CMSC 858L: Quantum Complexity

Instructor: Daniel Gottesman

Spring 2023

2 Lecture 2: Polynomial Time

Again, 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).

2.1 Polynomial Time

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 1 (Strong Church-Turing Thesis)**.** *If a physically reasonable model of computation decides language L with resources $f(|x|)$, then a 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 1.

- *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")*

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

Definition 3. *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 4. *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 and 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.*

We can define a quantum circuit *similarly, 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.*

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 circuit which are derived in a regular and not overly complicated way. The standard way to do this is to consider uniform circuits:

Definition 5. *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 1. *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.

2.2 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 cases where there is a distinction between the difficulty of a functional problem and a corresponding decision problem. I am not sure if we will encounter any, but I will call it out if it happens; otherwise, just assume they are effectively the same.

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.

2.3 BPP

What if we want to consider randomized algorithms? The first change we need to make is 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. (You will explore the relationships between different definitions of randomized algorithms in the problem set.)

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 6. *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 last time, many people believe that actually $\text{P} = \text{BPP}$ using derandomization ideas.