UG3 Computability and Intractability (2010-2011): Note 1

§1. General introduction. It is our intention in this course to study the fundamental nature of computation. A prerequisite for this study is a formal *model of computation.* This model should reflect our intuitive notion of computation: the things which can be computed within the formal model should be precisely the things we imagine to be 'computable'. We will study one such model in some detail later but before that we will see that it is possible to obtain some far reaching conclusions armed only with general considerations; a formal model acts as a safety check that we have not been kidding ourselves. In the first part of the course we will be concerned with what can be computed *in principle* and will address concerns about efficiency (what can be computed *in practice*) in the second part.

§1.1. Brief history. Notions of computing have been present in mathematical thinking for a very long time. An *algorithm* is a specific sequence of instructions for carrying out a task, where the instructions do not require any guesswork or 'intelligence' in their performance[1]. The earliest known algorithms were found on Babylonian tablets dating from 3000–1500 B.C. These algorithms are fairly crude and boring. Indeed there is no use of conditional tests so that the same basic algorithm is repeated as often as needed to cover different cases. There is also no use of negative numbers or of 0.

The next known algorithm is the famous one given by Euclid in Book 7, Propositions 1 and 2 of his *Elements.* This gives the familiar method of finding the greatest common divisor of two natural numbers. Again the algorithm does not use conditional tests but it does use iteration.

Perhaps the next most interesting development was provided by the 19th century mathematician Charles Babbage who designed two machines for computation. In Babbage's times navigation tables were very important but could only be produced by hand calculations. The people carrying out the calculations were called *computers* and were mostly sedate country clergymen augmenting the rewards of saving souls with the rather more tangible rewards of number crunching. Naturally such tables were prone to error. Babbage often had to check for these errors and got thoroughly sick of this drudgery. He therefore designed his *difference engine* which could calculate the tables via the method of differences. He eventually managed to get the government of the day to put up some finance to build this machine. It was part way through this process that he had the brilliant idea for

[1]The fact that the word *algorithm* is a close anagram of the Greek–derived word *logarithm* caused confusion to some etymologists. In fact the word is derived from the name of the 9th century Arab scholar *Mohammed ibn–Musa al–Khowarizmi* (it appears the change was via *algorism* to *algorithm*). It was he who introduced our present day word *algebra* through his *Al–jabr wa'l muqābalah.*

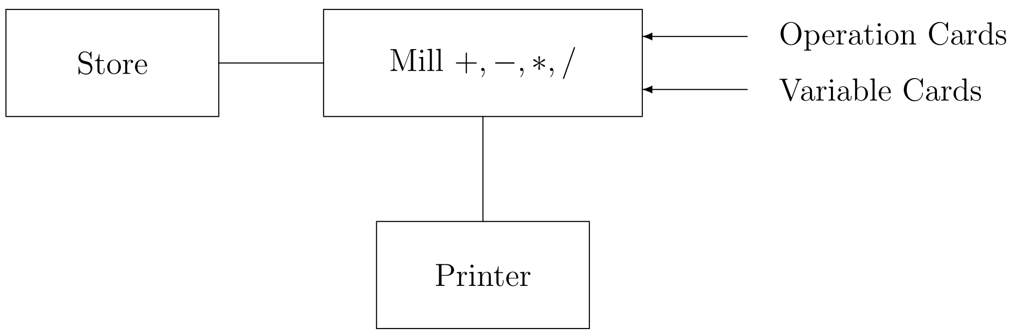

Figure 1: Schematic architecture of Babbage's analytical engine.

his *analytical engine*. The difference engine could not be programmed in anything resembling our meaning of the word, whereas the analytical engine's design and basic capabilities were remarkably like those of the modern day computer; see Figure 1. However the resemblance should not be overemphasized, e.g., it lacked the idea of a stored program; instructions were given on cards and data on separate variable cards. Babbage's major collaborator in developing the method of programming for his invention was Ada Augusta, Countess of Lovelace, a daughter of Lord Byron, the famous dissolute poet. Sadly neither of Babbage's two engines was ever completed; the major obstacle was that all the parts were mechanical.

Impressive as Babbage's achievement was, it cannot be claimed that he made any great inroads in understanding the notion of computability as a process; he was motivated by a rather specific set of problems and devoted to a particular design. The first major developments in understanding the fundamental nature of computability came in the 1930s and 1940s, the culmination of work that started around the turn of the 20th century. To a large extent these developments were motivated by Mathematical Logic and attempts by Hilbert and others to develop decision procedures for mathematics. The problem was: given a mathematical statement, decide whether or not it is provable[2]. Here the word 'decide' is troublesome and this led logicians and mathematicians to provide formal definitions of what it means to decide (as well as prove) something. Various definitions were proposed and remarkably they all turned out to be equivalent: anything decidable by one method was decidable by any of the others. One of the most interesting proposals was by Alan Turing who defined a class of abstract machines, now named after him. Turing's definition of a 'decidable problem' was: a problem is decidable if there is a Turing machine which when given a description of the problem as input eventually halts with one of two possible outputs, one output meaning 'yes' and

[2]The real aim was to decide the truth of a statement; as we will see there is a gap between (formal) proof and truth.

the other 'no' (of course it is a requirement that the output must be the correct answer). Remarkably enough, Turing was able to show that there are problems that are *undecidable*; this is not due to a lack of power in his machines, which are as powerful as any existing computer and can simulate standard logical reasoning.

Other developments of this period include Church's Lambda Calculus, Post's Production Systems, Zuse's Plan Calculus and the General Recursive Functions.

§1.2. Notation and conventions. For the sake of convenience we will express our notions of computation in textual terms (programs). However our discussion is not limited to software; it applies to any device that satisfies the few assumptions that we make and can express the behaviour described. We will assume that each program can be described as a string over some agreed alphabet A. Moreover inputs and outputs consist of a string over the same alphabet A (clearly multiple input and output can be catered for under this assumption, e.g., by reserving a special character to separate components in the case of multiple inputs or outputs—what goes wrong if we simply concatenate multiple inputs or outputs?). It is reasonable to insist that A is finite (we will revisit this point in NOTE 2). In fact there is no gain to be made by allowing infinite *countable* alphabets[3]: an alphabet $\{ a_0, a_1, a_2, \dots \}$ can be viewed as a subset of strings over the two 'letter' alphabet $\{ a, ' \}$ where a_0 is represented by a, a_1 by a' etc. Since numbers are an indispensable part of computing we will assume that we have an agreed way to represent them using our alphabet, e.g., we might assume that A contains the digits $0, 1, \dots, 9$.

We could proceed to list a few general assumptions but instead we will rely on straightforward intuition and comment on those assumptions that we do make on the way. First of all we require that our notion of computation has the ability to generate a list $P_0, P_1, P_2, \dots$ that consists of all valid programs[4], i.e., there is a program G (for Generator) that on input n outputs P_n. This assumption essentially relies on the idea that in any general notion of programming we have a program that given an input string S can decide if S is a valid program. To generate the entry P_n of the list $P_0, P_1, P_2, \dots$ we systematically enumerate all strings one after the other testing each one to see if it is a valid program; we output P_n as soon as we have have found $n + 1$ valid programs. (Of course the process just

[3]A set is said to be countable if it can be put into 1–1 correspondence with the set of natural numbers. We will discuss this further in §1.5.

[4]Note: *Validity* here should not be confused with *correctness*. All that we require is that a given string should be of the appropriate format. For example $n(n+1)/2$ is a valid mathematical expression while $n(n + 1/2$ is not; we know and claim nothing as regards the 'correctness' of the valid expression since we have not given any interpretation for it. If we also claim that the valid expression is the value of the sum $1+2+\cdots+n$ *then* we are in a position to claim its correctness as well. If on the other hand we claim that it is the value of the sum $1 + 2 + \cdots + n^2$ then it is incorrect but still a valid expression.

described is hardly a model of efficiency but in principle at least it does the job. Considerations of efficiency at this stage are hardly relevant since we are trying to set up a system in which we can then discuss such finer points.) Having chosen a method of enumerating all programs we call m the *index* of the program P_m. We can also generate all possible strings $I_0, I_1, I_2 \ldots$ over the alphabet A; these represent all possible inputs (and outputs). In fact we will encode all input strings as integers; this step is not necessary but it simplifies some points (note that we do not encode outputs). Such an encoding is quite easy to find: if $A = \{ a_1, \ldots, a_m \}$ set $\overline{a_i} = i$. Then we encode the empty string as 0 and a non-empty string $b_0 b_1 \ldots b_n$ as $\overline{b}_0 m^n + \overline{b}_1 m^{n-1} + \cdots + \overline{b}_n$. It is easy to see that for each string there is one natural number code and conversely for each natural number there is exactly one string corresponding to it.

§1.3. Non-termination. According to the discussion so far, what we require from a computational process is that we can take an input and transform it to an appropriate output: in mathematical terms we are dealing with functions. Of course we might regard certain strings as being inappropriate inputs for a particular process but we can cater for this by reserving a special character that is output when such strings are supplied. Thus we are in a situation where we want to be able to produce devices (programs) that can take in a string and eventually output a string, at which point they terminate. The key question we wish to address is the following: what do we mean by claiming that such a function is computable? We want to provide an answer that is independent of current technology, i.e., an inherent characterization of the notion of computability.

The preceding discussion shows that one feature that would clearly be desirable is to have a general theory of computation in which all programs are guaranteed to terminate and produce an output[5]. Suppose now that we can formulate such a theory and consider the following reasonable program description[6]:

```
on input $n$ let $R$ be the output of program $P_n$ on $n$;
if $R = I_0$ then return $I_1$ else return $I_0$;
```

Surely any general theory of programming should be able to express the given description: (i) in the first line we can obtain P_n by running G on input n (there

[5]We might justifiably object that for certain programs termination is undesirable, e.g., operating systems. However this is not a serious objection since our aim can be modified to that of having a general theory in which programs designed to produce an output always terminate and then introduce explicit features for non-termination. If this is possible then we have the happy situation whereby we can tell if a program terminates just at a glance.

[6]It is vital not to confuse such descriptions with actual programs. All a description does is specify the task to be carried out; it does not prescribe how it is to be done. An actual program for a task need not look anything like the description; all that is required is that the program produces the correct output for each input.

is also the reasonable assumption that we can then 'run' P_n within our program, which is effectively an assumption about compositionality); (ii) the second line assumes that we can test for equality of strings and carry out a conditional action. Assuming that we do indeed have a general theory of programming, there must be at least one program in the list $P_0, P_1, P_2, \ldots$ that expresses the above program description[7]; call this program P_m. Let S be the output of P_m when it is run with input m. If $S = I_0$ then according to the description the output must be I_1 which is clearly not possible since $I_0 \neq I_1$. So we must have $S \neq I_0$ but now we are in trouble again since, according to the description, the output must be I_0! So whatever the output is we end up with a contradiction.

The contradiction we have arrived at can easily be banished provided we give up our aim of producing a general theory in which programs always terminate. The new situation is that a valid program either terminates and produces a result or never terminates and thus does not produce a result. We can still carry out the preceding argument but now the first line of the program description is not justified, the description must be replaced by:

```
on input $n$ run program $P_n$ on $n$;
if this terminates (i.e., we get here) let the output be $R$;
if $R = I_0$ then return $I_1$ else return $I_0$;
```

This is a perfectly good description of a program and so must be represented by some P_m in our system. However the argument given above now shows that P_m does *not* halt on input m so that there is no contradiction.

We are now in a position to define (in this informal setting) what we mean by a *computable function*. We focus on functions that take strings over our alphabet and return such strings. Moreover we allow functions that may not be defined on some (or even all) strings: the domain of a function can be any subset of the set of strings. We say that such a function f is computable if there is a program P such that for all valid strings I we have that P on input I terminates (i.e., returns a value) if and only if $f(I)$ is defined and when this is so then the program returns the string $f(I)$.

As mentioned above, it is very important to understand the difference between the *definition* of a function and a method or *algorithm* for computing it (assuming there is one). Sometimes the definition gives us an obvious method and at other times we have to work hard to find one or show that the function is not computable.

[7]It is worth stressing that this argument simply shows that the description we have given does indeed correspond to a program within our system (given the assumption that all programs terminate). It does not claim, nor prove, that the described program is the only possible one within our system that could express the given behaviour. For example it is conceivable that our system has the ability to analyze P_n on n and deduce the output without actually running P_n at all.

In some cases we are unable to find which possibility applies. For example consider $f : \mathbb{N} \mapsto \mathbb{N}$ defined by

$$f(n) = \begin{cases} 1 & \text{if the first } 2^n \text{ digits of the decimal expansion of } \pi \\ & \text{have } n \text{ consecutive 7's;} \\ 0 & \text{otherwise.} \end{cases}$$

This is easily seen to be computable; there are many ways to compute π to any desired degree of accuracy so given n we compute to an accuracy of 2^n digits and check. Consider now the similar function $g : \mathbb{N} \mapsto \mathbb{N}$ defined by

$$g(n) = \begin{cases} 1 & \text{if the decimal expansion of } \pi \text{ has } n \text{ consecutive 7's;} \\ 0 & \text{otherwise.} \end{cases}$$

Here we cannot apply the previous strategy since we have no upper bound on how far to look and it might be the case that for a given n the expansion of π does not have n consecutive 7's. Indeed, at the time of writing, it is not known how to compute g. Finally let $\mathbb{P}$ be the set of all polynomials in x with integer coefficients, e.g., $x^5 - x + 1$ and $5x^{12} - 6x^7 + 33x^2 - x + 10$ are elements of $\mathbb{P}$. Consider the function $h : \mathbb{P} \mapsto \mathbb{N}$ defined by

$$h(p) = \text{number of real roots of } p.$$

For example $h(x^2 - 1) = 2$ while $h(x^2 + 1) = 0$. The definition of h gives us no clue as to how to compute it or even if it is computable. A naive attempt would be to try and find all the real roots of the given polynomial but this is full of problems (popular approaches such as Newton-Raphson iteration can miss roots or be fooled into thinking that a number is a root when in fact it simply makes the value of the polynomial very small but not actually 0). This question was a matter of great concern (and of practical importance) to researchers around the turn of the 18th century. There were many attempts to solve it, most of which failed or at best were only partially successful. It was not till 1835 that C. Sturm produced a surprisingly simple algorithm that computes h exactly. It is remarkable that Sturm's algorithm never attempts to find any roots and does not use any floating point arithmetic at all.

Exercise Let U be the function that is undefined for all strings. Write code in Java (or any other high level language) that computes U according to the definition given above.

Exercise We could object to the statement that a non-terminating program does not produce a result since we could imagine a mechanism whereby a program produces output and does not terminate. Convince yourself that this does not destroy the essence of our last argument. (Think carefully what we could really mean by a non-terminating program producing 'a result'.)

EXERCISE Sturm's algorithm is simple but very hard to discover. Try to find an algorithm for yourself but do not spend too much time on this! For a description of Sturm's algorithm see D. E. Knuth, *Seminumerical Algorithms*, (Second Edition), Addison-Wesley (1981).

§1.4. The Halting Problem. We have seen that non-termination is an inherent feature of any sufficiently general system of computing. However we could try to get round non-termination by finding a program H that takes arguments m, n (which, as observed above, can easily be encoded as a single argument) and returns TRUE if P_m halts on input n, otherwise it returns FALSE (here we use boolean identifiers since they are easier to remember but of course any two distinct values will do the job). Note that H itself should halt on all inputs; it would hardly serve the intended purpose otherwise. Surely such an H would be just as good as the original aim of guaranteed termination. Unfortunately it takes only a little thought to modify our previous construction to:

```
if $H(n,n)$ then loop forever
else halt (and return 0)
```

(The key point about the else part is that we halt; the returned value is of no interest.) Surely our system can express this description as a program P_m, say. But now consider the behaviour of P_m on input m. If it halts then $H(m,m)$ returns TRUE and the description of P_m says 'loop forever'! Thus P_m cannot halt on m but then $H(m,m)$ returns FALSE in which case the description of P_m says 'halt'! Thus we are once again enmeshed in a contradiction. The only way out is to abandon the assumption that H exists.

§1.5. Diagonalization. The two arguments we have presented above in §§1.3, 1.4 have the same flavour. Consider an infinite matrix with rows indexed by programs (or equivalently their indices) and columns by inputs (which we have encoded as natural numbers). At entry (m,n) we place $P_m(n)$ which we interpret as the result of running P_m on input n, i.e., the output in case of termination and a special symbol '$\perp$' (which we assume is not in A) in case of non-termination. (Note that we are not claiming to be able to determine whether $P_m(n)$ stands for a string or the special symbol; just that these are the only possibilities.) The infinite matrix looks like this:

$$\begin{array}{c|cccc}
 & 0 & 1 & 2 & \dots \\
\hline
P_0 & P_0(0) & P_0(1) & P_0(2) & \dots \\
P_1 & P_1(0) & P_1(1) & P_1(2) & \dots \\
P_2 & P_2(0) & P_2(1) & P_2(2) & \dots \\
\vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

This represents all possible situations (at the level of input and outcome) in our programming system. The idea behind each of the two arguments is to show that an assumption about a general system of programming cannot hold by using the assumption to build a program that disagrees with each possible program on at least one input: in this case we have a contradiction. We might be tempted to use a description such as

let P be a program that disagrees with each P_m on at least one input

However this is highly unsatisfactory; how do we know that there is such a P (under the assumption)? We must demonstrate beyond doubt that there is a program that meets our requirements. Looking at the matrix we see that our aim will be achieved if we construct a program that disagrees with all the entries in the main diagonal, i.e., $P_0(0), P_1(1), P_2(2), \ldots$. The key point is that in each case we achieve our aim by describing a construction that clearly corresponds to a program. This process of *diagonalization* is very powerful and has many uses in computing; if you go on to study *Computational Complexity* in CS4 you will meet the same process applied to the relative power of resource bounded computing.

It is a remarkable fact that diagonalization was invented by Georg Cantor (long before theories of computing) as part of his study of infinite sets in a series of papers in the last quarter of the 19^{th} century. We will proceed to describe two very beautiful results arising from this study.

It was known for a long time (at least since Galileo) that infinite sets have rather unusual behaviour. Consider the notion of counting; it can be extended from the finite to the infinite by saying that two sets have the same size (or *cardinality*) if we can find a 1-1 correspondence between their elements, in other words a bijective function from one set to the other. Thus the finite sets $\{0, 1, 2\}$ and $\{a, b, c\}$ have the same cardinality since we have the 1-1 correspondence $0 \leftrightarrow a$, $1 \leftrightarrow b$, $2 \leftrightarrow c$ (amongst many others). Now consider the set of all integers and the set of all even integers, the latter is a proper subset of the former and yet we have the 1-1 correspondence:

$$\begin{array}{ccccccc} \ldots & -2 & -1 & 0 & 1 & 2 & \ldots \\ \ldots & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \updownarrow & \ldots \\ \ldots & -4 & -2 & 0 & 2 & 4 & \ldots \end{array}$$

The general rule is given by $n \leftrightarrow 2n$. As another example we show that the set of all real numbers has the same cardinality as the open interval $(0, 1)$, i.e., all real numbers r such that $0 < r < 1$. We give a pictorial sketch of a 1-1 correspondence. First of all we introduce a step that is not necessary but leads to symmetric pictures: the interval $(0, 1)$ is clearly in 1-1 correspondence with the interval $(-1/2, 1/2)$ by $x \mapsto x - 1/2$. Now consider the parabola $y = x^2$ with

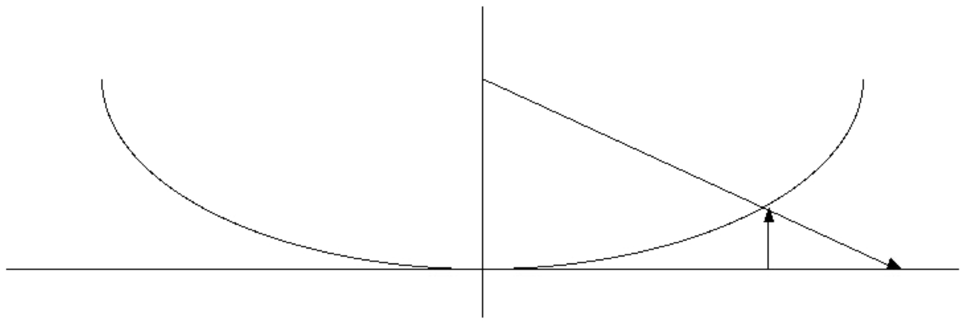

Figure 2: The 1-1 correspondence between $(0, 1)$ and the reals.

$x \in (-1/2, 1/2)$, i.e, consider the set of points $\{ (x, x^2) \mid x \in (-1/2, 1/2) \}$ in the plane. This is clearly in 1-1 correspondence with $(-1/2, 1/2)$ via $x \mapsto (x, x^2)$. Now we put the (portion of) the parabola in 1-1 correspondence with the reals: given any point P on the parabola, draw a straight line through P and the point $(0, 1/4)$ on the y-axis. This line meets the x-axis in exactly one point and the x-coordinate of this gives the real number corresponding to P. Putting the two correspondences together we obtain the claimed correspondence between the open interval $(0, 1)$ and the real numbers. The construction is shown pictorially in Figure 2; geometrically it is given by two projections.

Exercise Let L be the straight line passing through the points $(0, 1/4)$ and (s, s^2). Verify that L is given by the equation $x - 4sy/(4s^2 - 1) + s/(4s^2 - 1) = 0$. Deduce that $r \mapsto (2r - 1)/8r(1 - r)$ gives the 1-1 correspondence described above between the open interval $(0, 1)$ and the real numbers.

The examples discussed above are just two of many. It therefore seems possible that all infinite sets have the same cardinality. In fact this is not so as Cantor showed. We proceed to describe his proof that the set of real numbers has strictly larger cardinality than the set of natural numbers. Certainly the cardinality of the reals is at least as large as that of the natural numbers since the latter are a subset of the former. Now suppose that the claim is false so that there is a 1-1 correspondence between the two sets. Using the second example above we deduce that there must therefore be a 1-1 correspondence between the natural numbers and the open interval $(0, 1)$ of real numbers. Thus there is a list $\alpha_0, \alpha_1, \alpha_2, \ldots$ that enumerates *all* the numbers in $(0, 1)$ with each number occurring exactly once. Now each real number in $(0, 1)$ can be represented uniquely as an infinite decimal (recurring 0 being allowed) provided we forbid recurring 9 (recall that $0.999\ldots = 1$ so that a tail of recurring 9 can be replaced by 1 followed by recurring 0). So $\alpha_i = 0.\alpha_{i0}\alpha_{i1}\alpha_{i2}\ldots$ for $i \geq 0$. We proceed to construct a real number δ in $(0, 1)$ that differs from each α_i; since this is a contradiction the claim follows. Ignoring the leading 0 we write out the decimal digits of our list $\alpha_0, \alpha_1, \alpha_2, \ldots$ as follows:

$$\begin{array}{c|cccc} & 0 & 1 & 2 & \ldots \\ \hline \alpha_0 & \alpha_{00} & \alpha_{01} & \alpha_{02} & \ldots \\ \alpha_1 & \alpha_{10} & \alpha_{11} & \alpha_{12} & \ldots \\ \alpha_2 & \alpha_{20} & \alpha_{21} & \alpha_{22} & \ldots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Now we define

$$\delta_i = \begin{cases} 1, & \text{if } \alpha_{ii} \neq 1; \\ 2, & \text{if } \alpha_{ii} = 1. \end{cases}$$

Clearly the number $0.\delta_0\delta_1\delta_2\ldots$ is in $(0, 1)$ but is different from each α_i (here we have used the fact that if two of our decimal representations look different then they do indeed represent different numbers). This completes the proof of the claim.

Exercise Suppose that in the proof above we defined δ_i by

$$\delta_i = \begin{cases} 0, & \text{if } \alpha_{ii} \neq 0; \\ 1, & \text{if } \alpha_{ii} = 0. \end{cases}$$

This does the job just as well but we have to check one extra thing. Explain what that is and complete the proof.

Another of Cantor's results concerns the cardinality of the power set, $\mathcal{P}(X)$, of a set X. Recall that $\mathcal{P}(X) = \{\, Y \mid Y \subseteq X \,\}$, i.e., the set of all subsets of X. Cantor showed that there cannot be a function from X *onto* $\mathcal{P}(X)$ and so the cardinality of $\mathcal{P}(X)$ is strictly larger than that of X. His proof is again by contradiction: suppose there is a function $f : X \to \mathcal{P}(X)$ that is onto, i.e., for every $Y \in \mathcal{P}(X)$ there is a $y \in X$ such that $Y = f(y)$. Now, bearing in mind that $f(x)$ is a subset of X for each $x \in X$, consider the set $A = \{\, x \in X \mid x \notin f(x) \,\}$. Since A is a subset of X there must be an $a \in X$ such that $A = f(a)$. But now we have a contradiction since, by the definition of A, we have $a \in A$ if and only if $a \notin f(a)$, i.e., if and only if $a \notin A$! This contradiction shows that f does not exist.

Exercise A function $f : \mathbb{N} \to \mathbb{N}$ is non-decreasing if $f(0) \leq f(1) \leq f(2) \leq \ldots$, i.e., $f(n) \leq f(n + 1)$ for all $n \in \mathbb{N}$. Use Cantor's second result to prove that there are uncomputable such functions. (Hint: how many computable functions are there?)

§1.6. Paradise lost. Cantor's work on sets was based on the plausible assumption that if P is any property then there is a set consisting precisely of all those objects x such that P is true. Unfortunately this is too liberal and leads to contradictions as Bertrand Russell showed. He started from the following fact about sets: if A is any set and x is an object (of any kind, e.g., a set) then either $x \in A$ or $x \notin A$. So let us consider the set

$$R = \{\, x \mid x \text{ is a set and } x \notin x \,\}.$$

Since R is a set (if we follow Cantor's assumption) we can ask if $R \in R$. But now we are in trouble for by definition $R \in R$ if and only if $R \notin R$. (An alternative formulation is to consider catalogues; some list themselves and some do not. Try to build a catalogue of all catalogues that do not list themselves.)

It is clear, and somewhat ironic, that Russell's paradox (published in 1903) is based quite closely on Cantor's work. The problem lies in the self reference used; however it would be an over-reaction to conclude that all forms of self reference must be excluded. Russell's proposed solution was his 'Ramified Theory of Types' which allowed a controlled form of self reference. However this was quite complicated and very difficult to use. There are now various systems of axioms for set theory; the most widely used is due to Zermelo-Fraenkel. (We should comment here that the set A defined in Cantor's proof that the cardinality of $\mathcal{P}(X)$ is strictly larger than that of X is perfectly above board; it is justified by the 'Axiom of Subset Selection' of Zermelo-Fraenkel.) In fact most people work with sets at the intuitive level that Cantor took for granted, resorting to a formal theory only in tricky situations. Russell's paradox shows beyond doubt that we need to take great care, e.g., by having a formal account—this might still lead to contradictions but so far everything seems okay for the Zermelo-Fraenkel system.

§1.7. Truth and formal proof. The dream of Russell and many others (e.g., the very great mathematician David Hilbert) in the early part of the 20th century was to provide a formal system which could be used for all of Mathematics, at least in principle. The daunting *Principia Mathematica* by Russell and Whitehead was one example of these efforts. It came as quite a shock when in 1931 Kurt Gödel showed that in any consistent formal system capable of expressing arithmetic there are statements that are true but not provable. The heart of Gödel's proof is once again a very simple idea based on self reference. Consider the statement

$$S = \text{'This sentence is unprovable.'}$$

Now in a consistent system we can only prove true statements. Therefore the sentence is unprovable in which case it is true. This appears to be a contradiction since we seem to have just proved that the sentence is true! However once again we must state things with greater care: first of all what do we mean by 'proof'? One way to supply a satisfactory answer to this is to introduce a system of deduction, as Russell and others did. ‘Proof’ is then defined within such a system, let us call it D. The statement S now becomes

$$S_D = \text{‘This sentence is unprovable in system } D\text{.’}$$

However this is still unsatisfactory since we can hardly ask system D to deal with things that it cannot express. Therefore we don’t have anything more than an intriguing possibility until we can show that S_D can itself be expressed in system D. Gödel realized that the statements of a system D can be encoded as integers (now called Gödel numberings, cf. encodings of programs). He then showed that, for each natural number n, the sentence

$$S_{D,n} = \text{‘The statement in the system } D \text{ whose number is } n \text{ is unprovable in } D\text{’}$$

can be expressed in any formal system D that is powerful enough to express arithmetic. It can then be shown that there is a number m such that the Gödel number of $S_{D,m}$ is precisely m. We see immediately that we have a true statement in D, namely $S_{D,m}$, that is unprovable in D.

A detailed proof is fairly technical as might be expected but a knowledge of methods and results from computability leads to a simplified proof.

§1.8. Formal models of computing. We require the following properties from a formal model of computing.

1. Computation within the model should proceed by a sequence of steps, each step being entirely mechanical. We want the model to be, at least in principle, physically realisable.

2. The model should support the computation of all things that we intuitively believe to be computable. This requirement rules out finite state machines, which cannot perform such reasonable tasks as recognizing when a binary sequence is a palindrome, or when a sequence of parentheses is properly nested.

3. The model should be simple, so that a ‘theory of computation’ can be developed without unnecessary complications. This requirement rules out the use of a real computer as the model, a choice which would in other respects be quite attractive.

As noted in §1.1 a model of computation which appears to possess these three properties was proposed by Alan Turing in 1936, and now bears his name.

Acknowledgement. Most of these lecture notes were originally written by Mark Jerrum. This one, and NOTE 14, were written by Kyriakos Kalorkoti. Subsequent changes are due to Don Sannella.