Asymptotic Analysis

How does the amount of time required by an algorithm grow as the size of the problem grows? This is the *time complexity* of the algorithm. We need to be able to characterize the time complexity of algorithms concisely. To that end, here are a few useful definitions. These definitions all assume that the functions involved are positive for $n > 0$, n_0 is a positive integer, and c is a positive real number.

big-O notation — $f(n) = O\big(g(n)\big)$ means there are constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$. In other words,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c.$$

big-Ω notation — $f(n) = \Omega\big(g(n)\big)$ means there are constants c and n_0 such that $f(n) \geq cg(n)$ for all $n \geq n_0$. In other words,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \geq c,$$

Note that $f(n) = O\big(g(n)\big) \iff g(n) = \Omega\big(f(n)\big)$.

big-Θ notation — If $f(n) = O\big(g(n)\big)$ and $f(n) = \Omega\big(g(n)\big)$, then we write $f(n) = \Theta\big(g(n)\big)$.

little-o notation — $f(n) = o\big(g(n)\big)$ means that for any $c > 0$ there is a constant n_0 such that $f(n) < cg(n)$ for all $n \geq n_0$. In other words,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

little-ω notation — $f(n) = \omega\big(g(n)\big)$ means for any $c > 0$ there is a constant n_0 such that $f(n) > cg(n)$ for all $n \geq n_0$. In other words,

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty.$$

These definitions essentially make it possible to state inequalities on function growth. So, they can be thought of in analogy with the corresponding inequality:

Asymptotic notation	Analogy	Interpretation
$f(n) = O\big(g(n)\big)$	$f \leq g$	"f grows no faster than g"
$f(n) = \Omega\big(g(n)\big)$	$f \geq g$	"f grows no slower than g"
$f(n) = \Theta\big(g(n)\big)$	$f = g$	"f grows at the same rate as g"
$f(n) = o\big(g(n)\big)$	$f < g$	"f grows slower than g"
$f(n) = \omega\big(g(n)\big)$	$f > g$	"f grows faster than g"

With asymptotic notation, we can characterize the complexity of algorithms concisely without regard for lots of small details that become negligible as n gets very large.

For example, both selection sort and insertion sort are $\Theta(n^2)$ algorithms in terms of the number of comparisons required on average. In terms of the number of movements (assignments), selection sort is $\Theta(n)$ whereas insertion sort is $\Theta(n^2)$. Usually the quadratic complexity in terms of comparison dominates, so selection sort and insertion sort are both the same complexity. Practically speaking, this means that, given a list of n items, they will both take "about the same amount of time" to sort it. (We're assuming that run-time is proportional to the number of comparisons.) However, in special circumstances where movements are particularly expensive (e.g., rearranging large records in a file), selection sort may be much faster due to the linear complexity in terms of data movements. Additionally, an algorithm can have different behavior in special cases. The complexity of insertion sort is linear in terms of comparisons when the data is already "almost sorted". In fact, when the data is already completely sorted, insertion sort requires exactly $n - 1$ comparisons and 0 movements whereas selection sort is still $\Theta(n^2)$.

Note that Sedgewick uses the phrase "about $f(n)$" to mean $\Theta(f(n))$ and he couches it in terms of big-O notation as $f(n) + O(g(n))$ for a function g that is asymptotically small compared to f. Using our additional definitions from above, we would denote that f is asymptotically strictly larger than g with little-w notation as $f(n) = \omega\big(g(n)\big)$.

Because of the quadratic complexity of selection sort and insertion sort in general, when the size of the problem doubles, the amount of time required quadruples. This is not desirable and should be avoided if possible. Indeed, there are better sorting algorithms. By "better", of course, we mean having time complexity less than quadratic and, thus, when the problem size doubles the time required increases by less than a factor of 4.

On a side note, there is another quadratic sort called *bubble sort* that is historically popular to show although it is not useful in practice. It works by repeatedly traversing the list and swapping only adjacent elements as needed. This results in excessive comparisons and data movement, though still $\Theta(n^2)$, so asymptotically the same as selection and insertion in general.