Informatics 2: Introduction to Algorithms and Data Structures
Notes on Lecture 2
Inefficient vs. efficient algorithms

John Longley

In this lecture we'll look at some specific examples of algorithms, to start to get a feel for the difference between 'inefficient' and 'efficient' approaches to the same problem. We'll illustrate the differences here with the help of some simple Python experiments, but in the next few lectures, we'll show how to give a more mathematical analysis of what's going on.

Here we'll look at three different problems and a selection of algorithms for each of them.

Problem 1: Remainders modulo 9

Our first problem is just a toy example, not a problem of any practical importance, so we'll only touch on it briefly. The problem is this: Given a large whole number n given in decimal notation, calculate the *remainder* when n is divided by 9 (this is often written in programming languages as n mod 9 or something similar).

The most 'obvious' way to approach this, which we'll call *Method A*, is simply to do the division sum the way you learned at school, and see what the remainder is. (An example was given in the lecture slides.)

But there's also a 'less obvious' way, which we'll call *Method B*. Given n, first add up the decimal digits of n to obtain a new number n'. If n' itself has more than one digit, add the digits to get a new number n''. Keep doing this until we get down to a single digit d. This digit will itself be the desired remainder n mod 9 — except in the case $d = 9$, when the desired remainder is 0. (Again, an example was worked through in the slides.)

Why does this work? In contrast to Method A, which works just 'by definition', Method B seems to require some explanation. We can see why it works by doing a little bit of algebra. We'll illustrate the idea in the case of a number with 4 decimal digits, written as $abcd$, but you can see that the same thing works for numbers of any length.

First note that when we write a 4-digit decimal number as $abcd$, the number we mean is actually the value of the expression $1000a + 100b + 10c + d$, because that's how decimal notation works. We can rewrite this as $(999 + 1)a + (99 + 1)b + (9 + 1)c + d$, then expand that to $999a + a + 99b + b + 9c + c + d$. Now the crucial step: if we're only interested in the remainder mod 9, we can throw

away those terms that are divisible by 9, and we're left with $a + b + c + d$. So our original number n, written as *abcd*, has the same remainder mod 9 as the new number n' obtained as the sum of its digits. This means that we can go through this process as many times as we like without changing the remainder mod 9, so that the resulting single digit d will be equivalent to n mod 9. And that's why this algorithm works.

This is only a toy example, but it's worth pausing to see what we can say about these two methods:

- We might note that Method B is 'faster' than Method A. Certainly, if you're trying to do it in your head, you'll probably find Method B is easier and quicker. That said, it's not *spectacularly* better: most of the work will go into summing the digits of the original number, which is broadly speaking 'one addition operation per digit' — whereas with Method A, we have to identify the relevant multiple of 9, then calculate the remainder to carry forward, so perhaps 'two operations per digit'.

- Method B allows for a bit more flexibility. With Method A, we have to do things in a fixed order, working from left to right through the given digit sequence. With Method B, however, we're free to 'add the digits' in whatever order we like, which might makes things easier (at least for humans).

- Related to this, Method B lends itself to being *parallelized* more easily than Method A. E.g. for a 16-digit number, we could say "You add up the first 8 digits, and meanwhile I'll add up the last 8" — and then we pool our results. Or in machine terms, you could give those tasks to two different processors, and that would be a way of getting the job done in roughly half the time.

So Method B is the 'less obvious' method, but it brings some potential advantages. And although this is perhaps an overly simple and artificial example, it illustrates a general point: by applying a bit of mathematical insight, it's often possible to do better than the most obvious algorithm that comes to mind. So the better we understand the maths, the better placed we are to come up with improved algorithms and to explain why they work.

Problem 2: Modular exponentiation

Our second problem is this: Given positive whole numbers a, n, m (which may be large — say a few hundred digits each), compute $a^n \mod m$. For example, if $a = 2, n = 10, m = 17$, we can compute this directly as

$$2^10 \mod 17 = 1024 \mod 17 = 4\,.$$

This might at first look like another artificial problem — but believe it or not, this computational task is absolutely central to the way many modern cryptographic systems work. In particular, there's a very famous public-key cryptosystem known as *RSA* (so called after its inventors Rivest, Shamir and Adleman), which you may learn about in the Discrete Maths and Probability course, and which relies crucially on being able to this 'modular exponentiation' calculation

efficiently. We won't go into how this cryptosystem works here, but will just focus on the problem of *how* to compute $a^n \mod m$ efficiently for large a, n, m (this will complement what you learn in DMP).

We'll look at three possible methods in turn. Once again, *Method A* will be just to literally compute what we're asked for: calculate a^n, then calculate its remainder modulo m. As we've just seen, this is fine when a, n, m are very small. But what if n had say 100 decimal digits? In this case — even if $a = 3$, say — the number a^n would be too large to fit into your computer's memory (whether represented in binary or in decimal). And even long before we reached that point, we'd find ourselves doing multiplications involving numbers with billions of digits, and even a single such multiplication would be very time-consuming. So there's absolutely no hope at all for Method A in practice.

Our *Method B* will solve at least one of the problems of Method A — that of the numbers getting too big to store in memory. This method works as follows:

```
Expmod_B(a, n, m):      # (where n ≥ 1, m > 1)
    b = a
    for i = 2 to n
        b = (b × a) mod m
    return b
```

That is, we start with a, then do $(n-1)$ multiplications by a, but after each stage we reduce mod m to bring the number back down to something that's smaller than m. This works because if we're wanting to multiply two numbers x, y but are only interested in the result modulo m, it makes no difference if we reduce x, y modulo m first:

$$(x \times y) \mod m \;=\; ((x \mod m) \times (y \mod m)) \mod m$$

This at least solves the 'space problem' with Method A: using Method B, the numbers we deal with will never get larger than $m \times a$. However, it doesn't solve the 'time problem': if n were a 100-digit number, then executing the for-loop in the above code would take much longer than the age of the universe.

Incomparably better than either of the above is *Method C*. Here, again, we make use of a small mathematical observation: if we're wanting to compute $a^n \mod m$ (let's call this number e), this is quite easy to do if we happen to have already computed $d = a^{\lfloor n/2 \rfloor} \mod m$, where $\lfloor n/2 \rfloor$ is $n/2$ rounded down. In the case where n is even, we have $a^n = (a^{n/2})^2$, so working modulo m we can just take $e = d^2 \mod m$. This is dramatically better than Method B which, once we have reached $a^{n/2} \mod m$, would continue with a further $n/2$ multiplications by a: here we're achieving the effect of all these multiplications in a single bound just by squaring. If n is odd, things are just a little more complicated: here we have $a^{n-1} = (a^{n/2})^2$, so we can compute $a^{n-1} \mod m$ as $d^2 \mod m$ — then we need just one more multiplication by a to get up to $a^n \mod m$. But still dramatically better than the $\lfloor n/2 \rfloor + 1$ further multiplications that Method B would perform.

Of course, this method presuppose that we already have our hands on $d = a^{\lfloor n/2 \rfloor} \mod m$. How would we obtain that? The key observation is that this is just another instance of exactly the same problem (with $\lfloor n/2 \rfloor$ in place of n), so we can recursively use the same method to compute d — assuming we have already obtained $c = a^{\lfloor n/4 \rfloor}$, and so on. Of course, this recursion had better

stop somewhere — but we can notice that if we're repeatedly halving our n and rounding down, we'll eventually reach the value 0, for which the problem is trivial: we always have $a^0 \mod m = 1$, regardless of a and m. (We're assuming $m > 1$ here, otherwise the problem is silly.) This situation can therefore serve as a *base case* for our recursion.

In summary, the algorithm we've just can be schematically represented as follows:

```
Expmod_C (a,n,m):              # Computes a^n mod m
    if n=0 then return 1
    else
        d = Expmod (a,⌊n/2⌋,m)
        if n is even
            return (d × d) mod m
        else return (d × d × a) mod m
```

This is an example of a *recursive* algorithm because you can see that **Expmod_C** 'calls itself' at a certain point: there's a kind of apparent circularity about this definition. You have probably come across the idea of recursive function definitions, for example in Haskell, and the idea does take some getting used to. We'll come back to this after a short interlude on another topic.

Remarks on pseudocode

The above presentation of Method C is an example of what is known as *pseudocode*, which is something we'll be using quite a bit in this course. You'll see that this looks quite like an actual program for computing **Expmod_C** — though it's not exactly a program in any actual programming language, and indeed I've allowed myself a bit of English text in places ('n is even'), and also used the mathematical notation $\lfloor n/2 \rfloor$ which is a bit different from how it would appear in an actual program. So the idea of pseudocode is that it's a rather free and informal mix of programming language constructs, mathematical notation and English, intended to be easily readable by humans rather than by machines. The purpose of pseudocode, then, is to communicate the idea of this algorithm to human readers at whatever level of detail we wish to describe it. If we want to give just a high-level outline of some algorithm, we can use short English phrases for each of the main tasks performed — or if we want to give a more detailed description, each of these phrases might be replaced by something closer to actual code. Once we have some pseudocode for an algorithm at a reasonably detailed level, it's normally a fairly routine task to translate it into a program in whatever language you like.

There are no hard-and-fast rules for writing pseudocode — pretty much any notation is allowed as long as it communicates clearly.[1] Perhaps the only guidelines we would ask you to stick to are the following:

1. Use *indentation* to show the structure of the code, including the extent of any for-loops or while-loops etc., just as you would in a Python program.

[1] You may notice that Mary and I tend to use slightly different notation conventions in our lectures, and we will be similarly easygoing about whatever conventions you use for pseudocode in an assignment or exam.

2. Avoid relying on features that are specific to particular languages. Your pseudocode should basically be readable by anyone with experience of programming in any language. As an example of a language-specific feature to avoid: in Python, the integer 0 can play the role of 'False', and any other integer can play the role of 'True', so that we could write

```
x = 0
...
if x: return z
```

   But not all languages let you do this, and for all I know there could be a language that uses the opposite convention. So spell out what you intend by writing 'if x $\neq$ 0 ...' rather than just 'if x ...'.

Understanding recursion

Let's now unpack in more detail how our algorithm for **Expmod_C** works, using the computation of 2^{10} mod 17 as an example. Because **Expmod_C** is a *recursive algorithm*, we can see that the computation will involve several different evaluations of, or *calls to*, the function **Expmod_C** — and it's perhaps helpful to imagine that each of these calls is managed by a different 'person'. We will begin by calling **Expmod_C** with $a = 2, n = 10, m = 17$, so let's suppose that this 'top-level' call is managed by person A. This means it's A's job to work through the above pseudocode with these values for a, n, m. First, A will test whether $n = 10$, and because it isn't, he will move on to the else-clause. At this point, because $\lfloor n/2 \rfloor = 5$, A will need to know the value of **Expmod_C**(2,5,17) so that he can assign this value to d. So A launches a new call to **Expmod_C** with $a = 2, n = 5, b = 17$, which we'll suppose is managed by B, and A won't be able to proceed further until he receives an answer from B.

Now B starts working through the **Expmod_C** code with her own copy of the program variables a, n, m and d, and in particular with $n = 5$. Again, B will test whether $n = 0$, and because it isn't, will proceed to the else clause. Now B needs to know the value of **Expmod_C**(2,2,17), and so launches a new call with $a = 2, n = 2, b = 17$ which we suppose is managed by C. Now both A and B are on pause in that they're waiting for the results of their respective calls.

In the same way, C will at some point require the value of **Expmod_C**(2,1,17) and so will launch a call managed by D; then D will require the value of **Expmod_C**(2,0,17) and will launch a call managed by E. Now something different happens: for E, the condition $n = 0$ will be true, so E is able to return the value 1 to D without any further recursive calls. Now D can resume her work: she assigns this return value 1 to her copy of d, then because she is working with $n = 1$ which is odd, she computes $(d \times d \times a) \mod m = (1 \times 1 \times 2) \mod 17 = 2$, and returns this value to C. Now C can continue: because he is working with $n = 2$ which is even, he computes $(d \times d) \mod m = (2 \times 2) \mod 17 = 4$, and returns this to B. And so on back along the line, until A is able to return the result of the top-level call:

Us to A:	What's **Expmod_C** (2,10,17) ?
A to B:	What's **Expmod_C** (2,5,17) ?
B to C:	What's **Expmod_C** (2,2,17) ?
C to D:	What's **Expmod_C** (2,1,17) ?
D to E:	What's **Expmod_C** (2,0,17)?
E to D:	1
D to C:	$1 \times 1 \times 2 \mod 17 = 2$
C to B:	$2 \times 2 \mod 17 = 4$
B to A:	$4 \times 4 \times 2 \mod 17 = 15$
A to us:	$15 \times 15 \mod 17 = 4$.

Here I've used indentation to align each question to its answer. This is a fairly typical example of how recursion work: we end up with multiple calls to a function or procedure *nested* one within another, like Russian dolls. Pretty much every programming language lets us write recursive calls like this.

It's quite normal to find this recursive style a bit mind-mangling at first. Part of the problem, perhaps, is that if we write a program in this style, quite a lot needs to happen 'under the hood' in order for it to execute: the machine must correctly keep track of which calls to our function are in progress, where we're up to in each of them, and what the associated values of the variables are for each call. Not that we as programmers need to explicitly manage all of this — it's all done behind the scenes by the interpreter or compiled code. So, on the one hand, we might (initially) feel less 'in control' when writing a recursive program than when writing one in iterative style (e.g. using for-loops or while-loops), because we need to trust that the underlying machinery is working correctly. On the other hand, because so much bookkeeping is done without our needing to code for it explicitly, it's often possible to express algorithms very elegantly and concisely in a recursive style.

Runtime comparison

Remarkably, in contrast to Methods A and B, the above algorithm remains feasible even when a, n, m are very large — say around 1000 digits? Why is this? Let's think about roughly how many nested calls to **Expmod_C** there will be. We can see that this will depend on the value of n. Each time we do a recursive call, the value of n gets halved and rounded down — so the question is: for a given starting value of n, how many times must we halve it and round down in order to reach zero? If n happens to be a power of 2, say 2^k, it's easy to see that the answer will be $k + 1$ — and indeed, it's not to hard to see that this will also be the answer if n is anything from 2^k up to (but not including) 2^{k+1}. So in general, the number of recursive calls will be approximately the *logarithm* of n to base 2 — that is, the number r such that $n = 2^r$ — which we'll write in this course as $\lg n$. For example, if n is of the order of 10^{1000}, then $\log_{10} n \approx 1000$ and hence $\lg n = \log_2 n \approx 1000 * \log_2 10 \approx 3300$. And on a modern computer, it's perfectly feasible to do 3300 of these nested calls very quickly, whereas it wouldn't be remotely feasible to do the 10^{1000} multiplications that Method B would require.

In the lecture slides, I illustrated this with some Python experiments showing the relative runtimes of Methods A, B and C (where feasible). These show that for small values of n (say around 100), Method A is actually the fastest, owing

to the fact that here the bulk of the work is being done by Python's built-in exponentiation operator (for computing a^n), and this is evidently pretty efficient. However, for n around 1000, Method C is fastest, and we see how Method C puts the other two completely in the shade as n increases towards 100000000. Even for n of the order of 10^{100}, Method C is still taking less than a millisecond, whereas the other two methods would take much longer than the current age of the universe to run. That's the difference that a good choice of algorithm can make!

It's pretty clear here that it's not just that my particular Python program for Method C happens to be better than my particular Python program for A or B. Rather, it's that the *method itself* — the algorithm we're using — is vastly, fundamentally superior. Indeed, the impression we get from our experiments is that *any* program implementing Method C is eventually going to out-compete any implementation of A or B — in any language, on any machine. So we're really looking here at a fundamental property of the algorithm rather than of one specific program. And this raises the question: how can we make mathematically precise what it is about Algorithm C that makes it so much better than A or B? That's the kind of thing we'll be looking at in Lecture 3 when we discuss the theory of asymptotic analysis.

Application to primality testing

Before moving on, it's worth glancing at a surprising trick we can pull off with our fast modular exponentiation algorithm: it gives us a simple, efficient and nearly infallible test for whether a large number n is a prime number!

The key to this is the following result, which features in the Discrete Maths and Probability course:

Fermat's little theorem: If n is prime and $0 < a < n$, then $a^{n-1} \bmod n = 1$.

We won't go into the proof of this here, but we can easily check that it works for small numbers: e.g. if $n = 7$ and $a = 2$, we have $2^6 \bmod 7 = 64 \bmod 7 = 1$.

Notice that $a^{n-1} \bmod n$ is exactly the kind of thing our algorithm is able to compute efficiently. So given a large number n, we could compute (say) $2^{n-1} \bmod n$, and see whether the result is 1. If it's not, then by Fermat's theorem we can conclude with certainty that n *isn't* prime — even if we haven't a clue what any of its factors are. If the result *is* 1, this means that n is very likely to be prime. This isn't absolutely certain, because there are a sprinkling of numbers that masquerade as primes under this test (e.g. $2^{340} \bmod 341 = 1$, but $341 = 11 \times 31$) — such numbers are known as *pseudoprimes* or *Fermat liars*. However, these exceptions are rare, and for a randomly generated n of 100 decimal digits or more, the chance that we've hit on a pseudoprime is pretty much negligible in practice. So we have an extremely fast and virtually infallible way of testing whether numbers of 100 to 1000 digits are primes.

In the lecture slides, I illustrate this with one of the so-called *RSA challenge numbers* which have been proposed as a challenge for factorization algorithms. The example I give is a 270-digit number which can be shown not to be prime by 6 lines of Python code taking 5 milliseconds to run — so there must *be* some factors — but no one in the world has yet succeeded in finding any!

Again, this and related primality testing algorithms play an important role in cryptography, where the ability to find large primes easily is of real practical importance.

Problem 3: Sorting a list

Now let's turn to one of the most fundamental problems in computer science: that of efficiently sorting a list of items according to some given ordering. For definiteness, we'll focus on sorting a list of integers into ascending order, though almost everything we say (with minor qualifications) will apply equally to e.g. sorting a list of strings into alphabetical order. Once again, we'll start with a reasonably 'obvious' method for doing this, then move on to a less obvious method that is far superior in most cases. Other algorithms for sorting will crop up later in the course.

Our 'obvious' method is known as InsertSort. Given an array A of length n, the idea is simply to work through A from left to right, sorting increasingly long segments of A by inserting each new element at its correct position into the existing sorted segment. This will usually involve shuffling some of the already sorted items one place to the right to make room for the new element. One can easily apply this idea to sort the contents of A putting the result in another array B of length n; but it is more interesting (and obviously more space-efficient) to do the sorting *in place*, so that the sorted list ends up in A itself.[2]

The following pseudocode captures this idea. Recall that the cells of A are indexed by $0, \ldots, n-1$, so that in the outer for-loop, i runs from the second location to the last location in A. The variable x is used to hold the current element which we're trying to place in its correct position, and the variable j works its way leftward from position i until the correct position is found, shuffling array items to the right as it does so.

```
InsertSort(A):
    n = length(A)
    for i = 1 to n−1
        x = A[i]
        j = i−1
        while j ≥ 0 and A[j] > x
            A[j+1] = A[j]
            j = j−1
        A[j+1] = x
```

You should study this pseudocode until you are satisfied that it works. Note that in this case, our 'pseudocode' is very close to being actual code — though this won't always be the case.

Our second, less obvious method is known as MergeSort. Our starting point here is similar to the idea behind our fast modular exponentiation algorithm, where we noticed that solving our problem for n would be easy if only we could first solve it for $\lfloor n/2 \rfloor$. In the case of arrays, the corresponding observation is that sorting an array A of length n would be fairly easy if we could first sort the two halves of A separately (these halves will have lengths $\lfloor n/2 \rfloor$

[2] This of course means we lose the original ordering of the items in A — if we wanted to retain this, sorting into a new array B would be better.

and $\lceil n/2 \rceil$ respectively). This is because if we're given two *already sorted* lists, it's not too hard to *merge* them into a single sorted list.

Below we give some pseudocode that takes as input two already sorted arrays B and C (not necessarily of the same length), and merges their contents into a new sorted array D. To do this, we simply work through our arrays from left to right, filling up D as we go. The key observation is at each stage, there are only two possibilities for the next item to put into D (i.e. the smallest remaining item): it will either be the smallest remaining item in B (i.e. the next item in B, since B is already sorted), or the smallest remaining (i.e. next) item in C. So only a single comparison between these two items is necessary to be able to insert an item into D. (The process was illustrated by an example in the live lectures and pre-recorded videos.)

```
Merge (B,C):
    # merges already sorted lists B,C
  allocate new array D of length |B| + |C|
  i = j = 0
  for k = 0 to |D|−1
    if B[i] < C[j]      # Convention: B[i] or C[j] is ∞ if index out of range
      D[k] = B[i], i = i+1
    else
      D[k] = C[j], j = j+1
  return D
```

This is an example some pseudocode that doesn't quite have all the details we'd need in actual code. The idea is that the variables i, j record the position of the 'next items' in B and C respectively. But what happens if all the items in B (say) have already been copied into D? In this case, the value of i will be the length of B, so the attempt to access B[i] would trigger an *array index out of range* error. In actual code, we'd need to write a few more lines to handle the case where we've reached the end of B and just need to copy any remaining items from C into D — or *vice versa*. In the above pseudocode, however, I've cheated a bit by saying 'let's pretend an out-of-range array access returns the special value ∞', which we suppose is 'greater' than any genuine item. Then if B[i] is a genuine item of B but C[j] $= \infty$, our pseudocode will copy B[i] into D, which is what we want; likewise, if B[i] $= \infty$ but C[j] is a genuine item, we'll copy C[j] into D. [To ponder, why couldn't B[i] and C[j] *both* be ∞.] So our pseudocode is correct, but I've chosen here to use a simplifying convention for the purpose of conveying just the *main idea* of the algorithm without unnecessary clutter. You could of course write pseudocode at a more detailed level if you wanted, and it would then look pretty much like actual code. The point is that *you can choose* how much detail to include in your pseudocode, according to what you're wanting to convey or emphasize.

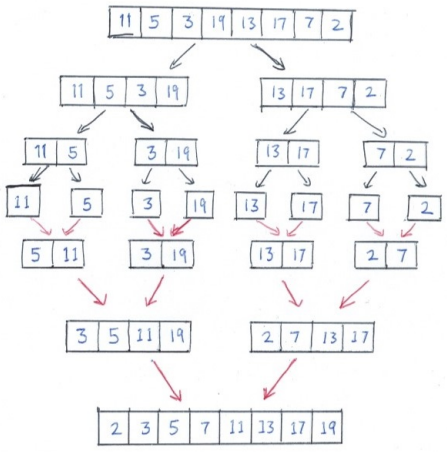
This shows us how to merge two halves of an array if each half has already been sorted. But how do we sort the two halves? Here, once again, the idea is to apply the very same method *recursively* to sort each of the halves — in each case, this will mean merging two sorted arrays of length roughly $n/4$ to obtain one of length $n/2$, which is then ready for the top-level merge. So our recursion will keep breaking down our arrays into roughly equal halves, until we reach portions of length 1, which of course don't require any sorting.

All of this is illustrated by the following pseudocode. The way I've organized this is to define a *helper function* **MergeSort** (A,m,n) which sorts the portion of A from position m to position n−1 (inclusive), returning the result in a newly created array D. This helper function is where the recursion happens: if the portion in question has length > 1, we calculate the position p of the approximate midpoint of this portion, then recursively call the helper function on each half separately. Then the actual merging uses the function **Merge** defined above. Of course, at the end of the day, we're interested in sorting the whole of A, not just some portion of it; but we can do this by calling our helper function with m=0 and n=|A| (this is a notation for the length of A).

```
MergeSort (A,m,n):      # sorts A[m], A[m+1], ..., A[n−1]
    if n−m = 1
        return [ A[m] ]
    else
        p = ⌊(m+n)/2⌋
        B = MergeSort (A,m,p)
        C = MergeSort (A,p,n)
        D = Merge (B,C)
        return D

MergeSortAll (A):
    return MergeSort (A,0,|A|)
```

Once again, you should study this pseudocode until you fully understand how it works. It's helpful here to visualize the recursive splitting and merging using a picture like the following, which illustrates MergeSort on an array of length 8. Notice that all the actual merges happen in the lower half of the picture — the upper half just illustrates the conceptual splitting that's represented by the various recursive calls.[3]



[3]We note in passing that this particular version of MergeSort isn't particularly good from the point of view of *space efficiency*: it creates a fresh array D in memory each time **Merge** is called, and the above picture gives a feel for how many there are. In Tutorial Sheet 2, we'll look at a more space-efficient version of MergeSort — but here we've gone for a simpler version to illustrate the basic idea with minimal distractions.

In this case, the picture is particularly pleasant because 8 is a power of 2, meaning that all our list portions split neatly in half at every level. But it would be a good exercise to draw the picture we'd get for an array of length 11, for example.

Runtime comparison

It might not be obvious at first that there'll be any dramatic difference in efficiency between InsertSort and MergeSort. However, a few simple Python experiments on randomly generated arrays of various lengths reveal a striking picture. For 'short' arrays, say of length $n = 10$, InsertSort is quicker. However, by the time our arrays reach length 100, MergeSort is winning; and as our arrays get longer, the difference becomes more and more pronounced. E.g. for an array of length $100,000$ the sorting took 15 minutes with InsertSort, but just over a second with MergeSort — another good example of the difference an efficient algorithm can make. (For more of the actual figures, see the lecture slides.)

Why is MergeSort so much faster on typical arrays? Can we explain what is going on here, in a way that gives insight into the difference? Related to this, can we say something about this difference that isn't specific to particular implementations of these algorithms — by particular programs in a particular language running on a particular machine, which is what my runtime figures refer to — but instead puts its finger on the essential difference between the *algorithms themselves*? Ideally, we'd like to say something which we know will apply to *any* implementations of our two algorithms: something to the effect that MergeSort will *always* beat InsertSort hands-down for sufficiently long lists, no matter what implementations we use.

Again this is the kind of thing that the theory of asymptotics is designed to address. We'll introduce the relevant mathematical concepts in Lectures 3 and 4 — then in Lecture 5 we'll return to InsertSort and MergeSort and see what an asymptotic analysis is able to say about them.