Informatics 2: Introduction to Algorithms and Data Structures
Notes on Lecture 1
Overview of course content

John Longley

These lecture notes are expanded versions of the lecture slides with a little more text added, similar to the oral explanation given in lectures. Not all of the images from the lecture slides are reproduced here, but some of the maths is worked through in a bit more detail.

What are algorithms and data structures?

What do we mean by an *algorithm*? Broadly, it is something like a method or recipe that can be followed for performing some computational task. And a little bit like a cooking recipe, an algorithm will typically consists of a sequence of instructions or steps to go through in order to perform that task: for example, the method for long multiplication that you learned at school would be an example of an algorithm for multiplying large numbers written in decimal. So before we sit down to write a program to solve some problem, we first need to know broadly what method we're going to follow: in other words, what *algorithm* we're going to use. So having a good grasp of algorithms for common tasks is one part of what we need to be good programmers.

What about *data structures*? Most of the things people want to do with computers involve working with *data* of some kind — whether that's a list of names and addresses, or a street map of Edinburgh. And a data structure is basically a way of storing, or representing, or structuring data so that it's easy to perform the operations we want to perform on that data. (We'll see some examples below). So once again, before we sit down to write our program that works with some kind of data, we'll need to give some thought to how the relevant data should be stored and structured so that the program works as efficiently as possible. So a good knowledge of data structures is another important part of a good programmer.

In fact, it turns out that algorithms and data structures are very closely intertwined, which is why it makes sense to study them together.

Some tasks calling for algorithms

To give a bit more of a flavour for what algorithms are about, let's look at some practical examples of computational tasks that call for some efficient method

or algorithm. (You might like to think a bit about how you might go about performing these various tasks.) Not all of these are *exactly* the problems we'll be addressing in this course, but they give the general flavour of the territory we'll be exploring.

Sorting a list of length 1000000000. A very common task is that of sorting a long list of items into some fixed order, such as alphabetical order or numerical order. This is naturally what we want to do if we want to create a database with records for a billion people, for example.

Crawling the Web. Suppose we're building a search engine for the World Wide Web. As part of that, we'll want a web crawler that goes looking for pages that can be reached from already known pages by repeatedly following links. To keep it simple, let's suppose that we give the crawler a single starting page, and we want it to find all the pages that are reachable from that page in any number of steps. We want to ensure that our web crawler doesn't get stuck going round and round in a loop, but also that it does eventually discover every page that's reachable.

Finding shortest/fastest/cheapest routes. Suppose we have a street map of Edinburgh, and we want to know the fastest route from A to B (this is what a SatNav tries to work out for you). What would be an efficient way of finding that?

Finding common substrings. Suppose we're given two strings of characters, e.g.

```
'working logarithmically'        'algorithmically speaking'
```

By a *substring* of one of these strings, we mean some *consecutive* sequence of characters within it. It's easy to see that the above strings have a number of substrings in common, e.g. `'or'` and `'king'` — but it's clear that the longest common substring in this case is `'ithmically'`. Finding this longest common substring might seem like an easy problem in this case — but how would you do this efficiently if both your strings were around 1000000 characters long?

Algorithms for this and related problems have a number of applications. One application concerns plagiarism detection: if two students both submit a piece of coursework, there are algorithms one can use to test whether they share large portions in common. More interestingly, perhaps, such algorithms are used in genetics to detect shared segments within long DNA sequences.

Primality testing. If I gave you a large number n, how would you tell if it was a prime number or not? If n were just a six-digit number, it would be feasible to do this by testing whether n had any factors up to $\sqrt{n}$. But what if n were a 100-digit number? That might sound like a purely recreational mathematical problem, but it turns out (as you may learn in the DMP course) that the ability to identify prime numbers is actually of fundamental practical importance for modern computer security and cryptography.

In summary, all of these are problems of practical importance that call for an efficient method for solving them — and in each case, there's something interesting to say about the kinds of algorithms one might use. For some of these problems, you might well be able to think straightaway of an 'obvious' method that would do the job. However, it often turns out that...

> *With a bit of cleverness, we can come up with an algorithm that's dramatically more efficient than the obvious one.*

That's really the motto for the whole of this course, and indeed for the whole subject of Algorithms and Data Structures: the idea that with some careful thought, we can often find algorithms that improve spectacularly on the obvious one and can be used on much larger examples.

What do we mean by *efficient* here? There are various kinds of efficiency we might be interested in. Most obviously, we'll be talking about ...

- *Time-efficiency*: finding a method that works as quickly as possible.
- *Space-efficiency*: find a method that requires as little memory as possible.

In practice, we're often interested in both of these — although which matters most may depend on what we're trying to do. We'll be considering both of them in this course, though *Time* will get rather more time/space than *Space* does. (One could also apply similar ideas to reason about other kinds of efficiency, e.g. minimizing energy consumption or number of disk accesses.)

Problems, algorithms, programs

There can be *many different algorithms for solving the same problem.* For example, if the problem is to sort a long list of numbers into increasing order, there are many algorithms that we might consider, including relatively 'obvious' algorithms like InsertSort and BubbleSort, and less obvious but generally more efficient ones like MergeSort, QuickSort and HeapSort. (All five of these algorithms will feature in this course.)

Also, there can be *many different programs that implement the same algorithm.* An algorithm is not the same as a specific program in a specific language: rather, the algorithm is something like the *general method* that a program might use.[1] For instance, if both you and I sit down to write a Java program that implements MergeSort, it's highly unlikely that we'll produce exactly the same code — although anyone who understands the MergeSort algorithm will recognize that both your program and mine are following this basic method. Alternatively, you could write a MergeSort program in Python, or any other language you like. Indeed, for the purpose of this course, it's fair to say that *any* reasonable algorithm can be implemented in *any* reasonable programming language — although it might be that some languages are a bit better suited to certain algorithms than others.

[1] So it's fair to admit that the concept of an algorithm is actually a slightly fuzzy one — we don't have a crisp mathematical definition of precisely what an algorithm is, or when two algorithms are 'the same'. But you'll soon get used to the way the term is used.

There were algorithms before there were computers

Obviously, in this course we're obviously most interested in doing things with computers. But if an algorithm is basically just a method or recipe for solving some problem, then algorithms are in principle just as relevant to hand calculation as they are to computer calculation. Indeed, in the days before computers, people were just as motivated to find efficient ways of calculating things as we are today — if not more so. In fact, the very word *algorithm* comes from the name of the 9th century Persian mathematician al-Khwārizmī, who wrote a very influential textbook on doing arithmetic in decimal notation — including methods for addition, subtraction, long multiplication, long division and square-rooting that are close to the ones you learned in school. And even earlier — in classical Greek times, before they were called 'algorithms' — some important examples of algorithms were known. A good example is Euclid's algorithm for computing the Greatest Common Divisor (GCD) of two numbers (also known as their Highest Common Factor). And even though this is more than 2000 years old, this is still a classic and beautiful example of a simple, clean and extremely efficient algorithm.We can represent Euclid's algorithm very concisely in a 'recursive' style:

GCD(m, n): # (where $m \geq n$)
  $r = m \mod n$
  if $r == 0$ then return n
  else return GCD(n, r)

For example, to compute GCD(4851,840), we first compute 4851 mod 840 = 651. Since this is not zero, our problem reduces to computing GCD(840,651). For this, we do the same again: 840 mod 651 = 189, so the problem reduces to GCD(651,189). Repeating the process, we get down to GCD(189,84), then GCD(84,21). And at this point, we find that 84 mod 21 = 0, so we return 21 as our final answer. It turns out that this method continues to work very efficiently even for numbers of a few thousand digits, and even today it's essentially the best method we know for numbers of this magnitude.[2]

So, there were algorithms before there were computers. *But* now that we have computers, algorithms are absolutely everywhere! Indeed, it's hard to think of any branch of computer science, or any major application of computing, that doesn't involve algorithms. For this reason, the whole topic of Algorithms and Data is seen as an absolutely core part of Computer Science and Informatics, and any CS degree programme at any university in the world will typically feature a course broadly similar to (if not quite as cool as) this one.

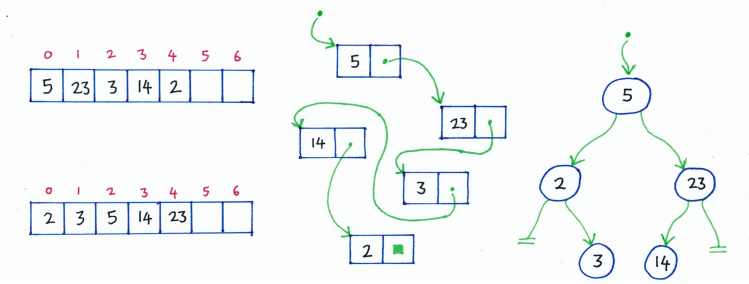
A first taste of data structures

Now let's look at some examples to give the flavour of data structures. As a simple problem to consider, let's think about how we might store a *finite set of whole numbers* (something like $\{5, 23, 3, 14, 2\}$) in a computer's memory. We are interested in being able to perform the following 'set operations' efficiently:

[2]When I was at school, they taught me that the way to compute the GCD of m and n was to factorize both numbers and then see what prime factors they had in common. That's actually not a good approach at all for very large numbers (e.g. 1000 digits each), because factorization is itself a notoriously hard problem. Euclid's method is far superior!

- Looking up whether a given number (e.g. 11) is a *member* of our set.
- Inserting a new number into the set.
- Deleting an existing member of the set.

The following picture informally shows four possible ways of representing our set $\{5, 23, 3, 14, 2\}$ in memory:



Here we'll just try to give the general flavour of these representations, leaving the details to later in the course.

First, on the left, I've illustrated two ways of storing our set using an *array*. In both cases, I've chosen an array with 7 cells, indexed by the numbers 0,. . . ,6, and used the first 5 of these cells to store the members of our set: the idea is just that I've included two spare cells in case we want to insert some new members. (We'll also want to keep a record somewhere of which array elements are currently in use, but let's not worry about that for now.) The difference between these two array representations is that one of them is *sorted* — we've carefully arranged the numbers in increasing order — while in the other, we've allowed the numbers to appear in any order they like.[3]

What practical difference does it make whether our array is sorted or not? Let's think about each of our set operations in turn:

- Looking up whether a number n is a member of a set will be much easier for a sorted array, in just the same way that in an old-fashioned dictionary, looking up a specific English word (such as *skulduggery*) is made easier by the fact that the words are arranged alphabetically. For an unsorted array, we would have no choice but to examine each array entry in turn until we either found our n or reached the end.
- On the other hand, inserting a new number is easier if we don't care about sorting: we can just stick the new number in the next available array cell (cell 5 in this case). If our array was sorted and we wanted to keep it sorted, we'd have to locate the correct position for the new entry, then shuffle any larger numbers one place to the right to make room for it. There is also a further concern that applies to both array representations: what happens if the array is already full and we want to insert a new number? We could perhaps move to a bigger array at this point, but that would take some real work.

[3]Remember that a *set* is just a collection of elements without any ordering imposed on them, so e.g. $\{5, 23, 3, 14, 2\}$ and $\{23, 14, 2, 3, 5\}$ are just two ways of writing the very same set. This is the difference between *sets* and *lists*: the lists $[5, 23, 3, 14, 2]$ and $[23, 14, 2, 3, 5]$ are definitely different.

- What about deleting a member of this set? If the array is sorted, it will be easier to locate the element we want to delete — but even so, things are not great, as we may need to shuffle some other elements to the left so that we're not left with a gap.

Later in the course, we will see how to quantify such efficiency differences more precisely. For now, we can just now that whether a sorted or unsorted array is better might depend on whether we care more about the efficiency of lookup or of insertion.

In the middle of the picture, I've shown a different kind of representation using a *linked list*. Here the idea is to use a bunch of *cells* that can live anywhere in the computer's memory they like. Each cell has two halves: the first half contains one of the members of our set, while the second half contains a *reference* (or *pointer*) to the memory address where the 'next' cell is to be found — or else a special marker to signal that we're at the end of the list. Given a reference to the first cell in the list, we're then able to search through the list by 'chasing pointers' from one cell to the next, and this gives us a way to see whether a number n is present. Linked lists are the usual way of representing lists in memory in functional languages like Haskell.

How efficient would the various set operations be using this linked list representation. Of course, lookup will be no better than for unsorted lists — we'll still have to search through the entire list if our given number n isn't present. But insertion can be done very efficiently — we can just create a new cell containing the new number, and make this cell point to the start of the old list — and in this case, we don't need to worry about the problem of array overflow, as we can just keep adding new cells until the available memory fills up. Linked lists also have other advantages, as we'll see later in the course. But if we're wanting to do a lot of lookup operations on our sets, they're not a great choice.[4]

So of the three data structures we've looked at so far, some are good for lookup and some for insertion — but none of them are good for all three of the set operations we care about. And this raises a natural question: Can we find a data structure for representing sets of numbers that gives us fast ways of doing all three operations: lookup, insertion and deletion? As we'll see around Lecture 9, there is indeed a way of achieving this, though it's far from obvious.

To give just a hint of the idea at this stage, at the right of our picture I've shown a representation of our set by an *ordered tree*. This consists of a bunch of *nodes* which each store a member of our set, and which are each equipped with references to at most two *child nodes*. (There are also *null references* that don't point to another node.) So, for example, at the top of the tree we have node carrying the number 5, equipped with pointers to two 'subtrees'. And the thing to note is that all the numbers appearing in the left subtree are less than 5, and all those in the right subtree are greater than 5. Likewise, if we look at the nodes carrying 2 or 23, we see that if there are any numbers in the left subtree, they will be less than the number on the node, and if there are any numbers in the right subtree, they will be greater. One can imagine that, as long as the tree doesn't have too many levels, this property will make it easy to lookup a number, in much the same way as we do with a sorted array. One can also perhaps imagine that inserting a new number won't be too much work

[4] Question to ponder: If we worked with *sorted* linked lists, what difference would that make to the efficiency of lookup and insertion?

either, as we can just add a new node with appropriate pointers, and don't need to do the kind of shuffling that's necessary for sorted arrays. It turns out that, with some further ingredients, one can make this idea fly and obtain a data structure for sets of numbers which supports good implementations of all three set operations (in a sense we'll be able to make precise).

Algorithms as a technology

In this lecture we've said a bit about algorithms and a bit about data structures, but let's conclude by putting all of this in a wider context.

What kinds of things make computers perform tasks faster? One can think of several answers to this: progress in hardware design allowing faster clock speeds; use of parallel processing; improvements to compilers and the optimization techniques they use, etc. But just as significant as any of these are advances in the area of algorithms and data structures. If you can devise a more efficient algorithm for some task, or a more efficient data structure to use, this will often have a dramatic impact on performance. And indeed, even on some quite old algorithmic problems that have been around for several decades now, new advances are still being made.[5] And this sometimes leads to dramatic improvements on real practical tasks. So in this sense, we can say that *algorithms and data structures are a technology*, no less than the other things we've mentioned. Certainly people like Google care a lot about them: it's quite common for Google to ask questions about algorithms and data structures at internship interviews. So there are plenty of reasons why the material in this course is relevant to computing in what some people like to call 'the real world'.

[The lecture concluded with a slide on how the course material will be structured, and one on suggested reading materials. We won't duplicate this information here.]

[5] Even for a problem as basic as sorting a list, it seems there's still more to say. For many years, the default sorting algorithm used by Python was one known as TimSort (modestly so named by its inventor Tim Peters), but in 2022 this was replaced by a refined version called PowerSort. (Both TimSort and PowerSort are based on the idea of MergeSort, which we'll consider in Lecture 2.) The idea of PowerSort draws on old algorithmic work by Mehlhorn on *binary search trees* (1977), which was not widely known until recently.