

Informatics 2: Introduction to Algorithms and Data Structures

Notes on Lecture 5

Asymptotics for INSERTSORT and MERGESORT

John Longley
© University of Edinburgh 2026

In the last two lectures, we've looked at the general concepts of asymptotics, and given a number of examples involving simple mathematical functions. In this lecture, we'll see how this theory can be applied to say things about the behaviour of algorithms—here we'll use our INSERTSORT and MERGESORT algorithms as examples, but we'll be applying similar ideas to other algorithms as the course proceeds.

Cost models

What sorts of things are we wanting to say about algorithms? Typically, we're wanting to talk about the 'cost' of executing an algorithm, for whatever notion of cost we might care about. Most often, as mentioned in the first lecture, we're interested in the *time* an algorithm takes to execute; but we might also be interested in the *memory space* it requires, or the number of disk accesses or network transaction, or even the total energy consumption. All of these are possible notions of *cost* we might choose to consider.

So it's clear that before we can analyse the 'cost' of an algorithm, we'll need to specify what kind of cost we have in mind, and how it is to be measured. And the definition we choose for the cost of computations is what is known as a *cost model*.

We'll begin, broadly speaking, by considering 'runtime' as our notion of cost. But even here, different models of runtime are possible, depending on the level of detail we want to go into. (Remember that we're trying to avoid getting sucked into the kinds of low-level detail that would be specific to a particular implementation running on a particular machine, since as far as possible we're wanting to say something about the algorithm itself.) For example, if we're thinking about sorting algorithms, we might try to estimate 'runtime' in the following ways.

1. If we're just wanting a fairly high-level, abstract view of what's going on, we might be content just to take at the *number of comparisons* between the items we're sorting—that is, the number of times our algorithm compare two of the items we're sorting to see which is the smaller. That will have the advantage of being relatively easy to analyse, and for sorting

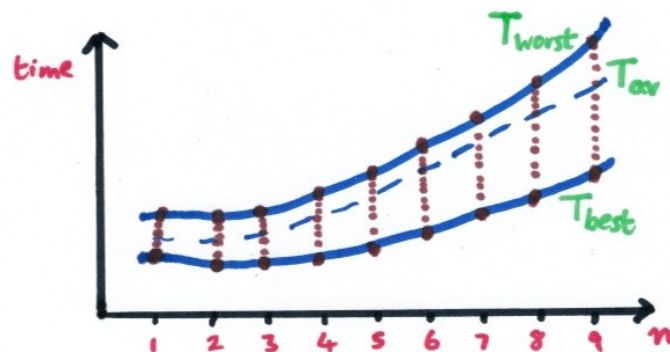
algorithms it will give us a pretty good overall idea of how much work the algorithm is doing—although this analysis won't take account of any 'thinking time' that the algorithm might be doing *between* doing those comparisons. Nor will it be sensitive to any variation in the time it might take to do the individual comparisons—we're just counting the number of comparisons and leaving it at that.

- Alternatively, if we're perhaps willing to work a bit harder, we might measure runtime in terms of some concept of the *number of basic steps* that we have to perform. Here we're thinking in terms of some perhaps slightly idealized machine on which we might be running our algorithm, and counting the number of computation steps on that machine. This means that to define our cost model, we'll need to say what our machine model is, and what we're willing to regard as a single basic step.

In this lecture, we'll start by analysing INSERTSORT and MERGESORT at the abstract level of 1 above, then move on to something more in the spirit of 2 to show what this looks like.

Worst, best and average cases

Let's suppose now that we've fixed on a definition of 'runtime' for sorting algorithms, which in the first instance will be simply number of comparisons. Broadly speaking, we're wanting to say something about the sorting time for lists of length n , as a function of n . However, we've already noted in Lecture 3 that different lists of the same length n may give quite different runtimes. This is illustrated by the following graph, with the list length n on the horizontal axis, and the 'runtime' on the vertical axis, and for each length n we have a whole spectrum of possible runtimes, schematically represented by a column of brown dots:



So there's no such thing as 'the' runtime for inputs of length n . However, for each n , we can ask what is the largest possible runtime for inputs of length n —this will be the *worst-case* runtime on lists of length n —and also what is the smallest possible runtime on such inputs—this will be the *best-case* runtime. This gives us two well-defined functions $T_{\text{worst}}(n)$ and $T_{\text{best}}(n)$ which we might hope to analyse in some way.

In many situations, it's also possible to define a notion of the *average* runtime for inputs of size n . That's particularly easy to do if there are only finitely

many possible inputs for each size n —or if there’s some finite set of inputs that we’re happy to take as representative of all of them—and if furthermore we’re happy to consider these inputs as being ‘equally likely’. In this case, we can simply take the average time over these finitely many representative inputs—the average time for the column of brown dots, in terms of the above picture. This will give us, for each n , a well-defined number which we can call $T_{\text{av}}(n)$, so we have another function T_{av} which we can ask about.

Notice that in the picture, the curve for T_{av} is drawn to be closer to T_{worst} than to T_{best} . This is to emphasize that $T_{\text{av}}(n)$ is *not* simply the average of $T_{\text{worst}}(n)$ and $T_{\text{best}}(n)$, but rather the average over *all* inputs of size n . Experience shows that usually—although not always—the average case time tends to resemble the worst case time more than the best case time. This seems to be just a sad fact of life.

So for a given sorting algorithm (and a given runtime cost model), we will obtain these three functions T_{worst} , T_{best} , T_{av} , and we can try to apply our theory of asymptotics to say how quickly they grow as n increases. In fact, there are a variety of things we might choose to say.

1. A typical thing we might wish to say is that $T_{\text{worst}}(n) = O(g)$ for some specific function g : for example, it might be that $T_{\text{worst}}(n) = O(n^2)$. What does this mean? This is saying that the T_{worst} curve falls below some multiple cg of g , at least once we’re to the right of some threshold value N . But if the worst-case time $T_{\text{worst}}(n)$ is below $cg(n)$, that tells us that *all* possible runtimes for length n inputs are below $cg(n)$ —that is, all of the brown dots will fall below cg , once we’re to the right of N . In other words, if the worst-case time is (eventually) bounded above by cg , then all possible runtimes will (eventually) be bounded above by that same cg . So in this situation, it’s reasonable to say that our algorithm *has runtime* $O(g)$ —or that the runtime is asymptotically bounded above by g . Even though, strictly speaking, there’s no such thing as ‘the’ runtime on inputs of size n , it’s perfectly reasonable to talk like this, because *all possible* runtimes are eventually bounded by cg .
2. Alternatively, we might say that $T_{\text{worst}}(n) = \Omega(g)$ for some function g . This would be saying that there’s some (small) multiple of g that (eventually) comes in below that T_{worst} curve. What does this mean in practice? Well, we’re saying that if we look at the particular inputs that give rise to these worst-case times—the worst possible input list for each size n —then the behaviour on those inputs is *essentially at least as bad as* g . So, slightly informally, we’re saying here that it is possible for the runtime to be ‘as bad as g ’. We’re not saying it’s always that bad—we’re just saying that there *are* inputs that manifest that bad behaviour. So in this case, we wouldn’t say without qualification that our algorithm *has runtime* $\Omega(g)$ —because not all possible runtimes need be this bad—rather, we’d say more specifically that the *worst-case* runtime was $\Omega(g)$. In any case, we can see that this expresses a quite different kind of property from $T_{\text{worst}}(n) = O(g)$.
3. What if we said that $T_{\text{worst}}(n) = \Theta(g)$? This would be saying *both* of the above things: the runtime is asymptotically bounded above by g , and furthermore there’s no hope of improving this statement further by

replacing g by something slower-growing, because the worst-case times show that the runtime *can* be as bad as g indicates.

If we're in the fortunate position of being able to make such a Θ -statement for T_{worst} , then we've precisely identified the asymptotic growth rate of this function. If we're not able to make such a statement, it may be that the best we can do (for the time being) is to give separate O - and Ω -bounds with a gap between them (e.g. T_{worst} is $O(n^2)$ and $\Omega(n \lg n)$). This leaves open the possibility that later, with further analysis, we may be able to improve on either or both of these bounds and perhaps close the gap.

4. Dual to all of the above, we can say similar things about T_{best} . For instance, if we say $T_{\text{best}}(n) = \Omega(g)$, we're saying that the best-case runtime, *and hence any possible runtime*, is eventually bounded below by some small multiple of g . Thus, in this situation it's perfectly reasonable to say 'the runtime is $\Omega(g)$ ', despite the fact that there's no one single runtime for each n .
5. By contrast, if we say $T_{\text{best}} = O(g)$, we're saying that it's possible for the runtime to be as good as g indicates: if we look at the the very best possible input for each length n , the runtimes on these inputs will grow no faster than g .
6. And finally, $T_{\text{best}} = \Theta(g)$ would be saying both of the above things: in this case, g gives us a tight asymptotic bound on the function T_{best} and so precisely identifies its asymptotic growth rate.

One can also make O , Ω or Θ statements about the function T_{av} if so desired. So the moral of all this is: be careful not to confuse the $O/\Omega/\Theta$ distinction with the $T_{\text{worst}}/T_{\text{best}}/T_{\text{av}}$ distinction. These are in principle quite independent factors: it's possible to apply any of these three relations about any of these three functions. (See also pages 48–49 of the CLRS book where they make a similar point.) It's sometimes commonly assumed that ' O is for worst-case, Ω is for best-case'—and while this is probably what these relations are most frequently used for, this by no means exhausts the range of things it's possible to say within our framework.

INSERTSORT: number of comparisons analysis

Here again is the pseudocode from Lecture 2 of the in-place version of INSERT-SORT (with line numbers added). Remember that the cells of A are indexed by $0, \dots, n-1$, so that in the outer for-loop, i runs from the second location to the last location in A .

```

0  InsertSort(A):
1      n = length(A)
2      for i = 1 to n-1
3          x = A[i]
4          j = i-1
5          while j ≥ 0 and A[j] > x
6              A[j+1] = A[j]
7              j = j-1
8          A[j+1] = x
    
```

Let's focus on the crucial expression ' $A[j] > x$ ' in line 5. This is the one point in the code where comparisons between our array items happen: we're comparing the item x that used to be at position i with the item that's currently at position j . So if we're trying to analyse the number of comparisons performed by INSERTSORT, we're interested in the number of times this particular expression gets invoked in the course of the algorithm.

Before we try to address this, it's worth pausing to note that our pseudocode here isn't very specific about the kinds of items we're trying to sort. In Lecture 2, we supposed for definiteness that the items stored in the array A were numbers, and that we were sorting them into numerical order: in other words, we were taking ' $>$ ' to mean the ordinary numerical $>$ relation. However, we can equally well use this very same algorithm to sort a list of strings into alphabetical order, provided we agree to interpret this ' $>$ ' as meaning alphabetical (or *lexicographic*) comparison between strings. In other words, we can use this algorithm to sort items of any kind that come with an ordering, as given by a comparison operation which we'll agree to write as ' $>$ '. Of course, if we're just thinking abstractly about the *number* of such comparisons that we're doing, it won't make any difference what kinds of things the items are.

Worst case

Let's now try to answer our question: how many times is our comparison operator invoked? In fact, the exact number will vary from one list to another—even for lists of the same length n —but let's first try to put an *upper bound* on the number of comparisons in terms of n .

Let's consider how our pseudocode works. The variable i runs from 1 up to $n - 1$, and for each value of i , we execute lines 3–8. So first, we can ask, for each individual value of i , how many times is the $>$ invoked in the course of executing these lines? comparison be invoked in the course of doing that? That's essentially the question of how many times we go round the j -loop in lines 5–7. This loop starts with $j = i - 1$, and j is decreased by 1 each time round. We quit the loop when j reaches -1 , *if not before*: we'll stop earlier if we find an item that's not $> x$. Finally, we see that when $j = -1$, the condition $j \geq 0$ on line 5 will evaluate to false, and we'll then immediately quit the loop without doing a further $>$ comparison. Thus, we see that the comparison will be invoked *at most i times*, as j is stepping down from $i - 1$ and no comparisons will happen for $j < 0$. This is for each individual value of i .

Now we recall that i itself is running from 1 up to $n - 1$. So the *total* number of comparisons will be, at most, the sum over all these i of the upper bound i that we've just established:

$$\text{Number of comparisons} \leq \sum_{i=1}^{n-1} i.$$

You may now remember that there is an exact formula for this summation: its value is $n(n - 1)/2$, which is quadratic in n . Or even if you didn't remember this, it's quite easy to see that $\sum_{i=1}^{n-1} i \leq \sum_{i=1}^{n-1} (n - 1) = (n - 1)^2$, so we can at least see that the summation is $O(n^2)$ (i.e. 'grows essentially no faster than n^2 '). So we can conclude that INSERTSORT *does $O(n^2)$ comparisons for input lists of length n* .

But we can also ask: Is the number of comparisons ever actually as bad as the above formula suggests? In other words, is the upper bound we've given actually *attained*? Is there, for each n , some input list of length n that will force the algorithm to do as many as $n(n-1)/2$ comparisons.

It's not hard to see that the answer is yes. Suppose that in our input array A , the items initially appear in *reverse order*, i.e. largest first. Then it's not too hard to see that for each i , the item initially at position i will be smaller than any of the items we've processed so far, so this item will have to work its way all the way to the left-hand end of the array. In other words, all the comparisons $A[j] > x$ we perform will return true, so the j -loop will go the full course, stopping only when $j = -1$. So for each value of i there will indeed be i comparisons, hence the total number of comparisons will be exactly $\sum_{i=1}^{n-1} i = n(n-1)/2$.

So in this case, we can say very specifically what the worst case looks like, and exactly how many comparisons there will be in that worst case. (This won't usually be the case: for a more complicated algorithm, it might be not at all easy to say precisely what the worst case is, but here we can.) In any case, the headlines we can now draw from this are that the worst-case number of comparisons for INSERTSORT is both $O(n^2)$ and $\Omega(n^2)$: it's never essentially worse than n^2 , but it can be that bad. Hence we can say that the worst-case number of comparisons is $\Theta(n^2)$, or of quadratic growth rate. So we've given a tight asymptotic bound for the worst case, and indeed in this case we've even been able to give an exact formula $n(n-1)/2$, though that's a bit unusual.

Best case

What about the *best case*? What's the smallest number of comparisons that a run of INSERTSORT could possibly perform on an input of length n ? Let's look again at the pseudocode that does the comparing. We see that for each individual value of i , when we enter the j -loop at line 5, we will have $j = i-1 \geq 0$, so at least one $>$ comparison will then get performed. So we can say that at least one comparison will be performed for each i , which means at least $n-1$ comparisons overall. Or if we're just interested in an asymptotic estimate: the best-case number of comparisons is $\Omega(n)$.

Once again, we can ask: is the bound we've given ever actually attained? In other words, is there, for each n , some input list of length n for which we really will do only $n-1$ comparisons? It's not hard to see that there is. Suppose, in particular, that our input list is *already sorted*. In this case, for each i , the initial comparison $A[j] > x$ which we do when we first enter the j -loop will of course return false, we will quit the j -loop straightaway and won't do any more comparisons for this value of i . This means we'll do just $n-1$ comparisons overall. In other words, if the input list is already sorted, the number of comparisons we need to do is only the bare minimum needed to check that it *is* already sorted, namely that each item is already larger than the one to the left of it.

So we've shown that the best-case number of comparisons is both $\Omega(n)$, because $n-1$ is a lower bound, and $O(n)$, because this lower bound is actually attained. So we can say that the best-case number of comparisons is $\Theta(n)$, or has linear growth rate. And of course, if we're wanting to say something now about the runtime on *arbitrary* inputs which needn't be either the worst or best case, we can say that the number of comparisons is $O(n^2)$ and $\Omega(n)$.

Average case

Let's also touch more briefly on what the *average case* number of comparisons looks like. First of all, how should we even define the average case number of comparisons performed by INSERTSORT?

Let's simplify just slightly and assume that we're sorting a list of n *distinct* items. Let's label them as x_0 up to x_{n-1} when they're listed in the correct (sorted) order. So the initial input array A will consist of some ordering or *permutation* of these n items: for instance, if $n = 2$, then A might be any of the following six lists.

x_0, x_1, x_2 x_0, x_2, x_1 x_1, x_0, x_2 x_1, x_2, x_0 x_2, x_0, x_1 x_2, x_1, x_0 ,

In general, there will be $n! = 1 \times 2 \times \dots \times n$ possible orderings for the input list. We shall suppose we're happy to assume that each of these $n!$ permutations is equally likely to occur as the input. For each of these input possibilities, we can ask about the number of comparisons made by INSERTSORT in order to bring the list into a sorted form. We can define the average-case number of comparisons $T_{\text{av}}(n)$ to be the average number of such comparisons over these $n!$ possible inputs.

We won't actually do the analysis here—it's not massively difficult, but would take another page or so of maths—that this $T_{\text{av}}(n)$ turns out to be close to $n^2/4$: more precisely, we can say that

$$T_{\text{av}}(n) = n^2/4 - O(n)$$

or $n^2/4$ minus some small change which is negligible by comparison. It's interesting to compare this with the *worst-case* estimate we obtained earlier, which is easily seen to be $n^2/2 - O(n)$. In the limit then, the number of comparisons in the average case is half that in the the worst case. This might seem intuitively quite plausible on the grounds that if we take an input list at random and look at any two items within it, they're equally likely to be in the right order or the wrong order.

In any case, from the point of view of asymptotics, this is enough to allow us to conclude that $T_{\text{av}}(n) = \Theta(n^2)$. So both the worst case and the average case are quadratic in n , whereas the best case is linear in n . This bears out the informal principle mentioned earlier that the average case tends to be more like the worst case than like the best case.¹

MERGESORT: number of comparisons analysis

Let's now try a similar analysis for MERGESORT. Recall that this was defined using an auxiliary procedure called Merge, whose purpose in life was to take two arrays B and C that were already sorted, and merge them into a single sorted array. This Merge procedure is, of course, where all the comparisons between items happen—so we'll want to start by looking at how many such comparisons Merge is performing.

The pseudocode for Merge looked like this:

¹We'll see an exception to this principle in later lectures: the randomized version of Quicksort has an average case that is more like the best case than the worst.

```
Merge (B,C):  
    allocate new array D of length |B| + |C|  
    i = j = 0  
    for k = 0 to |D|-1  
        if B[i] < C[j]  
            D[k] = B[i], i = i+1  
        else  
            D[k] = C[j], j = j+1  
    return D
```

The crucial expression, where all the comparisons happen, is ‘ $B[i] < C[j]$ ’. Remember that here we allowed the simplifying assumption that an out-of-range array access would return a dummy value ∞ , so some of these ‘comparisons’ will involve this dummy value rather than being genuine comparisons between lists. Obviously, we don’t want to count these dummy comparisons, which don’t really happen in practice, when we’re estimating the number of comparisons. Once we’ve used up all the items in either B or C, we’ll stop doing comparisons, and we’ll simply copy all the remaining items in the other array into our result array.

Comparison analysis for Merge

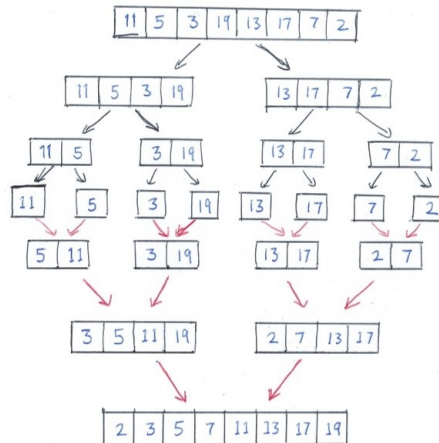
So how many genuine comparisons will happen when we merge B and C? It’s not hard to see that if m is the combined size of B and C (i.e. $m = |B| + |C|$), then there will be *at most* $m - 1$ comparisons. This is because we’re filling up our result array D which will have length m ; and each time we do a comparison, that will enable us to insert one more item into D (namely, the smaller of the two items we were comparing). But also we can note that the very last item we put into D won’t require a comparison at all: by that stage it’ll be the only item that’s left in either B or C, so there’ll be nothing that we need to compare it with. So we can see that Merge will never do more than $m - 1$ comparisons.

How about a lower bound? Well, we can see that as long as there are still items in both B and C that we haven’t processed, then we’ll need to keep on doing comparisons, so we won’t be able to stop until either B or C has been used up. So the minimum possible number of comparisons will be whichever is the smaller of the sizes of B and C. In general, that could be very small indeed: in the extreme (and silly) case when B is empty, we won’t need to do any comparisons at all! But we can also note that all the Merges that we do in the course of MERGESORT will have the property that the two arrays we’re merging are either of the same size, or else they differ by just one. Remember that MERGESORT repeatedly splits our list into two halves as evenly as we can, and that’s what gives rise to the sizes of the lists we’re merging. This means that the minimum size of B or C here will be either $m/2$ or $m/2 - 1/2$, depending on whether m is even or odd—so broadly speaking, we’re talking $m/2$ here.

So we can say that the number of comparisons done by a Merge within a MergeSort computation, although it may vary between these limits, is $\Theta(m)$ because it’s bounded above by $m - 1$, which is $O(m)$, and below by $m/2 - 1/2$, which is $\Omega(m)$.

MERGESORT: Worst case

How about MERGESORT itself? Here we'll reason slightly informally. Let's look again at the diagram of a complete MERGESORT computation:



Here we've made life slightly easy for ourselves by supposing that we're sorting a list of length 8, which is a power of 2. This means that the sizes of all the sublists involved are 1, 2 or 4, and the whole picture is nicely even and balanced. Recall too that although conceptually we're thinking about repeatedly splitting our lists and then re-combining them, the 'splitting' doesn't actually involve any significant work. All the real work, and certainly all the comparisons of items, happen in the bottom half of the picture where we do our repeated merging.

So what can we say about this? Well, we can see that this picture is nicely organized into different 'levels', and on each level, each of the original items appears once and only once, so the total size of each of these levels, or the sum of the sizes of all the lists on that level, is just our original n , in this case 8.

How many comparisons are performed by each of the Merges depicted here? We've just seen that the number of comparisons performed is less than the total number of items involved in the Merge—so less than 8 for the single merge in the last level, less than 4 for each of the two merges in the second-to-last level, and less than 2 for each of the four merges in the third-to-last level. So adding up the figures for all the merges on any given level, we can see that the total number of comparisons done by the merges at that level is less than the total number of items (8 in our case), as every item is involved in one and only one merge at each level. So, at most n merges in total for each level in the lower half of our picture.

How many levels are there? We can see in our picture that there are 3 levels of merging, which corresponds to the fact that 8 is 2^3 . So in general if our n is some number 2^k then the number of levels will be k , or in other words $\lg n$. And as we've seen, each of those levels involves $< n$ comparisons, so the *total* number of comps done by the whole computation will be $< n \lg n$. This gives our desired upper bound.

This is all quite nice and pleasant to think about when our n is a power of 2. But it's not too hard to see that broadly the same thing will apply even when it isn't. If n isn't a power of 2, then not all of our splits will be exactly even,

and this will mean that in some parts of this picture we'll need to go just one level deeper than in others (think about trying to mergesort a list of length 3). But with a little care, we can still argue that the total number of comparisons will be bounded above by $n\lceil\lg n\rceil$, where $\lceil\lg n\rceil$ means $\lg n$ rounded up to the nearest integer. That can be shown by a slightly more messy version of the argument I've just given; however, in Lecture 10 we'll be seeing a more general approach to managing this sort of situation fairly rigorously. In any case, we have an upper bound of $n\lceil\lg n\rceil$, and we can certainly say that this is $O(n\lg n)$.

MERGESORT: Best case

How about a lower bound on the number of comparisons? Can we say that the number of comparisons performed by MERGESORT on lists of length n will always be *at least* such-and-such?

Our analysis of Merge has shown that merging two sorted lists of combined length m requires at most $m - 1$ and at least $(m - 1)/2$ comparisons, so that's certainly $\Omega(m)$. And it turns out that, by going through an argument similar to the above and considering the summation over all the merges performed at all levels, we can show that even in the best case, the total number of comparisons will be $\Omega(n\lg n)$. Perhaps the easiest way to see it is that for any $m \in \mathbb{N}$, we have $(m - 1)/2 \geq m/3$, so that the number of comparisons for a merge of size m is always at least $m/3$. Summing over all the merges on a given level, whose total size is n , we have at least $n/3$ comparisons for the merges on that level; and because there are at least $\lceil\lg n\rceil$ levels, we have at least $n/3\lceil\lg n\rceil$ comparisons in total, and this is clearly $\Omega(n\lg n)$.

So we've shown that for lists of length n , all possible runtimes (as measured by number of comparisons) are sandwiched between two multiples of $n\lg n$. So in this case, it's quite legitimate to say 'the runtime is $\Theta(n\lg n)$ '. In fact, it's clear that all three functions of interest, T_{worst} , T_{best} and T_{av} will be sandwiched between these same two multiples of $n\lg n$, so without doing any further work, we can conclude that all these functions are $\Theta(n\lg n)$. And as an aside, it's worth noting that we've arrived at this estimate without going to the trouble of obtaining exact formulae for any of these three functions, which indeed might be quite tricky to get exactly right, with limited rewards. So the asymptotic viewpoint is definitely saving us some real work here.

Comparison analysis: Summary

We've seen that both in the worst case and on the average, the runtime of MERGESORT (measured by number of comparisons) has $\Theta(n\lg n)$ growth rate, whereas that of INSERTSORT has $\Theta(n^2)$ growth rate. To complete the picture, we may also note that

$$n\lg n = o(n^2),$$

which tells us that the worst and average case times for MERGESORT are fundamentally slower growing than those for INSERTSORT. We've therefore succeeded in explaining what we observed experimentally for the runtime figures we gave in Lecture 2.

The one saving grace for INSERTSORT is that it does better than MERGESORT in the best case: if our input list happens to be already sorted, then INSERTSORT has very little work to do and can complete in $\Theta(n)$ time, whereas

MERGESORT will have to go through the whole process of recursive splitting and merging regardless. If we're wanting to get the best of both worlds here, there are hybrid algorithms such as TIMSORT which are based broadly on MERGESORT, but are also able to detect if large portions of the input are already sorted (this being a common situation for real-world data). However, such algorithms tend to be a bit more complicated than the ones we've given here, and it's also fair to say that they are motivated by precisely the kind of analysis of the basic algorithms that we've just given.

Line execution analysis

Let's now look at what we might do if we're wanting a more detailed or comprehensive analysis of the overall 'runtime' of our algorithms, as opposed to just an estimate of the number of comparisons.

Suppose we have some piece of pseudocode such as the INSERTSORT code we gave earlier:

```
0  InsertSort(A):
1      n = length(A)
2      for i = 1 to n-1
3          x = A[i]
4          j = i-1
5          while j ≥ 0 and A[j] > x
6              A[j+1] = A[j]
7              j = j-1
8          A[j+1] = x
```

This time, for reasons that will become clear below, we shall specify that it's an array of *integers* that we're sorting: that is, the item comparison $A[j] > x$ performed on line 5 is a comparison between integers with the usual ordering.

There are just 8 lines of 'real code' here, but most of them will get executed several times in the course of a computation. And the main idea we wish to consider here is that if we're interested in accounting for all the work done by the algorithm, it's quite common to take *total number of line executions* as a measure of the runtime.

Let's just broadly think about why that's a reasonable thing to do, subject to certain qualifications. Think about how we might implement this pseudocode more concretely on an idealized computer, with memory consisting of words, and a bunch of registers, and instructions for reading from or writing to memory in a single step, and for performing arithmetical operations such as ' $-$ ' and ' $>$ ', and so on. So we're thinking of something very much like an actual machine.²

Given a machine implementation of this kind, we'd be able to talk about the time taken to run our code on various inputs. Of course, if it were literally true that every line execution here always took *exactly* the same length of time, then indeed number of line executions here would give a very precise measure of the runtime. In fact, we don't expect that to be exactly true: for instance,

²Strictly speaking, there's one significant idealization we wish to make. Since we're interested in the behaviour of our algorithm as n tends to infinity, we don't want fix on a particular word size such as 32 or 64 bits, but rather will allow ourselves to choose the word size, and indeed the number of words, to be as large as it needs to be to cope with the input we have in mind. However, this is a rather pedantic point and not really worth worrying over.

the simple array lookup here in line 3 would be likely to take less time than the evaluation of the more complex while-condition on line 5. But what *is* reasonable here is to suppose that each line execution here takes $\Theta(1)$ time. This means two things:

- Most importantly, there's a *maximum* possible time, or an upper bound on the time that any of these lines might ever take to execute. For example, we might be able to convince yourself by staring at this code that no line is ever going to need more than 10 machine-level instructions.
- As a more minor point, there's also a nonzero *minimum* possible time that any line might take—for instance, we could note that every line will require at least one machine instruction.

Thus, the execution time for a single line is bounded below by a positive constant c_1 and above by a constant c_2 . We can therefore say that the total execution time will be $\Theta(e)$ where e is the number of line executions, in that the total time will be between c_1e and c_2e . So from the point of view of asymptotics, where we don't care about the exact execution time but only about its growth rate, we can say that the number of line executions serves as a good model for runtime.

A warning is due here. The above ideas are only realistic when it really is true that every line execution is doing only a *bounded* amount of work—that is, when we can put some absolute bound c_2 on the execution time of single lines. In some situations, this may not be a reasonable assumption. For example, suppose that instead of sorting an array of integers, we used the above pseudocode to sort an array of *strings* into alphabetical order. We'd then be construing the ' $>$ ' here as the alphabetical comparison operation for strings. How long does this take to execute? If we're allowing strings of arbitrary length here, then there's no upper bound on the time it might take to do even a single comparison: for instance, if we have two strings of length 1000000 that differs only in the very last character, we'll need to look all the way down those strings in order to determine the alphabetical ordering. (While it was integers we were sorting, this wasn't a problem, because we had in mind that an integer would be stored in a single word, so that comparison of integers can be effected by a single machine instruction.) Thus, the moral is that number of line executions offers a good model of runtime as long as our pseudocode doesn't involve any lines that in practice could take an unbounded amount of time.

What do we find if we apply this 'number of line executions' analysis to INSERTSORT and MERGESORT. This is a case of looking at the pseudocode and applying the same kind of reasoning as before, except that we're now counting total number of line executions, not just number of invocations of the crucial ' $>$ ' operator. For example, in the above pseudocode, we see again that lines 3–8 are executed for each value of i , and that in each case go round the j -loop at most i times. So now, rather than simply saying the number of ' $>$ ' invocations will be at most i , we can say that the number of line executions will be at most $3i$ (this is for lines 5,6,7 which may be executed each time we go round the j -loop) plus 4 (this is for lines 2,3,4,8) which are executed one per value of i . (This $3i + 4$ is actually a slightly loose upper bound, because for the final value of j there will only be one line execution, but never mind). Summing over $i = 1, \dots, n - 1$, and adding 1 for the single execution of line 1,

we obtain an expression $1 + \sum_{i=1}^{n-1} (3i + 4)$ as an upper bound for the number of line executions, and this is clearly $O(n^2)$. So asymptotically speaking, this line execution analysis tells the same story as our comparison analysis: we obtain a runtime upper bound of $O(n^2)$ in each case. Indeed, it's not too hard to check that *all* the comparison analyses given above can be adapted to number of line executions and yield the same conclusions asymptotically:

	Worst	Average	Best
INSERTSORT	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$
MERGESORT	$\Theta(n \lg n)$	$\Theta(n \lg n)$	$\Theta(n \lg n)$

So in these cases, the picture isn't significantly altered when we take account of the work that the algorithms are doing in between comparisons. However, it's not guaranteed that this will always be the case in general: for instance, we may sometimes find that the 'thinking time' done by an algorithm in between comparisons is making a significant difference to the runtime. This is illustrated by the following.

Exercise: Devise a variant of INSERTSORT which, when given a *reverse-sorted* list of length n (i.e. one sorted in descending order), requires only $n - 1$ comparisons to detect that it *is* reverse-sorted, but requires $\Omega(n^2)$ line executions to actually sort it.

Space analysis

We conclude with a brief look at how similar analyses can be applied to talk about notions of cost other than runtime. In particular, let's look briefly at the *memory* requirements of our two sorting algorithms on inputs of length n .

A first observation is that if we're wanting to sort an input list of size n , it's taken for granted that we'll need enough memory space to store this input list. (For a list of n integers, this will pretty obviously be $\Theta(n)$ space: for instance, each entry in the list might occupy a single word in memory, and we might also have, say 4 words of header information at the start of the list.) Since this is something we can take as read, whatever sorting algorithm we're going to use, it seems sensible *not* to count the space occupied by the input as part of the space required by our algorithm—instead, let's say that we're only wishing to account for the additional space that the algorithm requires.

What do we find if we do this. Thinking intuitively, it's easy to convince ourselves of various things:

- Suppose we consider our original 'external' version of INSERTSORT: the one that take an input array A and puts the sorted result in a new array B. Then it's obvious that the main space requirement will be the space we need for that new array B, which will itself have size n . The only other space costs will be the (small) constant amount of space needed for the variables i , j and x . So it's pretty clear that the overall space requirement for this alg will be *linear in n* , that is, $\Theta(n)$.
- What about the *in-place* version of INSERTSORT (the one for which we gave the pseudocode above)? Here we don't need a new array B at all, since we're working entirely within the original array A. In fact, the *only* extra memory our algorithm will require is what's needed to store the

small handful of program variables that we're using, namely i , j , x . This will be just a small constant amount of space, independent of the input size n . So we can say that the extra space required for the in-place version of INSERTSORT is just $\Theta(1)$.

- What about MergeSort? The version we've been considering constructs a new target array D for each Merge performed, and all these arrays are shown in the lower half of our diagram of the recursive merge structure. This is quite costly on space: we can see that the total space for the target arrays on each level will be $\Theta(n)$, hence the total space for the whole computation is $\Theta(n \lg n)$.

This is actually quite wasteful. With a little more care, we can bring the space requirement for MERGESORT down to $\Theta(n)$, or linear space. In fact, we can make do with just our original array A of size n , and a single other array B of the same size. Roughly speaking, the idea is that we can alternate between merging two portions of A and putting the merged result in B , and merging two portions of B and putting the result in A , so that the data gets copied back and forth between A and B several times in the course of a mergesort. See for example problem 3 of Tutorial Sheet 2, in which this idea is worked out in more detail.

In summary: once we've found the space-efficient way to do MERGESORT, none of our approaches are too bad when it comes to space, but the in-place version of INSERTSORT is the clear winner in this respect.

This concludes the first part of the course, in which we have in some sense introduced all the main concepts and principles that we'll be using later. We've seen how we can express and analyse a variety of efficiency properties of algorithms, and we've developed some mathematical machinery to support this. Of course, there will be many more algorithms to come in later lectures, but the ways we'll be analysing them for efficiency will be broadly similar to what we've illustrated here for INSERTSORT and MERGESORT.