Complexity

A Hierarchy Of Standard Functions

We need to be aware of the sizes of some standard functions of a variable x, measured by their growth as x increases. (By 'growth' we mean growth in absolute value, i.e., getting further from zero.)

Formally, **polynomial** function is a functions whose output is a linear combination of powers of its input. In plainer terms, this means it is the sum of powers of x, such as x^2, or 2, or $-2.5x$, or $6x^7$, etc. The **degree** of a polynomial is the highest power of x which occurs in a non-zero term, and the term which contains it is called the **leading term**. For example, $3x + 1$ is a polynomial of degree 1 (a linear function), $5x^2 + 5x - 3$ is a polynomial of degree 2 (a quadratic function), $4x^3 - 2x^2 + 5x + 2$ is a polynomial of degree 3 (a cubic function), and 5 is a polynomial of degree 0 (a constant function). The higher the degree, the faster the polynomial grows as x grows.

The power of x in each term of a polynomial must be an integer. However, in computational complexity theory, where we're interested in how the time taken by an algorithm increases as the size of the input increases, we can also consider fractional powers such as $x^{\frac{1}{2}}$ (the square root $\sqrt{x}$), and $x^{\frac{3}{2}}$ (which is $x\sqrt{x}$).

An **exponential** function of x is some constant raised to the power of x, such as 2^x. Any exponential function will eventually overtake any polynomial function in size as x increases.

The **logarithm** of a number x with **base** b is the power to which b needs to be raised to give x, and is written $\log_b x$. In other words,

$$y = \log_b x \Leftrightarrow b^y = x.$$

For example, $\log_2 32 = 5$, since $2^5 = 32$. Since indices obey the rule $b^x b^y = b^{x+y}$, logarithms obey the rule $\log_b(uv) = \log_b u + \log_b v$. This leads to another useful rule: $\log_b(u^x) = x \log_b u$. Sometimes, when the base of the logarithm is obvious from the context (or we don't care what it is), we don't bother to write it, and just write $\log u$.

The most commonly used bases for logarithms are 2, 10, and a special number e, approximately 2.718. (The number e has the property that the gradient of the curve $y = e^x$ at the point (x, e^x) is e^x, the same value as the function itself.) Logarithms to the base e are called **natural** logarithms, and arise in mathematical work and statistics. We often write $\ln x$ for $\log_e x$. Logarithms to the base 2 are called **binary** logarithms, and arise more commonly in computer science. Logarithms to the base 10 were used to do approximate multiplications before calculations were widely available.

Scientific calculators usually allow logarithms to base e or 10 to be calculated directly. Logarithms to other based can be found by the rule

$$\log_b a = \frac{\log_e a}{\log_e b}.$$

Logarithmic functions are inverses of exponential functions, since $\log_b(b^x) = x$ and $b^{\log_b x} = x$. Just as exponential functions grow very quickly, logarithms grow very slowly. For example, $\log_2 1000000 < 20$. Any logarithmic function will be smaller than any polynomial function when the parameter x is big enough.

A summary of the order of growth of functions of the types we've discussed is:

- Constant functions such as 1 or 5 are the smallest (they don't grow as x increases).
- Logarithmic functions such as $\log_2 x$ increase without bound, but slowly.
- Roots and polynomial functions such as $x^{0.5}$, x, x^2, and x^3 increase faster than logarithmic functions. The higher the power of x, the higher the growth.
- Exponential functions such as 2^x and e^x grow extremely quickly. Any exponential function is much bigger than any polynomial function for large enough x. Factorial functions ($x! = x \times (x-1) \times (x-2) \times \ldots \times 3 \times 2 \times 1$, defined for $x \in \mathbb{N}$) are even bigger than e^x, though not as big as functions like 2^{x^2}.

Below we list some functions in increasing rate of growth.

constant	1			
logarithmic	$\log\log x$	$\log x$		
polynomial	$x^{0.1}$	$x^{0.5}$	x	$x\log x$
	$x^{1.1}$	x^2	x^3	x^{1000}
exponential	1.01^x	2^x	e^x	1000^x
	$x!$	x^x	$2^{(x^2)}$	

Other simple functions can easily be fitted into the list.

To give an example of why the distinction between polynomial and exponential functions is important, here is a table of some values of some functions.

x	x^2	$100x^2$	$100x^3$	2^x	3^x
10	100	1.0×10^4	1.0×10^5	1.0×10^3	5.9×10^4
20	400	4.0×10^4	8.0×10^5	1.0×10^6	3.5×10^9
30	900	9.0×10^4	2.7×10^6	1.1×10^9	2.1×10^{14}
40	1600	1.6×10^5	6.4×10^6	1.1×10^{12}	1.2×10^{19}
50	2500	2.5×10^5	1.3×10^7	1.1×10^{15}	7.2×10^{23}
60	3600	3.6×10^5	2.2×10^7	1.2×10^{18}	4.2×10^{28}

If these functions represent, for example, times in microseconds (millionths of a second, μs) for an algorithm to execute with input size x, then the times are:

x	x^2	$100x^2$	$100x^3$	2^x	3^x
10	<1 sec.	<1 sec.	<1 sec.	<1 sec.	<1 sec.
20	<1 sec.	<1 sec.	<1 sec.	1 sec.	58 min.
30	<1 sec.	<1 sec.	2.7 sec.	18 min.	6.5 years
40	<1 sec.	<1 sec.	6.4 sec.	13 days	3.9×10^5 years
50	<1 sec.	<1 sec.	13 sec.	36 years	2.3×10^{10} years
60	<1 sec.	<1 sec.	22 sec.	37000 years	1.3×10^{15} years

This illustrates that algorithms with exponential running time can deal only with input data which is very small.

Asymptotic ("big O") notation

Suppose that we've analysed some algorithms and found that the number of times some crucial statements were executed, with input size n, were bounded by $2(n-2)$ for one algorithm, $\frac{3}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right] - 3$ for another algorithm, and $\frac{1}{2}n^2 - \frac{1}{2}n$ for a third algorithm.

When n is big enough, these times are approximately $2n$, $1.34(1.6)^n$, and $\frac{1}{2}n^2$ respectively. So the times are approximately bounded by functions proportional to n, $(1.6)^n$, and n^2.

Some notation is needed to give a precise definition of statements like "the time is bounded by something approximately proportional to n^2 ".

Suppose that we have calculated that some algorithm we have written, when programmed for our computer, takes $67n^3+40n^2+6n+3\log_2 n+33$ microseconds to run when the input size is n. Is all this information important?

The largest term is $67n^3$. The terms $40n^2$ and $6n$ are polynomials of lower degree, $3\log_2 n$ is logarithmic and 33 is constant. So all are much smaller than $67n^3$ for large n. So the running time is approximately $67n^3$ μs for large n.

We say that $67n^3 + 40n^2 + 6n + 3\log_2 n + 33$ is of order $67n^3$, and can write

$$67n^3 + 40n^2 + 6n + 3\log_2 n + 33 \text{ is } O(67n^3).$$

How important is the number 67? If we run the program on a faster computer where each instruction takes half as long as before, the time would be approximately $33.5n^3$ μs.We want to evaluate the *algorithm* rather than the computer, so the significant fact is that the running time is approximately some multiple of n^3. So

$$67n^3 + 40n^2 + 6n + 3\log_2 n + 33 \text{ is } O(n^3).$$

Now we give a precise definition. Suppose that f and g are two real functions of a real variable n, and their values are positive for large enough n. (The restriction to positive values could be removed by taking absolute values in the definition; but we are mainly interested in functions representing running times of algorithms, which of course are always positive).

We say that

$$f(n) \text{ is } O(g(n)) \text{ as } n \to \infty$$

if $f(n)$ is less than a constant times $g(n)$ whenever n is big enough; more precisely, there exist constants c and n_0 such that

$$f(n) < cg(n) \text{ for all } n > n_0.$$

In words, we could say that $f(n)$ is asymptotically dominated by $g(n)$. Informally, $f(n)$ is $O(g(n))$ if $f(n)$ doesn't grow at a faster rate than $g(n)$ as $n \to \infty$ (though $f(n)$ may grow more slowly than $g(n)$). In predicate notation:

$$\exists c, n_0 \bullet \forall n > n_0 \bullet f(n) < cg(n).$$

The notation "$f(n)$ is $O(g(n))$" is normally used when g is a "simple" function such as $\log n$, n^2, e^n, etc. By "$f(n)$ is $O(1)$" we mean that $f(n)$ is bounded by a constant.

Some examples are:

1. $3x^2 - 2x + 4$ is $O(n^2)$ since $3x^2 - 2x + 4 < 3x^2$ whenever $n > 2$.
2. $67n^3 + 40n^2 + 6n + 3\log_2 n + 33$ is $O(n^3)$ since the function on the left is $< 70n^3$ for all $n > 13.561$.
3. $100n + 2$ is $O(n)$ since $100n + 2 < 101n$ for all $n > 1000$.
4. $f(n)$ is $O(n^k)$ whenever $f(n)$ is a polynomial of degree k.
5. if $f(n)$ is an exponential function, such as 2^n, then there is no number k such that $f(n)$ is $O(n^k)$.

Suppose that we have some convenient way of specifying the size of the objects on which some algorithm operators as a number. For example, a pair of $n \times n$ matrices to be multiplied could have n as a measure of their size.

The bounds given above are not necessarily the best possible. This doesn't matter, as all we're interested in is the behaviour of the functions as $n \to \infty$.

Suppose that we have some convenient way of specifying the size of the objects on which some algorithm operates as a number. For example, a pair of $n \times n$ matrices to be multiplied could have n as a measure of their size. When the algorithm is implemented, if the longest time taken for input size n is a function which is $O(g(n))$ for some (simple) function g, we say that the algorithm has complexity $O(g(n))$.

As we have mentioned, one reason for using the "O" notation in computational complexity is that it can show in simple terms how the running time grows with input size. Another reason is that it allows us to do simpler calculations in finding the running time. As long as we find an upper bound for the number of times the core parts of the algorithm are used, we can find its complexity in terms of "O". Detailed consideration of every instruction in an implementation would be very tedious.

Strictly speaking, to say that an algorithm has complexity $O(g(n))$ includes the possibility that it is $O(h(n))$ for some function h which is essentially smaller than g. Usually, however, when it is stated that an algorithm is $O(g(n))$ for some simple function g of n, the writer implies that this is the "smallest" simple function for which this is true, or, in other words, that the estimate given of the complexity is the best possible. If an algorithm has a worst case running time $6n^2$, for example, for input size n, then its complexity is $O(n^2)$.

But O gives just an upper bound, so it is also $O(n^3)$, $O(n^{100})$ and $O(2^n)$, etc. It would be more precise to use the "Big Theta" notation, and say that $6n^2$ is $\Theta(n^2)$. This means that it is $O(n^2)$ and we can't give a better bound.

The definition is that $f(n)$ is $\Theta(g(n))$ if there exist constants c_1, c_2 and n_0 such that

$$c_1 g(n) < f(n) < c_2 g(n) \text{ for all } n > n_0.$$

Another way to say this is that

$$f(n) \text{ is } \Theta(g(n)) \text{ if} f(n) \text{ is } O(g(n)) \text{ and } g(n) \text{ is } O(f(n)).$$

A good cheat sheet is available if you ever get confused about Big O notation.

Introduction to computational complexity

Computational complexity, or computational efficiency, is the study of the amount of computational resources needed to carry out computations. The complexity of a problem or algorithm is usually measured by the time it takes or the amount of memory needed. We shall concentrate on time, and think in terms of a machine carrying out the algorithm.

An early indication of the importance of this is the following quote:

> As soon as the Analytical Engine exists, it will necessarily guide the future course of the science. Whenever any result is sought by its aid, the question will then arise—By what course of calculation can these results be arrived at by the machine in the shortest time?
> *Charles Babbage, 1864*

Some of the computations we might like to carry out are:

- Given a list of numbers or other data, produce a list of the same data in increasing order. (Sorting).
- Given a database of employee information, output a set of pay slips.
- Given a program and some input, decide whether it will halt on that input. (The Halting Problem).
- Given a number, decide whether it is prime.
- Factorise a number.
- Multiply two matrices.
- Given a set of cities and a table of distances between them, find the shortest route which visits all cities and finishes at its starting point. (The Travelling Salesperson Problem).
- Given a telephone directory and a name, decide whether that name is in the directory.
- Given a telephone directory and a telephone number, decide whether that number is in the directory.
- Given a boolean expression, decide whether it is a tautology (i.e., true for all possible assignments of truth values to the variables).

Computational problems are functions; they take some input and produce some output. If the possible outputs are just True and False (or Yes and No) the problem is called a **decision problem**.

Some questions we can ask and tasks we can set about a problem are:

1. Is the function computable, or the decision problem decidable?
2. If yes, specify an algorithm.
3. Is our algorithm efficient?
4. Is there a more efficient algorithm?

If the answer to question 1 is "no", then we can go no further.

Finding an algorithm (task 2) may be easy or hard; in the cases above it is easy (except of course for the Halting Problem, when it is impossible).

Computational complexity theory is concerned with questions 3 and 4. We will discuss how efficiency is measured. The "obvious" algorithms for some of the problems above are very inefficient.

If we know we have an inefficient algorithm, question 4 is of great interest. In some cases, e.g., sorting numbers, an answer can be given without too much difficulty. In other cases (e.g., the Travelling Salesperson Problem), deciding whether a better algorithm is possible is one of the major unsolved problems of computer science.

We are interested particularly in how the computational effort grows with the size of the data. For example, searching for a name in a list of ten names is fast; but when searching for a name in a list of a million names it is worth looking for an efficient algorithm.

Of particular interest is the distinction between "easy" problems, which can be solved in a time proportional to a polynomial function of the size n of the input, and "hard" problems, many of which require exponential time — at least $c^{(n^k)}$ for some constants $c > 1$ and $k > 0$.

For many problems no polynomial algorithms are known (e.g., the Travelling Salesperson Problem), but it has not been proved that they do not exist.

For theoretical work, the size of the input to an algorithm is measured by the number of bits needed to specify it, and the time taken is measured by the number of instructions executed by a fixed (theoretical) computer such as a Turing Machine. For practical calculations involving complexity, it is often convenient to use some more natural measure. For example, in searching a list of n names it is natural to use n as a measure of size. The time taken is often measured by the number of times significant operations are carriedout, for example, multiplications and additions, or comparisons between data items. This kind of simplification does not affect the distinction between polynomial andexponential algorithms.

The time taken by an algorithm, or the number of operations, will usually depend on the particular input data as well as its size. In analysing an algorithm, we can find the average complexity, or the complexity in the worst case (i.e., the maximum time). It is usually more complicated to find the average complexity than the worst case complexity. The worst case complexity is also useful in giving a guarantee that a program will not take more than a specified time to run.

The P/NP problem

In this section we are concerned with the distinction between problems which can be solved in a time which is no more than some polynomial function of the input size and those which cannot. The latter are said to be **intractable**.

The class P of decision problems consists of those which can be solved by an algorithm running in polynomial time – 'easy' problems.

Reading is usually easier than writing, whether one is concerned with a novel or a mathematical proof. Similarly, verifying that the answer to a decision problem is *yes* when someone has solved it is often easier that solving the problem. Here is an example.

To decide whether the number 34905581 is composite (i.e., can be written as a product of smaller numbers) takes some effort; on the other hand, a possible proof that it is composite is the equation

$$34905581 = 4567 \times 7643,$$

and it is easy to check that this is correct.

One way to look at this is to say that a non-deterministic algorithm could factorise 34905581 by simultaneously trying all integers up to $\sqrt{34905581}$ to see whether they divide 34905581 (or,equivalently, guessing the right one). Alternatively, the factors 4567 and 7643 (or just one of them) constitute a certificate which proves that the number is composite.

The class NP is defined as consisting of problems that can be solved in polynomial time by a non-deterministic algorithm. An alternative definition, probably easier to understand, is that the class NP consists of problems for which there exist certificates which can be verified by an algorithm running in polynomial time – problems for which it is 'easy' to check a claimed answer. To understand the concept it is useful to bear both ideas in mind.

Many interesting and important problems belong to the class NP, but are not known to belong to class P. One example is the Travelling Salesperson Problem. (It can be made into a decision problem by asking whether there is a tour of length less than some specified number.)

A remarkable feature of the class NP is that it contains many problems, known as **NP-complete** problems, which are the hardest problems in NP, in the sense that they have the following property: if a polynomial algorithm exists for any one NP-complete problem, then polynomial algorithms exist for all problems in NP.

It is believed that no easy (i.e., polynomial) algorithms exist for any of them, and so NP is a strictly bigger class of problems than P, but no-one has been able to prove this.

Whether or not $P = NP$ is one of the major unanswered questions of computer science.