Chapter 3

Asymptotic Analysis

(Revision Date: 6 March 1997)

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.

3.1 Complexity

Complexity refers to the rate at which storage or time grows as a function of the problem size (*i.e.*, 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*.

3.2 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 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.

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

◇ **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

[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.

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 $f(n)$ 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 n 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.

◇ **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 $g(n)$ 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.

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.

◇ **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.

3.3 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.

3.3.1 Graphs of the Commonly Encountered Functions

Here we look at the functions $\lg(n)$, n, $n \lg(n)$, n^2, n^3, and 2^n to get some idea of their relative "size." In the first graph (Figure 3.1), it looks like n^2 and n^3 are larger than 2^n. They're not; the second graph (Figure 3.2) shows the same data on an expanded scale. Clearly $2^n > n^2$ when $n > 4$ and $2^n > n^3$ when $n > 10$.

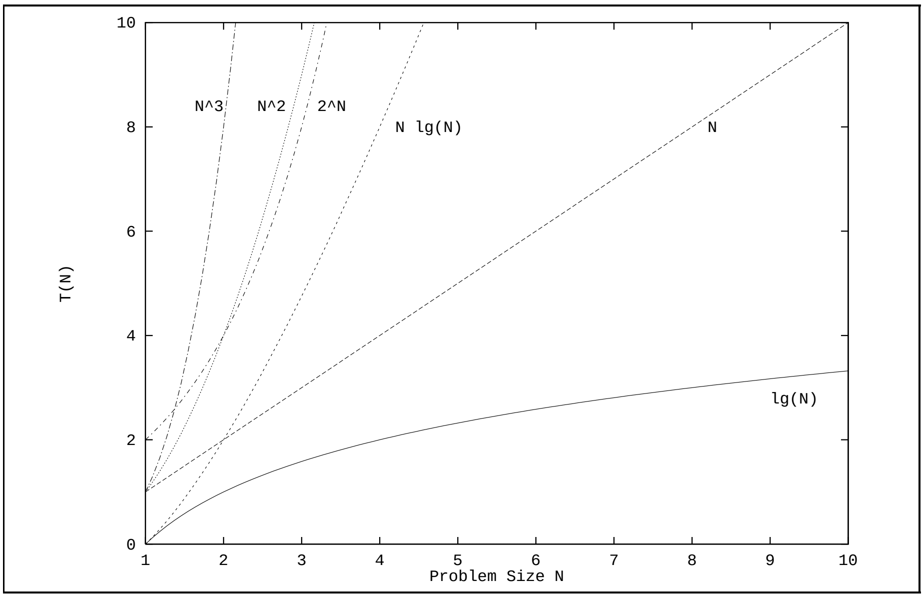

Figure 3.1: Graphs of Common Functions

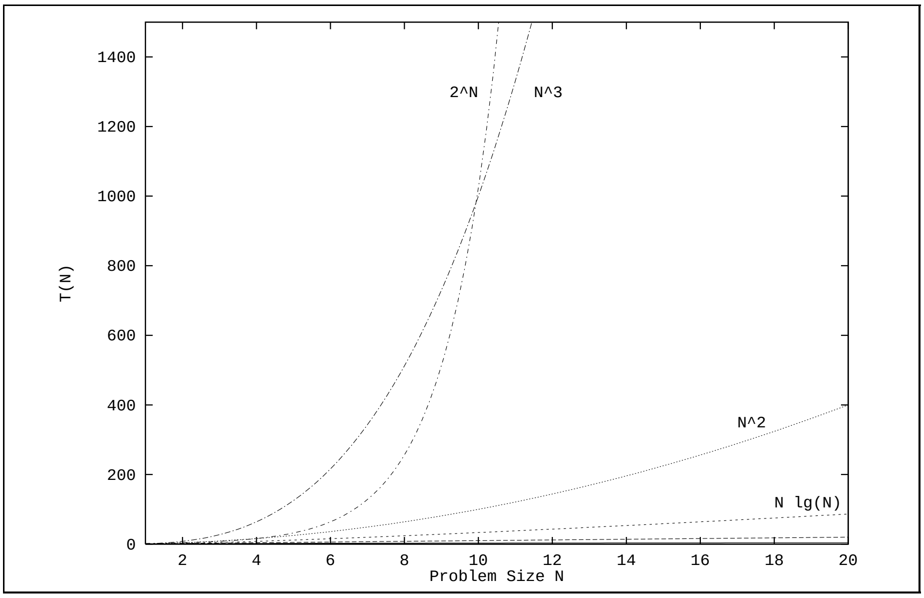

Figure 3.2: Graphs of Common Functions (expanded scale)

3.3.2 Table of Times for the Commonly Encountered Functions

Suppose you have a pretty fast computer capable of performing operations at a rate of 1 billion steps/second. The following table shows $T(n)$, the time to perform n steps on your computer for a variety of $T(n)$ functions.

n	$T(n) = n$	$T(n) = n\lg(n)$	$T(n) = n^2$	$T(n) = n^3$	$T(n) = 2^n$
5	.005 μs	.01 μs	.03 μs	.13 μs	.03 μs
10	.01 μs	.03 μs	.1 μs	1 μs	1 μs
20	.02 μs	.09 μs	.4 μs	8 μs	1 ms
50	.05 μs	.28 μs	2.5 μs	125 μs	13 days
100	.1 μs	.66 μs	10 μs	1 ms	4×10^{13} years

Notice that when $n \geq 50$, the compute time for $T(n) = 2^n$ has started to become too large to be practical. This is most certainly true when $n \geq 100$. You might hope to significantly by getting a much faster machine. However, even if your new machine was a million times faster, 2^n for $n = 100$ would still be 40,000,000 years. You might not want to wait that long for your program to complete.

3.4 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.

3.5 Example: Analysis of a Program

Suppose we have a program which takes some constant amount of time to set up, then grows linearly with the problem size n. The constant time might be used to open a file, for example. The time to open a file is not dependent on the amount of data in the file. After the setup operation, we read the data from the file and do something with it (say print it). The time required to read the file is certainly proportional to the amount of data in the file. There is therefore a lower bound on this program, it is in $\Omega(n)$, where n is the amount of data in the file. Now, our algorithm is in $O(n)$ because it grows linearly with n. Therefore, this program is in $\Theta(n)$. This might be better understood by looking at the graph of Figure 3.3. Let's assume that the setup time is really long, say 500 time units. Let's also assume that the time taken to read the data is $10n$, 10 time units for each data point read. The graph shows the function $500 + 10n$ plotted against n, the problem size. Also shown are the functions n and $20n$.

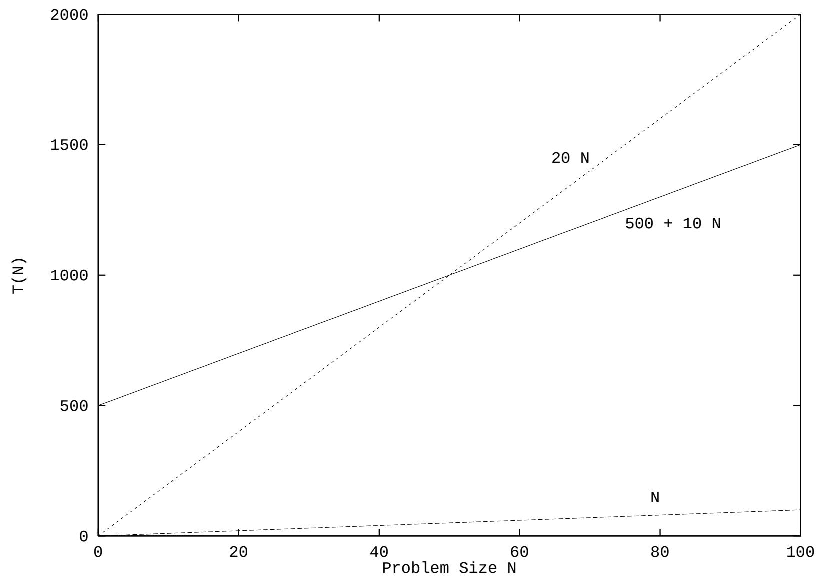

Figure 3.3: Data from "Example: Analysis of a Program"

Note that the function n will *never* be larger than the function $500 + 10n$, no matter how large n gets. However, there are constants c_0 and n_0 such that $500 + 10n \leq c_0 n$ when $n \geq n_0$. One choice for these constants is $c_0 = 20$ and

$n_0 = 50$. Therefore, $500 + 10n = O(n)$. (There are, of course, other choices for c_0; any value of $c_0 > 10$ will do.)

3.6 Example: `List` Implementation of `Queue`

As a final example, we look at 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 $O(n^2)$ (check the chart for $n = 100$, for instance).

3.7 BigOh Does Not Tell the Whole Story

Suppose you have a choice of two approaches to writing a program. Both approaches have the same asymptotic performance (for example, both are in $O(n \lg(n))$). Why select one over the other, they're both the same, right? They may not be the same. There is this not so small matter of the constant of proportionality.

Suppose algorithms **`Alg1`** and **`Alg2`** have the same asymptotic performance, $T_{Alg1}(n) = O(g(n))$ and $T_{Alg2}(n) = O(g(n))$. Now suppose that **`Alg1`** does ten operations for each data item, but algorithm **`Alg2`** only does three. It is reasonable to expect `Alg2` to be faster than `Alg1` even though both have the same asymptotic performance. The reason is that asymptotic analysis ignores constants of proportionality.

As a specific example, let's say that algorithm `Alg1` is

```
{
  set up the algorithm, taking 50 time units;
  read in n elements into array A;  /* 3 units per element */
  for (i = 0; i < n; i++)
    {
      do operation1 on A[i];  /* takes 10 units */
      do operation2 on A[i];  /* takes  5 units */
      do operation3 on A[i];  /* takes 15 units */
    }
}
```

Let's now say that algorithm `Alg2` is

```
{
   set up the algorithm, taking 200 time units;
   read in n elements into array A;  /* 3 units per element */
   for (i = 0; i < n; i++)
     {
       do operation1 on A[i];  /* takes 10 units */
       do operation2 on A[i];  /* takes  5 units */
     }
}
```

Algorithm `Alg1` sets up faster than `Alg2`, but does more operations on the data. The execution time of `Alg1` and `Alg2` will be

$$T_{Alg1}(n) = 50 + 3 * n + (10 + 5 + 15) * n = 50 + 33 * n$$

and

$$T_{Alg2}(n) = 200 + 3 * n + (10 + 5) * n = 200 + 18 * n$$

respectively.

Figure 3.7 graph shows the execution time for the two algorithms as a function of n. Algorithm `Alg1` is the better choice for small values of n. For values of $n > 10$, algorithm `Alg2` is the better choice. Remember that both algorithms have time complexity in $O(n)$.

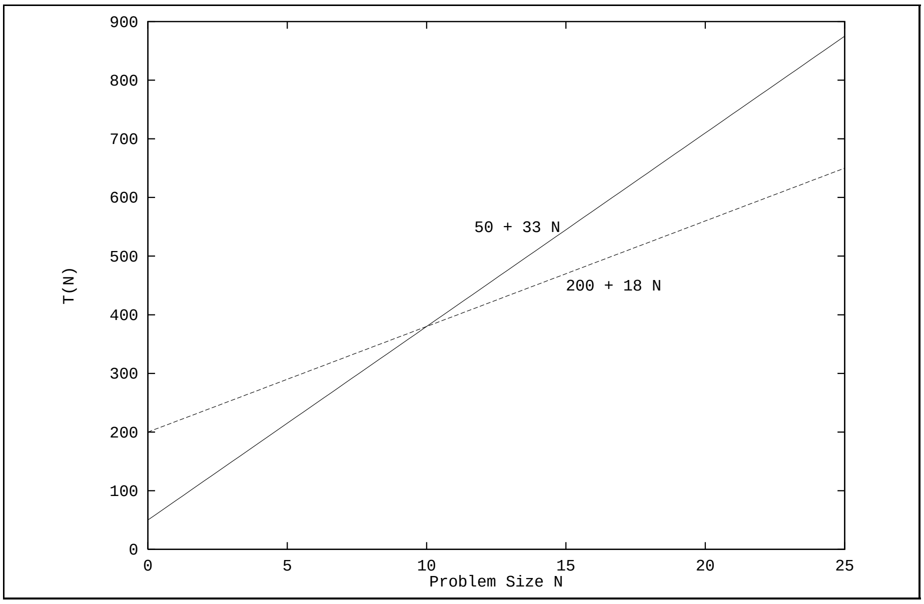

Figure 3.4: BigOh Does Not Tell The Whole Story

3.8 Theorems Involving BigOh

Theorem 3.1 $O(cf(x)) = O(f(x))$ *(constants don't matter)*

$\triangleright$ **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, $T(x) = O(f(x))$
$\triangleleft$

Theorem 3.2 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)*

$\triangleright$ **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

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

$\triangleleft$

Theorem 3.3 If $T(n)$ is a polynomial of degree x, then $T(n) = O(n^x)$

$\triangleright$ **Proof**: $T(n) = n^x + n^{x-1} + \cdots + k$ is a polynomial of degree x
By the sum rule (Theorem 3.2), the largest term dominates.
Therefore, $T(n) = O(n^x)$
$\triangleleft$

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 3.1), $O(\frac{n^2-n}{2}) = O(n^2 - n)$. By the sum rule (Theorem 3.2), $O(n^2 - n) = O(n^2)$.

Theorem 3.4 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)*

$\triangleright$ **Proof**: $T_1(n)T_2(n) \leq c_1c_2f(n)g(n) \leq cf(n)g(n)$ when $n \geq n_0$
Therefore, $T_1(n)T_2(n) = O(f(n)g(n))$
$\triangleleft$

The proofs of Theorems 3.5 and 3.6, 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 3.5 $\lg^k n = O(n)$ for any positive constant k

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

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

$\triangleleft$

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

Theorem 3.6 $n^k = O(a^n)$ for $a > 1$ *(no polynomial is bigger than an exponential)*

$\triangleright$ **Proof**: Use L'Hospital's rule

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

$\triangleleft$

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

We can now re-cap the commonly encountered orders of growth, in increasing order by "Big Oh":

any constant is in $O(1)$

$log^k n$ for $k < 1$

$logn$

$log^k n$ for $k > 1$

n^k for $k < 1$

n *(linear)*

$n \lg n$

n^{1+k} for $k > 0$ *(polynomials)*

2^n *(exponential)*