Chapter 1

Brief Introduction to Complexity Theory

Complexity theory is the rigorous analysis of the computational resources (space, time, ...) required to solve a problem. More precisely, it concerns the *scaling* of the resources as a function of the size of the problem. This is a rich and fascinating area of late-twentieth science, at the interface between mathematics and computer science. There are many examples of deep complexity-theoretic results that give us highly non-trivial insights into the nature of computation. Alas, all beyond the scope of this brief introduction!

This is *not* a course on complexity theory. We will briefly survey the basic definitions and concepts (which are all that are needed for this course), and then move on as quickly as possible to applying these concepts to quantum many-body physics.

But, lest you get the impression that complexity theory is nothing more than definitions and classification, take one of the modern textbooks on the subject out of the library [AB09; Pap03], read it, and this impression will quickly be dispelled. (And you will have learned some fascinating mathematics to boot!)

1.1 Computational Problems

Before we can talk about the computational complexity of problems, we need to know what a computational problem *is*! In a moment, we will give a precise definition. But it is instructive to first see some examples of how well-known mathematical questions can be formulated as computational problems.

Problem 1 (Factoring)
Input: $n \in \mathbb{N}$
Output: *Factors of* n

© 2014-2015 Toby Cubitt

Problem 2 (Travelling salesman)
Input: *Weighted graph G*
Output: *Hamiltonian cycle with minimum weight.*

Problem 3 (SAT)
Input: *Boolean expression made up of $\wedge$, $\vee$, $\neg$ over boolean variables $\{b_i\}$*
Output: *YES if $\exists$ variable assignment s.t. expression evaluates to "true"; otherwise NO.*

The SAT problem is an example of a decision problem: it has a binary YES/NO answer. Decision problems are less restrictive than they might at first appear. Many computational problems have equivalent* formulations as decision problems, e.g:

Problem 4 (Factoring (decision variant))
Input: *$n, k \in \mathbb{N}$*
Output: *YES if n has a factor $< k$; otherwise NO.*

Exercise 1 *Show that if you can solve the Factoring decision problem, you can also find the factors. Give a decision variant of the Travelling Salesman problem, and show that solving this lets you find the lowest-weight Hamiltonian cycle.*

The general definition of a decision problem is exactly what you'd expect from these examples:

Definition 5 (Decision problem) *A* decision problem *is a binary function $f : X \mapsto \{0, 1\}$ on a countably-infinite set X. Each input $x \in X$ defines a different* instance *of the problem.*

Note that a computational problem by definition consists of a countably-infinite family of questions, or *instances*. Each integer gives a different instance of the Factoring problem; each weighted graph gives a different instance of the Travelling Salesman problem; each boolean expression gives a different instance of the SAT problem. Since we are interested in the *scaling* of resources with problem size, it makes no sense in complexity theory to talk about the complexity of a single instance of a problem.

Strictly speaking, the size of a problem instance is the number of bits of information required to specify that instance. However, typically there is some natural measure of problem size which serves just as well. E.g. the number of digits in n for the Factoring problem, the number of vertices in the

*See Section 1.4 for a rigorous definition of "equivalent".

© 2014-2015 Toby Cubitt

graph for the Travelling Salesman problem, or the number of variables in a SAT problem. Where the number of bits of information needed to specify an instance scales polynomially in some more natural measure of problem size, it is common practice to take that to define the size of the problem.

In principle, we must be careful when talking about decision problems involving real numbers. The reals are not a countable set, so to have a well-defined computational problem we must restrict the real numbers involved to finite precision. Equivalently, we can define the problem over a dense subset of the reals, such as the rationals. The problem size therefore depends on the precision to which the input is given.

In practice, this rarely poses much difficulty. It is common practice to talk about computational problems involving real numbers, leaving implicit the fact that the problem should strictly speaking be discretised. Usually, the results go through (just with more notation and more tedious technicalities!) when one explicitly takes the discretisation into account.

Sometimes it makes sense to restrict a decision problem to a particular (but still countably-infinite) subset of instances. Think of this as if the person asking the question has promised that all the instances they will ask you about have some particular property. The promise can substantially change the difficulty of the problem, e.g. if we promise that the number will always be even, the Factoring decision problem becomes trivial to solve!

Computational problems in which the input is restricted to some subset are called *promise problems*. They give a useful generalisation of decision problems.

Definition 6 (Promise problem) *A promise problem is a partial binary function $f : X \mapsto \{0,1\}$ on a countably-infinite subset $X \in \{0,1\}^*$ of bit-strings, where the domain of f is restricted to a countably-infinite subset $S \subseteq X$.*

The promise is that the input will always come from S, the domain of f. Clearly, a decision problem can be viewed as a promise problem with the trivial promise $S = X$.

When solving a promise problem, we are only required to solve the problem under the assumption that the promise holds. We are *not* required to *check* the promise. Indeed, if we are given an instance for which the promise does not hold, we are allowed to output either answer, or even no answer at all.

1.2 Computation

Now that we have a precise definition of computational problems, we need to know what computation is. There are many different abstract models

© 2014-2015 Toby Cubitt

of computation: Turing machines, lambda calculus, recursive functions, circuits, cellular automata.... However, one of the foundational principles of theoretical computer science is that:

Thesis 1 (Church-Turing Thesis) *All reasonable models of computation are equivalent.*

If we restated the Church-Turing thesis is in terms of a particular model of computation (e.g. "all reasonable models of computation are equivalent to Turing machines") it would become* a definition of "reasonable model of computation". (Though this perhaps misses part of the point of the thesis.) In any case, for us the Church-Turing thesis serves as justification for picking the most convenient model of computation, safe in the knowledge that all the results we prove will apply equally well to any other model of computation.

We will describe computation in the *circuit model*. The circuit model is convenient both because it is intuitively simple† and—importantly for us—it has a straightforward generalisation to quantum computation.

1.2.1 Classical computation

Definition 7 (Logic gate) *A logic gate is a boolean function on one or two bits:* $G_1 : \{0,1\} \mapsto \{0,1\}$ *or* $G_2 : \{0,1\} \times \{0,1\} \mapsto \{0,1\}$.

The familiar boolean AND, OR and NOT operations are good examples of logic gates:

$$AND(x,y) = \begin{cases} 1 & x = y = 1 \\ 0 & \text{otherwise} \end{cases}, \qquad OR(x,y) = \begin{cases} 1 & x = 1 \text{ or } y = 1 \\ 0 & \text{otherwise} \end{cases}, \tag{1.1}$$

$$NOT(x) = \begin{cases} 0 & x = 1 \\ 1 & x = 0. \end{cases} \tag{1.2}$$

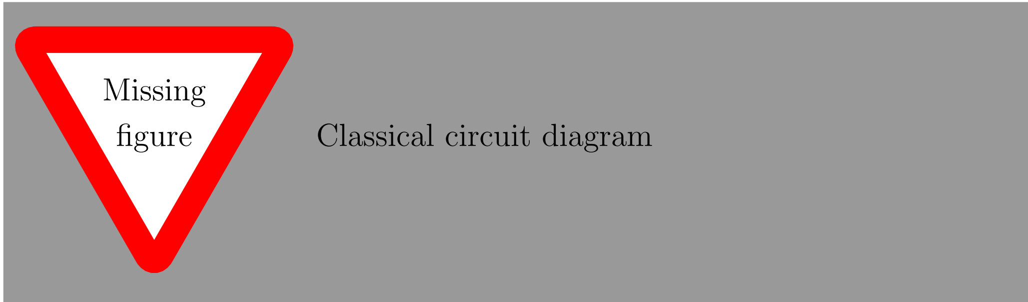
(It's a well known fact that these gates are universal: any boolean function can be computed by circuits made up of AND, OR and NOT.)

A circuit is a finite sequence of gates, taking n bits as input and producing m bits as output. Formally,

Definition 8 (Classical circuit) *A classical boolean circuit from n bits to m bits is a directed acyclic graph, with n vertices of in-degree 0, m vertices of out-degree 0, and all other vertices have out-degree 1 and in-degree 1 or 2. Each vertex with non-0 in-degree is labelled by a logic gate, where in-degree 1 (2) vertices are labelled by gates on 1-bit (2-bits).*

*Together with a precise definition of "equivalent".
†Though this apparent simplicity hides some irritating subtleties.

© 2014-2015 Toby Cubitt



1.2.2 Quantum computation

Definition 9 (Quantum gate) *A quantum gate is a unitary operator on one or two qubits:* $U_1 : \mathbb{C}^2 \mapsto \mathbb{C}^2$ *or* $U_2 : \mathbb{C}^2 \otimes \mathbb{C}^2 \mapsto \mathbb{C}^2 \otimes \mathbb{C}^2$.

Examples include:

$$U_{\text{CNOT}} = |00\rangle\langle 00| + |01\rangle\langle 01| + |11\rangle\langle 10| + |10\rangle\langle 11| \equiv \begin{pmatrix} 1 & & & \\ & 1 & & \\ & & 0 & 1 \\ & & 1 & 0 \end{pmatrix} \tag{1.3}$$

$$\begin{aligned} Z &= |0\rangle\langle 0| - |1\rangle\langle 1| \equiv \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \\ X &= |0\rangle\langle 1| + |1\rangle\langle 0| \equiv \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \\ Y &= i\,|1\rangle\langle 0| - i\,|0\rangle\langle 1| \equiv \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix} \\ H &= \frac{1}{\sqrt{2}}(X + Z) \equiv \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \\ T &= |0\rangle\langle 0| + e^{i\pi/4}\,|1\rangle\langle 1| \equiv \begin{pmatrix} 1 & 0 \\ 0 & e^{i\pi/8} \end{pmatrix} \end{aligned} \tag{1.4}$$

(In fact, though we will not prove it here, these form a universal quantum gate set, sometimes called the standard gate set. Any unitary can be approximated to arbitrary accuracy by quantum circuits made up of these gates.)

Note that unitarity of quantum gates implies that they are necessarily *reversible*: $U^\dagger U = \mathbb{1}$ so any gate U has an inverse gate $U^\dagger$. E.g. both the CNOT gate and Z gate are self-inverses.*

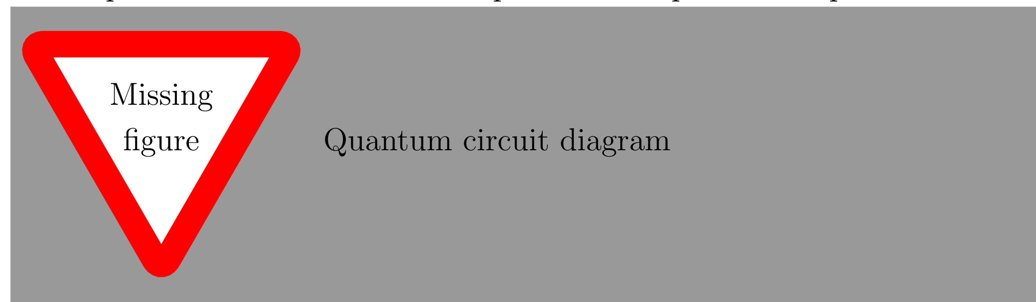
*We will not show this here, but it is also possible to perform all classical computation reversibly, implying that classical computation is a special case of quantum computation.

© 2014-2015 Toby Cubitt

A quantum circuit is a finite sequence of quantum gates, each acting on a specified subset of qubits. Formally:

Definition 10 (Quantum circuit) *A quantum circuit from n qubits to m qubits is a directed acyclic graph, with n vertices of in-degree 0, m vertices of out-degree 0, and all other vertices have in-degree 1 or 2 and out-degree equal to in-degree. Each vertex with non-0 in-degree is labelled by a quantum gate, where degree 1 (2) vertices are labelled by 1-qubit (2-qubit) quantum gates.*

The input $|\psi\rangle$ to a quantum circuit is by definition a *computational basis state*, i.e. $|\psi\rangle = \bigotimes_i |x_i\rangle$, $|x_i\rangle \in \{|0\rangle, |1\rangle\}$. The output is obtained by measuring the qubits in the computational basis, i.e. the outcome of performing the projective measurement $\{\Pi^{(0)}, \Pi^{(1)}\}$ on each qubit. Note that this implies that the outcome of a quantum computation is probabilistic.*



1.3 Complexity Classes

Now that we have defined computational problems and computation, we are in a position to introduce the main topic of complexity theory. Complexity classes classify computational problems according to the scaling of the computational resources required to solve them.

Literally hundreds of different complexity classes have been defined [Com]. Fortunately, we will only be concerned with the four most important classes in classical and quantum complexity theory.

1.3.1 Classical complexity classes

We first define two classical complexity classes. Indeed, these are the most important and most studied classes in complexity theory.

*One can also define probabilistic classical computation… but we will not need to for the purposes of this course.

© 2014-2015 Toby Cubitt

Definition 11 (Polynomial-time (P)) *P is the class of all decision problems for which there exists a family of classical circuits C_n s.t. $C_n(x) = 1$ iff x is a YES-instance of size n, where C_n has size $O(\text{poly}(n))$.*

In a mild but standard abuse of terminology, we will often refer to a family of circuits such as the one involved in Definition 11 as a *polynomial-sized circuit*. Since we are defining computation in the circuit model, we will use "time" and "circuit size" interchangeably. When we say a problem can be solved in polynomial-time, we mean that the problem is contained in P, and thus can be solved using a polynomial-sized circuit.

Many mathematical tasks (strictly speaking their decision variants) are in P. For example basic arithmetic, computing eigenvalues, inverting a matrix, testing whether an integer is prime (the latter was only proven in 2002! [AKS04]).

In some sense, the class P captures the problems we can solve on a (classical) computer. If a problem is in P, we say that it can be solved *efficiently*. This is a somewhat crude definition of "efficient" from a practical perspective; a problem that takes time n^{100} to solve is probably not solvable even on a supercomputer in any practical sense! Whereas a problem that takes time $2^{n/100}$ may be solvable in practice even for moderately large instances, but isn't efficiently solvable in the complexity-theoretic sense. The justification for ignoring polynomials in our definition of computational efficiency is the following strengthening of the Church Turing Thesis 1:

Thesis 2 (Strong Church-Turing Thesis) *All reasonable models of computation have the same efficiency up to polynomial overhead.*

Definition 12 (Non-deterministic polynomial-time (NP)) *NP is the class of all decision problems for which there exists a family of polynomial-sized circuits C_n s.t. if x is a*

YES-instance: $\exists$ polynomial-sized witness $w \in \{0,1\}^{\text{poly}(n)}$ *s.t. $C_n(x, w) = 1$;*

NO-instance: $\forall$ witnesses w, $C_n(x, w) = 0$.

It is helpful to think of NP as a game between all-powerful (but untrustworthy!) Merlin, and less powerful Arthur who (poor lad) can only run polynomial-time computations. Arthur asks Merlin a yes/no question x, and Merlin replies with his answer. But Arthur doesn't trust Merlin, so Merlin also gives Arthur a simple (poly-time checkable) proof w that the answer is correct. NP is the class of all problems for which Merlin can convince Arthur of a YES answer ($\exists w$), but cannot trick him into believing an incorrect NO answer ($\forall w$).

© 2014-2015 Toby Cubitt

Clearly, any problem in P is also in NP (Arthur can simply compute the answer for himself, ignoring the witness, and compare his answer with Merlin's). But NP also contains many mathematical tasks that are believed to be difficult, such as the Factoring, Travelling Salesman and SAT problems.

Exercise 2 *Convince yourself that these three problems are in NP.*

Note that NP does *not* stand for "non-polynomial", it stands for "non-deterministic polynomial-time". Other than knowing that NP does not mean "non-polynomial", it's best to forget what NP stands for. The name is historical, and comes from an earlier (equivalent) definition of NP which was much less clear. The modern definition of NP, given above, has largely obsoleted the original definition. Also note the asymmetry between YES and NO in the definition of NP. There is a class called co-NP which is defined very similarly, but with the roles of YES and NO swapped. However, we will not have any use for co-NP in this course.

1.3.2 Quantum complexity classes

If we allow quantum circuits to be used instead of classical circuits in the definitions of P and NP, we get quantum versions of these two classes. However, we have seen that quantum computation is inherently probabilistic, so we only demand correct answers with high probability.

We first define the quantum version of P:

Definition 13 (Bounded-error quantum polynomial-time (BQP)) *BQP is the class of all decision problems for which there exists a family of polynomial-sized quantum circuits U_n s.t. for instance x of size n*

$$\Pr(U_n \textit{ outputs "1" on input } |x\rangle) \begin{cases} \geq \frac{2}{3} & \textit{YES instance} \\ \leq \frac{1}{3} & \textit{NO instance.} \end{cases} \tag{1.5}$$

Note that an instance of a BQP problem is still specified by *classical* data (which we can w.l.o.g. take to be a length-n bit-string x). The input to the quantum circuit is therefore a computational basis state $|x\rangle = \bigotimes_i |x_i\rangle$, as required by the definition of quantum computation.

The probabilities $\frac{2}{3}, \frac{1}{3}$ are conventional but arbitrary. We could have chosen $1-\epsilon, \epsilon$ for any constant ϵ, or even any $\epsilon = \Omega(1/\operatorname{poly}(n))$.

Exercise 3 (BQP amplification) *Prove that the class BQP is independent of the choice of probabilities $1-\epsilon, \epsilon$ as long as $\epsilon = \Omega(1/\operatorname{poly}(n))$.*

© 2014-2015 Toby Cubitt

This may seem to be giving quantum computation an unfair advantage; we demanded exact answers in our classical complexity classes, whereas we only demanded answers with high probability in our quantum complexity classes. From a mathematical perspective, this is indeed an important distinction: the standard quantum complexity classes are really generalisations of the classical *probabilistic* versions of P and NP, which we haven't seen. Whether the classical probabilistic complexity classes are strictly larger than their deterministic counterparts (derandomisation) is a major open problem in complexity theory.

It's worth noting that, from a purely practical perspective, the distinction between probabilistic and deterministic computation is largely irrelevant. We can never hope to obtain a correct answer with probability higher than the probability that a component of our computer makes an error.

Just as P captures the problems we can solve efficiently on a classical computer, the class BQP in some sense captures those problems we can solve on a *quantum* computer. The strong Church-Turing thesis is under threat! If BQP≠P, then there exist problems that can be solved efficiently on a quantum computer, which cannot be solved efficiently by any model of classical computation. And, since nature itself appears to obey the laws of quantum mechanics, quantum computation is certainly a reasonable model of computation (notwithstanding spirited attempts by some scientists to deny this!).

By now, you can probably guess the definition of the quantum analogue of NP (again, the probabilities $\frac{2}{3}, \frac{1}{3}$ are arbitrary conventions):

Definition 14 (Quantum Merlin-Arthur (QMA)) *QMA is the class of all decision problems for which there exists a family of polynomial-sized quantum verified circuits U_n s.t. if x is a*

YES-instance: $\exists$ *polynomial-sized quantum witness* $|w\rangle \in \mathbb{C}^{\mathrm{poly}(n)}$ *s.t.* $\Pr(U_n$ *outputs "1" on input* $|x\rangle|w\rangle) \geq \frac{2}{3}$*;*

NO-instance: $\forall |w\rangle \in \mathbb{C}^{\mathrm{poly}(n)}$, $\Pr(U_n$ *outputs "1" on input* $|x\rangle|w\rangle \leq \frac{1}{3}$.

All of these complexity classes—P, NP, BQP and QMA—are decision classes: by definition they only contain decision problems. By allowing promise problems in the definitions, we immediately get four further complexity classes, called promise-P, promise-NP, promise-BQP and promise-QMA. Other than this one small change to the definition, there is little conceptual difference between the decision classes and their promise-versions. Indeed, some complexity theorists feel that the original definitions were wrong, and should have included promise-problems all along. We will therefore ignore this technicality, and freely talk about promise problems being in members of P, NP, BQP and QMA, when strictly speaking they are members of the promise versions of these classes.

© 2014-2015 Toby Cubitt

1.4 Reduction

So far, all we have done is define and categorise computational problems. The concept of reduction lets us compare the difficulty of different computational problems. This allows us to rigorously say that one problem is easier or harder than another. With this, we can go beyond mere cataloguing, and prove interesting results about computational problems and complexity classes.

There are a number of different types of reduction in complexity theory. We will only need the most common one:

Definition 15 (Polynomial-time many-one reduction) *A decision problem A* reduces to *B (denoted $A \leq B$) if $\exists$ map $R : A \mapsto B$ which maps instances a of A to instances $b = R(a)$ of B, s.t. b is a YES-instance iff b is a YES-instance, and R can be computed by a polynomial-sized circuit.*

Polynomial-time many-one reduction is sometimes called *Karp reduction.* It is also commonly referred to simply as "polynomial-time reduction".

If problem A reduces to problem B, then we can solve any instance of A by first transforming it into B (which we can do efficiently since we can compute R in polynomial-time), and solving the resulting instance of B. An efficient method of solving B therefore immediately gives us an efficient method of solving A. So, in a precise sense, problem A is no harder than problem B and, conversely, problem B is at least as hard as problem A.

Reduction defines a partial order on decision problems.

Definition 16 (Polynomial-time equivalence) *If $A \leq B$ and $B \leq A$, then we say that A and B are* equivalent *(denoted $A = B$).*

Polynomial-time reduction allows us to define the final complexity classes we will need, which are derived from the classes NP and QMA.

Definition 17 (NP-hard) *A computational problem B is* NP-hard *if $\forall A \in NP : A \leq B$.*

Definition 18 (NP-complete) *A decision problem B is* NP-complete *if $A \in NP \cap NP\text{-hard}$.*

NP-hard problems are—in the precise sense defined by reduction—at least as hard as any problem in NP. Solving any one of them would allow us to solve *every* problem in NP. Similarly, NP-complete problems are the hardest problems in NP. SAT is an example of an NP-complete problem.

QMA-hard and QMA-complete are defined in the obvious analogous way:

© 2014-2015 Toby Cubitt

Definition 19 (QMA-hard) *A computational problem B is* QMA-hard *if* $\forall A \in QMA : A \leq B$.

Definition 20 (QMA-complete) *A decision problem B is* NP-complete *if* $A \in QMA \cap QMA\text{-}hard$.

1.5 Complexity Zoo

What is the relationship between all the complexity classes we have defined? Some relationships are trivial. We have already seen that P $\subseteq$ NP. Similarly, BQP $\subseteq QMA$ for much the same reasons. Since classical computation is a special case of quantum computation*, we immediately have P $\subseteq$ BQP and NP $\subseteq$ QMA. However, the question of whether these inclusions are strict encompasses many of the most important open problems in complexity theory!

$P \stackrel{?}{=} NP$: This is the (in)famous P vs. NP problem, one of the most important open problems in mathematics!†

$P \stackrel{?}{=} BQP$: This is one way of capturing precisely the question "are quantum computers useful?". If P = BQP, then all decision problems that are efficiently solvable on a quantum computer could also be solved efficiently on a classical computer.

$BQP \stackrel{?}{=} QMA$: The quantum analogue of P vs. NP.

$NP \stackrel{?}{=} QMA$: This is open, but considered unlikely for reasons beyond the scope of this overview.

$NP \stackrel{?}{\subseteq} BQP$: Open, but considered unlikely.

The general belief is that all of the inclusions between these classes are strict (though it is not hard to find complexity theorists who will disagree with this). However, it could be that they are all equalities, and in fact P = BQP = NP = QMA! There are good mathematical arguments for believing the inclusions are strict. The only problem is, there are equally good arguments for believing the inclusions are in fact equalities.

*If you accept the claim—not proven in this brief overview—that classical computation can w.l.o.g. be made reversible.

†At least according to the Clay Mathematics Institute who included it among their seven Millennium Prize Problems in mathematics, each with a $1,000,000 prize attached.

© 2014-2015 Toby Cubitt

A philosophical argument that is sometimes made for why $\mathrm{P} \neq \mathrm{NP}$ is that it captures the empirical fact that proving a mathematical result can be very difficult, whereas checking correctness of an existing proof is typically much easier. This is not a particularly rigorous argument, but it does capture the flavour of the difference between the classes P and NP.

Perhaps the strongest complexity-theoretic evidence we have for quantum computation being strictly more powerful than classical computation is Shor's seminal result [Sho97] that Factoring is in BQP (whereas it is not known to be in P).* On the other hand, Factoring is one of the handful of NP problems that is not known to be NP-complete.

Another reason we expect quantum computation to be more powerful than classical computation—one very related to this course—is that quantum many-body systems are notoriously difficult to simulate on classical computers. Whereas simulating quantum systems is, unsurprisingly, rather easy on a quantum computer. Indeed, the field of quantum computation was kicked off by Feynman making exactly this suggestion [Fey85].

Strictly speaking, Definition 11 is wrong! It in fact defines a slightly different complexity class known as P/poly, which is strictly larger than P.† The reason for this is that are allowing too much freedom in the family of circuits. As we have defined it, the circuit C_{n+1} for problems of size $n+1$ doesn't have to bear any relation to the circuit C_n for problems of size n. Even though we know a circuit C_n exists for each n, it might be impossible to compute C_n given n.

To make Definition 11 correct, we should restrict to *uniform families* of circuits: families for which C_n can be computed in polynomial-time from n. This definition appears circular. We are defining polynomial-time computation in terms of uniform families, and defining uniform families in terms of polynomial-time computation!

In fact, uniform circuit families are defined to be circuits that can be computed in polynomial-time on a Turing Machine, a model of computation which doesn't suffer from these irritating issues. However, this technicality is almost never an issue in practice. Substituting "uniform circuit" for "circuit" in all the definitions and theorems in this course will make the statements strictly correct.

*If Factoring is proven to be in P, much of the public key cryptography used on the internet would be rendered insecure, as it relies on Factoring being hard.

†Indeed, P/poly contains uncomputable problems!

© 2014-2015 Toby Cubitt