Asymptotic Growth Rates and the “Big-O” Notation

In the first lecture of this thread we defined the worst-case running time of an algorithm, and we saw how to determine this for an algorithm by analysing its (pseudo) code. We discussed the fact that if we want to abstract away from factors such as the programming language or machine used (which might change every couple of years), then we can at best expect to be able to determine the running time up to a constant factor.

In this lecture we introduce a notation which allows us to determine running time without keeping track of constant factors[1]. This is called *asymptotic notation*, and it captures how the running time of the algorithm grows with the size of the input. Asymptotic notation is a basic mathematical tool for working with functions that we only know up to a constant factor. There are three types of asymptotic notation that are commonly used in algorithmic analysis, $O(\cdot)$ (“Big-O”), $\Omega(\cdot)$ (“Omega”) and $\Theta(\cdot)$ (“Theta”); the dot in the parentheses indicates that we would normally have something in that position (a function).

This note is organised as follows. We begin by introducing the most commonly used asymptotic symbol “big-O” in §2.1, in the context of functions in general rather than just functions related to runtimes. We give some examples (again just working with functions) and give some laws for working with $O(\cdot)$. In §2.2, we introduce our two other expressions $\Omega(\cdot)$ and $\Theta(\cdot)$, again in the context of functions. In §2.3 we explain how O, Ω and Θ are used to bound the worst-case running time of Algorithms and for basic Data Structure operations. Note that in Inf2B, we almost always work with worst-case running time, apart from our treatment of Dynamic Arrays. For these we perform an *amortized* analysis, where we bound the running time of a series of operations.

Before going into details for the various notations it is worth stating that like most powerful tools they take a little getting used to. However the relatively small effort required is more than amply repaid; these notations help us to avoid extremely fiddly detail that is at best tedious and at worst can prevent us from seeing the important part of the picture. The analysis of algorithms is greatly eased with the use of these tools; learning how to use them is an essential part of the course.

Finally Appendix A at the end this note that places some matters in a general setting and should help with some misconceptions that have happened in the past (arising from a lack of separation of concerns). Appendix B is a quick introduction to proof by induction in case you have not met it before or need a reminder.

2.1 The “Big-O” Notation

$\mathbb{R}$ denotes the set of real numbers.

[1]Most of you will have seen this before, however there might be some joint degree students who have not done so. As we will be using this notation throughout the course it is essential to ensure that all have met it.

Definition 2.1. Let $f, g : \mathbb{N} \to \mathbb{R}$ be *functions* We say that f is $O(g)$, pronounced f *is big-O of* g, if and only if there is an $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have

$$0 \leq f(n) \leq cg(n).$$

Aside: A variant of the definition you might come across is to use absolute values for comparing the values of the functions, i.e., the required inequality is $|f(n)| \leq |g(n)|$. This has the virtue of ensuring non-negativity but forces us to work with absolute values all the time. In the end there is very little difference, the definition used in these notes is the same as the one in CLRS (see Note 1). The concern over non-negativity has a simple explanation: $-1000000 < 1$ but in terms of *size* (e.g., number of bits required to encode a number) we want to say that -1000000 is bigger than 1; comparing absolute values does the job. [Exercise for the dedicated: recall that a rational number is the ratio of two integers (a fraction), can we keep things so simple if we want to consider encoding size for rational numbers?]

Returning to the main definition, there is a slight nuisance arising from the fact that $\lg(n)$ is not defined for $n = 0$ so strictly speaking this is not a function from $\mathbb{N}$. There are various ways to deal with this, e.g., assign some arbitrary value to $\lg(0)$. We will never need to look at $\lg(0)$ so we will just live with the situation. Note also that the definition would make sense for functions $\mathbb{R} \to \mathbb{R}$ but as we will be studying runtimes measured in terms if input size the arguments to our functions are always natural numbers.

The role of n_0 can be viewed as a settling in period. Our interest is in long term behaviour (for all large enough values of n) and it is often convenient to have the freedom that n_0 allows us so that we do not need to worry about some initial atypical behaviour[2]. See Figure 2.2 for an illustration of this point.

Note that we also insist that the functions take on non-negative values from n_0 onwards. This condition is necessary to ensure that certain desirable properties hold. The fundamental issue is the following: if we have $a_1 \leq b_1$ and $a_2 \leq b_2$ then we can safely deduce that $a_1 + a_2 \leq b_1 + b_2$. Can we also deduce that $a_1 a_2 \leq b_1 b_2$? An example such as $-4 \leq 1$ and $-1 \leq 2$ but $-4 \times -1 \not\leq 1 \times 2$ shows that care must be taken. In fact the deduction we want to make is safe provided we have the extra information that $0 \leq a_1 \leq b_1$ and $0 \leq a_2 \leq b_2$ then it is indeed the case that $0 \leq a_1 a_2 \leq b_1 b_2$. From the point of view of our intended application area, runtime functions, the condition will be automatically true since runtimes are never negative. However we will be illustrating some points, and stating properties, with general mathematical functions and it is for this reason that the condition is there. Apart from a few early examples we will take it for granted and never check it for actual runtimes; indeed even in most of our general examples we have $f(n) \geq 0$ for all n so we just need to find n_0 and $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$.

To understand the role of c let’s consider the two functions $f(n) = 10n$ and $g(n) = n$. Now these functions have exactly the same rate of growth: if we multiply n by a constant a then the values of $f(n)$ and $g(n)$ are both multiplied by a.

[2]In terms of the runtime of algorithms, we typically have some set up cost of variables and data structures which dominates things for small size inputs but is essentially negligible for large enough inputs where we see the real trend.

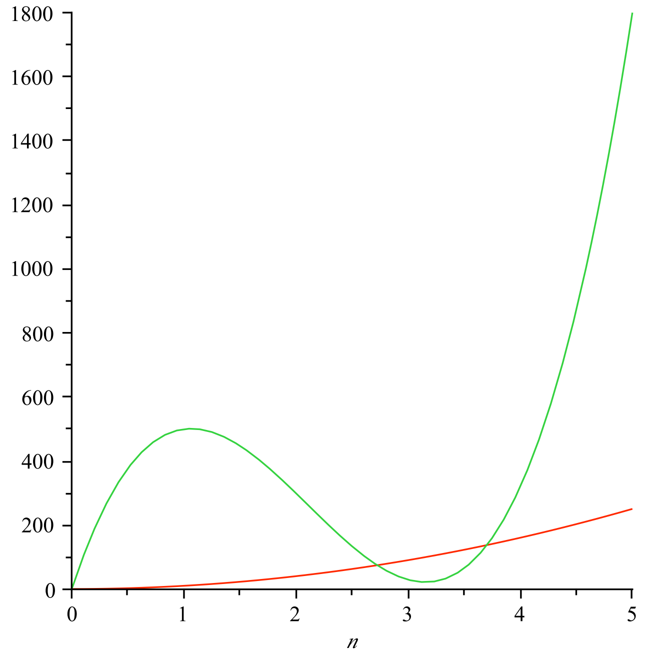

Figure 2.2. A graph of $f = 105n^3 - 665n^2 + 1060n$ and $g = 10n^2 + 1$. The graph illustrates the fact that $g(n) < f(n)$ for all $n > 4$ (indeed before that). So in verifying $g = O(f)$ we could take $n_0 = 4$, no need to make life hard by trying to find its exact value!

Yet we have $f(n) > g(n)$ for all $n > 0$. However choosing $c = 10$ (or any larger value) in the definition, we see that f is $O(g)$. Of course g is $O(f)$ as well but this is not always the case.

An informal (but useful) way of describing f is $O(g)$ is:

> *For large enough n, the rate of growth (w.r.t. n) of $f(n)$ is no greater than the rate of growth of $g(n)$.*

This latter informal description is useful, because it reminds us that if f is $O(g)$ it does *not* necessarily follow that we have $f(n) \le g(n)$, what *does* follow is that the rate of growth of f is bounded from above by the rate of growth of g.

More formally, we define $O(g)$ to be the following set of functions:

$$O(g) = \bigl\{ f : \mathbb{N} \to \mathbb{R} \bigm| \text{there is an } n_0 \in \mathbb{N} \text{ and } c > 0 \text{ in } \mathbb{R} \text{ such that for all } n \ge n_0 \text{ we have } 0 \le f(n) \le cg(n). \bigr\}$$

Then f is $O(g)$ simply means $f \in O(g)$.

It is worth making another observation here. If we say that f is a function then f is the *name* of the function and does not denote any value of the function. To denote a value we use $f(n)$ where n denotes an appropriate value in the domain of the function[3]. Thus writing $f \le g$ is meaningless unless we have defined some method of comparing functions as a whole rather than their values (we will not do this).

Notational convention. As we have seen, for a function g the notation $O(g)$ denotes a *set* of functions. Despite this it is standard practice to write $f = O(g)$ rather than $f \in O(g)$. This is in fact a very helpful way of denoting things, provided we bear in mind the meaning of what is being said. This convention enables us to pursue with ease chains of reasoning such as:

$$\begin{aligned}
871n^3 + 13n^2 \lg^5(n) + 18n + 566 &= 871n^3 + 13n^2 O(n) + 18n + 566 \\
&= 871n^3 + O(n^3) + 18n + 566 \\
&= O(n^3) + O(n^3) + O(n^3) + O(n^3) \\
&= O(n^3).
\end{aligned}$$

(Don't worry just now about the details, we will see this example later with full explanations.) To clarify, the first equality asserts that there is a function $g \in O(n)$ such that $871n^3 + 13n^2 \lg^5(n) + 18n + 566 = 871n^3 + 13n^2 g(n) + 18n + 566$ (this is because it can be shown that any *fixed* power of $\lg(n)$ is $O(n)$). The second line then asserts that whatever function h we have from $O(n)$ the function $13n^2 h(n)$ is $O(n^3)$. Looking now at the final equality, the assertion is that for any functions g_1, g_2, g_3, g_4 all from $O(n^3)$ we have that their sum (i.e., the function whose value at n is $g_1(n) + g_2(n) + g_3(n) + g_4(n)$) is again $O(n^3)$.

In our discussion we have also followed standard practice and suppressed the argument of a function, writing f rather than $f(n)$, whenever we do not need

[3]It is also common practice to use $f(n)$ as the name of the function when n is a variable in order to indicate the variable and that f is a univariate function. The point is that f by itself can never denote a value of the function and should never be used where a value is intended. Despite this, significant numbers of students persist in the meaningless practice and lose marks in exams as a result.

to stress it in any way. Mathematical notation is precise but is in fact closer to a natural language than non-mathematicians seem to imagine. Provided the conventions are understood a well established notation aids comprehension immensely. Judge for yourself which of the following is easier to understand and remember (they state the same thing):

- If $f_1(n) \in O\big(g_1(n)\big)$ and $f_2(n) \in O\big(g_2(n)\big)$ then $f_1(n) + f_2(n) \in O\big(g_1(n) + g_2(n)\big)$.
- If $f_1 = O(g_1)$ and $f_2 = O(g_2)$ then $f_1 + f_2 = O(g_1 + g_2)$.

Just one final word of warning about our usage. When we write $f = O(g)$ the equality is to be read in a directed way from left to right. Remember that what this really says is that $f \in O(g)$. This is of course at variance with the normal meaning of equality: $a = b$ means exactly the same as $b = a$. There is no possible confusion though if we bear in mind the intended meaning when a O appears, the gain is more than worth the need for interpreting equality in context.

Warning. The preceding discussion does *not* give you license to invent our own notational conventions (any more than speaking a language gives you license to invent arbitrary new words). Such conventions arise out of very long periods of usage with the useful ones being modified as necessary and finally adopted widely. A professional mathematician will introduce a new notational convention only after long thought and if it genuinely helps.

Mathematical writing. Here are some useful extracts from the notes for contributors to the London Mathematical Society:

(1) Organize your writing so that sentences read naturally even when they incorporate formulae.

(2) Formulae and symbols should never be separated merely by punctuation marks except in lists; one can almost always arrange for at least one word to come between different formulae.

(3) Draft sentences so that they do not begin with formulae or symbols.

(4) Never use symbols such as $\exists$ and $\forall$ as abbreviations in text.

In fact my view is that for the inexperienced user of Mathematics it is reasonable to say that excessive use of formal logical symbols often indicates a confusion of mind and represents the triumph of hope over understanding (based on the misguided belief that the symbols posess some kind of magic). Indeed for this course there is no need at all to use formal logical symbols such as $\exists$ and $\forall$ (with the exception of '$\Rightarrow$' and '$\Leftrightarrow$' which can be useful in a sequence of derivations—but again you must take great care to use them sensibly[4]). To put it briefly, a Mathematical argument must read fluently; the aim is to aid comprehension not

[4]I have seen, far too often, the use of these symbols where at least one side is not a statement! $P \Rightarrow Q$ and $P \Leftrightarrow Q$ are meaningless unless P and Q are both statements. It is also very common to misuse implication, getting it the wrong way round. To be precise if we have proved that $P \Rightarrow Q$ and that Q is true, we know *nothing* about P as a result of this reasoning. By contrast if we have proved that $P \Rightarrow Q$ and that P is true then we can deduce correctly that Q is also true. If, on the other hand, we know that Q is false then we can deduce that P is false.

to mask inadequate understanding. If your reasoning is faulty you are much more likely to spot this by expressing things clearly. This should come as no surprise, consider trying to understand and perhaps debug a well laid out computer program as opposed to one that is all over the place. The final presentation of your ideas should be *clear*, *concise* and *correct*.

Failure to ensure that a mathematical argument reads fluently with appropriate connecting and explanatory words accounts for a great number of common mistakes. Do not be misled by the fact that in many presentations (e.g., lectures) arguments are often sketched out. This is largely done to save time and the connecting material is usually presented orally rather than being written down. Naturally when we are first trying to find the solution to a problem we take such short cuts, the point is to work towards a final full presentation. Get into the habit of doing this if you have not already done so.

We end this digression by illustrating the need for clear precise language going hand in hand with clear precise thinking and understanding. Recall the key definition that is under discussion in this section:

> We say that f is $O(g)$ if and only if there is an $n_0 \in \mathbb{N}$ and a $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have $0 \leq f(n) \leq cg(n)$.

The wording is important. For example the phrase 'there is an $n_0 \in \mathbb{N}$ and a $c > 0$ in $\mathbb{R}$' tells us that for a given pair of functions f, g we must produce a *single choice* of n_0, and of c that do the required job (i.e., such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$). It would be wrong to start the definition with 'for all n there are $n_0 \in \mathbb{N}$ . . . ' This allows us to change n_0 (and c) with n which is certainly not the intention of the definition [would a definition that allowed us to change n_0 and c in this way but was otherwise as stated above be of any interest?]. When reading definitions (or any piece of mathematics) take care to understand such subtleties.

We now consider some examples where we just think about bounding functions in terms of other functions; this helps us to focus on understanding the notation rather than its applications. Later in this lecture note we will directly consider functions that represent the running-time of algorithms.

Examples 2.3.

(1) Let $f(n) = 3n^3$ and $g(n) = n^3$. Then $f = O(g)$.

PROOF: First of all we observe that $f(n) \geq 0$ for all n so we just need to find an n_0 and $c > 0$ such that $f(n) \leq g(n)$ for all $n \geq n_0$.

Let $n_0 = 0$ and $c = 3$. Then for all $n \geq n_0$, $f(n) = 3n^3 = cg(n)$.

Well that is clearly correct but how do we come up with appropriate values for n_0 and c? In this very simple case it is easy enough to see what values to choose. As a first illustration let's investigate what is needed for the claim to be true. We need to find $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$. We have

$$3n^3 \leq cn^3 \Leftrightarrow 3 \leq c, \quad \text{provided } n > 0.$$

Here we are using the simple fact that if $ab \leq ac$ and $a > 0$ then we may divide out by a to obtain $b \leq c$. Conversely, if $a > 0$ (actually $a \geq 0$ is enough here) and $b \le c$ then we may multiply by a to obtain $ab \le ac$. (What goes wrong if we ignore the requirement that $a \ge 0$?) It follows that if we take $c = 3$ and $n_0 = 1$ the requirement for $f = O(g)$ is satisfied.

We note here that in the preceding argument we do not need the full equivalence $3n^3 \le cn^3 \Leftrightarrow 3 \le c$ it is enough to have $3 \le c \Rightarrow 3n^3 \le cn^3$. This shows that we could take $n_0 = 0$ though this is not of any importance here.

(2) $3n^3 + 8 = O(n^3)$.

PROOF: As above we have $3n^3 + 8 \ge 0$ for all n.

For this example we will give two proofs, using different constants for c, n_0. This is just to show that there are alternative constants that can be chosen (though only certain c, n_0 pairs will work).

First proof: let $c = 4$ and $n_0 = 2$. Then for all $n \ge n_0$ we have $n^3 \ge 8$ and thus $3n^3 + 8 \le 3n^3 + n^3 = 4n^3 = cn^3$.

Second proof: let $c = 11$ and $n_0 = 1$. Then for all $n \ge n_0$ we have $n^3 \ge 1$, and therefore $3n^3 + 8 \le 3n^3 + 8n^3 = 11n^3 = cn^3$.

Again let's investigate the situation to find values for c and n_0. For a constant $c > 0$ we have

$$3n^3 + 8 \le cn^3 \iff 3 + \frac{8}{n^3} \le c, \quad \text{provided } n > 0.$$

Now we note that as n increases $8/n^3$ decreases. It follows that

$$3 + \frac{8}{n^3} \le 11, \quad \text{for all } n > 0.$$

So if we take $c = 11$ and $n_0 = 1$ all the requirements are satisfied. In fact this derivation combines both of the previous proofs. If we take $n_0 = 2$ then $8/n^3 \le 1$ so that $3 + 8/n^3 \le 4$ for all $n \ge n_0$. Indeed we see that by taking n_0 sufficiently large we can use for c any value that is *strictly* bigger than 3 [can we use 3 as the value of c?].

(3) $\lg(n) = O(n)$

PROOF: Note that $\lg(n) \ge 0$ for all $n \ge 1$ (in any case we cannot have $n = 0$ as $\lg(0)$ is undefined), this tells us that our choice of n_0 must be at least 1 but we need to look at possible further requirements for the main inequality to hold.

Based on our discussion in Note 1 we would expect that in fact $\lg(n) < n$ for all $n \ge 1$. So we will try to prove this claim (in effect we are taking $n_0 = 1$ and $c = 1$). We have

$$\lg(n) < n \iff n < 2^n, \quad \text{for all } n > 0.$$

(To be formal we are using the fact that the exponentiation and logarithmic functions are strictly increasing in their argument. For the purposes of this proof all we need is that $n < 2^n \Rightarrow \lg(n) < n$.)

We prove that $2^n > n$ for all $n > 0$ by induction on n. The base case $n = 1$ is clearly true. Now for the induction step let us assume that the claim holds for n. Then

$$2^{n+1} = 2 \cdot 2^n > 2n,$$

where the inequality follows from the induction hypothesis. To complete the proof we need to show that $2n \ge n + 1$. Now

$$2n \ge n + 1 \iff n \ge 1,$$

and we have finished.

(4) $8n^2 + 10n\lg(n) + 100n + 10000 = O(n^2)$.

PROOF: As before $8n^2 + 10n\lg(n) + 100n + 10000 \ge 0$ for all n (we are lucky here because there are no negative coefficients).

We have

$$\begin{aligned} 8n^2 + 10n\lg(n) + 100n + 10000 &\le 8n^2 + 10n \cdot n + 100n + 10000, \quad \text{for all } n > 0 \\ &\le 8n^2 + 10n^2 + 100n^2 + 10000n^2 \\ &= (8 + 10 + 100 + 10000)n^2 \\ &= 10118n^2. \end{aligned}$$

Thus we can take $n_0 = 1$ and $c = 1118$.

The value for c seems rather large. This is irrelevant so far as the definition of big-O is concerned. However it is worth noting here that we could decrease the value of c at the expense of a relatively small increase in the value of n_0. We can illustrate this point with the graph in Figure 2.4. Indeed any $c > 8$ will do, the closer c is to 8 the larger n_0 has to be (see the discussion above for the second example). In the context of our intended usage of the big-O notation there is no point at all in expending more effort just to reduce some constant. In practice for runtimes the constants will depend on the implementation of the algorithm and the hardware on which it is run. We could determine the constants, if needed, by appropriate timing studies. What is independent of the factors mentioned is the growth rate and that is exactly what asymptotic notation gives us.

Warning. Graphs are very helpful in suggesting the behaviour of functions. Used carefully they can suggest the choice of n_0 and c when trying to prove an asymptotic relation. However they do *not* prove any such claim. There are at least two objections. Firstly all plotting packages can get things wrong (admittedly they are very reliable in straightforward situations). Secondly, and more seriously, a displayed graph can only show us a finite portion of a function's behaviour. We have no guarantee that if the graph is continued the suggested trend will not change (it is easy enough to devise examples of this). Like many tools they are great if used with appropriate care, dangerous otherwise.

(5) $2^{100} = O(1)$. That is, $f = O(g)$ for the functions defined by $f(n) = 2^{100}$ and $g(n) = 1$ for all $n \in \mathbb{N}$; both functions are constants.

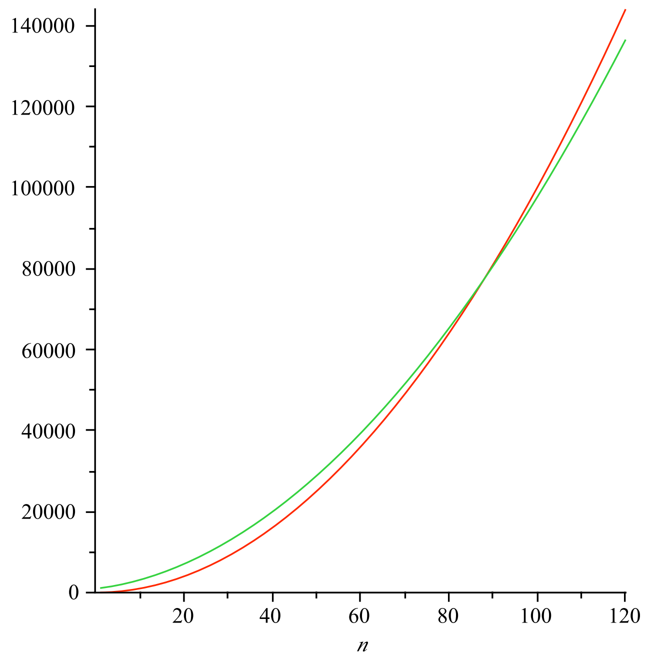

Figure 2.4. A graph of $8n^2 + 10n\lg(n) + 100n + 10000$ and $10n^2$.

PROOF: Let $n_0 = 1$ and $c = 2^{100}$.

It should be obvious that there is nothing special about 2^{100}, we could replace it by *any* non-negative constant and the claim remains true (with an obvious modification to the proof).

Note that all of the examples from (1) to (5) are proofs by first principles, meaning that we prove the "big-O" property using Definition 2.1, justifying everything directly. Theorem 2.5 lists some general laws, which can be generally used in simplifying "'big-O" expressions, and which make proofs of "big-O" shorter and easier.

Theorem 2.5. *Let $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}$ be functions, then*

(1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = O(g_1) \implies af_1 = O(g_1)$.

(2) $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 + f_2 = O(g_1 + g_2)$.

(3) $f_1 = O(g_1)$ and $f_2 = O(g_2) \implies f_1 f_2 = O(g_1 g_2)$.

(4) $f_1 = O(g_1)$ and $g_1 = O(g_2) \implies f_1 = O(g_2)$.

(5) For any $d \in \mathbb{N}$: if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = O(n^d)$.

(6) For any constants *$a > 0$ and $b > 1$ in $\mathbb{R}$: $n^a = O(b^n)$.*

(7) For any constant *$a > 0$ in $\mathbb{R}$: $\lg(n^a) = O(\lg(n))$.*

(8) For any constants *$a > 0$ and $b > 0$ in $\mathbb{R}$: $\lg^a(n) = O(n^b)$.*

Note that $\lg^a(n)$ is just another way of writing $(\lg(n))^a$.

The following example shows how the facts of Theorem 2.5 can be applied:

Example 2.6. We will show that $871n^3 + 13n^2\lg^5(n) + 18n + 566 = O(n^3)$.

$$\begin{aligned}
871n^3 + 13n^2\lg^5(n) + 18n + 566 &= 871n^3 + 13n^2 O(n) + 18n + 566 && \text{by Theorem 2.5(8)}\\
&= 871n^3 + O(n^3) + 18n + 566 && \text{by Theorem 2.5(3)}\\
&= 871n^3 + 18n + 566 + O(n^3)\\
&= O(n^3) + O(n^3) && \text{by Theorem 2.5(5)}\\
&= O(n^3) && \text{by Theorem 2.5(2)}
\end{aligned}$$

In §2.3 we will see how O is used in the analysis of the (worst-case) running time of algorithms.

We will not give the proof of every claim in Theorem 2.5. Most parts are straightforward consequences of the definition with (6) and (8) requiring extra facts. For illustration we will prove part (5). First recall that to say $f : \mathbb{N} \to \mathbb{R}$ is a polynomial function of degree d means that there are $a_0, a_1, \dots, a_d \in \mathbb{R}$ with $a_d \neq 0$ such that

$$f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0,$$

for all $n \in \mathbb{N}$. (In this definition if we allow the possibility that $a_d = 0$ then the polynomial is of degree *at most* d; the proof is fine with this provided the actual leading coefficient is strictly positive[5].) It is very important to note that for a given polynomial d is a *constant*; it can be different for different polynomials but cannot vary once a polynomial is chosen (only n varies). Our statement assumes that $a_d > 0$, this is because if $a_d < 0$ the polynomial takes on negative values for all large enough n. So our task here is to find an n_0 and $c > 0$ such that

(1) $f(n) \geq 0$ and

(2) $f(n) \leq cn^d$

for all $n \geq 0$.

We will show the second of these properties here. The first one will follow from our discussion on p. 13; so the final choice of n_0 will be the maximum of the one found here and the one found later on. The claim is not automatically true, for example if $f(n) = n - 100$ then $f(n) < 0$ for all $n < 100$. Now

$$\begin{aligned} a_dn^d + a_{d-1}n^{d-1} + \cdots + a_1n + a_0 &\leq |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0| \\ &\leq |a_d|n^d + |a_{d-1}|n^d + \cdots + |a_1|n^d + |a_0|n^d \qquad \text{for all } n > 0 \\ &= (|a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|)n^d. \end{aligned}$$

So we can take $n_0 = 1$ and $c = |a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|$. There is a slight subtlety if we allow $a_d = 0$ since then we might have $|a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0| = 0$ but the definition of O requires that $c > 0$. No problem we just take $c = \max(1, |a_d| + |a_{d-1}| + \cdots + |a_1| + |a_0|)$ and then our proof works in all cases.

We can prove the claim in a slightly different way (essentially generalising the discussion of Example 2 on page 7). We have

$$a_dn^d + a_{d-1}n^{d-1} + \cdots + a_1n + a_0 \leq |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0|,$$

for all n. Now for all $n > 0$

$$\begin{gathered} |a_d|n^d + |a_{d-1}|n^{d-1} + \cdots + |a_1|n + |a_0| \leq cn^d \iff \\ |a_d| + \frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d} \leq c \end{gathered}$$

The left hand side decreases as n increases. Taking $n_0 = 1$ gives us value for c we derived above (the same observation applies about the possibility that $a_d = 0$). By taking n_0 sufficiently large we can choose c to be as close to $|a_d|$ as we like (but not equal to it unless all the other coefficients are 0). You should think carefully about what can go wrong if we do not take the absolute values of the coefficients; when doing something like this simple examples can be very helpful. In this case consider a polynomial such as $n - 2$ and try the previous proofs without taking absolute values (you should find that in both cases things don't work out).

[5]By definition the leading coefficient of a polynomial is non-zero so in this context it would be enough to say that it is positive but we use 'strictly' to stress that it is non zero. For the sake of completeness it is worth pointing out that the zero polynomial (i.e., all coefficients are 0) has no degree and no leading coefficient, in contrast to all others. This is not worth worrying about for the applications of this course, no algorithm has 0 runtime!

2.2 Big-Ω and Big-Θ

"f is $O(g)$" is a concise and mathematically precise way of saying that, "up to a constant factor and for sufficiently large n, the function $f(n)$ grows at a rate no faster than $g(n)$". Sometimes, we also want to give *lower bounds*, i.e., make statements of the form "up to a constant factor, $f(n)$ grows at a rate at least as fast as $g(n)$". Big-Omega (written as Ω) is the analogue of big-O for this latter kind of statement.

Definition 2.7. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that f is $\Omega(g)$ if there is an $n_0 \in \mathbb{N}$ and $c > 0$ in $\mathbb{R}$ such that for all $n \geq n_0$ we have

$$f(n) \geq cg(n) \geq 0.$$

Informally, we say that f is big-Ω of g if there is some positive constant c such that for all *sufficiently large* n (corresponding to the n_0) we have $f(n) \geq cg(n) \geq 0$.

A more informal (but useful) way of describing $f = \Omega(g)$ is:

> *For large-enough n, the rate-of-growth (wrt n) of $f(n)$ is no less than the rate-of-growth of $g(n)$.*

Not surprisingly we can state laws for big-Ω that are similar to those for big-O:

Theorem 2.8. *Let $f_1, f_2, g_1, g_2 : \mathbb{N} \to \mathbb{R}$ be functions, then*

(1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = \Omega(g_1) \implies af_1 = \Omega(g_1)$.

(2) $f_1 = \Omega(g_1)$ and $f_2 = \Omega(g_2) \implies f_1 + f_2 = \Omega(g_1 + g_2)$.

(3) $f_1 = \Omega(g_1)$ and $f_2 = \Omega(g_2) \implies f_1f_2 = \Omega(g_1g_2)$.

(4) $f_1 = \Omega(g_1)$ and $g_1 = \Omega(g_2) \implies f_1 = \Omega(g_2)$.

(5) For any $d \in \mathbb{N}$: if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = \Omega(n^d)$.

(6) For any constant $a > 0$ *in $\mathbb{R}$: $\lg(n^a) = \Omega(\lg(n))$.*

Compare this with Theorem 2.5 for big-O and note that two items from there do not have corresponding ones in the current theorem [why?]. Let's prove the claim of item 5, bearing in mind that in the following we are assuming $a_d > 0$.

$$a_dn^d + a_{d-1}n^{d-1} + \cdots + a_1n + a_0 \geq a_dn^d - |a_{d-1}|n^{d-1} - \cdots - |a_1|n - |a_0|,$$

for all n. (Note that we do not take the absolute value of a_d since we have assumed that $a_d > 0$.) For all $n > 0$

$$\begin{gathered} a_dn^d - |a_{d-1}|n^{d-1} - \cdots - |a_1|n - |a_0| \geq cn^d \iff \\ a_d - \left(\frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d} \right) \geq c \end{gathered}$$

Now $|a_{d-1}|/n + \cdots + |a_1|/n^{d-1} + |a_0|/n^d$ is always non-negative and is a decreasing function of n which tends to 0 as n increases. So there is some value n_0 of n such that

$$a_d > \frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d}$$

for all $n \geq n_0$. So we can take $c = a_d - (|a_{d-1}|/n_0 + \cdots + |a_1|/n_0^{d-1} + |a_0|/n_0^d)$ to satisfy the definition of big-Ω and we can use the n_0 we have already identified. Note that we do indeed have $c > 0$ as required owing to the assumption that $a_d > 0$ and the choice of n_0. Since $cn^d \geq 0$ for all n (and hence for all $n \geq n_0$ we are done. Note that we have proved here that $a_dn^d + a_{d-1}n^{d-1} + \cdots + a_1n + a_0 \geq 0$ for all $n \geq n_0$ just as promised on p. 11.

You might find the preceding proof less satisfactory than the corresponding one for big-O because there we could give the value of the constants directly in terms of the given coefficients[6]. We can do the same here: let $b = \max\{|a_{d-1}|, \ldots, |a_1|, |a_0|\}$ then

$$\begin{aligned}\frac{|a_{d-1}|}{n} + \cdots + \frac{|a_1|}{n^{d-1}} + \frac{|a_0|}{n^d} &\leq b\left(\frac{1}{n} + \cdots + \frac{1}{n^{d-1}} + \frac{1}{n^d}\right)\\ &\leq \frac{bd}{n}.\end{aligned}$$

So if we ensure that $a_d > bd/n$ then we can take $c = a_d - bd/n$. Now $a_d > bd/n$ if and only if $n > bd/a_d$ (since $a_d > 0$) so we just need n to be at least as large as the next integer after bd/a_d. This is our value of n_0 and it gives us $c = a_d - bd/n_0$.

It is a good idea for you to consider why the claim is false (always) if $a_d \leq 0$. The fact that the preceding proof does not work is of course not enough to establish that the claim is false, maybe some other proof works (none does).

Finally, in this course our functions measure runtime and as a consequence the leading coefficient of any polynomial that we consider will necessarily be strictly positive. A polynomial with negative leading coefficient takes on negative values for all large enough n (indeed it goes to $-\infty$ as n goes to ∞), this gives a way to tackle the problem of the preceding paragraph).

We can combine big-O and big-Ω as follows:

Definition 2.9. Let $f, g : \mathbb{N} \to \mathbb{R}$ be functions. We say that f is $\Theta(g)$, or f *has the same asymptotic growth rate as* g, if f is both $O(g)$ and $\Omega(g)$.

Note that f is $\Theta(g)$ if and only if f is $O(g)$ and g is $O(f)$.

In the examples that follow we will just present the verification of each claim for the stated values of n_0 and c. Work out "investigation" type proofs as well.

Examples 2.10.

(1) Let $f(n) = 3n^3$ and $g(n) = n^3$. Then $f = \Omega(g)$.
(combining this with Example 2.3, (1), will give $3n^3 = \Theta(n^3)$)

PROOF: Let $n_0 = 0$ and $c = 1$. Then for all $n \geq n_0$, $f(n) = 3n^3 \geq cg(n) = g(n) \geq 0$.

[6]Note that the definition only requires that the constants exist, it does not ask for a method to find them. So as long as we prove their existence the definition is satisfied thus the first proof is perfectly OK.

(2) Let $f(n) = \lg(n)$ and $g(n) = \lg(n^2)$. Then $f = \Omega(g)$

PROOF: Let $n_0 = 1$ and $c = 1/2$. Then for every $n \geq n_0$ we have,

$$f(n) \;=\; \lg(n) \;=\; \frac{1}{2}2\lg(n) \;=\; \frac{1}{2}\lg(n^2) \;=\; \frac{1}{2}g(n).$$

The only interesting step above is the conversion of $2\lg(n)$ to $\lg(n^2)$. This follows from the well-known property of logs, $\log(a^k) = k\log(a)$.

(3) if f_1 is a polynomial of degree d with strictly positive leading coefficient then $f_1 = \Theta(n^d)$.

PROOF: By Theorem 2.5 $f = O(n^d)$ while, by Theorem 2.8, $f = \Omega(n^d)$.

2.3 Asymptotic Notation and worst-case Running Time

Asymptotic notation seems well-suited towards the analysis of algorithms. The big-O notation, $O(\,\cdot\,)$, allows us to put an *upper bound* on the rate-of-growth of a function, and $\Omega(\,\cdot\,)$ allows us to to put a *lower bound* on the rate-of-growth of a function. Thus these concepts seem ideal for expressing facts about the running time of an algorithm. We will see that this is so, asymptotic notation is essential, but we need to be careful in how we define things. It is helpful to remind ourselves of the definition of *worst-case* running time, as this is our standard measure of the complexity of an algorithm:

Definition 2.11. The *worst-case running time* of an algorithm A is the function $T_{\mathsf{A}} : \mathbb{N} \to \mathbb{N}$ where $T_{\mathsf{A}}(n)$ is the maximum number of computation steps performed by A over all inputs of size n.

We will now also define a second concept, the *best-case* running time of an algorithm. We do *not* consider best-case to be a good measure of the quality of an algorithm, but we will need to refer to the concept in this discussion.

Definition 2.12. The *best-case running time* of an algorithm A is the function $B_{\mathsf{A}} : \mathbb{N} \to \mathbb{N}$ where $B_{\mathsf{A}}(n)$ is the minimum number of computation steps performed by A over all inputs of size n.

One way to use asymptotic notation in the analysis of an algorithm A is as follows:

- Analyse A to obtain the worst-case running time function $T_{\mathsf{A}}(n)$.
- Go on to derive upper and lower bounds on (the growth rate of) $T_{\mathsf{A}}(n)$, in terms of $O(\,\cdot\,)$, $\Omega(\,\cdot\,)$ and $\Theta(\,\cdot\,)$.

In fact such an approach would fail to capitalise on one of the most useful aspects of asymptotic notation. It gives us a way of focusing on the essential details (the overall growth rate) without being swamped by incidental things (unknown constants or sub-processes that have smaller growth rate).

```
Algorithm linSearch$(A, k)$
  Input:  An integer array $A$, an integer $k$
  Output: The smallest index $i$ with $A[i] = k$, if such an $i$ exists,
          or $-1$ otherwise.

  1. for $i \leftarrow 0$ to $A.length - 1$ do
  2.     if $A[i] = k$ then
  3.         return $i$
  4. return $-1$
```

Algorithm 2.13

Linear search

We now give our first example using the linSearch algorithm of Note 1. For this first example we will carry out the analysis by the two step process (since we have already carried out the first step).

Recall that we use small positive constants c_1, c_2, c_3, c_4 to represent the cost of executing line 1, line 2, line 3 and line 4 exactly once. The worst-case running time of linSearch on inputs of length n satisfies the following inequality:

$$(c_1 + c_2)n + \min\{c_3, (c_1 + c_4)\} \leq \quad T_{\mathsf{linSearch}}(n) \leq \quad (c_1 + c_2)n + \max\{c_3, (c_1 + c_4)\}.$$

For linSearch, the best-case (as defined in Definition 2.12) will occur when we find the item searched for at the first index. Hence we will have $B_{\mathsf{linSearch}}(n) = c_1 + c_2 + c_3$.

Now we show that $T_{\mathsf{linSearch}}(n) = \Theta(n)$ in two steps.

$O(n)$: We know from our analysis in Note 1 that

$$T_{\mathsf{linSearch}}(n) \leq (c_1 + c_2)n + \max\{c_3, (c_1 + c_4)\}.$$

Take $n_0 = \max\{c_3, (c_1 + c_4)\}$ and $c = c_1 + c_2 + 1$ in the definition of $O(\cdot)$. Then for every $n \geq n_0$, we have

$$T_{\mathsf{linSearch}}(n) \leq (c_1 + c_2)n + n_0 \leq (c_1 + c_2 + 1)n = cn,$$

and we have shown that $T_{\mathsf{linSearch}}(n) = O(n)$.

$\Omega(n)$: We know from our Note 1 analysis that

$$T_{\mathsf{linSearch}}(n) \geq (c_1 + c_2)n + \min\{c_3, (c_1 + c_4)\}.$$

Hence $T_{\mathsf{linSearch}}(n) \geq (c_1 + c_2)n$, since all the c_i were positive. In the definition of Ω, take $n_0 = 1$ and $c = c_1 + c_2$, and we are done.

Finally by definition of Θ (in §2.2), we have $T_{\mathsf{linSearch}}(n) = \Theta(n)$. We then say that the *asymptotic growth rate* of linSearch is $\Theta(n)$.

The significance of the preceding analysis is that we have a guarantee that the upper bound is not larger than necessary and at the same time the lower bound is not smaller than necessary. We will discuss this further in §2.3

Finally note just how tedious is the process of keeping track of unknown constants. Not only that but readability is severely hampered. This disappears as soon as we switch to full use of asymptotic notation. Let's illustrate the point with linSearch.

Upper bound. As before we let n be the length of the input array. Line 1 is executed at most $n = O(n)$ times. It controls the loop body on lines 2 and 3. Each of these lines costs a constant, so each costs $O(1)$, and of course line 1 itself costs $O(1)$ each time it is executed. So the overall cost here is $O(n)(O(1)+O(1)+O(1)) = O(n)O(1) = O(n)$. Line 4 is executed once and costs $O(1)$ and so the total cost of the algorithm is $O(n) + O(1) = O(n)$.

Note: Strictly speaking line 1 is executed at most $n+1$ times, the extra one being to discover that the loop variable is out of bounds (i.e., $i > A.length - 1$) in those cases where the integer k does not occur in the array. This possible final time does not cause the execution of the loop body and costs a constant. Since our runtimes will always be at least a non-zero constant the cost contributed by such extra executions of loop control lines would be absorbed in the overall runtime (in terms of our analysis above the cost is $O(n) + O(1) = O(n)$). It follows that we can safely ignore this extra execution and we will do this from now on. A bit of thought shows that the same reasoning applies to nested loops as well. For the avoidance of doubt we will always count the cost of executions of a loop and its body but not the constant incurred in finding that a loop variable is out of bounds.

Lower bound. Consider any input where the integer k does not occur in the array. Then the condition in the loop is never true so we never execute the return in line 3. It follows that line 1 is executed at least $n = \Omega(n)$ times. Since each execution of the line costs $\Omega(1)$ the overall cost of this line alone is $\Omega(n)\Omega(1) = \Omega(n)$. Thus the cost of the algorithm is $\Omega(n)$.

Note: We did not bother to count the cost of lines 2 and 4 of the algorithm. This is because we already know the upper bound is $O(n)$ so once we have reached an $\Omega(n)$ lower bound we know that the rest will not increase this asymptotic cost. So why do the extra work? Let's be clear: counting the extra lines would of course have an effect on the *precise* runtime but the growth rate remains the same.

Misconceptions about O, Ω

This section is aimed at dispelling two common misconceptions, *both* of which arise from over-interpreting O's significance as an "upper bound" and Ω's significance as a "lower bound". Figure 2.14 concerning linSearch will be useful.

Misconception 1: If for some algorithm A we are able to show $T_{\mathsf{A}}(n) = O(f(n))$ for some function $f : \mathbb{N} \to \mathbb{R}$, then the running time of A is bounded by $f(n)$ for sufficiently large n.

FALSE: If $T_{\mathsf{A}}(n) = O(f(n))$, the *rate of growth* of the worst-case running time of A (with respect to the size n of the input) grows no faster than the *rate of growth* of f (wrt n). But $T_{\mathsf{A}}(n)$ may be larger than $f(n)$, *even* as n gets large—we are only

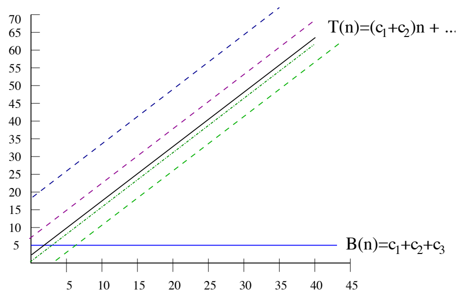

Figure 2.14. Best-case running time and worst-case running time for linSearch.

guaranteed that $T_{\mathsf{A}}(n) \le cf(n)$ for large n, where $c > 0$ is the constant used in proving that $T_{\mathsf{A}}(n)$ is $O(f(n))$.

As a concrete example of this, re-consider our analysis of the worst-case running-time of linSearch. We showed that $T_{\mathsf{linSearch}} = O(n)$ above. We never pinned down any particular values for c_1, c_2, c_3, c_4. However, whatever they are, we could have shown $T_{\mathsf{linSearch}} = O(\frac{1}{2}(c_1 + c_2)n)$ in *exactly* the same way as we showed $T_{\mathsf{linSearch}} = O(n)$ (but with a different value of c). However, it is certainly not the case that $T_{\mathsf{linSearch}}(n) \le \frac{1}{2}(c_1 + c_2)n$ for any n. In fact for any constant $\alpha > 0$, we could have shown $T_{\mathsf{linSearch}} = O(\alpha n)$. To see this graphically, look at Figure 2.3, where the various lines surrounding the "worst-case" line, both above and below the line for $T_{\mathsf{linSearch}}$ all represent potential $f(n)$ functions for $T_{\mathsf{linSearch}}(n)$ (there are infinitely many such functions).

Misconception 2: We know that the $T_{\mathsf{A}}(n) = O(f(n))$ implies that $cf(n)$ is an "upper bound" on the running time of the algorithm A on *any* input of size n, for some constant $c > 0$. This fact can sometimes mislead newcomers into believing that the result $T_{\mathsf{A}}(n) = \Omega(g(n))$ implies a lower bound on the running-time of A on all inputs of size n (with some constant being involved), which is **FALSE**. This confusion arises because there is an asymmetry in our use of O and Ω, when we work with *worst-case* running time.

O: Suppose we know that $T_{\mathsf{A}}(n) = O(f(n))$ for some function $f : \mathbb{N} \to \mathbb{R}$. Then we know that there is a constant $c > 0$ so that $T_{\mathsf{A}}(n) \le cf(n)$ for all sufficiently large n. Hence, because $T_{\mathsf{A}}(n)$ is the worst-case running time, we know that $cf(n)$ is an upper bound on the running time of A for *any input of size* n.

Ω: Suppose we know that $T_{\mathsf{A}}(n) = \Omega(g(n))$ for some function $g : \mathbb{N} \to \mathbb{R}$. Well, we know there is a constant $c' > 0$ such that $T_{\mathsf{A}}(n) \ge c'g(n)$ for all sufficiently large n. This part of the argument works fine. However, $T_{\mathsf{A}}(n)$ is the *worst-case running time*, so for any n, there may be many inputs of size n on which A takes far less time than on the worst-case input of size n. So

the $\Omega(g(n))$ result does *not* allow us to determine any lower bound at all on *general* inputs of size n. It only gives us a lower bound on worst-case running time, i.e., there is at least one input of size n on which the algorithm takes time $c'g(n)$ or more.

As a concrete example of this, note that in Figure 2.3, any of the parallel lines to $T_{\mathsf{linSearch}}(n)$ are potential $\Omega(\cdot)$ functions for $T_{\mathsf{linSearch}}(n)$. These lines are far above the line for the best-case of linSearch. In fact *every* function f satisfying $T_{\mathsf{linSearch}}(n) = \Theta(f(n))$ (including those parallel lines to $T_{\mathsf{linSearch}}(n)$) is guaranteed to overshoot the line for $B_{\mathsf{linSearch}}(n)$ when n gets large enough.

Note that the asymmetry is not in the definitions of O and Ω but in the use to which we put them, i.e., producing bounds on the worst case running time of algorithms. To illustrate this point, suppose we have a group of people of various heights and the tallest person has height 1.9m. Then any number above 1.92 is an upper bound on their height but contains less information than the exact one of 1.92. If we want to be sure that 1.92 is as good as we can get then all that is necessary is to produce at least one person whose height is at least 1.92m. Note the asymmetry: for the upper bound *all* persons must satisfy it, for the lower bound *at least one* must do so.

Why bother with Ω? Is O not enough?

Why use $\Omega(\cdot)$ to bound $T_{\mathsf{A}}(n)$, when upper bounds are what we really care about? We can explain this very easily by looking at linSearch again. In §2.3 we first proved that $T_{\mathsf{linSearch}}(n) = O(n)$. However, we would have had no trouble proving $T_{\mathsf{linSearch}}(n) = O(n \lg(n))$ or $T_{\mathsf{linSearch}}(n) = O(n^2)$. Note that n, $n\lg(n)$ and n^2 differ by significantly more than a constant. So the following question arises: *How do we know which $f(n)$ is the "truth", at least up to a constant*? We know it cannot be $n\lg(n)$ or n^2, but how do we know it is not a sub-linear function of n? The answer is that if we can prove $T_{\mathsf{A}}(n) = O(f(n))$ and $T_{\mathsf{A}}(n) = \Omega(f(n))$ (as we did for linSearch), then we know that $f(n)$ is the true function (up to a constant) representing the rate-of-growth of $T_{\mathsf{A}}(n)$. We then say that $f(n)$ is the *asymptotic growth rate* of the (worst case) runtime of A. The true growth rate of the worst-case running time of linSearch was pretty obvious. However for other more interesting algorithms, we will only be sure we have a "tight" big-O bound on $T_{\mathsf{A}}(n)$ when we manage to prove a matching Ω bound, and hence a Θ bound.

This is perhaps the place to mention another fairly common error[7]. Suppose we have an algorithm consisting of some loops (possibly nested) from which there is no early exit and we produce an upper bound of, say, $O(n^2)$. This does *not* entitle us to claim that $\Omega(n^2)$ is also a lower bound. The 'reasoning' given for such a fallacious claim is that as there is no early exit the algorithm always does the same amount of work (true) and so the lower bound follows. This cannot possibly

[7]I am aware that quite a few such misconceptions have been pointed out in this note and there a danger of overdoing things. However each misconception discussed has occurred regularly over the years, e.g., in exam answers. The common thread is a failure to understand or apply the definitions rigorously. Naturally there will be many readers who will have grasped the essential points already; unfortunately there is no way to produce notes that modify themselves to the exact requirements of the reader!

be a correct inference: if $O(n^2)$ is an upper bound then so are $O(n^3)$, $O(n^4)$ etc. If the inference were correct it would entitle us to claim a lower bound $\Omega(n^d)$ for any $d \geq 2$! Clearly false. A genuine proof of a lower bound $\Omega(n^2)$ reassures us that in deriving the upper bound we did not over estimate things.

Note that for some algorithms, it is not possible to show matching $O(\cdot)$ and $\Omega(\cdot)$ bounds for $T_{\mathsf{A}}(n)$ (at least not for a smooth function f). This is not an inherent property of the algorithms, just a consequence of our limited understanding of some very complicated situations.

Typical asymptotic growth rates

Typical asymptotic growth rates of algorithms are $\Theta(n)$ (*linear*), $\Theta\big(n\lg(n)\big)$ (*n-log-n*), $\Theta(n^2)$ (*quadratic*), $\Theta(n^3)$ (*cubic*), and $\Theta(2^n)$ (*exponential*). The reason why $\lg(n)$ appears so often in runtimes is because of the very successful divide and conquer strategy (e.g., as in binarySearch). Most textbooks contain tables illustrating how these functions grow with n (e.g., [GT] pp.19–20), see also Figures 2.15, 2.16.

2.4 The Running Time of Insertion Sort

We now analyse the *insertion sort* algorithm that you are likely to have seen elsewhere. The pseudocode for insertion sort is given below as Algorithm 2.17. The algorithm works by inserting one element into the output array at a time. After every insertion, the algorithm maintains the invariant that the $A[0\dots j]$ subarray is sorted. Then $A[j+1]$ becomes the next element to be inserted into the (already sorted) array $A[1\dots j]$.

Let the input size n be the length of the input array A. Suppose the inner loop in lines 4–6 has to be iterated m times. One iteration requires time $\big(O(1)+O(1)+O(1)\big) = O(1)$. Thus the total time required for the m iterations is

$$mO(1) = O(m).$$

Since m can be no larger than n, it follows that $O(m) = O(n)$ (i.e., any function that is $O(m)$ is necessarily $O(n)$, but *not* conversely—remember that we read equality only from left to right when asymptotic notation is involved). The outer loop in lines 1–7 is iterated $n-1$ times. Each iteration requires time $O(1)+O(1)+O(1)+O(n)+O(1) = O(n)$ (lines 1, 2, 3, and 7 cost $O(1)$ and the inner loop costs $O(n)$). Thus the total time needed for the execution of the outer loop and thus for insertionSort is

$$(n-1)O(n) = nO(n) = O(n^2).$$

Therefore,

$$T_{\mathsf{insertionSort}}(n) = O(n^2). \tag{2.1}$$

Note, once again, how the use of asymptotic notation has helped us to avoid irrelevant detail (such as giving names to unknown constants, or adding up several constants only to observe that this is another constant).

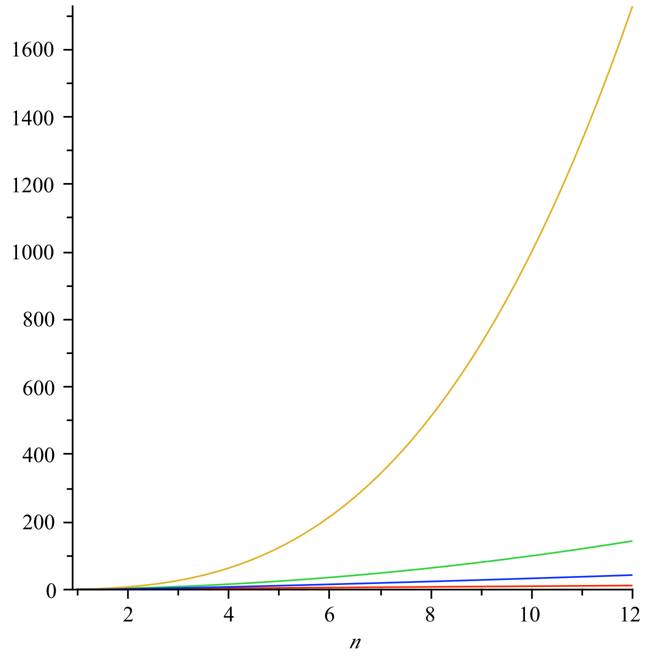

Figure 2.15. Graph of n, $n\lg(n)$, n^2 and n^3.

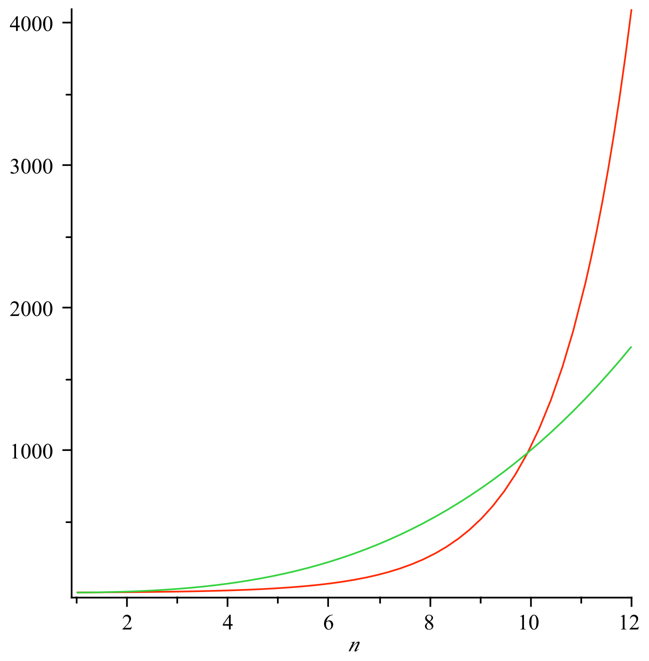

Figure 2.16. Graph of n^3 and 2^n.

```
Algorithm insertionSort(A)
  Input:  An integer array A
  Output: Array A sorted in non-decreasing order
1. for $j \leftarrow 1$ to $A.length - 1$ do
2.     $a \leftarrow A[j]$
3.     $i \leftarrow j - 1$
4.     while $i \geq 0$ and $A[i] > a$ do
5.         $A[i+1] \leftarrow A[i]$
6.         $i \leftarrow i - 1$
7.     $A[i+1] \leftarrow a$
```

Algorithm 2.17

This tells us that the *worst-case running time* of insertion sort is $O(n^2)$. But for all we know it could be better[8]. After all , it seems as though we were quite careless when we estimated the number m of iterations of the inner loop by n; in fact, m is at most j when the $A[j]$ element is being inserted.

We will prove that

$$T_{\text{insertionSort}}(n) \geq \frac{1}{2}n(n-1). \tag{2.2}$$

Since this is a statement about the *worst-case* running time, we only need to find *one* array A_n of size n for each n such that on input A_n, insertionSort does at least $\frac{1}{2}n(n-1)$ computation steps. We define this array as follows:

$$A_n = \langle n-1, n-2, \ldots, 0 \rangle.$$

Let us see how often line 5 is executed on input A_n. Suppose we are in the jth iteration of the outer loop. Since $A[j] = n-j-1$ is smaller than all the elements in $A[0\ldots j-1]$ (these are the elements $n-1, n-2, \ldots, n-j$, in sorted order) it follows that the insertion sort algorithm will have to walk down to the very start of $A[0\ldots j-1]$, so that line 5 is executed j times when inserting $A[j] = n-j-1$. Thus line 5 is executed j times for element $A[j]$. Thus overall line 5 is executed

$$\sum_{j=1}^{n-1} j = \frac{1}{2}n(n-1)$$

times. Clearly, each execution of line 5 requires at least 1 computation step. Thus (2.2) holds.

Finally $n(n-1)/2 = \Omega(n^2)$ and so our $O(n^2)$ upper bound was not an overestimate in terms of growth rate. We have shown that $T_{\text{insertionSort}}(n) = \Theta(n^2)$.

[8]There is nothing deep going on here. To return to a variant of an earlier example, if a person is 5 feet tall (approximately 1.5 metres) then it is correct to say that his/her height is no greater than $5, 6, 7, 8, \ldots$ feet. Each number gives correct but less and less accurate information.

It is worthwhile returning to the point made above about our estimation of the number m of iterations of the inner loop by n. How would we have a sense that the estimation is not so crude as to lead to too big an upper bound? Well we observed that m is at most j when the $A[j]$ element is being inserted and a little thought shows that there are inputs for which it is this bad (e.g., the input array above used for the lower bound). Now for $j \geq n/2$ we now know that $m = j = \Omega(n/2) = \Omega(n)$. As there are around $n/2$ values of j with $j \geq n/2$ we expect a runtime of around $n/2\,\Omega(n) = \Omega(n^2)$, so our estimation of j as $O(n)$ doesn't look so careless in terms of the asymptotic analysis. None of this is a precise proof but is the sort of intuition that comes naturally with enough practice and can be turned into a precise proof as shown above. Developing your intuition in this direction is important as it often helps to avoid getting bogged down in unnecessary detail.

2.5 Interpreting Asymptotic Analysis

In Lecture Note 1 we argued that determining the running time of an algorithm up to a constant factor (i.e., determining its asymptotic running time) is the best we can hope for.

Of course there is a potential problem in this approach. Suppose we have two algorithms A and B, and we find out that $T_{\mathsf{A}}(n) = \Theta(n)$ while $T_{\mathsf{B}}(n) = \Theta\big(n\lg(n)\big)$. We conclude that A is more efficient than B. However, this does not rule out that the actual running time of any implementation of A is $2^{1000}n$, whereas that of B may only be $100n\lg(n)$. This would mean that for any input that we will ever encounter in real life, B is much more efficient than A. This simply serves to underline that fact that in assessing an algorithm we need to study it along with the analyisis. This will usually give us a good idea if the analysis has swept any enormous constants under the carpet (they are not likely to be introduced by the implementation unless it is extremely inept; alas this cannot be ruled out!). Finally we can implement (carefully) any competing algorithms and experiment with them.

2.6 Further Reading

If you have [GT], you might have edition 3 or edition 4 (published August 2005). You should read all of the chapter "Analysis Tools" (especially the "Seven functions" and "Analysis of Algorithms" sections). This is as Chapter 3 in Ed. 3, Chapter 4 in Ed. 4.

If you have [CLRS], then you should read Chapter 3 on "Growth of Functions" (but ignore the material about the $o(\cdot)$ and $\omega(\cdot)$ functions).

Exercises

1. Determine which of the following statements are true or false. Justify your answer.

   (1) $n^3 = O(256n^3 + 12n^2 - n + 10000)$.

   (2) $n^2\lg(n) = \Theta(n^3)$.

   (3) $2^{\lfloor \lg(n) \rfloor} = \Theta(n)$.

2. In your *Learning and Data* lectures of Inf2B, you have seen (or soon will see) procedures for estimating the mean and variance of an unknown distribution, using samples from that distribution. What is the asymptotic running time of these mean-estimation and variance-estimation procedures?

3. Suppose we are comparing implementations of five different algorithms $A_1, \ldots, A_5$ for the same problem on the same machine. We find that on an input of size n, in the worst-case:

   - A_1 runs for $c_1 n^2 - c_2 n + c_3$ steps,
   - A_2 runs for $c_4 n^{1.5} + c_5$ steps,
   - A_3 runs for $c_6 n \lg(n^3) + c_7$ steps,
   - A_4 runs for $2^{\lfloor n/3 \rfloor} + c_8$ steps.
   - A_5 runs for $n^2 - c_9 n \lg^4(n) + c_{10}$ steps.

   Give a simple Θ-expression for the asymptotic running time of each algorithm and order the algorithms by increasing asymptotic running times. Indicate if two algorithms have the same asymptotic running time.

4. Determine the asymptotic running time of the sorting algorithm maxSort (Algorithm 2.18).

```
Algorithm maxSort(A)
  Input:  An integer array A
  Output: Array A sorted in non-decreasing order

1. for $j \leftarrow n-1$ downto 1 do
2.     $m \leftarrow 0$
3.     for $i = 1$ to $j$ do
4.         if $A[i] > A[m]$ then $m \leftarrow i$
5.     exchange $A[m], A[j]$
```

Algorithm 2.18

Can you say anything about the "best-case" function $B_{\mathsf{maxSort}}(n)$?

5. Use two appropriate theorems to prove the following.

   (1) For any constant $a > 0$ in $\mathbb{R}$: $f_1 = \Theta(g_1) \implies af_1 = \Theta(g_1)$.

   (2) $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2) \implies f_1 + f_2 = \Theta(g_1 + g_2)$.

   (3) $f_1 = \Theta(g_1)$ and $f_2 = \Theta(g_2) \implies f_1 f_2 = \Theta(g_1 g_2)$.

   (4) $f_1 = \Theta(g_1)$ and $g_1 = \Theta(g_2) \implies f_1 = \Theta(g_2)$.

Appendix A: Putting bounds on functions and runtimes

§**1. General setting.** This supplement discusses the process of putting upper and lower bounds from a slightly more general perspective. The material here is not new, it covers the same ground as the note but putting things this way might prove helpful to some. If you are already very clear about the issues reading this should be very quick, if this is not the case then draw your own conclusions!

Consider a function $F : \mathbb{N} \to \mathbb{R}$ defined by some means. Let's note in passing that the definition does not fix any method of computing $F(n)$ given n, it simply fixes a unique value by appropriate conditions[9]. We say that a function $U : \mathbb{N} \to \mathbb{R}$ is an *upper bound* for F if $F(n) \le U(n)$ for all n. Similarly a function $L : \mathbb{N} \to \mathbb{R}$ is a *lower bound* for F if $L(n) \le F(n)$ for all n. In many situations we don't mind if the inequalities fail to hold for some initial values as long as they hold for all large enough values of n (recall that this is the same as saying that there is an $n_0 \in \mathbb{N}$ such that the inequality holds for all $n \ge n_0$). So with this situation a lower and upper bound give us the information that

$$L(n) \le F(n) \le U(n) \tag{\dagger}$$

for all large enough values of n. Note that the definition of a lower bound is entirely symmetrical to that of an upper bound, the only difference is the inequality.

There can be various reasons for wanting to find upper and lower bounds, for us the main one is because although we have a precise definition of F we cannot obtain an exact formula for it. If we have upper and lower bounds we are of course interested in how good they are. To illustrate the point suppose we are seeking to find information about some integer M. After some work we find that $0 \le M \le 1000000$. This certainly gives us information but of a rather imprecise nature because the lower and the upper bounds are very far apart. If we worked a bit more and found that $12 \le M \le 20$ we'd be in a better position. In terms of functions we want $L(n)$ and $U(n)$ to be as close as possible. Of course the ideal is that $L(n) = U(n)$ in which case we know $F(n)$ but this is often not possible so we must settle for something less precise[10].

There are two further refinements we can make, we will consider the first one here and the second one later on. In this course we are interested in putting

[9]A frequent error is to talk about the runtime of a mathematical function. This is meaningless until we have chosen an algorithm to compute it and even then we are referring to the runtime of the algorithm. In any case it can be proved that for most functions there is no algorithm to compute them. (This is surprisingly easy given some fundamental notions.)

[10]In fact even when we know F explicitly we might be interested in bounding it from above and below by functions that are easier to work with and are still close enough to F.

bounds on functions whose values are non-negative and what is of interest is to bound the size of their values from above and below. So, e.g., if it so happens that (unknown to us) $F(n) = n$ it is not informative to produce the lower bound $-n^{10} \le F(n)$; anything negative is a lower bound. For this reason we amend (†) to

$$0 \le L(n) \le F(n) \le U(n) \tag{\ddagger}$$

for all large enough values of n. (An alternative is to take absolute values throughout but this is notationally heavier and is essentially equivalent to our approach.)

§**2. Putting bounds on runtimes.** Suppose we have an algorithm $\mathcal{A}$. Recall that we assume we have a method of assigning a size to the inputs of $\mathcal{A}$ such that for a given size there are only finitely many possible inputs. Now let R_n denote the set of runtimes that result by running $\mathcal{A}$ on all inputs of size n. (Of course this set depends on $\mathcal{A}$ as well but we have not indicated this in the notation just to keep it simple, no confusion can arise as we will only be considering $\mathcal{A}$ in the discussion.) Since there are only finitely many inputs to $\mathcal{A}$ for any given n it follows that R_n is a finite set and so it has a maximum and a minimum member. Recall that we made the following definitions in the course:

(1) The *worst case runtime* of $\mathcal{A}$ is the function $W : \mathbb{N} \to \mathbb{R}$ defined by

$$W(n) = \max R_n.$$

(2) The *best case runtime* of $\mathcal{A}$ is the function $B : \mathbb{N} \to \mathbb{R}$ defined by

$$B(n) = \min R_n.$$

(We are not terribly interested in the best case runtime except for illustrative purposes.) It is trivially the case that

$$0 \le B(n) \le W(n).$$

In other words $B(n)$ is a lower bound for $W(n)$ and of course $W(n)$ is an upper bound for $B(n)$. However these can be too far apart to be of much use. For example we saw that for linear search the best case runtime is a constant while the worst case is proportional to n. You should also consider the best and worst case runtimes of insertion sort, again they are far apart. Of course there are algorithms for which $B(n)$ is very close or even equal to $W(n)$ and for these cases it is a good lower bound to $W(n)$. The point is that this is not even remotely true of all of algorithms so we cannot use $B(n)$ as a good lower bound to $W(n)$ automatically.

Now pick any $r \in R_n$. By the definition of our two functions we have

$$B(n) \le r \le W(n).$$

Spelling this out, if we pick *any* input to $\mathcal{A}$ of size n and find the runtime then this is an upper bound for the best case runtime and also a lower bound for the worst case. Once again note the symmetry.

Let us now focus on $W(n)$ for some fixed n. By definition, an upper bound $U(n)$ to $W(n)$ is anything that satisfies

$$\max R_n \leq U(n).$$

Note that we just demand a bound on a *single* element of the set R_n but as this is the largest element it follows automatically that it is an upper bound to *all* elements of R_n. So when putting an upper bound we are in fact just concerned with one element of R_n but we don't know its value. So how can we put an upper bound on it? Well if we find that for *all* inputs of size n the runtime is at most $U(n)$ then of course this claim is also true of $\max R_n$, i.e., of $W(n)$. In practice we consider a general input of size n and argue from the pseudocode that the algorithm runtime will be at most a certain function of n (e.g., if we have a loop that is executed at most n times and the body costs at most a constant c then the cost of the entire loop is at most cn). This process leaves open the possibility that we have overestimated by a significant amount so to check this we look for lower bounds to $W(n)$.

Let us now consider putting a lower bound on $W(n)$. By definition a lower bound $L(n)$ to $W(n)$ is anything that satisfies

$$0 \leq L(n) \leq \max R_n.$$

Once again our interest is in putting a bound on a single element of R_n. We could do this by considering all inputs of size n but for most algorithms this would lead to a severe under estimate (recall linear search). As observed above, if we find the runtime r for any particular input of size n then we have found a lower bound to $\max R_n$, i.e., to $W(n)$. So when trying to put a lower bound we look at the structure of the algorithm and try to identify an input of size n that will make it do the *greatest* amount of work.

There is now an asymmetry in what we do. However this arises from the definition of the function which we seek to bound, *not* from the notion of upper and lower bounds.

At this point you should consider what it means to put upper and lower bounds on $B(n)$. It should be clear that in putting a lower bound we consider all inputs of size n but for an upper bound we need only consider a single appropriate input of size n. In this second case we look at the structure of the algorithm and and try to identify an input of size n that will make it do the *least* amount of work. Thus the situation is a mirror symmetry of that for $W(n)$.

§**3. Asymptotic notation.** We discussed above one refinement to the notion of bounds. For the second refinement we allow the possibility of adjusting the values of U and L by some strictly positive multiplicative constants, i.e., we focus on growth rates. This can be for various reasons, e.g., the definition of F involves unknown constants as in the case of runtimes of algorithms. So we amend the inequalities $(\ddagger)$ in the definition to allow the use of constants $c_1 > 0$ and $c_2 > 0$ s.t.

$$0 \leq c_1 L(n) \leq F(n) \leq c_2 U(n),$$

for all large enough values of n. Note that this says *exactly* the same as that $F = \Omega(L)$ and $F = O(U)$. Once again we ask just how good are such bounds?

The best we could hope for is that $L = U$ and if this is so then we say that $F = \Theta(U)$. Of course even if this is so we do not know the value of $F(n)$, for large enough n, because of the multiplicative constants but we do have a very good idea of the growth rate. For the final time, the definitions of O and Ω are entirely symmetrical, any asymmetry in arguments involving them comes from the nature of the functions being studied.

Appendix B: Proof by Induction

§**1. The most common version.** This supplement gives a brief description of a very useful and long established method of proof that we will rely on occasionally. In addition, you can find many discussions of the method on the web (but beware that some go into very exotic versions which we do not need for this course).

We often want to prove a statement about the natural numbers, examples include:

(1) $n < 2^n$ for all $n \geq 0$.

(2) $0 + 1 + \cdots + n = n(n+1)/2$ for all $n \geq 0$.

(3) If the function $f : \mathbb{N} \to \mathbb{N}$ is defined by

$$f(n) = \begin{cases} 0 & \text{for } n = 0; \\ 2f(n-1) + 1 & \text{for } n > 0, \end{cases}$$

then $f(n) = 2^n - 1$ for all $n \geq 0$.

In each case we have a statement that is parametrised by a natural number n and we often use a notation such as $P(n)$ to stand for the statement being made (this is just a matter of convenience so we don't have to write the whole thing out every time). Claiming that such a statement $P(n)$ is true is claiming that infinitely many things are true:

- $P(0)$ is true and
- $P(1)$ is true and
- $P(2)$ is true and . . .

Sometimes we make our claim for all natural numbers starting from some number n_0 onwards rather than 0. Wherever we start our claim, we call it the *base case*.

How can we prove our claim (assuming it is indeed true)? Clearly it is no good checking that $P(0)$ holds, then that $P(1)$ holds etc. Of course if any of these fail to hold then the claim is false (and typically we do a few checks) but if it is indeed true then we have infinitely many cases to check!

The task becomes possible by means of a simple and very powerful idea.

(1) Show that the base is true.

(2) Show that if the claim holds for n (where n is *not* fixed, it stands for any number whatever that is at least as large as the base case) then it necessarily holds for $n+1$.

These can be summarised as: (i) show that $P(0)$ holds (or $P(n_0)$ if we are not starting at 0) and (ii) show that $P(n) \Rightarrow P(n+1)$. The first task is referred to as the *base case* and the second as *the induction step*. Note that in the induction step we assume that $P(n)$ holds and prove that under this assumption $P(n+1)$ must also hold. The assumption referred to is called the *induction hypothesis*.

Establishing the induction step gives us the following guarantee. Let m be a fixed natural number that is at least as large as the base case. The induction step now tells us: "if you can prove (by whatever means) that the claim is true for $n = m$ then I guarantee that it is also true for $n = m+1$." So buy one get one free; in fact we get *much* more. Let's assume we have established both the base case and the induction step. Then we have the following sequence of facts:

- The claim holds for $n = 0$, this is the base case that we proved.
- Since the claim holds for $n = 0$ the induction step now shows us that the claim is also true for $n = 1$ (take $n = 0$ in the induction step proof).
- Since the claim holds for $n = 1$ the induction step now shows us that the claim is also true for $n = 2$ (take $n = 1$ in the induction step proof).
- Since the claim holds for $n = 2$ the induction step now shows us that the claim is also true for $n = 3$ (take $n = 2$ in the induction step proof).
- ...

In other words the claim holds for *all* natural numbers (from the base case onwards).

Let us now return to the three examples above and see induction at work.

(1) $n < 2^n$ for all $n \geq 0$ (this claim is what we referred to above as $P(n)$).

Our base case is $n = 0$ and so we must check that $0 < 2^0 = 1$ which is clearly true. Now for the induction step: we must show that if the claim holds for n then it also holds for $n+1$. So suppose $n < 2^n$; this is the induction hypothesis. We must show that from this assumption it follows that $n+1 < 2^{n+1}$; this is the induction step. Now if $n = 0$ then we must prove that $1 < 2$ which is clearly true (we don't need the induction hypothesis for this special case). If $n > 1$ then we have $n+1 \leq 2n$ (just subtract n form both sides). By the induction hypothesis we have that $n < 2^n$ and so $2n < 2 \cdot 2^n = 2^{n+1}$, so we have $n+1 \leq 2n < 2^{n+1}$ and hence $n+1 < 2^{n+1}$.

We did this example in the lectures but with the base case $n = 1$, i.e., we left out the claim for $n = 0$ (because we were applying it to deduce the inequality $\lg(n) < n$ and $\lg(0)$ is not defined).

(2) $0 + 1 + \cdots + n = n(n+1)/2$ for all $n \geq 0$.

For the base case we must check that $0 = 0(0+1)/2$ which is clearly true. so suppose the claim is true for n, we must show that it is then necessarily true for $n+1$. Now

$$\begin{aligned} 0+1+\cdots+n+(n+1) &= (0+1+\cdots+n)+(n+1) \\ &= n(n+1)/2+(n+1), \quad \text{by the induction hypothesis} \\ &= (n+1)(n/2+1) \\ &= (n+1)(n+2)/2. \end{aligned}$$

So if the summation formula holds for n then it also holds for $n+1$, which completes the proof of the induction step.

(3) If the function $f : \mathbb{N} \to \mathbb{N}$ is defined by

$$f(n) = \begin{cases} 0 & \text{for } n = 0; \\ 2f(n-1)+1 & \text{for } n > 0, \end{cases}$$

then $f(n) = 2^n - 1$ for all $n \geq 0$.

Here the base case is $f(0) = 2^0 - 1 = 1 - 1 = 0$ which is true from the definition of f. For the induction step we assume that $f(n) = 2^n - 1$ and show that then $f(n+1) = 2^{n+1} - 1$. From the definition of f we have $f(n+1) = 2f(n)+1$. By the induction hypothesis, $f(n) = 2^n - 1$ and so we have

$$\begin{aligned} f(n+1) &= 2f(n)+1 \\ &= 2(2^n-1)+1 \\ &= (2^{n+1}-2)+1 \\ &= 2^{n+1}-1. \end{aligned}$$

This completes the proof of the induction step.

Warning: The base case is usually easy to prove (often amounting to a simple check). Sometimes people new to induction overlook it altogether and focus on the more difficult induction step. However unless the base case does indeed hold then we cannot deduce anything about the truth of the claim. This should be clear from the discussion above. It is perfectly possible for the induction step to be true (i.e., as a logical implication $P(n) \Rightarrow P(n+1)$) even though the base is false and the claim is in fact false for all natural numbers. As an example consider the function f defined above and suppose we make the (false) claim that $f(n) = 2^{n+1} - 1$. The induction step assumes the claim for n and then for $n+1$ we have $f(n+1) = 2f(n)+1 = 2(2^{n+1}-1)+1 = 2^{n+2}-1$ which is the formula for $n+1$. Of course there is no foundation from which to deduce any actual fact about the values of $f(n)$, all we have established is that *if* $f(n) = 2^{n+1} - 1$ *then* $f(n+1) = 2^{n+2} - 1$ but the premise is just false!

A different pitfall is to establish the base case but make a subtle error in the induction step, such as using the induction hypothesis for a case to which it does not apply. For the uses of induction in this course you are unlikely to come across such pitfalls but be aware of them, the moral is that it is necessary to be careful about justifying the various steps.

§**2. Variants.** Sometimes we need to use more than one base case. In this situation we check all the base cases before going on to the induction step. In other situations, in establishing the induction step we need to use the assumption that the claim not only holds for n but for some (or possibly all) values up to n. This happens if, e.g., the truth of $P(n+1)$ depends not only on that of $P(n)$ but on $P(n-1)$ as well. This version is called *strong induction* but a little thought should convince you that it is really the same idea as that discussed above.

Finally sometimes our claim is not made for all natural numbers but for some infinite sequence, say $n_0, n_1, n_2, \ldots$, which we will call the relevant values (to the claim). We can, if we like, say that $P(0)$ is the claim for n_0, $P(1)$ is the claim for n_1 etc. In fact we can circumvent this by an induction step of the form: suppose the claim is true for some relevant value n then it is necessarily true for the next relevant value. The "strong" version assumes the claim is true for all relevant values up to n then it is shown to be necessarily true for the next relevant value.