Big O Notation

P. Danziger

1 Comparing Algorithms

We have seen that in many cases we would like to compare two algorithms. Generally, the *efficiency* of an algorithm can be guaged by how long it takes to run as a function of the size of the input. For example the efficiency of a graph algorithm might be measured as a function of the number of vertices n or edges m of the input graph. So, the Eulerian algorithm finds an Eulerian circuit after roughly m steps, where m is the number of edges in the input graph. and the Hamiltonian in roughly n^k, where k is the maximum degree in the graph.
When considering the efficiency of an algorithm we always consider the worst case.
For example, the Hamiltonian circuit problem can be solved in at most roughly n^k steps, by considering all possible circuits. We might be lucky and discover a Hamiltonian circuit on the first try, but we assume the worst case.
We now consider what we mean by *roughly*. We say that two functions f and g are of thee same order if they behave similarly for large values of n, i.e. $f(n) \approx g(n)$ for large n.
Consider the functions $f(n) = n^2$ and $g(n) = n^2 + 2n + 3$

n	1	10	20	50	100	200	300	400	500	1000
n^2	1	100	400	2500	10000	40000	90000	160000	250000	1000000
n^2+2n+3	6	123	443	2603	10203	40403	90603	160803	251003	1002003
2^n	2	1024	1048576	10^{15}	10^{30}	10^{60}	2×10^{90}	2.5×10^{120}	3×10^{150}	10^{301}

Clearly n^2 and $n^2 + 2n + 3$ are of the same order, whereas 2^n is not.

2 Big O

Definition 1 *Given a function $f : \mathbb{R} \longrightarrow \mathbb{R}$, or the corresponding sequnce $a_n = f(n)$.*

1. $\Omega(f)$ = functions that are of equal or greater order than f. All functions that are larger than some constant multiple of $|f(x)|$ for sufficiently large values of x.
$$\Omega(f) = \{g(x) \mid \exists\, A, a \in \mathbb{R}^+ \text{ such that } A|f(x)| \le |g(x)| \text{ for all } x > a\}$$
2. $O(f)$ = functions that are of equal or less order than f. All functions that are smaller than some constant multiple of $|f(x)|$ for sufficiently large values of x.
$$O(f) = \{g(x) \mid \exists\, A, a \in \mathbb{R}^+ \text{ such that } |g(x)| \le A|f(x)| \text{ for all } x > a\}$$
3. $\Theta(x)$ = functions that are of the same order as f. All functions that are always between two constant multiples of $|f(x)|$ for sufficiently large values of x.
$$\Theta(f) = \{g(x) \mid \exists\, A, B, a \in \mathbb{R}^+ \text{ such that } A|f(x)| \le |g(x)| \le B|f(x)| \text{ for all } x > a\}$$

4. $o(f)$ = functions that are of strictly lesser order than f. $o(f) = O(f) - \Theta(f)$

If $g \in O(f)$ we say g is of order f, many authors abuse notation by writing $g = O(f)$.
Alternately, we can define order by using the notion of the limit of the ratio of sequences tending to a value.

Definition 2 *Given a function $f : \mathbb{R} \longrightarrow \mathbb{R}$, or the corresponding sequnce $a_n = f(n)$.*

1. $\Omega(f)$.
$$\Omega(f) = \left\{ g(x) \;\middle|\; \lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| > 0 \right\}.$$
2. $O(f)$
$$O(f) = \left\{ g(x) \;\middle|\; \lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| < \infty \right\}.$$
3. $\Theta(f)$
$$\Theta(f) = \left\{ g(x) \;\middle|\; \lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| = L,\ 0 < L < \infty \right\}.$$
4. $o(f)$
$$o(f) = \left\{ g(x) \;\middle|\; \lim_{x\to\infty} \left|\frac{g(x)}{f(x)}\right| = 0 \right\}.$$

- $g \in \Omega(f)$ means that g is of equal or greater order than f.
- $g \in \Theta(f)$ means that f and g are roughly the same order of magnitude.
- $g \in O(f)$ means that g is of lesser or equal magnitude than f.
- $g \in o(f)$ means that g is of lesser magnitude than f.

Big O is by far the most commonly used and it is very common to say $f \in O(g)$ (f is at most order g) when what is really meant is that $f \in \Theta(g)$ (f and g are the same order).

Example 3
Show that $2^n \in o(3^n)$.
Consider $\displaystyle\lim_{n\to\infty} \frac{2^n}{3^n} = \lim_{n\to\infty} \left(\frac{2}{3}\right)^n = 0$. So $2^n \in o(3^n)$.
(Also $\displaystyle\lim_{n\to\infty} \frac{3^n}{2^n} = \lim_{n\to\infty} \left(\frac{3}{2}\right)^n = \infty$. So $3^n \in \Omega(2^n)$.)

Theorem 4 (9.2.1) *Let f, g, h and k be real valued functions defined on the same set of nonnegative reals, then:*

1. $\Omega(f) \cap O(f) = \Theta(f)$.
2. $f \in \Omega(g) \Leftrightarrow g \in O(f)$.

3. (Reflexivity of O) $f \in O(f)$, $f \in \Omega(f)$ and $f \in \Theta(f)$.
4. (Transitivity of O) If $f \in O(g)$ and $g \in O(h)$ then $f \in O(h)$.
5. If $f \in O(g)$ and $c \in \mathbb{R} - \{0\}$ then $c \cdot f(x) \in O(g)$
6. If $f \in O(h)$ and $g \in O(k)$ then $f(x) + g(x) \in O(G(x))$ where $G(x) = \max(|h(x)|, |k(x)|)$.
7. If $f(x) \in O(h)$ and $g \in O(k)$ then $f(x) \cdot g(x) \in O(h(x) \cdot k(x))$.

In fact a more useful rule than 6 is:

8. If $f \in O(h)$ and $g \in O(h)$ then $f(x) + g(x) \in O(h)$.
9. If $f \in o(g)$ then $g \in \Omega(f) - \Theta(f)$
10. $o(f) \subseteq O(f)$.

3 Types of Order

The orders form a hierarchy in the sense that if g is in a lower member of the hierarchy it is in all higher members. This is essentially the statement of transitivity (4).
The following is a list of common types of orders and their names:

Notation	Name
$O(1)$	Constant
$O(\log(n))$	Logarithmic
$O(\log(\log(n))$	Double logarithmic (iterative logarithmic)
$o(n)$	Sublinear
$O(n)$	Linear
$O(n\log(n))$	Loglinear, Linearithmic, Quasilinear or Supralinear
$O(n^2)$	Quadratic
$O(n^3)$	Cubic
$O(n^c)$	Polynomial (different class for each $c > 1$)
$O(c^n)$	Exponential (different class for each $c > 1$)
$O(n!)$	Factorial
$O(n^n)$	- (Yuck!)

Rules 5 and 8 are particularly useful for finding which class a function belongs.

Example 5

1. $2^n + n^2 \in O(2^n)$
2. $2n^3 + 3n^2 - 2n + 5$ is $O(n^3)$ (cubic).
3. $(2n^3 + 3n^2 - 2n + 5)^{\frac{1}{3}}$ is $O(n)$ (linear).

In general a polynomial is of the order of its highest term.

Theorem 6 *Given a polynomial f, if the highest power of f is x^r:*

- *If $r < s$ then $f \in o(x^s) \subseteq O(x^s)$.*
- *If $r = s$ then $f \in \Theta(x^s) \subseteq O(x^s)$.*
- *If $r > s$ then $f \in \Omega(x^s)$.*

Proof: Let f be a polynomial with highest power x^r. So $f(x) = a_r x^r + a_{r-1}x^{r-1} + \ldots + a_1 x + a_0$. Now let $s \in \mathbb{R}$ and consider

$$\begin{aligned}
\lim_{x\to\infty} \frac{f(x)}{x^s} &= \lim_{x\to\infty} \frac{a_r x^r + a_{r-1}x^{r-1} + \ldots + a_1 x + a_0}{x^s} \\
&= \lim_{x\to\infty} a_r x^{r-s} + a_{r-1}x^{r-1-s} + \ldots + a_1 x^{1-s} + a_0 x^{-s} \\
&= \begin{cases} 0 & r < s \\ a_r & r = s \\ \infty & r > s \end{cases}
\end{aligned}$$

Example 7

1. $2n^3 + 3n^2 - 2n + 5 \in O(n^3)$
2. $(10^{123})\, n^2 - 3670n + 5 \in o(n^3)$
3. $0.000000001n^3 \in \Omega(n^2)$

4 Algorithms

The primary use of order notation in computer science is to compare the efficiency of algorithms. Big O notation is especially useful when analyzing the efficiency of algorithms. In this case n is the size of the input and $f(n)$ is the running time of the algorithm relative to input size.
Suppose that we have two algorithms to solve a problem, we may compare them by comparing their orders.
In the following table we suppose that a linear algorithm can do a problem up to size 1000 in 1 second. We then use this to calculate how large a problem this algorithm could handle in 1 minute and 1 hour. We also calculate the largest tractable problem for other algorithms with the given speeds.

Running time	Maximum problem size		
	1 sec	1 min.	1 hr.
n	1000	6×10^4	3.6×10^6
$n\log(n)$	140	4893	2×10^5
n^2	31	244	1897
n^3	10	39	153
2^n	9	15	21

In the following table we suppose that the next generation of computers are 10 times faster than the current ones. We then calculate how much bigger the largest tractable problem is for the given algorithm speed.

Running time	After 10$\times$ speedup
n	$\times 10$
$n \log(n)$	$\approx \times 10$
n^2	$\times 3.16$
n^3	$\times 2.15$
2^n	$+3.3$

In general, an algorithm is called <u>efficient</u> if it is polynomial time, ie $O(n^k)$ for some constant k.
Of course for small values of n even an exponential algorithm can outpreform a polynomial one.
Given two problems, if there is a polynomial time algorithm which reduces an instance of one problem to an instance of the other, we say that the two problems are polynomial time equivalent.
Clearly if two problems are polynomial time equivalent and we have a polynomial time algorithm for one, then we have a polynomial time algorithm for the other.
The set of all polynomial time algorithms is denoted P.
There is another set of algorithms called NP (nondeterministic polynomial). It is known that $P \subseteq NP$, but unknown whether $P = NP$.
Within NP there are a set of algorithms called NP-complete which are all polynomial time equivalent to one another. NP-complete problems are as hard (computationally) as any problem in NP.
If we could find a polynomial time algorithm for any NP-complete problem we would have shown P = NP.
Determining if a graph has a Hamiltonian path is an example of an NP-complete problem.