CMSC 858L: Quantum Complexity
Unit 1: Overview and Core Concepts

Instructor: Daniel Gottesman

Fall 2026

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. I expect that most students will have taken 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 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/fall2026/cmsc858L/>. *Note:* No penalty for up to 1 day late assignment. 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 a day. 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 Chaitanya will have office hours on Wednesday noon-1 PM in his office Atlantic 3257. 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.

2 Overview of Complexity Classes

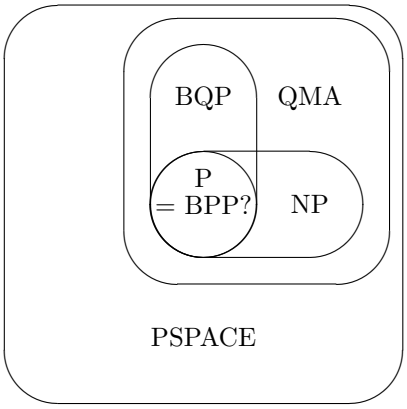
In complexity theory, we try to classify the difficulty of computational problems into *complexity classes*. The most common thing to study is the difficulty of *decision problems*, which are computational problems with yes/no (or true/false, or 1/0) answers. However, we will also see other types of 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 BPP rather than to P. Since a quantum computer can run any classical computation (with maybe a minor slow-down to make it reversible), P $\subseteq$ BQP. Obviously the point of building quantum computers is that we (meaning most quantum information researchers) believe that BQP $\neq$ 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 P $\subseteq$ 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 classes NP and BQP are incomparable.

The best quantum analogue of NP is 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 NP involves deterministic things, so QMA is actually more analogous to a randomized version of NP called MA. We have BQP $\subseteq$ QMA and NP $\subseteq$ QMA. And if BQP and NP are truly incomparable then that implies that QMA is 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. PSPACE is probably larger than QMA and the other classes we have discussed, but again, we cannot prove that.

3 Basic Definitions of Complexity Theory

3.1 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 $\{10, 11, 101, 111, 1011, 1101, 10001, ...\}$.*

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).

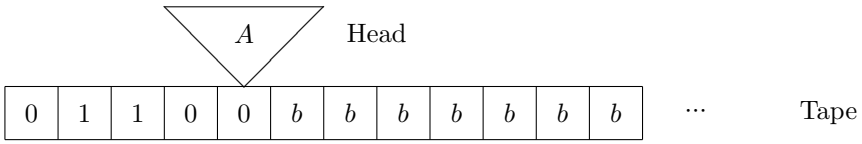
3.2 Turing Machines

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. We are not going to use Turing machines much in this class, but I want to define them to give a rigorous starting point. We will use Turing machines to define the circuit model, and mostly work with that.

Definition 5. *A* Turing machine *is a device with a* tape *which can hold an arbitrary sequence of symbols drawn from a finite* alphabet Σ *and a* head *which has an internal state in the finite 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 $s(|x|)$. For computing f in space $s(|x|)$, do not count the size of the output towards the space used, and the output bits should be write-only.*

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; we only care about the amount of *extra* space used. Similarly, we also do not count the output, which when computing a function could be long.

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.

4 Polynomial Time

4.1 P

Once we start thinking about how much time is used and try to come up with Turing machines to decide different types of problems, we rapidly notice that some are harder than others. One complication, however, is that different definitions of a Turing machine will take a different amount of time to decide the same language, and different universal Turing machines will also take a different amount of time. Moreover, there are alternative models of computation, and those also take a different amount of time. Sometimes, the change in the time to decide a language works differently for different languages, which means that on one universal Turing machine, language L may be easier (meaning it takes less time to decide) than M, whereas on a different universal Turing machine, language M is easier than L. This is apparently a problem for the goal of complexity theory: Classifying problems by difficulty doesn't make sense, or is at least a much less appealing problem, if that classification inherently depends on the details of which Turing machine or model of computation you have. Luckily, there is a strengthening of the Church-Turing thesis that rescues us:

Claim 2 (Strong Church-Turing Thesis)**.** *If a physically reasonable model of computation decides language L with resources $f(|x|)$, then a particular universal Turing machine can decide L with resources $g(f(|x|))$, where $g(y)$ is a polynomial function.*

Here resources could be either space or time. In particular, a Turing machine can simulate the behavior of other reasonable models of computation with at most a slowdown which is a polynomial function of the resources needed in the alternative model. Similarly, other reasonable candidates for alternative models of universal computation can simulate a Turing machine with at most a polynomial resource cost.

Now, the Strong Church-Turing Thesis, unlike the original Church-Turing thesis, seems to be falsified by quantum computers, which cannot be simulated in polynomial time by a classical Turing machine. However, it still holds true for a wide variety of different models of computation, which makes it still useful in guiding us to good complexity definitions. In particular, we want to consider complexity classes which don't depend sensitively on which Turing machine or model of computation we are working with.

In order to discuss complexity, we like to use big-O notation and its relatives:

Definition 8.

- *A function* $f(x) = O(g(x))$ *if* $\exists C, x_0$ *such that* $f(x) \leq Cg(x)$ *for all* $x > x_0$. *("Big-O")*
- *A function* $f(x) = o(g(x))$ *if* $\forall C\, \exists x_0$ *such that* $f(x) < Cg(x)$ *for all* $x > x_0$. *("Little-o")*
- *A function* $f(x) = \Omega(g(x))$ *if* $\exists C, x_0$ *such that* $f(x) \geq Cg(x)$ *for all* $x > x_0$. *("Big-omega")*
- *A function* $f(x) = \Theta(g(x))$ *if* $f(x) = O(g(x))$ *and* $f(x) = \Omega(g(x))$. *("Big-theta" or just "Theta")*

That is, big-O says that f grows at most as fast as g asymptotically, little-o says that it grows slower than g, big-omega says that it grows at least as fast as g, and big-theta says that f and g grow at the same speed asymptotically.

Definition 9. *A function* $f(x) = O(\text{poly}(g(x)))$ *if* $f(x) = O(h(g(x)))$ *for some polynomial* $h(x)$.

Definition 10. *The complexity class P is the set of languages that can be decided by a Turing machine in a time* $O(\text{poly}(|x|))$ *on input* x.

Note that this definition allows arbitrary Turing machines, but by the Strong Church-Turing Thesis, we can fix a single universal Turing machine and we will always get the same complexity class P. The name of this one is straightforward: the P stands for "polynomial."

At this point, we are about ready to abandon Turing machines. Let us switch to the circuit model of computation:

Definition 11. *A* t-bit gate *is a function acting on* t *bits with an output of at most* t *bits in length. Let* G *be a set of* t*-bit gates. A* circuit C *using the gate set* G *consists of a sequence of* layers. *Layer* 0 *contains a bit string* x_0*, the* input *to the circuit, and layer* i *contains a bit string* x_i *which is a function of* x_{i-1} *computed using gates drawn from* G *acting on disjoint subsets of bits. The* depth *of the circuit* C *is the number of layers in the circuit not counting the input. The* size *of* C *is the total number of gates used in* C *(summed over all layers). The final layer contains the* output *of the circuit. In many cases, the first bit in the last layer is used as the output of the circuit instead of the full bit string on the last layer.*

A complication when talking about complexity of a circuit is that we need different circuits depending on the size of the input since the gate structure is for a specific number of bits. We therefore need to talk about a *family* of circuits C_n, one for each input size n.

It seems natural to give an alternate definition of P using families of circuits C_n such that C_n has size $O(\text{poly}(n))$. However, this is not quite right; this gives us a different complexity class known as P/poly. The problem is that one can sneak a lot of computation in by switching the circuit for different n in a complicated way. For instance, suppose that C_n is a circuit that ignores the details of the input and simply outputs $h(n)$, where h is the halting function. Then this circuit family can solve the halting problem on input n by simply feeding it an input with n bits. So P/poly actually contains uncomputable problems!

To fix this definition and make sense of circuit complexity, we need to restrict attention to circuits which are derived in a regular and not overly complicated way. The standard way to do this is to consider uniform circuits:

Definition 12. *A family* C_n *of circuits is* uniform *if there exists a Turing machine that on input* 1^n *(i.e., a string of* n 1*'s) outputs* C_n *using* $O(\log n)$ *space.*

This will be one of our last needs for Turing machines. Note another point about this definition, which is the use of an input given in *unary*. This is a technical trick; since we are defining complexity as a function of the input *size* rather than the input itself, in the rarer cases when we want something that is actually a function of the input instead, we generally should put the input in unary so that its length is equal to its value.

Theorem 2. *P is equal to the set of languages that can be decided by a uniform family of circuits of size* $O(\text{poly}(|x|))$ *for instance* x.

Note that we are using the circuit size (number of gates) rather than the depth (which would be the number of time steps if the circuit is parallelized) to quantify the time complexity of the circuit. There are a separate set of complexity classes based on restricting the depth of circuits rather than the size.

4.2 Functional Problems and Sampling Problems

When we want to think about functional problems, we need to define different complexity classes, which we can do by adding the letter F in front of the name. For instance, FP is the class of functions that can be computed by a uniform family of circuits of size $O(\text{poly}(|x|))$ for input x.

Technically, complexity classes for functional problems are distinct from the corresponding complexity classes for decision problems, but conceptually and in terms of actual difficulty, they are closely related. This is because functional problems and decision problems can often be converted into each other.

For instance, consider the functional problem for factoring: Given N, let $f(N) = p$, where p is the smallest prime factor of N. This can be converted into a decision problem: Given (N, c), does N have a prime factor less than c? Clearly, if you can solve the functional problem, you can solve the decision problem by finding the smallest prime factor and comparing it with c. Conversely, if you can solve the decision problem, you can also solve the functional problem using $O(\log N)$ calls of your decision algorithm by doing a binary search on c: That is, start with $c = \sqrt{N}$ (since the smallest prime factor is always less than $\sqrt{N}$ unless N is prime), then call the decision problem, halving c each time until you get the answer "no." Once that happens, choose c to be in the middle of the interval between the last "no" answer and the last "yes" answer. This will rapidly narrow down the possible values of the lowest prime factor.

Another common way to relate decision problems and functional problems is to have the decision problem output the ith bit of the functional value.

There are, however, certain unusual cases where there is a distinction between the difficulty of a functional problem and the most natural corresponding decision problem.

Sampling problems are a different story. In a sampling problem, you want to output a sample bit string from a probability distribution. An example we will see in class is outputting the n-qubit measurement result from a family of quantum circuits. There is still sometimes a related decision problem, for instance if the first output bit is more likely to be 0 or 1, or perhaps some other function of the output string.

Being able to solve the sampling problem then may let you solve the decision problem: Repeatedly sample from the probability distribution and see if the first bit is 0 more often or 1 more often. This will work if your decision problem gives you a promise that there is a significant difference in probability between the two outcomes, but in some complexity classes that is not the case, in which case you would need very many samples to reliably distinguish the probabilities of 0 and 1.

And importantly, solving the decision problem definitely does not give you the ability to solve the sampling problem. Even being able to exactly compute the probability of 0 or 1 for any single bit does not necessarily tell you anything about the *correlations* between different bits. For instance, if you know that each bit in your n-bit output has a 50% chance of being 0 and a 50% chance of being 1, that still does not let you determine if the overall probability distribution outputs n uniformly random bits or outputs all 0s 50% of the time and all 1s 50% of the time.

4.3 BPP

What if we want to consider randomized algorithms? The first change we need to make is to allow randomness in the circuits (or Turing machine), of course. But the second change is to note that we should allow for algorithms that don't succeed with 100% probability. Alternatively, we could have algorithms which always succeed but take a variable amount of time depending on the random result.

If we have a randomized algorithm for a decision problem that doesn't always succeed, it will have some probability of outputting 1 and some probability of outputting 0. For a decision problem, one of these answers is correct (depending on if the instance is in the language or not), and a reasonable algorithm should have a higher probability of giving the correct answer than the incorrect one. (For functional problems and sampling problems, the considerations are a bit different.)

One important thing we need to bear in mind is that if we have an algorithm which gives one outcome with probability $1/2 + \epsilon$ and the other outcome with probability $1/2 - \epsilon$ for small ϵ, it will be difficult to tell by running the algorithm which outcome has the higher probability. Indeed, it will generally take $\Omega(1/\epsilon)$

trials to have a reasonable chance of guessing which is which. Given the ethos that we care about polynomial amounts of resources, this means that we must ensure that ϵ is at least $1/\text{poly}(n)$ on inputs of size n.

The standard approach is to let ϵ be a constant. In particular, we have the following definition for BPP:

Definition 13. *Let BPP be the set of languages L such that there exists a polynomial-time randomized algorithm $A(x)$ (i.e., a uniform family of polynomial-size circuits including the ability to generate random bits) with the following properties: For any instance x,*

1. *If $x \in L$, then* $\text{Prob}(A(x) = 1) \geq 2/3$.
2. *If $x \notin L$, then* $\text{Prob}(A(x) = 0) \geq 2/3$.

The name "BPP" standards for "bounded probability polynomial," since the definition puts non-trivial bounds on the acceptance and rejection probabilities.

If a language is in BPP, then with a few runs of the algorithm satisfying these conditions, we can rapidly determine if $x \in L$ or $x \notin L$, statistically speaking. We can't necessarily determine it with 100% probability, of course. Note, though, that $\text{P} \subseteq \text{BPP}$. This is clear from the algorithm, since a deterministic P algorithm satisfies both conditions (with probability 1 instead of 2/3). As I mentioned earlier, many people believe that actually $\text{P} = \text{BPP}$ using derandomization ideas.

5 Quantum Turing Machines and Circuits

5.1 Quantum Mechanics and Quantum Gates

This is intended to be a quick review of the quantum formalism and the quantum gates we will be using. (Pure) quantum states are vectors in a complex Hilbert space. We write them with angle brackets called *kets*. General pure are usually labelled with Greek letters, particularly ψ and ϕ, and can be expanded in a standard basis labelled by bit strings: e.g.,

$$|\psi\rangle = \alpha|00\rangle + \beta|01\rangle + \gamma|10\rangle + \delta|11\rangle \tag{1}$$

$$|\phi\rangle = a|00\rangle + b|01\rangle + c|10\rangle + d|11\rangle \tag{2}$$

The inner product of two quantum states is written using *bras* and kets: e.g.,

$$\langle\phi|\psi\rangle = a^*\alpha + b^*\beta + c^*\gamma + d^*\delta. \tag{3}$$

Here, the star indicates complex conjugation. In principle, quantum states need to be normalized $\langle\psi|\psi\rangle = 1$, but we often don't bother to do this explicitly.

One place where normalization is important is when measuring; a measurement in the standard basis has a probabilistic result, giving outcome a with probability equal to the absolute value squared of the component of a, e.g., the probability of getting outcome 00 if we measure $|\psi\rangle$ above in the standard basis for both qubits is $|\alpha|^2$. The normalization then ensures that the total probability sums to 1.

If we have multiple qubits or multiple quantum systems with Hilbert spaces H_1 and H_2, the appropriate Hilbert space for the combined system is the tensor product $H_1 \otimes H_2$. If $\{|\psi_i\rangle\}$ is a basis of H_1 and $\{|\phi_j\rangle\}$ is a basis of H_2, then $\{|\psi_i\rangle \otimes |\phi_j\rangle\}$ (running over all possible i and j separately) is a basis for $H_1 \otimes H_2$.

Quantum gates are unitary operations $UU^\dagger = I$ (which ensures they preserve the inner product and therefore total probability). Here $\dagger$ represents the Hermitian adjoint, which is the complex conjugate transpose

of the matrix representation. Standard example quantum gates are

$$H = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \quad \text{Hadamard} \tag{4}$$

$$R_\theta = \begin{pmatrix} e^{-i\theta} & 0 \\ 0 & e^{i\theta} \end{pmatrix} \quad \text{Phase gate} \tag{5}$$

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \quad \text{Controlled-NOT} \tag{6}$$

Another example gate is the Toffoli gate:

$$Tof|a, b, c\rangle = |a, b, c \oplus ab\rangle. \tag{7}$$

Technically, quantum states are vectors in a *projective Hilbert space*, because they are only defined up to a global phase (i.e., one that applies equally to all states). Note that changing by a global phase does not affect the probability of a measurement outcome.

We will be using some additional more advanced properties of quantum states, but this should get you started. Certainly you should know about mixed states and density matrices as well as pure states. If this material is unfamiliar, you should review it.

5.2 Quantum Turing machines

The main step in turning a classical Turing machine into a quantum Turing machine is to replace the head and tape states by Hilbert spaces. Thus, the state of the head is in K, which is a finite-dimensional Hilbert space. The state of the tape is in $T = \Sigma \otimes \Sigma \otimes \Sigma \otimes \cdots$, the infinite tensor product of the finite-dimensional Hilbert space Σ. There are some mathematical subtleties in defining infinite tensor products, but in this case they are moot: At any time, the state of the tape is always in $\Sigma^{\otimes n} \otimes |bbb\ldots\rangle$, with only finitely-many non-b symbols. Note that we are still assuming there is a blank state $|b\rangle \in \Sigma$, which is now one of the basis states. We will also assume that $|Y\rangle$ and $|N\rangle$ are basis states in K.

The other important step is to define the transition map $\delta : K \otimes \Sigma \to K \otimes \Sigma \otimes D$. Here D is an additional Hilbert space spanned by the states $\{|\text{left}\rangle, |\text{right}\rangle, |\text{halt}\rangle\}$. The map δ should be unitary (or more precisely, a partial isometry because the output Hilbert space is larger than the input, so it is not invertible on all states of $K \otimes \Sigma \otimes D$). There is one additional condition on δ, which is that it has no amplitude to produce $|\text{halt}\rangle$ in D unless it is also producing $|Y\rangle$ or $|N\rangle$ (or some superposition thereof) in K.

Each step of the Turing machine applies δ between the head Hilbert space K and the tensor factor of T corresponding to the location of the head. Then the D Hilbert space is measured, and the head moves one space left or right if the measurement outcome is $|\text{left}\rangle$ or $|\text{right}\rangle$, respectively. If the D measurement outcome is $|\text{halt}\rangle$, the Turing machine halts, measures K, and outputs the result (which should be either Y or N due to the extra constraint on δ).

Because we are measuring the direction after each step, we can assume the location of the head is a classical variable. I have subsumed the halt condition into the direction register so that halting is also a well-defined classical event. These are again details that don't matter very much — you could have a quantum Turing machine where the head is in a superposition of locations and could be in a superposition of halted and not halted states, which would be more confusing but give the same theory of quantum complexity.

There is one additional subtlety, which is what matrix elements we allow for δ. I will return to this issue shortly when we talk about quantum circuits, so for the moment just assume that δ is a matrix of well-behaved numbers, say rational or algebraic.

5.3 BQP

BQP is the central quantum complexity class, the class which captures what we mean by something that can be efficiently solved on a quantum computer (at least for decision problems). Because quantum measurement is random, we have to apply the same kind of definition as for probabilistic classical algorithms.

There is an additional complication in that we need to specify what kinds of quantum circuits we allow. First, there is a question of how it interfaces with the classical input, the instance x. I will assume the quantum circuit itself depends only on $|x|$ and the exact instance is provided as input in a basis state at the start of the circuit.

Definition 14. *We can define a* quantum circuit *similarly to a classical circuit, except that the gates are quantum operations and they act on qubits rather than bits. Usually we restrict to unitary gates (which means that the number of the output qubits must equal the number of input qubits), plus state preparation locations to introduce extra qubits in standard states (such as $|0\rangle$) and measurement locations which measure in a standard basis (such as the $|0\rangle$, $|1\rangle$ basis) and output the result as a classical bit.*

If $|\psi\rangle$ is a quantum state, let $M(|\psi\rangle)$ be the random variable produced by measuring the first qubit of $|\psi\rangle$ in the standard basis.

Then there is the question of what gates to allow. It would be natural to allow any quantum gates of bounded size. However, this is a definition that would allow us to sneak in extra computational power. For instance, imagine we have a gate R_ϕ for arbitrary ϕ that does a phase rotation by ϕ: $R_\phi|0\rangle = |0\rangle$, $R_\phi|1\rangle = e^{i\phi}|1\rangle$. This is fine as far as it goes, but what if ϕ is an uncomputable number? By applying R_ϕ many times in a phase estimation algorithm, we would be able to find ϕ to arbitrary precision (given enough time), and calculate this uncomputable number. To disallow tricks like this, we should restrict attention to gates that are themselves efficiently computable.

Definition 15. *Let $\mathcal{G}$ be a set of quantum gates acting on a bounded number of qubits (usually either 2 or 3). We say that $\mathcal{G}$ is a* universal *set of gates if for any unitary U, there exists a circuit (of any size) containing gates from $\mathcal{G}$ that realizes the transformation U. $\mathcal{G}$ is* approximately universal *if, for any unitary U and any error ϵ, there exists a circuit composed of gates from $\mathcal{G}$ that realizes a unitary U_ϵ such that $\|U - U_\epsilon\|_{sup} < \epsilon$. $\mathcal{G}$ is* efficiently computable *if every matrix element of every member of $\mathcal{G}$ can be computed to accuracy ϵ in a time $O(\mathrm{poly}(\log 1/\epsilon))$ (on a classical computer).*

In other words, a universal set of gates can exactly reproduce any unitary, whereas an approximately universal set can be merely get close to any unitary.

We will discuss the ins and outs of universal and non-universal sets of gates in more detail once we are done defininig the major complexity classes. For now, just notice that any (exactly) universal gate set must be infinite (since finite products of them give arbitary unitaries) and cannot be efficiently computable (since there exist some unitaries, as discussed above, that are not; and if the gate set can exactly reproduce one of them, it must also contain some noncomputable gates).

With these considerations, we therefore define BQP as follows:

Definition 16. *Fix a particular efficiently computable approximately universal gate set $\mathcal{G}$. Let BQP be the set of languages L such that there exists a uniform family of polynomial-size quantum circuits Q_n with the following properties: For any instance x,*

1. *The circuit $Q_{|x|}$ is composed of gates from $\mathcal{G}$.*
2. *If $x \in L$, then* $\mathrm{Prob}(M(Q_{|x|}|x\rangle) = 1) \geq 2/3$.
3. *If $x \notin L$, then* $\mathrm{Prob}(M(Q_{|x|}|x\rangle) = 0) \geq 2/3$.

"BQP" stands for "bounded quantum polynomial." Different gate sets $\mathcal{G}$ satisfying the conditions all define the same class BQP, as we will discuss later.

What if we use a quantum Turing machine to generate the BQP circuits? This gives the same class, as you will see on the problem set.

6 NP and QMA

6.1 NP

"NP" stands for "non-deterministic polynomial." The original definition is in terms of a non-deterministic Turing machine, which is a Turing machine that has a choice of possible transitions and can choose the "best" one. However, there is also a straightforward and more understandable definition in terms of checking the answer to the computation.

The idea is that for each yes instance x there is a "witness" w_x which proves that $x \in L$ and an efficient algorithm to check that the witness is correct. In particular, we have the following definition:

Definition 17. *Let NP be the set of languages L such that there exists a uniform family of polynomial-size circuits $C_{n,m}$ that takes two inputs (x, w) with the following properties: For any instance x,*

1. *If $x \in L$, then exists w_x with $|w_x| = O(\text{poly}(|x|))$ and $C_{|x|,|w_x|}(x, w_x) = 1$.*
2. *If $x \notin L$, then for any w_x with $|w_x| = O(\text{poly}(|x|))$, $C_{|x|,|w_x|}(x, w_x) = 0$.*

In the first case, w_x is the witness *for $x \in L$.*

That is, C is the checking algorithm, and if $x \in L$, then there exists some witness which will be accepted by the checking algorithm, whereas if $x \notin L$, than any possible witness will be rejected.

Definition 18. *3-SAT is a language defined by Boolean expressions that can be satisfied. The Boolean expressions we consider are the AND ($\wedge$) of* clauses*, each of which involves the OR ($\vee$) of 3 Boolean variables (True/False valued) or their negations (written here with a line over the variable).*

The definition uses 3 variables because we are considering 3-SAT; k-SAT would involve k variables per clause. The same variable can be repeated in a clause. For instance, the following are valid clauses:

$$x_0 \vee x_5 \vee x_7 \tag{8}$$
$$x_1 \vee \overline{x_3} \vee \overline{x_4} \tag{9}$$
$$x_2 \vee x_2 \vee \overline{x_6} \tag{10}$$
$$x_1 \vee \overline{x_1} \vee x_8 \tag{11}$$

So, for instance, the first clause evaluates to True if one or more of x_0, x_5, x_7 is True. The last clause above always evaluates to True, since either x_1 is True or $\overline{x_1}$ is True.

Then the Boolean expression is formed from these clauses by taking the AND: e.g.,

$$(x_0 \vee x_5 \vee x_7) \wedge (x_1 \vee \overline{x_3} \vee \overline{x_4}) \wedge (x_2 \vee x_2 \vee \overline{x_6}) \wedge (x_1 \vee \overline{x_1} \vee x_8). \tag{12}$$

This evaluates to True iff all the individual clauses making it up are True. So, for instance, if all the variables x_0 through x_8 are True, then this particular expression is True, but if x_1 is False and all other variables are True, the overall expression is False since the second clause is False.

A Boolean expression $B \in$ 3-SAT if there exists a truth assignment x_i to the variables of B such that $B(x_0, \ldots, x_n) =$ True.

Then 3-SAT is in NP: If $B \in$ 3-SAT, let the witness w_B be a satisfying truth assignment $(x_0, \ldots, x_n)$. The checking circuit evaluates B on the truth assignment. This checking circuit clearly runs in polynomial time and when we have a valid truth assignment, the checking circuit accepts. On the other hand, if $B \notin$ 3-SAT, then for any witness (any truth assignment), evaluating B on that truth assignment gives False, and the witness is rejected.

Note that there is an inherent asymmetry to the definition of NP: When $x \in L$, we can prove that and efficiently verify that the proof is correct, even though it might be hard to find the proof (i.e., the witness) efficiently. However, if $x \notin L$, we can't easily prove that this is true. Any witness we try will fail, of course,

but we can't be sure that we didn't just make a bad choice of witness and that there isn't another witness that would work.

Note also that P $\subseteq$ NP since we can just let the witness be empty and the checking circuit equal to the algorithm to find the answer given the instance x.

There is another class called co-NP, which is basically the same as NP except that the witness exists when $x \notin L$ and there is no witness when $x \in L$. It is believed that NP $\neq$ co-NP. Factoring and graph isomorphism are examples of problems in NP $\cap$ co-NP.

6.2 Reductions and NP-completeness

It is a famous open problem to show that P = NP. One approach to studying this problem is to look at the hardest problems in NP. What does that mean?

For some pairs of problems, it is difficult to say and may not be meaningful to say that one is harder than the other, but sometimes it is clearly sensible to say that. In particular, if solving problem A automatically gives us a soltuion to problem B as well, then it makes sense to say that A is at least as hard as B. This the underlying notion of *reduction*.

Definition 19. *Language L* reduces *to language M if there exists a polynomial-time computable function $f(x)$ such that for any instance x of L:*

1. *$f(x)$ is an instance of M,*
2. *If $x \in L$, then $f(x) \in M$,*
3. *If $x \notin L$, then $f(x) \notin f(M)$.*

There are a variety of different notions of reduction, but they all share a basic property with this definition: If we have an algorithm to decide M, we can combine it with the reduction to get an algorithm to decide L as well. In particular, given any instance x for L, we convert it to an instance of M using f and then determine whether $f(x) \in M$. Whatever the answer, we know it is the same as the answer to the original question : Is $x \in L$?

This lets us define a sensible partial order on problems with M "harder than" L if L reduces to M and L and M of equal hardness if they reduce to each other.

Remarkably, there are some problems which we can prove to be harder than any other problem in NP, in the sense that any NP problem can be reduced to one of them. Problems with this property are called "NP-complete."

Definition 20. *Let C be a complexity class. A language L is* C-hard *if for any $M \in C$, M reduces to L. A language L is* C-complete *if L is in C and L is C-hard.*

Example 2. *An example of an NP-complete problem is 3-SAT.*

To see how this works, let us consider an arbitrary NP problem M and see how to reduce it to 3-SAT. Since $M \in$ NP, there exists a family C_n of polynomial-size checking circuits for M.

Now, a classical circuit is just a sequence of gates, say AND, OR, and NOT, connected by wires. AND and OR takes two inputs and one output; NOT takes one input and one output. Suppose we assign a new variable y_j to each wire. Then the basic gates express a relationship between the variables corresponding to their inputs and outputs. For instance, a NOT gate with input y and output y' expresses

$$(y \wedge \overline{y'}) \vee (\overline{y} \wedge y') = (y \vee y') \wedge (\overline{y} \vee \overline{y'}). \tag{13}$$

This is a Boolean formula of the form needed for 3-SAT (adding an extra redundant variable to the two clauses). Similarly for AND and OR.

By combining all the Boolean formulas for all the gates in the circuit C_n, we get a Boolean formula for which any satisfying assignment of variables is equivalent to a "history" of the circuit, giving truth values

for the variables on the wires as they evolve under the gates of the circuit. However, this property doesn't constrain the input or output to the circuit. We can constrain some of the inputs to the circuit (corresponding to any ancilla bits or to the inputs for the instance we care about) by adding clauses which are true only when those bits have the desired values.

Now suppose we add one more clause. If x_f is the variable corresponding to the output wire of the circuit, add the clause $x_f \vee x_f \vee x_f$. With the extra clause, the Boolean formula $f(x)$ (when x is the original instance) is satisfied iff we have a set of possible input variables and a valid history of the circuit such that the output of the circuit is True (or 1). That is, the Boolean formula is satisfied exactly in those cases where the corresponding circuit input is accepted by the circuit C_n.

The input in this case is the witness for the instance x, so the Boolean formula is satisfied iff there is a witness that is accepted by the checking circuit. That is, $f(x) \in$ 3-SAT iff $x \in M$. We have reduced M to 3-SAT.

The argument above shows the following theorem:

Theorem 3 (Cook-Levin)**.** *3-SAT is NP-complete.*

There are many more NP-complete problems.

One significance of NP-completeness is that it offers a potential route to prove that P = NP: Simply find a polynomial-time algorithm for any NP-complete problem L. Since any other NP problem can be reduced to L, that automatically gives us an efficient algorithm for every language in NP. NP-completeness also helps us understand the relationships of complexity classes by distinguishing the hardest problems in NP from easier problems that are not NP-complete and therefore sometimes sit in lower complexity classes (such as NP $\cap$ co-NP).

6.3 QMA

To define a quantum analog of NP, we need to both handle the random aspect of quantum measurement and to convert everything to quantum states and circuits. The solution to add randomness is essentially the same as for BPP and BQP: Namely, use a checking circuit with a probability 2/3 of giving the correct answer. When we do this for a classical complexity class, we get a class known as MA, for "Merlin-Arthur." This class comes with a story attached to it: Arthur is a king, intelligent but merely mortal. He can do any polynomial-time computation. However, he has an advisor Merlin, a powerful wizard who can perform any computation. Unfortunately, Merlin can be a bit tricky and Arthur does not trust him, so he insists that whenever Merlin gives him an answer to any question that Merlin be able to prove that the answer is true. For instance, Arthur could ask Merlin "Is this Boolean formula satisfiable?" If Merlin answers "yes," then he can follow that up with a proof by giving a satisfying assignment for the formula and Arthur can then check that using his own computational abilities. This certainly allows Merlin to convince Arthur of the "yes" answer for any language in NP. However, Arthur is also satisfied with a statistical proof, so we allow the checking algorithm to be randomized, giving the potentially slightly larger class MA.

Making MA quantum (giving QMA, "quantum Merlin-Arthur") is actually a bit more challenging because there is a choice we have to make: We have two classical things in the definition of NP: A classical checking circuit and a classical witness. Which of these should we make quantum? Making just the witness quantum doesn't make sense since there is no way for the classical circuit to interact with it in a quantum way, so we should at least make the checking circuit quantum.

But should we allow the witness to be a quantum state or leave it as a classical bit string? These give what are apparently two different complexity classes, QMA and QCMA.

Definition 21. *Fix a particular efficiently computable approximately universal gate set $\mathcal{G}$. Let QCMA be the set of languages L such that there exists a uniform family of polynomial-size quantum circuits $Q_{n,m}$ (with gates drawn from $\mathcal{G}$) that takes two inputs (x, w) with the following properties: For any instance x,*

1. *If $x \in L$, then exists w_x with $|w_x| = O(\text{poly}(|x|))$ and $\text{Prob}(M(Q_{|x|,|w_x|}|x\rangle \otimes |w_x\rangle) = 1) \geq 2/3$.*
2. *If $x \notin L$, then for any w_x with $|w_x| = O(\text{poly}(|x|))$, $\text{Prob}(M(Q_{|x|,|w_x|}|x\rangle \otimes |w_x\rangle) = 0) \geq 2/3$.*

In the first case, w_x is the witness *for $x \in L$.*

Definition 22. *Fix a particular efficiently computable approximately universal gate set $\mathcal{G}$. Let QMA be the set of languages L such that there exists a uniform family of polynomial-size quantum circuits $Q_{n,m}$ (with gates drawn from $\mathcal{G}$) that takes two inputs $(x, |\psi\rangle)$ with the following properties: For any instance x,*

1. *If $x \in L$, then exists an n-qubit state $|\psi\rangle_x$ with $n = O(\text{poly}(|x|))$ and $\text{Prob}(M(Q_{|x|,n}|x\rangle \otimes |\psi\rangle_x) = 1) \geq 2/3$.*
2. *If $x \notin L$, then for any n-qubit state $|\psi\rangle_x$ with $n = O(\text{poly}(|x|))$, $\text{Prob}(M(Q_{|x|,n}|x\rangle \otimes |\psi\rangle_x) = 0) \geq 2/3$.*

In the first case, $|\psi\rangle_x$ is the witness *for $x \in L$.*

We will discuss these two classes again later in the class. QMA is usually considered the more important one, the most natural quantum analog of NP. One thing that is worth noting right now is that as far as we know, all the QMA-complete problems must be promise problems because of the requirement that the probability is at least 2/3 of either accepting or rejecting the witness. Certainly, the most natural complete problems we know don't have this probability separation without a restriction on the possible instances. Without the promise, they are generally *QMA-hard* instead.

7 PSPACE

Going to still larger complexity classes, we get PSPACE. PSPACE is the class of decision problems that can solved in polynomial *space* on a Turing machine. PSPACE is a large complexity class because we can run very long algorithms which use only a modest amount of space and then erase it and use it again.

Definition 23. *Let PSPACE be the set of languages that can be decided by a Turing machine using $O(\text{poly}(|x|))$ space for instance x.*

We could also define it as languages decided by circuit families which use a polynomial number of bits; however, we would have to use a broader definition of "uniform." This is because a Turing machine using only logarithmic space can't generate exponential-size circuits. Similarly, any PSPACE algorithm will halt in at most exponential time.

Theorem 4. *Let A be an algorithm that uses $f(|x|)$ space on input x. Then A on input x either halts or repeats itself after a time at most $2^{O(f(|x|))}$.*

Proof. Consider the state of the Turing machine tape (the *whole* tape that is used, not just one cell) at time i. If there are A different symbols in Σ, then there are total of $A^{f(|x|)}$ different possible states of the tape. There are also $|K|$ head states and $O(f(|x|) + |x|)$ head locations (conceivably there could be some blank spots on the tape between the input and the part that is used, but there have to be a limited amount of them or the algorithm would never start writing again), for a total of $O((f(|x|) + |x|)|K|A^{f(|x|)} = 2^{O(f(|x|))}$ different possible states of tape plus head.

The action of the Turing machine is completely determined by the state of the tape and the head. In particular, if it repeats the same tape state and the same head state, it will repeat the next time step and so on, looping. Since there are only $2^{O(f(|x|))}$ possible different states given the amount of space used by the algorithm, it must either halt or repeat itself after this time. □

PSPACE contains all of the complexity classes we have talked about so far (except for P/poly which is a bit of a special case). Here is a quick proof sketch that BQP $\subseteq$ PSPACE. The proof that QMA $\subseteq$ PSPACE is a bit more involved and we will get back to it later.

Proof that BQP $\subseteq$ PSPACE. A quantum circuit can be simulated on a classical circuit by multiplying out the $2^n \times 2^n$ matrices given by the quantum gates. This takes exponential time and naively takes exponential space. However, we can reduce this to exponential time and polynomial space.

Claim 3. *We can calculate any single matrix element using only polynomial space:*

Proof of claim.

$$\langle a| \prod_{i=0}^{n} U_i |b\rangle = \sum_{c_0,\ldots,c_{n-1}} [U_0]_{a,c_0} \prod_{i=1}^{n-1} [U_i]_{c_{i-1},c_i} [U_n]_{c_{n-1},b}. \tag{14}$$

Each term in this sum is a product of polynomially many terms so can be computed using polynomial time and space. We can step through the value of the n-tuple $(c_0, \ldots, c_{n-1})$ and keep a running total for the sum while doing so, discarding any previous partial sum. At any given time, then, we only need to keep track of the n-tuple (which requires linear space) and the current value of the sum (which requires polynomial space to keep track of with adequate precision). Thus, the whole sum can be computed in polynomial space. □

A single matrix element doesn't tell us the probability of getting outcome 1 when the first qubit is measured, but we can compute

$$\text{Prob(outcome 1)} = \sum_{a=1a_1a_2\ldots a_m} |\langle a| \prod_{i=0}^{n} U_i |b\rangle|^2 \tag{15}$$

(where $|b\rangle$ is the initial state of the computation), the total probability over all outcomes with 1 in the first qubit. Each term in the sum is computable with polynomial space by the claim, and again we can keep a running total of the sum with polynomial space. □

You might expect that at this point, we should be introducing another complexity class of problems that can be solved by quanutm algorithms that use only polynomial space, but it turns out that quantum algorithms with polynomial space give the same complexity class as classical algorithms with polynomial space. We will return to this when we talk about PSPACE at more length.

8 References

You can get the syllabus on the class web page or ELMS. 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).

An introductory textbook on quantum computation will cover the basics of quantum mechanics and quantum circuits and define the class BQP. My recommendation is Nielsen & Chuang, *Quantum Computation and Quantum Information*. I will give more references for QMA when we discuss it in more detail.