CS 61B: Lecture 20
Monday, March 10, 2014

Today's reading: Goodrich & Tamassia, Chapter 4 (especially 4.2 and 4.3).

ASYMPTOTIC ANALYSIS (bounds on running time or memory)
==========
Suppose an algorithm for processing a retail store's inventory takes:
  - 10,000 milliseconds to read the initial inventory from disk, and then
  - 10 milliseconds to process each transaction (items acquired or sold).
Processing n transactions takes (10,000 + 10 n) ms. Even though 10,000 >> 10, we sense that the "10 n" term will be more important if the number of transactions is very large.

We also know that these coefficients will change if we buy a faster computer or disk drive, or use a different language or compiler. We want a way to express the speed of an algorithm independently of a specific implementation on a specific machine--specifically, we want to ignore constant factors (which get smaller and smaller as technology improves).

Big-Oh Notation (upper bounds on a function's growth)

Big-Oh notation compares how quickly two functions grow as n -> infinity.

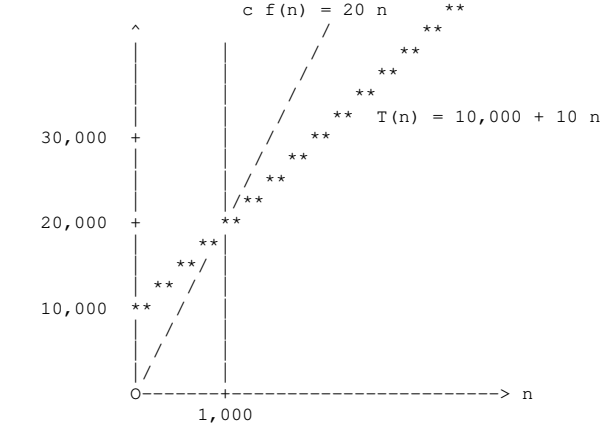
Let n be the size of a program's _input_ (in bits or data words or whatever).
Let T(n) be a function. For now, T(n) is the algorithm's precise running time in milliseconds, given an input of size n (usually a complicated expression).
Let f(n) be another function--preferably a simple function like f(n) = n.

We say that T(n) is in O(f(n)) IF AND ONLY IF T(n) <= c f(n)
WHENEVER n IS BIG, FOR SOME LARGE CONSTANT c.

 * HOW BIG IS "BIG"? Big enough to make T(n) fit under c f(n).
 * HOW LARGE IS c? Large enough to make T(n) fit under c f(n).

EXAMPLE: Inventory

Let's consider the function T(n) = 10,000 + 10 n, from our example above.
Let's try out f(n) = n, because it's simple. We can choose c as large as we want, and we're trying to make T(n) fit underneath c f(n), so pick c = 20.



As these functions extend forever to the right, their asymptotes will never cross again. For large n--any n bigger than 1,000, in fact--T(n) <= c f(n).
*** THEREFORE, T(n) is in O(f(n)). ***

Next, you must learn how to express this idea rigorously. Here is the central lesson of today's lecture, which will bear on your entire career as a professional computer scientist, however abstruse it may seem now:

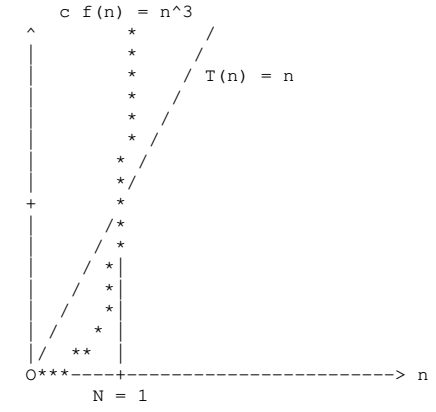
```
|=============================================================================|
| FORMALLY:  O(f(n)) is the SET of ALL functions T(n) that satisfy:           |
|                                                                             |
|   There exist positive constants c and N such that, for all n >= N,         |
|                              T(n) <= c f(n)                                 |
|=============================================================================|
```

Pay close attention to c and N. In the graph above, c = 20, and N = 1,000.

Think of it this way: if you're trying to prove that one function is asymptotically bounded by another [f(n) is in O(g(n))], you're allowed to multiply them by positive constants in an attempt to stuff one underneath the other. You're also allowed to move the vertical line (N) as far to the right as you like (to get all the crossings onto the left side). We're only interested in how the functions behave as n shoots off toward infinity.

EXAMPLES: Some Important Corollaries
---
[1] 1,000,000 n is in O(n). Proof: set c = 1,000,000, N = 0.
 -> Therefore, Big-Oh notation doesn't care about (most) constant factors. We generally leave constants out; it's unnecessary to write O(2n), because O(2n) = O(n). (The 2 is not wrong; just unnecessary.)

[2] n is in O(n^3). [That's n cubed]. Proof: set c = 1, N = 1.
 -> Therefore, Big-Oh notation can be misleading. Just because an algorithm's running time is in O(n^3) doesn't mean it's slow; it might also be in O(n). Big-Oh notation only gives us an UPPER BOUND on a function.



[3] n^3 + n^2 + n is in O(n^3). Proof: set c = 3, N = 1.
 -> Big-Oh notation is usually used only to indicate the dominating (largest and most displeasing) term in the function. The other terms become insignificant when n is really big.

Here's a table to help you figure out the dominating term.

Table of Important Big-Oh Sets

Arranged from smallest to largest, happiest to saddest, in order of increasing domination:

	function		common name
	O(1)	::	constant
is a subset of	O(log n)	::	logarithmic
is a subset of	O(log^2 n)	::	log-squared [that's (log n)^2]
is a subset of	O(root(n))	::	root-n [that's the square root]
is a subset of	O(n)	::	linear
is a subset of	O(n log n)	::	n log n
is a subset of	O(n^2)	::	quadratic
is a subset of	O(n^3)	::	cubic
is a subset of	O(n^4)	::	quartic
is a subset of	O(2^n)	::	exponential
is a subset of	O(e^n)	::	exponential (but more so)

Algorithms that run in O(n log n) time or faster are considered efficient. Algorithms that take n^7 time or more are usually considered useless. In the region between n log n and n^7, the usefulness of an algorithm depends on the typical input sizes and the associated constants hidden by the Big-Oh notation.

If you're not thoroughly comfortable with logarithms, read Sections 4.1.2 and 4.1.7 of Goodrich & Tamassia carefully. Computer scientists need to know logarithms in their bones.

Warnings
---

[1] Here's a fallacious proof:

n^2 is in O(n), because if we choose c = n, we get n^2 <= n^2.
-> WRONG! c must be a constant; it cannot depend on n.

[2] The big-Oh notation expresses a relationship between functions. IT DOES NOT SAY WHAT THE FUNCTIONS MEAN. In particular, the function on the left does not need to be the worst-case running time, though it often is. The number of emails you send to your Mom as a function of time might be in O(t^2). In that case, not only are you a very good child; you're an increasingly good child.

In binary search on an array,

- the worst-case running time is in O(log n),
- the best-case running time is in O(1),
- the memory use is in O(n), and
- 47 + 18 log n - 3/n is in O(the worst-case running time).

Every semester, a few students get the wrong idea that "big-Oh" always means "worst-case running time." Their brains short out when an exam question uses it some other way.

[3] "e^3n is in O(e^n) because constant factors don't matter."
"10^n is in O(2^n) because constant factors don't matter."
-> WRONG! I said that Big-Oh notation doesn't care about (most) constant factors. Here are some of the exceptions. A constant factor in an exponent is not the same as a constant factor in front of a term. e^3n is not bigger than e^n by a constant factor; it's bigger by a factor of e^2n, which is damn big. Likewise, 10^n is bigger than 2^n by a factor of 5^n.

[4] Big-Oh notation doesn't tell the whole story, because it leaves out the constants. If one algorithm runs in time T(n) = n log_2 n, and another algorithm runs in time U(n) = 100 n, then Big-Oh notation suggests you should use U(n), because T(n) dominates U(n) asymptotically. However, U(n) is only faster than T(n) in practice if your input size is greater than current estimates of the number of subatomic particles in the universe. The base-two logarithm log_2 n < 50 for any input size n you are ever likely to encounter.

Nevertheless, Big-Oh notation is still a good rule of thumb, because the hidden constants in real-world algorithms usually aren't that big.