CS 579: Computational Complexity Fall 2025

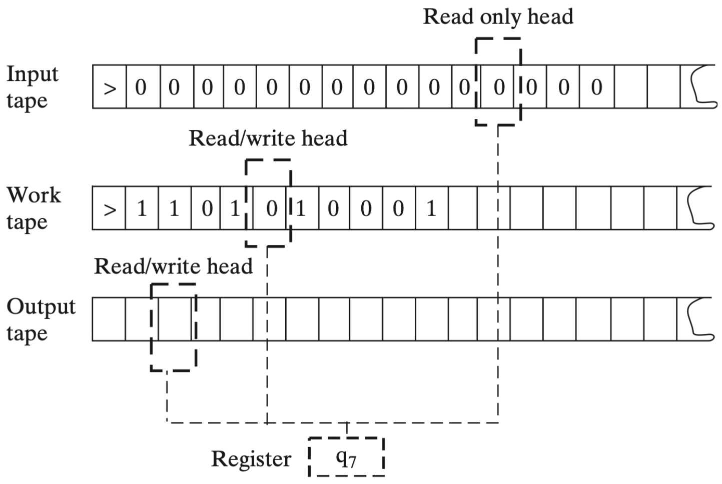
Lecture 01 - August 26

Prof. Fernando Granha Jeronimo *Super Scribes: William Gay & Sasha Levinshteyn*

Complexity Theory is concerned with classifying problems according to the amount of resources required to solve these problems. Some of these resources include: Time, Space, Randomness, Circuit Size/Depth, Interaction, Queries, Communication, Quantum Bits, and many more.

1 Computability Theory

In the early 20th century, a major open question in mathematics was the *Entscheidungsproblem.* This question asked whether there exists an effective procedure (an algorithm) which determines whether a given mathematical formula is true or false. However, answering this question would require mathematicians to formally define "algorithm". In 1936, Alan Turing published a paper in which he invented the Turing machine, an object which carries out operations "so elementary that it is not easy to imagine them further subdivided".



The Turing machine gave rise to the class of "Turing computable functions" (or just "computable functions"). Around this time, several other mathematicians proposed their own formal descriptions such as lambda calculus, recursive functions, and register machines. It would turn out that all of these formalizations gave rise to the same class of functions as the Turing machine! This lent credibility to the idea that the Turing machine is the "correct" formalization, and gave rise to the following idea:

Thesis 1 (Church-Turing Thesis)**.** *A function is effectively calculable if and only if it can be computed by a Turing machine.*

Note that the Church-Turing Thesis is not a formal mathematical theorem, but rather, a philosophical statement that the Turing machine captures our natural intuition about what an algorithm is. Nevertheless, it is a celebrated result which helped give rise to the field of theoretical computer science.

1.1 Undecidable Languages

With the invention of the Turing machine, it was proven that the Entscheidungsproblem had a negative answer. However, I view this as a good thing because I'd be out a job otherwise. Many other natural problems have been shown to be undecidable, such as the Halting and Diophantine Equations. An important result yielding a large class of undecidable problems is the following:

Theorem 2 (Rice's Theorem)**.** *Let $\mathcal{P}$ be a set of (partial)-computable functions, and let $L \subseteq \{0,1\}^*$ be the set of strings which encode a Turing machine that computes a function in $\mathcal{P}$. If $L \neq \varnothing, \{0,1\}^*$, then L is undecidable.*

This statement says that one cannot decide any non-trivial properties about the function computed by a Turing machine. This is the reason why one cannot develop an autograder which determines with certainty that some code solves a given problem.

Computability theory is a rich subject on which much of classical complexity theory is built.

2 Time Complexity

The most natural resource to consider when thinking about computation is time, i.e., number of "steps" needed to solve a problem.

Definition 3. Given $t : \mathbb{N} \to \mathbb{N}$, the set $\mathbf{DTIME}(t(n))$ consists of all languages which can be decided in time $O(t(n))$ on a multi-tape Turing machine.

The above definition is sensitive to the formalization chosen. For instance it is easily seen that the language of all strings which are palindromes is in $\mathbf{DTIME}(n)$, however, solving this problem requires $\Omega(n^2)$ time on a single-tape Turing machine. We define several classes which are robust with respect to the underlying model chosen:

Definition 4.

(i) $\mathbf{P} = \mathbf{DTIME}(\text{poly}(n)) = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(n^k)$. ("polynomial time")

(ii) $\mathbf{EXP} = \mathbf{DTIME}(2^{\text{poly}(n)}) = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}\big(2^{n^k}\big)$ ("exponential time")

We use $\mathbf{P}$ as our proxy for "efficiently decidable" languages. There are several other common time classes which are less robust than those above:

Definition 5.

(i) $\mathbf{QP} = \mathbf{DTIME}(n^{\text{polylog(n)}}) = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}\big(n^{\log^k(n)}\big)$. ("quasi-polynomial time")

(ii) $\mathbf{E} = \mathbf{DTIME}(2^{O(n)}) = \bigcup_{k\in\mathbb{N}} \mathbf{DTIME}(2^{kn})$ ("linear-exponential time")

Exponential time is often associated with brute-force algorithms. One can view a sub-exponential time algorithm as exploiting some structure in the problem at hand to solve it more efficiently than brute-force. There are two non-equivalent definitions of "sub-exponential time" that appear:

Definition 6.

(i) $\mathbf{SUBEXP}_1 = \bigcap_{\varepsilon\in\mathbb{R}^+} \mathbf{DTIME}\big(2^{n^\varepsilon}\big)$.

(ii) $\mathbf{SUBEXP}_2 = \mathbf{DTIME}(2^{o(n)})$.

An important aspect of the invention of the Turing machine was showing the existence of a *universal machine*. A universal machine is a Turing machine which can simulate all other Turing Machines. One can view this as being analogous to an interpreter of one's favorite programming language. There exists a universal machine which is very efficient:

Theorem 7 (Efficient Simulation). *There exists a universal machine which, given a Turing machine that runs in time $O(t(n))$, can simulate that Turing machine in $O(t(n)\log(t(n)))$, where the constants depend on the description of the machine being simulated.*

A function t is time-constructible if $t(n) \geq n$ and there exists a Turing machine which given x computes the binary representation of $f(|x|)$ in time $O(t(|x|)$. This allows one to equip a machine with a $t(n)$-step "timer". All functions we care about are time-constructible. Of course, a machine which has more time to compute can solve more problems

Theorem 8 (Time Hierarchy Theorem). *Let $f, g : \mathbb{N} \to \mathbb{N}$ be time constructible functions such that $f(n)\log(f(n)) = o(g(n))$. Then*

$$\mathbf{DTIME}(f(n)) \subsetneq \mathbf{DTIME}(g(n)).$$

The $\log(f(n))$ factor in the above theorem arises from the fact that the proof utilizes simulation on universal Turing machine.

Corollary 9. $\mathbf{P} \subsetneq \mathbf{EXP}$.

2.1 Non-Deterministic Time

In 1956, Gödel sent a letter to von Neumann inquiring about efficient generating proofs vs efficiently verifying the correctness of proofs. However, von Neumann did not respond to this letter (because he was dying of cancer), and Gödel pursued the subject no further. In 1971, Cook formulated more-or-less the same question as Gödel, and it became known as the **P** vs **NP** problem.

Definition 10. The class **NP** consists of all languages L such that there exists a polynomial time Turing machine $\mathcal{M}$ and a polynomial q such that for all strings $x \in \{0,1\}^*$,

$$x \in L \quad \iff \quad \exists\, w \in \{0,1\}^{q(|x|)},\ \mathcal{M}(w,x) = 1$$

One can think of w in the above definition as a "proof" or "witness" that $x \in L$. One can give an equivalent definition of **NP** in terms of non-deterministic machines. A non-deterministic machine is a Turing machine with multiple transition functions, so it can perform "branching computations". We say a non-deterministic machine accepts an input if any branch accepts.

Proposition 11. *A languages L is in* **NP** *if and only if there exists a non-deterministic machine that decides L such that each branch of computation takes polynomially many steps.*

Using the non-deterministic machines definition, one can define other complexity classes. In particular:

Definition 12. Given $t : \mathbb{N} \to \mathbb{N}$, the set $\mathbf{NTIME}(t(n))$ consists of all languages which can be decided by a non-deterministic machine which takes $O(t(n))$ steps in each computation branch.

With this, Proposition 11 can be restated as $\mathbf{NP} = \mathbf{NTIME}(\text{poly}(n))$.

As with deterministic time, more non-deterministic time allows for more languages.

Theorem 13 (Non-Deterministic Time Hierarchy Theorem)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be time-constructible function such that $f(n+1) = o(g(n))$. Then*

$$\mathbf{NTIME}(f(n)) \subsetneq \mathbf{NTIME}(g(n)).$$

One might ask about the relationship between deterministic time and non-deterministic time. It is rather trivial to show:

Proposition 14. $\mathbf{NTIME}(t(n)) \subseteq \mathbf{DTIME}(2^{O(t(n))})$.

However, anything stronger than this is much harder.

Open Problem. Does $\mathbf{P} = \mathbf{NP}$?

This problem is asking whether SAT can be solved by a poly-time algorithm.

3 Space Complexity

Space is another natural resource to consider. The reason we use a read-only input tape and separate work tapes in our formalization, is so we can measure space as being the cells written to *on the work tapes.*

Definition 15.

(i) Given $s : \mathbb{N} \to \mathbb{N}$, the set $\mathbf{DSPACE}(s(n))$ consists of all languages which can be decided in $O(s(n))$ space on a Turing machine.

(ii) $\mathbf{PSPACE} = \mathbf{DSPACE}(\text{poly}(n))$.

(iii) $\mathbf{L} = \mathbf{DSPACE}(\log(n))$.

A hierarchy theorem exists for space-bounded computation as well:

Theorem 16 (Space Hierarchy Theorem)**.** *Let $f, g : \mathbb{N} \to \mathbb{N}$ be time constructible functions such that $f(n) = o(g(n))$. Then,*

$$\mathbf{DSPACE}(f(n)) \subsetneq \mathbf{DSPACE}(g(n)).$$

Of course, one can define the non-deterministic versions of space classes as well.

Definition 17.

(i) Given $s : \mathbb{N} \to \mathbb{N}$, the set $\mathbf{NSPACE}(s(n))$ consists of all languages which can be decided in $O(s(n))$ space on a non-deterministic Turing machine.

(ii) $\mathbf{NPSPACE} = \mathbf{NSPACE}(\text{poly}(n))$.

(iii) $\mathbf{NL} = \mathbf{NSPACE}(\log(n))$.

The space analogue of $\mathbf{P}$ vs $\mathbf{NP}$ has been solved:

Theorem 18 (Savitch's Theorem)**.** *For all space constructible $s : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq \log(n)$, we have $\mathbf{NSPACE}(s(n)) \subseteq \mathbf{DSPACE}(s(n)^2)$.*

Corollary 19. $\mathbf{PSPACE} = \mathbf{NPSPACE}$.

We know even more about non-deterministic space:

Theorem 20 (Immerman–Szelepcsényi Theorem)**.** *For all space constructible $s : \mathbb{N} \to \mathbb{N}$ with $s(n) \geq \log(n)$, we have $\mathbf{NSPACE}(s(n)) = \mathbf{coNSPACE}(s(n))$. In particular, $\mathbf{NL} = \mathbf{coNL}$.*

The reason we study $\mathbf{L}$ is because it is a more restricted class than $\mathbf{P}$, which seems to make it easier to deal with. However, we cannot restrict the model much further, else we collapse into the class of regular languages:

Theorem 21. *If $s : \mathbb{N} \to \mathbb{N}$ is $o(\log\log(n))$, then $\mathbf{DSPACE}(s(n)) = \mathbf{DSPACE}(1)$.*

So far, our view is

$$\mathbf{L} \subseteq \mathbf{NL} \subseteq \mathbf{P} \subseteq \mathbf{NP} \subseteq \mathbf{PSPACE} \subseteq \mathbf{EXP}.$$

It is conjectured that all of the above inequalities are strict, but all are still open.

4 Randomized Computation

One can consider what happens if we allow randomness in computation. One can define a probabilistic Turing machine to be one which has access to a random bit generator.

Definition 22. The class **BPP** consists of all languages decidable by a poly-time probabilistic Turing machine which on any given input, gives the correct output with probability at least 2/3.

One might think that this is too lenient and may prefer one-sided error:

Definition 23. The class **RP** consists of all languages decidable by a poly-time probabilistic Turing machine on which the "yes" instances are accepted with probability at least 1/2 and the "no" instances are rejected with probability 1.

One might not even tolerate any error at all:

Definition 24. The class **ZPP** consists of all languages decidable by a probabilistic Turing machine which runs in expected polynomial time and correctly answers all inputs with probability 1.

One may have learned that you can pass between "Las-Vegas" and "Monty-Carlo" algorithms for a problem. This is formalized by the following:

Theorem 25. *$\mathbf{ZPP} = \mathbf{RP} \cap \mathbf{coRP}$.*

One may also consider a randomized version of **NP**.

Definition 26. The class **MA** consists of all languages L that can be decided by a poly-time probabilistic Turing machine where each yes instance has a witness that will cause it to accept with probability 2/3 and no witness will cause a no instance to accept with probability greater than 1/3.

5 Circuit Complexity

The boolean circuit model is another natural model to consider. Since a single circuit only computes on inputs of a fixed size, one needs an infinite family $\{C_n\}$ of circuit to compute a function on $\{0,1\}^*$, where C_n computes on inputs of length n. Note that *every* function is computable by some circuit family. Most functions on $\{0,1\}^n$ require a circuit of size $\tilde{\Omega}(2^n)$.

Definition 27. The class $\mathbf{P_{/poly}}$ consists of all functions computable by a circuit family whose size is polynomial bounded.

Note that $\mathbf{P}_{/\mathbf{poly}}$ contains undecidable languages, as every unary language is decidable by a polynomial bounded circuit family.

One can also consider the depth of a circuit as a resource:

Definition 28. For each $k \in \mathbb{N}$, the class $\mathbf{AC}^k$ consists of all languages decidable by a circuit family with polynomial size and depth $O(\log^k(n))$ using unbounded fan-in gates.

The class $\mathbf{AC}^0$ is interesting to study as it is very restrictive. Addition is in $\mathbf{AC}^0$, but multiplication is not. There are ever some regular languages (such as the parity function) which are not in $\mathbf{AC}^0$.

One may also consider other restricted circuit models, such as monotone circuits which only have AND and OR gates. It can be shown that the **NP**-hard clique problem requires an exponentially sized family of monotone circuits to solve. However, it can also be shown that certain problems in **P** also require exponentially sized monotone circuits.

One can also consider arithmetic circuits, in which the gates carry out arithmetic operations on some field or algebra. This gave rise to the study of algebraic complexity theory. Some central questions of algebraic complexity theory are whether the permanent can be computed efficiently and determining the matrix multiplication exponent.

6 Modern Complexity

Ryan Williams at MIT has made some notable advancements in classical complexity:

Theorem 29 (Williams, 2011). $\mathbf{NEXP} \not\subseteq \mathbf{ACC}^0$

Theorem 30 (Williams, 2025). $\mathbf{DTIME}(t(n)) \subseteq \mathbf{DSPACE}(\sqrt{t(n)\log(t(n))})$

6.1 Probabilistically Checkable Proofs

Definition 31. Let $r, q : \mathbb{N} \to \mathbb{N}$. The class $\mathbf{PCP}[r, q]$ consists of all languages L for which there exists a poly-time probabilistic oracle machine which on an input of length n uses $O(r(n))$ bits of randomness and makes $O(q(n))$ queries to the oracle, such that if $x \in L$ then there exists an oracle string π which will cause the machine to accept with probability 1 and if $x \notin L$ then the machine will reject with probability at least one half with any oracle string.

Theorem 32 (PCP Theorem). $\mathbf{NP} = \mathbf{PCP}[\log(n), 1]$.

The PCP theorem can be written in an equivalent form stating the **NP**-hardness of a problem relating to CSPs. The *Unique Games Conjecture* is another open problem which asks about the **NP**-hardness of a problem related to determining the value of a CSP. This conjecture is closely tied to many hardness-of-approximation results.

6.2 Meta Complexity

In meta complexity we ask about the complexity of problems in complexity. A central problem in meta complexity is determining whether MCSP is **NP**-complete.

Definition 33. MCSP is the language consisting of all pairs $(f \in \{0,1\}^{\{0,1\}^n}, k \in \mathbb{N})$ such that f can be computed by a circuit of size k.

6.3 Fine Grained Complexity

In fine grained complexity, we are interested in determining exact lower and upper bounds on problems, rather than just polynomial or exponential. We often operate under the exponential time hypothesis, which states that SAT requires exponential time to solve. Note that this is a stronger assumption than $\mathbf{P} \neq \mathbf{NP}$.

6.4 Quantum Computing

One can define quantum analogues of **P** and **NP**, namely, **BQP** and **QMA**. One is interested in determining whether these are actually more powerful than their classical analogues. It is open whether the following inclusions are strict:

$$\mathbf{P} \subseteq \mathbf{BPP} \subseteq \mathbf{MA} \subseteq \mathbf{QMA}.$$

We are also interested in whether or not a quantum version of the PCP Theorem holds.

6.5 Modern Complexity Tools

Some tools used in modern complexity are: error-correcting codes, spectral graph theory, Fourier analysis of Boolean functions, sum-of-squares optimization, communication complexity, and property testing. Some of these topics will be covered throughout the remainder of the course.