Big $\mathcal{O}$ Notation

Dylan Lane McDonald

CNM STEMulus Center
Web Development with PHP

November 11, 2015

The STEMulus Center
powered by CNM Ingenuity

Outline

1. Mathematics
   - Limits
2. Algorithmic Complexity
   - Graphical
   - Algebra
   - Calculus
   - Impact

Limits

Intuitively, the **limit** of a function $f(x)$ as x approaches a value c is where the function will arrive at c. Consider the function:

$$f(x) = \frac{x-1}{x-1} \tag{1}$$

At $f(1)$, we get $f(1) = \frac{0}{0}$, which does not exist. But, as we approach for values $x \pm \varepsilon$ for $\varepsilon \in \mathbb{R}^+$ and ε is very small, $f(x) = 1$. Informally, since both sides "agree" on the fact $f(x)$ is really tending toward 1, we define the limit of $f(x)$ as x approaches 1 to be 1. Symbolically, this is written:

$$\lim_{x \to 1} f(x) = \lim_{x \to 1} \left(\frac{x-1}{x-1} \right) = 1$$

Limits to Infinity

In computer science, the most useful limit to consider is the limit of a function as it approaches infinity. That is, as the function's input grow larger, where does the function approach? Most algebraic and transcendental functions will also tend toward ∞. Consider a function that doesn't tend toward ∞:

$$\lim_{x \to \infty} \left(\frac{1}{x} \right) = 0$$

Intuitively, this is so because as the denominator gets larger and larger, it "dominates" the one in the numerator and brings the ratio closer and closer to zero.

Algorithmic Complexity: English

Intuitively, computer scientists are interested in how long it takes a computer to solve a problem. But how long something takes in real time is too complicated to address because run times will vary from Windows to Linux to iPads, etc. Instead, the theoretical bounds on the problem with respect to the input size is studied. This is an introduction to a broad field called **algorithmic analysis**.

Algorithmic Complexity: English

Intuitively, computer scientists are interested in how long it takes a computer to solve a problem. But how long something takes in real time is too complicated to address because run times will vary from Windows to Linux to iPads, etc. Instead, the theoretical bounds on the problem with respect to the input size is studied. This is an introduction to a broad field called **algorithmic analysis**.

The main measure of algorithmic complexity is known as $\mathcal{O}$ ("Big O") notation. The intuitive question $\mathcal{O}$ sets out to answer is, "What is the longest I can expect this program to take?" That is, as the problem grows in size, how much longer will this program take to process the additional input?

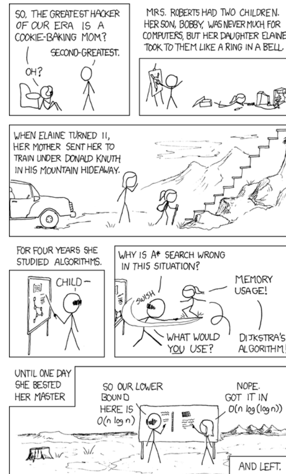

Figure 1: Adventures in $\mathcal{O}()$

Algorithmic Complexity: Graphical

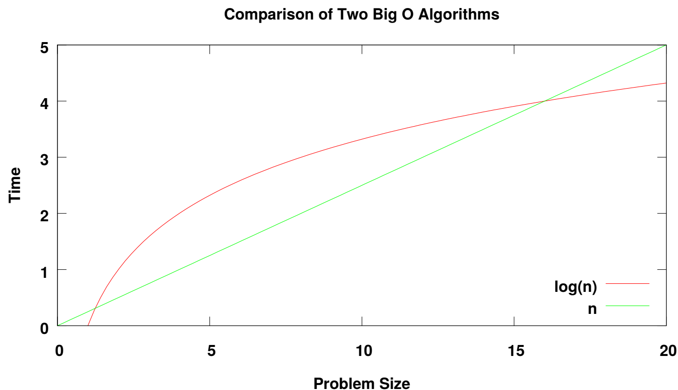

Figure 2: Two $\mathcal{O}()$ Functions

Algorithmic Complexity: Algebraic

Suppose the two functions in Figure 2 are two different programs that solve the same problem. The $\mathcal{O}(n)$ function is more efficient for small inputs and can be used when data sets a very small. The $\mathcal{O}(\log n)$[1] is more scaleable and suitable for larger inputs.

[1] In computer science, logarithms are base 2.

Algorithmic Complexity: Algebraic

Suppose the two functions in Figure 2 are two different programs that solve the same problem. The $\mathcal{O}(n)$ function is more efficient for small inputs and can be used when data sets a very small. The $\mathcal{O}(\log n)$[1] is more scaleable and suitable for larger inputs.

Note the $\mathcal{O}(n)$ function is not exactly $f(x) = x$, but instead $f(x) = \frac{1}{4}x$. This sets up one of the fundamental problems $\mathcal{O}$ answers: Given $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{R}^+$, when does is function $c \cdot f(n) \leq g(n)$ for all $n \geq n_0$? If such c and n_0 exist, it follows that $f(n) = \mathcal{O}(g(n))$.

[1] In computer science, logarithms are base 2.

Algorithmic Complexity: Algebraic

Suppose the two functions in Figure 2 are two different programs that solve the same problem. The $\mathcal{O}(n)$ function is more efficient for small inputs and can be used when data sets a very small. The $\mathcal{O}(\log n)$[1] is more scaleable and suitable for larger inputs.

Note the $\mathcal{O}(n)$ function is not exactly $f(x) = x$, but instead $f(x) = \frac{1}{4}x$. This sets up one of the fundamental problems $\mathcal{O}$ answers: Given $c \in \mathbb{R}^+$ and $n_0 \in \mathbb{R}^+$, when does is function $c \cdot f(n) \leq g(n)$ for all $n \geq n_0$? If such c and n_0 exist, it follows that $f(n) = \mathcal{O}(g(n))$.

In the example in Figure 2, $\log(n)$ is $\mathcal{O}(n)$ for $c = \frac{1}{4}$ and $n_0 = 16$.

[1] In computer science, logarithms are base 2.

Algorithmic Complexity: Calculus

Let $f(n)$ and $g(n)$ be two functions that return positive real numbers (i.e., $f : \mathbb{D}_f \to \mathbb{R}^+$ and $g : \mathbb{D}_g \to \mathbb{R}^+$). Define $f(n)$ as $\mathcal{O}(g(n))$ if and only if:

$$\lim_{n\to\infty} \left| \frac{f(n)}{g(n)} \right| < \infty \qquad (2)$$

That is, as n approaches ∞ in Equation 2, the quotient of the algorithm and the function will approach some value α. If α is a finite number, this implies that $f(n)$ and $g(n)$ are "sufficiently similiar" for the algorithm $f(n)$ as $\mathcal{O}(g(n))$. On the other hand, if $\alpha = \pm\infty$ (and the limit in Equation 2 therefore does not exist), then $f(n)$ and $g(n)$ are not similar and it is not the case that $f(n)$ is $\mathcal{O}(g(n))$.

Common $\mathcal{O}()$ Values

Value	Name	Example
$\mathcal{O}(1)$	Constant	Accessing an array member
$\mathcal{O}(\log n)$	Logarithmic	Searching an index or tree
$\mathcal{O}(n)$	Linear	Naïvely searching an array
$\mathcal{O}(n \log n)$	Loglinear	Sorting an array
$\mathcal{O}(n^2)$	Quadratic	Matrix multiplication
$\mathcal{O}(n^3)$	Cubic	Finding a determinant of a matrix
$\mathcal{O}(a^n)$	Exponential	Traveling salesman: dynamic programming
$\mathcal{O}(n!)$	Factorial	Traveling salesman: directly

Table 1: Common $\mathcal{O}$ Values & Examples

Impact

Each line of code has an associated $\mathcal{O}()$ value. As each line of code executes, the performance impact is felt. For instance, if we have three lines of code: two of which are $\mathcal{O}(n)$ and one is $\mathcal{O}(n \log n)$, the program is $\mathcal{O}(n \log n)$ since the $\mathcal{O}(n \log n)$ is larger and "dominates" the $\mathcal{O}(n)$ lines of code.

Impact

Each line of code has an associated $\mathcal{O}()$ value. As each line of code executes, the performance impact is felt. For instance, if we have three lines of code: two of which are $\mathcal{O}(n)$ and one is $\mathcal{O}(n \log n)$, the program is $\mathcal{O}(n \log n)$ since the $\mathcal{O}(n \log n)$ is larger and "dominates" the $\mathcal{O}(n)$ lines of code.

A loop has the general effect of multiplying the complexity into n. Using the three lines of code in the previous example, there are two lines of code that are $\mathcal{O}(n^2)$ and one that is $\mathcal{O}(n^2 \log n)$. Again, the $\mathcal{O}(n^2 \log n)$ "dominates" and the program is $\mathcal{O}(n^2 \log n)$.

Example

Theorem

A function that has an exact run time of $f(n) = 2^{n+10}$ *is* $\mathcal{O}(2^n)$.

Proof.

Rewrite $f(n)$ as $f(n) = 2^{10} \cdot 2^n = 1024 \cdot 2^n$. Defining the limit as seen in Equation 2, we get:

$$\lim_{n \to \infty} \left| \frac{1024 \cdot 2^n}{2^n} \right| = \lim_{n \to \infty} |1024| = 1024$$

Since, the limit exists, it follows that $f(n)$ is $\mathcal{O}(2^n)$. ∎

It should be noted that, in this case, $c = 1024$ and $n_0 = 10$, similar to what was seen in the Algebraic section.

Final Thought

The selection of algorithms and data structures often leads to very large consequences in how programs, whether on or off the web, perform. These are measured in $\mathcal{O}()$ form. Knowing common complexities for every day tasks is valuable and facilitates the informed choices of which algorithms to use as well being a powerful tool in performance optimization of programs.

Also, carefully consider using slower algorithms within loops, as they tend to be exacerbated by the fact loops generally introduce a factor of n to any operation being performed. This has been one of the largest sources of slow down in my professional experience.