MATH 22

Lecture O: 10/21/2003

FUNCTIONS: ORDER OF GROWTH

The old order changeth,
yielding place to new.
—Tennyson, *Idylls of the King*

Men are but children of a larger growth.
—Dryden, *All for Love*, Act 4, scene 1

Administrivia

- http://denenberg.com/LectureO.pdf
- **Exam: Monday 10/27, Robinson 253. Anyone know when?**
- Suggestions for review questions still accepted
- Comment on the function $\sin^2\theta$
- Project 4: Grader pounds me, I pound you:
  This is your last warning!
  - You can't manipulate non-equations.
  - "if p, then q" is always true when p is false.

Today: Growth rates of functions, preceded by awesome motivational discussion about problems and their algorithms

Problems & Algorithms

Today we study *growth rates of functions*. Why?

A *problem* is something that we solve with an *algorithm*. A problem consists of *instances*; an algorithm for the problem accepts an instance as input. Examples:

SORTING is the problem of putting numbers in order.
An *instance* of **SORTING** is a sequence of numbers.
An algorithm that solves **SORTING** takes such a sequence as input and outputs the sorted sequence.

FACTORING is the problem of finding prime factors.
An *instance* of **FACTORING** is a single number.
An algorithm that solves **FACTORING** takes a number *N* as input and outputs the factors of *N*.

VALIDITY is the problem of determining whether a propositional formula, e.g. $pq \quad (\neg r) \rightarrow q(r \quad s)$, is valid, that is, true for any interpretation of its variables.
An *instance* of **VALIDITY** is a propositional formula.
An algorithm that solves **VALIDITY** takes such a formula as input, and outputs either "yes" or "no".

(The Twin Prime Conjecture is *not* a problem since it doesn't have instances. No individual question is a problem in this "computer science" sense.)

Problem Complexity

Some problems seem to be harder than others: **SORTING** seems easier than **FACTORING**, but harder than **MAXIMUM** (i.e., finding the largest number of a sequence). But how do **FACTORING** and **VALIDITY** compare? How do we make these ideas precise?

Answer: We define the "difficulty" of a problem H, which we call the *computational complexity of H*, as the *difficulty of the best algorithm that solves H*.

Pretty lousy definition, eh? It just raises (*not* "begs"!!) the question: What is the difficulty of an algorithm? And what is "best", that is, how do we compare algorithm difficulties? Today we take up tools we need to answer these questions.

[Note: It's usually easy to measure the difficulty of an algorithm. But how do we know when you've got the best algorithm for a problem? We must prove that no better algorithm exists, and this is often very hard. There are many problems for which we're unsure of the best algorithm, including **FACTORING** and **VALIDITY**!]

Algorithm Complexity

The complexity of an algorithm is defined by the amount of resources it uses. Usually the resource we care about is *time*: Difficult problems take longer to solve.

How can we measure the time used by an algorithm? The time depends on the particular machine and the efficiency of the coding. Most critically, it depends on the particular instance of the problem! (It usually takes more time to sort 1000000 numbers than to sort 10000. But it might take *less* time if the 1000000 numbers happen to be in order already, as they could be.)

Here's how we cope with these issues:

0. We don't measure time in seconds, but rather in basic operations or program steps that are approximately equivalent and generally computer-independent. We'll see more on this point next lecture.

1. Every instance of a problem gets assigned a *size*. For example, the size of an instance of **SORTING** might be the number of numbers to be sorted; for an instance of **FACTORING** it might be the number of digits in the number to be factored. Having assigned sizes . . .

Complexity, continued

2. We measure the time an algorithm takes as a *function of the size of the input*. That is, an algorithm's complexity is not a *number*, but a *function*. If A is an algorithm, its complexity is a function $f_A(n)$ that tells how much time A requires on instances (inputs) of size n.

3. But algorithms usually take varying amounts of time on different inputs, even of the same size. (Some 1000-digit numbers are easier to factor than others!) So we define the complexity function to measure *worst-case* performance. That is, the *complexity of algorithm A* is a function f_A such that $f_A(n)$ is the greatest possible amount of time required for A to solve a problem instance of size n. (Sometimes we use *average case*.)

4. We don't worry about time used on "small" inputs, which take little time anyway—we can always handle any finite number of instances as special cases. So we compare complexities only "at infinity", that is, for large inputs, though we can concoct cases where this is absurd.

5. Constant factors don't matter: Two algorithms whose time differs by a factor of 2 (say) are the same. Special-case hardware or new technology can always increase speeds by constant factors, so ignoring them makes us technology-independent. (This can also be absurd.)

The Story of *O*

With these ideas in mind we make the following definition: Let g be a function from positive numbers (e.g. $\underline{\boldsymbol{Z}}^+$ or $\underline{\boldsymbol{R}}^+$) to positive numbers. We define $O(g)$ to be the set of all functions f for which the following condition holds: there exists some integer N and some positive constant c such that $f(n) \le cg(n)$ for all $n > N$. It's our job today to understand this completely.

First: For f to be in $O(g)$ it's *not required* that $f(n) \le g(n)$, just that $f(n) \le cg(n)$, where c is a constant that we can make as large as we like. For example, if $g(n) = 2n$ and $f(n) = 800n$, then f is in $O(g)$, since we can pick $c = 500$ to boost g up. (We'll write this as $2n \in O(800n)$.)

Second: It's also *not necessary* that $f(n) \le cg(n)$ for *all* n. It's enough for cg to exceed f for those n larger than some specific N which we can pick as large as we please. For example, n^{20} is in $O(2^n)$ even though n^{20} is bigger for $n \le 143$. Once $n \ge 144$ (and forever after), 2^n wins.

Loosely, you can think of $O(g)$ as *the set of all functions less than or equal to g* in the sense of "functional comparison" that we said we'd use for algorithm complexity: consider only behavior at infinity and ignore constant factors.

Examples

Example: As you showed in Project 4, $4^n \in O(n!)$ because for all $n > 10$ we have $4^n < n!$. Here we've used the definition with $N = 10$ and $c = 1$.

Example: As we said, not only is $2n$ obviously in $O(800n)$, but also $800n \in O(2n)$. Similarly, we have $5n^3 \in O(2n^3)$ and vice versa. In general, we can *ignore multiplicative constants* when deciding whether $f \in O(g)$ because we can always boost g by picking a bigger c. That is, if $f \in O(g)$ then $k_1 f \in O(k_2 g)$ for any positive constants k_1 and k_2, no matter how big or small. (This shouldn't be surprising; we defined it that way!)

Example: $n^2 \in O(n^3)$, and indeed $n^a \in O(n^b)$ for any $a \le b$. Other examples: $n \in O(n)$, $n^4 \in O(n^{4.1})$.

Example: $n^k \in O(b^n)$ for any k and any $b > 1$. This is what we mean by saying that *any exponential function eventually exceeds any polynomial*. But it only happens "at infinity"; you must choose a *very* large N to have $n^{100000000000000} \le 1.0000000000000001^n$ for all $n > N$.

Negative Results

As it happens, $n^3 \notin O(n^2)$. How can we prove this? Back to basic quantifier logic! $f \notin O(g)$ means

$$(\exists N)(\exists c)(\exists n)\ \ n \geq N \ \ \exists \ \ f(n) \leq cg(n)$$

Negating this statement, as you surely know, yields

$$(\forall N)(\forall c)(\forall n)\ \ n \geq N \ \ \forall \ \ f(n) > cg(n)$$

That is, $f \notin O(g)$ means that no matter how big N and c, there's some still-bigger n such that $f(n)$ exceeds $g(n)$ even when $g(n)$ is boosted up by a factor of c.

Let's prove that $n^2 \notin O(n)$. We must prove that for any N and c there's some $n > N$ such that $n^2 > cn$. So given N and c, can we find such an n? Pick $n > c$ and it's certainly true! Of course n must also exceed N. So the proof starts like this: given N and c, let $n = max(N,c)+1$. We can similarly prove that $n^a \notin O(n^b)$ for any $a > b$.

Remember: To prove $f \notin O(g)$, *you* get to find N and c to make $f(n) \leq g(n)$ for **all** $n > N$. But to prove $f \notin O(g)$ you have to show how to find, for any N and c picked by *someone else*, **at least one** $n > N$ that makes $f(n) > cg(n)$. Knowing how to use these definitions will enhance your enjoyment of the upcoming exam!

General Results

For any function f, $f \in O(f)$. [Homework problem.]

If f_1 and f_2 are both in $O(g)$, then the function $f_1 + f_2$ is in $O(g)$. Sketch of proof: Since $f_1 \in O(g)$ there exists an N_1 and a c_1. . . Then $f_2 \in O(g)$ gives us an N_2 and a c_2. . . Now pick $N = max(N_1,N_2)$ and $c = c_1+c_2$ to finish.

If $f \in O(g)$ and $g \in O(h)$, then $f \in O(h)$. Said another way, if $g \in O(h)$, then $O(g) \subseteq O(h)$.

$O(f) = O(g)$ if and only if both $f \in O(g)$ and $g \in O(f)$. [Another homework problem. Do we understand it?]

As we said, $f \in O(g)$ loosely means "g is at least as big as f." But beware of being too simplistic; functions are more complicated than numbers and can wiggle around in tricky ways. Another homework problem gives you specific functions f and g and asks you to prove that $f \in O(g)$ *and* $g \in O(f)$! A blackboard picture will give you the intuition behind this. Problem: Find two such functions f and g that are monotonically increasing.

A Useful Result

Suppose we have several functions f_1, f_2, f_3, . . . , f_n, and one of them, say f_1, is the "biggest" in the sense that f_2, f_3, . . ., f_n are all in $O(f_1)$. Then we know by a result on the preceding page that $f_1 + f_2 + f_3 + \ldots + f_n$ is in $O(f_1)$. And certainly $f_1 \in O(f_1 + f_2 + f_3 + \ldots + f_n)$.
Therefore, by another result on the preceding page,

$$O(f_1) = O(f_1 + f_2 + f_3 + \ldots + f_n) \text{ if each } f_i \in O(f_1)$$

The bottom line here is that we can ignore all terms of a sum except the biggest. So, for example,

$$O(10n^4 + 50n^3 + 2n^2 + 8n + 10000) = O(n^4)$$

where we've also ignored the multiplicative constant 10.
That is, $O(\textit{any polynomial}) = O(\textit{its largest power})$.

Another example:

$$O(10n^4 + 500n^{3.99999} + 2n^{2.5} + 8n + 100n^{-1}) = O(n^4)$$

Some Mathematics

For any positive a and b, $O(\log_a n) = O(\log_b n)$. In particular, $O(\log n) = O(\lg n) = O(\ln n)$. This is true because changing the base of logarithms is just multiplication by a constant factor. Learn this.

Reminder: $n^k \notin O(b^n)$ for any k and any $b > 1$; any exponential *eventually* grows faster than any polynomial. Taking logs, we find that $\log n \notin O(n)$. Indeed, we have $\log^k n \notin O(n^b)$ for any k no matter how big and any $b > 0$. This means, e.g., that $\log^{100} n \notin O(\sqrt{n})$.

Although you can ignore overall multiplicative constants, you can't ignore them everywhere! For example, 2^{2n} is *not* in $O(2^n)$; constant factors in the exponent are not ignorable. Similarly, $O(2^n) \neq O(3^n)$.

What does $O(1)$ mean? By definition, function f is in $O(1)$ if there are N and c such that $f(n) < c$ for all $n > N$. That is, the functions in $O(1)$ are those *bounded by a constant*; the functions that never get to infinity at all! What would $O(70)$ mean?

Other Sets

$O(g)$ is the set of functions "no bigger than g". The set of functions "at least as big as g" is written $\Omega(g)$; formally, f is in $\Omega(g)$ if there exist N and $c > 0$ such that $f(n) \geq cg(n)$ for all $n > N$. As homework, you're going to prove that $f \in \Omega(g)$ if and only if $g \in O(f)$.

If we take the functions that are both "no bigger than f" and "at least as big as f" we get the set of functions that have *the same order of growth* as f. This set is written $\Theta(f)$ and is defined as $O(f) \cap \Omega(f)$. Knowing the exact order of growth of a problem is the typical goal.

We also have notation for functions that are "definitely smaller than g": We say that $f \in o(g)$ if for all $c > 0$ there exists N such that $f(n) < cg(n)$ for all $n > N$. That is, no matter how small you pick c, eventually the ratio $f(n)/g(n)$ is $< c$; no constant factor can build f up to g.

Analogously, $\omega(g)$ is the set of functions that are "definitely bigger than g"; you can guess the definition.

It's very instructive to study the distinction between "$f \in O(g)$" and "$f \in \Omega(g)$"; these are not the same! The latter is $(\forall c)(\exists N)(\forall n)$; the former is $(\exists c)(\exists N)(\forall n)$. Again, this is because functions are complicated.

Two G Warnings

Warning #1: Grimaldi uses the phrase "g dominates f" to mean f $\in$ O(g). So he would say, for example, that any function f dominates itself, or that p(n) = 0.00001n dominates q(n) = 10000000n.

Not everyone uses this terminology, myself in particular. By "g dominates f" I would mean f $\in$ o(g), that is, f is definitely smaller than g in the limit. Functions p and q above have the same order of growth (namely linear, i.e., both are in Θ(n)), but neither dominates the other.

Warning #2: All functions in this lecture have been assumed to be *from* positive numbers *to* positive numbers. Grimaldi's presentation permits negative numbers as well, so his definition is full of absolute values and disclaimers about dividing by zero: He defines O(g) as the set of functions f for which there exist N and c such that $|f(n)| < c|g(n)|$ for all $n > N$ where $g(n) \neq 0$. This means that f can't exceed g in absolute value, i.e., it can't go hugely negative below g.

Final Fun Facts

There are all sorts of ways to slip orders of growth between each other. For example, if f(n) = $n^4 \log n$, then f is in $\Theta(n^4)$ but is in $o(n^{4.000000000000001})$.

As another example, consider f(n) = $n^{\log n}$. This function is $\Omega(n^k)$ for any k no matter how big, but is in $o(b^n)$ for any $b > 1$. That is, it's strictly between the polynomials and the exponentials. Can you find a function that lies strictly between $\log^k n$ and n^b for huge k and tiny b?

Sometimes the notation O(f) is used to mean "some function in O(f)". For example, we might write

$$g(n) = n^4 + O(n^3)$$

to mean that the difference between g(n) and n^4 is some function that never gets bigger than n^3. (Is it the same thing to write $g(n) = n^4 + o(n^4)$? Which is stronger?) Also, older books write f = O(g) to mean f $\in$ O(g).

The facts concerning the problems we used as examples: MAXIMUM is $\Theta(n)$, SORTING is $\Theta(n \log n)$. Nobody knows the complexity of FACTORING, but it was just proven a year ago that just testing a number to see if it is prime is *polynomial*, that is, it's in $\Theta(n^k)$ for some k. Whether VALIDITY is polynomial is the celebrated "P=NP?" question, still the most important unsolved problem in computer science.