Lectures 3: Algorithms

The aim of this section is to explain some ideas that will help you understand aspects of cryptography. I will not be going into details since this is really a subject in its own right called *complexity theory*.

Algorithms

The first idea we shall need is that of an *algorithm*. Informally, this is a procedure for solving a problem that works mechanically without the need for intelligent intervention and delivers an answer in a finite amount of time. Learning maths involves learning a lot of algorithms.

Well-known algorithms include the usual ones learnt at school for adding, subtracting, multiplying and dividing numbers by hand. Another well-known algorithm is the one for multiplying two compatible matrices together — first row into first column, first row into second column, etc.

In the first half of the twentieth century, mathematicians became interested in defining exactly what an algorithm was. They needed to do this in order to solve some long-standing questions on the foundations of pure maths. A number of different definitions were suggested by various mathematicians living different places. Remarkably, they all turned out to be equivalent; so the vague idea of what an algorithm is can be made mathematically rigorous but we shan't need that rigour in this course.

One of the most influential definitions was due to Alan Turing, *the Turing machine*, who, in effect, described the conceptual blueprint for the computer. The consequence of this is that not only are programs algorithms but every algorithm, if it is an honest-to-goodness algorithm, can be implemented by a program. Thus algorithms and programs are two sides of the same coin: a program is an algorithm that works. [1]

You might think that once you had an algorithm to solve a problem your woes would be over. But this isn't true because there are other woes to worry about. I shall explain what they are by looking at an example called the *travelling saleman problem (TSP)*.

The particular example I have chosen was stolen from Keith Devlin's book and I recommend his Chapter 3 for further reading. A travelling saleman, based in S(pringfield), needs to peddle his wares in three other towns: O(ldtown), M(idtown) and N(ewtown) returning home at the end. He shouldn't visit any town more than once and to save on fuel and shoe-leather he needs to minimize the total distance travelled. The distances between the town are given below in some units — let's say kilometers.

[1] A joke

	S	O	M	N
S	0	54	17	79
O	54	0	49	104
M	17	49	0	91
N	79	104	91	0

In order to solve the problem, we have to list all possible routes and then find the one whose total distance is a minimum.

Route	Distance
S(OMN)S	273
S(ONM)S	266
S(MNO)S	266
S(MON)S	249
S(NOM)S	249
S(NMO)S	273

In this case, we see that we have to check 6 possible routes and we find that the route $S - M - O - N - S$ whose distance is 249km is a shortest route.

Although this is an example, we can easily convert it into an algorithm. The input to the algorithm would be a table of distances such as the one above. If there are n towns, in addition to the hometown, then we shall say that the input has *size* n. In the example above, $n = 3$. The algorithm works by listing all the permutations of the n towns, this is $n!$. Above $n = 3$ and so $n! = 6$ which is correct. For each such permutation, we calculate the total distance described by that route. We then find the route of smallest overall length (if there is more than one then we use some rule for picking one) and output that route. This is an algorithm even if work would have to be done to convert it into a program.

So, what's the problem? Let's suppose that $n = 27$. Let's also suppose that each step of the algorithm — generate a permutation of the n towns and add up the total distance — can be performed really, really fast say 10^{-9} seconds. How long would the algorithm take to run in years? The answer is $27! \times 10^{-9}$ seconds that is approximately $10^{28} \times 10^{-9} = 10^{19}$ seconds. The table below will help you get some idea of how long this is.

Seconds	Years (approx)
10^{9}	30
10^{11}	3 thousand
10^{15}	31 million
10^{18}	31 billion

To put this into perspective, the universe is approximately 15 billion years old. Thus to solve the above problem for the input size above and using our algorithm would take over twenty times the current age of the universe to accomplish.

Now, let me emphasise something: *there is nothing wrong with my algorithm*; it is not wrong or faulty. For small sized inputs it would even deliver an answer in a reasonable amount of time. But as the input size increased the time taken would start to get out of hand even if we could speed up the time taken on each step. One tenth the age of the universe, even a millionth the age of the universe is still a very long time. And, even if it were possible to speed up each step enough that my problem size $n = 29$ could be solved quickly, I could just choose n to be a bit bigger and we would have the same problem all over again.

The difficulty lies in the relationship between input size and the number of steps that the algorithm needs to use to solve the problem is a really nasty function: it is something like $n \mapsto n!$. We don't need to be overly precise because however you dress it up it is still nasty.

The root of the problem is that my algorithm does nothing very clever: it is a what is called a *brute-force* method that looks at all possibilities in order to arrive at an answer.

Let me stress again that the algorithm is correct.

This example tells us that simply having an algorithm is not enough: we need one that works efficiently.

I shall now describe some of the maths that's used to describe the efficiency of algorithms.

I have left hanging an obvious question: is there is better way of solving (TSP)? In order to introduce an element of drama into the lectures, I shall return to this question later.

We now come to the important definition which I have tried to motivate by the above example.