Why Complexity Matters: A Brief Tour

Christopher Granade

April 26, 2009

Abstract

In this paper, complexity theory is presented as a language for describing problems in computer science. This development proceeds by the construction of computational models such as the deterministic Turing machine, and by the construction of complexity classes such as P and NP.

The language of complexity theory is then applied to three key applications: the implications for physical theories which violate the Church-Turing Thesis, the impact of computational complexity upon evaluating physical theories, and the objective benefits of quantum computers.

Contents

I Preliminaries **3**

1 Computability Theory **3**

2 Models of Computation **4**

2.1 Problems, Functions and Languages 4
2.2 Deterministic Finite Automata 5
2.3 Turing Machine 6
2.3.1 Why the Turing machine is descriptive. 7
2.4 Non-deterministic Turing Machine 8
2.5 Boolean Circuit Model 9

II Complexity Theory **10**

3 The Polynomial Hierarchy **10**

3.1 DTIME $(f(n))$ 12
3.2 P 12
3.3 NP and coNP 13
3.4 NP-complete 14
3.5 PH 15
3.6 Beyond PH: E, EXP, NEXP and EEXP 17

4 Space-bounded Classes 17
4.1 PSPACE 17
4.2 L 18

5 Circuit-Model Classes 18
5.1 AC 19
5.2 NC 19

6 Probabilistic Classes 20
6.1 ZPP 20
6.2 BPP 21
6.3 PP 21
6.4 MA and AM 22

7 Quantum Computation 22
7.1 Reversible Classical Computation 22
7.1.1 A Strange Realization 23
7.2 Modeling Quantum Computation 25
7.2.1 The Quantum Circuit Model 25
7.2.2 Universal Gates for QCM 26
7.3 BQP 27

8 Other Classes 27
8.1 P/poly 28
8.2 SZK 28
8.3 #P 29

III Applications and Implications 29

9 Why We Care About Quantum Computers 29
9.1 The Church-Turing Thesis 30
9.2 $\mathsf{P} = \mathsf{BQP}$? 30
9.3 $\mathsf{P} \subsetneq \mathsf{BQP}$? 31
9.4 How Quantum Computers Might Fail 31

10 Complexity of Evaluating Physical Theories 32

11 Physical Theories and Computational Power 34
11.1 Hypercomputers 34
11.2 Fundamental Limits 36
11.3 An Extended Church-Turing Thesis 36

12 Concluding Thoughts 37

IV Appendices 38

A Big-O Notation **38**

B The Pumping Lemma **38**

C Random Access Machines **39**

D Turing Machine Variations **41**
D.1 Multi-track . . . 41
D.2 Bidirectional . . . 41
D.3 Multi-tape . . . 42

References **42**

List of Figures

1 A DFA that accepts strings with an even number of zeros. 6
2 Executing a single step of a Turing machine. 7
3 Boolean circuit calculating MAJORITY for length 3 strings. 9
4 Emulating large fanin with large depth. 10
5 Known inclusions among complexity classes. 11
6 Operation of the Fredkin gate. 24
7 Billiards-ball based switch gate. 24
8 Sample quantum circuits. 26

Part I
Preliminaries

1 Computability Theory

As computers were first being invented, a natural question emerged: what can a computer *do*? In the wake of upsets such as those delivered to Hilbert's program to axiomatize mathematics by Gödel's famous theorems, the limits of what computers could do were not obvious. In time, that simple question blossomed into an entire rich field of mathematics called *computability theory*.

To see how such a seemingly simple question can quickly become confounding, consider the famous Halting Problem. Speaking informally, the Halting Problem (often written HALTING) asks whether a program X will ever halt if run with a description of itself as input. That is, will $X(X)$ return any output, or will it continue to execute forever? Suppose that there exists a program H which answers HALTING; that is, $H(X)$ always halts and returns a value such that $H(X)$ is true if and only if $X(X)$ halts. Then, we can use H to construct a program P which acts in a most peculiar fashion. Given an input X, $P(X)$ will stop if $H(X)$ is false and will enter an infinite loop if $H(X)$ is true. Now,

consider what happens if we execute $P(P)$. Our peculiar program will halt if and only if $H(P)$ is false, which contradicts that $H(P)$ is true if and only if $P(P)$ halts! Since we only assumed that a program solving HALTING exists, we are forced to conclude that no program can ever solve HALTING.

Thus, there exists at least one well-defined mathematical problem which no computer will *ever* be able to solve. Given that, why should there be only one such problem? Of course, in order to make this argument sound, we must develop a vernacular for describing the processes of computation, and we must develop a rigorous model of computation about which we may make precise mathematical statements.

2 Models of Computation

At their most basic, computers are devices which are given some input, manipulate that input and return an output. Thus, any model of computation must somehow capture these dynamics. We present here a few different attempts at modeling these properties in a precise and formal way. Our treatment of this topic shall be incomplete, as many important models of computation are not directly relevant to our goals here. In particular, though models such as push-down automata, context-free grammars, Minsky machines and the many varieties of cellular automata are very interesting and useful in their own right, this article is not intended to be a full compendium of interesting models of computation. Rather, we restrict our focus here to those classes that either are directly used in constructing complexity theoretic objects, or that illustrate concepts needed to do so.

2.1 Problems, Functions and Languages

Before we can intelligently discuss a model of computation, we must first define a bit of terminology so that we may state our goals. Historically, the vocabulary used to describe computation in an abstract way has been borrowed from linguistics. This history follows from one of the earliest attempts to nail down the limitations of computation: interpreting human languages. This programme of research produced one of the earliest taxonomies of computational models, the *Chomsky hierarchy*. Though this hierarchy failed to produce a useful understanding of human languages, it provided very valuable insights into the nature of computation, and its relation to languages. It is this linguistic influence that provides much of the terminology that we now use.

For instance, almost always when we discuss a computation problem, we consider a machine of some kind that takes as input a *string* of symbols drawn from some *alphabet*. An alphabet is some arbitrary set of distinct symbols, and is usually denoted by Σ. In this article, when we do not explicitly say otherwise, we shall presume that we are working with the binary alphabet $\Sigma = \{0, 1\}$. Next, a string is a finite (and possibly zero-length) sequence of symbols drawn from an alphabet. For example, $x = 0110$ is a string over $\Sigma = \{0, 1\}$. We call

a string x with no symbols the *empty* string, and write it as ϵ. The set of all strings over an alphabet Σ is denoted Σ^*, where the superscript star indicates "a finite sequence drawn from."

Most often, we shall be interested in *Boolean functions* of the form $f : \{0,1\}^* \to \{0,1\}$. These functions may be interpreted as defining a property which a particular string can either have or not have. As an example, consider the function:

$$f(x) = \begin{cases} 0 & \text{if } x \text{ has an odd number of 0 symbols} \\ 1 & \text{if } x \text{ has an even number of 0 symbols} \end{cases}$$

It is easy to see that this function is well-defined (has exactly one value for each string x). Informally, $f(x) = 1$ is a "yes," and $f(x) = 0$ is a "no." Often, it will be useful to shift perspective by letting $L_f = \{ x \in \Sigma^* \mid f(x) = 1 \}$ be the *language* of strings for which f gives a "yes" answer. Intuitively, we may think of a language as a set of strings that is allowed by some set of rules. Shifting momentarily to human languages, we may represent English as a set of sentences which are permissible by the rules of English grammar (setting aside the inconvenient fact that defining a single set of deterministic rules is an impossible task in and of itself). By analogy, a Boolean function tells us what strings we permit to be in a given language.

Finally, a *machine* is some device that takes a string, performs some work on the string and either returns a "yes" or "no," or fails to halt. We say that a machine M *accepts* a string x if M returns "yes" when given x as input. If a machine M halts for all inputs, then there is then a very natural Boolean function corresponding to M: does M accept a string x? This function is so useful that we often simply write $M(x)$ to mean the Boolean function corresponding to the machine M. We can extend this definition to machines which may not halt by introducing another symbol $\nearrow$ into our output alphabet and by letting $M(x) = \nearrow$ if M does not halt when given x as input. There thus corresponds to each machine M a language L_M of strings accepted by M, called the language *decided by* M.

With this terminology under our belt, we can now precisely state what we want to be able to do with a computational model. Given a Boolean function f, we wish to construct a machine M such that $L_f = L_M$, and such that M halts on all input. If we have such a machine, we say that M solves L_f, or that M solves f.

2.2 Deterministic Finite Automata

One of the simplest models of a computer is the deterministic finite automata (DFA) model. For a DFA, the internal state of the computer is one of a finite set of discrete states. To each state, there corresponds a set of rules which specify when the DFA should *transition* to another state. A DFA looks through its input, one symbol at a time, transitioning based on the symbol pulled from the input. When the input runs out, either the DFA is in a specially-marked *accepting state*, or it isn't. In the former case, the DFA returns a "yes" answer.

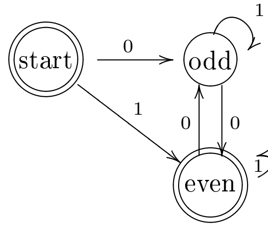

Figure 1: A DFA that accepts strings with an even number of zeros.

A common way of representing a DFA is by means of a *directed graph*, such as the one shown in Figure 1. In such diagrams, the states of a DFA are drawn as circles with arrows between them. Accepting states are shown by double-line borders, while starting states are indicated by special names (such as "start"). The transitions of the DFA are represented by arrows, labeled with the corresponding input symbols.

As a concrete example, let's run the DFA in Figure 1, designed to accept strings with an even number of "0" symbols, with the input "010". The machine starts off at "start", then reads a "0" from the input. It then follows the "0" arrow to the "odd" state. Next, it reads a "1" from the input, and thus follows the "1" arrow from "odd" back to itself. The machine then reads the last "0" from the input, following the "0" arrow to "even". Since the input has been fully read, the machine stops on the "even" state. Since this state is accepting, the machine returns a "yes" answer.

The deterministic finite automata model is useful in that it is relatively simple, but there are some very basic languages that are unsolvable for DFA machines, limiting its usefulness. As an example, it can be shown via a useful result known as the Pumping Lemma (see Theorem 3) that the language of balanced parentheses (that is, the language over $\Sigma = \{(,)\}$ such that each opening paren is matched by exactly one closing paren somewhere later in the string) is unsolvable for DFAs. Since this language can be rather easily decided by modern computers, it is clear that DFAs are not powerful enough to encapsulate what we mean by a computer.

2.3 Turing Machine

To solve these problems, we introduce the Turing Machine (TM) model of computation. At first, Turing machines seem highly unintuitive and seem to have little to do with computers, but as we will see, they are the most powerful model for classical computation that we have yet devised.

The key to Turing machines is that we essentially introduce a *tape* to a DFA, and allow the DFA to write to the tape as it proceeds. For our purposes, a tape is an one-dimensional array of memory cells extending to infinity in one direction. Each of the cells on a tape can store a symbol drawn from a tape alphabet Γ, which must be strictly bigger than the input alphabet Σ.

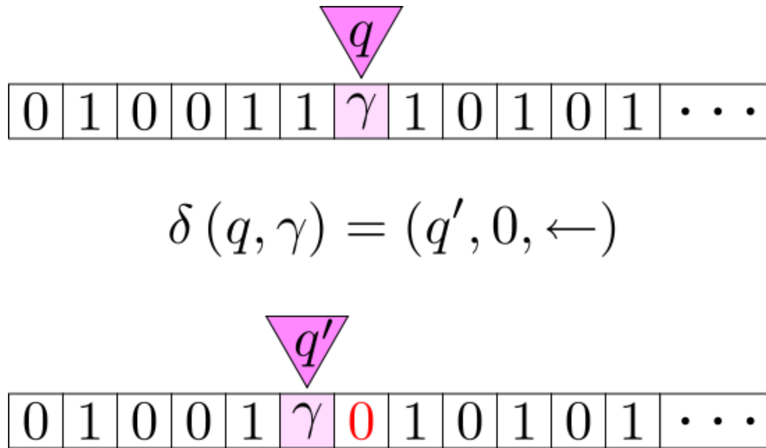

Figure 2: Executing a single step of a Turing machine.

When a Turing machine starts, the tape is initially filled with the input to the machine surrounded by infinitely many "blank cells"; that is, cells holding a special "blank" value which we shall write as $\sqcup$. Next, we imagine that there is a read/write head that points at a single cell on the tape, and which can decide to move either left or right after looking at its cell and possibly writing a new value. To make this decision, the head can be in one of a finite number of *states*, much like our DFAs. The head transitions between states depending on the value of the cell beneath it, and each transition is labeled with a new value to be recorded and a direction to move.

Formally, we say that a Turing machine M is a tuple $M = (Q, \Sigma, \Gamma, \delta)$, where Q is the set of possible states for the machine, Σ is the input alphabet, Γ is the tape alphabet (which does not include the special blank symbol) and δ is the *transition function* $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\}$ which indicates the new state, the new tape symbol to be written and the direction (if any) to move the head. We require that Q include two special states, q_{accept} and q_{reject}. If the machine ever enters either state, it accepts or rejects the input accordingly.

Following our earlier notation, we shall say that a Turing machine M decides a decision problem f if $M(x) = 1$ if and only if $f(x) = 1$. Note that this definition allows M to loop forever if the answer to a decision problem is "no." That is, we only require that M halts if $f(x) = 1$.

2.3.1 Why the Turing machine is descriptive.

At first, the Turing machine may seem overly esoteric and arcane. Why not model computation in a way that more closely resembles the familiar von Neumann architecture? The short answer is that the we can: the dynamics of a Turing machine can be shown to be equivalent to those of any other classical computational model that we are interested in, up to some relatively small

overhead. Moreover, a Turing machine has some very nice properties that make writing proofs about its capabilities much easier. First and foremost, the dynamics of Turing machines are *local*. We only ever need to describe the interaction of the head with its immediate neighbors. Second, a Turing machine can be described by a relatively terse set of information. In order to reproduce a given TM, we need to know how large Q, Σ and Γ are, and we need to know δ. Third, the axioms governing a Turing machine are much easier to describe. In order to describe a von Neumann architecture, we must know what the *instruction set* is, as well as knowing what program is being run. By contrast, all Turing machines differ only in their transition functions and the sizes of their alphabets.

Taking these together, we find that despite their apparent strangeness, Turing machines are well-suited to formal descriptions of computational processes. In practice, the overhead factor is large enough that building a physical Turing machine is an exercise in futility, but this overhead is well-behaved enough that all of our proofs will transform nicely between various computational models, with very few exceptions, leaving us free to choose a model based upon their mathematical appeal rather than the practicality of their physical realizations.

2.4 Non-deterministic Turing Machine

We are also interested in what computations can be quickly *verified*, irrespective of how long they take. To do so, we extend our idea of a Turing machine in a way that is, at least initially, highly unintuitive. Rather than giving our Turing machine a single transition function δ, we imbue each machine with a pair of transition functions δ_0, δ_1, and say that the machine transitions between *sets* of states Q, Q' by $Q' = \{ \delta_0(q), \delta_1(q) \mid q \in Q \}$, and that each computation branch has its own copy of the tape[1]. Machines operating in this manner are called *non-deterministic* Turing machines (NTM). We then say that an NTM M accepts a string x if any set of states Q reached by a transition from the initial state and tape includes a special state Q_{accept}. Note that we can "trace" the accepting branch of M by noting at each transition whether the δ_0 or δ_1 transition function will eventually lead to Q_{accept}. Thus, to verify an accepting computation of M, we need only run a deterministic Turing machine M', choosing to apply either δ_0 or δ_1 at each transition according to some *witness string*. Thus, an NTM is a way of formalizing our idea of what it means to verify a computation, even if many of the details of that computation are unknown to us. Equivalently, non-determinism allows us to model what guesses allow us to perform; an NTM essentially tries each possible branch and selects only those which lead to an accepting state.

[1]Note that this construction of Q' is slightly misleading, as we have said that δ_0 and δ_1 are functions to tuples $(Q \times \Gamma \times \{\leftarrow, \bullet, \rightarrow\})$. For brevity, I have adopted a notation whereby one only considers the member of Q in each tuple when constructing Q'.

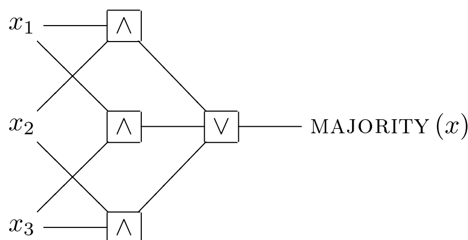

Figure 3: Boolean circuit calculating MAJORITY for length 3 strings.

2.5 Boolean Circuit Model

It is important to note that not all models of computation describe the process of computation as some machine stepping through the steps of an algorithm. Indeed, one can start by looking at the logical operations that must be performed during computation, and define a model of a *circuit* that implements these operations as *gates*. We define a gate to be one of a set of "pre-made" Boolean functions, such as NOT, AND and OR (written $\neg$, $\wedge$ and $\vee$, respectively). These gates are then combined with input *bits* (that is, symbols from the alphabet $\Sigma = \{0, 1\}$) to produce output bits.

For instance, if we want to compute the MAJORITY function (which is 1 if and only if at least half of the input bits are 1) for a string of bits of length 3, we can combine a few AND and OR gates. Let the input string $x = x_1x_2x_3$. Then, MAJORITY $(x) = 1$ if and only if at least two bits are one, and so we can OR together an AND each possible pairing of bits:

$$\text{MAJORITY}(x) = (x_1 \wedge x_2) \vee (x_1 \wedge x_3) \vee (x_2 \wedge x_3)$$

This expression for MAJORITY tells us that the input bits are combined in the respective AND gates to produce three intermediate bits, which are then in turn combined into a final output bit. Of course, it is often easier to represent this kind of an expression as a *directed acyclic graph*, where each gate is drawn as a *node* with *directed edges* from the inputs to the outputs. One such graph is shown in Figure 3.

The graph representation makes the dependence between the gates, inputs and the output more clear, and allows us to quickly judge the relative *size* and *depth* of the circuit. We say that a circuit with n gates has size n, and that the depth of a circuit is the length of the longest path from an input to the output. For example, the circuit for MAJORITY shown above has size 4 and depth 2.

Of course, the MAJORITY circuit shown in Figure 3 only works for strings of length exactly 3, so we still need to somehow extend our model to allow for arbitrary-length input strings. The way to do this is to discuss not individual circuits, but rather *families* of circuits corresponding to different input lengths. In order to prevent certain kinds of absurdities, we will often require that each circuit in a family of circuits be constructable by a Turing machine bounded by

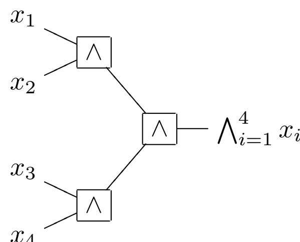

Figure 4: Emulating large fanin with large depth.

some reasonable function of the length. This requirement is called *uniformity*, and will be revisited in Section 8.1.

At times, we may also place limitations on the maximum *fanin* of a Boolean circuit or family thereof. The fanin of a circuit is defined as the largest number of inputs that a gate acts upon. For instance, the circuit in Figure 3 has a fanin of 3. Since it is difficult to make arbitrarily large logic gates, we generally consider low fanin to be less costly in the same way that we consider size and depth (corresponding respectively to space and time resources in a Turing machine) to be resources for computation. In fact, it is often possible to trade between fanin and depth. For instance, the depth 2, fanin 2 circuit shown in Figure 4 emulates a single gate of fanin 4.

Part II
Complexity Theory

Complexity theory is, informally, the study of what computers can do *quickly*. This is largely done by studying sets of languages called *complexity classes* that share some property, such as being decidable in what is called *polynomial time*. Our goal shall be to introduce some of the most fundamental concepts of complexity theory by constructing some of the more important complexity classes, developing our conceptual understanding as we go. Thus, our approach will be to take a tour of the wide range of classes with which complexity theorists concern themselves.

3 The Polynomial Hierarchy

Our tour starts with the construction of a series of complexity classes based upon the time complexity of a problem. These classes deal with our notions of efficient algorithms, and will come to represent our ideas of which problems are tractable and which are intractable.

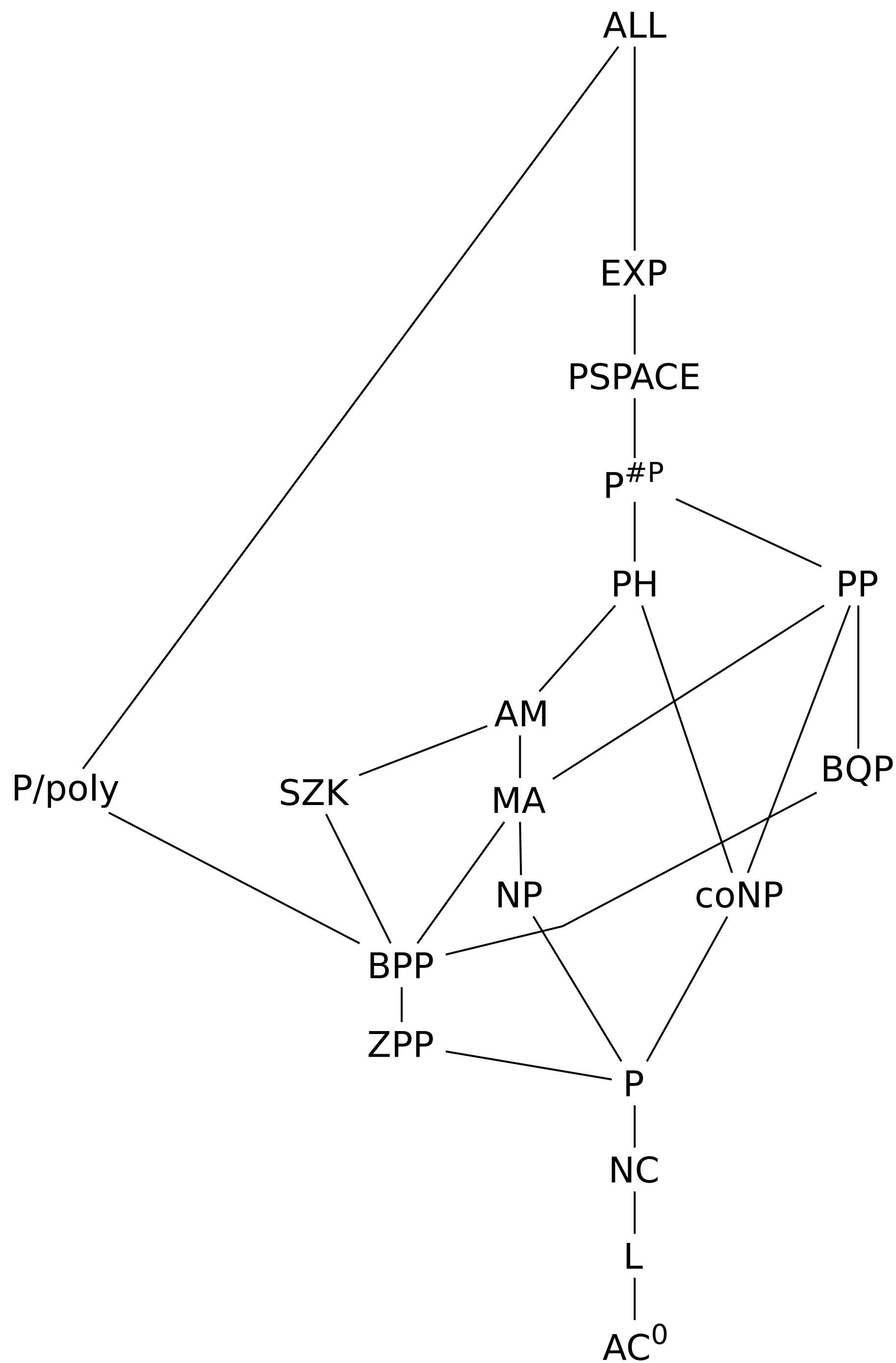

Figure 5: Known inclusions among complexity classes.

3.1 $\mathsf{DTIME}(f(n))$

Given a Turing machine $M = (Q, \Sigma, \Gamma, \delta)$, we think of each application of δ as representing a distinct time step. Intuitively, when we apply δ, the head "moves" to some new cell, a process that presumably takes some non-trivial amount of time. Hence, we are interested in how many time steps are required in order for M to halt given an input x, if at all. Formally, we say that a function $t : \mathbb{N} \to \mathbb{N}$ is a *time bound* for the Turing machine M if, for all inputs x having length n, M halts within $t(n)$ time steps. If for some language L there exists a Turing machine M deciding L and having a time bound $t(n) \in O(f(n))$ for some function $f : \mathbb{N} \to \mathbb{N}$, then we say that L is in the class $\mathsf{DTIME}(f(n))$ (deterministic time-bounded)[2].

When constructing classes, it is often convenient to think of them as containing not only languages, but also the machines meeting the restrictions imposed by the definitions of our classes. Thus, we would say that a machine M which has a time bound $t(n) \in O(f(n))$ for some function f is in the class $\mathsf{DTIME}(f(n))$, just as we would say that $L_M \in \mathsf{DTIME}(f(n))$. We shall ourselves make this switch to emphasize the relationship between the difficulty of a problem and the computational power whose existence is implied by the ability to solve a problem. Both views are perfectly valid, and are useful in different contexts.

3.2 P

With the definition of $\mathsf{DTIME}(f(n))$ in hand, we may now quickly define a new class:

$$\mathsf{P} = \bigcup_{k \in \mathbb{N}} \mathsf{DTIME}\left(n^k\right)$$

The class P is the set of all languages which admit solutions via Turing machines bounded in time by a function belonging to $O\left(n^k\right)$ for some k, and is often called the set of "polynomial-time" languages. Interestingly, complexity theory does not in general distinguish between languages for which the best-known Turing machine for some problem has a time-bound no better than $O\left(n^{100}\right)$ and those languages for which we can find time bounds in $O(n)$, despite that in designing algorithms for real-world computers, we generally consider anything worse than $O(n \ln n)$ to be too slow to be practical.

The most important reason for this choice is that the composition of two polynomials is always itself a polynomial. This allows us great power in proving things about P. For instance, we needn't concern ourselves with whether our Turing machines are multiple-tape machines or not, as simulating a multi-tape machine on a single-tape machine incurs at most a polynomial slowdown. Thus, any polynomial-time algorithm on a multi-tape machine remains a polynomial-time algorithm on a single-tape machine.

More generally, it is the composition property of polynomials which allows us to use a Turing machine to model so well in the first place. We can simu-

[2]See Appendix A for the definition of $O(\cdot)$.

late, for example, the execution of a random access machine [3] (RAM) with only a polynomial slowdown, and so polynomial-time RAM algorithms translate to polynomial-time Turing machine algorithms. This is why we can consider matrix multiplication to be polynomial-time on a Turing machine. Even the most naive but reasonable implementation of a matrix multiplication algorithm on a RAM will have a time-complexity bound no better than $O\left(n^{3/2}\right)$ for a square matrix having n^2 elements (algorithms such as Strassen's lower the exponent to about 1.4). Thus, even if we simulate a RAM algorithm for matrix multiplication without optimizing for execution on a Turing machine, we will have a polynomial-time Turing machine algorithm for matrix multiplication.

3.3 NP and coNP

One application of complexity theory is to answer questions of whether changes to our computational model allow us any additional efficiency. As such, we consider the impact on our computational power afforded by allowing our Turing machines to be non-deterministic, as described in Section 2.4. In particular, we shall define the class $\mathsf{NTIME}\left(f\left(n\right)\right)$ in analogy to $\mathsf{DTIME}\left(f\left(n\right)\right)$ to include those languages L for which there exists a non-deterministic Turing machine M solving L such that M has a time bound in $O\left(f\left(n\right)\right)$. We then define NP in analogy to P:

$$\mathsf{NP} = \bigcup_{k\in\mathbb{N}} \mathsf{NTIME}\left(n^k\right)$$

The problem with this definition, however, is that it betrays the subtlety of NP. Whereas P tells us about what problems can be solved efficiently, NP tells us about what problems can be *verified* efficiently. Put differently, consider a NTM $M = (Q, \Sigma, \Gamma, \delta_0, \delta_1)$ solving some language L. By assumption, for all input $x \in L$, at some point the set of states of M will include Q_{accept}. But then, Q_{accept} will be included by either $\delta_0\left(q_1\right)$ or $\delta_1\left(q_1\right)$ for some $q_1 \in Q$ (perhaps there are multiple– in that case, we only consider one such state arbitrarily). If $q_1 \neq q_{\text{start}}$, then it must also have been reached by either $\delta_0\left(q_2\right)$ or $\delta_1\left(q_2\right)$. In this way, we can construct a finite sequence of states $\{q_n, q_{n-1}, \ldots, q_1, q_{\text{accept}}\}$ starting with $q_n = q_{\text{start}}$ for some n and some string of bits $y = y_n y_{n-1} \cdots y_1$ such that $\delta_{y_i}\left(q_i\right) = q_{i+1}$. Given the string y, a deterministic verifier can check the computation by choosing which of the two transition functions to apply at each step. Thus, y acts to *prove* that $x \in L$. Of course, since M is a machine in NP, we must have that the computation proving $x \in L$ is not longer in time than a polynomial function of $|x|$. In particular, $|y| \in O\left(|x|^k\right)$ for some $k \in \mathbb{N}$.

Thus, we may think of NP as telling us what problems we can solve efficiently if we are given hints. Yet another interpretation tells us that NP is the class of problems which may be solved efficiently by a computer that can make guesses.

Perhaps surprisingly, despite its profound importance, proving things about NP turns out to be extraordinarily difficult. In fact, one of the greatest outstanding problems in mathematics is whether $\mathsf{P} = \mathsf{NP}$ or not. While it is strongly

[3]See Section C on page 39 for a brief treatment of these machines.

suspected that $\mathsf{P} \neq \mathsf{NP}$, proving that conjecture has confounded every proof technique currently known to complexity theory. We may even conclusively prove which proof techniques will be insufficient!

Note that we have not said anything about rejections by NTMs; this is a fundamental asymmetry in the definition of a NTM, and is reflected in the definitions of $\mathsf{NTIME}(f(n))$ and NP. In order to talk about the languages L for which the claim $x \notin L$ can be quickly verified, we define the *complement* of NP, coNP. We say that a language $L \in \mathsf{coNP}$ if the language $\bar{L} = \{ x \mid x \notin L \}$ is in NP.

3.4 NP-complete

Given the difficulty of proving the conjecture that $\mathsf{P} \neq \mathsf{NP}$, complexity theorists have imitated the proverbial drunk under the streetlight and have sought out to answer questions of a more limited scope that still provide us with information about the structure of NP. In this spirit, we are interested in determining which problems in NP are the "hardest," in the hopes that we can find some language $L \in \mathsf{NP} \backslash \mathsf{P}$. To do so, however, we must have some idea of how to identify those problems which best encapsulate what it means to be in NP.

The tool used to do so is the process of *reduction*. If being able to solve a problem L quickly allows us to solve some other problem M quickly, then we say that L must be at least as hard as M, since solving it somehow captures the difficulty of solving M. For example, we commonly want to know if a *Boolean formula* $f(x_1, x_2, \ldots, x_k)$ (that is, a function of the variables composed only of the binary operations $\vee$ and $\wedge$ together with the unary operation $\neg$) in some set of variables $\{x_i\}_{i=1}^{k}$ has a set of assignments such that f evaluates to 1. This problem, called SAT (for "satisfiability") can be shown to reduce to the special case where f is expressed in conjunctive normal form, wherein f is the AND of a set of *clauses* formed by the conjunction of one or more variables and negations of variables, and where all clauses of f have exactly 3 variables. We call solving this special case 3SAT, and we say that formulas in this special form are in 3-CNF.

More specifically, for every Boolean formula f, there exists a 3-CNF Boolean formula f' whose length is polynomial in that of f, and such that f has a satisfying assignment if and only if f' has a satisfying assignment. We can transform a Boolean formula to its 3-CNF form in polynomial-time using a Turing machine. Note that to carry out this transformation, we need not know what these satisfying assignments are, or even if they exist. Instead, this transformation allows us to reduce the problem of deciding the satisfiability of a SAT instance to the special case of deciding if a 3-CNF formula admits any satisfying assignments. Thus, if there exists an algorithm for solving the 3SAT decision problem in polynomial time, there exists an algorithm solving the SAT decision problem in polynomial time: reduce the general SAT formula to 3-CNF and apply our P algorithm for 3SAT.

The most famous reduction of all is that proved by the Cook-Levin Theorem, which translates the axioms of the Turing machine into Boolean formulas such

that any accepting NP computation corresponds to a satisfying assignment. Thus, the theorem states that any problem in NP reduces to 3SAT, and thus that 3SAT must be at least as hard as any other problem in NP. We would like to define reductions such that they are transitive, so that any problem to which 3SAT reduces shares this property. We then call NP-hard the class of decision problems[4] that are at least as hard as any problem in NP.

Obviously, SAT $\in$ NP-hard, since SAT reduces to 3SAT. By similar reasoning, we can show that NP-hard contains a very large number of interesting problems, such as graph coloring and the traveling salesman problem (TSP).

If a problem L is both in NP-hard and in NP, then we say that L is *complete* in NP:

$$\mathsf{NP\text{-}complete} = \mathsf{NP} \cap \mathsf{NP\text{-}hard}$$

This has the implication that if there exists any language L in NP-complete $\cap$ P, then we may conclude that P = NP. The failure to find any such language is part of the rationale for believing that P $\neq$ NP. This kind of condition is common enough in complexity theory that it gets a special name: we say that NP *collapses to* P if NP-complete and P are not disjoint.

3.5 PH

In computer science, it is often taken for granted that a program can execute a *subroutine*; that is, a program can consult some other, smaller program in order to arrive at some answer. In the complexity theory, however, one likes to take as little for granted as possible. Thus, we introduce *oracles* as a means of formalizing what is meant by a subroutine. Oracles must be defined in reference to a particular kind of machine, and so we shall proceed to discuss the impact of oracles upon variants of Turing machines. In particular, we shall be discussing oracles with respect to multi-tape Turing machines, the details of the construction of which is left to Appendix D.3.

Suppose that a 2-tape deterministic Turing machine $M = (Q, \Sigma, \Gamma, \delta)$ has three special states $q_{\text{oracle}}, q_{\text{yes}}, q_{\text{no}} \in Q$, and consider some language L. Then, the machine M^L, upon entering the state q_{oracle} instantaneously branches into either q_{yes} or q_{no}, depending on whether the contents of the second tape represents a string in L or not. We call M^L an *oracle machine* with access to L. This construction provides an abstract way to allow for a machine to "ask questions" about some other language.

As with any extension of our model of computation, we can use oracles to define new complexity classes which utilize them. For instance, the class P^{P} consists of those languages which may be solved in polynomial time by an oracle machine with access to some arbitrary language in P. Since the composition of two polynomials is itself a polynomial, however, we have that $\mathsf{P}^{\mathsf{P}} = \mathsf{P}$.

[4] Note that some authors define NP-hard to include not only decision problems, but more general *function problems* to which problems in NP reduce. Since we are largely restricting our focus to decision problems in this article, I shall presume that NP-hard only contains such problems.

More interesting results come about if we allow a P machine access to an oracle for an NP language. In fact, we can define an entire *hierarchy* of oracle classes. Let $\Delta_0\mathsf{P} = \Sigma_0\mathsf{P} = \Pi_0\mathsf{P} = \mathsf{P}$, and then define the following for all $i \in \mathbb{N}$:

$$\begin{aligned} \Delta_i\mathsf{P} &= \mathsf{P}^{\Sigma_{i-1}\mathsf{P}} \\ \Sigma_i\mathsf{P} &= \mathsf{NP}^{\Sigma_{i-1}\mathsf{P}} \\ \Pi_i\mathsf{P} &= \mathsf{coNP}^{\Sigma_{i-1}\mathsf{P}} \end{aligned}$$

Finally, we may define PH:

$$\mathsf{PH} = \bigcup_{i\in\mathbb{N}} (\Delta_i\mathsf{P} \cup \Sigma_i\mathsf{P} \cup \Pi_i\mathsf{P})$$

At this point, the reader may be well justified in asking why such a construction is considered at all interesting. Certainly, a model of computation that allows for a machine to instantly receive answers to hard computation problems seems fantastical at best. On the other hand, knowing that PH is big and contains machines that are unreasonably powerful gives us tools in analyzing other classes. If some model of computation leads to a class that contains PH , then we may take that as evidence that our new model of computation is probably unreasonable.

Another way that PH helps us is that we can use it to describe *collapses*. That is, if some hypothesis would imply that PH is equal to one of $\Delta_i\mathsf{P}$, $\Sigma_i\mathsf{P}$ or $\Pi_i\mathsf{P}$ for some finite i, then we generally consider that assumption to be less likely. For instance, one of the reasons that we conjecture that $\mathsf{P} \neq \mathsf{NP}$ is because if $\mathsf{P} = \mathsf{NP}$, then $\mathsf{NP}^{\mathsf{P}} = \mathsf{P}^{\mathsf{P}} = \mathsf{P}$, and so $\mathsf{PH} = \mathsf{P}$. Similarly, if we assume that $\mathsf{NP} = \mathsf{coNP}$ (that is, that the existence of witness strings for "yes" answers implies the existence of witness strings for "no" answers), then $\mathsf{PH} = \mathsf{NP}$.

One can also define the levels of PH in a way that doesn't rely upon this kind of a recursive definition by using *alternating qualifiers*. To do so, we borrow the terminology laid out by [3] and introduce the idea of a *balanced relation*. We say that R is a polynomially-balanced $(i+1)$-ary relation if there is some k such that for all $(x, y_1, y_2, \ldots, y_i) \in R$, each $|y_i|$ is bounded by $|x|^k$. Using this construction, we say that a language L is in $\Sigma_i\mathsf{P}$ if there exists a polynomially-balanced $(i+1)$-ary relation R such that:

$$L = \{\, x \mid (\forall y_1 \exists y_2 \cdots Q y_i)\ (x, y_1, y_2, \ldots, y_i) \in R \,\}$$

where Q is either $\forall$ or $\exists$, depending on whether i is odd or even, respectively. Then, we see that PH is the set of all languages defined by a finite sequence of alternating qualifiers. This construction applies very naturally to the analysis of two-player deterministic games such as chess[5], as we can define a language of board states which are a win for White in terms of alternating qualifiers: "a state is a win for White if, for all moves for Black, there exists a response for

[5]Technically, a slightly modified version of chess in which we cannot "loop" between some pair of board states. That is, we assume a rule like the *ko* rules of Go.

White such that for all of Black's possible moves, ... such that White wins." Here, it is understood that the ellipses cover enough qualifiers to describe the longest possible game of chess. Many other games admit similar descriptions, and so PH is a natural fit for problems relating to deciding such games.

3.6 Beyond PH: E, EXP, NEXP and EEXP

We can go still further than the polynomial hierarchy, however, and define *exponential-time* classes in analogy to the polynomial-time classes that we've already defined. In particular, we define EXP to be the union of $\mathsf{DTIME}\left(2^{p(n)}\right)$ for all polynomials $p(n)$ and NEXP to be the union of $\mathsf{NTIME}\left(2^{p(n)}\right)$. These are classes of problems that grow in complexity so fast that even under the most generous of assumptions, we have no real hope of being able to solve any but the smallest instances of these problems. Again, one may ask why such huge classes are interesting. These classes allow us to "scale up" and "scale down" results about other classes. For instance, if $\mathsf{P} = \mathsf{NP}$, then to deterministically decide languages in NEXP, we exponentially pad out instances to turn them into very large instances of NP problems. From there, we know that since (by assumption) $\mathsf{P} = \mathsf{NP}$, we can solve these padded instances in time polynomial in the padded length; that is, in exponential time. Thus, we have that $\mathsf{P} = \mathsf{NP} \Rightarrow \mathsf{EXP} = \mathsf{NEXP}$. By taking the contrapositive, if $\mathsf{EXP} \neq \mathsf{NEXP}$, we can use that to prove that $\mathsf{P} \neq \mathsf{NP}$ as a corollary.

This pattern of taking results in one strata and scaling them to another strata of classes is part of a general pattern in complexity theory. Equality typically propagates up, while separations (proofs demonstrating that two classes are not equal) propagate down. Thus, we often introduce classes of problems far too large and complex to ever reasonably solve directly in an attempt to answer questions about classes within our grasp. Such classes include not only exponential time as defined here, but exponential time with linear exponents (E) and doubly exponential time (EEXP), where we allow time complexities of the form $2^{2^{\mathrm{poly}(n)}}$.

4 Space-bounded Classes

4.1 PSPACE

In addition to discussing the time complexity of various computational problems, we can also examine what problems can be solved in a limited amount of *space*. To do so, we need to formalize what is meant by using space as a resource. Note that if a TM always halts, then, as a consequence, it can access only a finite number of cells on its tape before halting. Thus, we say that a problem has space complexity bounded by $f : \mathbb{N} \to \mathbb{N}$ if there exists a machine M solving the problem such that M modifies no more than $f(n)$ tape cells for all strings of length n. Using this definition, we can construct a space-complexity analogue to P, using $\mathsf{DSPACE}(f(n))$ in place of DTIME to indicate deterministic space

complexity. We say that a language L is in PSPACE if L has space complexity bounded by $f : \mathbb{N} \to \mathbb{N}$ for some $f \in O\left(n^k\right)$ and for some $k \in \mathbb{N}$.

Clearly, $\mathsf{P} \subseteq \mathsf{PSPACE}$, since we can access no more than one tape cell in a given timestep. On the other hand, we should not expect that $\mathsf{PSPACE} \subseteq \mathsf{P}$, since space winds up being a much more potent resource. Specifically, note that $\mathsf{NP} \subseteq \mathsf{PSPACE}$, since we can simply iterate through all possible witness strings for some input, and accept only if at least one of the witness strings verifies properly.

In fact, there is an important theorem that confirms our suspicions: the time and space hierarchy theorem, as stated by [3].

Theorem 1. *For all non-decreasing and constructable functions $f : \mathbb{N} \to \mathbb{N}$, the following hold*[6]*:*

- $\mathsf{DTIME}\left(f\left(n\right)\right) \subsetneq \mathsf{DTIME}\left(f\left(n\right)\log^2 f\left(n\right)\right)$
- $\mathsf{DSPACE}\left(f\left(n\right)\right) \subsetneq \mathsf{DSPACE}\left(f\left(n\right)\log f\left(n\right)\right)$

This theorem allows us to quickly state some very important results, such as $\mathsf{P} \subsetneq \mathsf{EXP}$. We shall state an analogous result for space complexity shortly.

4.2 L

We can make our space restrictions even tighter, and consider machines that are only allowed to modify the contents of $O\left(\log n\right)$ tape cells. If we call the class of languages solvable by such machines by L, then the space-hierarchy theorem guarantees that $\mathsf{L} \subsetneq \mathsf{PSPACE}$. But then, note that $\mathsf{L} \subseteq \mathsf{P}$, since an L-machine admits no more than a constant number of configurations for each possible setting of memory cells, thus meaning that in order for an L-machine to halt (as opposed to reentering an earlier configuration), it must have a time complexity bound in $O\left(2^{\log n}\right) = \bigcup_{k\in\mathbb{N}} O\left(n^k\right)$. Since $\mathsf{P} \subseteq \mathsf{PSPACE}$, we must then have by the hierarchy theorems that at least one of $\mathsf{L} \subsetneq \mathsf{P}$ or $\mathsf{P} \subsetneq \mathsf{PSPACE}$ is strict. This is a remarkable result, as it tells us that either additional time or additional space must buy us something, but without telling us how. This is a wonderful example of a non-constructive result; a common pattern in complexity theoretic proofs.

5 Circuit-Model Classes

Sometimes, it will be useful to discuss the complexity of computational problems by referencing the circuit model of computation, rather than Turing machines. Just as shifting between the integral and differential forms of Maxwell's equations let physicists solve problems more easily by adapting their approach to the problem at hand, complexity theorists shift between the different models of computation in order to analyze problems.

[6]Note that in the statement of the hierarchy theorems, and throughout the rest of the article, we use $\subsetneq$ to indicate a *strict* subset.

As an example of how to make this shift, it turns out that the class P corresponds exactly to those languages solved by uniform families of polynomial-size Boolean circuits[7]. Using this equivalence, we can examine some of the structure *inside* P, such as the degree of parallelizability admitted by certain polynomial-time problems are.

5.1 AC

We can think of each gate in a Boolean circuit as performing some useful computational work, rather than thinking of a whole circuit as an indivisible system. In this view, we imagine that each gate takes some finite amount of time to "update," and to produce its answer. Thus, if a computation can be performed using a circuit where the longest path is relatively short, then no part of the computation takes very long, and we consider the problem to be somewhat easier. In particular, even those problems that take polynomially many gates to solve may only require *constant-depth* circuits. Such problems would only take a short amount of time to solve, even if the circuits required are large, and so we gain computational efficiency by adding more gates in parallel.

The class AC is a particular way of formalizing this line of reasoning. We say that a language L is in AC^k if there exists an unbounded-fanin uniform family of circuits $\mathcal{C}$ solving L such that the size of each circuit of $\mathcal{C}$ is polynomial in the length of its input, and such that every circuit in $\mathcal{C}$ has depth bounded by a function in $O\left(\log^k n\right)$ for inputs of length n. In the same way that we constructed P as the union of $\mathsf{DTIME}\left(n^k\right)$ over all k, we can now define AC:

$$\mathsf{AC} = \bigcup_{k \in \mathbb{N}} \mathsf{AC}^k$$

Aside from the union AC, specific levels of the AC-hierarchy are interesting in their own right for what they tell us about the difficulty of seemingly simple problems. For instance, the problem MAJORITY discussed in Section 2.5 is not in AC^0; we *cannot* compute MAJORITY in constant time, no matter how many gates we throw at the problem. Similarly, we cannot answer whether the number of 1 bits in a string is odd or even (the PARITY problem) in AC^0.

5.2 NC

In our construction of AC, we allowed for unbounded fanin, and thus did not account for an important resource. It isn't clear that we get a reasonable view of parallel computation by ignoring fanin, and so we define a hierarchy of classes NC^k as before, but with the fanin of each circuit restricted to 2. (Note that any larger constant fanin can be emulated by a fanin 2 circuit by adding at most a constant factor to the depth, so the choice of 2 is largely for simplicity.) As

[7] A proof of this equivalence is given in [3], and is beyond the scope of this article.

before, we define NC by taking the union over all integer exponents:

$$\mathsf{NC} = \bigcup_{k \in \mathbb{N}} \mathsf{NC}^k$$

It is easy to see that for any $k \in \mathbb{N}$, $\mathsf{NC}^k \subseteq \mathsf{AC}^k$, since adding more fanin cannot possibly reduce the computational power of a circuit. What may be surprising, however, is that we can emulate unbounded fanin by adding to the depth at most a logarithmic factor. That is, for all k, $\mathsf{AC}^k \subseteq \mathsf{NC}^{k+1}$. Thus, we must have that $\mathsf{AC} = \mathsf{NC}$.

6 Probabilistic Classes

Having thus far discussed the impacts on computational power resulting from time and space bounds, from non-determinism, and from shifting to a different model of computation, we now have the tools needed to answer a very simple question: what can a computer do with the ability to flip a coin? So far, we have assumed that no matter how many times we let a machine tackle a particular input, the output will be the same. We have assumed that our computers are sufficiently well-behaved that we do not have to "double-check" their answers. As with any assumptions, however, it is natural to ask whether or not our assumptions actually help.

6.1 ZPP

The most simple way to allow a Turing machine to exploit randomness would be to specify two transition functions δ_0, δ_1, and to randomly choose which one to apply at each time step. This construction then lends itself to a very natural definition. We say that a language $L \in \mathsf{ZPTIME}\left(f\left(n\right)\right)$ if there exists a probabilistic Turing machine M such that M accepts a string x if and only if $x \in L$ and such that the expected time bound for M is within $O\left(f\left(n\right)\right)$. Then, we define the class $\mathsf{ZPP} = \bigcup_{k \in \mathbb{N}} \mathsf{ZPTIME}\left(n^k\right)$.

These classes represent what a probabilistic Turing machine can accomplish with zero probability of error (hence the ZP abbreviation). In practice not many problems admit random algorithms with zero probability of error that aren't also amenable to deterministic analysis. For instance, following the discovery of a ZPP algorithm for primality testing, the AKS algorithm was invented to test primality in P [9]. Thus, it is not presently clear if ZPP is at all different from P. On the other hand, it does seem clear that this naive approach to describing probabilistic Turing machines has problems. Most importantly, we have not at all relaxed our requirement that a machine always give the correct answer. Given that we are equipping our Turing machines with the ability to flip coins, we might expect that occasionally a machine get a wrong answer.

6.2 BPP

The most natural next step in understanding probabilistic Turing machines, then, is to allow our machines to occasionally arrive at the wrong conclusion. If we let our machines make too many errors, however, we might not wind up any better than if we had just flipped a coin ourselves. Thus, we define the class BPP to be those problems solvable in bounded-error polynomial time. More formally, if we let the probability that a machine M accepts a string x be denoted $p_M(x)$, then in order for M to solve a language L, we require that $p_M(x) \geq 2/3$ if and only if $x \in L$ and that $p_M(x) \leq 1/3$ otherwise.

The choices of $1/3$ and $2/3$ are actually arbitrary here; if you are only willing to accept an error probability of $1/9$, for instance, then you simply run a BPP machine multiple times to amplify the probability of a correct answer. Thus, another way of thinking about BPP is that we allow for the success probability to be amplified arbitrarily by bounding the probabilities to lie outside of a constant interval about $1/2$.

That said, there is still a third way to imagine BPP, and one that leads to a fairly subtle insight. Returning to our construction of non-deterministic Turing machines in Section 2.4, recall that each NTM M was specified by a pair of transition functions δ_0 and δ_1, just as we specified for ZPP in Section 6.1. Thus, we may think of our random choices as exploring the possible paths of an NTM. Taking this view, our requirement that $p_M(x) \geq {}^2\!/_3$ if $x \in L$ and $p_M(x) \leq {}^1\!/_3$ otherwise becomes a requirement on the number of paths of our BPP machine M terminating in Q_{accept}. Using this construction, we can bring our understanding of non-determinism to bear on problems of probability.

6.3 PP

Seeing that relaxing our constraints on error probabilities gives us so much more ability to describe computational processes involving randomness, we may be tempted to relax these constraints still further. Indeed, what if we only require that a probabilistic Turing machine give us some benefit over flipping a coin and calling that our answer? Define the class PP to be the set of languages L such that there exists a polynomial-time NTM M such that for any string $x \in L$ at least ${}^1\!/_2$ of the computation paths of $M(x)$ accept, while for any string $x \notin L$, strictly less than ${}^1\!/_2$ of the computation paths of $M(x)$ accept.

This class winds up being *much* harder to analyze, as we can no longer amplify the success probability; for all we know, the gap between the acceptance probabilities for $x \in L$ and $x \notin L$ could shrink with the length of x, requiring us to run the algorithm an increasing number of times to maintain some success probability. In fact, relaxing this requirement gives so much power that PP contains languages for each $k \in \mathbb{N}$ which cannot be solved by circuits of size $O\left(n^k\right)$ [11].

6.4 MA and AM

Finally, we want to develop probabilistic analogues to NP; that is, we want to find out what kinds of problems we can quickly verify using a probabilistic verifier. To do so, it is most convenient to discuss a *protocol*, rather than a specific machine. In particular, we will define two different protocols between Arthur, a BPP verifier, and Merlin, who finds himself in possession of unbounded computational resources.

In our first attempt at encapsulating what it means to quickly verify problems in a probabilistic fashion, we suppose that for some language L and input x, Merlin uses his unbounded resources to generate a proof string y having length in $O\left(|x|^k\right)$ for some $k \in \mathbb{N}$ such that Arthur will accept the pair (x, y) with probability at least $2/3$ if $x \in L$, and with probability at most $1/3$ otherwise. The class of problems solvable by this protocol, called the Merlin-Arthur protocol, is called MA. It is easy to see that MA is at least as big as NP, as the only change in moving to MA is the introduction of a probabilistic verifier.

Of course, now that we've framed the verification process in terms of two parties communicating via a protocol, we want to know if we can do more with this kind of a protocol. To answer this, we let Arthur have a first look at the input x and generate a *challenge* string based on the input. He sends Merlin the input, his challenge string and the random choices used in generating the challenge. Merlin then uses these strings to generate a response such that Arthur accepts the response with probability at least $2/3$ if $x \in L$ and at most $1/3$ otherwise. The class of problems solvable by this Arthur-Merlin protocol is called AM, following the naming scheme for MA.

As with so many questions in complexity theory, it is currently unknown whether allowing Arthur to generate a challenge affords us any more power. We can answer many seemingly unrelated questions in an attempt to build up a set of *lemmas* that we hope will eventually be used by a proof of the power of AM and MA.

7 Quantum Computation

Of course, no discussion of the importance of complexity theory could possibly be complete without quantum computation. In many ways, quantum computation is different enough that it is the second distinct model of computation, aside from the Turing machine (classical computation). As such, we would very much like to bring the tools of complexity to bear to figure out what this second model of computation *is*. Before we may do so, however, we must build up a little bit more terminology.

7.1 Reversible Classical Computation

When dealing with quantum bits, called *qubits*, it is very important to isolate them from their environs. When a qubit is measured, it decoheres like any other

quantum state. As such, when a qubit interacts with its environment, errors are introduced which harm the fidelity of the qubit.

This requirement has profound implications for how we design quantum computers. Thermodynamics tells us that in order to erase information, we must increase the entropy of a system, and so the only way to make a quantum computer is to never erase information during a computation. Put differently, quantum computations must be *reversible* in precisely the sense that the AND and OR operations discussed in Section 2.5 are not.

Returning to the classical picture, if you are given the value $f(x, y) = x \wedge y$ for some unknown bits x, y , it is in general impossible to recover x and y, and so energy must have been expended to erase the information originally contained in x and y. Thus, we would like to define a set of gates such that an entire computation may be reversed to recover the input, given some output. As it turns out, there is actually a single function $f : \{0,1\}^3 \to \{0,1\}^3$ which is both universal for classical computation and reversible, known as the *Toffoli gate*:

$$f(x, y, z) = (x, y, z \oplus xy)$$

To see that this single gate is universal, it suffices to show that we can emulate the NAND gate, which is universal (but not reversible) for classical computation. Of course, since NAND is a 2-bit to 1-bit gate and the Toffoli gate both accepts and produces three bits, we must introduce an auxiliary bit (known as an *ancilla bit*) to actually emulate NAND. That is, we would like to produce $f(x, y, z) = (x, y, x \text{ NAND } y) = (x, y, \neg x \vee \neg y)$. But then, since $1 \oplus xy = \neg(xy)$, we're done if we set our ancilla bit to 1.

This in turn shows that we can build a reversible version of *any* Boolean circuit by adding enough ancilla bits to our computation. Moreover, we need not concern ourselves with the requirement that our ancilla bits be set to 1, since we can *uncompute* any changes to the ancilla bits by reversing our computation. Finally, note that our simulation of Boolean circuits using reversible computation did not add more than a constant factor of additional gates, and so we lose nothing in the asymptotic sense.

7.1.1 A Strange Realization

In their paper introducing the idea of reversible computation as a model of computation, Fredkin and Toffoli also proposed a physical example of reversible computation, forging a link between physical theories and computational processes [6]. By using Newton's laws of motion and assuming perfectly elastic billiards balls, we can think of a wire as being some path along a table upon which there may either be a ball or not. Our logical elements then become interactions between two balls, guided by fixed obstacles called *mirrors*.

Fredkin and Toffoli applied this model to implement the *Fredkin gate*, shown in Figure 6 on the next page. The Fredkin gate is also known as the controlled-swap gate, as the operation of the gate is to swap two of its inputs if the *control input* is 1, and to act as the identity gate if the control is 0. One particular billiard-ball implementation of this gate is built up using *switch gates*, where

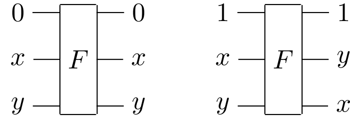

Figure 6: Operation of the Fredkin gate.

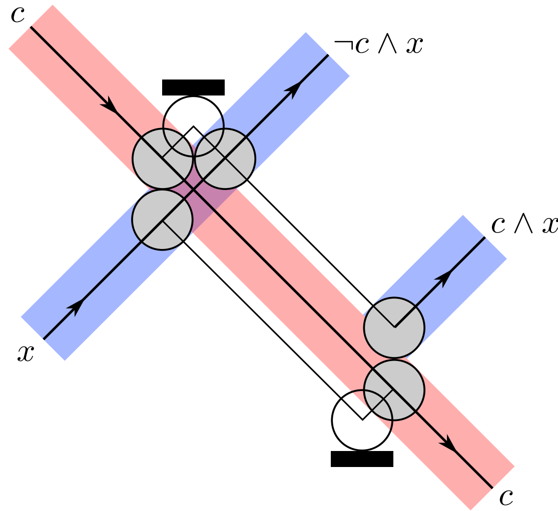

Figure 7: Billiards-ball based switch gate.

a ball is redirected onto one of two different paths based on whether or not a ball is present along a control path. An example of a billiard-ball switch gate is shown in Figure 7, where the path of a ball along path x is changed based on a whether there is a ball along the control path c. Note that if c and x are both set (have balls along the paths), then the balls that eventually winds up on paths c and x swap with each other. Since we consider individual balls to be indistinguishable, this is fine, and greatly simplifies the construction of the gate.

The Fredkin gate is also universal for computation, and so the billiards-ball computer is demonstrably equivalent to the Turing machine in power. Since we only assumed Newton's laws in constructing the billiards-ball computer, this tells us that we cannot in general analytically solve said laws of motion. If we could, then we could also solve the HALTING problem, as each instance of the problem would be a specific arrangement of mirrors and balls, and would reduce to deciding if a ball ever exits some region of our table. Thus, we have already shown something very profound: physical theories carry with them some inherent computation difficulty. This realization will be revisited in Section 10.

7.2 Modeling Quantum Computation

7.2.1 The Quantum Circuit Model

It turns out that, in practice, while the Turing machine is well-suited to describe classical computation, extending it to the quantum case is messy at best. While we may meaningfully discuss quantum Turing machines (QTMs), our attention will be better spent by (at least temporarily) focusing on an analogue of the classical acyclic Boolean circuit model described in Section 2.5 called, unsurprisingly, the quantum circuit model (QCM) [2]. Just as we exploited equivalences between classical Boolean circuits and Turing machines, we shall use the QCM to model the same processes for which we might have instead invoked a QTM.

In the QCM, we model the time evolution of a quantum system admitting only discrete energy states by the application of a sequence of matrices, called *gates*. To illustrate the principle, we can formulate a similar construction for classical gates, such as representing the NOT gate as $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. In order for the reversibility constraint to be upheld, we require that any quantum gate must be *unitary*; that is, a gate U must satisfy $I = UU^\dagger$ where I is the identity, and $U^\dagger$ is the Hermitian of U.

If we represent a qubit $|\psi\rangle = a|0\rangle + b|1\rangle$ by $|\psi\rangle = \begin{bmatrix} a \\ b \end{bmatrix}$, then all of our classical gates immediately have quantum circuit realizations. Moreover, we can construct gates with no classical equivalent, such as the Hadamard gate H:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The effect of this gate is to shift the basis of ψ from $\{|0\rangle, |1\rangle\}$ to $\{|+\rangle, |-\rangle\}$, making the Hadamard gate very useful in exploiting the linearity of quantum states. Another interesting set of single-qubit gates are those *phase rotation gates* generated by the function $R(\theta)$:

$$R(\theta) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\theta} \end{bmatrix}$$

Some quantum gates acting on multiple qubits can be built up by using the *tensor product* of simpler gates. For instance, given a single-qubit gate U, we can apply U to only the second of two qubits by applying the tensor product to the identity gate I and to U: $I \otimes U$. In this construction, we represent a two-qubit product state $|\psi\rangle|\phi\rangle$ as the tensor product of the two single-qubit

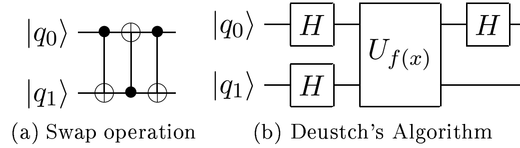

Figure 8: Sample quantum circuits.

state vectors:

$$\begin{aligned}
|\psi\rangle &= \begin{bmatrix} a \\ b \end{bmatrix} \\
|\phi\rangle &= \begin{bmatrix} c \\ d \end{bmatrix} \\
|\psi\rangle|\phi\rangle &= \begin{bmatrix} a \\ b \end{bmatrix} \otimes \begin{bmatrix} c \\ d \end{bmatrix} = \begin{bmatrix} ac \\ ad \\ bc \\ bd \end{bmatrix}
\end{aligned}$$

Other gates include the controlled-NOT gate (CNOT), represented by an $\oplus$ symbol in quantum circuit diagrams such as the example circuit in Figure 8a demonstrating how to swap two qubits. The matrix representation of C cannot be written as the tensor product of any two smaller matrices, showing that we cannot factor CNOT gates into any simpler gates:

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Put differently, $C|x, y\rangle = |x, x \oplus y\rangle$. Of course, we can make any unitary 2×2 matrix U controlled in this manner:

$$C(U) = \begin{bmatrix} I & 0 \\ 0 & U \end{bmatrix}$$

Since quantum circuits are quite different from Boolean circuits, it is helpful to have a way of representing them graphically. We draw gates as squares, with a mathematical symbol representing what the gate is. Quantum wires are drawn as single lines, while classical wires are drawn as double lines. For instance, the circuit shown in Figure 8b implements the famed algorithm of Deustch.

7.2.2 Universal Gates for QCM

Just as with classical computation, we are very interested in a set of quantum gates such that any other quantum gate can be constructed by repeated applications of our universal set of gates. It turns out that the Hadamard and

$R\left(\pi/4\right)$ single qubit gates together with the phase rotation and controlled-NOT two qubit gates form a universal set, in the sense that any other gate can be approximated with arbitrarily small error by compositions of these gates [10].

This is quite advantageous, as it shows that many-qubit gates need not be directly constructed, but rather, we can focus on constructing a set of small and local gates. Similarly, the existence of a small set of universal gates makes proving many things about quantum computation easier, in the same way that knowing that NAND was universal for classical computation allowed us to quickly conclude in Section 7.1 that the Toffoli gate was as well.

7.3 BQP

As we discussed in Section 5 on page 18, P can be formulated as the set of languages solvable by uniform families of polynomial-size circuits. Thus, in analogy to that definition, we would like to define a polynomial-time class for quantum computation by using the QCM model developed above. Unfortunately, we run into a problem: quantum measurement is inherently probabilistic, and so we must account for the possibility that our quantum algorithms generate wrong answers. To do so, we rely upon the discussion of BPP in Section 6.2 on page 21 and require that our quantum algorithms succeed at least $2/3$ of the time.

Putting it together, a language L is in BQP if there exists a uniform family of quantum circuits $\mathcal{C}$ such that:

$$\Pr\left[\mathcal{C}\left(x\right) \text{ accepts}\right] : \begin{cases} \geq 2/3 & x \in L \\ \leq 1/3 & x \notin L \end{cases}$$

where $\mathcal{C}\left(x\right)$ indicates the application of x as input to the circuit in $\mathcal{C}$ for inputs of length $|x|$.

With BQP, complexity theory has given us the language that we need to answer a very important question: are quantum computers more powerful than classical computers? We will discuss this question in more detail in Sections 9.1 through 9.3, but for now, suffice to say that we have already discovered BQP algorithms for important problems that are not currently believed to be tractable by classical computers. For instance, the famous Shor's Algorithm shows that factoring integers and finding discrete logarithms are both problems that are tractable by quantum computers [12].

8 Other Classes

We have barely scratched the surface of what questions complexity theory is poised to answer. There are many more classes of problems which are open to study, and we could not possibly do justice to all of them here. Nonetheless, before leaving our tour of complexity theory, we shall explore a few of the more exotic classes.

8.1 P/poly

Amongst the strange powers which we can bestow upon our Turing machines is that of *advice*. Loosely, P/poly is the class of problems solvable in polynomial-time by a Turing machine M which takes as input a string x and an advice string y which depends only on the length of x. Elaborating a bit, a P/poly machine is one which is given some arbitrarily difficult to compute string y along with each input, subject to the constraints that y cannot depend on the input, but only on its length, and that y has a length bounded by a polynomial in the length of the input. Advice represents a relaxation of the uniformity constraint on families of Boolean circuits discussed in Section 2.5, since an advice string is essentially a way of representing that different sizes of Boolean circuits may be generated by different algorithms.

To see the incredible power that advice allows for, note that P/poly includes at least one uncomputable language. In particular, suppose that L is an uncomputable language (this is a safe assumption, as we have demonstrated the existence of at least one uncomputable language). Then, consider the language $L' = \{ 1^x \mid x \in L \}$, where 1^x means a sequence of ones whose length is the number described by the binary string x. We may then construct a machine which solves L' by checking that a string contains only ones, and by giving the machine one bit of advice for input x':

$$y = \begin{cases} 0 & |x'| \notin L \\ 1 & |x'| \in L \end{cases}$$

Of course, P/poly cannot compute arbitrary uncomputable languages, as the construction of L' above depended on an exponential enlarging of the original uncomputable language L. Nonetheless, we see that adding non-uniformity (advice) to a Turing machine can afford large amounts of computational power. This insight goes a long way towards explaining why we require uniformity in the construction of Boolean circuits.

8.2 SZK

In all of our discussions of verifications and witness strings as employed by classes like NP (Section 3.3) and MA (6.4), the verifier could turn around and send whatever witness string he was given to a second verifier, and thus convince them of the veracity of some statement. In some cases, however, this is an undesirable property, as we would like to convince someone of the veracity of some claim without the verifier learning anything about the problem. That is, we would like a verifier to have *statistical zero knowledge* after the verification protocol has completed. The technical details of how to state this requirement are beyond the scope of this article, but we can give an example that communicates the spirit of this idea.

Consider two graphs, G and H, and suppose that Arthur wants to know if G and H are isomorphic to each other, and that he has access to BPP machines.

Then, if Merlin wants to prove to Arthur that G and H are non-isomorphic, using his vast computational power, but such that Arthur cannot then convince anyone else that $G \ncong H$, he can offer Arthur the following protocol. First, Arthur randomly permutes the vertices of both G and H, then offers both to Merlin. Merlin then tells Arthur which he believes to be the graph permuted from G. If $G \cong H$, Merlin can do only this with probability $1/2$, while if $G \ncong H$, and if Merlin can solve graph isomorphism, then he can identify G with probability 1. Thus, with each round of this protocol, Arthur can halve the probability with which a cheater would fool him. On the other hand, replaying this protocol to some other agent would not convince them that $G \ncong H$, as Arthur presumably knew which graphs he permuted into which. Thus, while Arthur has indeed learned that $G \ncong H$, he has not learned anything that would allow him to reproduce this proof.

We call the class of problems admitting such a protocol by SZK.

8.3 #P

Thus far, we have restricted our view of complexity to the study of Boolean functions. While it can be seen that this view affords a lot of explanatory power, we need not exclude other conceptions of what a problem is. We can, for example, consider *function problems* of the form $f : \{0,1\}^* \to \{0,1\}^*$. The most basic class of such function problems is #P, the set of function problems solvable by counting the number of accepting paths of an NP machine. That is, a problem $f(x)$ is in #P if there exists a machine $M \in \mathsf{NP}$ such that $M(x)$ has exactly $f(x)$ paths that end in an accepting state. Clearly, such a construction generalizes NP to the world of function problems, an in some sense implies a model of computation strictly more powerful than that implied by NP.

The most direct application of #P is as an oracle to other classes. In particular, the class $\mathsf{P}^{\#\mathsf{P}}$ is powerful enough to contain all of PH [14].

Part III
Applications and Implications

9 Why We Care About Quantum Computers

In science, nothing is ever quite as exciting as if we are wrong. Being wrong allows us to explore all sorts of new possibilities, new theories and new applications. Being wrong demonstrates that science, at some basic level, is successfully self-correcting. Most importantly, though, when we find out that we are wrong, we may set about becoming *right*.

Quantum computers offer us an unprecedented opportunity to be wrong. If feasible, they threaten an assertion known as the Church-Turing Thesis. If they aren't feasible, then presumably, they must fail because we were wrong at

some fundamental level, and we can then go about developing new fundamental physics. Either way, we learn something of profound importance about our world.

9.1 The Church-Turing Thesis

For nearly one hundred years, we have considered the Church-Turing Thesis as stating the edge of computational power. Informally, the Church-Turing Thesis states that no physically-realizable computer is more powerful than a Turing machine. Alternately, the Thesis asserts that *any* computational process can be simulated by a Turing machine. In particular, the Church-Turing Thesis seems to claim that quantum computers are no better than classical computers, as we can run simulations of quantum computers using classical computational processes. Of course, to evaluate this claim, we must make formal what we mean by "more powerful," and we shall find that the language of complexity theory is well-suited for the task.

The Church-Turing Thesis actually can be interpreted in two different reasonable ways, which have become known as the weak and strong forms of the thesis. The weak form claims that there does not exist a language which can be solved by a physically-realizable computer but which is unsolvable for a Turing machine. The strong form claims that no language can be solved faster on a physically-realizable computer (in the asymptotic sense) than on a Turing machine. Since we already know that classical computers such as Turing machines can simulate quantum computers, if only very slowly, it is specifically the strong form of the Church-Turing Thesis with which we are concerned.

One specific consequence of the Strong Church-Turing Thesis that is more amenable to analysis than the general claim is the claim that there does not exist any language solvable in polynomial time by a quantum computer, but which requires strictly more than polynomial time to solve on a classical computer. The existence of such a language would give us a concrete proof that quantum computational processes can not be *efficiently* simulated by classical computers. Thus, if we discover that $\mathsf{P} \subsetneq \mathsf{BQP}$, then we will have disproved the Strong Church-Turing Thesis, and in doing so, will have blown open the frontiers of computing.

9.2 $\mathsf{P} = \mathsf{BQP}$?

The question then becomes, is P strictly smaller than BQP or not? At the present, this is an open question in complexity theory, but as always, we still seek evidence towards one conclusion or the other. What might such evidence look like? If $\mathsf{P} = \mathsf{BQP}$, then we should expect to find some algorithm that can quickly simulate the execution of a quantum computer. Such an algorithm would have to provide a polynomial-time simulation, and must *derandomize* quantum algorithms; that is, the algorithm would have to simulate the quantum computer without access to randomness, and without any probability of error. Thus, we would also expect to find an algorithm by which we could derandomize classical

probabilistic machines, instructing us that if we wish to demonstrate $\mathsf{P} = \mathsf{BQP}$, it would be helpful to first demonstrate that $\mathsf{P} = \mathsf{BPP}$.

To date, we have not found any algorithm which derandomizes arbitrary machines in BPP, but we have been able to derandomize many specific algorithms. For instance, the problem of testing an integer for primality admits a fairly simple (for number theory, anyway) BPP algorithm. It was long thought that the problem of primality testing was somehow intrinsically probabilistic, and thus that we should not expect to see a deterministic algorithm. Recently, however, a deterministic polynomial-time algorithm was invented for primality testing, lending credence to the idea that derandomization may be more generally possible [9].

9.3 $\mathsf{P} \subsetneq \mathsf{BQP}$?

By contrast, evidence that $\mathsf{P} \subsetneq \mathsf{BQP}$ may take the form of demonstrating a language $L \in \mathsf{BQP} \backslash \mathsf{P}$. While this may sound straightforward, it is often very difficult to prove that a language is *not* in some specific class, because you must make a statement for every machine in that class. Specifically, proving $\mathsf{P} \subsetneq \mathsf{BQP}$ would follow directly from proving that:

$$(\exists M \in \mathsf{BQP})\,(\forall N \in \mathsf{P})\,(\exists x \in \Sigma^*) : M(x) \neq N(x)$$

This kind of an alternating-qualifier statement packs a very large amount of content into very terse mathematical language, and hides the difficulty of formulating a proof. As such, current efforts have largely focused on proving smaller claims about specific languages in the hopes of developing a framework by which one may show $\mathsf{P} \subsetneq \mathsf{BQP}$ if indeed it is the case.

9.4 How Quantum Computers Might Fail

On the other hand, what if quantum computation is found to be infeasible? Then, we must ask *how* quantum computers fail. For instance, it could be that there is something at fault with quantum theory itself.

In that case, quantum computers will have very successfully given us profound hints into the workings of the universe. Though this may sound like an exaggeration, it is not. Something we can learn from the quantum circuit model is that every quantum computational process can be described precisely using the language of quantum mechanics. Of course, it is always possible to describe non-existent phenomenon using any valid theory, such as writing down a force that could not possibly correspond to any physically realizable process. This is why it is advantageous to speak of quantum circuits in terms of a small set of elementary gates: in order for the whole circuit to correspond to some physical process, we only need that every gate within the circuit is realizable. Therefore, in a very concrete sense, the validity of quantum mechanics and of quantum computation are quite closely married indeed. The strength of this coupling have even led to a system of thought where quantum mechanics is the

"operating system" on which any process may run, and where the "real world" corresponds to a specific choice of a Hamiltonian operator.

It may also be that decoherence turns out to be insurmountable, and that our qubits will never be able to last throughout a whole computation. This mode of failure, however, is not well supported by experiment. From our present experience, we suspect that keeping decoherence down to a tolerable level is only exceedingly difficult, not impossible. We have discovered no fundamental reason why the problem of decoherence should be paramount, and indeed, we have made great strides in reducing the problem over the past 20 years. For this trend to halt, we must discover some new frustration that prevents us from properly isolating a quantum computational process from the outside world. Such a new frustration would again give us great insight, and could even possibly be turned around to provide some immediate concrete benefit.

Another possibility often posed by critics is that even under the best of circumstances, the error rates in preparing and communicating quantum states and in applying gates would be too high to overcome. This scenario, however, stands in direct contradiction with the findings from a rich field of study in quantum error correction. Just as it is possible to transmit classical data with arbitrarily high fidelity across a noisy channel by the use of error-correcting codes so long as the error rate of the noisy channel is below some reasonable threshold, it is likewise possible to achieve arbitrarily high fidelity in a noisy quantum channel. Just as we amplified the success rate of machines in BPP and BQP (see Sections 6.2 and 7.3, respectively), we may iteratively repeat an error-correcting algorithm to reduce the error probability associated with a quantum channel or gate. Thus, in order for error-correction to fail in such a way as to render quantum computation infeasible, the assumptions underlying error-correction algorithms must fail. Specifically, that would imply that errors are not randomly distributed, but are adversarily chosen to frustrate error correction. Such an adversarial model of error would require quite a lot of computational power, indeed; no less computational power than the machine it is trying to defeat, certainly. This is an important point, and reflects a general process in thinking about quantum computers: in order to be infeasible, there must be some other process at work of even greater computational power. Thus, if we are wrong and quantum computers are infeasible, we should expect *more* computational power may be exploited, and not less.

10 Complexity of Evaluating Physical Theories

With the increasing use of computational simulation in theoretical physics over the past several decades, physicists have suddenly found themselves called upon to understand how to *efficiently* program their simulations. In rising to the call, many physicists have learned how to program in a variety of different languages and environments. Debates rage endlessly about which language is most efficient for carrying out large linear algebra calculations and how to properly vectorize arithmetic to exploit modern processors. Moreover, physics students are taught

what toolkits, libraries and techniques to use in order to effectively and efficiently perform simulations.

As we have seen, though, there are much larger concerns. All of these concerns affect only the multiplicative constants in our time complexities– the very constants that complexity theorists deem too insignificant to consider! As Donald Knuth famously opined, "we should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. [5]" All the vectorization in the world won't help a program that uses bubble-sort, an $O\left(n^2\right)$ algorithm, as opposed to quicksort, which has a time complexity in $O\left(n \log n\right)$. Even worse are those simulations which demand the solving of problems that aren't even in P.

In response to the difficulties of choosing appropriate algorithms, one may well ask for reasonable lower bounds on the complexity of evaluating some physical theory. A particularly poignant example, and one which we visited in detail in Section 9, is what difficulty is *inherent* to simulating quantum mechanics. If any simulation of quantum mechanics could be completed using only polynomial-time algorithms, then we could simply simulate the perfect operation of a quantum computer and thus show that $\mathsf{P} = \mathsf{BQP}$. Thus, finding a lower bound to the complexity associated with quantum mechanics itself would be instrumental in separating P from BQP. This is an important point, and one that bears repeating: if is true that $\mathsf{P} \subsetneq \mathsf{BQP}$, then that is because quantum mechanics as a theory requires super-polynomial amounts of computing power to simulate on a Turing machine. Thus, in a very real sense, $\mathsf{P} \subsetneq \mathsf{BQP}$ implies that quantum mechanics is a complex theory.

Even in the classical case, we can make nontrivial conclusions about the complexity inherent to physical theories. Ben-Hur and Siegelmann, for instance, defined a model of computation based on some initial real-valued function *relaxing* to a final result via a differential equation [1]. By restating the maximum flow problem to exploit such a continuous model of computation, Ben-Hur and Siegelmann argued that solving a single ordinary differential equation having polynomial-time convergence is inherently as hard as simulating polynomial-time computation. Thus, for those physical theories based upon ODEs, it can be argued that these theories are at least as complicated as any problem in P.

Finding less obvious examples, however, is a demanding project. With that in mind, I would ask the reader to indulge me the leisure of speculating about future research. As we shall see in Section 11, we suspect that universe simply does not allow for extremely powerful computational models to operate. Thus, we should not expect any physical theory to require similarly extreme resources to evaluate, or else we would have that the universe is performing computation in a manner inaccessible to us. In spite of these difficulties, however, it is my sincere belief that this line of reasoning provides much in the way of hidden promise. As our understanding of the interconnections between computer science and physics grows and matures, we should expect to see such interdisciplinary questions as these to be explored more and more thoroughly. The relatively nascent field of complexity theory is rapidly developing the rich toolkit needed to understand questions of profound and direct importance to the practice of physics.

11 Physical Theories and Computational Power

For much of the history of physics, we have held that the law of conservation of energy is correct, and that it can be used to evaluate potential physical theories for correctness. We reject nearly out of hand any physical theory that fails to properly conserve energy, for we have had centuries of experience telling us that such theories are always incorrect. Even though it was not until Noether's famed theorem of 1918 that this law was escalated to a theorem, we have successfully derived much value from the *empirical* statement that correct physical theories respect the conservation of energy.

Of course, this guideline has not survived intact through the ages, and has been revised as we come to understand the laws of physics better. For instance, we have discovered, thanks to Einstein, that conservation of mechanical energy as formulated in the vocabulary of Newtonian physics cannot hold, but rather we must change our understanding of energy to incorporate the energy stored as rest mass. Similarly, with quantum mechanics, we have learned that conservation of energy must be a statistical law, or else we would contradict Heisenberg's Uncertainty Principle. What has remained, however, is the basic principle that energy is neither created nor destroyed. This principle has given us a great tool with which we may evaluate potential theories.

Other similar guidelines have been used throughout the history of physics to guide our intuitions when the plausible theoryspace is too large to fully evaluate. For example, we have tended to reject theories which fail to respect causality and locality, though again, these principles required some refining in light of quantum theory. We have built up a venerable library of things which we do not expect correct physical theories to allow, including events occurring before their causes, transmitting information faster than the speed of light, violation of unitarity, globally decreasing entropy and absolute zero temperatures.

Some complexity theorists, most notably Aaronson, argue that we should likewise not expect to *ever* be able to solve NP-complete problems efficiently [15]. Taking this view, we presently live in a time much like just before Noether's Theorem was proved, in that we believe that we will one day be able to prove conclusively that NP-complete problems are inherently intractable, rather than merely awaiting some new innovation. Thus, even in the absence of such a proof, we may start to act on the *assumption* that $\mathsf{P} \neq \mathsf{NP}$, and in fact, that classes including all of NP are physically unrealizable. This kind of a principle can be arrived at by studying some hypothetical computational models, called *hypercomputers*, that would allow the efficient solving of intractable problems.

11.1 Hypercomputers

Essentially, we can construct a model of hypercomputer by giving a Turing machine access to some kind of resource that we generally consider to be physically unreasonable. Of these resources, perhaps the most obvious extension would be to consider a Turing machine that can compute an infinite number of transitions in a finite time interval, essentially revoking the time-complexity bounds

of our more physical classes. There is even an obvious story we can tell that may explain how such a machine works: we take advantage of time dilation by moving our Turing machine at ever-faster velocities, such that we observe the first transition in one second, the second transition in $1/2$ s, the third in $1/4$ s, and so on. Then, an infinite number of steps could be completed within a finite interval, allowing for the expenditure of an infinite amount of energy. It should be clear that an infinite-time hypercomputer could, for instance, be able to solve all of NP by simply iterating through all possible witness strings and checking them.

We would like to find a way of affording additional computational power without infinite time, however, to further suggest that super-Turing power may in fact be physically unrealizable. Thus, we extend the time resource of a Turing machine not by allowing an infinite amount of time, but by allowing a Turing machine to exploit bits along *closed timelike curves* (CTCs). Put differently, we can let a Turing machine send bits back in time to itself. Of course, we would have to restrict the machine to be causally consistent, and so we would essentially be letting Nature set the values of the CTC bits to a fixed point for the computation being carried out by the TM. That is, we would presume that the universe will choose values for our CTC bits such that the our computation does not change them, thus resolving all causality paradoxes. As it turns out, such a Turing machine would be able to solve all of PSPACE in polynomial time, as one might expect: in the context of classical computation, time travel essentially allows us to treat time as a space resource, overwriting our CTC bits until we arrive at a stable answer.

Of course, there are much less ridiculous examples; indeed, if the universe allows for analog computers (also known as *real computers* in that they work over the real numbers), we may well be able to exploit them to perform efficient computation. While the specific model of analog computer has a great amount of bearing on what we can accomplish with one, we can rather informally construct a model that gives rise to immense amounts of computational power.

We should not expect, however, that analog computers of this kind are any more reasonable than any other of our hypercomputers. Presumably, the universe must store our real-valued registers *somewhere*. Either we must assume that the storage of our registers takes up an infinite amount of space in order to represent the full precision, or we must be able to store arbitrary amounts of information within a finite volume. In the first case, we shall argue in Section 11.2 that we cannot possibly interact with all of the bits of our infinite-volume analog register, and so we must presume that analog registers take a finite volume. This assumption, however, is fraught with its own problems. Since our model of analog computation allows us to extract bits of information from analog registers, we duplicate all the problems attending to storing an infinite number of bits in a finite volume. Again, we shall argue in Section 11.2 why this is unreasonable, but for now suffice to say that analog computation does not save us from the limits imposed on computation by the universe, but rather shows us how fundamental these limits truly are.

11.2 Fundamental Limits

Of course, we can also turn the question around, and can construct more limited models of computation that try to take into account fundamental limitations imposed by the laws of physics. As an informal example, note that since the universe is expanding, the bits involved in a computation may recede outside of a computational device's lightcone, resulting in some of the bits involved in a computation becoming causally separated from the rest. Of course, one could not simply pack the bits into an arbitrarily tight space, since the holographic principle places a strict bound on the entropy that may ever be contained within a region of space. Thus, the expansion of the universe together with the existence of black holes places a severe restriction upon what any computational process may achieve.

We should, however, expect this to be the case: if computation is inherently physical, we should expect that some computational processes are unphysical in exactly the same sense that some mechanical processes are unphysical. For instance, the holographic bound developed by Susskind shows that computation is constrained in the amount of entropy that can be involved [8]. In the same vein, Bousso finds an upper bound of $N = 3\pi/\Lambda$ on the number of bits of information which ever may be causally related in a de Sitter space, where Λ is the cosmological constant [13].

These kinds of results show us specific examples of principles that are already well-understood; namely, they show us that the universe imposes limits on the scope of computation. It is not enough to merely understand that these limits exist, though. We must understand these limits, using the language that complexity theory gives us.

11.3 An Extended Church-Turing Thesis

We have seen that the laws of our universe can be exploited to perform computation, but that the universe in turn limits what computations we can perform. That is, computational processes are physical processes. For example, the billiards-ball computer described in Section 7.1.1 depends crucially on Newtonian mechanics, and is implied by the existence of a world describable by Newtonian mechanics. Similarly, quantum computers depend on quantum mechanics, and are inherently a part of any world in which quantum states evolve in a unitary fashion.

On the other hand, we can also simulate physical systems using computational processes. As a specific example, we can simulate the dynamics of quantum systems using Turing machines, even if we cannot do so quickly. If we believe quantum mechanics to be a good description of the universe, then that implies that we can simulate arbitrary physical systems. Note, however, that we cannot necessarily *evaluate* these systems, as we discovered with the billiards-ball computer. Rather, we can simulate each specific timestep of a quantum system, even if doing so may lead us into an infinite loop. That ability does not imply that our system will ever reach an "answer state."

Of course, we don't believe quantum mechanics to be *complete*, but in the same sense that general relativity and quantum mechanics both approximate to Newton's laws of motion in the right regimes, we should expect that a complete description of the universe will not invalidate simulations of quantum systems. Rather, we should expect that a complete description will allow us to simulate systems currently outside of the regime in which we find that quantum mechanics is valid, such as inside black holes. Indeed, neither GRT nor quantum mechanics in any way prevent us from making meaningful simulations of what we now call classical physics, but rather each extends our reach to allow for *more* simulation power.

These arguments, taken together, suggest something very profound: that any physical process can be simulated by a universal computational device. This assertion is known as the Church-Turing-Deutsch (CTD) Thesis after its original formulator, David Deutsch [4]. Note that we can quickly recover our familiar Church-Turing Thesis by considering the simulation of a physical process which performs some useful computation. Thus, the CTD Thesis is truly a generalization of the the Church-Turing Thesis that completes the connection of computation back to the physical world.

12 Concluding Thoughts

Having laid out my arguments, I would like to return for a moment to the title of this article: *Why Complexity Matters*. In short, it has been my experience that complexity theory offers us a bridge between computer science and physics, to say nothing of formal mathematics. Physics and computer science are both, at their heart, ways of understanding the world around us. As such, I have found that complexity offers us a language in which to formulate our questions and our pursuits.

In the same way that physicists have come to rely upon group theory and other mathematical tools to frame their understanding and to serve as a language for describing problems, I believe that computer scientists and physicists alike will come to value complexity theory. Thus, while I cannot and do not want to demand that students and researchers expose to themselves to complexity theory, I will readily encourage my peers and teachers alike to partake of this opportunity to connect their fields of research to a broader whole.

In summary, we have seen that complexity theory is, among other things, a language for describing problems in computer science. In particular, complexity theory allows us to evaluate objectively the power of quantum computation, to state claims about the scope of computation allowed in the physical world and to formally describe how simulation relates the complexity of computation and of physical theories.

Since we have seen by way of these examples that computation and the physical world are intrinsically tied, we have that complexity theory is thus also a language for describing problems in physics.

Part IV
Appendices

A Big-O Notation

In computer science, we very often care only for how quickly a problem grows and not for specific details of the hardware that is currently available. A bad algorithm gets *worse* as computers get faster, as the complexity starts to outpace the growth in hardware capabilities. Thus, notation has evolved to reflect this emphasis. We are not concerned, for instance, with the difference between $f(n) = n^2 + 100$ and $g(n) = 2n^2 + 3$. In fact, we say that both $f(n)$ and $g(n)$ are in the set of functions $O(n^2)$. Some authors will write $f(n) = O(n^2)$, but I shall avoid this notation as I find it quite abhorrent.

Definition 2 ($\Omega(\cdot)$, $O(\cdot)$ and $\Theta(\cdot)$)**.** Given functions $f, g : \mathbb{N} \to \mathbb{N}$, we say that $f(n) \in O(g(n))$ if there exists a constant $c \in \mathbb{N}$ such that $f(n) \leq cg(n)$ for all $n \in \mathbb{N}$. Similarly, we say that $f(n) \in \Omega(g(n))$ if there exists a constant $c \in \mathbb{N}$ such that $cf(n) \geq g(n)$ for all $n \in \mathbb{N}$. If $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$, then we write that $f(n) \in \Theta(g(n))$.

We conclude with some examples of big-O notation:

$$\begin{aligned} \ln n &\in O(n) \\ n \ln n,\, n^2 &\in O(n^2) \\ 1000n^4 &\in \Theta(n^4) \\ 2^n &\in \Omega(1) \\ 1 &\in O(2^n) \end{aligned}$$

B The Pumping Lemma

In order to demonstrate that simple models of computation such as the deterministic finite automata cannot accurately model computation in general, we present here the Pumping Lemma for deterministic finite automata, based on the development of [7].

Theorem 3 (Pumping Lemma)**.** *For each language L solved by a deterministic finite automata, there exists $n \in \mathbb{N}$ such that for all strings $w \in L$ such that $|w| \geq L$, we can write w as the concatenation of three other strings, x, y and z, such that the following hold:*

1. *y is not the empty string.*
2. *$|x| + |y| \leq n$.*
3. *For all $k \in \mathbb{N} \cup \{0\}$, the string $xy^kz \in L$, where y^k is the concatenation of y with itself k times.*

Proof. Let L be a language solved by a deterministic finite automata M having n states, and consider some input $w = w_1 w_2 \cdots w_n$. Note that at any point during the processing of w, M can be completely described by noting its present state. Thus, let $q_0, q_1, q_2, \ldots, q_n$ be a sequence of states such that M is in state q_i after reading i symbols from w; in particular, let $q_0 = q_{\text{start}}$. But then, we note that by the pigeonhole principle, the states $\{q_i\}$ are not all distinct, since the sequence $q_0, \ldots, q_n$ is $n+1$ states long, and since there are only n distinct states in M. Thus, let $i, j \in \mathbb{N} \cup \{0\}$ such that $0 \leq i < j \leq n$ and such that $q_i = q_j$. Note that the substring $y = w_{i+1} w_{i+2} \cdots w_j$ leads M in a loop, provided M starts reading y while in state q_i. Thus, we can repeat y as many or as few times as we please without changing the result of M's computation.

More formally, let $x = w_1 w_2 \cdots w_i$, $y = w_{i+1} w_{i+2} \cdots w_j$, and let $z = w_{j+1} w_{j+2} \cdots w_n$. Then, since $i < j$, we have that $|y| > 0$. Moreover, since $|x| + |y| + |z| = n$, we have that $|x| + |y| \leq n$. Finally, since the number of repetitions of y cannot change the computation of M on w, and since M solves L by assumption, we have that since $xy^k z$ is accepted by M for all k, $xy^k z \in L$ for all k. We have thus confirmed all three claims of our theorem. □

Example 4. We can apply the Pumping Lemma to prove an earlier assumption that the language L of balanced parentheses is not solvable by DFAs. To see this, suppose that a DFA M exists such that $L_M = L$. Then, the Pumping Lemma gives us that there exists some n such that we can pump all strings in L that are at least as long as n. In particular, consider $w = (^n)^n$. This string is obviously in L, and so we there exists some substrings x, y, z such that $w = xyz$ and such that $xy^k z \in L$ for all k. In particular, consider $xyyz$. Since w begins with n opening parens, we have that y must be comprised entirely of opening parens, and thus $xyyz$ has strictly more opening parens than xyz. But then, we have not added any closing parens, and so $xyyz \notin L$. This is a contradiction, and so we have that L is not solvable by any DFA.

C Random Access Machines

We have thus far glossed over the connection between Turing machines and the computers we are more familiar with. Thus, in this appendix, we show that a *random access machine*, which we allow to have access to *registers* much like those of conventional computers, can be simulated by a Turing machine without introducing more than a polynomial overhead.

In defining a random access machine, we follow the construction of [3]. We say that a random access machine (RAM) has access to an infinite number of registers $\{r_i\}_{i=0}^{\infty}$, each of which can store an arbitrarily large integer. Each RAM executes a RAM *program* $\Pi = (\pi_1, \pi_2, \ldots, \pi_m)$, where each π_i is an *instruction* to the RAM. An instruction tells the RAM to perform a particular operation. We shall allow the READ, STORE, LOAD, ADD, SUB, HALF, JUMP, JPOS, JZERO, JNEG and HALT instructions, as per Table 1. Some instructions take an *operand*, which is an integer by which the semantics of the instruction are parameterized.

Name	Description
READ(j)	Sets r_0 to r_j.
READ$'(j)$	Sets r_0 to r_{r_j}.
STORE(j)	Sets r_j to r_0.
STORE$'(j)$	Sets r_{r_j} to r_0.
LOAD(x)	Sets r_0 to x.
ADD(x)	Sets r_0 to $r_0 + x$.
SUB(x)	Sets r_0 to $r_0 - x$.
HALF	Sets r_0 to $\lfloor r_0/2 \rfloor$.
JUMP(k)	Sets κ to k.
JPOS(k)	If $r_0 > 0$, sets κ to k.
JZERO(k)	If $r_0 = 0$, sets κ to k.
JNEG(k)	If $r_0 < 0$, sets κ to k.
HALT	Halts execution.

Table 1: Instructions allowed by a RAM.

Note that in defining these instructions, we consider r_0 differently from all other registers. In that sense, r_0 is a *work register*, also known as an *accumulator*.

All instructions other than the jump instructions and HALT increment a special register κ, called the *program counter*. At each step, the RAM executes the instruction π_κ. When given an input x, represented as a finite sequence of integers $(x_1, x_2, \ldots, x_n)$, we initially set registers r_1 through r_n so that $r_i = x_i$. When and if a RAM halts, we consider its output to be the contents of r_0. Moreover, we shall consider the language L solved by some program Π to be the set $L = \{\, x \mid \Pi(x) = 1 \,\}$, provided Π halts on all inputs.

Though a proof of such is beyond the scope of this appendix, it can be shown that for any language L solved by a RAM program Π in no more than $f(n)$ instructions for any input of length n, L can be solved by a seven-tape Turing machine[8] with a time complexity in $O\left(f^3(n)\right)$. The basic idea is that we use four tapes to store the current state of the RAM and three tapes to implement each instruction. The first tape will be read-only and set to the input to Π. The second tape will hold the value in each register, stored as a list of binary numbers separated by some delimiter symbol. The third tape will hold a binary representation of κ. Finally, the fourth tape will hold the index of the register currently being looked for on the second tape.

A similar proof would show that RAMs can simulate TMs of time complexity $f(n)$ using $O(f(n))$ instructions. Thus, up to a polynomial overhead, we have that RAMs and TMs are identical in computational power. It is easy to see that dealing with Turing machines is often much more convenient, since we need not consider a plethora of distinct instructions when proving things, but instead we may deal with a transition function in the abstract. This, together with the fact that polynomials are closed under function composition, shows why we are

[8]See Section D.3 for a construction of multi-tape TMs.

justified in using Turing machines to construct P, along with other complexity classes.

D Turing Machine Variations

The Turing machine model outlined in Section 2.3 is but one of a set of variations of the same basic idea. We can modify our model of a Turing machine so that it has multiple tracks, multiple tapes, or has a tape that extends to infinity in both directions.

D.1 Multi-track

One of the more basic variations that we can make to a Turing machine is to include a tape with two separate "tracks," each of which has its own independent memory cells. We would still like the read/write head to be able to point only at one tape position at a time, however. We shall write the description of such a multi-track Turing machine with n different tracks as $M = (Q, \Sigma, \Gamma, \delta)$ where $\delta : Q \times \Gamma^n \to Q \times \Gamma^n \times \{\leftarrow, \bullet, \rightarrow\}$ is the new transition function. This gives us a hint as to how to make an equivalent single-track Turing machine: we let $M' = (Q, \Sigma, \Gamma', \delta')$, where $\Gamma' = \Gamma^n$ with the transition function $\delta'(q, \gamma_1\gamma_2 \cdots \gamma_n) = (q', \gamma_1'\gamma_2' \cdots \gamma_n', d)$, where $\delta(q, \gamma_1, \gamma_2, \ldots, \gamma_n) = (q', \gamma_1', \gamma_2', \ldots, \gamma_n', d)$. The transformed machine M' obviously finishes in exactly the same amount of time as M, since each application of δ corresponds to exactly one application of δ'. This shows us that multiple tracks do not afford us any additional computational power, and so we may use them with impunity whenever it makes our life simpler.

D.2 Bidirectional

Many authors choose to define Turing machines to have tapes that extend to infinity in both directions, rather than just the one. Using the multi-track construction of the previous section, we can now show concretely that a bidirectional-tape machine has access to no more computational power than a unidirectional model.

Given a machine $M = (Q, \Sigma, \Gamma, \delta)$ having a bidirectional tape, we can construct an 2-tape machine $M' = (Q', \Sigma, \Gamma, \delta')$ that simulates M by "folding" the bidirectional tape about some origin. Denote the special symbol past the left edge of the tape on M' by $\rhd$ (since M' is a 2-track machine, the left edge symbol will technically be $\rhd \times \rhd$, but for brevity we shall denote the edge by $\rhd$). Let $Q' = Q \cup \overline{Q}$, where $\overline{Q}$ is a copy of Q that indicates that directions are reversed. Then, specify δ' such that $\delta'(q, \rhd) = (\overline{q}, \rhd, \rightarrow)$, where $\overline{q}$ is a the matching state for q in $\overline{Q}$ (or in Q if $q \in \overline{Q}$) that indicates that directions are reversed. For all other symbols, let $\delta(q, \gamma) = (q', \gamma', d)$ and specify that:

$$\begin{aligned} \delta'(q, \gamma, \gamma_2) &= (q', \gamma', \gamma_2, d) \\ \delta'(\overline{q}, \gamma_1, \gamma) &= \left(\overline{q'}, \gamma_1, \gamma', -d\right) \end{aligned}$$

where $-d$ indicates that the direction d is reversed. Note that this construction never causes more than twice as many applications of δ', since in the worst case scenario, every transition of M crosses the origin (at $\triangleright$), and since only in the case of an origin crossing do we introduce even one extra transition. Since we do not consider constant factors in our $O(\cdot)$ notation, we can thus rest assured that bidirectional tapes are no more powerful than unidirectional tapes.

D.3 Multi-tape

It is often useful to consider Turing machines which have multiple infinite tapes on which to store data. For instance, the simulation of a random access machine by a Turing machine described in Section C depended crucially on the availability of multiple tapes, each with its own read/write head. Formally, we consider an n-tape Turing machine to be a tuple $M = (Q, \Sigma, \Gamma, \delta)$ where the transition function has the form $\delta : Q \times \Gamma^n \to Q \times \Gamma^n \times \{\leftarrow, \bullet, \rightarrow\}^n$.

To simulate the behavior of an n-tape machine M using a single-tape machine $M' = (Q', \Sigma, \Gamma', \delta')$, we construct M' with $2n$ tracks, partitioned into pairs. Each pair of tracks represents one of M's tapes, with the first track containing only a special marker symbol at the position of M's read/write head for that tape, and with the second track containing the full contents of the corresponding tape in M. To simulate an application of δ, M' scans its tape to find the symbols currently being read by each of M's tapes and stores these symbols in its finite state. We can do this by letting Q' have the form $Q \times \Gamma^n$. Once the read symbols have been found, δ' transitions M' to a state corresponding to the next state of M and proceeds to write out the new symbols to its tape.

This process involves no more than two full scans of the tape of M' for each iteration of M. Since in the worst case, M' cannot have a tape longer than the number of iterations of M, this means that the time complexity of M' is in $O(f^2(n))$, where $f(n)$ is the time complexity of M. A more formal version of this argument can be found in [3].

References

[1] A. Ben-Hur, H. T. Siegelmann, and S. Fishman. A theory of complexity for continuous time systems. *Journal of Complexity*, 18(1):51–86, 2002.

[2] Andrew Chi-Chih Yao. Quantum Circuit Complexity. *Proceedings of IEEE FOCS'93*, pages 352–361, 1993.

[3] Christos H. Papadimitriou. *Computational Complexity.* Addison Wesley Longman, 1994.

[4] David Deutsch. Quantum theory, the Church-Turing principle and the universal quantum computer. *Proceedings of the Royal Society of London*, 400:97–117, 1985.

[5] Donald Knuth. Structured Programming with Goto Statements. *Computing Surveys*, 6(4):261–301, 1974.

[6] Edward Fredkin and Tomasso Toffoli. Conservative Logic. *International Journal of Theoretical Physics*, 21:219–253, 1982.

[7] John Hopcroft, Rajeev Motwani, and Jeffery D. Ullman. *Automata Theory, Languages and Computation*. Addison Wesley, 3 edition, 2006.

[8] L. Susskind. The World as a Hologram. *Journal of Math Physics*, 36:6377–6396, 1995.

[9] Manindra Agrawal, Neeraj Kayal, and Nitin Saxena. PRIMES in P. *Annals of Mathematics*, 160(2):781–793, 2004.

[10] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information*. Cambridge University Press, 2000.

[11] N. V. Vinodchandran. A note on the circuit complexity of PP. *ECCC*, 2004.

[12] Peter Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, 26(5):1484–1509, 1997.

[13] Raphael Bousso. Positive vacuum energy and the N-bound. *Journal of High Energy Physics*, 18 December 2000.

[14] S. Toda. On the computational power of PP and P. *Proceedings of IEEE FOCS'89*, pages 514–519, 1989.

[15] Scott Aaronson. NP-complete problems and physical reality. *ACM SIGACT News*, March 2005.

Attributions

Figure 8a is a sample provided with the qasm2circ software package. Used under the terms of the GNU GPL version 2.