Asymptotic Notation

Asymptotic notation deals with the behaviour of a function in the limit, that is, for sufficiently large values of its parameter. Often, when analysing the run time of an algorithm, it is easier to obtain an approximate formula for the run-time which gives a good indication of the algorithm performance for large problem instances. For example, suppose the exact run-time *T(n)* of an algorithm on an input of size n is $T(n) = 5n^2 + 6n + 25$ seconds**.** Then, since n is ≥ 0, we have $5n^2 \leq T(n) \leq 6n^2$ for all $n \geq 9$. Thus we can say that *T(n)* is roughly proportional to n^2 for sufficiently large values of n. We write this as $T(n) \in \Theta(n^2)$, or say that "*T(n)* is in the exact order of n^2**"**.

The main feature of this approach is that we can ignore constant factors and concentrate on the terms in the expression for *T(n)* that will dominate the function's behaviour as n becomes large. This allows us to partition run-time functions into broad efficiency classes. Generally, an algorithm with a run-time of $\Theta(n \log n)$ will perform better than an algorithm with a run-time of order $\Theta(n^2)$**,** provided that n is sufficiently large. However, for small values of n the $\Theta(n^2)$ algorithm may run faster due to having smaller constant factors. The value of n at which the $\Theta(n \log n)$ algorithm first outperforms the $\Theta(n^2)$ algorithm is called the *break-even* point for the $\Theta(n \log n)$ algorithm.

We now define this notation more precisely.

Definition

Let *f(n)* be an arbitrary function from the natural numbers $N = \{0,1,2,\ldots\}$ to the set $R^{\geq 0}$ of nonnegative reals. Let R^+ denote the set of positive reals. Define

$$O(f(n)) = \{t : N \to R^{\geq 0} \ : \ \exists c \in R^+ \ \exists n_0 \in N \ \forall n \geq n_0 \ (t(n) \leq cf(n))\}$$

In other words, $O(f(n))$ - pronounced "big Oh of *f(n)*" – is the *set* of all functions $t : N \to R^{\geq 0}$ such that $t(n) \leq cf(n)$ for all $n \geq n_0$ for some positive real constant c and integer threshold n_0. Notice that $O(f(n))$ is a set, so that we refer to a function *t(n)* as being *in the order of f(n)* if $t(n) \in O(f(n))$. This differs from the way in which $O(f(n))$ is sometimes defined elsewhere in the literature.

Example

Suppose that an algorithm takes in the worst case $t(n) = 27n^2 + \frac{355}{113}n + 12$ microseconds on a certain computer to solve an instance of size n of a problem. We can show that $t(n)$ is in the order of n^2 by the following argument:

$$\begin{aligned} t(n) &= 27n^2 + \tfrac{355}{113}n + 12 \\ &\le 27n^2 + \tfrac{355}{113}n^2 + 12n^2 \quad \text{(provided } n \ge 1\text{)} \\ &= 42\tfrac{16}{113}n^2 \end{aligned}$$

Thus there exist constants n_0 $(=1)$ and $c(=42\frac{16}{113})$ such that $t(n) \le cf(n)$ for all $n \ge n_0$. That is, $t(n) \in O(n^2)$.

Principle of Algorithm Invariance

The *principle of invariance* states that implementations of the same algorithm on different computers will not differ in efficiency by more than some multiplicative constant. More precisely, if two implementations of the same algorithm take $t_1(n)$ and $t_2(n)$ seconds, respectively, to solve a problem instance of size n, then there exist positive constants c and d such that $t_1(n) \le ct_2(n)$ and $t_2(n) \le dt_1(n)$ whenever n is sufficiently large. In other words, the running time of either algorithm is bounded by a constant multiple of the running time of the other.

This principle is not something that we can prove: it simply states a fact that can be confirmed by observation. Moreover it has very wide application. The principle remains true whatever the computer used to implement an algorithm, regardless of the programming language and the compiler used, and regardless of the skill of the programmer (provided he or she does not actually modify the algorithm).

[Note: we are assuming that the computer used is of conventional design. Much research lately has been carried out on investigating whether new computers, based on DNA operations or on quantum physics could be built in the future. If they could be constructed, then they would offer massive improvements in problem solving capabilities, even to the extent that problems unsolvable on conventional machines (such as the Halting Problem) could in fact be solved on he new machines.]

Thus a change of machine may allow us to solve a problem 10 times or 100 times faster, giving an increase in speed by a constant factor. Also, clever fine tuning techniques, such as shifting constant evaluations outside loops or the use of registers for frequently accessed variables, can have similar effects. A change of algorithm, on the other hand, may give us an

improvement that gets more and more marked as the size of the instance increases.

For this reason, when referring to the running time, *t(n)*, of an algorithm, we don't need to worry about whether the unit of time is microseconds, milliseconds, seconds, or minutes, for example. This only affects the running time by a constant factor, which is irrelevant when we are analysing the order of *t(n)*.

A nice consequence of all this is that the order of the running time of an algorithm is a property of the algorithm itself, rather than its implementation. This enables us to split algorithms into broad classes according to the orders of their running times. For example, an $O(n^2)$ sorting algorithm runs in time $O(n^2)$ on any conventional computer, in any language, under any compiler, and with any programmer. For large enough n it will perform worse than any $O(n\log n)$ sorting algorithm. Common algorithm orders are $\log n$ (logarithmic), n (linear), $n\log n$, n^2 (quadratic), n^3 (cubic), n^k (polynomial), and c^n (exponential), where k is an integer constant ≥ 1, and c is a constant >1.

More on asymptotic notation

Omega notation

An algorithm that is $O(n\log n)$ is also $O(n^2)$, or $O(n^3)$, or even $O(c^n)$. This is because the O notation really only provides us with an indication of an asymptotic upper bound for the running time of the algorithm. Thus it may not be helpful in revealing the true running time if the bound is not particularly tight. Clearly we also need a notation for *lower bounds*. This is provided by the *omega* or Ω notation, which is defined as follows:

$$\Omega(f(n)) = \{t : N \to R^{\geq 0} \ : \ \exists c \in R^+ \ \ \exists n_0 \in N \ \ \forall n \geq n_0 \ \ (t(n) \geq cf(n))\}$$

Note that this is the same as the definition of *O(f(n)* except that we have replaced the $\leq$ relation by $\geq$. Again note that $\Omega(f(n))$ is a *set* of functions, so that we refer to a function *t(n)*as *"being in* $\Omega(f(n))$*"*.

We now state the **duality rule**:

$t(n) \in O(f(n))$ if and only if $f(n) \in \Omega(t(n))$.

This is easily seen by noting that $t(n) \leq df(n)$ if and only if $f(n) \geq \frac{1}{d}t(n)$.

Note that we need to be a little careful with the omega notation when referring to the *worst-case* running times of algorithms. Assume that *t(n)* is the worst-case running time of an algorithm, and that $t(n) \in \Omega(f(n))$. This means that there exists a real positive constant d such that

$t(n) \geq df(n)$ for all sufficiently large n. Because *t(n)* denotes the worst-case behaviour of an implementation of the algorithm, we may infer that, for each sufficiently large value of n, there exists *at least one* instance of size n such that the implementation takes at least *df(n)* microseconds on that instance. This does not rule out the possibility of much faster behaviour on other instances of the same size. Thus, there may be infinitely many instances on which the implementation takes less than *df(n)* microseconds. For example, the worst case running time of insertion sort is both in $O(n^2)$ and $\Omega(n^2)$. However, there are infinitely many instances on which insertion sort takes *linear* time.

The Ω notation is useful in that it gives us an indication of the best efficiency we can achieve when designing an algorithm to solve a particular problem. For example, it can be shown that any algorithm that successfully sorts n elements must take a time in $\Omega(n \log n)$, provided that the only operation carried out on the elements to be sorted is to compare them pair-wise to determine whether they are equal and, if not, which is the greater. (This is called a *comparison-based* sorting algorithm.) As a result we say that the problem of sorting by comparisons has running time *complexity* in $\Omega(n \log n)$. This allows us to partition *problems* into broad complexity classes based on lower bounds on the performance of algorithms used to solve them.

Note that we still have a problem in that comparison-based sorting also has complexity $\Omega(n)$, even though an *O(n)* algorithm is impossible to achieve. Thus we need a notation that provides us with *tight* lower (and upper) bounds.

Theta notation

We say that *t(n)* is in Theta of *f(n)*, or equivalently that *t(n)* is in the *exact order* of *f(n)*, denoted by $t(n) \in \Theta(f(n))$, if *t(n)* belongs to both *O(f(n))* and $\Omega(f(n))$. The formal definition of Θ is

$$\Theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

This is equivalent to saying that

$$\Theta(f(n)) = \{t : N \to R^{\geq 0} \ : \ \exists c,d \in R^+ \ \ \exists n_0 \in N \ \ \forall n \geq n_0 \ \ (df(n) \leq t(n) \leq cf(n))\}$$

The maximum rule

A useful tool for proving that one function is in the order of another is the **maximum rule**.

Let $f,g : N \to R^{\geq 0}$ be two arbitrary functions from the natural numbers to the nonnegative reals. The maximum rule says that

$O(f(n)+g(n)) = O(\max(f(n),g(n)))$. More specifically, let $p,q : N \to R^{\geq 0}$ be defined for each natural number n by $p(n) = f(n)+g(n)$ and $q(n) = max(f(n),g(n))$, and consider an arbitrary function $t : N \to R^{\geq 0}$. The maximum rule says that $t(n) \in O(p(n))$ if and only if $t(n) \in O(q(n))$.

The maximum rule also applies with Θ notation.

Exercise

Prove the maximum rule.

Example

Consider an algorithm that proceeds in three steps: initialisation, processing and finalisation, and that these steps take time in $\Theta(n^2)$, $\Theta(n^3)$ and $\Theta(n \log n)$ respectively. It is therefore clear that the complete algorithm takes a time in $\Theta(n^2 + n^3 + n \log n)$. From the maximum rule

$$\begin{aligned}\Theta(n^2 + n^3 + n\log n) &= \Theta(\max(n^2, n^3 + n\log n)) \\ &= \Theta(\max(n^2, \max(n^3, n\log n))) \\ &= \Theta(\max(n^2, n^3)) \\ &= \Theta(n^3)\end{aligned}$$

From the above we would suspect that the maximum rule generalises to any finite constant number of functions, and this is indeed true.

Be careful not to use the rule if some of the functions are negative infinitely often, otherwise you risk reasoning as follows:

$$\Theta(n) = \Theta(n + n^2 - n^2) = \Theta(\max(n, n^2, -n^2) = \Theta(n^2)$$

The reasoning is faulty since $-n^2$ is not a function $N \to R^{\geq 0}$. However, the rule does apply to functions that are non-negative *except for a finite number of values of n*. For example, consider the function

$$t(n) = 12n^3 \log n - 6n^2 + \log^2 n + 36$$

In determining the order of *t(n)* we might reason as follows:

$$\begin{aligned}\Theta(t(n)) &= \Theta(\max(12n^3 \log n, -6n^2, \log^2 n, 36) \\ &= \Theta(12n^3 \log n) \\ &= \Theta(n^3 \log n)\end{aligned}$$

This reasoning is incorrect as the function $-6n^2$ is always negative. However, the following reasoning *is* correct:

$$\begin{aligned}\Theta(t(n)) &= \Theta(\max(11n^3\log n + n^3\log n - 6n^2 + \log^2 n + 36)\\ &= \Theta(\max(11n^3\log n, n^3\log n - 6n^2, \log^2 n, 36)\\ &= \Theta(11n^3\log n)\\ &= \Theta(n^3\log n)\end{aligned}$$

Even though $n^3\log n - 6n^2$ is negative for small values of n, for n sufficiently large (i.e. ≥ 4), it is always non-negative.

The limit rule

This powerful rule states that, given arbitrary functions f and $g: N \to R^{\geq 0}$,

1. if $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in R^+$ then $f(n) \in \Theta(g(n))$
2. if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ then $f(n) \in O(g(n))$ but $f(n) \notin \Theta(g(n))$
3. if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$ then $f(n) \in \Omega(g(n))$ but $f(n) \notin \Theta(g(n))$

de l'Hôpital's rule

Suppose that $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = 0$, or $\lim_{n\to\infty} f(n) = \lim_{n\to\infty} g(n) = \infty$. Suppose further that the domains of f and g can be extended to some real interval $[n_0, +\infty)$ in such a way that (a) the corresponding new functions $\hat{f}$ and $\hat{g}$ are differentiable on this interval, and also that (b) $g'(x)$, the derivative of $\hat{g}$, is never zero for $x \in [n_0, +\infty)$, then

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{x\to\infty} \frac{\hat{f}'(x)}{\hat{g}'(x)}$$

Example

Consider the two functions $f(n) = \log n$ and $g(n) = \sqrt{n}$. Since both *f(n)* and *g(n)* tend to infinity as n tend to infinity, we use de l'Hôpital's rule to compute

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{\log n}{\sqrt{n}} = \lim_{n\to\infty} \frac{1/n}{1/(2\sqrt{n})} = \lim_{n\to\infty} \frac{2}{\sqrt{n}} = 0$$

Thus we know that $f(n) \in O(g(n))$ but $f(n) \notin \Theta(g(n))$.

Conditional asymptotic notation

Many algorithms are easier to analyse if initially we restrict our attention to instances whose size satisfies a certain condition, such as being a power of 2. Consider, for example, the divide and conquer algorithm for multiplying large integers that we saw in the Introduction. Let n be the size of the integers to be multiplied. The algorithm proceeds directly if $n = 1$, which requires a microseconds for an appropriate constant a. If $n>1$, the algorithm proceeds by multiplying four pairs of integers of size $\lceil n/2 \rceil$ (or three if we use the better algorithm). Moreover, it takes a linear amount of time to carry out additional tasks. For simplicity, let us say that the additional work takes at most bn microseconds for an appropriate constant b.

[Note: In actual fact, the recursive algorithm carries out one multiplication of two $\lceil n/2 \rceil$ digit integers, one multiplication of two $\lfloor n/2 \rfloor$ digit integers, and two multiplications of a $\lceil n/2 \rceil$ digit integer and a $\lfloor n/2 \rfloor$ digit integer. For simplicity we shall assume that the algorithm always carries out four multiplications of two $\lceil n/2 \rceil$ digit integers. It turns out that the running time of this simplified method is in the same order as the running time of the more detailed method.]

The worst case time taken by this algorithm is therefore given by the function $t: N \to R^{\geq 0}$ recursively defined by

$$t(1) = a$$
$$t(n) = 4t(\lceil n/2 \rceil) + bn \quad for \ \ n > 1$$

We shall be studying techniques for solving recurrences in the next section, but unfortunately this equation cannot be handled directly by those techniques because the ceiling function $\lceil n/2 \rceil$ is rather troublesome. Nevertheless, our recurrence is easy to solve provided we consider only the case when n is a power of 2. In this case $\lceil n/2 \rceil = n/2$ and the ceiling vanishes. The techniques of the next section yield

$$t(n) = (a+b)n^2 - bn$$

provided that n is a power of 2. Since the lower-order term "$-bn$" can be neglected, it follows that $t(n)$ is in the exact order of n^2, still provided that n is a power of 2. This is denoted by $t(n) \in \Theta(n^2 \mid n \text{ is a power of } 2)$.

More generally, let $f, t: N \to R^{\geq 0}$ be two functions from the natural numbers to the nonnegative reals, and let $P: N \to \{true, false\}$ be a property of the integers. We say that $t(n)$ is in $O(f(n) \mid P(n))$ if $t(n)$ is bounded above by a positive real multiple of $f(n)$ for all sufficiently large n such that $P(n)$ holds. Formally, $O(f(n) \mid P(n))$ is defined as

$$\{t : N \to R^{\geq 0} \ : \ \exists c \in R^{+} \ \exists n_0 \in N \ \ \forall n \geq n_0 \ \ (P(n) \to t(n) \leq cf(n))\}$$

The sets $\Omega(f(n) \mid P(n))$ and $\Theta(f(n) \mid P(n))$ are defined in a similar way.

Conditional asymptotic notation is more than a mere notational convenience: its main interest is that it can generally be eliminated once it has been used to facilitate the analysis of an algorithm. For this we need a few definitions. A function $f : N \to R^{\geq 0}$ is *eventually nondecreasing* if there exists an integer threshold n_0 such that *f(n)* ≤ *f(n+1)* for all $n \geq n_0$. This implies by mathematical induction that *f(n)* ≤ *f(m)* whenever $m \geq n \geq n_0$.

Let *b* ≥ *2* be any integer. Function *f* is *b-smooth* if, in addition to being eventually nondecreasing, it satisfies the condition $f(bn) \in O(f(n))$. In other words, there must exist a constant *c* (depending on *b*) such that *f(bn)* ≤ *cf(n)* for all $n \geq n_0$. A function is *smooth* if it is *b*-smooth for every integer *b* ≥ *2*.

Most functions we are likely to encounter in the analysis of algorithms are smooth, such as $\log n$, $n \log n$, n^2, or any polynomial whose leading coefficient is positive. However, functions that grow too fast, such as $n^{\log n}$, 2^n or $n!$ are not smooth because the ratio *f(2n)/f(n)* is unbounded. For example

$$(2n)^{\log(2n)} = 2n^2 n^{\log n}$$

which shows that $(2n)^{\log(2n)} \notin O(n^{\log n})$ because $2n^2$ cannot be bounded above by a constant. Functions that are bounded above by a polynomial, on the other hand, are usually smooth provided they are eventually nondecreasing; and even if they are not eventually nondecreasing there is a good chance that they are in the exact order of some other function that is smooth. For instance, let *b(n)* denote the number of bits equal to 1 in the binary expansion of *n* – such as *b(13) = 3* because 13 is written as 1101 in binary – and consider $f(n) = b(n) + \log n$. It is easy to see that *f(n)* is not eventually nondecreasing – and therefore it is not smooth – because $b(2^k - 1) = k$ whereas $b(2^k) = 1$ for all *k*. Nevertheless, $f(n) \in \Theta(\log n)$, a smooth function.

A useful property of smoothness is that if *f* is *b*-smooth for any specific integer *b* ≥ *2,* then it is in fact smooth. To prove this, consider any two integers *a* and *b* not smaller than *2*. Assume that *f* is *b*-smooth. We must show that *f* is *a*-smooth as well. Let *c* and n_0 be constants such that *f(bn)* ≤ *cf(n)* and *f(n)* ≤ *f(n+1)* for all $n \geq n_0$. Let $i = \lceil \log_b a \rceil$. By definition of the logarithm , $a = b^{\log_b a} \leq b^{\lceil \log_b a \rceil} = b^i$. Consider any $n \geq n_0$. It is easy to show by mathematical induction from *b*-smoothness of *f* that $f(b^i n) \leq c^i f(n)$. But

$f(an) \le f(b^i n)$ because f is eventually nondecreasing and $b^i n \ge an \ge n_0$. It follows that $f(an) \le \hat{c} f(n)$ for $\hat{c} = c^i$, and thus f is a-smooth.

Smoothness rule

Smooth functions are interesting because of the **smoothness rule.**

Let $f : N \to R^{\ge 0}$ be a smooth function and let $t : N \to R^{\ge 0}$ be an eventually nondecreasing function. Consider any integer $b \ge 2$. The smoothness rule asserts that $t(n) \in \Theta(f(n))$ whenever $t(n) \in \Theta(f(n) \mid n \text{ is a power of } b)$. The rule applies equally to O and Ω notation. We shall not prove this rule. However let's look an example of the function $t(n) \in \Theta(n^2 \mid n \text{ is a power of } 2)$, which we discussed above. The smoothness rule allows us to infer directly that $t(n) \in \Theta(n^2)$, provided we verify that n^2 is a smooth function and $t(n)$ is eventually nondecreasing. The first condition is immediate since n^2 is obviously nondecreasing and $(2n)^2 = 4n^2$. The second is easily demonstrated by mathematical induction from the recurrence relation. Thus the use of conditional asymptotic notation as a stepping stone yields the final result that $t(n) \in \Theta(n^2)$ unconditionally.

Break-even points

Consider algorithms A and B with worst case running times $t_A(n) \in \Theta(n \log n)$ and $t_B(n) \in \Theta(n^2)$ respectively. By the definition of Θ we know that provided n is large enough, algorithm A will outperform algorithm B (in the worst case) on any problem instance of size n. However, for small values of n, algorithm B may in fact outperform algorithm A. The value of n where A starts to outperform B is called the break-even point of algorithm A with respect to B. We illustrate this with the following example.

Example

Suppose we have a computer that can execute a total of 100 million ($=10^8$) instructions per second. Suppose also that we have five algorithms, A_1, A_2, A_3, A_4 and A_5 which can solve an instance of size n of a certain problem using a total of $10^5 n$, $10^5 n \log_2 n$, $10^5 n^2$, $10^5 n^3$, and $10^5 2^n$ instructions, respectively.

Consider the following table:

Alg	Complexity	1 sec.	1 min.	1 hour	Before speedup	After speedup
A_1	$\Theta(n)$	1000	60,000	3,600,000	s_1	$10s_1$
A_2	$\Theta(n \log n)$	140	4893	204,094	s_2	$\approx 10s_2$
A_3	$\Theta(n^2)$	31	244	1897	s_3	$\sqrt{10}s_3 \approx 3.2s_3$
A_4	$\Theta(n^3)$	10	39	153	s_4	$\sqrt[3]{10}s_4 \approx 2.2s_4$
A_5	$\Theta(2^n)$	9	15	21	s_5	$s_5 + \log_2 10 \approx s_5 + 3.3$

In the above table, columns 3, 4 and 5 give the maximum size of a problem that can be solved in 1 second, 1 minute and 1 hour respectively. Columns 6 and 7 show the maximum sizes of problem that can be solved before and after a 10-times speedup in the computer being used to implement the algorithms. The table shows how the payoff from either increasing the time allowed to solve the problem or from using a faster computer progressively becomes worse as we move to algorithms with higher complexities.

Now let's look at the break-even points for the algorithms. With the current constant of proportionality of 10^5 for each algorithm, the break-even points are as follows:

(1) for algorithm A_1 with respect to algorithm A_2: $n=3$;
(2) for algorithm A_2 with respect to algorithm A_3: $n=2$;
(3) for algorithm A_3 with respect to algorithm A_4: $n=2$;
(4) for algorithm A_4 with respect to algorithm A_5: $n=10$.

However if the actual complexities are as follows:

A_1: $10^8 n$
A_2: $10^7 n \log_2 n$
A_3: $10^6 n^2$
A_4 : $10^5 n^3$
A_5: $10^5 2^n$

Under these conditions

A_5 is best for $2 \le n \le 9$
A_3 is best for $10 \le n \le 58$
A_2 is best for $59 \le n \le 1024$
A_1 is best for $1024 < n$