Asymptotic Analysis

Thomas A. Anastasio

August 26, 2003

1 Introduction

As a programmer, you often have a choice of data structures and algorithms. Choosing the best one for a particular job involves, among other factors, two important measures:

- *Time Complexity*: how much time will the program take?
- *Space Complexity*: how much storage will the program need?

You will sometimes seek a tradeoff between space and time complexity. For example, you might choose a data structure that requires a lot of storage in order to reduce the computation time. There is an element of art in making such tradeoffs, but you must make the choice from an informed point of view. You must have some verifiable basis on which to make the selection of a data structure or algorithm. Complexity analysis provides one such basis.

2 Complexity

Complexity refers to the rate at which storage or time grows as a function of the problem size (for example, the size of the data on which an algorithm operates). The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. You would expect most programs to run much faster on a supercomputer than on an old desktop PC (and that often turns out to be the case).

We would like to have a way of describing the inherent complexity of a program (or piece of a program), independent of machine/compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a "proportionality" approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as *asymptotic analysis*.

3 Asymptotic Analysis

Asymptotic analysis is based on the idea that as the problem size grows, the complexity will eventually settle down to a simple proportionality to some known

function. This idea is incorporated in the "Big Oh," "Big Omega," and "Big Theta" notation for asymptotic performance. These notations are useful for expressing the complexity of an algorithm without getting lost in unnecessary detail.

Speaking very roughly, "Big Oh" relates to an upper bound on complexity, "Big Omega" relates to a lower bound, and "Big Theta" relates to a tight bound. While the "bound" idea may help us develop an intuitive understanding of complexity, we really need formal and mathematically rigorous definitions. You will see that having such definitions will make working with complexity much easier. Besides, the "bounds" terminology is really not quite correct; it's just a handy mnemonic device for those who know what asymptotic analysis is really about.

3.1 Definitions of Big-Oh, Big-Omega, and Big-Theta

$\diamond$**Definition**: $T(n) = O(f(n))$ if and only if there are constants c_0 and n_0 such that $T(n) \leq c_0 f(n)$ for all $n \geq n_0$.

The expression $T(n) = O(f(n))$ is read as "T of n is in BigOh of f of n," or "T of n is in the set BigOh of f of n," or "T of n is BigOh of f of n."

The definition may seem a bit daunting at first, but it's really just a mathematical statement of the idea that eventually $T(n)$ becomes proportional to $f(n)$. The "eventually" part is captured by "$n \geq n_0$;" n must be "big enough" for $T(n)$ to have settled into its asymptotic growth rate. The "proportionality" part is captured by c_0, the constant of proportionality.[1]

The definition is "if and only if." If you can find the constants c_0 and n_0 such that $T(n) \leq c_0 f(n)$, then $T(n) = O(f(n))$. Also, if $T(n) = O(f(n))$, then there are such constants. It works both ways. It may be helpful to recognize that $O(f(n))$ is a set. It's the set of all functions for which the constants exist and the inequality holds.

If a function $T(n) = O(f(n))$, then eventually the value $cf(n)$ will exceed the value of $T(n)$. "Eventually" means "after n exceeds some value." Does this really mean anything useful? We might say (correctly) that $x^2 + 2x = O(x^{25})$, but we don't get a lot of information from that; x^{25} is simply too big. When we use BigOh analysis, we implicitly agree that the function f we choose is the smallest one which still satisfies the definition. It is correct *and* meaningful to say $x^2 + 2x = O(x^2)$; this tells us something about the growth pattern of the function $x^2 + 2x$, namely that the x^2 term will dominate the growth as x increases.

While BigOh relates to upper bounds on growth, BigOmega relates to lower bounds. The definitions are similar, differing only in the direction of the inequality.

[1] To be mathematically precise about the notation, we should point out that it's the absolute value of the functions that is being compared. Since the time or storage of computer science problems is always positive, we can neglect this nicety.

$\diamond$**Definition**: $T(n) = \Omega(f(n))$ if and only if there are constants c_0 and n_0 such that $T(n) \geq c_0 f(n)$ for all $n \geq n_0$.

In this case, the function $T(n)$ is proportional to $f(n)$ when n is big enough, but the bound is from below, not above. As with BigOh, the expression $T(n) = \Omega(f(n))$ is read as "T of n is BigOmega of f of n," or "T of n is in BigOmega of f of n," or "T of n is in the set BigOmega of f of n." $\Omega(f(n))$, like $O(f(n))$, is the set of functions for which the constants exist and the inequality holds.

As an example, it can be proven that any sorting algorithm that works by comparison of elements requires at least $\Omega(n \log n)$ time.

Finally, we define BigTheta. It is defined in terms of BigOh and BigOmega. A function is in BigTheta if it's both in BigOh and in BigOmega.

$\diamond$**Definition**: $T(n) = \Theta(f(n))$ if and only if $T(n) = O(f(n))$ and also $T(n) = \Omega(f(n))$.

Intuitively this means that $f(n)$ forms *both* an upper- and lower-bound on the function $T(n)$; it is a "tight" bound.

For example, the heapsort algorithm is $O(n \lg n)$, and (as with all sorting algorithms) is $\Omega(n \lg n)$. Therefore, heapsort is $\Theta(n \lg n)$.

4 Some Commonly Encountered Functions

The following functions are often encountered in computer science complexity analysis. We'll express them in terms of BigOh just to be specific. They apply equally to BigOmega and BigTheta.

- $T(n) = O(1)$. This is called constant growth. $T(n)$ does not grow at all as a function of n, it is a constant. It is pronounced "BigOh of one." For example, array access has this characteristic. The operation `A[i]` takes the same number of steps no matter how big `A` is.
- $T(n) = O(\lg(n))$. This is called logarithmic growth. $T(n)$ grows as the base 2 logarithm of n (actually, the base doesn't matter, it's just traditional to use base-2 in computer science). It is pronounced "BigOh of log n." For example, binary search has this characteristic.
- $T(n) = O(n)$. This is called linear growth. $T(n)$ grows linearly with n. It is pronounced "BigOh of n." For example, looping over all the elements in a one-dimensional array would be an $O(n)$ operation.
- $T(n) = O(n \lg(n))$. This is called "n log n" growth. $T(n)$ grows as n times the base 2 logarithm of n. It is pronounced "BigOh of n log n." For example, heapsort and mergesort have this characteristic.
- $T(n) = O(n^k)$. This is called polynomial growth. $T(n)$ grows as the k-th power of n. Computer applications with k greater than about 3

are often impractical. They just take too long to run for any reasonably large problem. Selection sort is an example of a polynomial growth rate algorithm. It is in $O(n^2)$, pronounced "BigOh of n squared."

- $T(n) = O(2^n)$. This is called exponential growth. $T(n)$ grows exponentially. It is pronounced "BigOh of 2 to the n." Exponential growth is the most-feared growth pattern in computer science; algorithms that grow this way are basically useless for anything but very small problems. In computer science, we traditionally use the constant 2, but the any other positive constant could be used to express the same idea.

The growth patterns above have been listed in order of increasing "size." That is,

$$O(1) = O(\lg(n)) = O(n) = O(n \lg(n)) = O(n^2) = O(n^3) = O(2^n)$$

It may appear strange to use the equals sign between two BigOh expressions. Since BigOh defines a *set* of functions, the notation $O(f) = O(g)$ means that the set of functions $O(f)$ is *contained* in the the set of functions $O(g)$. Note that it is not true that if $g(n) = O(f(n))$ then $f(n) = O(g(n))$. The "=" sign does not mean equality in the usual algebraic sense.

5 Best, Worst, and Average Cases

It's usually not enough to describe an algorithm's complexity by simply giving an expression for O, Ω, or Θ. Many computer science problems have different complexities depending on the data on which they work. Best, worst, and average cases are statements about the *data*, not about the algorithm.

The *best* case for an algorithm is that property of the data that results in the algorithm performing as well as it can. The *worst* case is that property of the data that results in the algorithm performing as poorly as possible. The *average* case is determined by averaging algorithm performance over all possible data sets. It is often very difficult to define the average data set.

Note that the worst case and best case do not correspond to upper bound and lower bound. A complete description of the complexity of an algorithm might be "the time to execute this algorithm is in $O(n)$ in the average case." Both the complexity and the property of the data must be stated. A famous example of this is given by the quicksort algorithm for sorting an array of data. Quicksort is in $O(n^2)$ worst case, but is in $O(n \lg n)$ best and average cases. The worst case for quicksort occurs when the array is already sorted (*i.e.*, the data have the property of being sorted).

One of the all-time great theoretical results of computer science is that any sorting algorithm that uses comparisons must take at least $n \lg n$ steps. That means that any such sorting algorithm, including quicksort, must be in $\Omega(n \lg n)$. Since quicksort is in $O(n \lg n)$ in best and average cases, and also in $\Omega(n \lg n)$, it must be in $\Theta(n \lg n)$ in best and average cases.

6 Example: `List` Implementation of `Queue`

Consider a simple list implementation of queue. In this implementation elements are put on the queue at the end of the list and taken off the queue at the head. Since a list requires sequential access to its elements, each `enqueue` operation requires traversing the entire list to find the end, then "attaching" the new item at the end. What is the asymptotic time complexity of this operation as a function of n, the number of items in the queue? First of all, to find the end of the list, we must traverse the entire list. This takes n operations (one "next" operation for each item on the list). Therefore, a single `enqueue` operation has asymptotic complexity $T(n) = O(n)$. But, suppose we `enqueue` n items, one after another? The asymptotic complexity will be $T(n) = O(n^2)$.

You may object that the list is not n long until all n items have been enqueued. Look at it this way; the first item takes one operation, the second takes two operations, the third takes three operations, etc. Therefore, the total number of operations to `enqueue` n items is

$$1 + 2 + 3 + \ldots + n$$

We can express this as

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \frac{n^2 + n}{2} = O(n^2)$$

A better implementation would provide $O(1)$ performance for each `enqueue` operation, thereby allowing n items to be `enqueue`d in $O(n)$ time. This is significantly better than $O(n^2)$.

7 Theorems Involving BigOh

Theorem 1 $O(cf(x)) = O(f(x))$ *(constants don't matter)*

▷ **Proof**: $T(x) = O(cf(x))$ implies that there are constants c_0 and n_0 such that $T(x) \leq c_0(cf(x))$ when $x \geq n_0$
Therefore, $T(x) \leq c_1 f(x)$ when $x \geq n_0$ where $c_1 = c_0 c$
Therefore, $T(x) = O(f(x))$
◁

Theorem 2 Let $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$.
Then, $T_1(n) + T_2(n) = O(\max(f(n), g(n)))$ *(the sum rule)*

▷ **Proof**: From the definition of Big Oh, $T_1(n) \leq c_1 f(n)$ for $n \geq n_1$ and $T_2(n) \leq c_2 g(n)$ for $n \geq n_2$
Let $n_0 = \max(n_1, n_2)$
Then, for $n \geq n_0$, $T_1(n) + T_2(n) \leq c_1 f(n) + c_2 g(n)$

Let $c_3 = \max(c_1, c_2)$
Then

$$\begin{aligned} T_1(n) + T_2(n) &\leq c_3 f(n) + c_3 g(n) \\ &\leq 2c_3 \max(f(n), g(n)) \\ &\leq c \max(f(n), g(n)) \end{aligned}$$

$\lhd$

As an example, suppose your program consists of a sequence of statements, each with its own complexity. What is the complexity of the entire program? Designating the time to complete statement i as T_i, the entire program will run in time $T = \sum_{i=1}^{k} T_i$. From the sum rule, the complexity of the program is given by the maximum complexity of the statements.

Theorem 3 If $T(n)$ is a polynomial of degree x, then $T(n) = O(n^x)$

$\rhd$ **Proof**: $T(n) = n^x + n^{x-1} + \cdots + k$ is a polynomial of degree x
By the sum rule (Theorem 2), the largest term dominates.
Therefore, $T(n) = O(n^x)$
$\lhd$

As an example, it can be shown that in `BubbleSort`, `SelectionSort`, and `InsertionSort`, the worst case number of comparisons is $\frac{n(n-1)}{2}$. What is $O(\frac{n(n-1)}{2})$? Well, $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$ which is a polynomial. Since constants don't matter (Theorem 1), $O(\frac{n^2-n}{2}) = O(n^2 - n)$. By the sum rule (Theorem 2), $O(n^2 - n) = O(n^2)$.

Theorem 4 If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$,
then $T_1(n)T_2(n) = O(f(n)g(n))$ *(the product rule)*

$\rhd$ **Proof**: Since $T_1(n) \leq c1f(n)$ and $T_2(n) \leq c2g(n)$, then $T_1(n)T_2(n) \leq c_1c_2f(n)g(n) \leq cf(n)g(n)$ when $n \geq n_0$
Therefore, $T_1(n)T_2(n) = O(f(n)g(n))$
$\lhd$

Theorem 5 If $T(n) = O(log_B(n))$, then $T(n) = O(\lg(n))$ *(base of logarithm does not matter)*

$\rhd$ **Proof**: $T(n) = O(log_B(n)$ implies that there are constants c_0 and n_0 such that $T(n) <= c_0 log_B(n)$ when $n > n_0$. But, $log_B(n) = \lg(n)/\lg(B)$. Therefore, $T(n) <= \frac{c_0}{\lg B} \lg(n) = c_1 \lg(n)$ when $n > n_0$. Thus, $T(n) = O(\lg(n))$ $\lhd$

The proofs of Theorems 6 and 7, below, use L'Hospital's rule. Recall that this rule pertains to finding the limit of a ratio of two functions as the independent

variable approaches some value. When the limit is infinity, the rule states that

$$\lim_{x\to\infty} \frac{f(x)}{g(x)} = \lim_{x\to\infty} \frac{f'(x)}{g'(x)}$$

where $f'(x)$ and $g'(x)$ denote the first derivatives with respect to x.

We use L'Hospital's rule to determine O or Ω ordering of two functions.

$$f(x) = O(g(x)) \quad \text{if} \quad \lim_{x\to\infty} \frac{f(x)}{g(x)} = 0$$

$$f(x) = \Omega(g(x)) \quad \text{if} \quad \lim_{x\to\infty} \frac{g(x)}{f(x)} = 0$$

Theorem 6 $\lg^k n = O(n)$ for any positive constant k

▷ **Proof**: Note that $\lg^k n$ means $(\lg n)^k$
We must show $\lg^k n \le cn$ for $n \ge n_0$. Equivalently, we can show $\lg n \le cn^{\frac{1}{k}}$. Letting $a = \frac{1}{k}$, we will show that $\lg n = O(n^a)$, for any positive constant a. Use L'Hospital's rule:

$$\lim_{n\to\infty} \frac{\lg n}{cn^a} = \lim_{n\to\infty} \frac{\frac{\lg e}{n}}{acn^{a-1}} = \lim_{n\to\infty} \frac{c_2}{n^a} = 0$$

◁

As a really outrageous (but true) example, $\lg^{1000000}(n) = O(n)$. $\lg^k$ grows *very* slowly!

Theorem 7 $n^k = O(a^n)$ for $a > 1$ *(no polynomial is bigger than an exponential)*

▷ **Proof**: Use L'Hospital's rule

$$\begin{aligned}
\lim_{n\to\infty} \frac{n^k}{a^n} &= \lim_{n\to\infty} \frac{kn^{k-1}}{a^n \ln a} \\
&= \lim_{n\to\infty} \frac{k(k-1)n^{k-2}}{a^n \ln^2 a} \\
&= \cdots \\
&= \cdots \\
&= \lim_{n\to\infty} \frac{k(k-1)\cdots 1}{a^n \ln^k a} \\
&= 0
\end{aligned}$$

◁

As a really outrageous (but true) example, $n^{1000000} = O(1.00000001^n)$. Exponentials grow *very* rapidly!

8 Experimental Measurement

We may wish to check the complexity of a program by measuring the time it takes to complete the program with various data sizes. There are two approaches to doing this, both requiring measurement of the cpu time used by the program.

In the first approach, we measure the time T_1 for a given data size and the time T_2 for twice that data size. If the complexity is $O(f(n))$, we expect $\frac{T_2}{T_1} = \frac{f(2n)}{f(n)}$. For example, for a cubic complexity, we expect $\frac{T_2}{T_1} = \frac{(2n)^3}{n^3} = 8$. Thus, the program should take a factor of 8 longer when we double the data size.

In the second approach, to verify that a program is $O(f(n)$, we compute the ratio $\frac{T(n)}{f(n)}$ for a range of values of n. If $T(n) = O(f(n))$, the ratio should converge to some positive value as n increases. When the ratios converge to zero, we know that $T(n) = O(f(n))$, but that we have chosen a function f that is too large (recall how $x^2 + x = O(x^3)$, but this is not very useful information because x^3 is too large). When the ratios diverge, we know that $T(n) \neq O(f(n))$.