I.2 Big Oh notation

The 'Big Oh' notation has already been used to describe the behavior of the running time of insert sort, we said

$$T(n) \in O(n^2) \tag{1}$$

Here we want to formalize this notation. Basically $O(n^2)$ is a set of function, it is all the function which, for large values of n go to infinity like n^2 at the fastest. By saying $T(n) \in O(n^2)$ we are saying that $T(n)$ is one of these functions, its large n behavior is, as worst, like n^2.

Specifically, the definition for $g(n)$ for all n

$$O(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ and } c > 0 \in \mathbf{R} \text{ with } |f(n)| \leq c|g(n)| \, \forall n \geq n_0\} \tag{2}$$

This definition is quite dense, but we can break it down: it says that $O(g(n))$ is a set of functions, the curly brackets mean 'set'. $f(n)$ is in the set if it has a particular property: the '|' can be read as 'such that' or 'with the property that' and so this is the set of $f(n)$'s where $f(n)$ has the property on the right of the |. Now, '$\exists$' means 'there exists' and '$\forall$' means 'for all', so the defining property says it is possible to find a positive natural number n_0 and a positive real number c for that if you choose a value of n at least as big as n_0 then $f(n)$ is no bigger than $cg(n)$, $\mathbf{N}$ and $\mathbf{R}$ stand for the natural and real numbers. Notice the absolute value signs, this is about $|f(n)|$ and $|g(n)|$, in fact, here we are interested in run times, so we will deal with functions that are non-negative, or are non-negative provided n is larger than some threshold, for example, $\log_2 n$ will be important, $\log_2 n$ is positive provided $n > 1$.

In short, $f(n)$ can do all sorts of crazy stuff for small values of n but, if you take n large enough, its behavior is bounded by the behavior of $g(n)$. Now it doesn't say it is bounded by $g(n)$, it is a statement about the behavior, that's the role of the c. If you know the formal definition of limits you can see that the definition of $O(g(n))$ has this wrapped up in it, if says

$$f(n) \in O(g(n)) \iff \lim_{n\to\infty} \frac{f(n)}{g(n)} < \infty \tag{3}$$

Here are some examples, say

$$T(n) = 5n^2 + n + 6 \tag{4}$$

then

$$T(n) \in O(n^2) \tag{5}$$

by, for example, taking $c = 5 + 1 + 6 = 12$ then

$$12n^2 \geq 5n^2 + n + 6 \tag{6}$$

provided $n \geq 1$ so $n_0 = 1$ here. This could also be succinctly demonstrated using the limit

$$\lim_{n\to\infty} \frac{5n^2 + n + 6}{n^2} = \lim_{n\to\infty} 5 + \lim_{n\to\infty} \frac{1}{n} + \lim_{n\to\infty} \frac{6}{n^2} = 5 < \infty \tag{7}$$

However,

$$T(n) \notin O(n) \tag{8}$$

Say we chosen some value c then

$$5n^2 + n + 6 > cn \tag{9}$$

for large enough n, to check this divide both sides by n so we need to show that n can be chosen so that

$$5n + 1 + \frac{c}{n} > c \tag{10}$$

Since

$$5n + 1 + \frac{c}{n} > 5n + 1 \tag{11}$$

then, if $n > c/5$

$$5n + 1 + \frac{c}{n} > 5\frac{c}{5} + 1 = c + 1 > c \tag{12}$$

so, no matter what value of c is chosen, making $n > 5/n$ implies

$$5n^2 + n + 6 > cn \tag{13}$$

so $5n^2 + n + 6 \notin O(n)$. Again, the limit does the same job

$$\lim_{n\to\infty} \frac{5n^2 + n + 6}{n} = \lim_{n\to\infty} 5n + \lim_{n\to\infty} 1 + \lim_{n\to\infty} \frac{6}{n} = \infty \tag{14}$$

In practice, if

$$T(n) = a_r n^r + a_{r-1} n^{r-1} + \ldots + a_1 n + a_0 \tag{15}$$

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:

$$\lim_{n\to\infty} \frac{\log_2 n}{n} = 0 \tag{16}$$

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. 2. Now, just as $\log_2 n$ grows very slowly, 2^n grows very fast,

$$\lim_{n\to\infty} \frac{n^r}{2^n} = 0 \tag{17}$$

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

$$n! = n(n-1)(n-2)...1 \tag{18}$$

If you 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 $g(n)$ is $T(n) \in O(g(n))$ 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 $O(g(n))$.

The logarithm is the opposite of the exponent: if

$$a^b = c \tag{19}$$

then

$$\log_a c = b \tag{20}$$

or, written in one line

$$a^{\log_a c} = c \tag{21}$$

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

$$\begin{aligned} \log_a c_1 c_2 &= \log_a c_1 + \log_a c_2 \\ \log_a \frac{c_1}{c_2} &= \log_a c_1 - \log_a c_2 \\ \log_a c^d &= d \log_a c \end{aligned} \tag{22}$$

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

$$b \log_{a_2} a_1 = \log_{a_2} c \tag{23}$$

and then solve for b

$$b = \frac{\log_{a_2} c}{\log_{a_2} a_1} \tag{24}$$

and, substituting back the formula for b

$$\log_{a_1} c = \frac{\log_{a_2} c}{\log_{a_2} a_1} \tag{25}$$

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

$$\log_2 x = \frac{\log_e x}{\log_e 2} \approx \frac{\log_e x}{0.6931} \tag{26}$$

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

$$\frac{d}{dx} \ln x = \frac{1}{x} \tag{27}$$

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

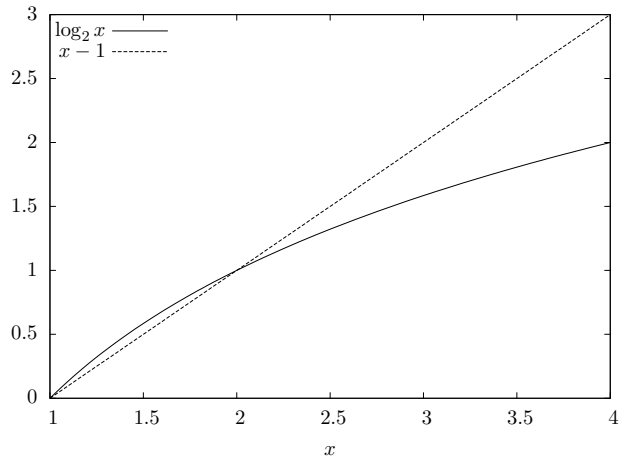

Figure 1: This shows $\log_2 x$ and $x - 1$ plots for $x \in [1, 4]$, the one has been taken from x to make them easier to compare, the key point is that the x grows faster.

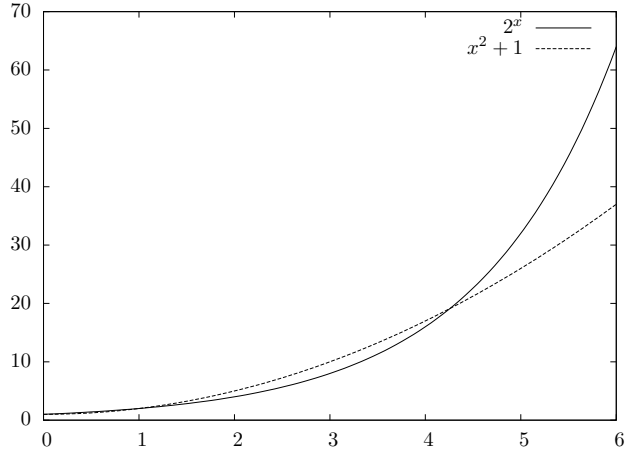

Figure 2: This shows 2^x and $x^2 - 1$ plots for $x \in [0, 6]$, clearly 2^x quickly overtakes $x^2 + 1$, this will happen for any power of x.

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.

Table 2: Different values of n for some functions.

I.2.1 Examples: linear and binary search

Lets do a quick example; searching for the index of an element in a sorted list. A completely terrible way to do this is linear search, this is terrible because it doesn't make use of the fact that the list is sorted. A code listing is given in Table 3. We can see straight away that the code between lines 6 and 10 is run n times in the worst case, everything else is run once and so this algorithm is $O(n)$.

A much better way to search a sorted array is binary search. This is an example of a 'divide and conquer' algorithm, many of the fastest algorithms use divide and conquer. It will be clear to you that this algorithm would be better written using recursion, this is typical of divide and conquer, but we haven't looked at analysing recursion yet. The idea is to divide the array in half and check which half, the half with bigger numbers or the half with smaller numbers, the value we are searching for belongs to and to keep doing this, dividing the remaining part of the array into two parts again and again until the remaining part of the array that is being search has only one element. A code listing is given in Table 4.

This search is extremely fast. There is a chance that a[mid]==val, after a small number of iterations, indeed, if the middle value of the array is the search value it will halt after only one iteration. However, as usual, we assume the worst case, in which case the algorithm runs to end, dividing the number of elements in half each time. Ignoring the integer rounding effects, it goes like Table 5. Starting with n states each subsequent iteration halves the number of states until the last one when there is one state left. Thus

$$1 = \frac{n}{2^{T(n)-1}} \tag{28}$$

and taking the log of both sides

$$0 = \log_2 n - (T(n) - 1) \tag{29}$$

using $\log_2 1 = 1$, $\log_2 a^b = b \log_2 a$ and $\log_2 2 = 1$. Hence

$$T(n) = \log_2 n + 1 \tag{30}$$

and this algorithm is $O(\log_2 n)$.


```
1  int search(int a[],int n, int val)
2  {
3
4    int i;
5
6    for(i=0;i<n;i++)
7      {
8        if(a[i]==val)
9          return i;
10     }
11
12   return -1;
13 }
```

Table 3: Linear search. This function searches the entries in the array a and returns the index when it finds val, if it doesn't find val it returns -1. The program `linear_search.c` implements this..

```
1  int search(int a[],int n, int val)
2  {
3    int mid, low=0, high=n-1;
4
5    while(low<=high)
6      {
7        mid=(low+high)/2;
8        if(a[mid]==val)
9          return mid;
10       else if(val>a[mid])
11         low=mid+1;
12       else
13         high=mid -1;
14     }
15
16   return -1;
17 }
```

Table 4: Binary search. This function starts in the middle of the array and checks if the value there is bigger or smaller than val, if it is bigger then it does the same in the top half of the array, if it is smaller, in the bottom half and then repeats until there are no elements left. The program `binary_search.c` implements this..

1	2	3	4	...	k	...	T(n)
n	$\frac{n}{2}$	$\frac{n}{4}$	$\frac{n}{8}$	...	$\frac{n}{2^{k-1}}$	...	1

Table 5: The number of elements left for binary search.

So, to reiterate; the usual way to examine the behavior of an algorithm is to look at the worst case run time. This is because the best case run time is often exceptional, like the one for binary search if the first guess happens to be correct. The average run time is often hard to calculate, both because it is often difficult to do the mathematics and because it would often mean having some description of how the initial data is distributed. Typically the worst run time is also 'of the same order' as the average run time. We will see an exception to this later on in the case of quick sort in which the worst case behavior is unusual. The big-Oh notation is used for describing an algorithm, if the algorithm is said to be $O(g(n))$ we mean $T(n) \in O(g(n))$ no matter what the initial condition. Since $O(g(n))$ involves an upper bound $T(n) < cg(n)$ this makes sense.

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

$$\Omega(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ and } c > 0 \in \mathbf{R} \text{ with } |f(n)| \geq c|g(n)| \, \forall n \geq n_0\} \tag{31}$$

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 finishs in one step.

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

$$\Theta(g(n)) = \Omega(g(n)) \cap O(g(n)) \tag{32}$$

This works because it is possible for

$$c_1 g(n) \leq f(n) \leq c_2 g(n) \tag{33}$$

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 6 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:

$$\emptyset(g(n)) = \{f(n) | \exists n_0 > 0 \in \mathbf{N} \text{ so that } |f(n)| \geq c|g(n)| \, \forall n \geq n_0 \text{ and } \forall \, c > 0 \in \mathbf{R}\} \tag{34}$$


```
1  int search(int a[],int n)
2  {
3
4    int i;
5    int best_val=a[0];
6
7    for(i=1;i<n;i++)
8      {
9        if(a[i]>best_val)
10         best_val= a[i];
11     }
12
13   return best_val;
14 }
```

Table 6: 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`..