Quantum Complexity Theory Sevag Gharibian
Summer 2019, University of Paderborn

Lecture 1: Classical Complexity Theory Review

The future is not laid out on a track. It is something that we can decide, and to the extent that we do not violate any known laws of the universe, we can probably make it work the way that we want to.
— Alan Kay

Contents

1 Introduction

Welcome to Quantum Complexity Theory! In this course, we ask the central question: What *quantum* computational resources are required to solve certain tasks? These tasks might involve "classical" problems, such as factoring large integers or solving systems of linear equations, or "quantum" problems, such as computing properties of physical systems in Nature. Along the way, we shall encounter both "expected" complexity classes, such as quantum generalizations of P and NP, as well as "unexpected" classes with no interesting classical analogue, such as QMA(2), whose exact computational power remains frustratingly unknown. In a nutshell, our aim will be to flesh out an "answer" to Alan Kay's statement above — what does Nature allow us to do, or not do, from a computational perspective?

The first step in this journey is the all-important question:

Why should quantum complexity theory be interesting?

There are many reasons for this; let us focus on arguably the most famous one. Every day, millions of online consumers rely on cryptosystems such as Rivest-Shamir-Adleman (RSA) to keep (say) their credit card information secure online. There is, however, a theoretical problem with this — the RSA scheme has never been *proven* secure. Rather, it is *presumed* to be secure, assuming the mathematical problem of factoring large integers (denoted FACTOR) is "hard". Unfortunately, noone really knows how hard FACTOR really is. For example, it is neither known to be efficiently solvable classically (i.e. it is not known to be in P), nor is it provably intractable (e.g. FACTOR is not known to be NP-hard). In 1994, Peter Shor thread the needle (and in the process, shocked the theoretical computer science community) by demonstrating that a *quantum* computer can solve FACTOR efficiently. Thus, if large-scale quantum computers can be built, then RSA is broken and your credit card information is no longer secure. Moreover, we are faced with the confounding question of how the existence of Shor's algorithm should be interpreted:

- *Is Shor's algorithm evidence that FACTOR is actually in P, and that we as a community have failed to be clever enough to find good classical algorithms for it?*

- *Or perhaps the algorithm hints that quantum computers can solve vastly more difficult problems, including NP-complete ones?*
- *And maybe neither of these holds — maybe quantum computers can outperform classical ones on "intermediate" problems such as FACTOR, but not genuinely "hard" ones such as NP-hard problems?*

To date, we do not know which of these three possibilities holds. What we *do* know, is that we are at a very exciting crossroads in computer science history. Thanks to Shor's algorithm, one of the following statements must be *false*:

1. The Extended Church-Turing Thesis is true.
2. FACTOR cannot be solved efficiently on a classical computer.
3. Large-scale universal quantum computers can be built.

We will return to this crossroads at the end of the lecture, but for now let us note that the fact that quantum computers appear "more powerful" than classical ones is not entirely surprising. Indeed, the physics community has long known that simulating quantum systems with classical computers appears to require an inevitable exponential overhead in time. And as far back as the 1970's, visionaries such as Stephen Wiesner began to realize that quantum information seemingly allows one to do the impossible, such as create physically uncloneable "money" using quantum states. Thus, the question "What *quantum* computational resources are required to solve certain tasks?" indeed appears extremely interesting.

Scope of course. Our focus in this course is quantum complexity theory, strictly construed. The field itself is far too large to capture in a single course; thus, we shall aim to cover many fundamental algorithms and complexity classes, while striking a balance between approachable difficulty for a Masters course and more advanced results. Along the way, useful mathematical frameworks such as Semidefinite Programming (SDPs) and the multiplicative weights method will be covered, which find applications in areas beyond quantum computation.

There are unfortunately many complexity-related topics which, due solely to time constraints, we will likely be unable to cover. A select sample of these include quantum Turing machines, additional complexity classes (such as Quantum Statistical Zero Knowledge (QSZK), Stoquastic MA (StoqMA), Quantum Multi-Prover Interactive Proofs (QMIP), QMA with Polynomial-Size Advice (QMA/qpoly), complexity classes with exponentially small promise gap (such as QMA_{exp}), Quantum Polynomial-Time Hierarchies (QCPH and QPH), etc...), advanced topics from the subarea of Quantum Hamiltonian Complexity such as area laws, tensor networks, and perturbation theory gadgets for simulating local interaction terms, advanced circuit-to-Hamiltonian constructions such as 1D constructions and space-time constructions, undecidable problems in quantum information such as detecting spectral gaps of local Hamiltonians, other candidate models for demonstrating "quantum supremacy" such as IQP and DQC1, classical simulations of quantum circuits which minimize T-gate count, and so forth.

Resources. The entry point for any course on complexity theory should arguably be the "Complexity Zoo", which provides brief information on over 535 (as of 2019) complexity classes:

`https://complexityzoo.uwaterloo.ca/Complexity_Zoo`.

Existing surveys focusing on quantum complexity theory include:

- Quantum NP - A Survey (Aharonov and Naveh, 2002): `https://arxiv.org/abs/quant-ph/0210077`,
- Quantum Computational Complexity (Watrous, 2008): `https://arxiv.org/abs/0804.3401`,
- QMA-Complete Problems (Bookatz, 2014): `https://arxiv.org/abs/1212.6312`,
- Quantum Hamiltonian Complexity (Gharibian, Huang, Landau, and Shin, 2015): `https://arxiv.org/abs/1401.3916`.

Prerequisites. Before we can make sense of ideas in quantum complexity theory, we require a brief review of the basics of both classical complexity theory and quantum computing. This course assumes some familiarity with both areas (such as a previous introductory course, particularly for quantum computation), but we shall attempt to make the content as self-contained as possible.

Acknowledgements. We thank Jannes Stubbemann for catching a number of typos in these notes.

2 Notation

Throughout this course, the symbols $\mathbb{N}, \mathbb{R}, \mathbb{Z}, \mathbb{C}$ refer to the sets of natural, real, integer, and complex numbers, respectively. We define $\mathbb{N}$ to include 0, and $\mathbb{Z}^+, \mathbb{R}^+$ to be the sets of non-negative integers and real numbers, respectively. The set of all n-bit strings is denoted $\{0,1\}^n$. The terminology $|x|$ is overloaded: If $x \in \mathbb{C}$, then $|x| = \sqrt{xx^*}$ (for x^* the complex conjugate of x), and if x is a string, $|x|$ denotes the length (i.e. number of symbols) of x. The notation $[n]$ denotes set $\{1, \ldots, n\}$. We use $:=$ to denote a definition.

3 Classical complexity theory

The "basis" of complexity theory is the pair of complexity classes Polynomial-Time (P) and Non-Deterministic Polynomial-Time (NP). To define these, we must recall the definition of a Turing machine. Note that while the remainder of this lecture works exclusively with the Turing machine model of classical computing, one can equivalently work in the *circuit* model, which we will switch to in Lecture 2 when discussing quantum computation. Nevertheless, the notion of a Turing machine will be crucial to defining "uniformly generated quantum circuit families" in Lecture 2, and is good to keep in mind for the remainder of the course. As this section is a review, we shall move rather quickly.

3.1 Turing machines

In order to formally study the nature of computing, we require a standard model of what it means to "compute". This role is played by the *Turing machine*, an idealized model for computing proposed by Alan Turing in 1936. In hindsight, the model is remarkably basic — it consists of a "brain" or "control" unit which makes decisions, an infinitely large memory stored on an infinitely long tape, and a "head" to read and write data from the tape. A formal definition is given below.

Definition 1 (Turing Machine (TM))**.** *A Turing Machine is given by a 7-tuple* $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$*, defined as:*

- Q *- a finite set denoting the* states *the TM can be in.*
- Σ *- a finite set denoting the input* alphabet*, i.e. set of symbols which can be written on the tape to encode the input before the TM starts. (*Σ *is assumed not to contain the special blank symbol* $\sqcup$*.)*
- Γ *- a finite set denoting the tape alphabet, i.e. set of symbols which can appear on the tape. Note* $\Sigma \subseteq \Gamma$ *and* $\sqcup \in \Gamma$*.*
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ *- the* transition function*, which takes in the current state of the TM and tape symbol under the head, and outputs the next state, tape symbol to write under the head, and whether to move the head left (L) or right (R) one cell on the tape.*
- $q_0 \in Q$ *- the designated start state of the TM.*
- $q_{\text{accept}} \in Q$ *- the designated accept state of the TM, i.e. the TM halts and accepts immediately upon entering state* q_{accept}*.*

- $q_{\text{reject}} \in Q$ - *the designated reject state of the TM, i.e. the TM halts and rejects immediately upon entering state* q_{reject}.

With this model in place, we can crucially define what is meant by "one step of a computation". Namely, this means our TM reads the current tape symbol under the head, updates the state of the TM, writes a new symbol under the head, and moves the head left or right precisely one cell.

Recall that a TM now operates as follows. Before it begins, we assume the input to the computational problem to be solved is written on the tape using symbols from Σ. Once the TM machine starts, it executes a sequence of computation steps. If the TM enters either state q_{accept} or q_{reject}, the TM halts and accepts or rejects, respectively. Without loss of generality, we may assume that if the TM also produces some output string z upon halting, then only z appears on the output tape once the machine halts. Notably, a TM need not ever enter either q_{accept} or q_{reject}, in which case it runs forever.

Exercise. Sketch a TM which, given input string $x \in \{0,1\}^n$, outputs 0 if x contains only zeroes, and outputs 1 otherwise. (In other words, the TM computes the OR function on all bits of x.) How many steps does your algorithm asymptotically take (i.e. state your answer in Big-Oh notation)? Does it always halt? A formal definition in terms of Q, δ, etc..., is not necessary; simply sketch the TM's action at a high level.

Languages and decision problems. Throughout this course, we assume strings consist of bits, i.e. $x \in \{0,1\}^n$. Recall that a *language* L is a subset of strings $L \subseteq \{0,1\}^*$, and that computational problems are often modeled as follows: Fix a language L. Given input $x \in \{0,1\}^*$, is $x \in L$? This is known as a *decision* problem, since the answer is either YES or NO.

We say a TM M *decides* a language L if, given any input $x \in \{0,1\}^n$, M halts and accepts if $x \in L$ (rejects if $x \notin L$). Note this definition says nothing about the worst-case number of steps required by M to decide L. Recall that there do exist *undecidable* languages, which encode computational problems which provably cannot be solved by any TM in a finite amount of time. Although undecidable problems do occur in quantum complexity, in this course we will unfortunately not have a chance to pursue such directions.

Exercise. Recall that the canonical undecidable problem is the Halting Problem (HALT): Given as input a description of a TM M and an input x, does M halt on input x? Can you prove that HALT is undecidable?

Exercise. Although HALT cannot be solved on a TM, on a real-life computer, HALT *can* be solved - why? What is the crucial difference between a real life computer and a TM which allows this distinction?

Significance. Two important remarks are in order. First, although the TM model nowadays seems "obvious", in that modern computers essentially work in a similar fashion (with the notable exception of solid state drives, which unlike electromechanical drives, do not have a physical "read-write head"), it is precisely due to Turing's inspiration that modern computers developed as they did. Moreover, although the informal concept of "algorithm" has existed for millennia (e.g. Euclid's algorithm for computing the greatest common divisor, dating to around 300 B.C.), the TM model finally gave a standard formal definition of what "algorithm" means. This, in turn, allowed one to show certain computational problems simply cannot be solved by *any* computer in finite time (modulo our next remark below).

The second important remark involves the longstanding *Church-Turing thesis*, which roughly says that TMs truly capture the "full power of computing". Namely, according to the thesis, any model of computation can be simulated by a TM (slightly more precisely, if there exists a mechanical process for computing some function f, then there also exists a Turing machine computing f). Thus, if our goal is to understand the nature of computing (as it is here), it suffices to restrict our attention to TMs. Note, however, that this is a *thesis*, and not a formally proven theorem — indeed, there does not appear to be any way to prove the Church-Turing thesis! And while it is generally believed that the Church-Turing thesis is on solid footing, a less well-known strengthening of the thesis, known as the *Extended Church-Turing thesis*, appears to potentially be in peril due to quantum computation (see section 3.4).

3.2 P and NP

P and NP are arguably the most famous exports of computational complexity theory. Roughly, they denote the sets of decision problems which can be solved efficiently on a TM, and verified efficiently on a TM, respectively. In this course, we define these complexity classes as follows.

Definition 2 (Polynomial-Time (P))**.** *A language $L \subseteq \{0,1\}^*$ is in P if there exists a (deterministic*[1]*) TM M and fixed polynomial $r_L : \mathbb{N} \mapsto \mathbb{R}^+$, such that for any input $x \in \{0,1\}^n$, M halts in at most $O(r_L(n))$ steps, and:*

- *(Completeness/YES case) If $x \in L$, M accepts.*
- *(Soundness/NO case) If $x \notin L$, M rejects.*

Note that we define "efficient" computation, i.e. P, as taking at most *polynomially* many steps in the input size.

Exercise. Are all languages in P decidable?

Exercise. Can the choice of polynomial r_L also depend on the input, x? Why or why not?

A simple example of a decision problem in P is integer multiplication (MULTIPLY): Given as input $x, y \in \mathbb{Z}$, and threshold t, is the product $xy \leq t$? (All inputs are implicitly specified in binary.) The reverse of this problem is the integer factorization problem (FACTOR): Given $z \in \mathbb{Z}^+$ and threshold t, does z have a non-trivial factor $x \leq t$? In other words, do there exist integers $1 < x \leq t$ and y such that $xy = z$?

Exercise. You have implicitly known since elementary school that MULTIPLY is in P. Sketch a TM which decides MULTIPLY (i.e. recall your childhood algorithm for multiplication).

Exercise. If we change the definition of MULTIPLY to "given input $x, y \in \mathbb{Z}$, output the product xy", is this problem still in P?

There is a reason why multiplication is typically taught before factorization in school - unlike MULTIPLY, FACTOR is *not* known to be in P. It can, however, be efficiently *verified* by a TM if the TM is given some help in the form of a "short"/polynomial-size proof. Namely, given candidate factors $x, y \in \mathbb{Z}^+$ of z, a TM can efficiently check whether $x \leq t$ and whether $xy = z$. This is exactly the phenomenon encountered with puzzle games like Sudoku — filling out or *solving* the puzzle is difficult, but if someone gives you the solution, *verifying* the solution is correct is easy.

Definition 3 (Non-Deterministic Polynomial-Time (NP))**.** *A language $L \subseteq \{0,1\}^*$ is in NP if there exists a (deterministic) TM M and fixed polynomials $p_L, r_L : \mathbb{N} \mapsto \mathbb{R}^+$, such that for any input $x \in \{0,1\}^n$, M takes in a "proof" $y \in \{0,1\}^{p_L(n)}$, halts in at most $O(r_L(n))$ steps, and:*

- *(Completeness/YES case) If $x \in L$, then there exists a proof $y \in \{0,1\}^{p_L(n)}$ causing M to accept.*
- *(Soundness/NO case) If $x \notin L$, then for all proofs $y \in \{0,1\}^{p_L(n)}$, M rejects.*

Thus, FACTOR $\in$ NP. Note the only difference between P and NP is the addition of the polynomial-size proof y. Also, recall the original definition of NP was in terms of *non-deterministic* TMs; here, we shall omit this view, as non-determinism appears generally less useful of a concept in quantum complexity theory.

[1]In this course, all TM's are assumed to be deterministic.

The Boolean Satisfiability Problem. Finally, recall a canonical problem in NP is the Boolean k-Satisfiability Problem (k-SAT): Given a Boolean formula $\phi : \{0,1\}^n \mapsto \{0,1\}$ in conjunctive normal form (CNF), is ϕ satisfiable? Here, a k-CNF formula has form (we demonstrate with an explicit example for $k = 3$)

$$\phi := (x_1 \vee \overline{x_2} \vee x_4) \wedge (\overline{x_2} \vee \overline{x_5} \vee x_3) \wedge \cdots \wedge (x_7 \vee x_4 \vee x_{11}),$$

where $\vee$ and $\wedge$ denote the logical OR and AND functions, respectively, and $\overline{x_i}$ is the negation of bit x_i. More generally, a k-CNF formula is an AND over "clauses" of size k, each of which is an OR over precisely k literals (a literal is either x_i or $\overline{x_i}$ for some variable x_i). We say ϕ is *satisfiable* if there exists $x \in \{0,1\}^n$ such that $\phi(x) = 1$.

Exercise. Why is k-SAT in NP? Does this still hold if k scales with n?

Exercise. Why is k-SAT not obviously in P?

Exercise. Is P $\subseteq$ NP? Is NP $\subseteq$ P?

3.3 The Cook-Levin Theorem, NP-completeness, and Reductions

What makes P versus NP such a prolific framework is that it captures many (if not most) practical computational problems of interest — from finding shortest paths between pairs of vertices in graphs, to solving systems of linear equations, to scheduling on multiple processors, all of these problems are in NP (with the first two also being in P). Indeed, the number of studied problems in NP number in the *thousands*.

But if the generality of P and NP is the "Eiffel Tower" of complexity theory, then the Cook-Levin Theorem is its "Arc de Triomphe" — for the Cook-Levin theorem showed that, remarkably, to solve *every* problem in NP efficiently (and thus most practical computational problems we might care about), it suffices to just solve a single problem — the Boolean Satisfiability problem. Coupled with Karp's "21 NP-complete problems", the community quickly realized that many longstanding problems which appeared difficult to solve, from 3-SAT to CLIQUE to KNAPSACK, are all one and the same problem as far as complexity theory is concerned.

We now recall the formal theory of NP-completeness, for which we first require the notion of a reduction.

Reductions. Intuitively, recall a reduction "reduces" one problem A to another problem B, so that if we can solve B, we can also solve A. A real-life example of this which you likely applied already at age two is this — the problem of opening the fridge (whose handle is too high up for a toddler!) can be reduced to getting a parent's attention; the ability to do the latter allows the toddler to accomplish the former. In this case, the reduction is being computed by the toddler; let us replace our toddler with a Turing machine.

Definition 4 (Reduction). *Let $A, B \subseteq \{0,1\}^*$ be languages. A* reduction *from A to B is a computable*[2] *function $f : \{0,1\}^* \mapsto \{0,1\}^*$, such that for any input $x \in \{0,1\}^*$, $x \in A$ if and only if $f(x) \in B$. If such a reduction exists, we write $A \leq B$. We further say the reduction is* polynomial-time *if the TM computing it runs in time polynomial in the input size, $|x|$; in this case, we write $A \leq_p B$.*

In essence, a reduction maps an instance x of problem A to an instance $f(x)$ of problem B such that x is a YES instance of A if and only if $f(x)$ is a YES instance of B.

Exercise. Define language ADD as the set of $(k+1)$-tuples $(x_1, \ldots, x_k, t) \subseteq \mathbb{Z}^{k+1}$, such that $x_1 + \cdots + x_k \leq t$. Give a reduction from MULTIPLY to ADD. (This reduction is, in fact, likely how you first learned the concept of multiplication in elementary school.)

[2] A *computable* function f is one for which there exists a TM M which, given input x, halts and outputs $f(x)$.

Many-one/Karp versus Turing reductions. The notion of reduction in Definition 4 is arguably the most natural one, as it maps a single instance of A to a single instance of B (note the mapping need not be a bijection). In the literature, this is typically called a *many-one*, *mapping*, or *Karp* reduction. A generalization of many-one reductions which also appears in quantum complexity theory is a *Turing* reduction. To define the latter, we refine our view of a Karp reduction. Suppose we have access to a "black box" or oracle O_B which magically decides B for us[3]. Then, a many-one reduction can be viewed as mapping instance x of A to instance $f(x)$ of B, and then immediately feeding $f(x)$ to the oracle O_B to obtain the answer to the question: Is $x \in A$? In particular, we call the oracle O_B precisely once, and immediately return its output as our final answer. In a polynomial-time Turing reduction, we relax the latter constraint so that O_B can be called polynomially many times, and we may postprocess the answers to these queries using any polynomial-time computation we like before returning a final answer.

Exercise. Sketch a polynomial-time Turing reduction from the problem of finding a desired number in a sorted list of numbers to the problem of comparing which of a pair of integers is larger. What overhead does your reduction asymptotically require, if we treat each number as occupying a single cell of the TM's tape for simplicity? Does this overhead change if we allow the TM *random-access*, i.e. the ability to jump to any desired cell in a single time step?

There is formal evidence that many-one and Turing reductions are not the same in terms of "reduction power". A specific example in quantum information theory where this distinction seems to occur is the *Quantum Separability Problem*: Given a classical description of bipartite quantum state, we wish to determine if the state is entangled. This problem is strongly NP-hard under polynomial time Turing reductions, but it remains open for over 15 years whether a similar result holds under polynomial-time many-one reductions. (This phenomenon holds also for certain genuinely "quantum" problems — the CONSISTENCY problem[4] only has a known QMA-hardness proof under Turing reductions. Here, QMA is a quantum generalization of NP, and will be discussed in future lectures.)

NP-completeness. With reductions in hand, we can recall the definition of *NP-hard*, which in turn allows us to define NP-complete.

Definition 5 (NP-hard). *A language $B \subseteq \{0,1\}^*$ is NP-hard if, for any language $A \in NP$, there exists a polynomial-time reduction from A to B.*

Definition 6 (NP-complete). *A language $B \subseteq \{0,1\}^*$ is NP-complete if it is both in NP and NP-hard.*

Note that the definition of completeness for NP extends naturally to many other complexity classes. For example, we may replace NP with an arbitrary class C and define a lanuage as C-complete if it is both in C and C-hard. However, a subtlety in doing so is that depending on the class C, we may wish to use a different notion of reduction than "polynomial-time". For example, for P-complete problems, the notion of reduction used is sometimes a "logspace" reduction, meaning it should use at most a logarithmic amount of space. In this course, we shall clearly state (unless the instructor forgets) if a notion of reduction other than polynomial-time is used.

Intuitively, recall that NP-complete problems are the "hardest" problems in NP, and as far as polynomial-time reducibility is concerned, they are all equivalent in difficulty. (The latter statement is *not* generally true if one considers other definitions of "difficulty", such as *approximability* instead of exact solvability. As far as, say, approximability is concerned, some NP-complete problems really are "harder" than others.) In general, a C-complete problem for complexity class C should be thought of as "capturing" the difficulty of class C.

[3]This is typically formalized via an oracle Turing machine. Such a TM has an extra tape, denoted the oracle tape, on which it gets to place a string corresponding to a query input. Once the TM's query input is ready on the query tape, the TM "calls" the oracle O_B, which replaces the query input with the query output on the query tape in a single step.

[4]CONSISTENCY is roughly defined as follows. The input is a set of k-qubit reduced density operators ρ_i acting on subsets of qubits $S_i \subseteq [n]$, respectively. The question is whether there exists a global density operator ρ acting on all n qubits, such that for each i, $\text{Tr}_{[n]\setminus S_i}(\rho) = \rho_i$.

The Cook-Levin Theorem. The canonical NP-complete problem has traditionally been 3-SAT, as established by the Cook-Levin theorem (which showed NP-completeness for SAT) and Karp's followup work (which showed NP-completeness for 3-SAT). Here, SAT is the generalization of k-SAT in which the clauses can be of arbitrary size.

Theorem 7 (Cook-Levin theorem)**.** *SAT is NP-complete.*

An example of a reduction. Let us recall how reductions work in practice by showing 3-SAT is NP-hard under polynomial-time mapping reductions.

Lemma 8. 3*-SAT is NP-complete.*

Proof. Containment in NP is trivial. We sketch a reduction from SAT, which by Theorem 7 is NP-complete. The idea is as follows: Let $\phi : \{0,1\}^n \mapsto \{0,1\}$ be an input SAT instance. We sketch how to efficiently construct a 3-SAT instance $\phi' : \{0,1\}^n \mapsto \{0,1\}$, such that ϕ is satisfiable if and only if ϕ' is.

Let c be an arbitrary clause of ϕ of size s, so that we may write

$$c = (l_{c,1} \vee l_{c,2} \vee \cdots \vee l_{c,s})$$

for arbitrary literals $l_{c,i}$ (recall a *literal* is a variable which may or may not be negated). We show how to map this to a pair of clauses c_1 and c_2 of sizes $s-1$ and 3, respectively. Introduce a new variable y_c, and define

$$c_1 = (l_{c,1} \vee l_{c,2} \vee \cdots \vee l_{c,s-2} \vee y_c) \qquad c_2 = (l_{c,s-1} \vee l_{c,s} \vee \overline{y_c}).$$

Note that the new variable y_c will only occur in this particular pair of clauses. This operation is clearly polynomial-time, and repeating this until all clauses have size 3 also takes polynomial-time. We leave correctness as an exercise.

Exercise. Suppose $x_{c,1} \cdots x_{c,s}$ satisfies clause c above. Prove that there is a setting of y_c such that $x_{c,1} \cdots x_{c,s} y_c$ satisfies both c_1 and c_2. Similarly, show that if $x_{c,1} \cdots x_{c,s}$ does not satisfy clause c above, then $x_{c,1} \cdots x_{c,s}, y_c$ cannot satisfy both c_1 and c_2 regardless of the setting of y_c. □

Tip. For any reduction proof, your proof should be structured as follows. First, give the construction for the reduction itself. Second, argue what overhead or resources the reduction requires. Third, prove correctness of the reduction.

Exercise. Prove that 3-SAT remains NP-hard even if we restrict each variables x_i to appear at most twice in the 3-SAT formula ϕ. (Hint: Simulate equality constraints on pairs of variables.)

3.4 The Extended Church-Turing Thesis

Let us close this lecture by revisiting the "crossroads" stated in Section 1. Namely, we have reviewed the notions of P, NP, reductions, and NP-completeness. All of this relied heavily on the Turing machine model, which is "validated" by the Church-Turing thesis.

Due to Shor's factoring algorithm, we now know that one of the following statements must be *false*:

1. The Extended Church-Turing Thesis is true.
2. FACTOR cannot be solved efficiently on a classical computer.
3. Large-scale universal quantum computers can be built.

Formally proving any of these statements to be false would be a major step forward in theoretical computer science (although the first statement is arguably the least "earth-shattering" of the set, as it is less widely accepted). Here, the *Extended* Church-Turing Thesis says that any "reasonable" or "physically realizable" model of computation can be simulated by a Turing machine with at most polynomial overhead.

Any two of the three statements above together now imply the third statement is false. For example, if FACTOR is intractable classically and large-scale quantum computers can be built, then the Extended-Church Turing thesis is false, since quantum computers *can* solve factoring efficiently. This is a best-case scenario for quantum computation. In the opposite direction, if the Extended Church-Turing Thesis is true and FACTOR is not classically tractable, then it must be that even though quantum computers can *theoretically* solve FACTOR, unfortunately a large-scale quantum computer cannot be built in practice.

Of course, here it should be noted that even in this "worst-case" latter scenario, all is not lost. Non-universal (i.e. special-purpose) quantum computers may nevertheless be possible on a large scale — indeed, there are already companies such as ID Quantique who build special-purpose quantum devices for cryptographic tasks such as quantum key distribution! (Another important point to bear in mind is that the development of quantum computation and information has had a dramatic impact on our understanding of theoretical physics, such as in the areas of entanglement theory, condensed matter physics, black holes, channel theory, etc... . This gained knowledge is unaffected by whether or not large-scale universal quantum computers can be built.)