Lecture 16: Introduction to Asymptotic Analysis

Algorithmic complexity is a very important topic in computer science. Knowing the complexity of algorithms allows you to answer questions such as

- How long will a program run on an input?
- How much space will it take?
- Is the problem solvable?

These are important bases of comparison between different algorithms. An understanding of algorithmic complexity provides programmers with insight into the efficiency of their code. Complexity is also important to several theoretical areas in computer science, including algorithms, data structures, and complexity theory.

Asymptotic Analysis

When analyzing the running time or space usage of programs, we usually try to estimate the time or space as function of the input size. For example, when analyzing the worst case running time of a function that sorts a list of numbers, we will be concerned with how long it takes as a function of the length of the input list. For example, we say the standard insertion sort takes time $T(n)$ where $T(n)=$ $c*n^2+k$ for some constants *c and k*. In contrast, merge sort takes time $T'(n) = c'*n*log_2(n) + k'$.

The **asymptotic** behavior of a function *f(n)* (such as *f(n)=c*n* or *f(n)=c*n*^{2}, etc.) refers to the growth of *f(n)* as *n* gets large. We typically ignore small values of *n*, since we are usually interested in estimating how slow the program will be on large inputs. A good rule of thumb is: the slower the asymptotic growth rate, the better the algorithm (although this is often not the whole story).

By this measure, a linear algorithm (*i.e., f(n)=d*n+k*) is always asymptotically better than a quadratic one (*e.g., f(n)=c*n*^{2}*+q*). That is because for any given (positive) *c,k,d*, and *q* there is always some *n* at which the magnitude of *c*n*^{2}*+q* overtakes *d*n+k*. For moderate values of *n*, the quadratic algorithm could very well take less time than the linear one, for example if *c* is significantly smaller than *d* and/or *k* is significantly smaller than *q*. However, the linear algorithm will always be better for sufficiently large inputs. Remember to **THINK BIG** when working with asymptotic rates of growth.

Worst-Case and Average-Case Analysis

When we say that an algorithm runs in time *T(n)*, we mean that *T(n)* is an upper bound on the running time that holds for all inputs of size *n*. This is called *worst-case analysis*. The algorithm may very well take less time on some inputs of size *n*, but it doesn't matter. If an algorithm takes *T(n)=c*n*^{2}*+k* steps on only a single input of each size *n* and only *n* steps on the rest, we still say that it is a quadratic algorithm.

A popular alternative to worst-case analysis is *average-case analysis*. Here we do not bound the worst case running time, but try to calculate the expected time spent on a randomly chosen input. This kind of analysis is generally harder, since it involves probabilistic arguments and often requires assumptions about the distribution of inputs that may be difficult to justify. On the other hand, it can be more useful because sometimes the worst-case behavior of an algorithm is misleadingly bad. A good example of this is the popular quicksort algorithm, whose worst-case running time on an input sequence of length n is proportional to n^2 but whose expected running time is proportional to n log n.

Order of Growth and Big-O Notation

In estimating the running time of `insert_sort` (or any other program) we don't know what the constants c or k are. We know that it is a constant of moderate size, but other than that it is not important; we have enough evidence from the asymptotic analysis to know that a `merge_sort` (see below) is faster than the quadratic `insert_sort`, even though the constants may differ somewhat. (This does not always hold; the constants can sometimes make a difference, but in general it is a very good rule of thumb.)

We may not even be able to measure the constant c directly. For example, we may know that a given expression of the language, such as `if`, takes a constant number of machine instructions, but we may not know exactly how many. Moreover, the same sequence of instructions executed on a Pentium IV will take less time than on a Pentium II (although the difference will be roughly a constant factor). So these estimates are usually only accurate up to a constant factor anyway. For these reasons, we usually ignore constant factors in comparing asymptotic running times.

Computer scientists have developed a convenient notation for hiding the constant factor. We write $O(n)$ (read: "order n") instead of "cn for some constant c." Thus an algorithm is said to be $O(n)$ or *linear time* if there is a fixed constant c such that for all sufficiently large n, the algorithm takes time at most cn on inputs of size n. An algorithm is said to be $O(n^2)$ or *quadratic time* if there is a fixed constant c such that for all sufficiently large n, the algorithm takes time at most cn^2 on inputs of size n. $O(1)$ means *constant time*.

Polynomial time means $n^{O(1)}$, or n^c for some constant c. Thus any constant, linear, quadratic, or cubic ($O(n^3)$) time algorithm is a polynomial-time algorithm.

This is called *big-O notation*. It concisely captures the important differences in the asymptotic growth rates of functions.

One important advantage of big-O notation is that it makes algorithms much easier to analyze, since we can conveniently ignore low-order terms. For example, an algorithm that runs in time

$10n^3 + 24n^2 + 3n \log n + 144$

is still a cubic algorithm, since

$10n^3 + 24n^2 + 3n \log n + 144$
$<= 10n^3 + 24n^3 + 3n^3 + 144n^3$
$<= (10 + 24 + 3 + 144)n^3$
$= O(n^3)$.

Of course, since we are ignoring constant factors, any two linear algorithms will be considered equally good by this measure. There may even be some situations in which the constant is so huge in a linear algorithm that even an exponential algorithm with a small constant may be preferable in practice. This is a valid criticism of asymptotic analysis and big-O notation. However, as a rule of thumb it has served us well. Just be aware that it is *only* a rule of thumb--the asymptotically optimal algorithm is not necessarily the best one.

Some common orders of growth seen often in complexity analysis are

$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n \log n)$	"n log n"
$O(n^2)$	quadratic
$O(n^3)$	cubic
$n^{O(1)}$	polynomial
$2^{O(n)}$	exponential

Here *log* means $\log_2$ or the logarithm base 2, although the logarithm base doesn't really matter since logarithms with different bases differ by a constant factor. Note also that $2^{O(n)}$ and $O(2^n)$ are not the same!

Comparing Orders of Growth

O

Let *f* and *g* be functions from positive integers to positive integers. We say *f* is $O(g(n))$ (read: "*f* is order *g*") if *g* is an upper bound on *f*: there exists a fixed constant *c* and a fixed n_0 such that for all $n \geq n_0$,

$f(n) \leq cg(n)$.

Equivalently, *f* is $O(g(n))$ if the function $f(n)/g(n)$ is bounded above by some constant.

o

We say *f* is $o(g(n))$ (read: "*f* is little-o of *g*") if for all arbitrarily small real $c > 0$, for all but perhaps finitely many *n*,

$f(n) \leq cg(n)$.

Equivalently, *f* is $o(g)$ if the function $f(n)/g(n)$ tends to 0 as *n* tends to infinity. That is, f is small compared to g. If *f* is $o(g)$ then *f* is also $o(g)$

Ω

We say that *f* is $\Omega(g(n))$ (read: "f is omega of g") if *g* is a *lower* bound on *f* for large *n.* Formally, f is Ω(g) if there is a fixed constant *c* and a fixed n_0 such that for all $n > n_0$,

$cg(n) \le f(n)$

For example, any polynomial whose highest exponent is n^k is $\Omega(n^k)$. If $f(n)$ is $\Omega(g(n))$ then $g(n)$ is $O(f(n))$. If $f(n)$ is $o(g(n))$ then $f(n)$ is *not* $\Omega(g(n))$.

Θ

We say that f is $\Theta(g(n))$ (read: "f is theta of g") if g is an accurate characterization of f for large n: it can be scaled so it is both an upper and a lower bound of f. That is, f is both $O(g(n))$ and $\Omega(g(n))$. Expanding out the definitions of Ω and O, f is $\Theta(g(n))$ if there are fixed constants c_1 and c_2 and a fixed n_0 such that for all $n>n_0$,

$c_1 g(n) \le f(n) \le c_2\ g(n)$

For example, any polynomial whose highest exponent is n^k is $\Theta(n^k)$. If f is $\Theta(g)$, then it is $O(g)$ but not $o(g)$. Sometimes people use $O(g(n))$ a bit informally to mean the stronger property $\Theta(g(n))$; however, the two are different.

Here are some examples:

- $n + \log n$ is $O(n)$ and $\Theta(n)$, because for all $n > 1$, $n < n + \log n < 2n$.
- n^{1000} is $o(2^n)$, because $n^{1000}/2^n$ tends to 0 as n tends to infinity.
- For any fixed but arbitrarily small real number c, $n \log n$ is $o(n^{1+c})$, since $n \log n / n^{1+c}$ tends to 0. To see this, take the logarithm

  $\log(n \log n / n^{1+c})$
  $= \log(n \log n) - \log(n^{1+c})$
  $= \log n + \log \log n - (1+c)\log n$
  $= \log \log n - c \log n$

  and observe that it tends to negative infinity.

The meaning of an expression like $O(n^2)$ is really a set of functions: all the functions that are $O(n^2)$. When we say that $f(n)$ is $O(n^2)$, we mean that $f(n)$ is a member of this set. It is also common to write this as $f(n) = O(g(n))$ although it is not really an equality.

We now introduce some convenient rules for manipulating expressions involving order notation. These rules, which we state without proof, are useful for working with orders of growth. They are really statements about sets of functions. For example, we can read #2 as saying that the product of any two functions in $O(f(n))$ and $O(g(n))$ is in $O(f(n)g(n))$.

1. $cn^m = O(n^k)$ for any constant c and any $m \le k$.
2. $O(f(n)) + O(g(n)) = O(f(n) + g(n))$.
3. $O(f(n))O(g(n)) = O(f(n)g(n))$.
4. $O(cf(n)) = O(f(n))$ for any constant c.
5. c is $O(1)$ for any constant c.
6. $\log_b n = O(\log n)$ for any base b.

All of these rules (except #1) also hold for Θ as well.

Shortcomings of asymptotic analysis

In practice, other considerations beside asymptotic analysis are important when choosing between algorithms. Sometimes, an algorithm with worse asymptotic behavior is preferable. For the sake of this discussion, let algorithm *A* be asymptotically better than algorithm *B*. Here are some common issues with algorithms that have better asymptotic behavior:

- Implementation complexity
  Algorithms with better complexity are often (much) more complicated. This can increase coding time and the constants.
- Small input sizes
  Asymptotic analysis ignores small input sizes. At small input sizes, constant factors or low order terms could dominate running time, causing *B* to outperform *A*.
- Worst case versus average performance
  If *A* has better worst case performance than *B*, but the average performance of *B* given the expected input is better, then *B* could be a better choice than *A*. Conversely, if the worst case performance of *B* is unacceptable (say for life-threatening or mission-critical reasons), *A* must still be used.

Complexity of algorithms from class

	Average	Worst case
Binary search tree	--	O(max height of tree)
Red-black tree	--	O(log n)
Splay tree	O(log n)	O(n)
DFS, BFS	--	O(m+n)

	Insert	Minimum	Extract Min	Union	Decrease	Delete
Binary heap	O(log(n))	O(1)	O(log(n))	O(n)	O(log(n))	O(log(n))
Binomial heap	O(log(n))	O(log(n))	O(log(n))	O(1)	O(log(n))	O(log(n))