ASYMPTOTIC ANALYSIS 4

GEORGE WANG
gswang.cs61a@gmail.com
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley

June 24, 2010

1 Review from Last Lecture

1.1 CS Illustrated

Functions as Arguments: http://csillustrated.berkeley.edu/PDFs/functions-as-data.pdf.

1.2 Procedures We Learned[2]

```
lambda
let
keep
```

2 Motivation

Okay! We're moving from talking about functions to kind of mentally taking a few steps back and talking about how to talk about functions. In particular, although most of this course we're focused on writing programs that work, we'll be talking about being able to judge the efficiency of a program today and tomorrow.

First, let's think about this problem. The obvious way to test the quality of a program is to just run the darn thing, right? Just break out a stopwatch, and set the timer, and see how long it takes to run. But the problem with this is to think of all the constraints of reality we've talked about. What if you have other programs running on your computer in the background? What if your CPU kind of sucks, and it's really old? Well, what we want to do, is divorce the hardware that you're running these programs on with the abstract software programs that you are writing.

[2]See Lecture 3 notes for a more detailed treatment, including syntax.

2.1 An Example[4]

Suppose you're writing a program to manage the inventory for a retail store. Suppose that it takes **10,000ms** to load the initial inventory, and **10ms** to process each transaction.

Well, if we're talking about a small number of transactions, say < 1000, then it's pretty straightforward to see that the initial load time is what takes a long time. However, note that we're only talking about *20 seconds* total here. We don't really care about the differences here when it's not such a big deal.

However, if we're talking about a large number of transactions, say $> 1,000,000$, then the initial load time becomes massively outweighed by the time per transaction. For comparison, with 1 million transactions, the time to process the transactions is about 3 hours, whereas the 10 second load time really doesn't matter much.

Thus, whatever we use to think about runtimes of programs must be *more focused on large problems*, and less focused on smaller problems.

3 Big-O Notation

Essentially, we're looking for ways to divide up runtimes of functions. It's a lot like boxing. You have featherweights, welterweights, and all the way up to super heavyweights. The idea is that although within each class you have variability, you have huge differences between classes than within classes.

This concept can be a little tough to understand, so I'm going to try to explain it 4 ways. As long as you get one of them, you should be okay. You will need to be able to classify functions into a specific class, as we'll explain in a bit. The big idea of big O is that it is an *upper bound*.

3.1 An Intuitive Definition

Hi, I'm George, I'm not all that fast. I get to race Usain Bolt, who does happen to be very fast. What if we start imposing handicaps?

What if for every n meters I run, he has to run $1.5n$ meters? Well, if Usain Bolt is twice as fast as me, then it doesn't matter what n is. I'll never win!

But what about a different handicap? For every n meters that I run, Usain Bolt has to run n^2 meters. Maybe he can beat me if n is small, like say 3 or less, but if n is big, say 100, he has no chance. This is because he's running 10km to my 100m.

Thus, we're talking about running *very* far. If the handicap allows me to win at very far distances, than the function for my handicap fits in big-O of the bigger handicap.

3.2 The Algebraic Definition

Let f and g be functions. $f \in O(g)$ if and only if there exists c and N such that for all $n > N$, $cf(n) < g(n)$.

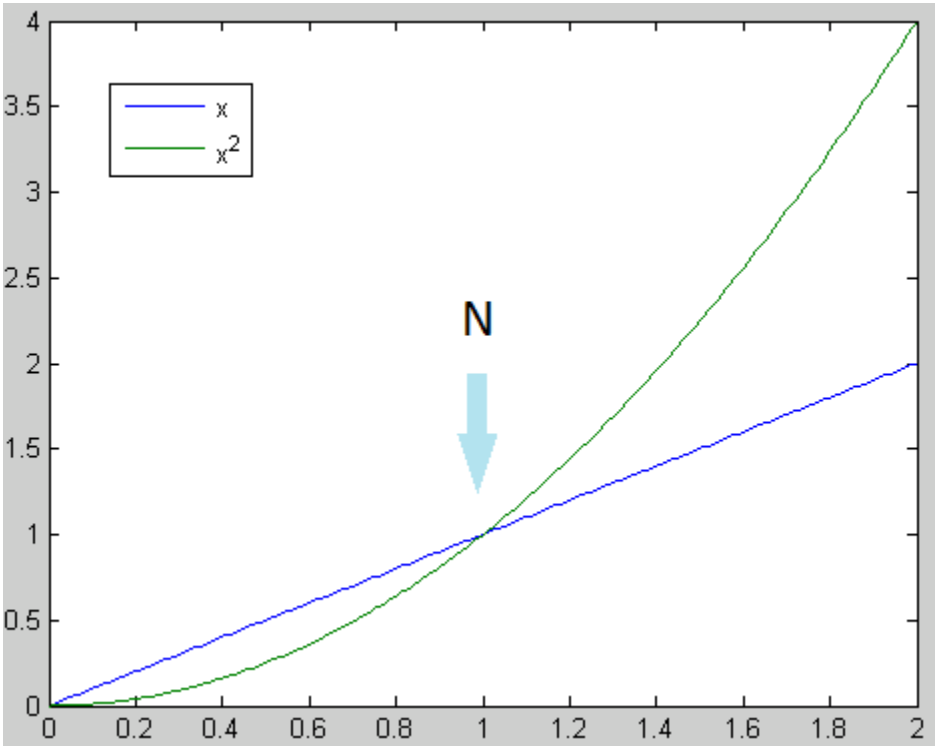
Let's look at this. Basically, this is saying that even if you blow up f by multiplying it by a million, we can pick an N big enough after which it doesn't matter.

Again, looking at our racing example, let's try this. $f(n) = n$. $g(n) = n^2$. Let's pick $c = 10$ and $N = 1,000$. Well, we can see that $10f(1000) = 10 \times 1000 = 10,000$, but $g(1000) = 1,000,000$. Thus, we've found the constants c and N to make this work. Thus, we can say $f \in O(g)$.

[4]This section adapted from Jonathan Shewchuk's lecture notes.

3.3 The Graphical Definition

Let's look at this graphically.



Note that although there's a region during which the green line is lower than the blue line, after some point, the blue line is going to be *always* less than the green line. Here, we can say that $x \in O\left(x^2\right)$. And so long as f is less than g for *some* region to the right of some point, we can say that $f \in O\left(g\right)$.

3.4 The Calculus Definition

Ever heard of the ratio test? It deals with whether something converges or not, right? We can adapt it to test whether some things are bigger than other things. Essentially, we can say $f \in O\left(g\right)$ if and only if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C$, where C is some constant, including 0.

4 Big Omega and Big Theta

4.1 Big Omega

We said that big O is the upper bound. There turns out to be something that means a lower bound, it's called Omega, written Ω. If $f \in \Omega\left(g\right)$, then $g \in O\left(f\right)$. Get it? In other words, if f is always bigger than g past a point, then conversely, g is always smaller than f past a point.

It's strict definition is written:

Let f and g be functions. $f \in \Omega\left(g\right)$ if and only if there exists c and N such that for all $n > N$, $f\left(n\right) > cg\left(n\right)$.

Notice how I flipped the greater-than sign for this?

4.2 Big Theta

Okay, so this should be coming. You have an upper bound, and you have a lower bound. Can we get an exact bound? Turns out, we can! If $f \in O(g)$ AND $g \in O(f)$, then $f \in \Theta(g)$ and $g \in \Theta(f)$. The way to think about them is that *up to a constant factor, $f(n)$ and $g(n)$ are equal.*

5 Why This Matters

Why don't we care about small values of n? Because for small inputs, your program will be fast enough anyway. Let's say one program is 1000 times faster than another, but one takes a millisecond and the other takes a second. Big deal.

Why don't we care about constant factors? Because for large inputs, the constant factor will be drowned out by the order of growth—the exponent in the $O(x^i)$ notation. Here is an example taken from the book Programming Pearls by Jon Bentley (Addison-Wesley, 1986). He ran two different programs to solve the same problem. One was a fine-tuned program running on a Cray supercomputer, but using an $\Theta(n^3)$ algorithm. The other algorithm was run on a Radio Shack microcomputer, so its constant factor was several million times bigger, but the algorithm was $\Theta(N)$. For small n the Cray was much faster, but for small n both computers solved the problem in less than a minute. When n was large enough for the problem to take a few minutes or longer, the Radio Shack computer's algorithm was faster.

```
;;;;; In file cs61a/lectures/1.2/bentley
            t1(N) = 3n^3          t2(N) = 19,500,000n
      N     CRAY-1 Fortran        TRS-80 Basic
     10     3.0 microsec          200 millisec
    100     3.0 millisec          2.0 sec
   1000     3.0 sec               20 sec
  10000     49 min                3.2 min
 100000     35 days (est.)        32 min
1000000     95 yrs  (est.)        5.4 hrs
```

6 Common Functions[6]

	Function		Common Name	Commonly found in
	$O(1)$		constant	Searching
is a subset of	$O(\log n)$		logarithmic	Searching
is a subset of	$O(\sqrt{n})$		root-n	
is a subset of	$O(n)$		linear	Searching, Sorting
is a subset of	$O(n \log n)$		n log n	Sorting
is a subset of	$O(n^2)$		quadratic	Sorting
is a subset of	$O(n^3)$		cubic	Matrix Multiplication
is a subset of	$O(2^n)$		exponential	NP
is a subset of	$O(e^n)$		exponential	NP

[6] This section adapted from Jonathan Shewchuk's lecture notes.

7 Efficiency Improvements[8]

```
;;;;; In file cs61a/lectures/1.2/pascal.scm
(define (pascal row col)
  (cond ((= col 0) 1)
        ((= col row) 1)
        (else (+ (pascal (- row 1) (- col 1))
                 (pascal (- row 1) col) ))))
```

This program is very simple, but it takes $\Theta(2^n)$time! Try some examples. Row 18 is already getting slow.

Instead we can write a more complicated program that, on the surface, does a lot more work because it computes an *entire row* at a time instead of just the number we need:

```
;;;;; In file cs61a/lectures/1.2/pascal.scm
(define (new-pascal row col)
  (nth col (pascal-row row)) )
(define (pascal-row row-num)
  (define (iter in out)
    (if (empty? (bf in))
        out
        (iter (bf in) (se (+ (first in) (first (bf in))) out)) ))
  (define (next-row old-row num)
    (if (= num 0)
        old-row
        (next-row (se 1 (iter old-row '(1))) (- num 1)) ))
  (next-row '(1) row-num) )
```

This was harder to write, and seems to work harder, but it's far faster because it's $\Theta\left(n^2\right)$. The reason is that the original version computed lots of entries repeatedly. The new version computes a few unnecessary ones, but it only computes each entry once. Moral: When it really matters, think hard about your algorithm instead of trying to fine-tune a few microseconds off the obvious algorithm.

Note: Programming project 1 is assigned tomorrow, and due Friday, July 2nd, at noon. Although you have slip hours, I strongly recommend you reserve them for later projects.

[8]For those using Brian Harvey's notes, we're skipping recursive v. iterative processes