3 Big Oh notation

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

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

or, equivalently

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

In practice, if

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

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

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

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

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 1, 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

$$\Omega(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ and } c > 0 \in \mathbf{R} \text{ with } |f(n)| \geq c|g(n)| \, \forall n \geq n_0\} \tag{7}$$

or

$$f(n) \in \Omega(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} > 0 \tag{8}$$

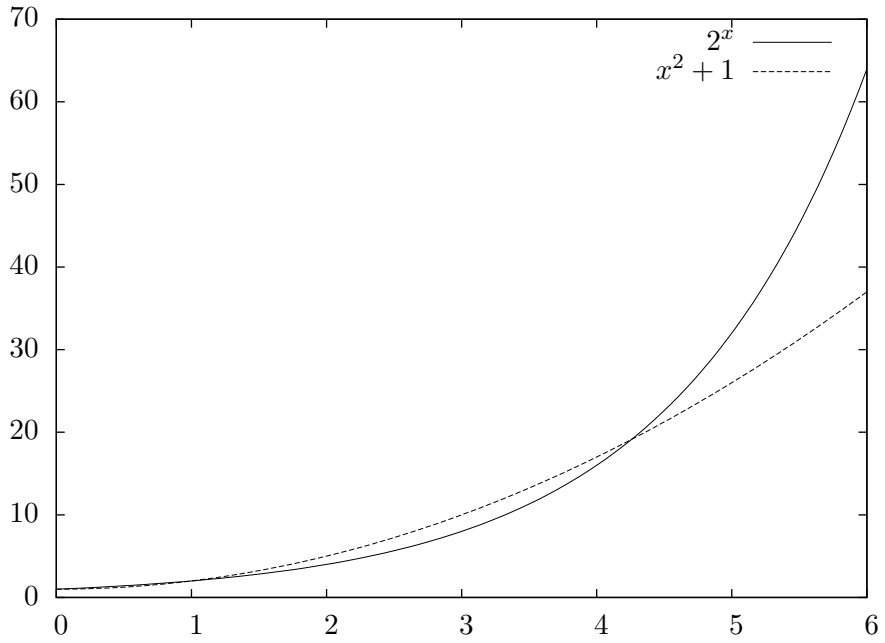

Figure 1: This shows 2^x and $x^2 - 1$ plots for $x \in [0, 6]$, clearly 2^x quickly overtakes $x^2 + 1$, this will happen for any power of x.

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 1: Different values of n for some functions.

```
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 2: 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`..

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 theorists 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 solved, the array already sorted for example, and the algorithm finishes in one step.

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

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n)) \tag{9}$$

This works because it is possible for

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{10}$$

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 2 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 greater than $cg(n)$ for any choice of c, if n is large enough:

$$o(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ so that } |f(n)| > c|g(n)| \, \forall n \geq n_0 \text{ and } \forall \, c > 0 \in \mathbf{R}\} \tag{11}$$