CMSC 657: Introduction to Quantum Information Processing Lecture 8

Instructor: Daniel Gottesman

Fall 2024

1 Complexity Theory

Recall we were discussing complexity theory and had made some definitions:

Definition 1. *$\{0,1\}^*$ is the set of all bit strings of any length. The size $|x|$ of $x \in \{0,1\}^*$ is the length of the bit string. A* language *is a subset of $\{0,1\}^*$.*

Definition 2. *An algorithm $A(x)$* decides *language L if $\forall x \in \{0,1\}^*$, $A(x) = 0$ (i.e., no) if $x \notin L$ and $A(x) = 1$ (yes) if $x \in L$. x is called an* instance *of L (a "yes instance" if $x \in L$ and a "no instance" if $x \notin L$). In a* promise problem*, the instance is drawn from a proper subset of $\{0,1\}^*$.*

To give the complexity of deciding a language L, we should use the complexity of the *best* (shortest) algorithm $A(x)$ that decides L. But how exactly do we want to quantify complexity?

We will consider *worst case* complexity, which means that the algorithm $A(x)$ must succeed for all inputs. It may work faster on some inputs than others, and we calculate the run time on the worst possible input of a given size.

The exact number of steps depends on details of the machine, as noted earlier. We shouldn't care about constant factors in the complexity. However, more dramatic changes in the computational model (e.g., Turing machine vs. circuit model, 1-D vs. 3-D circuits, etc.) can change the number of steps using in an algorithm by polynomial blow-ups $O(g(|x|)) \to O(\text{poly}(g(|x|)))$. In order to avoid this problem, we prefer to define the complexity of a problem in a way that is robust to polynomial changes in the length of circuits.

We usually talk about the complexity of an algorithm using $O(n)$ ("big O") notation, as well some other related notation:

Definition 3. *A function $f(n)$ is $O(g(n))$ (or $f(n) \in O(g(n))$) if there exist constants C, n_0 such that $f(n) \leq Cg(n)$ for all $n > n_0$. $f(n) \in o(g(n))$ if for all constant C, there exists a constant n_0 such that $f(n) < Cg(n)$ for all $n > n_0$. $f(n) \in \Omega(g(n))$ if there exist constants C, n_0 such that $f(n) \geq Cg(n)$ for all $n > n_0$. $f(n) \in \Theta(g(n))$ if $f(n) \in O(g(n))$ and $f(n) \in \Omega(g(n))$.*

This reflects that constant factors (and addition of constants) are not really meaningful when talking about complexity. Then $O(g(n))$ says that asymptotically for large n and ignoring constant factors, $f(n)$ grows no faster than $g(n)$, whereas $o(g(n))$ says $f(n)$ grows strictly slower. Then $\Omega(g(n))$ is the opposite, saying that $f(n)$ grows at least as fast as $g(n)$ (again, asymptotically and ignoring constant factors), and $\Theta(g(n))$ says that it grows at the same rate as $g(n)$.

These notations are good for describing the running time on a particular family of machines, but they don't take into account polynomial factors that can arise with very different architectures. For that, we go to complexity classes.

1.1 Complexity Classes and the Church-Turing Thesis

Definition 4. *A* complexity class *is a set of languages. P is the set of languages that can be decided in time $f(|x|)$ when f is a polynomial. If $L \in P$, we say that L can be decided by an* efficient *algorithm; colloquially, it is "easy".*

The complexity class P is robust to polynomial changes in complexity, since if f and g are both polynomials, then $f(g(n))$ is a polynomial as well, albeit a larger one.

PRIMES is an example of a language in P, but the algorithm that decides it is definitely non-trivial and was not discovered until 2002. Not all languages are in P by a long shot, which is already a somewhat profound statement: It says that some computational problems are simply too hard to be solved in a reasonable amount of time. Indeed, a much stronger statement is true: There are some languages which cannot be decided by any algorithm at all! The standard example is the Halting Problem (given the code for a program and an input, does it halt in a finite amount of time, or does it keep running forever?).

This raises an important question: If different types of computational models can take different amounts of time to solve a problem, how do we even know that they can solve the same problems? Maybe the set of undecideable problems depends on the computational model.

Conjecture 1 (Church-Turing Thesis)**.** *All physically reasonable models of computation have the same set of decideable functions.*

The Church-Turing Thesis appears to be true. People have come up with models of computation that decide more functions than the circuit model (or Turing machine, etc.), but they are unreasonable in one way or another. For instance, analog computers with infinite precision can solve some undecideable problems, but infinite precision is unrealistic.

Recall that we said that different architectures and designs of a computer usually lead to polynomial blow-ups. That leads the to Strong Church-Turing Thesis, which is a a refinement of the Church-Turing thesis which says that classifying the difficulty of problems up to polynomial factors makes sense.

Conjecture 2 (Strong Church-Turing thesis)**.** *The complexity of languages differs by only polynomial factors between physically reasonable models of computation.*

Note that quantum computers do not violate the Church-Turing thesis. Given a quantum algorithm for a problem, we can run essentially the same algorithm classically by multiplying together the $2^n \times 2^n$ matrices representing the unitary gates in the circuit. However, that this involves not a polynomial blow-up in computation time, but an *exponential* blow-up in the number of bits vs. qubits and classical gates vs. quantum gates.

Indeed, quantum computers apparently invalidate the Strong Church-Turing thesis! This is a pretty profound statement, and I think there are some computer scientists, even today, who doubt quantum computers can ever work primarily for this reason. The consequence is that the rules of computation fundamentally depend on which theory of physics you are working with.

But is that even enough? Perhaps different kinds of quantum computers differ by more than polynomial factors. Luckily, this seems not to be the case. All the reasonable quantum models people have looked at are equivalent to the usual quantum circuits we have been discussing.

Conjecture 3 (Quantum Church-Turing thesis)**.** *Any physically reasonable model of computation can be simulated in polynomial time on a quantum computer.*

1.2 BPP and BQP

We need to define a new "polynomial-time" complexity class that takes account of quantum algorithms. Because quantum algorithms inherently involve randomness, we need to take that into account in our evaluation of the success of the algorithm. To understand how to do that, let us first define a classical randomized complexity class that is otherwise analogous to P.

Definition 5. *BPP (which stands for "bounded probability polynomial") is the set of languages L for which there exists a classical algorithm $A_L(x)$ such that*

1. *$A_L(x)$ runs in a time polynomial in $|x|$*
2. *If $x \in L$, then $A_L(x) = \text{yes}$ with probability $\geq 2/3$*
3. *If $x \notin L$, then $A_L(x) = \text{no}$ with probability $\geq 2/3$.*

The B "bounded" refers to the fact that the probabilities for yes and no outcomes are well-separated. The number 2/3 doesn't really matter and changing it to any other constant gives the same complexity class (since we can repeat the algorithm multiple times to increase our confidence). However, if we used probabilities like $1/2 - 2^{-|x|}$ and $1/2 + 2^{-|x|}$, we would have a problem distinguishing the yes and no cases.

We don't know for sure, but probably most complexity theorists believe that P = BPP. Randomness does seem to help in practice, but only by reducing the degree of the polynomial needed to find the answer. Why only polynomial factors? One argument that P should be the same as BPP goes as follows: Use a pseudorandom number generator to create a long pseudorandom string from a small seed and use them in place of the random numbers in the randomized algorithm. The algorithm should behave basically the same way with the pseudorandom numbers and the truly random numbers. Then try this for all possible values of the seed and accept the language if the randomized algorithm accepts for at least 2/3 of the seed values. The reason this is just a conjecture is that we don't know for sure if there exist pseudorandom functions with the right properties so that this always works correctly.

We can then define a quantum complexity analogously:

Definition 6. *BQP (which stands for "bounded quantum polynomial") is the set of languages L for which there exists a* quantum *algorithm $A_L(x)$ such that*

1. *$A_L(x)$ runs in a time polynomial in $|x|$*
2. *If $x \in L$, then $A_L(x) = \text{yes}$ with probability $\geq 2/3$*
3. *If $x \notin L$, then $A_L(x) = \text{no}$ with probability $\geq 2/3$.*

Here, the bounded probability aspect is really needed. You can define a class called EQP which requires a quantum algorithm which works with success probability 1, but it is a bit artificial, since you often end up having to fine-tune the states to get exactly success probability 1, and seems to be significantly smaller than BQP.

1.3 NP

How do we show if a problem is easy or hard? To show it is easy (i.e., in P or in BQP), we simply need to come up with an efficient classical or quantum algorithm to decide the language. Showing it is hard is much more challenging because even when an algorithm exists, it might be quite difficult to find it. So if we fail to find an algorithm, maybe that just means we are not clever enough. One thing we can do is to try to classify types of problems that are harder than P or BQP.

Definition 7. *NP (which stands for "nondeterministic polynomial") is the class of languages L such that there exists a "verifier" algorithm $V(x, w)$ such that*

1. *$V(x, w)$ runs in a time polynomial in $|x|$ for all x, w such that $|w| = \text{poly}(|x|)$*
2. *$\forall x \in L$, $\exists w_x$ such that $V(x, w_x) = \text{yes}$ and $|w_x| = \text{poly}(|x|)$*
3. *$\forall x \notin L$, $\forall w$ such that $|w| = \text{poly}(|x|)$, $V(x, w) = \text{no}$.*

w_x is the witness *for a yes instance x.*

That is, given a language in NP, for every yes instance, there is a witness that allows x to pass the check by the verifier function, whereas any no instance will fail the verifier function for all possible witnesses.

Example 1 (k-SAT, or k-satisfiability)**.** *The instance is a binary encoding of a formula on n variables x_i. The formula consists of the AND of $m = \mathrm{poly}(n)$ clauses C_i. Each clause C_i is the OR of k atoms, each of which is either x_a or NOT x_a for some a. The Boolean formula $f(\mathbf{x}) = C_1$ AND C_2 AND $\ldots C_m$ is a yes instance of k-SAT if $\exists\{x_i\}$ (a "satisfying assignment") such that $f(\mathbf{x}) = \mathrm{TRUE}$.*

It is easy to see that k-SAT is in NP: The witness for a yes instance is simply the satisfying assignment, and the verifier is to evaluate the formula on the satisfying assignment. In fact, we know that not only is k-SAT in NP, but it is one of the hardest problems in NP (for $k \geq 3$).