2 Big Oh notation

To recap, the definition of $O(g(n)$, called 'big oh' of $g(n)$, is

$$O(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ and } c > 0 \in \mathbf{R} \text{ with } |f(n)| \leq c|g(n)| \, \forall n \geq n_0\} \tag{1}$$

and

$$f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{2}$$

In practice, if

$$T(n) = a_r n^r + a_{r-1} n^{r-1} + \ldots + a_1 n + a_0 \tag{3}$$

then $T(n) \in O(n^r)$.

The logarithm has the funny property that it goes to infinity, but it does so slower than n:

$$\lim_{n\to\infty} \frac{\log_2 n}{n} = 0 \tag{4}$$

Here, in line with standard practice in computer science, we are using the log to the base two, in fact changing bases only causes a change of an overall constant, see Table 1 for a reminder of the properties of the log. The limit of $\log_2 n/n$ can be calculated using l'Hôpital's rule, here we'll just look at a plot, Fig. 1. Now, just as $\log_2 n$ grows very slowly, 2^n grows very fast,

$$\lim_{n\to\infty} \frac{n^r}{2^n} = 0 \tag{5}$$

for any finite value of r, worse still is $n!$, pronounced n-factorial

$$n! = n(n-1)(n-2)...1 \tag{6}$$

If your algorithm is in $O(n!)$ you will probably need a different algorithm. A table of different values is given as Table 2, mostly to emphasis how quickly $n!$ gets big.

Now, in mathematics we call something 'an abuse of notation' if it is common to write something that doesn't quite make sense but acts as a shorthand for something that does. Now $O(g(n))$ is a set of functions whose large n behavior is bounded by $g(n)$ so in algorithms, being in $O(g(n))$ is a property of $T(n)$, the formula for the running time of the algorithm. However, it is a standard abuse of notation to say an algorithm is $O(g(n))$ for some $g(n)$ if $T(n) \in O(g(n))$ for all cases. This is another way of saying that the worst behavior of the algorithm isn't any worse than the behavior of $g(n)$ for large n so all possible $T(n)$ are elements of $O(g(n))$. Finding $O(g(n))$ for an algorithm will be refered to as 'finding the big-oh complexity'. In the context of a more precise approach to the topic we might talk about 'finding the asymptotic complexity'.

Other big Letter notations, small oh notation.

There is another set, $\Omega(g(n))$ with a definition similar to $O(g(n))$ that is used for describing the best case behavior. This requires a lower bound rather than an upper bound, so the obvious definition is

$$\Omega(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ and } c > 0 \in \mathbf{R} \text{ with } |f(n)| \geq c|g(n)| \, \forall n \geq n_0\} \tag{16}$$

The logarithm is the opposite of the exponent: if

$$a^b = c \tag{7}$$

then

$$\log_a c = b \tag{8}$$

or, written in one line

$$a^{\log_a c} = c \tag{9}$$

All the laws of logs can be worked out from the laws of exponents. Hence, since $a^0 = 1$ we have $\log_a 1 = 0$. In a similar way, other rules of logs can be deduced like

$$\begin{aligned} \log_a c_1 c_2 &= \log_a c_1 + \log_a c_2 \\ \log_a \frac{c_1}{c_2} &= \log_a c_1 - \log_a c_2 \\ \log_a c^d &= d \log_a c \end{aligned} \tag{10}$$

and so on.
As for the change of base, let $b = \log_{a_1} c$ so $a_1^b = c$. Now take the log to the base a_2 of both sides

$$b \log_{a_2} a_1 = \log_{a_2} c \tag{11}$$

and then solve for b

$$b = \frac{\log_{a_2} c}{\log_{a_2} a_1} \tag{12}$$

and, substituting back the formula for b

$$\log_{a_1} c = \frac{\log_{a_2} c}{\log_{a_2} a_1} \tag{13}$$

Thus we see, that changing bases is just a matter of a multiplicative factor. For example, to change from base e to base two

$$\log_2 x = \frac{\log_e x}{\log_e 2} \approx \frac{\log_e x}{0.6931} \tag{14}$$

Common bases are $\log_2 x$ used in computer science, $\log_e x$ sometimes written $\ln x$ used in mathematics and $\log_{10} x$ used in chemistry. The base two is used because of its link to bits and also, as we will see, because of its relationship with algorithms that divide data into two piles. The natural log $\ln x$ is used where differential equations are common since

$$\frac{d}{dx} \ln x = \frac{1}{x} \tag{15}$$

Table 1: A reminder about logarithms. This is a quick summary of some of the laws of logs.

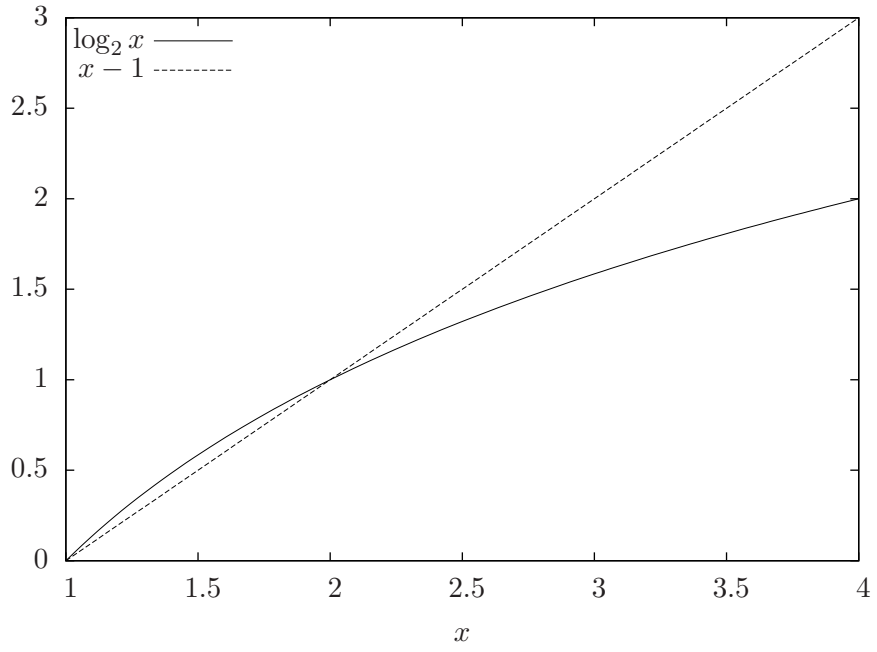

Figure 1: This shows $\log_2 x$ and $x - 1$ plots for $x \in [1, 4]$, the one has been taken from x to make them easier to compare, the key point is that the x grows faster.

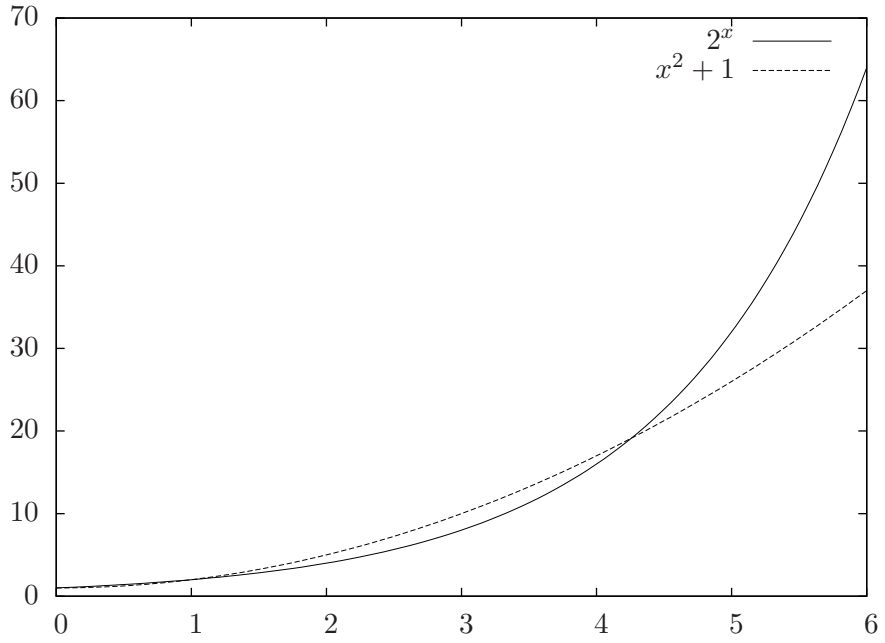

Figure 2: This shows 2^x and $x^2 - 1$ plots for $x \in [0, 6]$, clearly 2^x quickly overtakes $x^2 + 1$, this will happen for any power of x.

n	1	2	4	16	128	1024
$\log n$	0	1	2	4	7	10
$n \log n$	0	2	8	64	896	10240
n^2	1	4	16	256	16384	1048576
2^n	2	4	16	65536	3.4×10^{38}	1.8×10^{307}
$n!$	1	2	24	2.1×10^{13}	3.85×10^{305}	5.4×10^{2369}

The website
`http://markknowsnothing.com/cgi-bin/calculator.php`
was used for the 2^n calculations and
`http://www.calculatorsoup.com/calculators/discretemathematics/factorials.php`
for the $n!$ calculations; these give answers even when the answer is very large. Another easy way to calculate with large numbers is to use Python.

Table 2: Different values of n for some functions.

in other words, the same thing, but with the $\leq$ symbol replaced by a $\geq$. In fact, there is some ambiguity about this definition, number theorist use a slightly different one. Either way, it isn't used very often in computer science because algorithms are very frequently $\Omega(1)$; in the best case scenario the problem is in some sense already solve, the array already sorted for example, and the algorithm finishes in one step.

There is also a set of function that are both bounded above and below by the same $g(n)$

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n)) \tag{17}$$

This works because it is possible for

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{18}$$

for different c_1 and c_2. It would be very unusual for this to apply to an algorithm, it would mean that $T(n)$ has the same behavior for large n no matter whether it is the best case or the worst case scenario. There is a naïve largest element function in Table 3 which is $\Theta(n)$. It searches for the largest value in an unsorted array by looking at each element in turn. In fact, for a completely unsorted array this is the best algorithm, but, in practice, if finding the largest element in a set is an important and frequent procedure, a special data structure, called a heap, is used to keep track of which element is largest.

Finally, little oh notation is a stricter version of big Oh notation that is used in some more mathematical context, basically $f(n) \in o(g(n))$ is $f(n)$ is less than $cg(n)$ for any choice of c, if n is large enough:

$$o(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ so that } |f(n)| \geq c|g(n)| \, \forall n \geq n_0 \text{ and } \forall \, c > 0 \in \mathbf{R}\} \tag{19}$$

```
1  int search(int a[],int n)
2  {
3     int i;
4     int best_val=a[0];
5
6     for(i=1;i<n;i++){
7        if(a[i]>best_val)
8           best_val= a[i];
9     }
10
11   return best_val;
12 }
```

Table 3: Search for the largest element in an unsorted list. This function searches all the elements to see which is the largest, the inner loop always runs $n-1$ times since it doesn't know until it has looked at every element which is going to be the largest. This program is implemented as `find_largest.c`..