# 5 Complexity

We have already used the **O** notation to denote the general behaviour of an algorithm as a function of the problem size. We have said that an algorithm is $\mathbf{O}(\log n)$ if its running time, $T(n)$, to solve a problem of size **n** is proportional to $\log n$.

## 5.1 The O notation

Formally, $O(g(n))$ is the *set* of functions, $f$, such that for some $c > 0$,

$$f(n) < cg(n)$$

for all positive integers, $n > N$, ie for all sufficiently large $N$. Another way of writing this is:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \leq c$$

Informally, we say the $O(g)$ is the set of all functions which grow *no faster* than $g$. The function $g$ is an *upper bound* to functions in $O(g)$.
We are interested in the set of functions defined by the $O$ notation because we want to argue about the relative merits of algorithms - independent of their implementations. That is, we are not concerned with the language or machine used; we want a means of comparing algorithms which is relevant to *any* implementation.
We can define two other functions: $\Omega(g)$ and $\Theta(g)$ .
$\Omega(g)$ the set of functions $f(n)$ for which $f(n) \geq cg(n)$ for all positive integers, $n > N$, and

$$\Theta(g) = \Omega(g) \cap O(g)$$

We can derive:

$$f \in \Theta(g)$$

if

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c$$

Thus, $\Omega(g)$ is a lower bound - functions in $\Omega(g)$ grow *faster* than $g$ and $\Theta(g)$ are functions that grow *at the same rate* as $g$. In these last two statements - as in most of the discussion on complexity theory - "within a constant factor" is understood. Different languages, compilers, machines, operating systems, etc will produce different constant factors: it is the general behaviour of the running time as $n$ increases to very large values that we're concerned with.

## 5.2 Properties of the O notation

The following general properties of **O** notation expressions may be derived:

1. Constant factors may be ignored:

   For all $k > 0$, **kf** is $\mathbf{O}(f)$.

   *e.g.* $an^2$ and $bn^2$ are both $\mathbf{O}(n^2)$.

2. Higher powers of **n** grow faster than lower powers:

   $n^r$ is $\mathbf{O}(n^s)$ if $0 \leq r \leq s$.

3. The growth rate of a sum of terms is the growth rate of its fastest growing term:

   If **f** is $\mathbf{O}(g)$, then $\mathbf{f} + \mathbf{g}$ is $\mathbf{O}(g)$.

   *e.g.* $an^3 + bn^2$ is $\mathbf{O}(n^3)$.

4. The growth rate of a polynomial is given by the growth rate of its leading term (*cf.* (2), (3)):

   If **f** is a polynomial of degree **d**, then **f** is $\mathbf{O}(n^d)$.

5. If **f** grows faster than **g**, which grows faster than **h**, then **f** grows faster than **h**.

6. The product of upper bounds of functions gives an upper bound for the product of the functions:

   If **f** is $\mathbf{O}(g)$ and **h** is $\mathbf{O}(r)$, then $fh$ is $\mathbf{O}(gr)$

   *e.g.* if **f** is $\mathbf{O}(n^2)$ and **g** is $\mathbf{O}(\log n)$, then **fg** is $\mathbf{O}(n^2 \log n)$.

7. Exponential functions grow faster than powers:

   $n^k$ is $\mathbf{O}(b^n)$, for all $b > 1, k \geq 0$,

   *e.g.* $n^4$ is $\mathbf{O}(2^n)$ and $n^4$ is $\mathbf{O}(exp(n))$.

8. Logarithms grow more slowly than powers:

   $\log_b n$ is $\mathbf{O}(n^k)$ for all $b > 1, k > 0$

   *e.g.* $\log_2 n$ is $\mathbf{O}(n^{0.5})$.

9. All logarithms grow at the same rate:

   $\log_b n$ is $\Theta(\log_d n)$ for all $b, d > 1$.

10. The sum of the first **n** $r^{th}$ powers grows as the $(r+1)^{th}$ power:

$$\sum_{k=1}^{n} k^r is \Theta(n^{r+1})$$

$$e.g.\ \sum_{k=1}^{n} i = \frac{(n+1)n}{2} is \Theta(n^2))$$

## 5.3 Polynomial and Intractable Algorithms

### 5.3.1 Polynomial time complexity

An algorithm is said to have *polynomial time complexity iff* it is $\mathbf{O}(n^d)$ for some integer $\mathbf{d}$.

### 5.3.2 Intractable Algorithms

A problem is said to be *intractable* if no algorithm with polynomial time complexity is known for it. We will briefly examine some intractable problems in a later section.

## 5.4 Analysing an algorithm

### 5.4.1 Simple Statement Sequence

First note that a sequence of statements which is executed once only is $\mathbf{O}(1)$. It doesn't matter how many statements are in the sequence - only that the number of statements (or the time that they take to execute) is constant for all problems.

### 5.4.2 Simple Loops

If a problem of size **n** can be solved with a simple loop:

```
for(i=0;i<n;i++)
    { s; }
```

where s is an $\mathbf{O}(1)$ sequence of statements, then the time complexity is $\mathbf{nO}(1)$ or $\mathbf{O}(n)$.
If we have two nested loops:

```
for(j=0;j<n;j++)
    for(i=0;i¡n;i++)
        { s; }
```

then we have **n** repetitions of an $\mathbf{O}(n)$ sequence, giving a complexity of: $\mathbf{nO}(n)$ or $\mathbf{O}(n^2)$.
Where the index 'jumps' by an increasing amount in each iteration, we might have a loop like:

```
h = 1;
while( h ≤ n )
    { s;
    h = 2*h; }
```

in which `h` takes values 1, 2, 4, ... until it exceeds **n**. This sequence has $1 + \lfloor \log_2 n \rfloor$ values, so the complexity is $\mathbf{O}(\log_2 n)$.
If the inner loop depends on an outer loop index:

```
for(j=0;j<n;j++)
    for(i=0;i<j;i++)
        { s; }
```

The inner loop `for(i=0; ..` gets executed **i** times, so the total is:

$$\sum_{1}^{n} i = \frac{n(n+1))}{2}$$

and the complexity is $\mathbf{O}(n^2)$. We see that this is the same as the result for two nested loops above, so the variable number of iterations of the inner loop doesn't affect the 'big picture'.

However, if the number of iterations of one of the loops decreases by a constant factor with every iteration:

```
h = n;
for(j=0;j<h;j++)
    {
    for(i=0;i<n;i++)
        { s; }
    h = h/2;
    }
```

Then: There are $\log_2 n$ iterations of the outer loop and the inner loop is $\mathbf{O}(n)$, so the overall complexity is $\mathbf{O}(n \log n)$. This is substantially better than the previous case in which the number of iterations of one of the loops decreased by a constant for each iteration!