider the possibility of the left hand side being $2n^2+500n+2$ as one of many. For this case, we could set the right hand side to be $2n^2+500n+2$ as well, which is in $O(n^2)$. In the second example, the only possible function for the left hand side is given to us, but we are free to choose the function $50n^3 + 500n^2 + 10$ for the right hand side as one way to make the inequality true.

3.3.3 Important functions in asymptotic analysis

A ***polynomial in n of degree d*** is a function $p(n) = \sum_{i=0}^{d} a_i n^i$ where each a_i is a ***coefficient*** of the polynomial and $a_d \neq 0$. We have $p(n) = \Theta(n^d)$. For any real constants $\beta > \alpha \geq 0$, we have $n^\alpha = o(n^\beta)$. A function $f(n)$ is ***polynomially bounded*** if $f(n) = O(n^k)$ for some constant k.

Exponential functions grow faster than polynomial functions, and the base of the exponential matters. In particular, $n^k = o(a^n)$ for any constant k and constant $a > 1$. Also, $a^n = o(b^n)$ for any constants $b > a > 0$.

On the other hand, logarithmic functions grow slower than polynomials, even ones that are in $\Theta(n^a)$ for really small constant $a > 0$. A function $f(n)$ is ***polylogarithmically bounded*** if $f(n) = O(\log^k n)$ for some constant k. We have $\log^k n = o(n^a)$ for any constant k and constant $a > 0$. We will generally use log to denote the logarithm of base 10, lg to denote the logarithm of base 2, and ln to denote the natural logarithm of base e (Euler's constant). However, from earlier facts we may observe that

$$\log_a n = \frac{\log_b n}{\log_b a} = \Theta(\log_b n)$$

for any constants $a, b > 1$. In other words, the base of the logarithm does not matter when doing asymptotics, as long as that base is a constant and you are merely multiplying by the logarithm. The base does matter if the logarithm is in an exponent, though. For example, $n^{\ln 5} = o(n^{\lg 5})$, because $\ln 5 < \lg 5$.

In the context of algorithm design, correct algorithms with exponential running times are usually easy to find; these algorithms often correspond to some kind of "brute force" solution where we try every possible output and return the best one. We usually don't consider an algorithm to be "efficient" unless it has a polynomial running time, but even then, we prefer to get the running time as close to linear (meaning $O(n)$) as possible. When the input is very special, say a sorted array, we may be able to avoid reading the whole thing and achieve an even better polylogarithmic running time. In practice, the constants hidden in the O-notation do matter, but most of the algorithms in this class are simple enough that asymptotic behaviors take over even for moderately large values of n.