CSE530: Algorithms & Complexity
Notes on Lecture 2: Asymptotic Notations

Antoine Vigneron

February 26, 2018

1 Motivation

The analysis of Insertion Sort shows that, in the worst case, it sorts an array of n numbers in time

$$T_1(n) = a_1 n^2 + b_1 n + c_1,$$

where a_1, b_1 and c_1 are unknown constants. These constants depend on the hardware, the compiler, and implementation details. Two other sorting algorithms, Bubble Sort and Merge Sort, have worst case running times

$$T_2(n) = a_2 n^2 + b_2 n + c_2$$
$$T_3(n) = a_3 n \log n + b_3 n$$

where a_2, b_2, c_2, a_3, b_3 are unknown constants. The analysis also reveals that a_1, a_2 and a_3 are positive.

We would like to be able to make meaningful comparisons between the running times of theses algorithms in spite of the fact that these constants are unknown. The *asymptotic notations* (also known as Landau's notations) will allow us to do it.

In order to see how these functions compare, let us choose the constants arbitrarily. For instance, we set $a_1 = 4$, $b_1 = -3$, $c_1 = 6$, $a_2 = 3$, $b_2 = 6$, $c_2 = -2$, $a_3 = 8$, $b_3 = 7$. The graphs at two different resolutions can be found in Figure 1.

We observe that, for large values of n, the functions $T_1(n)$ and $T_2(n)$ grow at roughly the same rate. It is not very surprising: $T_1(n)$ is roughly equal to $4n^2$ for large values of n, and $T_2(n)$ is roughly $3n^2$, so $T_1(n)$ is roughly $4T_2(n)/3$. We will introduce in this lecture the notation $\Theta(\cdot)$ that expresses this relation: $T_1(n) = \Theta(T_2(n))$ means that T_1 and T_2 are within a constant factor from each other for large values of n. We can also write $T_1(n) = \Theta(n^2)$ and $T_2(n) = \Theta(n^2)$. The advantage of this notation is that it allows us to discard the unknown constant factors and expresses what we really know: $T_1(n)$ and $T_2(n)$ grow like n^2. We will also say that $T_1(n)$ and $T_2(n)$ are *quadratic*.

We also observe that $T_3(n)$ is much smaller than $T_1(n)$ and $T_2(n)$ for large values of n. This is not surprising either, as $\lim_{n\to\infty} T_3(n)/T_1(n) = 0$. We will use the notation $o(\cdot)$ to express this relation, and thus we will write $T_3(n) = o(T_1(n))$, or even just $T_3(n) = o(n^2)$. This shows that Merge Sort is a much better algorithm than the other two, at least for sorting large inputs. Again, the point of this notation is that it allows us to ignore the unknown constant factors that appear in the running times of our algorithms.

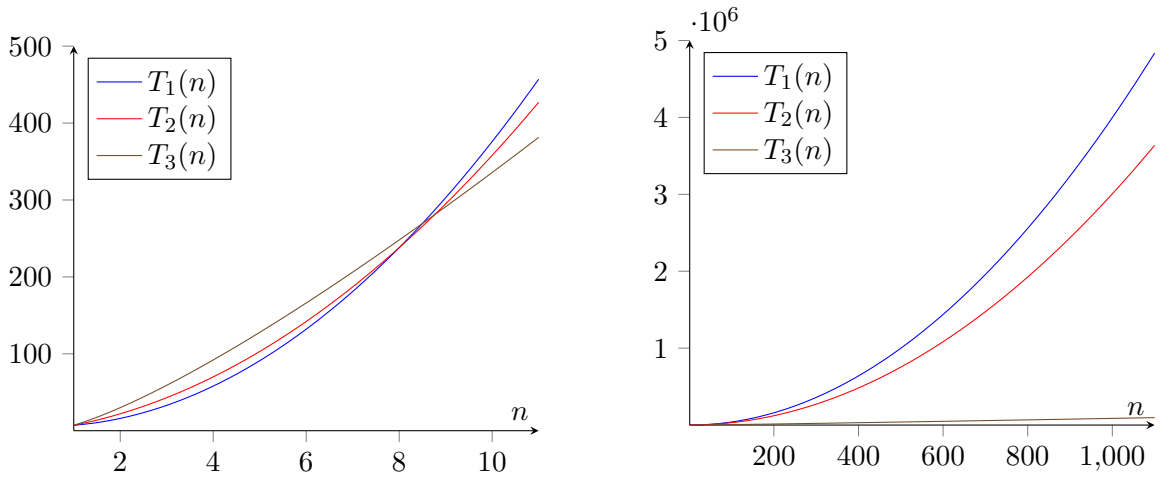

Figure 1: Graphs of the functions $T_1(n) = 4n^2-3n+6$, $T_2(n) = 3n^2+6n-2$ and $T_3(n) = 8n\log n+7n$. The asymptotic relations $o(\cdot)$ and $O(\cdot)$ are illustrated by the graph on the right side, with large values of n. In particular, we have $T_3(n) = o(T_1(n))$, $T_3(n) = o(T_2(n))$, $T_1(n) = O(T_2(n))$, $T_2(n) = O(T_1(n))$, $T_1(n) = \Theta(T_2(n))$ and $T_2(n) = \Theta(T_1(n))$.

2 Little-o notation

In this Lecture, we consider real-valued functions of an integer n, often denoted $f(n)$, $g(n)$. They can also be seen as sequences of real numbers.

Definition 1. *Let f and g be two real-valued functions of integer variables. We write $f(n) = o(g(n))$ if and only if there exists a function ρ such that $f(n) = \rho(n)g(n)$ for all n, and $\lim_{n\to\infty} \rho(n) = 0$.*

This definition is not equivalent to saying that $\lim_{n\to\infty} f(n)/g(n) = 0$, because it would implicitly require that $g(n)$ is nonzero. But if $g(n)$ is nonzero, or if it is nonzero for large enough n, we have the following more convenient formulation.

Proposition 1. *Suppose that there exists an integer n_0 such that $g(n) \neq 0$ for all $n \geqslant n_0$. Then*

$$f(n) = o(g(n)) \text{ if and only if } \lim_{n\to\infty} \frac{f(n)}{g(n)} = 0.$$

Proof. Suppose that there exists an integer n_0 such that $g(n) \neq 0$ for all $n \geqslant n_0$. Then for all $n \geqslant n_0$, we have $\rho(n) = f(n)/g(n)$. So $\rho(n)$ and $f(n)/g(n)$ either have the same limit at infinity, or do not converge. We conclude that $f(n) = o(g(n)) \Leftrightarrow \lim_{n\to\infty} \rho(n) = 0 \Leftrightarrow \lim_{n\to\infty} f(n)/g(n) = 0$ □

So intuitively, $f(n) = o(g(n))$ means that $f(n)$ is much smaller than $g(n)$ for large values of n.

Using Proposition 1 and your knowledge of calculus, you can easily derive the following examples.

$$\begin{aligned} 1/n + 1/n^2 &= o(1) \\ 2n + 5 &= o(n^2) \\ 10n \log n + 7n + 5 &= o(n^2) \\ n^2 + 2n + 1 &= o(n^3) \\ n^{10} + 5n^3 &= o(2^n) \\ 2^n + 3^n + 4^n &= o(n!) \end{aligned}$$

The definition of $o(\cdot)$ can also be expressed as follows, using the definition of a limit:

Definition 2. *We say that $f(n) = o(g(n))$ if, for every real number $\varepsilon > 0$, there exists an integer n_0 such that $n \geqslant n_0$ implies $|f(n)| \leqslant \varepsilon |g(n)|$.*

More intuitively, it means that for every fixed $\varepsilon > 0$, the absolute value of $f(n)$ is less than the absolute value of $g(n)$ for *large enough* values of n. The real number ε should be understood as being very small, for instance this should hold for $\varepsilon = 10^{-9}$. This definition can be written in a more compact way using quantifiers: $f(n) = o(g(n))$ means that

$$\forall \varepsilon > 0 \;\; \exists n_0 : \;\; n \geqslant n_0 \Rightarrow |f(n)| \leqslant \varepsilon |g(n)|$$

Example. We want to prove that $2n + 5 = o(n^2)$ using this definition. So given $\varepsilon > 0$, we want to find an integer n_0 such that $2n + 5 \leqslant \varepsilon n^2$ for all $n \geqslant n_0$. Assuming that $n \geqslant 1$, it suffices to have $7n \leqslant \varepsilon n^2$, and thus $\varepsilon \geqslant 7/n$. So we just choose $n_0 = \lceil 7/\varepsilon \rceil$, and then for all $n \geqslant n_0$, we have $2n + 5 \leqslant \varepsilon n^2$.

The properties below can be derived from the definition. When the functions are nonzero, they follow immediately from Proposition 1. We first consider the relation between common mathematical functions.

Proposition 2. *For all real numbers α, β, γ*

(a) $\log n = o(n^\alpha)$ *whenever* $\alpha > 0$.

(b) $n^\alpha = o(n^\beta)$ *whenever* $\alpha < \beta$.

(c) $n^\beta = o(\gamma^n)$ *whenever* $\gamma > 1$.

(d) $\gamma^n = o(n!)$

The relation $f(n) = o(g(n))$ is sometimes denoted $f(n) \prec g(n)$. (Notation of *Hardy*.) So Proposition 2 can be rewritten in a more compact form:

$$\log n \prec n^\alpha \prec n^\beta \prec \gamma^n \prec n! \qquad \text{whenever } 0 < \alpha < \beta \text{ and } \gamma > 1$$

Figure 2 gives special cases of these relations. This notation has the advantage to indicate that this relation is *transitive*, so for instance (a) and (c) show that $\log n \prec \gamma^n$, in other words $\log n = o(\gamma^n)$ for all $\gamma > 1$. Transitivity is stated below in Proposition 3 (iv).

Proposition 3. *For every functions f, g and h, and for every constant $\lambda > 0$,*

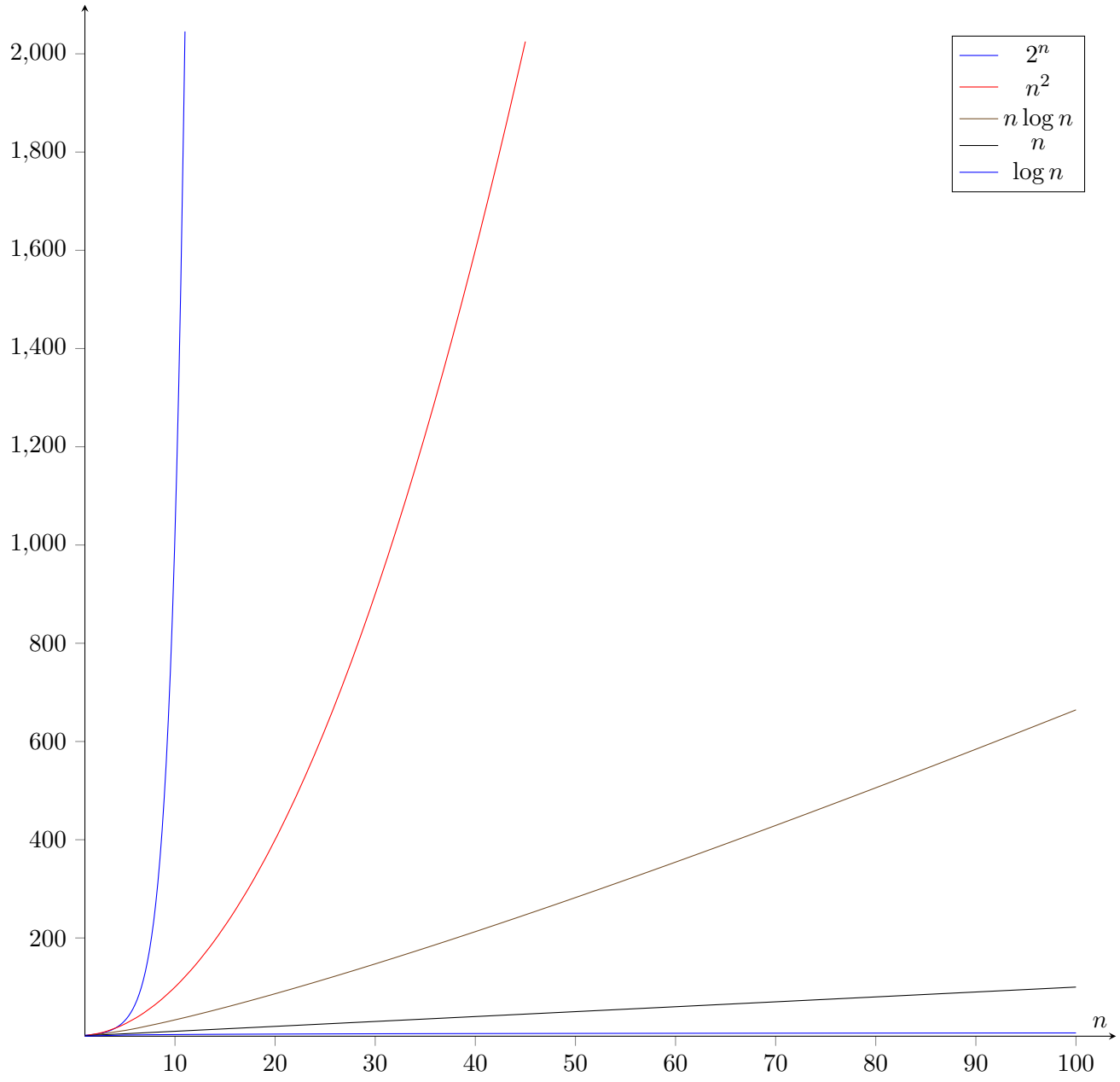

Figure 2: Graphs of common functions: $\log n \prec n \prec n \log n \prec n^2 \prec 2^n$. It illustrates the difference between several classes of running times: $\log n$ is logarithmic, n is linear, n^2 is quadratic, and 2^n is exponential.

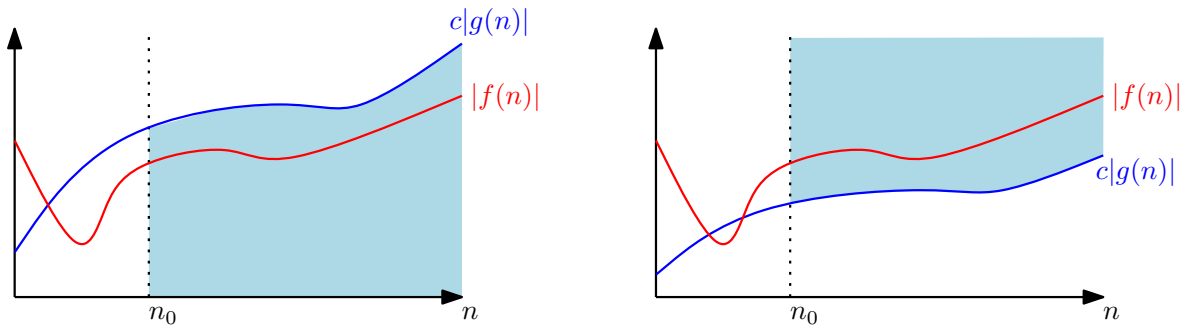

Figure 3: (Left) Illustration of the relation $f(n) = O(g(n))$. For values of n larger than n_0, the graph of $|f(n)|$ lies below $c|g(n)|$. (Right) Illustration of $f(n) = \Omega(g(n))$.

(i) $f(n) = o(h(n))$ *implies* $\lambda f(n) = o(h(n))$.

(ii) $f(n) = o(h(n))$ *and* $g(n) = o(h(n))$ *implies* $f(n) + g(n) = o(h(n))$.

(iii) $f(n) = o(h(n))$ *implies* $f(n)g(n) = o(g(n)h(n))$.

(iv) $f(n) = o(g(n))$ *and* $g(n) = o(h(n))$ *implies* $f(n) = o(h(n))$.

Examples. (a) implies that $\log^2 n = o(n)$, so by (iii) we have $\sqrt{n}\log^2 n = o(n\log^2 n)$. By (c) and (d), we have $n^3 = o(2^n)$ and $2^n = o(n!)$, so (iv) implies $n^3 = o(n!)$. Similarly, applying (b), (i) and (ii), we get $5n^2 + 2n + 4 = o(n^3)$.

As a final remark, it follows immediately from the definition that $f(n) = o(1)$ means that $\lim_{n\to\infty} f(n) = 0$. For example, we can write $1/n = o(1)$, or $(\sin n)/n = o(1)$.

3 Big-O notation

The $o(\cdot)$ notation presented in the previous section allows to compare two functions such as $T_1(n) = 4n^2 - 3n + 6$ and $T_3(n) = 8n\log n + 7n$ by writing $T_3(n) = o(T_1(n))$. However, it does not allow to compare $T_1(n)$ with $T_2(n) = 3n^2 + 6n - 2$, because $\lim_{n\to\infty} T_1(n)/T_2(n) = 4/3$, so the statements $T_1(n) = o(T_2(n))$ and $T_2(n) = o(T_1(n))$ do not hold. Intuitively, we would like to be able to say that $T_1(n)$ and $T_2(n)$ are, asymptotically, within a constant factor from each other. This can be expressed using the big-O notation $O(\cdot)$, defined as follows.

Definition 3. *We write* $f(n) = O(g(n))$ *if there exist two constants* $c > 0$ *and* $n_0 \in \mathbb{N}$ *such that* $n \geqslant n_0$ *implies* $|f(n)| \leqslant c|g(n)|$.

In other words, $f(n)$ is within a constant factor from $g(n)$ for large enough n. (See Figure 3, left.) It can be rewritten using quantifiers as follows: We say that $f(n) = O(g(n))$ if and only if

$$\exists c > 0\ \exists n_0 \in \mathbb{N}\ :\ n \geqslant n_0 \Rightarrow |f(n)| \leqslant c|g(n)|.$$

Compared with the definition of $o(\cdot)$, the difference is that we do not need the upper bound to hold for arbitrarily small $\varepsilon > 0$, it only needs to hold for some $c > 0$. Therefore, $f(n) = o(g(n))$ implies $f(n) = O(g(n))$. The converse is not true, however. For instance, we have $2n = O(n)$, but the statement $2n = o(n)$ is wrong.

It also follows directly from the definition that $f(n) = O(f(n))$ for all function f. So the relation $f(n) = O(g(n))$ is sometimes denoted $f(n) \preccurlyeq g(n)$. As we shall see below, this relation is also transitive. So $o(\cdot)$ can be thought as being analogous to $<$, and $O(\cdot)$ is analogous to $\leqslant$.

Here is an example of how to prove that $T_1(n) = O(T_2(n))$ using Definition 3, when $T_1(n) = 4n^2 - 3n + 6$ and $T_2(n) = 3n^2 + 6n - 2$. First observe that for all $n \geqslant 1$, we have

$$\begin{aligned} T_1(n) &= 4n^2 - 3n + 6 \\ &\leqslant 4n^2 + 0 + 6n^2 \\ &= 10n^2. \end{aligned}$$

On the other hand, for all $n \geqslant 1$, we have $3n^2 \leqslant T_2(n)$, and thus $T_1(n) \leqslant (3/10)T_2(n)$. So we can just apply the definition with $n_0 = 1$ and $c = 3/10$.

In practice, this type of proof can often be done by applying the properties below. Properties (i) and (ii) have been mentioned earlier.

Proposition 4. *For all functions f, g, h, φ and for all constant $\lambda > 0$,*

(i) $f(n) = O(f(n))$

(ii) $f(n) = o(g(n))$ *implies* $f(n) = O(g(n))$.

(iii) $f(n) = O(h(n))$ *implies* $\lambda f(n) = O(h(n))$.

(iv) $f(n) = O(h(n))$ *and* $g(n) = O(h(n))$ *implies* $f(n) + g(n) = O(h(n))$.

(v) $f(n) = O(h(n))$ *and* $g(n) = O(\varphi(n))$ *implies* $f(n)g(n) = O(h(n)\varphi(n))$.

(vi) $f(n) = o(g(n))$ *and* $g(n) = O(h(n))$ *implies* $f(n) = o(h(n))$.

(vii) $f(n) = O(g(n))$ *and* $g(n) = o(h(n))$ *implies* $f(n) = o(h(n))$.

(viii) $f(n) = O(g(n))$ *and* $g(n) = O(h(n))$ *implies* $f(n) = O(h(n))$.

Properties (vi)–(viii) are more intuitive if we use Hardy's notation. For instance, Property (vi) can be reformulated as follows:

$$f(n) \prec g(n) \text{ and } g(n) \preccurlyeq h(n) \text{ imply } f(n) \prec h(n).$$

We now prove Property (iv): If $f(n) = O(h(n))$ and $g(n) = O(h(n))$, then $f(n)+g(n) = O(h(n))$. As $f(n) = O(h(n))$, there exist constants n_1, c_1 such that $n \geqslant n_1$ implies that $|f(n)| \leqslant c_1|h(n)|$. Similarly, there exist constants n_2, c_2 such that $n \geqslant n_2$ implies $|g(n)| \leqslant c_2|h(n)|$. So let $c = c_1 + c_2$ and $n_0 = \max(n_1, n_2)$. For all $n \geqslant n_0$, we have $n \geqslant n_1$ and $n \geqslant n_2$, and therefore

$$\begin{aligned} f(n) + g(n) &\leqslant c_1|h(n)| + c_2|h(n)| \\ &= (c_1 + c_2)|h(n)| \\ &= c|h(n)|. \end{aligned}$$

The proofs of the other properties are similar, and are left as exercises.

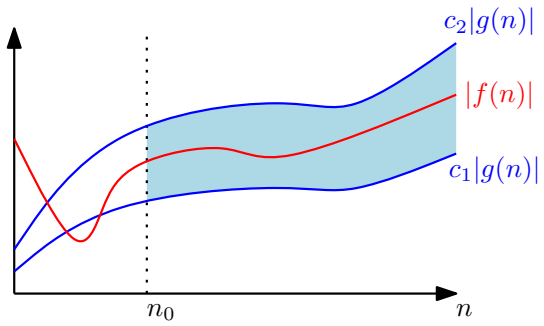

Figure 4: Illustration of the relation $f(n) = \Theta(g(n))$.

Applications. Consider the function $f(n) = 5n^3 - 2n^2 + 5$. As we have seen in the previous section, we have $-2n^2 + 5 = o(n^3)$. So by (ii), we also have $-2n^2 + 5 = O(n^3)$. By (i), we know that $n^3 = O(n^3)$, so (iii) implies that $5n^3 = O(n^3)$. Then it follows from (iv) that $f(n) = O(n^3)$. More generally, we can prove the following:

Proposition 5. *We say that $f(n)$ is a degree-d polynomial in n if there are constants $a_0, \dots, a_d$ such that $a_d \neq 0$, and $f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0$ for all n. If $f(n)$ satisfies these conditions, then $f(n) = O(n^d)$.*

As a final remark, $f(n) = O(1)$ simply means that $f(n)$ is *bounded*. In other words, there exists a constant $C > 0$ such that $|f(n)| \leqslant C$ for all $n \in \mathbb{N}$. In computer science, we rather say that $f(n)$ is *constant* when $f(n) = O(1)$.

4 Big-Ω notation

The Big-Ω notation is defined as follows. (See Figure 3, right.)

Definition 4. *We write $f(n) = \Omega(g(n))$ if there exist two constants $c > 0$ and $n_0 \in \mathbb{N}$ such that $n \geqslant n_0$ implies $|f(n)| \geqslant c|g(n)|$. In other words, $f(n) = \Omega(g(n))$ means that $g(n) = O(f(n))$.*

The properties of $\Omega(\cdot)$ can be derived by symmetry from the properties of $O(\cdot)$, so we will not describe them here. This notation will be mostly used to give lower bounds on the running times of algorithms. For instance, it can be proved that any sorting algorithm takes $\Omega(n \log n)$ time. It means that the running time of any sorting algorithm is at least $cn \log n$ for some constant $c > 0$, and for large enough n.

5 Big-Θ notation

There are functions $f(n)$ and $g(n)$ such that $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$. For instance, if $f(n)$ and $g(n)$ are polynomials with the same degree d, then these two relations hold. In this case, we will write $f(n) = \Theta(g(n))$. This situation is depicted in Figure 4. It means that $f(n)$ and $g(n)$ are within a constant factor from each other for large enough n.

Definition 5. *We write $f(n) = \Theta(g(n))$ if there exist three constants $c_1 > 0$, $c_2 > 0$ and $n_0 \in \mathbb{N}$ such that $n \geqslant n_0$ implies $c_1|g(n)| \leqslant |f(n)| \leqslant c_2|g(n)|$. In other words, we say that $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.*

The proof of the proposition below follows directly from this definition.

Proposition 6. *For all functions $f(n)$, $g(n)$, and $h(n)$,*

- $f(n) = \Theta(f(n))$. *(Reflexivity)*
- $f(n) = \Theta(g(n))$ *iff* $g(n) = \Theta(f(n))$. *(Symmetry)*
- $f(n) = \Theta(g(n))$ *and* $g(n) = \Theta(h(n))$ *implies* $f(n) = \Theta(h(n))$. *(Transitivity)*

In other words, the relation $\Theta(\cdot)$ is an equivalence relation*:*

The properties below will help proving Θ-relations without going back to the definition.

Proposition 7. *(i) Let $\lambda \neq 0$ be a constant. Then $\lambda f(n) = \Theta(f(n))$.*

(ii) If $f_1(n) = \Theta(g_1(n))$ and $f_2(n) = \Theta(g_2(n))$, then $f_1(n)f_2(n) = \Theta(g_1(n)g_2(n))$.

(iii) If $f(n) = o(g(n))$, then $f(n) + g(n) = \Theta(g(n))$.

(iv) If there exists n_0 such that $0 \leqslant f(n) \leqslant g(n)$ for all $n \geqslant n_0$, then $f(n) + g(n) = \Theta(g(n))$.

(v) If $f(n)$ is a degree-d polynomial, then $f(n) = \Theta(n^d)$.

Proof. We only prove (iii) and (v), the other proofs are simpler and are left as exercises. We now prove (iii). As $f(n) = o(g(n))$, there exists n_0 such that $n \geqslant n_0$ implies that $|f(n)| \leqslant |g(n)/2|$. It follows that $|g(n)| - |g(n)/2| \leqslant |f(n) + g(n)| \leqslant |g(n)| + |g(n)/2|$ for all $n \geqslant n_0$. We have just proved that $|g(n)|/2 \leqslant |f(n) + g(n)| \leqslant 3|g(n)|/2$ for all $n \geqslant n_0$, and thus $f(n) + g(n) = \Theta(g(n))$.

Proof of (v). We can write $f(n) = a_d n^d + a_{d-1} n^d + \cdots + a_1 n + a_0$ where $a_d > 0$. We define $g(n) = a_{d-1} n^d + \cdots + a_1 n + a_0$. Then $\lim_{n\to\infty} g(n)/a_d n^d = 0$, so $g(n) = o(a_d n^d)$. As $f(n) = a_d n^d + g(n)$, it follows from Property (iii) that $f(n) = \Theta(a_d n^d) = \Theta(n^d)$. □

6 Little-ω notation

Definition 6. *We write $f(n) = \omega(g(n))$ if and only if $g(n) = o(f(n))$.*

So if there exists n_0 such that $f(n) \neq 0$ for all $n \geqslant n_0$, then

$$f(n) = \omega(g(n)) \text{ iff } \lim_{n\to\infty} \frac{g(n)}{f(n)} = 0.$$

In other words, $f(n) = \omega(g(n))$ means that $f(n)$ grows much faster than $g(n)$.

7 Further Comments

We say that the $O(\cdot)$ notation gives an *asymptotic upper bound*, the $\Omega(\cdot)$ notation gives an *asymptotic lower bound*, and $\Theta(\cdot)$ gives an *asymptotically tight bound*. In algorithms analysis, it is preferable to obtain tight bounds, but sometimes no good lower bound is known and we just give an upper bound.

7.1 Absolute values

The definitions of the asymptotic notations use absolute values: we have bounds on $|f(n)|$ and $|g(n)|$ instead of just $f(n)$ and $g(n)$. In an algorithms course, it does not make much difference because the functions we consider are usually running times, and hence they are obviously positive. The definition with absolute values was given here as it is the standard definition in mathematics. It also applies to functions that change sign: for instance, $\sin n = o(n \sin n)$ or $(2n^2+n)\cos n = \Theta(n^2 \cos n)$. We will not study this issue further as it is not very important for this course.

7.2 Getting rid of constants and lower-order terms

Suppose that $T(n) = 4n^2 + 5n + 1$. Then we can write that $T(n) = O(4n^2)$, but this is not a good idea. The reason is that $O(4n^2)$ means exactly the same as $O(n^2)$. Similarly, we could write $T(n) = O(n^2 + n + 1)$, but again $n + 1$ gives not information as $O(n^2)$ and $O(n^2 + n + 1)$ mean exactly the same.

More generally, when we give a bound using one of the four asymptotic notations above, it is advisable to remove constant factors, and also remove lower order-terms, as they give no information, and could possibly lead to mistakes.

7.3 Asymptotic Notation in Equations

We will often use asymptotic notations in equations. For instance, we will write that the running time $T(n)$ of Merge Sort satisfies the relation

$$T(n) = 2T(n/2) + \Theta(n).$$

It means that there is a function f such that $f(n) = \Theta(n)$ and $T(n) = 2T(n/2) + f(n)$.

We may also write that a function S satisfies the relation

$$S(n) = \sum_{i=1}^{n} O(i).$$

It means that there is a function $g(n) = O(n)$ such that $S(n) = \sum_{i=1}^{n} g(i)$. Then we will be able to prove that $S(n) = O(n^2)$.

7.4 Algorithms analysis

The running time of algorithms is often expressed using the asymptotic notations. For instance, we establish the following results in CSE331.

Algorithm	Worst-case running time on input of size n
Binary Search	$\Theta(\log n)$
Linear Search	$\Theta(n)$
Merge Sort	$\Theta(n \log n)$
Insertion Sort	$\Theta(n^2)$

Here we use the notation $\Theta(\cdot)$ because we can prove upper bounds and matching lower bounds for all these algorithms. For instance, we know that on some input, Insertion Sort takes $c_1 n^2$ time

for some constant $c_1 > 0$, and we also know that there is a constant c_2 such that it takes time at most $c_2 n^2$ on every input.

Often these running times are given using the $O(\cdot)$ notation. In the case above, it is less precise as it does not indicate the lower bound. It could also mean that no tight bound is known, so only the best known upper bound is given. In any case, one advantage of the asymptotic notations is that they allow us to throw away unknown constants and lower order terms.

7.5 Classes of running times

The statement $f(n) = O(1)$ means that there is a constant c such that $|f(n)| \leqslant c$ for large enough n. It means that $f(n)$ is bounded by a constant for all positive integers n. So we will say that $f(n)$ is *constant*. More generally, worst-case running times $T(n)$ will be described as follows:

- If $T(n) = O(1)$, then we say that the running time is *constant*.
- If $T(n) = O(\log n)$, we say that it is *logarithmic*.
- If $T(n) = O(n)$, we say that it is *linear*.
- If $T(n) = O(n^2)$, we say that it is *quadratic*.
- If $T(n) = O(n^k)$ for some integer constant k, we say that it is *polynomial*. For instance, a running time $O(n^4 + 3n^2 + 1)$ is polynomial.
- If $T(n) = O(2^n)$, we say that it is *exponential*. More generally, it suffices that $T(n) = O\left(2^{n^k}\right)$ for some integer constant k.

These different classes are illustrated in Figure 2. An exponential running time implies that only small instances can be solved in practice, while a polynomial running time means that large instances can be solved. (This is not entirely true: an n^{100} running time is polynomial, but still only trivial instances can be solved.) So an important issue when studying a computation problem is to determine whether it can be solved in polynomial time, or whether it requires exponential time.

In algorithms design, it is often desirable to find a polynomial-time algorithm for the problem being considered. On the other hand, exponential-time algorithms are often considered too slow. For instance, if the worst case running time is $\Omega(2^n)$, we may only be able to solve very small instances of the problem. Unfortunately, for many problems, the best known algorithms are exponential. In particular, it is true for **NP**-hard problems. (Will be described later this semester.)