

Introduction to Algorithms and Complexity

Guy Lebanon

September 26, 2006

An algorithm is a way of solving a problem, expressed in a way suitable for computer implementation. When describing an algorithm, it is essential to carefully specify the data structure which is the way in which the data is stored in memory. We are interested in whether the algorithm correctly completes its task, how long it takes, and how much memory is used in the process. In this note we focus on the measuring the amount of time an algorithm takes to complete its task.

Unfortunately we can't quantify that precise amount of time since it depends on the particular hardware, assembly language, compiler used and other factors. It is nearly impossible to account for all of these factors. Moreover, even if we could account for these factors and compute the precise running time - it would be useless when we replace the hardware or compiler or recode the algorithm in a different programming language. As a result, we choose to obtain a vague description of running time - that is hardware and software independent. Such a description is correct up to some constants in a manner that will be formalized below. What we gain in return for the vagueness is that our analysis is tied to the algorithm itself - rather than the particular details of its implementation, software and hardware used.

Since we give up on the idea of coming up with the precise running time, we can base our analysis on a typical implementation of the algorithm in a language such as C. The actual operations run by the processor (resulting from compiling the C code) will be generally different by a constant since each C command is translated into a bounded number of assembly language instructions. This is not the case of R or Matlab (consider for example square matrix multiplication - one command in R which takes time proportional to n^3). Expressing an algorithm in R or Matlab obscures the implementation and makes it impossible to analyze its complexity.

We assume that the algorithm can be run on input in different sizes. This size could be the number of data points, data dimensionality, or any factor (or combination of factors) related to the data that influences the running time. We will measure the running time not in absolute terms - but as functions of the above measure¹ n . In this way, we can examine not just the running time for one particular n , but more generally for all n . As we wish to ignore constant, we will say for example finding a particular number in an array of length n takes time proportional to n - that is the running time is $an + b$ for some constants a, b . The constants will be determined by the compiler and hardware and as described above can be neglected. A formal description of classes of functions, aggregated by the above mentioned vagueness is given below.

Definition 1.

$$\Theta(g(n)) = \{f(n) : \text{there exists } c_1, c_2, N \geq 0 \text{ such that } \forall n > N, 0 \leq c_1g(n) \leq f(n) \leq c_2g(n)\}$$

$$O(g(n)) = \{f(n) : \text{there exists } c, N \geq 0 \text{ such that } \forall n > N, 0 \leq f(n) \leq cg(n)\}$$

$$o(g(n)) = \{f(n) : \text{for any } c > 0, \text{ there exists } N \geq 0 \text{ such that } \forall n > N, 0 \leq f(n) < cg(n)\}$$

The third definition is sometimes expressed in a more intuitive manner as $o(g(n)) = \{f(n) : \lim_n \frac{f(n)}{g(n)} = 0\}$.

$\Theta(g)$ is thus the set of functions that asymptotically (for large n) grow in the same manner as g . $O(g)$ is the set of function that asymptotically grow in the same manner as g or slower. $o(g)$ is the set of functions

¹In some cases, we will have more than one measure of the data size - for example number of data points and dimensionality - m and n

that asymptotically grow strictly slower than g . It is easily seen that $o(g) \subset O(g)$ and $\Theta(g) \subset O(g)$. A conventional abuse of notation is to write $f(n) = O(g(n))$ and to insert the above classes into an equation - making the equation valid for all members of the class.

Examples: (a) $\frac{1}{2}n^2 - 3n = \Theta(n^2)$ since $c_1n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2n^2$ or $c_1 \leq \frac{1}{2} - 3/n \leq c_2$ is satisfied by $c_2 = 1/2, c_1 = 1/14, N = 7$. (b) $6n^3 \neq \Theta(n^2)$ since $6n \not\leq c_2/6$ for large n (for any value of c_2).

It can be easily shown that O is transitive (that is $f = O(g), g = O(h) \Rightarrow f = O(h)$) and Θ is transitive as well as symmetric (the latter part is non-trivial). As a result Θ induces an equivalence relation or partition of the set of functions into classes. All functions within a class are Θ of each other. Any two functions from two different classes are not Θ of each other. These equivalence classes are ordered (partially) by the relation O .

All polynomials of degree d are in the same equivalence class - which we denote by $\Theta(n^d)$. That is $n^6 = \Theta(n^6 - 13n^3 + 2n - 10)$ and vice versa. The classes of d -order polynomials form a chain of classes $\{\Theta(n^d) : d = 0, 1, \dots\}$ that we can order by the o relation: $n^k = o(n^{k+l})$ for $l > 0$. If we represent this ordering by the symbol \prec we have $\Theta(1) \prec \Theta(n) \prec \dots \prec \Theta(n^{10}) \prec \dots$. The exponential function e^n as well as 2^n are strictly larger in the sense that $n^d = o(e^n)$ for all d (this can be seen from expanding e^x as a power series). In a similar manner $\log(n) = o(n)$ and $n \log n = o(n^2)$ but $n = o(n \log n)$. Putting all of this into succinct notation we have

$$\Theta(1) \prec \Theta(\log(n)) \prec \Theta(n) \prec \Theta(n \log n) \prec \Theta(n^2) \prec \dots \prec \Theta(n^{10}) \prec \dots \prec \Theta(2^n).$$

Of course there are much more equivalence classes and the above hierarchy is only partial.

The above partition of all functions into complexity class using the relation Θ lets us formalize measuring running time of algorithm. For example, we will show that given n numbers in an array, it takes $O(n \log n)$ time to sort them. This means that the precise sorting time as a function of n is bounded asymptotically by $n \log n$.

A word of caution concerning using the coarse equivalence class formalism and neglecting the constants is in order. An algorithm whose running time is $O(2^n)$ can be much faster than an algorithm whose running time is $O(1)$, for all n of reasonable size (say the number of atoms in the universe). Similarly, the hidden constants $b_1 \in O(1)$ and $b_2 2^n$ can be extremely meaningful if b_1 is trillions of seconds. We need to understand that the complexity formalism above is asymptotic and the hidden constants may very well reverse conclusions that we arrive at by neglecting them. However, as mentioned above, it is very difficult to measure these constants and the measurement is hardware and software dependent and so we should in general (excluding a few extreme cases) be content with the asymptotic analysis of running time.

Before we conclude, we solve a few recursion relations that will be useful later on in measuring the complexity of algorithms. The usefulness comes from the fact that is often difficult to look at C code and count operations. Instead we can express running time for input size n as a function of the running time of smaller input sizes.

- $c_n = c_{n-1} + n$ for $n \geq 2$ with $c_1 = 1$. Expanding, we have $c_n = c_{n-1} + n = c_{n-2} + (n-1) + n = \dots = 1 + 2 + \dots + n = n(n+1)/2$ where the last equality follows from adding $1 + 2 + \dots + n$ to itself in reverse order and dividing by 2. An algorithm with such running time is $\Theta(n(n+1)/2) = \Theta(n^2)$.
- $c_n = c_{n/2} + 1$ for $n \geq 2$ with $c_1 = 1$ (assuming n is a power of 2). Expanding, we have $c_n = c_{n/2} + 1 = c_{n/4} + 1 + 1 = \dots = 1 + \dots + 1 = \log n + 1$ which is of complexity $\Theta(\log n)$.
- $c_n = 2c_{n/2} + n$ for $n \geq 2$ with $c_1 = 0$ (assuming n is a power of 2). Expanding, we have $c_{2^k} = 2c_{2^{k-1}} + 2^k = 2^2c_{2^{k-2}} + 2 \cdot 2^{k-1} + 2^k = \dots = (1 + \dots + 1)2^k = k2^k$ or $c_n = n \log n$.
- $c_n = 2c_{n/2} + 1$ for $n \geq 2$ with $c_1 = 1$. Expanding, we have $c_{2^k} = 2c_{2^{k-1}} + 1 = 2 \cdot 2c_{2^{k-2}} + 2 \cdot 1 + 1 = \dots = 1 + 2 + 4 + 8 + \dots + 2^{\log k}$ which gives us $c_n = \Theta(2n) = \Theta(n)$.

As a final example, we sometime insert equivalence classes into equations. For example $n + O(n) = O(n)$ should be interpreted by choosing some representative functions from every class in the expression and seeing if the class of the left hand side agrees with the class of the right hand side. Other examples are $O(n^2)O(n^3) + n = O(n^5) + O(n) = O(n^5)$ and $(n + O(1))(n + O(\log n) + O(1)) = n^2 + O(n) + O(n \log n) + O(\log n) + O(n) + O(1) = O(n^2)$.