Lecture 4: Complexity theory

In the last lecture, I described to you a bad algorithm to an important problem. In this lecture, I want to make precise what I mean by a good algorithm and so by extension what I mean by a good problem.

Let A be an algorithm. The *time complexity* of A is the function f from $\mathbb{N}$ (or perhaps $\mathbb{N}\backslash\{0\}$)) to $\mathbb{N}$ where $f(n)$ is the **maximum** number of steps which A uses on any input of length n. What do we mean by 'steps'? We mean the 'important' operations needed to calculate the solution over all inputs of length n; what are considered 'important' operations varies from problem to problem. Although this sounds a bit vague, in practice it is usually clear what we mean and, in any event, I don't want to go too deeply into this theory.

Thus the time complexity for our algorithm to solve the (TSP) is given by the function $n \mapsto n!$. And here 'step' refers to the process of listing the next permutation of towns and I'm not bothering about counting the number of operations needed to add up all the distances.

Three important points. First, the word *complexity* is NOT being used as a judgement on how hard the problem is for you or me to understand. Second, although the word *time* is used, how fast (in terms of seconds) a program based on the algorithm takes depends, amongst other things, on how much money you have spent on your computer which is clearly irrelevant to mathematicians. The time-complexity of an algorithm is a measure of how much work the algorithm has to do for each input size. Third, in defining the complexity of an algorithm you look at the *worst-case* for each input size. It might be that for most inputs of size n things work well, and that it is only the occassional input of size n that causes problems. It's also possible that for the particular inputs you are looking at the bad inputs don't arise or that the average behaviour is just fine. These are all valid points and complexity theory has ways of handling them which I shall not need in this course.

To calculate the exact value of a time complexity function is usually a difficult mathematical question. However, the exact value of $f(n)$ is rarely needed. Instead, we are happy with a calculation which gives us a good idea of what $f(n)$ is. In addition, we tend to concentrate on the behvaiour of $f(n)$ for large values of n: this is because for small inputs the algorithm may need to deal with special cases which take an uncharacteristic time to calculate. Finally, if we think of the time complexity function as measuring maximum time taken to run a program over all inputs of length n, then we would like our estimate of

$f(n)$ to be insensitive to either the unit of time used or the speed at which the machine on which we run the program operates.

Mathematicians had already devised a way of being precisely vague about functions in a way that deals with the above three points.

Let f and g be two functions from $\mathbb{N}$ to itself. We shall now compare the sizes of f and g

- We say that $f(n) = O(g(n))$ if there is an n_0 and a number $c_1 > 0$ such that $f(n) \leq c_1 g(n)$ for all $n \geq n_0$.
- We say that $f(n) = \Omega(g(n))$ if there is an n_0 and a number $c_2 > 0$ such that $f(n) \geq c_2 g(n)$ for all $n \geq n_0$.
- We say that $f(n) = \Theta(g(n))$ if both $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Loosely speaking, the O is a generalization of $\leq$, the Ω is a generalization of $\geq$ and the Θ is a generalization of $=$.

This terminology is used to describe the time complexity of algorithms as follows. We say that an algorithm is $O(n^2)$ meaning that the time complexity of the algorithm is given by a function $f(n)$ such that $f(n) \leq c_1 n^2$ for some positive number c_1 and for n large enough. We are therefore giving an *upper bound* to the running time of the algorithm.

If we say that an algorithm is $\Theta(n^2)$ then we are being more precise; the definition of Θ implies that we are sandwiching the running time between two constant multiples of n^2 for n large enough.

Actually in this course I shall only use $O(g)$ notation.

In complexity theory, certain functions tend to crop up again and again. I shall recall the definitions here just for completeness.

Polynomial functions have the form

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$$

where n is called the *degree* of the polynomial (if $a_n \neq 0$). Observe that polynomials are formed using addition, subtraction and multiplication so they are easy to calculate. In complexity theory the polynomials which arise usually have low degrees:

- $f(x) = a$ *constant.*
- $f(x) = ax + b$ *linear.*
- $f(x) = ax^2 + bx + c$ *quadratic.*
- $f(x) = ax^3 + bx^2 + cx + d$ *cubic.*

Exponential functions have the form a^x where $a > 0$ usually $a = 2$ or 10. Some important rules for manipulating exponentials are:

- $a^{x+y} = a^x a^y$ addition rule.
- $(a^x)^y = a^{xy}$ product rule.

- $a^{\frac{1}{n}} = \sqrt[n]{a}$.
- $a^{-n} = \frac{1}{a^n}$.
- $x < y \Leftrightarrow a^x < a^y$.

Logarithmic functions are the inverses of exponential functions. Thus we define $\log_a(x)$ in such a way that $x = a^{\log_a(x)}$. Some important rules for manipulating exponentials are:

- $x < y \Leftrightarrow \log_a(x) < \log_a(y)$.
- $\log_a(x) + \log_a(y) = \log_a(xy)$. We prove this result as follows:

  $$xy = a^{\log_a(x)} a^{\log_a(y)} = a^{\log_a(x)+\log_a(y)}$$

  using the addition rule for exponentials. The result now follows by taking $\log_a$ of both sides.
- $\log_a(x^r) = r \log_a(x)$. We prove this result as follows:

  $$x^r = a^{\log_a(x^r)}$$

  but $x = a^{\log_a(x)}$ and so

  $$x^r = (a^{\log_a(x)})^r = a^{r \log_a(x)}$$

  by the multiplication rule. The result now follows by taking $\log_a$ of both sides.
- $\log_a(x) = \log_a(b) \cdot \log_b(x)$. We prove this as follows. By definition

  $$x = b^{\log_b(x)}.$$

  Take $\log_a$ of both sides to obtain

  $$\log_a(x) = \log_a(b^{\log_b(x)}) = \log_b(x) \log_a(b)$$

  using the result above.

Good and bad problems and algorithms

We say that an algorithm runs in *polynomial time* if its time complexity function if O(some polynomial). Problems that can be solved by such algorithms are said to be *tractable*. The class of such problems is denoted by **P**. These are good problems and their algorithms are likewise good. If an algorithm runs in a time worse than polynomial time we say that it runs in *exponential time*. These are clearly bad algorithms.

Our algorithm for solving (TSP) is therefore a bad one because it runs in exponential time. But that doesn't prove that (TSP) isn't in **P**. Perhaps if I were cleverer I could find a polynomial time algorithm to solve it?

In fact, although there is much room for improvement in my own cleverness department I don't feel too bad. It is, in fact, one of the great unsolved questions of mathematics: is there a polynomial time

algorithm for solving (TSP)? Almost certainly there isn't and the problem really is intractable but the crucial point is that we don't (yet) have a proof.

This question is usually stated in its equivalent form: is $\mathbf{P} = \mathbf{NP}$? But the approach above avoids the technicalities of defining what I mean by **NP**.

Deciding if a number is prime

Now that we have more information about algorithms we shall now apply it to some questions connected with numbers. Here is a very simple one: find an algorithm to determine whether a given number is prime. We call this the **prime problem**.

We describe an algorithm which uses only the simplest ideas. To determine whether n is prime or not we do the following: for each whole number $m \leq \sqrt{n}$ we check to see if $m \mid n$. If at any point, we find such an $m \mid n$ we stop and output the answer 'the number is not prime'. On the other hand if no number $m \leq \sqrt{n}$ divides n we output 'the number is prime'. (If you don't already know you should be able to figure out why you only need to check $m \leq \sqrt{n}$.)

What is the time complexity of this algorithm? Well, the important step is that of trial division and we have to carry out about $\sqrt{n}$ of those. Thus on the face of it we might think that the complexity function is $n \mapsto n^{\frac{1}{2}}$ which is certainly faster that $n \mapsto n$ and so we might be led to believe that the algorithm runs in polynomial time. However, there is a problem. The number represented by n is not the length of the input. The length of the input is the number of digits in n.

Suppose that n has d digits in base 10. Then $10^{d-1} \leq n < 10^d$. Thus $d - 1 \leq \log(n) < d$ where the logarithm is taken to base 10. It follows that

$$d = \lfloor \log(n) \rfloor + 1$$

where $\lfloor r \rfloor$ means the largest integer less than or equal to the real number r. We should really measure the size of our input using d and not n. Roughly $n = 10^d$ and so the time complexity is more like $d \mapsto 10^{\frac{d}{2}}$. Thus to determine whether a number with d digits is a prime or not is going to need about $10^{\frac{d}{2}}$ trial divisions. This is exponential not polynomial.

Thus the obvious algorithm for deciding whether a number is prime or not is, whilst completely correct, absolutely useless except for small numbers.

Can we do better? The algorithm I have used is a brute-force method. But the problem doesn't feel that different from the (TSP) which might lead us to think it might be intractable. However, in 2002

the following remarkable theorem was proved.

Theorem [Agrawal, Kayal, Saxena (AKS)] *The problem of determining whether a number is prime can be decided in polynomial time.*

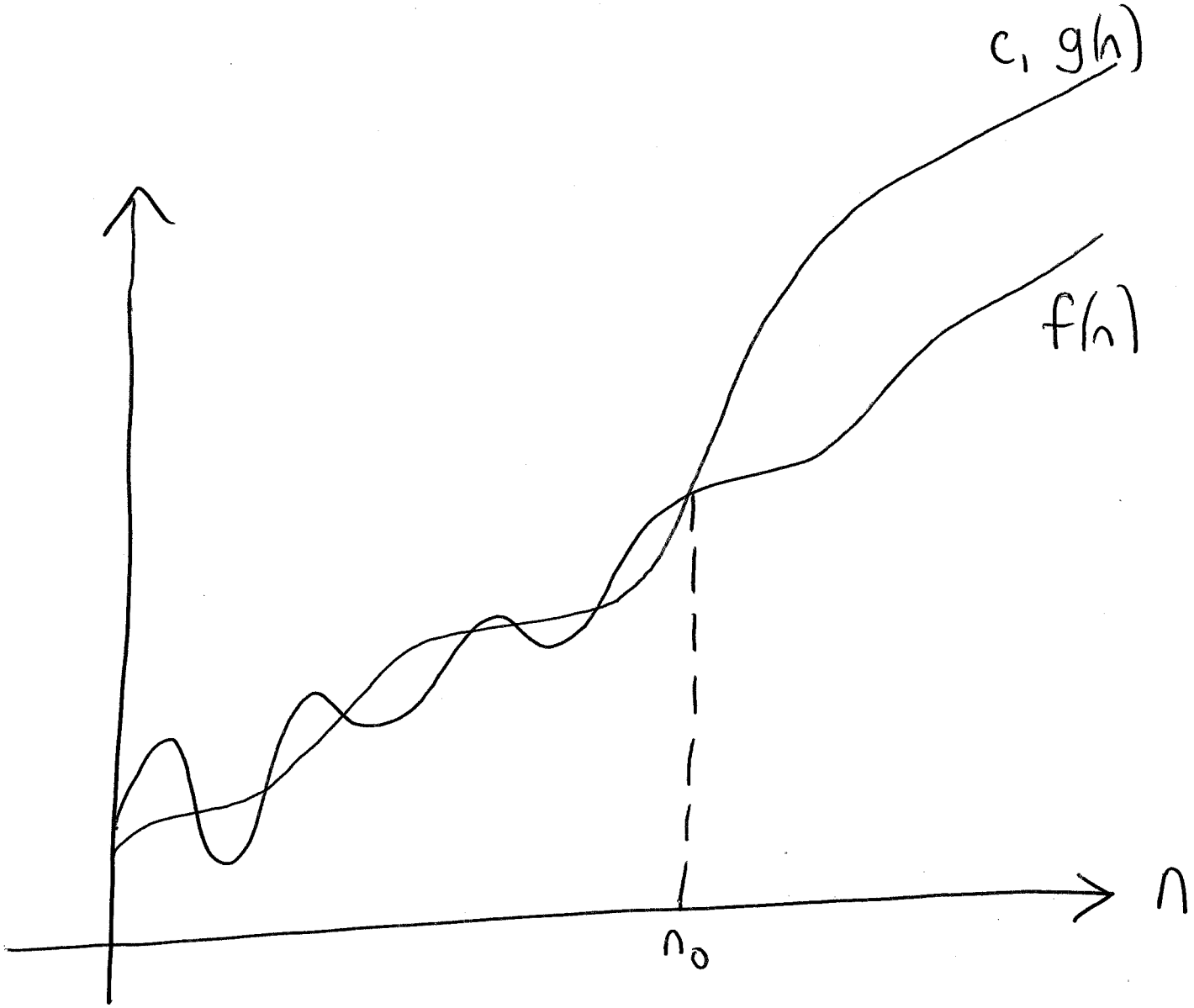
I won't be able to prove this theorem in this course, but I wanted to mention it because it is so surprising. It tells us that there is extra structure in the natural numbers that we don't immediately see but which mathematics (after many centuries) can exploit.

The factorization problem

When you look at the details of the above theorem something very strange emerges: *if I input a number that is composite it will tell me that that number is composite, but it will not produce a single factor.*

So, here is a problem that sounds a bit like the above one. **The factorization problem**: Given a composite number n find two non-trivial factors p and q such that $n = pq$. We now come to the rub: despite hundreds of years of attempts by the best mathematical minds (and remember that includes Gauss) no polynomial time algorithm is known to solve this problem. Of course, this isn't a proof that one doesn't exit. However, this is more than of purely mathematical interest: your security depends on this problem not having a polynomial time algorithm.

We shall investigate the prime problem and the factorization problem in more detail in later lectures.



$$f(n) = O(g(n))$$

This picture shows you the idea behind the definition.