Lecture 3

Asymptotic Notation

The result of the analysis of an algorithm is usually a formula giving the amount of time, in terms of seconds, number of memory accesses, number of comparisons or some other metric, that the algorithm takes. This formula often contains unimportant details that don't really tell us anything about the running time. For instance, when we analyzed selection sort, we found that it took $T(n) = n^2 + 3n - 4$ array accesses. For large values of n, the $3n - 4$ part is insignificant compared to the n^2 part.

When comparing the running times of two algorithms, these lower order terms are unimportant when the higher order terms are different. Also unimportant are the constant coefficients of higher order terms; an algorithm that takes a time of $100n^2$ will still be faster than an algorithm that takes n^3 for any value of n larger than 100. Since we're interested in the *asymptotic* behavior of the growth of the function, the constant factor can be ignored.

Upper Bounds: Big-*O*

We need a formal way of expressing these intuitive notions. One popular way is "big-Oh" notation. It tells us that a certain function will never exceed another, simpler function beyond a constant multiple and for large enough values of n. For example, we can simplify $3n^2 + 4n - 10$ to $O(n^2)$. We write "$3n^2 + 4n - 10 = O(n^2)$" and say "three n squared plus four n minus ten is big-Oh of n squared." We might also say "...is *in* big-Oh..," but we don't say "...is *equal* to big-Oh..," the equal sign in this case is more like a set-membership sign.

Let's define big-Oh more formally:

> $O(g(n))$ = { the set of all f such that there exist positive constants c and n_0 satisfying $0 <= f(n) <= cg(n)$ for all $n >= n_0$ }.

This means that, for example, $O(n^2)$ is a set, or *family*, of functions like $n^2 + n$, $4n^2 - n \log n + 12$, $n^2/5 - 100n$, $n \log n$, $50n$, and so forth. Every function $f(n)$ bounded above by some constant multiple $g(n)$ for all values of n greater than a certain value is in $O(g(n))$.

Examples:

- Show $3n^2 + 4n - 2 = O(n^2)$.
  We need to find c and n_0 such that:

  $3n^2 + 4n - 2 <= cn^2$ for all $n >= n_0$.

  Divide both sides by n^2, getting:

  $3 + 4/n - 2/n^2 <= c$ for all $n >= n_0$.

If we choose n_0 equal to 1, then we need a value of c such that:

$3 + 4 - 2 <= c$

We can set c equal to 6. Now we have:

$3n^2 + 4n - 2 <= 6n^2$ for all $n >= 1$.

- Show $n^3 != O(n^2)$. Let's assume to the contrary that

  $n^3 = O(n^2)$

  Then there must exist constants c and n_0 such that

  $n^3 <= cn^2$ for all $n >= n_0$.

  Dividing by n^2, we get:

  $n <= c$ for all $n >= n_0$.

  But this is not possible; we can never choose a constant c large enough that n will never exceed it, since n can grow without bound. Thus, the original assumption, that $n^3 = O(n^2)$, must be wrong so $n^3 != O(n^2)$.

Big-Oh gives us a formal way of expressing *asymptotic upper bounds*, a way of bounding from above the growth of a function. Knowing where a function falls within the big-Oh hierarchy allows us to compare it quickly with other functions and gives us an idea of which algorithm has the best time performance.

Lower Bounds: Omega

Another way of grouping functions, like big-Oh, is to give an *asymptotic lower bound*. Given a complicated function f, we find a simple function g that, within a constant multiple and for large enough n, bounds f from below. Define Ω:

$\Omega(g(n))$ = { the set of all f such that there exist positive constants c and n_0 satisfying $0 <= cg(n) < f(n)$ for all $n >= n_0$ }.

This gives us a somewhat different family of functions; now i any function f that grows strictly faster than g is in $\Omega(g)$. So, for example, $n^3 = \Omega(n^2)$.

Tight Bounds: Theta

Neither big-Oh or Omega are completely satisfying; we would like a tight bound on how quickly our function grows. To say it doesn't grow any faster than something doesn't help us know how slowly it grows, and vice-versa. So we need something to give us a tigher bound; something that bounds a function from both above and below. We can combine big-Oh and Omega to give us a new set of functions, Theta:

$\Theta(g(n))$ = { the set of functions $f(n)$ such that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ } .

It is equivalent to say:

$\Theta(g(n))$ = { the set of all f such that there exist positive constants c_1, c_2 and n_0 satisfying 0 <= $c_1 g(n)$ <= $f(n)$ <= $c_2 g(n)$ for all n >= n_0 }.

Whenever possible, we try to bound the running time of an algorithm from both above and below with Theta notation.

Convenient Theorems

For polynomials, and indeed a large class of functions that are the sums of monotonically increasing terms, we can simplify the process of O etc. notation by noticing these rules. Below, p and q are arbitrary polynomials:

- $c\ p(n) = \Theta(p(n))$.
- $p(n) + q(n) = \Theta(\max(p(n), q(n))$.
- $n^k = O((1 + \varepsilon)^n)$ for any constant k and any positive constant ε (i.e., any polynomial is bounded by any exponential).