CMSC 858L: Quantum Complexity

Instructor: Daniel Gottesman

Spring 2023

1 Lecture 1: Introduction to Complexity Theory

For this lecture, you can get the syllabus on the class web page, and the basics of complexity theory you can find in many places, for instance Wikipedia (articles on Computational complexity theory, Turing machine) or in any standard classical complexity theory textbook. Another useful website is the Complexity Zoo (https://complexityzoo.net).

1.1 About the Class

Complexity theory is the study and classification of the difficulty of computational problems. In quantum complexity theory, we add new quantum-inspired complexity classes and study their relationships to classical complexity classes and problems. We also study the complexity of quantum-related computational problems.

This class will give an overview of the major results and techniques of quantum complexity. The prerequisite is CMSC 657 or an introduction to quantum computing at a similar level. If you have significant previous experience with classical complexity theory but a weaker background in quantum information, you may be able to follow the class with some additional catch-up work; please let me know if this applies to you.

I am going to be starting the complexity theory from the beginning, so that is not a prerequisite. However, I will be going through the classical stuff rather quickly.

We will be using lots of different kinds of math in this class. In many cases, I will just be writing down theorems and hope you can follow from that. If we need to make more extensive use of a specific kind of math, I will either write up a few notes or include a longer discussion in class.

Grades will be determined by 35% problem sets, 35% project, 30% final exam.

- The final exam will probably be take-home.
- The problem sets will be once per 2 weeks, turned in on Gradescope. They will be available on the course web page: <https://www.cs.umd.edu/class/spring2023/cmsc858L/>. *Note:* No penalty for up to 3 days late assignment; but don't expect further extensions without a good reason. The point of this policy is so that neither you nor I need to worry about minor extensions or slight delays, not to push the deadline back 3 days. The problem sets may be challenging, so don't leave them until the last minute.
- The project will be a group project to summarize the main points from some relevant research papers. Or you can try to do some original research if you want. There will be a written aspect to the project and a presentation in the last couple of lectures.

My office hours will be 11-noon on Tuesdays in my office Atlantic 3251, or you can make an appointment at some other time for an in-person or Zoom discussion. The TA Amadeo will have office hours 9-10 AM on Zoom. Questions outside of class or office hours should be posted on Piazza. Only ask a private question if your question is specific to you or if you need to include parts of a problem set solution to ask it.

I am not aware of a textbook that covers this material, so you will need to rely on the lecture notes, which will be posted on the class web page. I will try to put references for specific topics in the lecture notes in case you need more detail.

1.2 Overview of Complexity Classes

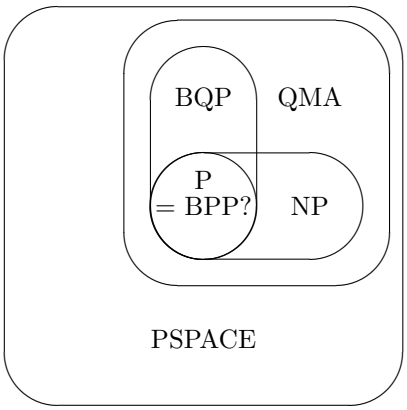
In complexity theory, we try to classify the difficulty of computational problems into *complexity classes*. The most common thing to study is the difficulty of *decision problems*, which are computational problems with yes/no (or true/false, or 1/0) answers. However, we will also see other types of problems in this class, including *functional problems*, where the answer could take a wider range of values, and *sampling problems*, in which the answer is not a single value but one or more samples drawn from a particular probability distribution.

Perhaps surprisingly, many different problems tend to have similar levels of difficulty. Therefore, it makes sense to subsume them into complexity classes of problems which require a similar amount of resources. For instance, one of the most standard and basic complexity classes is P, the class of (decision) problems that can be solved in polynomial time (as a function of the problem size) on a classical computer. Other complexity classes can be defined by assuming other types of devices or resources.

While we can define a lot of different complexity classes and in some cases identify many problems of a similar level of difficulty, unfortunately, we are not generally able to prove that complexity classes are different from each other except in extreme cases. Instead, complexity theorists rely on other forms of evidence, such as separations relative to oracles, reductions, and conditional results that say that if one pair of complexity classes are distinct, then other classes must be distinct as well. We will talk about all of these later in the class.

Now let me briefly introduce some of the main complexity classes we will be using in this class. The full definitions will come later. All of these are decision classes, i.e., classes of decision problems.

A close relative of P is BPP, the class of problems solvable (with high probability) in polynomial time on a classical computer using randomness. Most people today seem to think P= BPP, but that is not known for certain. Then we get BQP, which is the class of problems solvable (with high probability) using a quantum computer in polynomial time. Since quantum measurement is random, quantum complexity classes tend to involve probabilities as well, so the natural "poly-time" quantum class is analogous to BPPrather than to P. Since a quantum computer can run any classical computation (with maybe a minor slow-down to make it reversible), $\mathrm{P} \subseteq \mathrm{BQP}$. Obviously the point of building quantum computers is that we (meaning most quantum information researchers) believe that $\mathrm{BQP} \neq \mathrm{P}$, but that is not proven.



Another very important classical complexity class is NP, which is, roughly speaking, the class of problems whose answers can be checked in polynomial time on a classical computer. Certainly $\mathrm{P} \subseteq \mathrm{NP}$, but whether

they are equal or not is a famous open problem. We believe that NP $\not\subseteq$ BQP and BQP $\not\subseteq$ NP for reasons that we will discuss later in the class (or at least, we will discuss some of the reasons). That is, the two class NPand BQPare incomparable.

The best quantum analogue of NPis QMA, which is, again roughly speaking, the class of problems whose answers can be checked in polynomial time on a quantum computer. The reason for the difference in name is that the definition of NPinvolves deterministic things, so QMAis actually more analogous to a randomized version of NPcalled MA. We have BQP $\subseteq$ QMA and NP $\subseteq$ QMA. And if BQPand NPare truly incomparable then that implies that QMAis strictly bigger than each of them. This is an example of a conditional separation of complexity classes. But we can't actually prove any of these things, so it is possible that all the classes I have mentioned so far are equal.

Moving into even larger classes, we have PSPACE, the class of problems that can be solved on a classical computer with possibly exponential time but using only a polynomial amount of scratch space. At this point, it would be natural to wonder about a quantum computer with a polynomial amount of scratch space, but it turns out that the corresponding complexity class is equal to PSPACE. PSPACEis probably larger than QMAand the other classes we have discussed, but again, we cannot prove that.

1.3 Languages and Complexity Classes

Now let's start to define things rigorously.

Definition 1. *Let $\{0,1\}^*$ be the set of all bit strings of any length. A* language *is a subset of $\{0,1\}^*$. A specific bit string $x \in \{0,1\}^*$ is an* instance *of the language L under consideration. If $x \in L$, it is a* yes instance. *If $x \notin L$, it is a* no instance.

We interpret the elements of a language as the inputs to a decision problem that have a "yes" answer. In some cases it is convenient to consider languages using a larger alphabet, i.e., more choices per symbol than just 0 and 1, but it doesn't lead to fundamentally different definitions.

Example 1. *We can interpret bit strings as non-negative integers written in binary. Then the language PRIMES is the set of bit strings x that have the "yes" answer to the question: Is x prime? That is, the language PRIMES is the set of prime numbers $\{10, 11, 101, 111, 1011, 1101, 10001, \ldots\}$.*

101 (i.e., 5) is a yes instance for PRIMES and 110 (i.e., 6) is a no instance.

Definition 2. *A* promise *is a subset $P \subseteq \{0,1\}^*$. A* promise problem *is a language L which is known to be a subset of some non-trivial promise P. The yes instances of L are again the elements of L but the no instances are only elements of $P \setminus L$. Bit strings which are not in P are not considered instances of L.*

Definition 3. *An algorithm* decides *a language L if on input x, it outputs 0 when $x \notin L$ and 1 when $x \in L$. When L is a promise problem, the algorithm only needs to give the correct output when the input is an instance of L; if somehow fed a non-instance, any answer is allowed.*

Definition 4. *A* decision complexity class *is a set of languages.*

Generally we are interested in complexity classes defined as a set of languages that share a particular property, generally that they can be decided by algorithms using a particular amount of some sort of resource (such as time or space on a given machine).

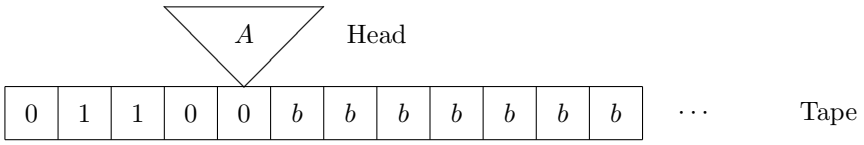
1.4 Turing Machines

I mentioned "algorithms" but haven't defined that. Also, how do we quantify what resources are needed? We need some sort of concrete computational model. There are many, but the usual starting point is a Turing machine. I am not going to use Turing machines much in this class, but I want to define them to give a rigorous starting point. After that we'll switch to the circuit model, but there is one thing that we'll need to define using Turing machines.

Definition 5. *A* Turing machine *is a device with a* tape *which can hold an arbitrary sequence of symbols drawn from an* alphabet Σ *and a* head *which has an internal state in the set of symbols K. Σ contains a special symbol b ("blank"), and the initial state of the tape contains only finitely many non-b symbols. K contains special symbols Y ("halt with answer yes") and N ("halt with answer no"). There is also a* transition rule $\delta : K \times \Sigma \to K \times \Sigma \times \{Left, Right, Stay\}$*, with the property that $\delta(Y, x) = (Y, x, Stay)$ and $\delta(N, x) = (Y, x, Stay)$. At any given time step, the head is located at a location i on the tape containing symbol S_i. If the state of the head is H, let $\delta(H, S_i) = (H', S_i', D)$. Then at the next time step, the state of the head is changed to H', the symbol at the ith location is changed to S_i', and the head moves in the direction indicated by D: i.e., if $D = Left$, the head moves to location $i - 1$; if $D = Right$, the head moves to location $i + 1$; and if $D = Stay$, the head stays in the same location. If the head state is Y or N, we say that the Turing machine has* halted*, and if it first reaches the state Y or N at time step t, we say that it* halts after t steps.

There are a lot of alternative definitions with small variations and also different kinds of Turing machines with bigger changes, such as multiple tapes.

When a Turing machine halts, the state of the head and all locations on the tape are fixed and do not change for future times. Note that a Turing machine doesn't always halt; it is straightforward to come up with Turing machines that run forever.



Often, we are interested in Turing machines for which $0, 1 \in \Sigma$. When the input state is a bit string x followed by only b symbols, we say that x is the *input* of the Turing machine.

Definition 6. *A Turing machine* decides *a language L if on input $x \in L$, the Turing machine halts with head state Y and on input $x \notin L$, the Turing machine halts with head state N. A Turing machine* computes the function $f(x)$ *if on input x, the Turing machine halts and the state of the tape when it halts is $f(x)$ followed by only b symbols.*

On the face of it, we need a different Turing machine for each language we want to decide or each function we want to compute. However, there exist universal Turing machines which can simulate any other Turing machine. This means that they take as input a description of the Turing machine T to be simulated as well as an input x to T and give an output of which part is the output of T on input x. (There may be additional material in the output of the universal Turing machine, such as the description of T.)

Theorem 1. *Let T be a universal Turing machine and let $h(x) = 1$ if T halts on input x and $h(x) = 0$ if T does not halt on input x. Then there is no Turing machine that decides h.*

This is the famous uncomputability of the halting problem, and it shows that even universal Turing machines have their limits and there are things that they cannot compute. There are alternative models of computation, but they all seem to either be unrealistic in some way or another (e.g., continuous computation with infinite precision) or to have the same limits as Turing machines. This is known as the Church-Turing Thesis:

Claim 1 (Church-Turing Thesis)**.** *All physically reasonable models of universal computation compute the same set of functions and decide the same set of languages.*

(There are weaker models of computation, of course, that can compute fewer things, but none that compute more.)

Note that the Church-Turing thesis still applies to quantum computers, because we haven't made any restrictions about time or space needed to compute things. That means that a classical computer can perfectly simulate a quantum computer by multiplying out the $2^n \times 2^n$ unitary matrices for each quantum gate in an n-qubit system. But without making any restrictions about time or space, we can't sensibly talk about complexity.

Definition 7. *If x is a bit string, let $|x|$ be the length of the bit string. A Turing machine T* decides a language L in time $t(|x|)$ *if T decides L and it halts in at most $t(|x|)$ steps on input x for all x. A Turing machine T* decides a language L using space $s(|x|)$ *if T decides L, leaves the input locations unchanged, and during the course of the computation never has more than $s(|x|)$ additional tape locations changed from the initial value b. There are similar definitions for computing a function f in time $t(|x|)$ or space $s(|x|)$.*

When we want to limit the space used by the computation, we make the input read-only. This lets us consider cases where the space used is less, even much less, than the size of the input; only we care about is the amount of *extra* space used.

Note that we compute resources (time and space) as a function of *the number of bits* of the input, not the actual value of the input.

There are also *quantum Turing machines*, but we will not discuss them in this class (unless someone wants to pick that for a project topic).