Index of /~cs61b/lectures/17

Name	Last modified	Size	Description
Parent Directory	24-Sep-98 21:22	-	

CS61b: Lecture 17
Friday, October 2

ANNOUNCEMENTS
=============

Skip Secton 5.3 of Weiss unless you are very ambitious. Do read the other sections of Chapter 5.

ASYMPTOTIC ANALYSIS
===================

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 large.

We also know that these numbers 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 running time or memory)

Big-Oh notation tells us how badly a program might perform as its input grows.

Let n be the size of a program's _input_ (in bits or data words or whatever).
Let T(n) be the function precisely equal to the algorithm's worst-case (maximum possible) running time, given an input of size n.
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"? As big as necessary to make T(n) fit under c f(n).
 * HOW BIG IS c? As big as necessary 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.

```
                    c f(n) = 20 n     **
           ^               /        **
           |       |      /       **
           |       |     /      **
           |       |    /     **
```

```
           |       |    /    **  T(n) = 10,000 + 10 n
   30,000  |       |   /   **
           |       |  /  **
           |       | / **
           |       |/**
   20,000  |       **
           |     **|
           |   **/ |
           | ** /  |
   10,000  **  /   |
           |  /    |
           | /     |
           |/      |
           O--------------------------------> n
                 1000
```

As these functions extend forever to the right, their asymptotes will never cross again. For large n -- any n bigger than 1000, 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 = 1000.

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 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 they shoot 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 constant factors.
     We generally leave constants out; it's unnecessary to write O(2n).

[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
     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 the running time or memory use.

```
              c f(n) = n^3
           ^        *      /
           |        *     /
           |        *    / T(n) = n
           |        *   /
           |        *  /
           |        * /
           |       * /
           |       */
       1   |       *
           |      /*
```

```
 |     / *
 |    / *|
 |   /  *|
 |  /   *|
 | /   * |
 |/  **  |
 O***-----------------------------> n
       N = 1
```

[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 running time. 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

Algorithms that are O(n log n) or faster are considered efficient. Algorithms that take n^4 time or more are usually considered useless. In the regions between n log n and n^4, 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 Section 5.5 of Weiss carefully. Computer scientists need to know logarithms in their bones.

Avoid These Pitfalls

[1] 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] 0.0000001 n^2 is in O(n), because if we choose n = 1,000,000, we get 100,000 <= 1,000,000, and it's clear that 1,000,000 is BIG.
  -> WRONG! The inequality does not hold for ALL n >= 1,000,000, so the conclusion is wrong. You have to choose n even larger to reveal the fallacy. For a correct result, we must choose N big enough so that the asymptotes of the two functions never cross to the right of N.

[3] Big-Oh notation can be misleading because it leaves out the constants. If one algorithm runs in time T(n) = n log 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) if your input size is roughly equal to current estimates of the number of subatomic particles in the universe. T(n) is always better in practice, because log n < 50 for any input size you are ever likely to

encounter. Nevertheless, Big-Oh notation is still a good rule of thumb, because the hidden constants don't vary that greatly in most algorithms regularly encountered in software development.