Text:

1. Growth of Functions

If we have two algorithms which perform the same task on n inputs, and the first has a computing time which is $O(n)$ and the second $O(n^2)$, which is superior? It is easy to see that for sufficiently large values of n, the time for the second algorithm will be larger than the time for the first. For example, if the actual computing times for these algorithms are $2n$ and n^2 respectively, then algorithm one is faster (i.e. has a smaller value) than algorithm two for all $n > 2$. On the other hand, if the actual computing times are $104\ n$ and n^2 then algorithm two is faster for all $n < 104$. For $n > 104$ algorithm one is faster.

So, we cannot decide which of the two algorithms is better unless we know something about the constants associated with the orders of magnitude. If the constants are comparable then the lower order algorithm is better than the higher order algorithm. But this is not the whole story. The point at which one algorithm requires fewer operations than another also depends upon the low order terms. In practice these terms and their coefficients depend on many factors, such as the language and the machine one is using. Alas, it is far more difficult to derive the entire formula for the computing time than the leading term. Thus for a priori analysis, we content ourselves with determining the order of magnitude, and the establishment of its constant will be postponed until after the program has been written and executed. We will not usually derive any terms other than the order of magnitude, unless those terms significantly influence the comparison of two algorithms.

As an example of the usefulness of improving an algorithm by an order of magnitude, suppose we have two algorithms for solving the same task which require n^2 and $n \log n$ operations on n inputs. For $n = 1024$ they require $1{,}048{,}576$ versus $10{,}240$ operations. If it takes one microsecond to perform each operation, then algorithm one requires about 1.05 seconds while algorithm two requires $.01$ seconds on the same input. If we double n to 2048, then the operation counts become $4{,}194{,}304$ versus $22{,}528$ or roughly 4.2 seconds versus $.02$ seconds. When the n is doubled an $O(n^2)$ algorithm takes four times as long to complete while an $O(n \log n)$ algorithm takes only a little more than twice as long to complete. Since an n of several thousand is not especially large, we see how important an order of magnitude improvement such as this can be.

The most common computing times for algorithms we will see here are

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n)$$

$O(1)$ means that the number of executions of basic operations is fixed and hence the total time is bounded by a constant. The first six orders of magnitude have an important property in common, they are bounded by a polynomial. $O(n)$, $O(n^2)$, and $O(n^3)$ are

themselves polynomials referred to by their degrees: linear, quadratic, and cubic. However, there is no integer m such that n^m bounds 2^n, or

$2^n \neq O(n^m)$ for any integer m. The order of this formula is $O(2^n)$

An algorithm whose computing time is bounded below by $\Omega(2^n)$ is said to require exponential time. As *n* gets large, there becomes a tremendous difference between exponential and polynomial time algorithms. If one finds an algorithm which reduces the time to solve a problem from exponential to polynomial, that is a great accomplishment.

Figure 6.1 and Table 6.1 show how the computing times for six of the typical functions grow with a constant equal to one. Notice how the times *O(n)* and *O(n log n)* grow much more slowly than the others. For large data sets, algorithms with a complexity greater than *O(n log n)* are often impractical. An algorithm which is exponential will be practical only for very small values of *n* and even if we decrease the leading constant, say by a factor of 2 or 3, we will not improve the amount of data we can handle by very much.

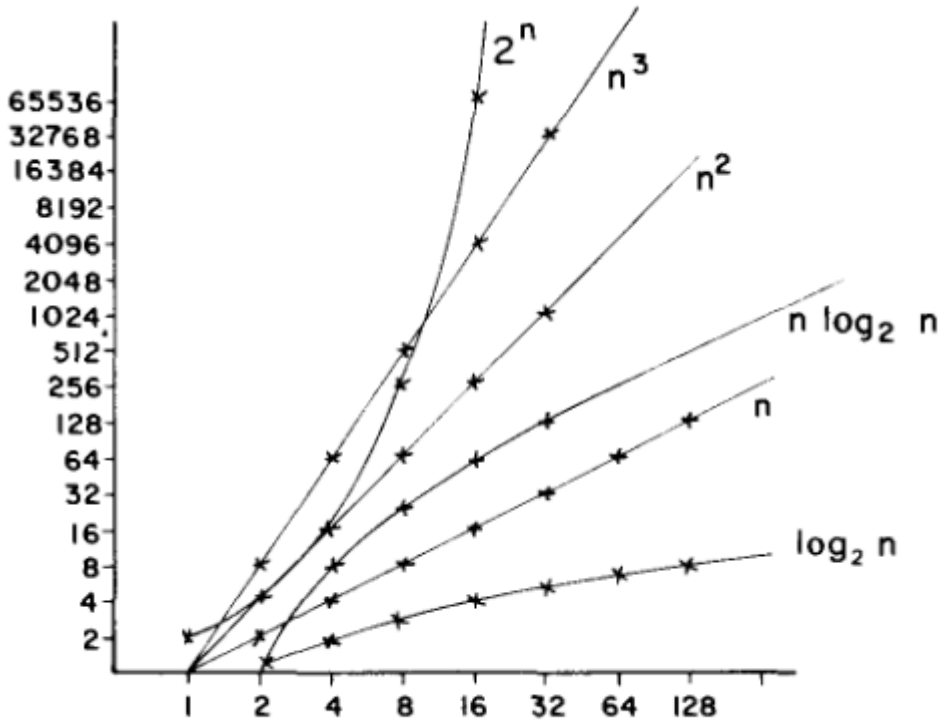

Figure 6.1 Rate of growth of common computing time functions

log n	N	n log n	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65536
5	32	160	1024	32768	4294967296

Table 6.1 Values for computing functions

1.1. Comparison of Functions:

Often times, the running time complexities of different algorithms are presented in the form of functions *T(n)*. In order to know which algorithm is better and to make a quantitative judgement, we have to compare functions (*T(n)*). For simpler functions, we can make the comparisons based on heuristics, however, we must know the technical way to do the same. The following two methods are usually used to compare functions:

i. Substituting "*n*"
ii. Using Logarithm

Let us see some examples to understand the working of these methods

Example 6.1:

f(n) = n^2 & g(n) = n^3

(i) Substituting values of *n*, we get

N	***f(n) = n^2***	***g(n) = n^3***
2	4	8
3	9	27
4	16	64

∴, *f(n) < g(n)*

(ii) Applying log to both functions

f(n) = log n^2 = 2 log n

g(n) = log n^3 = 3 log n

Since, *2log n < 3 log n*

=> *f(n) < g(n)*

Example 6.2:

f(n) = n^2 log n & g(n) = n $(\log n)^{10}$

Applying log to both functions

f(n) = log (n^2 log n) = log n^2 + log log n = 2 log n + log log n

g(n) = log (n $(\log n)^{10}$) = log n + log $(\log n)^{10}$ = log n + 10 log log n

since, *2log n > log n* (log log n is very small term)

=> f(n) > g(n)

Example 6.3:

f(n) = 3 $n^{\sqrt{n}}$ & g(n) = $2^{\sqrt{n} \log n}$

g(n) = $2^{\log n^{\sqrt{n}}}$ = $(n^{\sqrt{n}})^{\log 2}$ = $n^{\sqrt{n}}$ ($a^{\log b} = b^{\log a}$)

Here, *3 $n^{\sqrt{n}}$ > $n^{\sqrt{n}}$* (value wise)

But, *f(n) = g(n)* (asymptotically order is same because we didn't apply log)

Example 6.4:

f(n) = 2^n & g(n) = 2^{2n}

Applying log to both functions

f(n)= log 2^n = n log 2 = n

g(n)= log 2^{2n} = 2n log 2 = 2n

since, *n < 2 n*

f(n) < g(n)

(after taking log, we cannot say that they are asymptotically equal, even if they are of same order)