

Informatics 2: Introduction to Algorithms and Data Structures

Notes on Lecture 3

Asymptotic notation: Little o and ω

John Longley

© University of Edinburgh 2026

In this lecture and the next, we'll be introducing the core concepts of *asymptotic analysis*, which will provide our framework for analysing algorithms and their efficiency throughout the entire course. In Lecture 5, we'll illustrate the use of the theory to say a variety of interesting and important things about algorithms, using INSERTSORT and MERGESORT from Lecture 2 as our main examples. These three lectures (3–5) are thus crucial to the whole course. Don't worry if you find the ideas difficult and unfamiliar at first—most people do! The ideas will get reinforced as the course proceeds, so you may find that if you revisit these lectures later in the course, things will seem clearer.

More specifically, asymptotic analysis centres around the use of the following notations, which we'll refer to as the **Gang of Five**:

o O Θ Ω ω

These are often read aloud as 'little o , big O , big Theta, big Omega, little omega'. As we'll see, there is a nice symmetry here: ω is the 'dual' of o , and Ω is the dual of O , so in some sense there are only three distinct concepts to get used to. In this lecture, we'll be getting to know o and ω ; in Lecture 4, we'll discuss O , Ω and Θ .¹

Why asymptotics?

The purpose of asymptotics within this course will be to give us a way of making mathematically precise, quantitative statements about the efficiency properties of *algorithms themselves*, as distinct from specific implementations of them. For instance, we'll be interested in questions such as the following:

- Can we nail down the precise sense in which MERGESORT is not just faster than INSERTSORT for sorting long lists . . . but vastly, *fundamentally* faster? (We'll see how o captures this idea.)

¹It's a somewhat distinctive feature of our presentation here that we're putting o and ω up front—most textbooks start with the other three, which are indeed the ones we'll use most commonly. One reason for this is that the idea of o can be explained quite naturally in terms of our observations from Lecture 2 on the runtimes of MERGESORT and INSERTSORT.

- We can imagine a lot of different implementations of MERGESORT (in different programming languages, running on different hardware, etc. etc.), and these will all have different runtimes. But can we say what it is that *all* of these runtimes will have in common? (We'll see that Θ is good for this.)

What are the benefits of the asymptotic approach? There are two main ones, both of which we'll see illustrated in this lecture:

1. It lets us say some fundamental and important things about algorithms in themselves that will apply to any implementation of them—in contrast to, say, the specific runtime ratios we listed at the end of Lecture 2, which might all change if we tweaked our Python code a bit or ran it on a different machine.
2. It simplifies things and saves us work, but without sacrificing mathematical precision and rigour. By concentrating on just the *broad headlines* of how algorithms behaves, we are freed up from doing a lot of messy and potentially difficult analysis that would be needed if we were after a precise runtime.

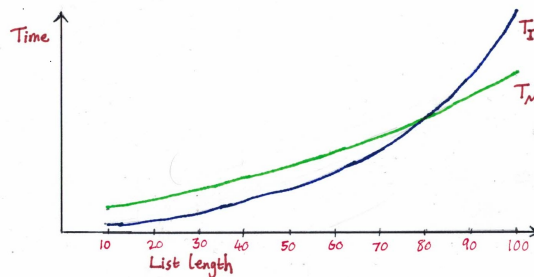
Introducing ‘little o ’: INSERTSORT and MERGESORT again

As a way in to the ideas, let's think again about our two sorting algorithms, and ask how we might compare their runtimes. Let's start by supposing we have some specific implementations of Insert Sort and Merge Sort, such as the Python programs I was talking about last time, running on my laptop. Broadly speaking, we're interested in how long these implementations take to sort lists of various lengths. In other words, we're interested in certain functions which we might call T_I and T_M , where $T_I(n)$ is the time taken (say in milliseconds) to sort a list of length n using INSERTSORT, and $T_M(n)$ is the time taken using MERGESORT. We're asking how $T_I(n)$ and $T_M(n)$ compare for various values of n .

But this immediately raises a question. *Which* list of length n are we talking about? After all, there are many different lists of length 100 (say), and the time taken to sort them with INSERTSORT (say) will vary wildly from one list to another. So we need to be clearer about what $T_I(n)$ and $T_M(n)$ actually refer to. Are we talking about the time it takes to sort a list of length n in the fastest possible case, or in the slowest possible case, or perhaps in some sort of ‘typical case’? All of these are things we might reasonably want to talk about—but for now, let's define $T_I(n)$ and $T_M(n)$ to be the *worst-case* runtimes for INSERTSORT and MERGESORT on list of length n —that is, the maximum possible times they might take on lists of this length.²

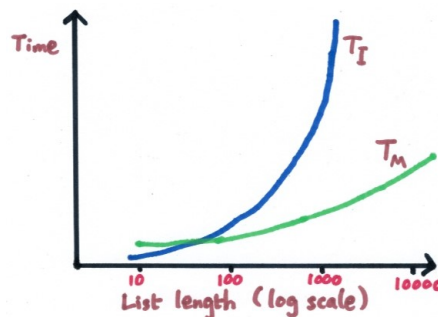
So that clarifies which functions T_I and T_M we're asking about. How might we compare them? As a first stab, we might imagine trying to plot graphs of these functions, showing how these worst-case runtimes increase with the length of the list:

²To nail things down even further, let's assume the lists we're talking about are lists of 32-bit integers—this will ensure that there really *is* a maximum runtime for any given list length n .



Like all the graphs in this lecture, this is just a rough sketch to illustrate the idea—but you’ll see that it’s broadly consistent with the figures at the end of the last lecture, which shows that my MERGESORT implementation takes longer than my INSERTSORT one on lists of length 10, but is faster on lists of length 100. (Further experiments with my implementations show that the crossing point is indeed somewhere around length 80, as the above curves suggest.)

This is fine as far as it goes—but the above graph only shows the behaviour for lengths 10–100, whereas we’re actually interested in anything from length 10 up to 1,000,000 or more. Obviously it wouldn’t be practical to show what’s going on here using an ordinary *linear* scale for the horizontal axis—so instead, let’s switch to a *logarithmic* scale, so that we can see how the runtimes look when we step up in powers of 10.³ For the ‘time’ axis, we’ll continue to use an ordinary linear scale. The picture might then look like something like this.



This is now starting to show what we remarked on last time. It’s not just that INSERTSORT ‘takes longer’ than MERGESORT for lists of a certain length. Rather, it takes vastly, spectacularly longer, by a proportional gap that just keeps on growing as our lists get longer and longer. In other words, $T_M(n)$ isn’t just less than $T_I(n)$ for $n \geq 100$; rather, $T_M(n)$ ultimately *pales into insignificance* compared to $T_I(n)$ —or to put it more positively, our MERGESORT program is *fundamentally* more efficient than our INSERTSORT one. Let’s now see how we can capture this idea mathematically.

We’ve actually already expressed in words what we want to capture: namely, that the proportional gap between $T_I(n)$ and $T_M(n)$ grows without limit as n gets larger. Indeed, the essence of what we’re trying to say is that the ratio

³The use of a log scale here isn’t a big deal and you don’t need to worry about it—especially since all our graphs are just schematic and illustrative anyway. But having a log scale will help a bit in the discussion that follows.

$T_M(n)/T_I(n)$ tends to zero as n gets larger. This is sometimes written as

$$T_M(n)/T_I(n) \longrightarrow 0 \text{ as } n \rightarrow \infty.$$

(We could equally well say that the ratio $T_I(n)/T_M(n)$ tends to infinity, but we'll stick with the above formulation here.)

Equivalently, we might say that the *limit* of this ratio as n goes to ∞ is zero:

$$\lim_{n \rightarrow \infty} T_M(n)/T_I(n) = 0.$$

But what exactly does *this* mean? We need to delve a little deeper here, because we're looking for a definition that tells us exactly what we'd need to show if we wanted to prove that a statement of this kind was true.⁴

One way to unpack this idea of 'tending to zero' is as follows: we're saying that the ratio $T_M(n)/T_I(n)$ gets 'as close to zero as we like': it will eventually become less than 0.1, less than 0.01, less than 0.001, ... For instance, taking 0.01 as an example, we're saying that once n is large enough, we have $T_M(n)/T_I(n) < 0.01$. A first stab at capturing this idea might be to write

$$\exists n. T_M(n)/T_I(n) < 0.01,$$

or in words,

there exists a number n such that $T_M(n)/T_I(n) < 0.01$.

However, just saying there's *one* such number n doesn't capture what we have in mind, because it doesn't rule out the (slightly bizarre) possibility that the ratio $T_M(n)/T_I(n)$ might go back to being somewhere around 1 when we look at even larger n . What we're really wanting to say is that once n is large enough, the ratio we're interested in falls below 0.01, *and stays below 0.01* as n increases further. In other words, once n has passed some 'threshold value'—let's call it N —it's guaranteed that the ratio will *always* be below 0.01.

The following formula gives a way of capturing this:

$$\exists N. \forall n \geq N. T_M(n)/T_I(n) < 0.01$$

or in words,

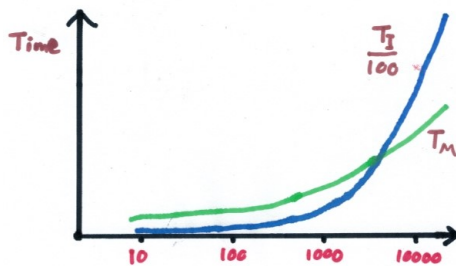
*there exists a threshold length N such that
 for every $n \geq N$ we have $T_M(n)/T_I(n) < 0.01$.*

Or here's another way to think of it. Let's mildly recast our formula using multiplication instead of division, by multiplying through by $T_I(n)$:

$$\exists N. \forall n \geq N. T_M(n) < 0.01 \cdot T_I(n)$$

This is saying that T_I doesn't just overtake T_M , but even if we tried to *squish down* our T_I curve by multiplying it by 0.01, it would still eventually break out and overtake T_M . This is illustrated by the following picture:

⁴At the moment, we're just trying to be crystal clear about what our 'limit' statement really means. Proving that it's actually *true* for typical implementations of INSERTSORT and MERGESORT is something we'll come to in Lecture 5.



Here you can see that if we took our threshold N to be 10000, say, it would be the case that $T_M(n) < 0.01 \cdot T_I(n)$ for every $n \geq N$ —assuming the trend suggested by our graph continues for even larger n , which in fact it does.

This is now getting close to what we want to say. For the final step, we simply note that we want to say that this is the situation for a squish-down factor of 0.01, but also for 0.001, 0.0001, $\dots$: in fact, for *any* positive real number c , however close to zero. (You should take a moment to imagine what would happen if we squished down the blue curve in the picture above even more, say by dividing T_I by 1000 or 10000. The idea is that it would then take longer for the blue curve to overtake the green one—the crossing point would move further and further to the right as we squished down more and more—but nonetheless, it would eventually do so.⁵) One way to think of this is that we’re not just saying MERGESORT is faster than INSERTSORT for long lists: we’re saying that even if we gave INSERTSORT an unfair advantage by running it on a machine 1000 or 10000 times as fast, or whatever, MERGESORT would still beat it

We can express this idea by the following formula:

$$\forall c > 0. \exists N. \forall n \geq N. T_M(n) < c \cdot T_I(n)$$

or in words by:

*for any positive real number c , however close to zero,
 there will be some threshold N (which will depend on c) such that
 for every $n \geq N$ we have $T_M(n) < c \cdot T_I(n)$.*

This now precisely captures the idea we’re after: the idea that the ratio T_M/T_I tends to 0 in the limit as n tends to infinity. What’s more, this formulation makes it clear exactly what we need to show in order to confirm that this is the case—how one goes about this is something we’ll come to shortly. The above formula thus expresses a crucial relationship between our two functions, for which we’ll introduce the notation:

$$T_M \text{ is } o(T_I),$$

or in words, T_M is *little-o* of T_I . Of course, we’ll also be interested in many other pairs of functions besides T_M and T_I —so more generally, given *any* two functions f, g , we shall say f is $o(g)$ (and say ‘ f is little- o of g ’) if a similar relationship holds between f and g .

Definition: f is $o(g)$ means $\forall c > 0. \exists N. \forall n \geq N. f(n) < c \cdot g(n)$.

⁵As an aside, a general piece of advice for understanding what’s going on in asymptotics is: draw lots of pictures! Sketching lots of graphs to illustrate schematically what’s going on is an excellent way to develop an intuition for the subject.

This is the central definition of this lecture, and it's one you need to understand and memorize. We can rephrase the statement ' f is $o(g)$ ' informally in various ways to remind ourselves what it means: e.g. ' f grows fundamentally more slowly than g ', or ' f is *asymptotically smaller* than g '. Here you can broadly understand the word 'asymptotically' here as meaning something like 'eventually', or 'ultimately', or 'in the limit'—although of course, the precise formulation above is the official definition of what we mean by it.

For clarification, we're supposing here that both f and g are functions from natural numbers $(0, 1, 2, \dots)$ to real numbers ≥ 0 (we can write this as $f, g : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0}$). Most often, our f and g will be functions that give estimates of the runtime of some algorithm on inputs of a given size n . We also take it that the variable c in the above discussion can stand for any *real* number ≥ 0 , while N and n range over *natural* numbers.

Some simple examples of o

Let's now see how we might actually show rigorously, for specific functions f and g , that f is $o(g)$ (or alternatively that this isn't the case). Ultimately, we have our sights on functions like T_I and T_M that give runtimes for various programs or algorithms. However, these are actually rather complicated functions to consider, so we'll start with some examples involving much simpler mathematical functions f and g , just to get used to the mechanics of the o definition. We'll come back to the runtimes for INSERTSORT and MERGESORT in Lecture 5.

Example 1. n^2 is $o(n^3)$: true or false?

For our first example, we'll ask whether f is $o(g)$, where $f(n) = n^2$ and $g(n) = n^3$ for all n . We can write this statement just as n^2 is $o(n^3)$. Informally, this is asking whether the function n^3 is fundamentally faster-growing than n^2 .⁶

Whenever we ask a question like this in an exam or tutorial sheet, there are three levels of answer we might be looking for:

1. We might just ask for a yes/no answer, without any justification: is this statement true or false?
2. We might ask for an *informal* justification of the answer, showing that you have a clear intuitive grasp of *why* the statement is true or false.
3. We might sometimes ask for a *formal* or *rigorous* justification based on the official definition of o . This will show that you have a crystal-clear understanding of what our statement $f = o(g)$ is saying.

⁶It's common mathematical practice to write an expression like n^3 to refer to a certain *function* from $\mathbb{N}$ to $\mathbb{R}$. Strictly speaking, this is a bit sloppy, because really n^3 is just a single number, whose value depends on the that of the variable n . A more precise way to refer to the function as a whole would be to use lambda notation: $\lambda n. n^3$ means the function that maps any given n to n^3 . In these notes, we'll typically follow the common practice, but it's important to bear in mind that o and its comrades actually express relationships between functions—complete curves on a graph if you like. It doesn't make any sense to ask whether n^2 is $o(n^3)$ for just a single value of n , say $n = 10$.

Notice that we're using f and g to refer to functions, so our notation ' f is $o(g)$ ' is strictly correct. In other texts, you may see this written as ' $f(n)$ is $o(g(n))$ ', which is the sloppy version, but in this case the sloppiness is easy to avoid.

We may ask a mix of questions at all of these three levels. Look out for the key words or phrases like *you need not justify...*, *informal*, *formal* or *rigorous*, as indicators of what we're wanting.

Let's consider our statement ' n^2 is $o(n^3)$ ' at each of these levels.

1. Is this statement true or false? Is n^3 'fundamentally faster-growing' than n^2 in the way we've been talking about. From thinking about these two functions, it's intuitively clear that the answer is a resounding **YES**.
2. For an informal justification, it's often a good idea to look at the ratio between our two functions, and see how this behaves when n gets large. In this case, we're looking at $f(n)/g(n) = n^2/n^3 = 1/n$, and we all know, at least informally, that $1/n$ tends to 0 as n tends to ∞ . This shows that we basically understand what's going on here.
3. For a rigorous justification, what we want to show is that our o formula says something true when $f(n)$ and $g(n)$ are instantiated to n^2 and n^3 respectively:

$$\forall c > 0. \exists N. \forall n \geq N. n^2 < cn^3.$$

This is the part that a lot of people find hard to master, so let's go slowly.

Whenever we see a complicated statement like this with lots of $\forall$ and $\exists$ symbols, it's quite helpful to think of it in the following way. The statement is saying that *whatever* small number c we're given—whether 0.01 or 0.001 or whatever—we can come with a number N that's large enough that $n^2 < cn^3$ for every $n \geq N$. Notice that we don't get to choose a value for c here: rather, we have to be able to cope with any c that's given to us. However, once c has been given, we *do* get to choose a value of N in response, which is allowed to depend on c . And we want to choose N in such a way that *whatever* number $n \geq N$ we're given—again, we don't get to choose n —it will turn out that $n^2 < cn^3$.

So it's useful to think of the meaning of our statement in terms of a dialogue or game between ourselves, who are trying to uphold the truth of the statement, and an imaginary opponent or demon who is trying to challenge the statement and shoot it down if they can. Here we suppose that the demon gets to choose the values of the ' $\forall$ ' variables, in this case c and n , while we control the ' $\exists$ ' variables, in this case just N . So a sample dialogue might look like this:

Demon: What about $c = 0.01$?
 Us: (To ourselves.) How large does n have to be to ensure that $n^2 < 0.01n^3$? We want the extra factor of n on the RHS to outweigh the effect of the 0.01.
 (Out loud.) OK, let's take $N = 101$.
 Demon: Er ... I'll choose $n = 120$.

Now we can do a simple calculation to see who has won the round:

$$n^2 = 14400, \quad cn^3 = 0.01 \cdot 1728000 = 17280, \quad \text{so } n^2 < cn^3.$$

So in this case *we win the round*, having successfully defended our statement against the demon's challenge.

Of course, even though the statement is true in this case, the demon might still win if we don't play sensibly. Here's another dialogue:

Demon: What about $c = 0.001$?
Us: Er ...let's take $N = 101$ again, that seemed to work last time.
Demon: Aha! I have you there. Let's now take $n = 120$.

Now we see that $n^2 = 14400$ again, but cn^3 comes out at just 1728, so $n^2 \geq cn^3$ and *the demon wins the round*. To defeat the demon in this case, we should have taken N to be 1001 or more.

When approaching questions like this, it's often a good idea first to play out a few rounds of such a dialogue, to get a feel for what we need to do in order to beat the demon. Indeed, we'd encourage you to spend some time playing with the Python program I've written, which lets you engage in such dialogues against the machine as a way of developing your intuition. However, to *prove* that our statement is true: we need something more than just a few specific dialogues: rather, we need to show that we can *always* win the game, whatever numbers the demon throws at us.

Having done a few examples, we can reflect on what we need to do in general in order to win. When c was 0.01, the reason we picked $N = 101$ was that we wanted $N > 100 = 1/c$. So suppose we now say that *in general*, whatever c we're given, we pick some N that's $> 1/c$. We then need to show that this choice will beat the demon. Notice that it's not enough just to show that $N^2 < cN^3$: we need to show that $n^2 < cn^3$ for *every* $n \geq N$. This can be shown by a bit of algebra, given that any $n \geq N$ will also be $> 1/c$:

$$cn^3 = cn.n^2 \geq cN.n^2 > c(1/c)n^2 = n^2.$$

To summarize, here's how you'd write out a rigorous proof that $n^2 = o(n^3)$, based on the formal definition.

We wish to show $\forall c > 0. \exists N. \forall n \geq N. \underline{n^2 < cn^3}$.

Given any $c > 0$, take any $N > 1/c$.

Then for any $n \geq N$, we have: $cn^3 = cn.n^2 \geq cN.n^2 > c(1/c)n^2 = n^2$.

This would get full marks in an exam. Notice that a proof of *any* o statement will follow this template: indeed, the only parts of this solution that are specific to the statement ' n^2 is $o(n^3)$ ' are the three parts I've underlined—the rest is 'boilerplate' which won't vary from one o proof to another, but which is important for showing the logical structure of the argument.

In most cases, the trickiest part is to see how N should be chosen (depending on c). When the solution is presented as above, it can seem as though we've just plucked a rabbit from a hat here—in practice, some rough working on the side may be needed to arrive at a suitable choice of N , perhaps by starting with a few specific cases (i.e. thinking about $c = 0.01$, then $c = 0.001$), then seeing what the general pattern is. In any case, once we've understood how to choose a suitable N , checking that it actually works is usually not so hard (a little bit of algebra in the above example).

Example 2. $100\sqrt{n}$ is $o(n)$: **true or false?**

Let's do another example (one needs to see a few in order to get the idea). This time, we asking whether $100\sqrt{n}$ is 'little- o ' of n : that is, whether $100\sqrt{n}$ is slower-growing in the long run than n . Once again, we'll consider our three levels of answer.

1. Is it true? For smallish values of n , we can see that actually $100\sqrt{n} > n$, owing to the boosting effect of that '100'. But it's clear from thinking about these functions that that '100' won't make a very significant difference in the long run. The answer, again, is **YES**, it's true that $100\sqrt{n}$ is asymptotically slower-growing than n .
2. How might we justify this informally? Once again, it's a good idea to look at the ratio: $(100\sqrt{n})/n = 100/\sqrt{n}$. And now it's clear that as n tends to ∞ , $\sqrt{n}$ will also tend to ∞ (albeit more slowly than n does), so $100/\sqrt{n}$ will tend to zero.
3. For a rigorous proof from the formal definition, we're wanting to show:

$$\forall c > 0. \exists N. \forall n \geq N. 100\sqrt{n} < cn .$$

Once again, let's start by thinking about a sample value of c , and how large N would need to be in each case. In fact, let's consider $c = 1$. When is it the case that $100\sqrt{n} < n$?

A bit of algebra can help us here. Dividing both sides by $\sqrt{n}$, we see that this is equivalent (for $n > 0$) to $100 < \sqrt{n}$; and squaring both sides gives us $10000 < n$. So if we take our $N = 10001$ (say), then for all $n \geq N$ we'll indeed have $100\sqrt{n} < n$.

The way we arrived at this suggests that we can do a similar bit of algebra for any value of c :

$$100\sqrt{n} < cn \Leftrightarrow 100 < c\sqrt{n} \Leftrightarrow 100/c < \sqrt{n} \Leftrightarrow (100/c)^2 < n .$$

So for any given c , as long as we take our N greater than $(100/c)^2 = 10000/c^2$, we should be fine. This kind of working backwards from what we're wanting our N to achieve is a common way at arriving at the N we want. Having gone through a process like this in rough working, we can now write up our polished proof like this:

We wish to show $\forall c > 0. \exists N. \forall n \geq N. 100\sqrt{n} < cn$.
 Given any $c > 0$, take any $N > 10000/c^2$.
 Then for any $n \geq N$, we have: $\sqrt{n} \geq \sqrt{N} > 100/c$, so
 $100\sqrt{n} = c(100/c)\sqrt{n} < c\sqrt{n}\sqrt{n} = cn$.

(Once again, I've underlined the parts that are specific to this problem.)

Example 3. $n + 1000000$ is $o(6n)$: **true or false?**

Our final example (for now) illustrates some slightly different points. Is it true that $n + 1000000$ is 'asymptotically smaller' than $6n$? Here it makes sense

to consider our level 1 and 2 answers together before looking at the rigorous justification.

Our first thought might be: obviously $n + 1000000$ is bigger than $6n$ for small values of n , but once n itself reaches 1000000 or so, the effect of the ‘+ 1000000’ will start to become negligible, and $6n$ will clearly take the lead and the gap will continue to widen. So we might at first think that $n + 1000000$ is $o(6n)$.

Is this correct? Let’s look again at what o means. A statement $f = o(g)$ is saying that the *proportional* gap between f and g gets wider without limit. So in this instance, we’re interested in the ratio $6n/(n + 1000000)$ rather than the straightforward difference $6n - (n + 1000000)$. And it’s clear that this ratio never rises above 6. Put another way: although $6n$ might seem to have the upper hand, this could be neutralized by applying a squish-down factor of, say, $c = 1/7$. Thinking now about $6n/7$, we can see that this will never break out and rise about $n + 1000000$, however large we take n to be. So the answer in this case is **NO**, and what we’ve just said amounts to a pretty good informal justification, which actually comes close to being a formal one.

The moral of this example is that when we’re doing asymptotics, *constant factors* (like 6 in this example) are considered as not mattering at all. We don’t think of $6n$ as growing *fundamentally* faster than n , say, because the difference can easily be adjusted for by a suitable squish-down factor. This may seem strange at first—after all, in practice, we’d certainly consider a runtime speedup by a factor of 6 to be significant. The point, however, is that we can use o to highlight even more essential differences between algorithms—differences that couldn’t be mitigated by some constant factor speedup. Just saying that MERGESORT is 6 times faster than INSERTSORT once our lists reach a certain length does begin to do justice to the fundamental difference between the two algorithms.

Let’s now look at this example from the perspective of the formal definition. The claim ‘ $n + 1000000$ is $o(6n)$ ’ would translate to the formal statement:

$$\forall c > 0. \exists N. \forall n \geq N. n + 1000000 < c \cdot 6n .$$

What we want to show is that this is *false*: in other words, *the demon can always win* the game for this one if he plays sensibly. Indeed, we’ve already noted that the demon is guaranteed to win if he plays $c = 1/7$. To confirm this, we need to show that there is *no* possible N we can pick that would lead to a win for us. But this is clear, since for any $n \in \mathbb{N}$ whatsoever, it’s the case that $n + 1000000 \geq 6n/7$.

One way to think about what we’re doing here is that we’re now wanting to prove the *negation* of the usual o formula. Using the symbol $\neg$ for ‘not’, we’re wanting to show:

$$\neg(\forall c > 0. \exists N. \forall n \geq N. n + 1000000 < c \cdot 6n) .$$

It’s a well-known piece of logic that we can transform this to an equivalent statement by pushing the negation inward, turning each $\forall$ into $\exists$ and *vice versa*. This is because e.g. saying it’s *not* the case that for all $c > 0$, some property P holds is the same as saying there must exist a $c > 0$ for which $\neg P$ holds, and similarly with $\forall$ and $\exists$ reversed. Applying this procedure to the above formula,

we end up with:

$$\exists c > 0. \forall N. \exists n \geq N. n + 1000000 \geq c \cdot 6n$$

as the thing we're wanting to show. In other words the roles of the demon and ourselves are now reversed: we get to choose c , then the demon chooses N , and finally we choose n . We can see that this is basically what we did above. So we can express the argument succinctly like this:

Consider $c = 1/7$. Then given any N , take $n = N$ (noting that $n \geq N$).
Then $n + 1000000 > n > 6n/7 = c \cdot 6n$.

Using o notation

We've talked about what our little- o notation means, and how we might go about showing that some function is (or isn't) little- o of another function. But how do we actually use this notation in practice, and why is it helpful?

This is a good point to mention some important conventions that we'll use throughout the course. First, let's think for a moment about what we mean when we write something like $o(g)$ (or $o(n^2)$, or whatever). Clearly, $o(g)$ doesn't itself denote a number, or any particular function: rather, it's a *property* that some functions f will possess while others don't. Mathematically, we might therefore say that $o(g)$ is, strictly speaking, a *set of functions*, consisting of those functions f that stand in a certain relationship to g . In symbols:

$$o(g) = \{f : \mathbb{N} \rightarrow \mathbb{R}_{\geq 0} \mid \forall c > 0. \exists N. \forall n \geq N. f(n) < c \cdot g(n)\}$$

For instance, $o(n^2)$ is a certain set of functions, containing for example the function $3n + 2$ (because we'd say $3n + 2$ is $o(n^2)$), but not containing the function n^2 itself (because we wouldn't say n^2 is $o(n^2)$: no function is slower-growing than itself). So from this point of view when we say ' f is $o(g)$ ', this would strictly speaking be expressed in mathematical notation by ' $f \in o(g)$ '.

That's the official story about what $o(g)$ technically 'is'. However, this use of notation isn't actually the most convenient in practice. The common convention is that whenever we write $o(g)$, we usually mean *some* function, which we're not bothering to specify precisely, which has the property of being asymptotically smaller than g . (That is, we allow ourselves to write $o(g)$ to stand for an arbitrary member of the set we're officially calling $o(g)$.) So according to this convention, we're allowed to write $f = o(g)$ to mean that f is little- o of g , because this is saying that f is equal to *some* function with this property. Likewise, we could write, for example, $f(n) = 3n^2 + o(n)$ to mean that $f(n)$ is $3n^2$ plus some term we're not bothering to specify exactly, except to say that it's slower growing than n when considered as a function of n . That's quite useful when we don't care particularly about the details of what this term is — when all we care about is that $f(n)$ is ' $3n^2$ plus *small change*'. So the notation is very handy for reducing clutter and simplifying calculations when all we care about are the broad headlines of how functions behave, not the exact quantities.

Obviously, this convention requires a bit of care, because it doesn't exactly fit with our usual way of thinking about '='. For example, we could say that $n = o(n^2)$, and that $2n = o(n^2)$. Both of these statements are true, but we obviously shouldn't infer from them that $n = 2n$ for all n ! So, the convention

may seem a little odd at first, and some care is needed, but one fairly quickly gets accustomed to it and comes to appreciate its power. Indeed, there are a lot of useful reasoning principles that are valid for this notation and which we can use in calculations. For example: we're allowed to say that $o(g) + o(g) = o(g)$: informally, if we have two quantities that both become negligible compared with g in the asymptotic limit, then their sum is also negligible compared with g in the same sense. (Indeed, if you're interested in the more mathematical side of things, it's a nice exercise to see if you can prove formally from the definition of o that if both f and f' are $o(g)$ then so is $f + f'$.)

It might seem that by writing things like $o(n)$, we are compromising precision and resorting to a loose, hand-waving style of argument: we are only saying 'roughly' how our functions behave. However, it's important to bear in mind that when we use asymptotic notation, everything we say still has a precise mathematical meaning—and if we use the notation correctly, every step of our arguments can be fully and rigorously justified. So we might say that the beauty of asymptotics is that it gives us a way of being *imprecise in a precise way*: it lets us talk about just the broad-brush, 'approximate' behaviour of the quantities we're interested in, but also without sacrificing mathematical rigour and precision. So we can safely keep track of the relative importance of the various terms in our calculation without getting bogged down in messy details, allowing us to focus on the most essential features of the situation. In this course, we'll be using asymptotics a lot to say something clear-cut about (mostly) the runtimes of various algorithms—often in situations where giving exact expressions for these runtimes would be messy at best, and impracticably difficult at worst.

Besides its simplifying power, the other important point about asymptotics is that it lets us say some quite *robust* things about the relationships between functions. In other words, if for particular functions f and g we've established that $f = o(g)$, this is likely to remain true if we tweak our f and g in various ways. For instance, for any reasonable f and g where $f = o(g)$, we could take $f'(n) = f(n) + 5$ and $g'(n) = 0.2 \cdot g(n)$, and it will still be true that $f' = o(g')$, because the definition of o itself allows us to absorb such minor variations without difficulty. So asymptotics lets us say things about functions that are likely to be stable under various kinds of tweaking.

This ties in with the idea that asymptotic statements can be expected to apply not just to a specific implementation of our algorithms, but to *any* reasonable implementation. Because a statement like $T_M = o(T_I)$ is robust with respect to minor variations in the functions concerned, we can expect this statement to hold quite generally: *any* implementation of MERGESORT, running on any machine, will eventually beat *any* implementation of INSERTSORT once the lists get long enough. It is in this sense that asymptotics gets to the heart of what can be said about the algorithms themselves, as distinct from particular implementations of them.

Little ω : the dual of little o

To end this lecture, let's briefly introduce another of our Gang of Five, namely ω (pronounced 'little omega'). This is now very easy and doesn't really involve any new ideas, because ω is simply the *dual* of o . Remember that $f = o(g)$ is a way of saying f is *asymptotically smaller* than g : in other words, however much

we scale g down by, it will still eventually overtake f . By contrast, we can read $f = \omega(g)$ as saying that f is *asymptotically larger* than g : however much we scale g up by, it will still be overtaken by f . The formal definition, too, is in a clear sense dual to that of o :

Definition: f is $\omega(g)$ means $\forall C > 0. \exists N. \forall n \geq N. f(n) > C \cdot g(n)$.

Here I've used an uppercase C for the scale-up factor, because the thought here is 'for any scale-up factor C , *however large*, ...', whereas in the case of o the thought was 'for any scale-down factor c , *however small*, ...'. Other than that, the only difference is that we've swapped ' $<$ ' for ' $>$ ' in ' $f(n) > C \cdot g(n)$ '.

The reason this doesn't really introduce anything new is that ' $f = \omega(g)$ ' is in fact precisely equivalent to ' $g = o(f)$ '. This isn't hard to see: if we're just interested in a proportional comparison of $f(n)$ and $g(n)$, then scaling g up by some (large) factor C will have the same effect as scaling f down by the (small) factor $c = 1/C$. (Again, if you're drawn to the mathematical side of things, you might like to prove rigorously from the definitions that $f = \omega(g)$ if and only if $g = o(f)$.)

So in one sense, we don't really 'need' both o and ω . But it's quite handy to have both symbols, in just the same way that it's useful to have both ' i ' and ' j ', even though in theory we could make do with just one of these. There's perhaps a subtle difference in emphasis between saying $f = \omega(g)$ and $g = o(f)$: in the first case, we're considering the function g as a 'point of reference' and saying that f grows asymptotically faster than it, while in the second, we're considering f as the reference point and saying that g grows asymptotically slower than it.

In this course, ω will in practice be the least often used of all our Gang of Five. But it's worth including for the sake of symmetry, and it's possible you may come across it in your reading. In any case, any question we might set involving ω can be translated, if you prefer, into a question involving o : for instance, asking whether $n = \omega(\log n)$ is the same as asking whether $\log n = o(n)$... and in this case, the answer is of course **YES**.