A "Gentler" Introduction to Order Notation and Asymptotic Analysis

Daniel J. Kaiser
Computer Science Department
Southwest State University
dkaiser@southwest.msus.edu

Abstract

Students of computer science commonly find their introduction to *big oh* notation to be a stumbling block. Most textbooks on data structures and algorithms cover asymptotic notation in varying degrees of mathematical rigor starting with the definition of *big oh*. However, more and more computer science programs are requiring less and less mathematics of their majors. As a result, when students first encounter the definition of *big oh*, they frequently lack the mathematical sophistication necessary to fully appreciate the treatment it is given in most texts. Furthermore, they are unable to reconcile the definition with any intuitive understanding of complexity they have developed from their programming experience.

The first part of this paper describes a more concrete conceptual framework for motivating the definition of *big oh*. This motivation uses concepts familiar to a student who has taken or is taking the normal CS 1 course. The second part introduces the other asymptotic forms using only concepts normally covered in Discrete Mathematics.

Introduction

The treatment of asymptotic notation in many, if not most, textbooks in Data Structures is to begin with a mathematical definition of *big oh* similar to the following, found in [1].

Definition (Big Oh)

> Consider functions f and g, which are non-negative for all integers $n \geq 0$. We say that "$f(n)$ is *big oh* of $g(n)$," for which we write $f(n) = O(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.

The typical approach is to then use this definition to prove the *big oh* relationship for selected functions. Eventually, theorems concerning the various properties of the *big oh* relation are stated and proven. The texts normally concentrate on how to "use" the definition and seldom give much discussion of its motivation.

The typical student majoring in computer science or information systems will not have developed the mathematical maturity necessary to fully appreciate the association between the abstract definition of the *big oh* relation and the relative complexities of programs they have written. Without mathematical sophistication, the proofs of the *big oh* relation's properties become something to memorize for the quiz or exam and then forget.

This paper presents an approach that allows students to "discover" the *big oh* relation in terms that are familiar to their programming experience. Given this more concrete foundation for the formal definition of the *big oh* relation, the proofs of its properties, while not necessarily easier, are less intimidating.

Background

Students in computer science at Southwest State University start with a one-year sequence in programming using C++ and then have a two-semester sequence in Data Structures and Algorithms. The first semester of the programming sequence concentrates on the basic programming constructs such as primitive data types, functions, selection, iteration, recursion, arrays and the techniques of structured design. The second semester concentrates on the object-oriented paradigm and introduces the elementary data structures of lists, stacks and queues.

Within the first-year sequence, students are introduced to the ideas of time complexity by examining the number of basic operations performed by various algorithms. For example, students consider the difference between the number of comparisons performed in a sequential search and the number performed in a binary search. Complexities are also considered when studying the various ways to implement the elementary data structures. Students thus develop some intuition regarding run time complexity. The *big oh* notation my even be used without formally defining it.

Asymptotic notation is introduced in the first Data Structures and Algorithms course. Students generally take Discrete Mathematics during one of their first two semesters. The Discrete Mathematics course introduces the students to prepositional and predicate logic, elementary set theory, functions, relations, graphs, trees and elementary combinatorics. Thus, it is assumed that the students are reasonably proficient programmers and have studied some mathematical logic before being confronted with the *big oh* relation.

"Discovering" the Definition

The goal of asymptotic analysis is to provide a means of comparing algorithms regardless of their implementations. That is, we what to answer the question, "Can we define what it means for one algorithm to be *better* than another?" To begin their investigations students are presented with the following.

Hypothetical Situation

> Consider two algorithms, A_1 and A_2, for solving the same problem. Suppose we have determined that the number of basic operations performed by A_1 on an instance of size n is n^2 while A_2 performs $4n^2 + 1000$ of its basic operations. It would appear that A_1 is a "faster" solution of the problem than A_2. However, suppose we can implement A_1 on a machine M_1, that can perform 100 of its basic operations per second while A_2 can be implemented on a machine M_2, that can perform 500 of its basic operations per second. Is A_1 really *better* than A_2? Will the faster implementation of the "slower" algorithm eventually catch up with the slower implementation of the "faster" algorithm?

To study the situation for varying instance sizes, we will use a spreadsheet. Table 1 shows the number of operations performed by A_1 and A_2 and the execution times of their respective implementations for various instance sizes. As the table indicates, for some input size between 30 and 35, the faster implementation of the A_2 begins to outperform the slower implementation of A_1.

Table 1: n^2 vs. $4n^2 + 1000$

	Number of Operations		Running Time (seconds)		
Input Size	A_1	A_2	A_1 on M_1		A_2 on M_2
5	25	1100	0.25	<	2.20
10	100	1400	1.00	<	2.80
15	225	1900	2.25	<	3.80
20	400	2600	4.00	<	5.20
25	625	3500	6.25	<	7.00

30	900	4600	9.00	<	9.20
35	1225	5900	12.25	>	11.80
40	1600	7400	16.00	>	14.80
45	2025	9100	20.25	>	18.20
50	2500	11000	25.00	>	22.00
	$T_1(n) = n^2$		M_1 @	100	ops/sec
	$T_2(n) = 4n^2 + 1000$		M_2 @	500	ops/sec

The students can use the spreadsheet to investigate the behavior when various changes are made to the situation. For example, what if M_2 were able to execute only 300 of A_2's basic operations per second instead of 500. Table 2 shows that in this case, as the input size increases, algorithm A_1 running on M_1 increasingly out performs A_2 running on M_2.

The spreadsheet can be set up so that the student can vary the ratio of execution speeds and find the input size at which the faster implementation of the "slower" algorithm does indeed outperform the slower implementation of the "faster" algorithm. The student can also investigate the behavior of other function pairs using the same spreadsheet. If set up properly, the student would only have to change the value in the first row of the *Input Size* column and the remaining rows would change accordingly. Similarly, if the student changed the value of the cell containing the number of operations per second for either M_1 or M_2 the running times would recalculate. To investigate other pairs of functions, the formula for the new functions would be entered into the first cells of columns A_1 and A_2. These formulae would then have to be copied to the remainder of the columns.

Table 2: Slower M_2

	Number of Operations		Running Time		(seconds)
Input Size	A_1	A_2	A_1 on M_1		A_2 on M_2
5	25	1100	0.25	<	3.67
10	100	1400	1.00	<	4.67
15	225	1900	2.25	<	6.33
20	400	2600	4.00	<	8.67
25	625	3500	6.25	<	11.67
30	900	4600	9.00	<	15.33
35	1225	5900	12.25	<	19.67
40	1600	7400	16.00	<	24.67
45	2025	9100	20.25	<	30.33
50	2500	11000	25.00	<	36.67
	$T_1(n) = n^2$		M_1 @	100	ops/sec
	$T_2(n) = 4n^2 + 1000$		M_2 @	300	ops/sec

After investigating various pairs of second degree polynomial functions, the students usually conclude that if one runs the "slower" algorithm on a "fast enough" machine, it will "eventually" out perform the "faster" algorithm running in a slower implementation. Students are then asked to do the same with n^2 and n^3 (table 3).

Table 3: n^2 vs. n^3

Number of Operations	Running Time	(seconds)

Input Size	A_1	A_2	A_1 on M_1		A_2 on M_2
5	25	125	1.25	>	0.25
10	100	1000	5.00	>	2.00
15	225	3375	11.25	>	6.75
20	400	8000	20.00	>	16.00
25	625	15625	31.25	=	31.25
30	900	27000	45.00	<	54.00
35	1225	42875	61.25	<	85.75
40	1600	64000	80.00	<	128.00
45	2025	91125	101.25	<	182.25
50	2500	125000	125.00	<	250.00
	$T_1(n) = n^2$		M_1 @	20	ops/sec
	$T_2(n) = n^3$		M_2 @	500	ops/sec

After trying larger and larger machine speed ratios, they are forced to conclude that in this case, no matter how much faster the implementation of the n^3 algorithm is than the implementation of the n^2 algorithm, the n^2 algorithm "eventually" outperforms the n^3 algorithm.

After performing the above investigations, the students are ready to attempt a definition of what it means for one algorithm to be *no-slower-than* another. The usual consensus definition is some variation on the following.

Definition (No-slower-than)

> Given algorithms A_1 and A_2 that solve the same problem, we will say that A_2 is *no-slower-than* A_1 if we can run A_2 on a machine that is "enough faster than" the machine on which we run A_1, then the faster implementation of A_2 will "eventually" out perform the slower implementation of A_1.

After discussing the need to make our definition more precise, we decide to make a second attempt in which we try to quantify the phrases "enough faster than" and "eventually". This produces the following refinement.

Definition (No-slower-than — Refinement)

> Given algorithms A_1 and A_2 that solve the same problem, we will say that A_2 is *no-slower-than* A_1 if there is some number R such that if we run A_2 on a machine that is R times faster than the machine on which we run A_1, there will exist some instance size N, such that the faster implementation of A_2 will out perform the slower implementation of A_1 on every instance of size greater than N.

Other Relations

Reflecting on our definition of *no-slower-than* we see that we can use it to define what it means for algorithms to be *about-the-same-speed*. Certainly, if A_1 is *no-slower-than* A_2 and A_2 is *no-slower-than* A_1 then it must be the case that A_1 and A_2 are *about-the-same-speed*.

We may further notice that *about-the-same-speed-as* is an equivalence relation on the set of all algorithms that solve the same problem. It is trivial to show that *about-the-same-speed-as* is reflexive (an algorithm is *about-the-same-speed-as* itself), symmetric (if A_1 is *about-the-same-speed-as* A_2 then A_2 is certainly *about-the-same-speed-as* A_1) and transitive (if A_1 is *about-the-same-speed-as* A_2 and A_2 is *about-the-same-speed-as* A_3 then A_1 is clearly *about-the-same-speed-as* A_3).

Looking further we also see that *no-slower-than* is almost a partial order relation on the set of algorithms that solve the same problem. It is very easy to argue that it has the reflexive property since an algorithm is *no-slower-than* itself. The transitive property follows just as easily since if A_1 is *no-slower-than* A_2 and A_2 is *no-slower-than* A_3 then A_1 is surely *no-slower-than* A_3. We can't quite get the anti-symmetric property. The best we can do is argue that if A_1 is *no-slower-than* A_2 and A_2 is *no-slower-than* A_1 then A_1 and A_2 are *about-the-same-speed*. Thus *no-slower-than* is a partial order on the set of equivalence classes under the relation *about-the-same-speed-as*. So, *no-slower-than* behaves on the equivalence classes of *about-the-same-speed-as* just as "$\leq$" behaves on the integers.

If A_1 is *no-slower-than* A_2 then in certainly follows that A_2 is *no-faster-than* A_1. Hence, *no-faster-than* is a relation that behaves on the equivalence classes of *about-the-same-speed-as* just as "$\geq$" behaves on the integers.

Now for an algorithm A_1, to be *inherently-faster-than* another algorithm A_2 it must be the case that A_1 is *no-slower-than* A_2 and that A_1 is not *about-the-same-speed-as* A_2. It is easy to verify that *is inherently-faster-than* is irreflexive and transitive and is therefore a quasi-order [2] on the set of all algorithms that solve the same problem. Thus, *inherently-faster-than* behaves on the equivalence classes of *about-the-same-speed-as* in the same way that "<" behaves on the set of integers.

Summary

Given algorithms A_1 and A_2 that solve the same problem, we have developed the following definitions.

Definition (No-slower-than)

> A_2 is *no-slower-than* A_1 if there is some number R, such that if we run A_2 on a machine that is R times faster than the machine on which we run A_1, there will exist some instance size N, such that the faster implementation of A_2 will out perform the slower implementation of A_1 on every instance of size greater than N.

Definition (No-faster-than)

> A_1 is *no-faster-than* A_2 provided A_2 is *no-slower-than* A_1.

Definition (About-the-same-speed-as)

> A_1 is *about-the-same-speed-as* A_2 provided A_1 is *no-slower-than* A_2 and A_2 is *no-slower-than* A_1.

Definition (Inherently-faster-than)

> A_1 is *inherently-faster-than* A_2 provided A_1 is *no-slower-than* A_2 and A_1 is not *about-the-same-speed-as* A_2.

If, in each of the above definitions, we replace algorithms A_1 and A_2 with their complexity functions, that is, "functions f and g, which are non-negative for all integers $n \geq 0$" we have the familiar definitions of *big oh*, *big omega*, *big theta* and *little oh.*

Definition (Big oh)

> We say that "$f(n)$ is *big oh* of $g(n)$," which we write as $f(n) = O(g(n))$, if there exists an integer n_0 and a constant $c > 0$ such that for all integers $n \geq n_0$, $f(n) \leq cg(n)$.

Definition (Big omega)

We say that "$f(n)$ is *big omega* of $g(n)$," which we write as $f(n) = \Omega(g(n))$, if $g(n) = O(f(n))$

Definition (Big theta)

We say that "$f(n)$ is *big theta* of $g(n)$," which we write as $f(n) = \Theta(g(n))$, if $f(n) = O(g(n))$ and $g(n) = O(f(n))$.

Definition (Little oh)

We say that "$f(n)$ is *little oh* of $g(n)$," which we write as $f(n) = o(g(n))$, if $f(n) = O(g(n))$ and $f(n) \neq \Theta(g(n))$.

Note that the constants "c" and "n_0" from the definition of *big oh* are simply the ratio of machine speeds R, and the instance size N, from the definition of *no-slower-than*.

Concluding Remarks

I submit that the conceptual framework presented has merit for the following reasons.

- The student is allowed to "discover" the definition and work through its refinement instead of being presented with a polished version.
- The approach draws off of the students' programming experience.
- The only mathematical background required is a typical course in discrete mathematics.
- The approach reinforces the ideas of equivalence and partial order relations introduced in discrete mathematics.

References

1. Preiss, B. (1999). *Data Structures and Algorithms*. Wiley.

2. Ross, K. and Wright, C. (1999). *Discrete Mathematics*. Prentice Hall.