1. First Lecture : 29/10

Arithmetic as a subject of serious mathematical research has its origins in the work of Euclid. It is no exaggeration to say that the contents of Books 7-9 of Euclid's Elements (especially Book 7) form the basis of the entire modern subject, though it would take almost 2000 years for anyone to build significantly on his work[1]. Euclid is best known for proving the following two fundamental theorems about prime numbers :

Theorem 1.1. (Fundamental Theorem of Arithmetic) *Every natural number has a unique representation as a product of primes.*

Theorem 1.2. *There are infinitely many primes.*

Today we will prove Theorem 1.1 in detail. The really significant part of the theorem is the word 'unique'[2]. If we drop this word, then we get a statement which is much easier to prove, namely

Lemma 1.3. *Every natural number has some representation as a product of primes.*

Proof. The proof is by (strong) induction on the natural number n. The lemma is certainly true for $n = 1$ (which has an empty representation as a product of primes) and $n = 2$. Assume it is true for $n = 1, 2, ..., k - 1$ and consider the integer k. Two cases arise :

Case 1 : k is prime. Then $k = k$ is an admissable representation.
Case 2 : k is composite. Then, by definition of what that means, there must be numbers u, v such that $k = uv$ and $1 < u, v < k$. By the induction hypothesis, each of u, v is a product of primes. But then so is uv (just concatenate the representations of u and v). $\square$

Remark 1.4. The above simple proof is interesting in that it is probably one of the oldest documented examples of a proof by *induction*. This is an important method for proving statements about the natural numbers, which, expressed in more modern set-theoretic language, relies on the so-called

Well-Ordering Principle *The set $\mathbb{N}$ of natural numbers is well-ordered, i.e.: every non-empty set of natural numbers has a least element.*

To illustrate the connection between WOP and the method of induction, let's reformulate the proof above.

Reproof of Lemma 1.3. Let S be the set of natural numbers which cannot be written as a product of primes. We are claiming that S is empty. If it weren't then, by WOP, it would have a least element, say k. Clearly, $k > 2$ and, by definition, each of the numbers $1, 2, ..., k - 1$ can be expressed as a product of primes. But now repeat the

[1]Unlike Euclid's encyclopediac recording of the fruits of Greek geometry, it is speculated that much of what he wrote on arithmetic was, in fact, his own work.

[2]It is my experience that this fact is generally overlooked in high school curricula, where the focus is usually placed on computing prime factorisations. This is a classic example of putting the cart before the horse.

argument above to deduce that so can k. This contradiction completes the proof.

Fermat made expert use of the WOP in his work, though he called it the *method of infinite descent*. This is an appropriate choice of phrase for the kinds of applications he made, which basically were about showing that certain algebraic equations had no integer solutions. We will come to Fermat's work later on.

Now let us continue with the proof of Theorem 1.1. It remains to prove the uniqueness of prime decompositions. The only known way to do this involves what at first looks like a long detour, through the concept of *greatest common divisor*. The first person to figure this out (i.e.: probably Euclid) must therefore have been really clever !

Definition Let $a, b \in \mathbb{N}$. The *greatest common divisor* of a and b, denoted GCD(a, b), is the largest integer c such that $c|a$ and $c|b$.

Euclid's key insight is the following :

Euclid's Lemma *Let $d = GCD(a, b)$. Then d equals the smallest positive integer c for which the equation*

$$ax + by = c \tag{1.1}$$

has an integer solution $x, y \in \mathbb{Z}$.

Proof. Note that, by WOP, the set S of positive integers c for which (1.1) has a solution does indeed have a least element (since S is obviously an infinite set). Let d be this least element. We shall show that

(i) $d|a$ and $d|b$,
(ii) if $c|a$ and $c|b$ then $c|d$,

and these obviously suffice to prove that $d =$ GCD(a, b) (note that they show that GCD(a, b) automatically has a stronger property, namely (ii), than what is required of it by its definition). First of all, property (ii) is obvious, for if c is any common divisor of a and b, then c also divides $ax + by$ for any integers x, y, hence divides d.

We prove (i) by contradiction, Suppose that $d \dagger a$, for example. Then there exist integers q, r such that

$$a = qd + r, \quad 0 < r < d. \tag{1.2}$$

Let x, y be any integers satisfying $ax + by = d$. Substituting (1.2) into this equation yields (after some rewriting)

$$ax' + by' = r, \quad \text{where } x' = 1 - qx, \quad y' = -qy. \tag{1.3}$$

But this contradicts the definition of d as the least positive integer for which (1.1) has a solution. $\square$

For the proof of Theorem 1.1, we apply Euclid's Lemma to deduce the following :

Lemma 1.5. *Let $a, b \in \mathbb{N}$ and p be a prime. If $p|ab$ then $p|a$ or $p|b$.*

Proof. Suppose p divides neither a nor b. Since p is prime, this implies that GCD$(a,p) =$ GCD$(b,p) = 1$. By Euclid's Lemma, it follows that there exist $x, y, z, w \in \mathbb{Z}$ such that

$$ax + py = bz + pw = 1. \tag{1.4}$$

Consequently,

$$1 = (ax + py)(bz + pw) = (yw)p^2 + (axw + byz)p + (xz)ab. \tag{1.5}$$

The left-hand side of (1.5) is obviously not divisible by p. But the right-hand side is, since $p|ab$. This contradiction completes the proof of the lemma. □

Corollary 1.6. *If $a_1, ..., a_k \in \mathbb{N}$ and p is a prime such that $p|a_1 \cdot ... \cdot a_k$, then $p|a_i$ for at least one i.*

Proof. Lemma 1.5 is the case $k = 2$. The general case is easily established by induction on k. □

We are now ready to complete the proof of Theorem 1.1. Let $n \in \mathbb{N}$ and let

$$n = p_1 \cdot ... \cdot p_k = q_1 \cdot ... \cdot q_l \tag{1.6}$$

be two hypothetical prime decompositions of n (thus repititions among the p_i or the q_j are allowed). We show that each p_i occurs among the q_j and vice versa, which clearly suffices to conclude that the primes occuring in the two decompositions are identical. Take p_1 for example. Eq. (1.6) implies that $p_1|q_1 \cdot ... \cdot q_l$. Then Corollary 1.6 implies that $p_1|q_j$ for some j. But since q_j is also prime, this forces $p_1 = q_j$. The same argument can obviously be applied to any p_i or q_j, so the proof of Theorem 1.1 is complete.

It is interesting to speculate to what extent Euclid was interested in having an *effective* proof of FTA, i.e.: a proof which also yielded an efficient algorithm for finding the prime factorisation of an integer input. One can always factorise a number by brute force exhaustive search, but this is both aesthetically unsatisfying and hopelessly impractical for large inputs. The crucial point is that the above proof does NOT appear to be effective. In modern times, this curious fact has acquired great attention because of the advent of the RSA public key cryptosystem, which ingeniously exploits the continued intractibility of the integer factorisation problem for its own security. Note that it is an open problem as to whether integer factorisation is 'intrinsically difficult' : to be more precise, it is not known for example whether the problem is NP-complete[3].

A hint that Euclid was indeed interested in such computational issues is given by the fact that he did present an effective version of his Lemma. What has become known as *Euclid's Algorithm* remains to this day a state-of-the art algorithm for computing the GCD of two natural numbers and exhibiting a solution to (1.1).

[3]We will return to issues of algorithmic complexity on many occasions during the course. In the next lecture, a quick introduction to the basic concepts will be given.

Euclid's Algorithm *Let $a, b \in \mathbb{N}$. The sequence of divisions*

$$\begin{aligned} a &= q_1 b + r_1, \qquad (1.7)\\ b &= q_2 r_1 + r_2,\\ r_1 &= q_3 r_2 + r_3, ... \end{aligned}$$

eventually terminates with some $r_k = 0$. In that case, the last non-zero remainder r_{k-1} coincides with $d = GCD(a, b)$. Furthermore, if we then run backwards through the sequence of equations, we get an explicit solution $(x, y) \in \mathbb{Z}^2$ to the equation $ax + by = d$.

Proof. It is clear that the sequence terminates since the r_i form a strictly decreasing sequence of non-negative integers (we're using WOP again !). Let the last equation read

$$r_{k-2} = q_k r_{k-1} + 0. \qquad (1.8)$$

We shall show that r_{k-1} satisfies properties (i) and (ii) in the proof of Euclid's Lemma. First of all, (1.8) implies that $r_{k-1}|r_{k-2}$. The next-to-last equation will have read

$$r_{k-3} = q_{k-1} r_{k-2} + r_{k-1}. \qquad (1.9)$$

From this and the fact that $r_{k-1}|r_{k-2}$ we can conclude in turn that $r_{k-1}|r_{k-3}$. Continuing the backward substitution yields in the same manner that r_{k-1} divides the left-hand term in each equation, hence finally that it divides both a and b. Thus r_{k-1} satisfies property (i).

To verify property (ii) we run forwards again. Let c be any common divisor of a and b. Then from the first equation in (1.7) we can conclude that $c|r_1$. Substituting this information into the next equation yields that $c|r_2$. And so on, until finally we find that $c|r_{k-1}$. $\square$

Next day, we shall start by studying the computational complexity of Euclid's algorithm in more detail.

2. Second Lecture : 31/10

Today is going to consist mostly of a crash-course introduction to some basic concepts in the theory of *algorithmic complexity*. Though it may seem like a long detour away from number theory, I am doing this for several reasons :

1. In the computer age, the vast computational resources at our disposal make it relevant in far more situations than ever before to ask whether a certain computation is practically feasible.

2. This is a fundamental type of question in all modern discrete mathematics, not just number theory, but number theory is a particularly rich source of problems.

3. Within number theory, computational issues are especially important for applications, for example in cryptography.

4. Many math students never seem to learn about complexity theory - it's usually left to the computer scientists. But developing and analysing high-level algorithms is more math than computer science.

WARNING ! The discussion to follow is only meant as a quick exposure to some basic notions. If you want to learn this stuff properly, then you need to take a course or read a book on your own.

The basic unit of information in computer science is the *bit*, i.e.: a single binary digit. The *(information-theoretic) complexity* of a natural number n is the number of its binary digits, in other words, the number of bits of information needed to represent it. You can talk about the information content of other types of data, e.g.: text, provided you have a way of translating it into natural numbers.

For our purposes, an *algorithm* is a procedure for computing a function $f : \mathbb{N} \to \mathbb{N}$. We can talk about the *input* and *output* of the algorithm, which are thus natural numbers. It's natural to want to be able to consider algorithms which have multiple inputs or outputs. One way of doing this within the above framework is to think of a sequence of inputs $n_1, ..., n_k$ as a single input, by concatenating their binary representations. In fact, from some general set-theoretic nonsense it follows that the framework of functions $f : \mathbb{N} \to \mathbb{N}$ is sufficient for any thinkable application.

There are two basic notions of complexity for an algorithm, namely complexity in *time* and *space*. Informally, the former measures the amount of time required for the algorithm to run, as a function of the input size (where *size* is an informal term for 'information-theoretic complexity'), whereas the latter measures the amount of storage space required by the algorithm. In what follows I will primarily be interested in temporal complexity.

When studying time complexity, one first needs to make a choice of a *computational* *unit*. This should be some basic type of operation which should run in the same time on any computer equipped with 'state-of.the-art' hardware, i.e.: the time required should depend ONLY on the hardware, and thus not be an issue for mathematicians to worry about. Traditionally, the choice of unit is the so-called *bit operation*. This involves adding two bits modulo 2, and carrying a 1 in the case of $1 + 1 = 0$. We then choose as our unit of time that required to perform a single bit operation, so that the time complxity of an algorithm is just the number of bit operations it performs.

A 'fast' or 'good' algorithm is one which requires few bit operations. This is, of course, a hopelessly vague statement. To be more precise, the most important general notion of 'goodness' for an algorithm is the following :

Definition An algorithm is said to be *polynomial-time* if the number of bit operations it performs grows polynomially in the input size. In other words, an algorithm for computing a function $f : \mathbb{N} \to \mathbb{N}$ is polynomial-time if there exists a polynomial $p(x) \in \mathbb{Z}[x]$ such that, for input $n \in \mathbb{N}$, the algorithm computes $f(n)$ after no more than $p(\log_2 n)$ bit operations.

If $p(x)$ can be taken to be a linear function, then the algorithm is said to be *linear*. If $p(x)$ can be taken quadratic, we say the algorithm is *quadratic*. And so on ... Note that this choice of terminology means, for example, that if we say an algorithm is cubic, then it doesn't rule out the possibility that there is a faster, say quadratic, algorithm for performing the same computation. In fact, it doesn't even rule out the possibility that a more careful analysis of the same algorithm will reveal it to be quadratic.

The reason why this notion is important is because the basic operations of arithmetic, addition and multiplication, are polynomial-time.

Proposition 2.1. *Addition is linear and multiplication is quadratic. In other words, there is SOME algorithm which adds two numbers in linear time, and SOME algorithm which multiplies two numbers in quadratic time.*

Proof. (*sketch)* The algorithms in question are the ones everyone learns in elementary school, just with base 2 instead of base 10. Consider addition, for example. We give two numbers $a, b \in \mathbb{N}$ as input. Thus the size of the input is $\lceil \log_2 a \rceil + \lceil \log_2 b \rceil + \epsilon$, where $\epsilon \in [0, 2]$. To add them, we write them out in binary, one on top of the other, tag on some zeroes on the left for the smaller number if necessary, so that we have the same number of digits in both numbers, and then add column-by-column. We perform a single bit operation for each column, thus the algorithm requires no more than $\lceil \log_2(\max\{a, b\}) \rceil + 1$ bit operations. This is clearly linear in the input size : in the notation of the previous definition, we can take $p(x) = x + 1$ for example.

For multiplication, we use the usual long multiplication algorithm. You'll have a bunch of rows to add, one for each digit in the larger of a and b (with zeroes tagged on appropriately on the left so that we have the same number of digits in each row). We don't count the time required to write out each row, since in base 2 this is just transcription and involves no bit operations. We then add the rows one-by-one. Each addition is linear in the input size by what we showed above, and the number of additions equals the number of rows minus one, which is also linear in the input size. Thus the total number of bit operations will be quadratic in the input size. A more careful analysis will show that one can take $p(x) = 2x^2$, for example. $\square$

Remark 2.2. There is, believe it or not, a way of multiplying numbers which is faster when the numbers become large. In fact, there are a plethora of such *fast multiplication algorithms*. However, the math behind all of them is far more sophisticated (which explains why they aren't taught in elementary school !). If you're interested then do a Wikipedia search for the *Schönhage-Strassen algorithm*.

Now let's go back to something resembling number theory, and verify that Euclid's algorithm is indeed 'fast'.

Theorem 2.3. *Euclid's algorithm runs in polynomial time. In fact, it is cubic.*

Proof. Let's consider only the 'forward part' of the algorithm, i.e.: the input is a pair of numbers $a, b \in \mathbb{N}$ and the output is $d = \text{GCD}(a, b)^4$. As in (1.7), the algorithm consists of a sequence of steps, each of which has the form : take a pair of number $n_1 > n_2$ as input and output integers q, r satisfying

$$n_1 = qn_2 + r, \quad 0 \leq r < n_2. \tag{2.1}$$

Thus each step is essentially just a division. I leave it as an exercise for you to check that, just as with multiplication, the elementary school division algorithm is quadratic. Thus to prove the theorem, it suffices to show that the number of steps cannot be more than linear in the input size. I will prove something more precise, namely :

CLAIM : For input $a, b \in \mathbb{N}$ with $a > b$, the number of steps in Euclid's algorithm is never more than $2(\log_2 b + 1)$.

Proof of Claim. The idea is to show that any two consecutive steps at least halve the remainder. Two such generic steps can be written as

$$\begin{aligned} r_{i-1} &= q_{i+1}r_i + r_{i+1}, \\ r_i &= q_{i+2}r_{i+1} + r_{i+2}. \end{aligned} \tag{2.2}$$

I am claiming that $r_{i+2} \leq \frac{1}{2}r_i$. This is obvious if already $r_{i+1} \leq \frac{1}{2}r_i$, since the remainders are decreasing. If $r_{i+1} > \frac{1}{2}r_i$ then, in the second equation of (2.2) we will have $q_{i+2} = 1$, and then $r_{i+2} = r_i - r_{i+1} < r_i - \frac{1}{2}r_i = \frac{1}{2}r_i$, as desired.

Now suppose $2^k \leq b < 2^{k+1}$. The number b is the 'zeroth' remainder in the algorithm. Every pair of steps reduces this by at least half. The algorithm terminates when the remainder is zero. Since each remainder is an integer a priori, this is equivalent to saying that the algorithm terminates when the remainder is less than one. To ensure this, we will need to halve b a total of $k + 1$ times, so the algorithm will certainly require no more than $2(k + 1)$ steps. But $k \leq \log_2 b$. This establishes our claim and completes the proof of the theorem. $\square$

[4] If you also want as output an explicit solution to $ax + by = d$, then basically the run-time doubles, since you also have to run the whole algorithm backwards.

Remark 2.4. Some interesting stuff is known about average- and worst-case scenarios for the number of steps required by Euclid's algorithm. See

http://algo.inria.fr/seminars/sem92-93/daude.ps

for a recent synopsis. Note, in particular, that

(i) the 'average' (one has to make precise what one means by this) number of steps required is $C \log_2 b$ with $C = 12/\pi^2 \approx 1.216...$

(ii) the worst-case scenario is when the inputs a, b are consecutive *Fibonacci numbers*, i.e.: $b = f_k$ and $a = f_{k+1}$ for some k, where the sequence (f_n) is defined recursively by

$$f_1 = f_2 = 1, \quad f_n = f_{n-1} + f_{n-2} \;\forall\; n > 2. \tag{2.3}$$

Notation One denotes by $\mathscr{P}$ the class of functions $f : \mathbb{N} \to \mathbb{N}$ which can be computed (by some algorithm, and for arbitrary inputs) in polynomial time. One denotes by $\mathscr{N}\mathscr{P}$ the class of functions f with the property that, given n and k, the truth or falsity of the statement '$f(n) = k$' can be decided in polynomial time.

A function in the class $\mathscr{P}$ must also be in $\mathscr{N}\mathscr{P}$. For if $f \in \mathscr{P}$ and we want to check the validity of the statement '$f(n) = k$' then we just run a polynomial-time algorithm for computing $f(n)$ and then check whether the output equals k. Thus, we can write symbolically, $\mathscr{P} \subseteq \mathscr{N}\mathscr{P}$.

I think it is intuitively reasonable to expect that the converse is not true, i.e.: to expect that there should be some computational problems out there for which it is much easier to check the validity of a candidate solution than to find the solution in the first place. Amazingly, it is not known if this intuition is valid. I personally regard the following as the most important open problem in mathematics :

Conjecture 2.5. $\mathscr{P} \neq \mathscr{N}\mathscr{P}$.

There are many interesting examples of functions which are obviously in $\mathscr{N}\mathscr{P}$ but for whom it is unknown whether or not they are in $\mathscr{P}$. One of the most important examples is the *integer factorisation problem* in number theory. In its simplest form, this takes as input a natural number n which is known to be a product of two distinct primes p_1, p_2 and outputs these two primes. Because the security of the RSA cryptosystem relies on the intractability of this problem, it has received a lot of attention in the last 30 years or so (though it is probable that even Euclid was aware of its thorny nature). To the best of my knowledge, the fastest general algorithms are based on ideas developed at least 15 years ago[5] and have run-time of the order of $e^{\sqrt[3]{\log n}}$. This is certainly not polynomial in $\log n$ and, with current technology, becomes impractical for inputs n consisting of a couple of thousand decimal digits.

I will probably not discuss the factorisation problem directly any more in this course (though I will dicuss several similar but easier problems), but there is a wealth of information about it in the literature if you are interested.

[5] These go under the name of the *general number field sieve (GNFS)*. The mathematics involved is *algebraic number theory* and beyond the scope of this course.

Remark 2.6. A function $f : \mathbb{N} \to \mathbb{N}$ is said to be $\mathscr{NP}$*-complete* or $\mathscr{NP}$*-hard* if it is in $\mathscr{NP}$ and if it is known that the statement

$$`f \in \mathscr{P} \Rightarrow \mathscr{P} = \mathscr{NP}'$$

is true. This is a very weird notion if you haven't seen it before, but amazingly there are many concrete examples known of $\mathscr{NP}$-complete functions. Many come from graph theory : for example, the problem[6] of deciding whether a connected graph has a Hamilton cycle is obviously in $\mathscr{NP}$ (to check whether a candidate path is a cycle that visits every vertex exactly once is trivial), and is known to be $\mathscr{NP}$-complete.

If it is true, as conjectured, that $\mathscr{P} \neq \mathscr{NP}$, then every $\mathscr{NP}$-complete problem lies outside $\mathscr{P}$. This is how most people view the matter[7], i.e.: that these $\mathscr{NP}$-complete problems satisfy our intuitive understanding that there should be problems for which finding a solution really is more difficult than simply checking one. Significantly, it is not known (to the best of my knowledge) whether the integer factorisation problem is $\mathscr{NP}$-complete.

[6]The words 'problem' and 'function' are used interchangeably in complexity theory. The point is that it is often more natural to describe some computational exercise in words, i.e.: as a problem to be solved, rather than as a function to compute. But one can always make the translation to functions in a purely formal manner.

[7]The alternative point of view is that, in order to drop a bombshell and prove that $\mathscr{P} = \mathscr{NP}$, all you need to do is find a clever solution to any single $\mathscr{NP}$-complete problem.