Algorithm Complexity

Warren MacEvoy

September 21, 2018

Understanding asymptotic behavior of a series or function is useful.

introduction

A key classifier in algorithm design is the order of operations required to implement it. For example, the bubble sort's $O(n^2)$ vs merge sort's $O(n \log n)$ asymptotic behavior categorizes merge sort as a fundamentally better algorithm.

Definition 1. Big O. This is a specific meaning of "less than or about the same size as" for functions.

$O(g(n))$ is a set of functions that, for large enough n, eventually are smaller in magnitude than some constant multiple of $|g(n)|$. That is $\tilde{g}(n) \in O(g(n))$ if, and only if, there exists some $c > 0$ and some n_0 so that $|\tilde{g}(n)| \leq c|g(n)|$ for all $n \geq n_0$.

In an expression, writing $O(g(n))$ means, "some specific function in $O(g(n))$. You can also write $f(n) = O(g(n))$ as $f(n) \lessapprox g(n)$.

Definition 2. Little o. This is a specific meaning of "much less than" for functions.

$o(g(n))$ is a set of functions that, for large enough n, eventually are smaller in magnitude than *any* constant multiple of $|g(n)|$. That is $\tilde{g}(n) \in o(g(n))$ if, and only if, for any $c > 0$ there is some n_0 so that $|\tilde{g}(n)| \leq c|g(n)|$ for all $n \geq n_0$.

In an expression, writing $o(g(n))$ means, "plus some specific function in $o(g(n))$. You can also write $f(n) = o(g(n))$ as $f(n) \ll g(n)$.

Definition 3. Big Ω. This is a specific meaning of "more than or about the same size as" for functions.

$\Omega(g(n))$ is a set of functions that, for large enough n, eventually are larger in magnitude than some constant multiple of $|g(n)|$. That is $\tilde{g}(n) \in \Omega(g(n))$ if, and only if, there exists some $c > 0$ and some n_0 so that $|\tilde{g}(n)| \geq c|g(n)|$ for all $n \geq n_0$.

In an expression, writing $\Omega(g(n))$ means, "some specific function in $\Omega(g(n))$. You can also write $f(n) = \Omega(g(n))$ as $f(n) \gtrapprox g(n)$.

Definition 4. Little ω. This is a specific meaning of "much more than" for functions.

$\omega(g(n))$ is a set of functions that, for large enough n, eventually are larger in magnitude than any constant multiple of $|g(n)|$. That is

$\tilde{g}(n) \in \omega(g(n))$ if, and only if, for any $c > 0$ there is some n_0 so that $|\tilde{g}(n)| \geq c|g(n)|$ for all $n \geq n_0$.

In an expression, writing $\omega(g(n))$ means, "some specific function in $\omega(g(n))$. You can also write $f(n) = \omega(g(n))$ as $f(n) \gg g(n)$.

set	relation	constant	for all $n \geq n_0 \ldots$
$O(g(n))$	$\tilde{g}(n) \lessapprox g(n)$	some $c > 0$	$\lvert\tilde{g}(n)\rvert \leq c\lvert g(n)\rvert$
$o(g(n))$	$\tilde{g}(n) \ll g(n)$	any $c > 0$	$\lvert\tilde{g}(n)\rvert \leq c\lvert g(n)\rvert$
$\Omega(g(n))$	$\tilde{g}(n) \gtrapprox g(n)$	some $c > 0$	$\lvert\tilde{g}(n)\rvert \geq c\lvert g(n)\rvert$
$\omega(g(n))$	$\tilde{g}(n) \gg g(n)$	any $c > 0$	$\lvert\tilde{g}(n)\rvert \geq c\lvert g(n)\rvert$

O categories

$O(1)$ means "bounded" - $f(n) \in O(1)$ means $|f(n)| \leq c$ for some c and every $n > n_0$. Any algorithm that takes some maximum number of steps to complete *independent* of the number of elements it is working with would be constant complexity. Indexing into an array or computing a hash are $O(1)$ operations.

$O(\log n)$. Logarithms grow very slowly, and so "almost constant". An algorithm that can work with $n = 2^{128}$ items, more than can conceivably ever be stored on a planet sized computer, would only take at most 128 times longer than working with 2 items. A binary search in a sorted array or balanced tree are $O(\log n)$ operations.

$O(n)$. Linear. Any algorithm that typically goes through some fixed fraction of all the elements it contains to complete would be linear in complexity. Indexing into a linked list or searching for an element in an unsorted array are $O(n)$ operations.

$O(n \log n)$ Log linear. An algorithm that must do an $O(\log n)$ step on each of it's elements would have this kind of complexity. Since inserting an element is $O(\log n)$ for a balanced tree, creating a balanced tree from n items is an $O(n \log n)$ step. Sorts based on comparison and swapping are, at best, $O(n \log n)$.

$O(n^p)$ Polynomial. Nested linear loops where the sub-loops go though a significant fraction of the entire set tend to be polynomial with p the number of nested loops. Bubble sort is $O(n^2)$ while a traditional matrix multiply is $O(n^3)$.

$O(b^n)$ Exponential. Algorithms that have b branches to explore at every level of an n depth tree are b^n complex.

$O(n!)$ Factorial. This is faster than any exponential. Trying every permutation is an example of factorial complexity[1]

[1] Stirling's approximation is $n! \approx n^n e^{-n}\sqrt{2\pi n}$, or the slightly better Gosper's approximation $n! \approx n^n e^{-n}\sqrt{(2n + 1/3)\pi}$.

Note $a^n n^b (\log n)^c \ll A^n n^B (\log n)^C$ whenever a and A are positive and (a, b, c) is lexicographically less than (A, B, C): ($a < A$) or ($a = A$ and $b < B$) or ($a = A$ and $b = B$ and $c <$ C).

O manipulations

Theorem 1. $f(n) \lessapprox g(n) \iff g(n) \gtrapprox f(n)$

Suppose $f \in O(g)$. So there is a $c > 0$ and n_0 so that $|f(n)| \leq c|g(n)|$ when $n \geq n_0$. This can be re-written as $|g(n)| \geq \frac{1}{c}|f(n)|$. Using $\tilde{c} = 1/c$, and the same n_0, this is the required criteria for $g(n) \in \Omega(f(n))$. Going backwards is similar.

Theorem 2. $f(n) \ll g(n) \iff g(n) \gg f(n)$

Suppose $f \in o(g)$. So for any $c > 0$ there is an $n_0(c)$ so that $|f(n)| \leq c|g(n)|$ when $n \geq n_0$. This can be re-written as $|g(n)| \geq \frac{1}{c}|f(n)|$. So for any $\tilde{c}$ in the ω criteria, choosing $c = 1/\tilde{c}$ gives the required $\tilde{n}_0(\tilde{c}) = n_0(1/\tilde{c})$. Going backwards is similar.

Other useful facts:

- Adding a finite number of small terms to a bounded term is bounded: $O(g(n)) + o(g(n)) = O(g(n))$.
- Adding a finite number of small terms to a small term is small: $o(g(n)) + o(g(n)) = o(g(n))$.
- if $f(n) \ll g(n)$, then $O(f(n))$ and $o(f(n))$ are small compared to $g(n)$.
- Adding a finite number of bounded terms to a bounded term is bounded: $O(g(n)) + O(g(n)) = O(g(n))$.
- Nonzero constants don't matter: $O(\alpha g(n)) = O(g(n))$ and $o(\alpha g(n)) = o(g(n))$, so long as $\alpha \neq 0$.

estimating order

Suppose you have $f(n)$ experimentally for some increasing sequence of n, $\{n_k\}_{k=0}^{N-1}$, and would like to determine it's leading order behavior,

$$f(n) \approx cn^p(\log n)^q\,. \tag{1}$$

You can use any base for the log you like, so long as you are consistent in what follows: always use the same base, and when you exponentiate, use the base of the log. The common bases are $B = 2$ (computer science), $B = e$ (math), and $B = 10$ (engineering).

Step 1. Look at the exponential growth estimator

$$r_k = \left| \frac{\log(\log|f(n_k)|) - \log(\log|f(n_{k-1})|)}{n_k - n_{k-1}} \right| \tag{2}$$

If that ratio does not trend toward zero, then $f(n)$'s growth is exponential or worse. If so, replace $f(n)$ by $\tilde{f}(n) = \log f(n)$ in the remaining analysis to estimate

$$\tilde{f}(n) \approx \tilde{c} n^{\tilde{p}} (\log n)^{\tilde{q}} \tag{3}$$

then exponentiate the result (B is the base of the log),

$$f(n) \approx B^{\tilde{f}(n)} = n^{\tilde{q}} B^{\tilde{c} n^{\tilde{p}}}\,. \tag{4}$$

Step 2. Compute the auxiliary values $\{(x_k, y_k)\}_{k=1}^{N-1}$:

$$x_k = \frac{\log(\log n_k) - \log(\log n_{k-1})}{\log n_k - \log n_{k-1}} \tag{5}$$

and

$$y_k = \frac{\log\log f(n_k) - \log\log f(n_{k-1})}{\log n_k - \log n_{k-1}} \tag{6}$$

Step 3: Estimate p, q and c for $k = 2, \ldots, N-1$ by:

$$q_k = \frac{y_k - y_{k-1}}{x_k - x_{k-1}} \tag{7}$$

$$p_k = y_k - q_k x_k \tag{8}$$

$$c_k = \frac{f(n_k)}{n^{p_k} (\log n)^{q_k}} \tag{9}$$

Step 5: You need at least N=3 samples to get any estimate for c, p, and q. I recommend you use enough samples to plot a trend line for these estimates. These steps are implemented as a google spreadsheet in `goo.gl/qx5mNS` , which you may copy and replace the n and $f(n)$ columns with your data and adjust the number of rows after the third row of sample data.

Step 6. If you determined that the growth of your data is exponential in step 1, don't forget to exponentiate your results as described in that step.

Estimating Sums

Summing is a linear operation, so

$$\sum_{k=0}^{n-1} [\alpha f(k) + \beta g(k)] = \alpha \sum_{k=0}^{n-1} f(k) + \beta \sum_{k=0}^{n-1} g(k)\,. \tag{10}$$

Summing powers of k:

$$\sum_{k=0}^{n-1} 1 = n\,. \tag{11}$$

$$\sum_{k=0}^{n-1} k = \frac{1}{2} n(n-1) = \frac{1}{2} n^2 + O(n)\,. \tag{12}$$

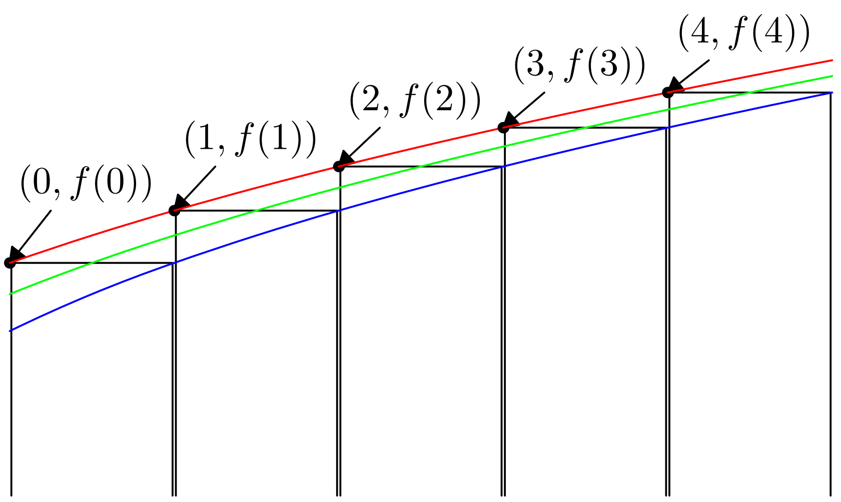

Figure 1: Comparing sum (black rectangles) with integrals of increasing f (red), f shifted -1/2 (green), and f shifted -1 (blue).

$$\sum_{k=0}^{n-1} k^2 = \frac{1}{3}n(n-1/2)(n-1) = \frac{1}{3}n^3 + O(n^2)\,. \tag{13}$$

$$\sum_{k=0}^{n-1} k^3 = \frac{1}{4}n^2(n-1)^2 = \frac{1}{4}n^4 + O(n^3)\,. \tag{14}$$

Generally, for $p > 0$,

$$\sum_{k=0}^{n-1} k^p \approx \frac{1}{p+1}(n-1/2)^{p+1} = \frac{1}{p+1}n^{p+1} + O(n^p)\,. \tag{15}$$

Estimating sums with integrals.
If $f(x)$ is increasing, then

$$\int_{a-1}^{b} f(x)\,dx \le \sum_{k=a}^{b} f(k) \approx \int_{a-1/2}^{b+1/2} f(x)dx \le \int_{a}^{b+1} f(x)\,dx \tag{16}$$