Asymptotic Order Notation

Burton Rosenberg

September 10, 2007

Introduction

We carefully develop the notations which measure the asymptotic growth of functions. The intuition is to group functions into function classes, based on its "growth shape". The most used such grouping is $O(f)$ ("big-oh of f"), where a function f serves as a model for an entire collection of functions, the set $O(f)$, including all functions growing not faster than the function f.

Definition 1 (Big-Oh) *Let $f : X \to Y$ be a function.*

$$O(f) = \{g : X \to Y \mid \exists\, x_o, c > 0 \text{ s.t. } \forall x \geq x_o, cf(x) \geq g(x) \geq 0\}$$

By abuse of notation, $g \in O(f)$ is written $g = O(f)$, rather than the more correct $g \in O(f)$. The phrase "$O(f) = g$" has no meaning. In general, the sets X and Y are the sets of real numbers, integers or naturals, according to the situation.

Computer science is interested in functions which give only positive values, or less drastically, functions which eventually assume only positive values. There is some long-term simplicity in adopting this restriction, although short-term it requires additional definitions and care.

Definition 2 (Eventually Non-negative, Positive) *A function $f : X \to Y$ is eventually non-negative if there exists an x_o such that for all $x \geq x_o$, $f(x) \geq 0$. The function is eventually positive if there exists an x_o such that for all $x \geq x_o$, $f(x) > 0$.*

Theorem 1 *If f is not eventually non-negative, $O(f)$ is empty. If f is eventually non-negative, then $0 = O(f)$, and $O(f)$ is therefore non-empty. Every element of $O(f)$ is eventually non-negative.*

The function $0 : X \to Y$ is the zero function, defined as $0(x) = 0$ for all x. In general any constant C can be viewed as a function of constant value C.

There is a related but stronger notion for the classification of functions, "Little-Oh of f". Whereas a function in $O(f)$ is bounded above by the function f, a function in $o(f)$ is increasingly insignificant to the function f.

Definition 3 (Little-Oh) *Let $f : X \to Y$ be a function.*

$$o(f) = \{\, g : X \to Y \,|\, \forall c > 0, \exists x_o > 0 \text{ s.t. } \forall x \geq x_o,\ cf(x) > g(x) \geq 0 \,\}$$

Theorem 2 *If f is not eventually positive, $o(f)$ is empty. If f is eventually positive, then $0 = o(f)$, and $o(f)$ is therefore non-empty. Every element of $o(f)$ is eventually non-negative.*

Note that while $O(0)$ is not empty (it contains all functions that are "eventually zero"), $o(0)$ is empty. So the converse of the following theorem is not true.

Theorem 3 (Little-Oh is Stronger than Big-Oh) *If $g = o(f)$ then $g = O(f)$.*

Proving a function in $O(f)$ requires demonstrating x_o and c satisfying the definition. For some demonstrations, the function is so strongly a member of $O(f)$ that the subtleties of Big-Oh are a waste of time. The class of functions $o(f)$ ("little-oh of f") has stronger demands, hence simpler demonstrations. A function in $o(f)$ is also in $O(f)$, so one use of Little-Oh is to provide demonstrations for Big-Oh.

Theorem 4 (Analytic Definition of Little-Oh) *Let $f : X \to Y$ be eventually positive. Let $g : X \to Y$ be eventually non-negative. Then $g = o(f)$ if and only if:*

$$\lim_{x \to \infty} g(x)/f(x) \to 0$$

Example 1 $\log x = O(x)$. *Use L'Hopital's rule to show* $\log x/x \to 0$. *This gives* $\log x = o(x)$, *and therefore* $\log x = O(x)$. *In the same way, it can be shown that* $n \log n = O(n^{1+\epsilon})$ *for any real* $\epsilon > 0$.

Example 2 *For $j' > k > 0$, $x^k = o(x^j)$, hence $x^k = O(x^j)$.*

Order Properties of the Notation

The order notation arranges functions from smaller to larger, reminiscent of the ordering of numbers from smaller to larger. In this analogy, Big-Oh is non-strict inequality, $g = O(f)$ can be interpreted as $g \leq f$. Likewise, Little-Oh is strict inequality, $g = o(f)$ is similar to $g < f$. Given three functions, f, g and h, we expect that ordering is transitive: if $h \leq g$ and $g \leq f$ then $h \leq f$. The is indeed true for order notation.

Theorem 5 (Transitivity) *If $h = O(g)$ and $g = O(f)$ then $h = O(f)$. If $h = o(g)$ and $g = o(f)$ then $h = o(f)$.*

Big-Oh and Little-Oh express non-strict and strict "less-than". Other notations express non-strict and strict "greater-than".

Definition 4 (Big-Omega) *Let $f : X \to Y$ be a function.*

$$\Omega(f) = \{\, g : X \to Y \mid \exists x_o, c > 0 \text{ s.t. } \forall x \geq x_o,\ g(x) \geq cf(x) \geq 0 \,\}$$

Theorem 6 *If f is not eventually non-negative, $\Omega(f)$ is empty. Every function in $\Omega(f)$ is eventually non-negative. $\Omega(0)$ is the collection of all eventually non-negative functions.*

Definition 5 (Little-Omega) *Let $f : X \to Y$ be a function.*

$$\omega(f) = \{\, g : X \to Y \mid \forall c > 0, \exists x_o > 0 \text{ s.t. } \forall x \geq x_o,\ g(x) > cf(x) \geq 0 \,\}$$

Theorem 7 *If f is not eventually non-negative, $\omega(f)$ is empty. Every function in $\omega(f)$ is eventually positive. $\omega(0)$ is the collection of all eventually positive functions.*

We claim that these definitions are the appropriate anti-symmetric variants of Big and Little-Oh.

Theorem 8 (anti-symmetry) *$g = O(f)$ exactly when $f = \Omega(g)$. $g = o(f)$ exactly when $f = \omega(g)$.*

Given the anti-symmetry, theorems concerning Big and Little-Oh can be quickly reestablished for Big and Little-Omega.

Corollary 9 (Strength) *If $g = \omega(f)$ then $g = \Omega(f)$.*

Corollary 10 (Transitivity) *If $h = \Omega(g)$ and $g = \Omega(f)$ then $h = \Omega(f)$. If $h = \omega(g)$ and $g = \omega(f)$ then $h = \omega(f)$.*

The following theorem illustrates the strictness versus non-strictness of the various notations.

Theorem 11 (Reflexivity) *Let f be eventually non-negative. Then $f = O(f)$ and $f = \Omega(f)$. Hence the intersection is non-empty. In contrast, the intersection of $o(f)$ and $\omega(f)$ is always empty. In particular, neither $o(f)$ nor $\omega(f)$ contains f.*

The intersection of $O(f)$ and $\Omega(f)$ produce the class $\Theta(f)$, the class of functions sharing the asymptotic growth of f. Just as $x \leq y$ and $y \leq x$ implies $x = y$, $g = O(f)$ and $f = O(g)$ implies $g = \Theta(f)$ (since $f = O(g)$ can be transformed to $g = \Omega(f)$, so g is in the intersection).

Definition 6 (Theta) *Let $f : X \to Y$ be a function,*

$$\Theta(f) = \{\, g : X \to Y \mid \exists x_o, c_1, c_2 > 0 \text{ s.t. } \forall x \geq x_o,\ c_1 f(x) \geq g(x) \geq c_2 f(x) \geq 0 \,\}$$

Theorem 12 *$g = \Theta(f)$ if and only if $g = O(f)$ and $f = O(g)$. The set $\Theta(f)$ is the intersection of $O(f)$ and $\Omega(f)$.*

The sets defined by Theta form equivalence classes. Equivalence classes are defined by a relation which is reflexive, symmetric and transitive.

Corollary 13 (Equivalence Properties) *The relation Θ enjoys the following properties:*

1. *Reflective: If f is eventually non-negative, $f = \Theta(f)$;*
2. *Symmetry: If $g = \Theta(f)$ then $f = \Theta(g)$;*
3. *Transitive: If $h = \Theta(g)$ and $g = \Theta(f)$ then $h = \Theta(f)$.*

We close this section with a warning concerning the ordering of functions. Functions have greater complexity than numbers. The ordering of numbers by size leaves little room for variety. Given two numbers, one is the smaller, or if not, the numbers are the same. Consider these functions:

$$f_e(x) = \begin{cases} 0 & \text{if } x \text{ is even} \\ 1 & \text{else} \end{cases}$$

$$f_o(x) = \begin{cases} 0 & \text{if } x \text{ is odd} \\ 1 & \text{else} \end{cases}$$

Neither of these functions is Big-Oh the other. By order of growth, these two functions are incommensurable. (But see an exercise in the class text for variant definitions which avoid this.)

Example 3 *Show $ax+b = O(x)$ and $x = O(ax+b)$ for $a > 0$. Conclude that $a_f x+b_f = \Theta(a_g x+b_g)$ for $a_f, a_g > 0$. Often the "simplest" function of a Θ class lends its name to the class. These functions would be described as $\Theta(x)$.*

The Order of an Approximation

The order notation can be used to notate a function whose shape is not known exactly, but to an approximation suitable for the purpose. The Harmonic series is the sum of the reciprocals of first n integers. Since $\log x$ is the integral of $1/x$, it is not surprising that the Harmonic series should approximate the log.

$$H_n = \sum_{i=1}^{n} 1/i = \ln n + O(1)$$

This equation tells us that the approximation is perfect with the addition of an unknown $O(1)$ error function. That is,

$$H_n = \ln n + e(n) \text{ for some } e(n) = O(1)$$

In general, the presence of the order notation inside an expression on the right hand side of an equality indicates that some specific function makes the equality true, however the most we know or care to know about that function is that it is a member of a certain order class.

Efficiency of Algorithms

The efficiency of an algorithm is given by the order of growth in run time with respect to input size. For instance, numerous simple sorting routines essentially compare all elements pairwise in the course of sorting. For n elements, this makes $O(n^2)$ comparisons, and this places a bound on runtime.

Suppose two algorithms are composed. The output of algorithm A with run time f_A is the input to algorithm B with run time f_B. What is the efficiency of the combined algorithm? Assuming the first algorithm leaves the size unchanged, it is $f_A(x)+f_B(x)$. This theorem allows for simplification.

Theorem 14 *Let $g_1, g_2 = O(f)$ and a_1, a_2 be non-negative constants. Then $a_1 g_1 + a_2 g_2 = O(f)$.*

Suppose we would like to calculate the mode of x numbers, that is, the value which occurs most often. This can be accomplished by sorting the numbers and then sweeping through the sorted numbers keeping track of the longest run of consecutive, equal values. If the sorting is done in time $f_A(x) = O(x^2)$, and the sweep is done in time $f_B(x) = O(x)$, the entire algorithm runs in time $f_A(x) + f_B(x) = O(x^2)$.

In this algorithm, sorting is the bottle neck. Improve the sort to $O(x \log x)$ (e.g. using merge-sort) and immediately the entire algorithm improves to $O(x \log x)$. The moral of the story: when an algorithm can be broken down into a sequence of sub-algorithms, the sub-algorithm with largest run time rules the runtime of the combined algorithm.

It should be intuitive that,

$$a_d x^d + a_{d-1} x^{d-1} + \ldots + a_0 = O(x^d)$$

for $a_d > 0$. It is actually a bit of a mess to prove, due to the fact that for $i < d$ some or all of the a_i might be negative.

Theorem 15 *If $f' = O(f)$ and $g' = O(g)$ then $f'g' = O(fg)$.*

Given this theorem, a fairly neat way about a proof is the factorization:

$$f(x) = a_d x^d \left(1 + \sum_{i=0}^{d-1} (a_i/a_d) x^{i-d} \right)$$

followed by showing $1 + \sum (a_i/a_d) x^{i-d} = O(1)$. A little more work gives the matching Ω bound.

Corollary 16 *Let $f(x) = \sum_{i=0}^{d} a_i x^i$ with $a_d > 0$. Then $f = \Theta(x^d)$.*

Example 4 *A quick summary of our results so far:*

$$0 < 1 < \log x < x < x \log x < x^{1+\epsilon} < x^2 < x^3 < \ldots$$

An algorithm running with any of these run times is consider *feasible*. It runs in polynomial time, that is, $O(x^k)$ for some k. Algorithms taking strictly more time than polynomial are consider *infeasible*.

Theorem 17 *$x^k = o(a^x)$ for any k and any $a > 1$.*

Example 5 *Our hierarchy continues:*

$$x^2 < x^3 < \ldots < 2^x < e^x < \ldots < x! < x^x = 2^{(\log x)x} < 2^{x^{1+\epsilon}} < 2^{x^2} < \ldots$$

An algorithm taking more than polynomial time, such as an exponential time algorithm running in $O(2^x)$ is considered infeasible. The reason for this is that small changes in input size increase computation time greatly. In practice, any useful algorithm will be applied to increasingly complex problems. Due to the growth rate, computation power will not keep up with the demands of problem size. For all practical purposes, then, the problem in not solvable by this algorithm.

Exponential time algorithms sometimes invoke the following very simple strategy: "try all possible combinations of outputs and pick the one that's correct." Here is an exponential time algorithm for sorting a pack of cards. There are 52 cards. They can be rearranged into 52! orders. Try all orderings one by one, until the correct one appears. To give the reader an idea of the inefficiency of this method, it wouldn't be much less efficient to sort cards by repeatedly shuffling them, each time looking to see if the cards just happened to fall in order.

This should give an intuition to the practical infeasibility of exponential time algorithms.

Exercises

1. Give proofs for all theorems and corollaries given in these notes.
2. Give proofs for the examples given in these notes.
3. Prove or disprove: if $g_1, g_2 = \Omega(f)$ and a_1, a_2 are positive constants, then $a_1 g_1 + a_2 g_2 = \Omega(f)$.
4. Prove or disprove: if $f' = \Omega(f)$ and $g' = \Omega(g)$ then $f'g' = \Omega(fg)$.
5. Prove or disprove: if $f' = O(f)$ and $g' = O(g)$ then f' and g' are both $O(f + g)$.
6. Give the analogous analytic definition for Ω which agrees with the transpose symmetry of o and Ω.. Warning: the limit requires that the denominator be eventually positive.