Complexity Theory

Instructor: Alexander Razborov, University of Chicago.
razborov@cs.uchicago.edu
Scribe: Yakov Shalunov, University of Chicago.
Course Homepage:
`https://people.cs.uchicago.edu/~razborov/teaching/spring25.html`

Spring 2025

Contents

Part 1

Uniform Models

Lecture 1

Date: March 25th, 2025

1.1 History

1.1.1 Introduction

Historical lecture, many people date complexity theory back to one particular paper [1], but we'll cover some older things first.

Discrete computations can be represented as functions $f : \{0,1\}^* \to \{0,1\}^*$ taking finite binary strings to finite binary strings. You can encode arbitrary structures using binary strings.

Remark (Alphabets). We are choosing the alphabet $\Sigma = \{0,1\}$, rather than working in an arbitrary alphabet. In computability theory, it is typical to use $\Sigma = \{1\}$ and, e.g., represent $n \in \mathbb{N}$ in the unary encoding 1^n.

In computability theory, this makes no difference, but in *complexity* theory, this makes an enormous difference, since for $n \in \mathbb{N}$, the unary representation has length x and the binary representation has length approximately $\log_2 x$.

The exponential difference is enormous, covering the span of most complexity theory.

Any other fixed finite alphabet Σ, e.g., $\{0,1,\ldots,9\}$, this will not change much because then the length of x is about $\log_{10} x$, which is a constant factor difference from $\log_2 x$, which is immaterial to complexity theory.

Remark (Constant factors). We use "big-O" notation, where $f = O(g)$ if $\exists C \in \mathbb{R}_+, N \in \mathbb{N}$ such that $\forall n \geq N$, $f(n) \leq C \cdot g(n)$. We generally do not care about these constant factor differences and will generally make these constants implicit via big-O rather than explicit.

Similarly, $f = \Omega(g)$ if $\exists C \in \mathbb{R}_+, N \in \mathbb{N}$ such that $\forall n \geq N$, $f(n) \geq C \cdot g(n)$.

Finally, $f = \Theta(g)$ if $f = O(g)$ and $f = \Omega(g)$.

Remark (Machine independence). Due to ignoring constant factors, most results are machine/language and technology independent.

We are interested in lower-bounds and upper-bounds on how hard problems are. Upper-bounds are generally proven by providing algorithms which solve the problem and then proving that those algorithms have some complexity, bounding the complexity of the problem.

On the other hand, lower-bounds require a variety of more conceptually distinct techniques and are generally harder to come by.

Mostly for historical reasons, we use the model of "Turing machines," though it is functionally equivalent for our purposes to, e.g., Python.

Definition 1 (Turing machine)**.** A Turing machine consists of two finite alphabets, Σ (the "symbol alphabet") and Q the "internal alphabet" or "set of states." We presume that there is are a distinguished "blank" symbol $s_0 \in \Sigma$, a "start state" $q_1 \in Q$, and a "halt state" $q_h \in Q$.

While Turing machines can be defined many ways, we will assume that there is a single, infinite tape on which the machine can write symbols. All but finitely many symbols of the tape will always be the "blank" symbol s_0.

The machine has a "head" located at some position on the tape and is in some internal state $q \in Q$. The Turing machine then has a program or "transition function," $T : \Sigma \times Q \to Q \times (\Sigma \cup \{L, R\})$.

The transition function governs how the machine evolves from one time step to the next. The machine reads the symbol $s \in \Sigma$ under the head, and then combined with the current internal state $q \in Q$, it outputs a new state and either a new symbol $s' \in \Sigma$ to write to the tape or one of L and R, representing left and right movements of the head along the tape.

One standard alphabet is $\Sigma = \{0, 1, s_0\}$ (where s_0 is often written as an underscore). To run the machine on an input $x \in \{0,1\}^*$, the tape is initialized such that it contains x surrounded by infinite cells containing s_0 and the head is placed on the first character of x.

The machine then evolves according to T until it reaches q_h, at which point the output is the contents of the tape with all s_0 removed. Note that a Turing machine is permitted to never reach q_h, in which case the output is undefined.

Correspondingly, every Turing machine M computes a partial function $M : \{0,1\}^* \to \{0,1\}^*$, where $M(x)$ is undefined if M on input x never reaches q_h and is otherwise the output.

Definition 2 (Halting problem)**.** The problem HALT : $\{0,1\}^* \to \{0,1\}^*$ is defined to be 1 on a description of a Turing machine M and input x to that machine if M halts on x, and 0 otherwise.

Theorem 1 (Halting problem)**.** *There is no Turing machine computing* HALT.

Proof. Suppose H computes HALT. Let

$$H'(M) = \begin{cases} \text{output } 0 & \text{if } H(M, M) = 0 \\ \text{loop forever} & \text{if } H(M, M) = 1 \end{cases}$$

Observe then that if $H'(H')$ halts, $H(H', H') = 1$, so $H'(H')$ loops forever. Conversely, if $H'(H')$ loops forever, $H(H', H') = 0$, so $H'(H') = 0$ halts. Thus, we have a contradiction and so can conclude H cannot exist. □

However, for our purposes, we are interested in Turing machines running in bounded time. We can consider our machines to keep a running clock and once the clock elapses, they halt regardless of other state of the computation.

We are in particular interested in decision problems, which are equivalent with only a "moderate" slowdown to arbitrary computational problems.

Definition 3 (Decision problems and languages)**.** A computational problem $f : \{0,1\}^* \to \{0,1\}^*$ is a decision problem if the image of f is $0, 1$.

The corresponding language $L \subseteq \{0,1\}^*$ is $f^{-1}(1)$.

We will often use decision problems and their corresponding languages fairly interchangeably.

Definition 4 (Time and space complexity of a Turing machine)**.** Given a Turing machine M, we can define the time complexity $t_M(x)$ to be the number of steps the machine runs for before halting, and we define the space complexity $s_M(x)$ to be the number of *distinct* cells that the machine reads during its execution.

If $M(x)$ is undefined, so are $t_M(x)$ and $s_M(x)$.

These metrics, and the trade offs between them, are of primary interest to complexity theory.

We have defined time and space complexity of a specific Turing machine. But we would also like to define them for a *computational problem* $f : \{0,1\}^* \to \{0,1\}^*$.

Definition 5 ((Naive) complexity of a function)**.** We define the complexity of a computable $f : \{0,1\}^* \to \{0,1\}^*$ to be the complexity of the "best" machine computing f, where "best" depends on specific property we're interested.

Unfortunately, as we will see in Theorem 3 below, "best" is impossible to define.

1.1.2 Digression: Kolmogorov complexity (A success story)

Rather than computing anything on x, we will divert to the task of reproducing x with minimal information.

Definition 6 (Kolmogorov complexity with respect to a machine)**.** If we fix a Turing machine M, the Kolmogorov complexity with respect to M is $K_M(x) = \min\{|y| : M(y) = x\}$.

We can think of M as a decompression algorithm, and K_M measures the compressibility of x.

It turns out that in this case, the "best" machine does exist.

Theorem 2 (Kolmogorov)**.** *There exists a universal machine U such that for any other M, $K_U(x) \leq K_M(x) + c_M$.*

Proof. The universal machine U takes an encoded pair (M, x) and simulates $M(x)$. Then if $K_M(y) = x$, $K_U(y') = x$ for $y' = (M, y)$, where $|y'| \leq y + c_M$. (We can choose an encoding of pairs which makes $|(M, y)| = c_M + y$.) □

We can fix a standard enumeration of all Turing machines $M_1, M_2, \ldots, M_e, \ldots$, and then let $U(e, x) = M_e(x)$. (It is convenient to identify the Turing machine M_e with its description/identifier e.)

We can then define Kolmogorov complexity in terms of K_U.

1.1.3 Machine-independent complexity

Introduced by Blum in [2].

Note that the enumeration $M_1, \ldots, M_e, \ldots$ gives a fixed enumeration of all partial computable functions $\varphi_1, \ldots, \varphi_e, \ldots$. (Note that this list has a large number of duplicates, since there are many Turing machines computing a given function.)

Definition 7 (Abstract complexity measure)**.** An abstract complexity measure is a sequence of functions $\Phi_1, \ldots, \Phi_e, \ldots$ satisfying Blum's axioms:

1. $\mathrm{dom}(\Phi_e) = \mathrm{dom}(\varphi_e)$.

2. The predicate (in 3 arguments) $\Phi_e(x) = y$ is decidable.

Example 1 (Time complexity). The sequence $t_1, \ldots, t_e, \ldots$ where $t_e = t_{M_e}$ represents an abstract complexity measure. The first axiom is trivial, since the domain of time complexity is the same as the domain of underlying machine.

The second axiom is satisfied since we can simply simulate machine M_e on input x for exactly y steps to see if $t_e(x) = y$.

Similarly, $s_1, \ldots, s_e, \ldots$ is an abstract complexity measure, with the first axiom again trivial and the second following from the fact that if we bound the machine to using s cells, it only has $|\Sigma|^s \cdot |Q| \cdot s$ distinct full configurations (the state of the tape, the internal state, and the location of the head), and if the configuration ever repeats, computation must proceed exactly the same way from it as it did the first time, so it must then loop forever. By pigeon hole principle it suffices to simulate the machine for $|\Sigma|^s \cdot |Q| \cdot s$ steps.

Theorem 3 (Blum's speedup theorem)**.** *Assume that Φ is any abstract complexity measure and $s(n)$ is an arbitrary total computable function such that $\lim_{n\to\infty} s(n) = \infty$.*

Then there exists a total computable f such that the following holds:

For any e such that $\varphi_e = f$, there exists another e' such that $\varphi_(e') = f$ and for almost all x, $\Phi'_e(x) \le s(\Phi_e(x))$.

Given the previous theorem, on the "problem-complexity plane" there does not really exist a function that assigns to every problem its complexity in any reasonable sense. In Complexity Theory we usually change the axe in a way somewhat akin to the theory of Lebesgue integration: start with a complexity bound and look at the set of all problems solvable by *at least* one algorithm obeying that bound. This leads to the concept of *complexity classes*, defined in terms of resource "budgets."

1.2 Deterministic Time

1.2.1 Deterministic Time Classes

We will talk for now about time complexity. The worst-case time complexity of a machine M is $t_M(n) = \max_{|x|\le n} t_M(x)$.

Sometimes we will be interested in complexity in terms of some structural parameters of the input rather than just length, e.g., for graph algorithms, we may be interested in complexity in terms of the number of vertices or edges, rather than total description size (which depends on the representation).

Definition 8 (Deterministic time)**.** For an arbitrary monotone function $t(n)$, we define the class $\underline{\text{deterministic time } t(n)}$ to be

$$\mathsf{DTIME}\,[t(n)] = \{f | \exists M \text{ computing } f \text{ s.t. } t_M(n) \le O(t(n))\}$$

It is worth noting that for low levels of the hierarchy, e.g., $\mathsf{DTIME}\,[n]$, we do actually care about the details of the model, such as number of heads and tapes.

We then have $\mathsf{DTIME}[n] \subsetneq \mathsf{DTIME}\left[n \log^{O(1)} n\right]$ where the $O(1)$ in the exponent is taken to mean $\mathsf{DTIME}\left[n \log^{O(1)} n\right] = \bigcup_{k\in\mathbb{N}} \mathsf{DTIME}\left[n \log^{k} n\right]$. We have

$$\begin{aligned}
&\mathsf{DTIME}[n] && \text{(linear time)}\\
\subsetneq&\mathsf{DTIME}\left[n \log^{O(1)} n\right] && \text{(quasilinear time)}\\
\subsetneq&\mathsf{DTIME}\left[n^2\right] && \text{(quadratic time)}\\
\subsetneq&\mathsf{DTIME}\left[n^{O(1)}\right] = \mathsf{P} && \text{(polynomial time)}\\
\subsetneq&\mathsf{DTIME}\left[2^{O(n)}\right] = \mathsf{E} && \text{(simply exponential time)}\\
\subsetneq&\mathsf{DTIME}\left[2^{n^{O(1)}}\right] = \mathsf{EXP} && \text{(exponential time)}\\
&\cdots
\end{aligned}$$

where the fact that the inclusions are proper follows from Theorem 5 (next lecture).

Lecture 2

Date: March 27th, 2025

We can prove an even stronger (in a sense) form of Blum's speedup Theorem 3, which seems even more damning, since rather than merely producing a single function which can be sped up, it tells us that there exist entire deterministic time classes which can be sped up:

Theorem 4 (Borodin–Trakhtenbrot Gap Theorem [3, 4])**.** *Let $s : \mathbb{N} \to \mathbb{N}$ be an arbitrary computable function satisfying $s(n) \le n$ and $\lim_{n\to\infty} s(n) = \infty$. Then there exists a computable time bound $T(n)$ such that*

$$\mathsf{DTIME}[s(T(n))] = \mathsf{DTIME}[T(n)].$$

This seems like a final nail in the coffin of complexity theory, but it turns out that $T(n)$ needs to be pathological in order for this to happen, and this isn't an issue with "reasonable" functions $T(n)$.

1.2.2 Hierarchies

Definition 9 (Time constructible)**.** A function $T(n)$ is time constructible if there exists an algorithm computing the function $1^n \mapsto T(n)$ that runs in time $O(T(n))$.

Definition 10 (Little-O notation)**.** We have $f = o(g)$ if $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

Theorem 5 (Time Hierarchy Theorem [1])**.** *Suppose we have two time bounds $g(n)$ and $t(n)$ such that*

1. *$t(n) \ge n$ is time constructible.*
2. *$g(n) = o\left(\frac{t(n)}{\log t(n)}\right)$.*

Then

$$\mathsf{DTIME}[g(n)] \subsetneq \mathsf{DTIME}[t(n)]$$

Proof. The proof is by diagonalization, constructing a machine which differs from every $\mathsf{DTIME}\left[g(n)\right]$ machine on at least one input but runs in time $O(t(n))$.

We need two ingredients. Throughout, define $n = n(x) = |x|$.

First, fix a function $p : \{0,1\}^* \to \{0,1\}^*$ such that:

(a) $p \in \mathsf{DTIME}\left[n\right]$,

(b) p is infinite-to-one, i.e., $\forall x$, $\left|p^{-1}(x)\right| = \infty$, and

(c) $|p(x)| \le \log n$.

(The third condition is not strictly necessary but it will make the proof simpler.)

Second, we need our universal function U satisfying $U(e,x) = \varphi_e(x)$.

We are interested in $U(p(x),x)$ but need to address that U is nontotal. To do so, we simulate $T(n) = \left\lfloor \frac{t(n)}{\log t(n)} \right\rfloor$ tacts of the machine $M_{p(x)}$, and we define f to be

$$f(x) = \begin{cases} 1 & \text{if computation halts in } T(n) \text{ steps and outputs } 0 \\ 0 & \text{otherwise} \end{cases}$$

We claim $f \in \mathsf{DTIME}\left[t(n)\right] \setminus \mathsf{DTIME}\left[g(n)\right]$.

$f \notin \mathsf{DTIME}\left[g(n)\right]$ Assume towards contradiction that it is. Then f is computed by some machine e. By condition 2. of the theorem, we have that there is some $N \in \mathbb{N}$ such that if $n \ge N$, $U(e,x)$ halts in $g(n) < T(n)$ steps. On the other hand, since p is infinity-to-one, we can pick an arbitrarily large x such that $p(x) = e$. Choose some x such that $|x| \ge N$ and $p(x) = e$. Then $f(x) \ne U(e,x)$, which is a contradiction.

$f \in \mathsf{DTIME}\left[t(n)\right]$ We need to bound the simulation overhead of simulating $U(e,x)$, where $|e| \le \log|x|$, for $T(n)$ steps of machine e. It is easy to do quadratic overhead by simply passing back and forth between the machine description e and head location.

In order to get down to logarithmic, we will do a couple of tricks. By using a larger alphabet, we can simulate any constant number k tapes, as long as the head position is the same on all of them (specifically, to simulate tapes with alphabets $\Sigma_1, \dots, \Sigma_k$, we use 1 tape with alphabet $\Sigma_1 \times \cdots \times \Sigma_k$).

We will need 3 tapes. The first tape is the "computational tape," simulating the single work tape of M_e as it is simulated. The second is the "instruction tape," containing the description e, and the final tape is the "clock tape," containing the number of simulation steps taken so far, written in binary.

We note that by the assumption $|e| \le \log|x|$, the content of the instruction tape is of length $O(\log n) \le O(\log T(n))$, and the length of the step counter is also $O(\log T(n))$.

This has not yet solved our problem, since with one head, we'd still need to go back and forth between the start where the instruction and counter are and the head location.

The second idea, and key trick, is the "turtle principle:" we let the main simulation head "carry all its baggage." Whenever we want to move the simulation head, we first move the contents of the clock and instruction tape with it, thus incurring only a $O(\log T(n))$ overhead on running the machine.

Thus, the running time of f is $O(T(n)\log T(n)) = O(t(n))$ as desired. □

Remark. As mentioned above, the condition $|e| \leq \log|x|$ can be dropped but this makes the proof significantly more complicated, see [5, Theorem 1.9] for details.

For the sake of completeness, let us give a similar statement for the space complexity.

Definition 11 (Space-constructible function)**.** A function $S(n)$ is space constructible if there exists an algorithm computing the function $1^n \mapsto S(n)$ that runs in space $O(S(n))$.

Theorem 6 (Space Hierarchy Theorem)**.** *Suppose we have two space bounds $g(n)$ and $s(n)$ such that*

1. $s(n) \geq \log n$ *is space constructible.*
2. $g(n) = o(s(n))$.

Then

$$\mathsf{DSPACE}\left[g(n)\right] \subsetneq \mathsf{DSPACE}\left[s(n)\right].$$

Proof. This proof is similar, except that we don't need to worry about simulation overhead. □

Remark. In the space regime, it turns out that it makes sense to consider sublinear space, in particular the class $\mathsf{L} = \mathsf{DSPACE}\left[\log n\right]$.

This is defined by having separate "read-only" memory for the input, "write-only" memory for the output, and a small work tape; we will talk more about it in a couple of weeks.

Space and time interleave:

$$\mathsf{L} \subseteq \mathsf{P} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP} \subseteq \mathsf{EXPSPACE}$$

The reason for the space-in-time containments is that, as mentioned when justifying that space is an abstract complexity measure, is that a machine using only s bits of space only has roughly $|\Sigma|^s \cdot |Q| \cdot s$ possible distinct configurations, and if a configuration repeats, it must loop forever.

The time-in-space containments follow from the fact that a machine running in time t touches t total cells and thus at most t distinct cells.

Remark (Open Problems)*.* We know by space and time hierarchy theorems that $\mathsf{L} \subsetneq \mathsf{PSPACE}$ and $\mathsf{P} \subsetneq \mathsf{EXP}$. Thus, we know that either $\mathsf{L} \subsetneq \mathsf{P}$ or $\mathsf{P} \subsetneq \mathsf{PSPACE}$ and similarly either $\mathsf{P} \subsetneq \mathsf{PSPACE}$ or $\mathsf{PSPACE} \subsetneq \mathsf{EXP}$.

However, in neither case do we know *which*, though we suspect both.

We have some examples of lower bounds, where we know the problems are decidable but that they do not belong to the lower levels of the hierarchy.

Definition 12 (Presburger Arithmetic)**.** The Presburger arithmetic is the formal theory with the symbols $\langle 0, 1, + \rangle$ (like arithmetic which is undecidable, but with multiplication removed) interpreted over the integers.

We define the language PRESBURGER ARITHMETIC to be the set of all encodings of true first order statements made in this language.

Theorem 7. PRESBURGER ARITHMETIC $\in$ $\mathsf{TRIPLE\ EXP}$ *and there exists $\varepsilon > 0$ such that any program solving* PRESBURGER ARITHMETIC *runs in time $\Omega\left(2^{2^{\varepsilon n}}\right)$, so* PRESBURGER ARITHMETIC $\notin$ EXP.

Definition 13 (Tarski's Algebra)**.** The Tarski's algebra is the formal theory in the language $\langle 0, 1, +, \times \rangle$, interpreted over the reals.

We define the language TARSKI ALGEBRA to be the set of all encodings of true first order statements made in this language.

Surprisingly, passing to the real numbers makes this easier. Not only is it decidable (unlike realizing the same formal language with multiplication over the naturals), it is more efficient than Presburger arithmetic.

Theorem 8. TARSKI ALGEBRA $\in$ DOUBLE EXP *and there exists* $\varepsilon > 0$ *such that any program solving* TARSKI ALGEBRA *runs in time* $\Omega(2^{\varepsilon n})$*, so* TARSKI ALGEBRA $\notin$ P.

Lower bounds in Theorems 7 and 8 are proved by ad hoc simulations of an arbitrary time-bounded machine in the respective theories, followed by a straightforward application of Theorem 5; we refer to [6, Theorem 6] for further details. In modern complexity theory, however, we crucially depend on more systematic concept of *reductions* that make one of its main tools.

1.2.3 Reductions

We start with Cook-Turing reductions even if they are used less frequently that many-one Karp reductions. We will assume going forward that all machines are equipped with a clock and halt in some time such that they are total.

Definition 14 (Oracle Turing Machine)**.** An oracle Turing machine is a Turing machine with an extra "oracle tape" and 3 designated states, q_{ask}, q_{yes}, and q_{no}.

When the Turing machine enters state q_{ask}, it "asks" an "oracle" whether the contents of the oracle tape belongs to some language L, and it branches to one of the states q_{yes} or q_{no} depending on the answer.

If M is an oracle machine then M^L is the realization of the oracle machine with a language $L \subseteq \{0, 1\}^*$ for the oracle.

If we write something like $\mathsf{P}^{\mathsf{PSPACE}}$, it is important to remember that P here is a class of machines whereas PSPACE is a class of languages, i.e.,

$$\mathsf{P}^{\mathsf{PSPACE}} = \left\{ M^L : M \text{ polytime oracle machine}, L \in \mathsf{PSPACE} \right\}.$$

Definition 15 (Cook-Turing Reduction)**.** We say that L reduces to L', and write $L \preceq_p L'$, if $L \in \mathsf{P}^{L'}$.

Example 2. If $L_1 \notin \mathsf{P}$ and $L_1 \preceq_p L_2$ then we must have that $L_2 \notin \mathsf{P}$ (if $L_2 \in \mathsf{P}$ then $L_1 \in \mathsf{P}^{L_2} \subseteq \mathsf{P}^{\mathsf{P}} = \mathsf{P}$ (a polynomial oracle doesn't help a polynomial machine, since it can simply use a subroutine to simulate the oracle)).

Lecture 3

Date: April 1st, 2025

Cook–Turing reductions is one possibility but in complexity theory we generally care about a simpler and more restricted form of reductions called Karp reductions or many-one reductions.

For the first time, it is critical that we are talking about *languages* rather than arbitrary problems.

Definition 16 (Many-one reduction)**.** We say that <u>L many-one reduces to L'</u> (generally just say "reduces") and write $L \leq_p L'$ if and only if $\exists f : \{0,1\}^* \to \{0,1\}^*$ such that $f \in \mathsf{P}$ and $\forall x$, $x \in L \iff f(x) \in L'$).

Remark (Cook–Turing reductions vs Karp reductions). Karp reductions are clearly no stronger than Cook–Turing reductions, since if $L \leq_p L'$ by a function $f \in \mathsf{P}$, we can take the machine M computing f and modify it to simply write $f(x)$ to the oracle tape, then query the oracle and output whatever the oracle says.

Conversely, $L \preceq_p L'$ need not imply that $L \leq_p L'$, since $L \preceq_p$ co-L trivially (simply write the input to the oracle tape, then output the negation of the oracle's answer) while if we take any computably-enumerable-complete language L, it does not hold that $L \leq_p$ co-L.

Definition 17 (Hard and complete languages)**.** Given a complexity class $\mathcal{C}$, a language L is called <u>$\mathcal{C}$-hard</u> if for every language $L' \in \mathcal{C}$, $L' \leq_p L$.

L is called <u>$\mathcal{C}$-complete</u> if L is $\mathcal{C}$-hard and $L \in \mathcal{C}$.

Remark (Existence of complete languages). Most "reasonable" complexity classes have complete languages. This includes all complexity classes discussed so far. In particular, the "universal machine" type languages work for them.

Remark. Every language in P and smaller classes are complete with respect to polynomial time reductions. Hence when we consider classes smaller than P, such as linear time or the siblings of L, we will be interested in weaker reductions like logspace. We will talk about this in due time.

1.3 Nondeterministic time, NP-completeness, and Cook–Karp–Levin theorem

1.3.1 Nondeterministic time

Definition 18 (Nondeterministic Turing machines)**.** A <u>nondeterministic Turing machine</u> is a Turing machine with an additional triple of states, $q_?$, q_{yes}, and q_{no}. If the machine enters the state $q_?$, it nondeterministically transitions to either q_{yes} or q_{no} (it can be thought of as "guessing" the "correct" answer).

We say that a nondeterministic machine M accepts on x if there exists at least one sequence of nondeterministic "guesses" such that $M(x)$ reaches an accept state.

Definition 19 (Nondeterministic time)**.** We say that the class of <u>nondeterministic time $t(n)$ languages</u>, denoted $\mathsf{NTIME}[t(n)]$, is the class of languages

$$\{L : \exists \text{ nondeterministic TM } M \text{ s.t. } M \text{ decides } L \text{ and has running time } O(t(n))\},$$

where the running time of a nondeterministic machine is defined to be the maximum run time over all paths.

We define the class <u>nondeterministic polynomial time</u>, $\mathsf{NP} = \bigcup_{k \geq 0} \mathsf{NTIME}\left[n^k\right]$.

Remark (Relationship of NP to other classes). Clearly, $\mathsf{P} \subseteq \mathsf{NP}$ since we've simply added a feature which we are not obliged to use. On the other hand $\mathsf{NP} \subseteq \mathsf{PSPACE}$ (to be explained later, but roughly because we can simply try each possible sequence of guesses).

We know

$$\mathsf{P} \subseteq \mathsf{NP} \subseteq \mathsf{PSPACE} \subseteq \mathsf{EXP}$$

and we know by time hierarchy theorem that $\mathsf{P} \subsetneq \mathsf{EXP}$, but have not been able to prove that any one of these containments is proper.

Theorem 9 (Nondeterministic time Hierarchy Theorem, Cook (1972))**.** *If $t(n)$ is time-constructible and $g(n+1) = o(t(n))$ then $\mathsf{NTIME}[g(n)] \subsetneq \mathsf{NTIME}[t(n)]$.*

Additionally, it is worth noting (we will prove this later) that we need not bother to define nondeterministic space because:

Theorem 10 (Savitch's theorem)**.** *$\mathsf{NSPACE}[S(n)] \subseteq \mathsf{DSPACE}\left[S(n)^2\right]$. In particular, $\mathsf{NPSPACE} = \mathsf{PSPACE}$.*

Theorem 11 (Characterization of NP)**.** *$L \in \mathsf{NP}$ if and only if there exists a polynomial $p(n)$ and an "acceptor" language $A(x, w) \in \mathsf{P}$ such that*

$$x \in \mathsf{L} \iff \exists w \in \{0,1\}^{p(n)} \ \text{s.t.}\ A(x, w).$$

The string w is generally called a "witness," "certificate," or "proof" and the acceptor A may be called the "verifier."

Proof. $\implies$ Given a nondeterministic polynomial time machine, we can modify it such that it is deterministic and instead takes an extra input sequence of instructions $w_1, \ldots, w_{p(n)}$ such that every time the machine enters $q_?$, instead of guessing which state to enter next, it reads the next bit of w to determine which way to go.

Then any NP machine is equivalent to a P machine with an extra polynomial length (running time bound) input and the NP machine accepts if and only if there exists any sequence of guesses, i.e., any witness w, such that the P machine accepts.

$\Longleftarrow$ If L has a $p(n)$ and acceptor $A(x, w)$ satisfying these terms then we simply have a nondeterministic machine which first guesses a witness w and then simulates A.

□

Remark (Characterization of NP). This is generally the easier way to think of NP. This characterization of NP describes it as problems where there exists a "proof" that may be hard to find but easy to check.

Theorem 12. $\mathsf{NP} \subseteq \mathsf{PSPACE}$.

Proof. Write down a guess $w = 0^{p(n)}$. Then simulate $A(x, w)$ and accept if it accepts. If it rejects, erase everything except w and x, increment w, and try again.

If every single guess w rejects, then reject. □

1.3.2 NP-completeness and satisfiability

Theorem 13 (Levin (73)). *The language*

$$U_{\mathsf{NP}} = \left\{(e, x, 1^m, 1^t) : \exists u \in \{0,1\}^m \text{ s.t. } \varphi_e(x,u) = 1 \text{ within } t \text{ steps}\right\}$$

is NP*-complete.*

Proof. Belongs to NP by alternate characterization more or less immediately. NP-hard because for every NP problem there exists some machine e, witness length m, and time bound t, both polynomial in $n = |x|$, which can be output by the reduction. □

Definition 20 (Formulas and conjunctive normal form). A propositional formula is a boolean formula using $\neg, \wedge, \vee$ on variables $x_1, \ldots, x_n$. A literal ℓ is an expression of the form x_i or $\neg x_i$ (which we also write as $\bar{x}_i$). A clause is a disjunction of literals, e.g., $(x_3 \vee \bar{x}_7 \vee x_{11})$. A conjunctive normal form formula or CNF is a conjunction of a set of clauses.

We say that a formula φ is satisfiable if there exists an assignment of variables $x_1, \ldots, x_n$ such that $\varphi(x_1, \ldots, x_n)$ is true, and unsatisfiable otherwise.

We define SAT to be the set of all satisfiable formulas and 3-SAT to the set of all satisfiable CNF formulas where the clauses consist of at most 3 literals each.

Theorem 14 (Cook (72)). SAT *is* NP*-complete.*

Proof. It is trivial that SAT $\in$ NP, since the witness is exactly the candidate assignment of variables, and evaluating a formula to see if the given candidate assignment is satisfying can be done in polynomial time.

It remains to show that $L \in \mathsf{NP} \implies L \leq_p$ SAT. We will do this using locality of Turing machines (i.e., the fact that the evolution of a given cell on the tape from one time step to the next depends only on it and the neighboring cells, considering the head to be part of the cell it's over). The high level idea is that we will write down a formula where the input is interpreted as a description of an entire computation history (a list of configurations), and the formula will:

- validate that each individual configuration is valid (i.e., assigns one symbol to each tape cell and has exactly one head in exactly one state),
- validate that the initial configuration is a legal initial configuration (i.e., the head is at the beginning of the tape in the start state) and that it matches the hard-coded input being considered,
- validate that each successive configuration is a legal transition from the previous configuration [1] (i.e., for each cell, if the head wasn't in one of the neighboring cells, that cell remains unchanged, and if it was in one of the neighboring cells, that the new state of the cell is one of the possible ones given the previous state), and
- finally, confirm that the final configuration of the computation history is an accepting configuration.

[1] note that since these are nondeterministic machines, there may be multiple legal transitions; since a nondeterministic machine accepts if there exists any accepting computation history, we allow any one of the legal moves at each step

In more detail, if we want to reduce from a language $L \in \mathsf{NP}$: Let M be a machine deciding L and running in time exactly $p(n)$ (such a machine exists since given any machine M_0 deciding L, we can take any polynomial bounding t_{M_0} and add a clock to the machine counting up to that polynomial; if the machine would hit a halt state before running out the clock, we just wait to run out the clock while preserving that halt state). We will construct a CNF formula $f(u)$, where u is the input.

We use a space-time diagram. We can create a grid of cells, where each row of the table represents a configuration of the Turing machine at a given time step, each cell storing the symbol stored there along with whether the head of the machine is in it and, if so, the state of the head.

This grid represents the entire computation history of the machine, and we will write down a propositional formula expressing that the formula must be a valid accepting computation history.

Note that the height and width of the grid are both $p(n)$, so the whole thing has polynomial size.

We use i to represent the horizontal coordinate (space) and j for the vertical coordinate (time). For a symbol $s \in \Sigma$, we add a variable x_{ijs} to represent "the cell i at time j has symbol s." (Note that it is just an intuition, this intended meaning will be enforced by the formula.)

Similarly, for $q \in Q$, we let y_{ijq} represent "at time j, the head is in state q and the ith cell." Then we add the constraints, as described above.

Individual configurations First, we constrain that each cell must have exactly one state: for every i, j, we write down $\bigvee_{s \in \Sigma} x_{ijs}$ and for each $s_1 \neq s_2 \in \Sigma$, we add $\bar{x}_{ijs_1} \vee \bar{x}_{ijs_2}$.

Next, we constrain that the head exists in exactly one place and state at each time: for each j, we write $\bigvee_{i, q \in Q} y_{ijq}$ and for each $(i_1, q_1) \neq (i_2, q_2)$ we add $\bar{y}_{i_1 j q_1} \vee \bar{y}_{i_2 j q_2}$.

Initial and final configurations We add clauses x_{i0u_i} for $i \leq n$ and x_{i0s_0} for $i \geq n+1$, and similarly final condition: $\bigvee_i y_{i,p(n),q_{\text{acc}}}$ (at the final time step, the head is in the accepting state, wherever it may be).

Transition between configurations Finally, we must constrain that the evolution from one configuration to the next is valid. Note that we can determine x_{ijs} and y_{ijq} from $x_{i-1,j-1,s'}, x_{i,j-1,s'}, x_{i+1,j-1,s'}$ (ranging over all $s' \in \Sigma$) and $y_{i-1,j-1,q'}, y_{i,j-1,q'}, y_{i+1,j-1,q'}$ (ranging over all $q' \in Q$), with potentially multiple legal options based on nondeterminism. Hence the fact that the transition is locally legal can be expressed by a Boolean expression depending only on a *bounded* (i.e. depending only on the sizes of the alphabets Σ, Q) number of variables. It can be converted to a CNF in a straightforward way, and the size remains bounded (exponent of a constant is still a constant!).

Our reduction from L to SAT on input u produces this propositional formula. Then if $M(u)$ accepts, there is a valid computational history which accepts and the description of this computational history will satisfying $f(u)$. On the other hand, if $M(u)$ rejects, there will be no assignment of variables which represents both a valid and an accepting configuration because we have ensured that the computation history evolves legally and starts correctly. $\square$

The classification of problems into P and NP-complete problems is remarkably successful: for an enormous number of "reasonable" problems that you may naturally think of, they are either straightforwardly in P or straightforwardly NP-complete. The primary tool for showing NP-completeness is *reductions*, rather than going directly as we did for SAT.

Because Karp reductions are transitive (simply compose the reduction functions), if L is NP-complete and $L \leq_p L'$ then L' must also be NP-complete (if $L'' \in \mathsf{NP}$ and reduces to L by f and L reduces to L' by g then L'' reduces to L' by $g \circ f$.)

Because we want to reduce our NP-complete languages to other languages, it is easier to work with "weaker" languages. For example, the language 3-SAT is easier to reduce from than SAT, since 3-SAT is a strict "sub-problem". But we will show in the next lecture that 3-SAT is NP-complete as well.

Lecture 4

Date: April 3rd, 2025

Recall:

- $L \leq_p L' \equiv \exists f \in \mathsf{P}$ such that $x \in L \iff f(x) \in L'$.
- L is NP-complete iff $L \in \mathsf{NP}$ and $\forall L' \in \mathsf{NP}$, $L' \leq_p L$.
- if $L \leq_p L'$ and L is NP-hard then L' is NP-hard. Correspondingly, if $L' \in \mathsf{NP}$, then it becomes NP-complete.
- SAT, the language of satisfiable boolean CNF formulas, is NP-complete.

Theorem 15. SAT $\leq_p$ 3-SAT. *Correspondingly,* 3-SAT *is* NP*-complete.*

Proof. We will define our reduction f as follows:

Given a clause $\ell_1 \vee \cdots \vee \ell_r$ for some $r \geq 4$, we will convert it into several clauses of 3 literals by introducing additional variables, $y_{C,i}$ for $1 \leq i \leq r$. The idea will be to constrain the variables such that $y_{C,i} \equiv \ell_1 \vee \cdots \vee \ell_i$ and then add an additional constraint that $y_{C,i}$ holds.

We will add clauses for $y_{C,1} \iff \ell_1$ and $y_{C,i} \iff y_{C,i-1} \vee \ell_i$, and $y_{C,r}$. Each of these clauses can be written as 3-CNF (using multiple clauses):

$$\begin{aligned}
(y_{C,1} \iff \ell_1) &\equiv (y_{C,1} \implies \ell_1) \wedge (y_{C,1} \impliedby \ell_1) \\
&\equiv (\bar{y}_{C,1} \vee \ell_1) \wedge (y_{C,1} \vee \bar{\ell}_1) \\
(y_{C,i} \iff y_{C,i-1} \vee \ell_i) &\equiv (y_{C,i} \implies y_{C,i-1} \vee \ell_i) \wedge (y_{C,1} \impliedby y_{C,i-1} \vee \ell_i) \\
&\equiv (\bar{y}_{C,i} \vee y_{C,i-1} \vee \ell_i) \wedge (y_{C,1} \vee \overline{y_{C,i-1} \vee \ell_i}) \\
&\equiv (\bar{y}_{C,i} \vee y_{C,i-1} \vee \ell_i) \wedge (y_{C,1} \vee \bar{y}_{C,i-1}) \wedge (y_{C,1} \vee \bar{\ell}_i).
\end{aligned}$$

Then if φ is satisfiable, we can simply evaluate the values of $y_{C,i}$ for each clause according to their intended meaning ("do not lie, tell the truth" is a good guiding principle in this direction). Conversely, if $f(\varphi)$ is satisfiable then we must have that simultaneously each $y_{C,r}$ is true and $y_{C,r} \iff y_{C,r-1} \vee \ell_r \iff \cdots \iff \ell_1 \vee \cdots \vee \ell_r$ so φ is satisfiable ("every liar will get caught"). □

For the future, we will assume all 3-SAT instances have all clauses of width exactly 3, which we can do since, e.g., $(\ell_1 \vee \ell_2)$ is the same as $(\ell_1 \vee \ell_1 \vee \ell_2)$.

1.3.3 The importance of $\mathsf{P} \overset{?}{=} \mathsf{NP}$

We care about $\mathsf{P} \overset{?}{=} \mathsf{NP}$ because an enormous class of varied problems are either in P or NP-complete (and thus have no efficient algorithm unless $\mathsf{P} = \mathsf{NP}$). A constructive proof that $\mathsf{P} = \mathsf{NP}$ would potentially imply efficient solutions to all of them, including the problem of checking for the existence of a short proof of any mathematical theorem. A proof of $\mathsf{P} \neq \mathsf{NP}$ (generally expected to hold) would imply that a broad class of problems are fundamentally hard.

Logic

Consider the Peano arithmetic PA or set theory ZFC. The question of whether PA $\vdash \varphi$ is undecidable, but the question PA $\vdash_n \varphi$, i.e., if there exists a proof of length n, posed as $(\varphi, 1^n)$, is NP-complete.

In order to reduce from SAT, we map a given $\varphi(x_1, \ldots, x_n)$ to (an encoding of)

$$(\exists x_1, \ldots, x_n \text{ s.t. } \varphi(x_1, \ldots, x_n), 1^{p(|\varphi|)})$$

for an appropriate polynomial $p(n)$. Then if φ is satisfiable, there will be some suitable short proof of the statement in the first-order logic based on the satisfying assignment. Conversely, if we assume that PA is sound then if that φ is unsatisfiable, there will be no proof of any length.

Number Theory

We ask whether a polynomial $p(x_1, \ldots, x_n) \in \mathbb{Z}[x_1, \ldots, x_n]$ has any integer roots, i.e.,

$$\exists x_1, \ldots, x_n \; p(x_1, \ldots, x_n) = 0$$

By pulling out all negative coefficients to the other side, we can rewrite as $p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n)$ for p, q with positive coefficients.

This is again undecidable, but if we replace q with some positive integer γ and write everything in binary, we observe that this problem is now in NP: if $p(a_1, \ldots, a_n) = \gamma$ then since all coefficients are positive, we must have $a_1, \ldots, a_n \leq \gamma$, so the total length of the solution is polynomial.

Thus, the problem is in NP.

Theorem 16 (Manders, Adleman [7]). *The above problem is NP-complete. Further, $\varphi \in$ SAT reduces to an equation of the form $Ax^2 + By = r$ for positive integers A, B, and r.*

Proof. Omitted. □

Graph Theory

Definition 21 (Independent set). Let G be an arbitrary graph. Let $\alpha(G)$ be the size of the largest independent set (a set of vertices is independent if no pair of them has an edge between them). We define the problem INDEPENDENT SET to be the set of pairs (G, k) such that $\alpha(G) \geq k$.

Theorem 17. INDEPENDENT SET *is NP-complete.*

Proof. We will proof by reducing from 3-SAT. Given an instance $\varphi \equiv C_1 \wedge C_2 \wedge \cdots \wedge C_m$ where each $C_i \equiv \ell_{i,1} \vee \ell_{i,2} \vee \ell_{i,3}$, we will construct a graph G and a natural number k such that φ is satisfiable if and only if $\alpha(G) \geq k$.

For each clause C_i, we create a triangle T_i of pairiwise connected vertices (i.e. a clique) in the graph and we set $k = m$. We will "label" (for our own convenience) the vertices of the triangle T_i with literals $\ell_{i,1}, \ell_{i,2}, \ell_{i,3}$.

Then we add an edge $(\ell_{i,a}, \ell_{i',a'})$ whenever $\ell_{i,a} = \bar{\ell}_{i',a'}$. For example, if $\ell_{i,a} = x_j$ and $\ell_{i',a'} = \bar{x}_j$, we will add the edge $(\ell_{i,a}, \ell_{i',a'})$. That completes the description of $f(\varphi) = (G, k)$.

The high level idea is that:

- By connecting the literals in each clause in a triangle, we ensure that at most one of them is selected in any independent set. By requiring, at the same time, an independent set of size m, we then ensure that *at least* one vertex is selected from each clause.
- The additional edges ensure that one cannot select two "inconsistent" literals at the same time (i.e., literals which are negations of each other).

By ensuring that you must select at least one literal from each clause and at the same time ensuring that you cannot select inconsistent literals, we make independent sets of size k equivalent to satisfying assignments.

Remark. This is a simple example of what is called a "gadget reduction" where we've created gadgets in the graph to represent the clauses of our instances. Gadget reductions can get much more complicated.

In more detail: we must show $\varphi \in$ 3-SAT $\iff f(\varphi) \in$ INDEPENDENT SET.

$\Longrightarrow$ If φ is satisfiable, pick any satisfying assignment, and then from each clause C_i arbitrarily pick one of the satisfied literals. Add the corresponding vertex $\ell_{i,a} \in T_i$ to our independent set S.

Trivially, S has size m since we added one vertex from each of m clauses. Further, S is independent, since we selected only one vertex from each triangle and we cannot have selected both $\ell_{i,a}$ and $\bar{\ell}_{i,a}$, so we cannot have selected two connected vertices in different triangles.

$\Longleftarrow$ Suppose we are given an independent set S of size m. First, we must have exactly one vertex selected in each T_i, since there are m triangles and we cannot select two vertices in one triangle.

If a literal $\ell_{i,a}$ appears in S, we assign the underlying variable to make $\ell_{i,a} = 1$ (i.e., if $\ell_{i,a} = x_j$, we assign $x_j = 1$, and if $\ell_{i,a} = \bar{x}_j$ then we assign $x_j = 0$).

Observe that this process can never cause us to assign $x_j = 1$ and $x_j = 0$, since if $\ell_{i,a} = x_j$ and $\ell_{i',a'} = \bar{x}_j$ then we have an edge $(\ell_{i,a}, \ell_{i',a'})$ and so cannot have them both in S.

By selecting one from each triangle, we have satisfied a literal in every clause and thus created a satisfying assignment. If a variable has not been assigned in the process, we can assign it arbitrarily.

□

Definition 22 (Clique)**.** We define $\omega(G)$ to be the size of the largest clique in G. We define

$$\textsc{Clique} = \{(G,k) : \omega(G) \geq k\}$$

Definition 23 (Vertex cover)**.** We define $\textsc{Vertex Cover}$ to be the set of pairs (G,k) such that there exists a subset $S \subseteq V$ with $|S| \leq k$ such that every edge has in E has an end point in S.

Theorem 18. *$\textsc{Clique}$ and $\textsc{Vertex Cover}$ are NP-complete.*

Proof. For $\textsc{Clique}$, simply take graph complement to reduce to $\textsc{Independent Set}$.

For $\textsc{Vertex Cover}$, simply take k to $|V| - k$, since if S is an independent set then $V \setminus S$ is a vertex cover and if S is a vertex cover then $V \setminus S$ is an independent set. □

Subset Sum Problems

Definition 24 (Subset vector sum)**.** We define

$$\textsc{Subset Vec Sum} = \left\{(d, v_1, \ldots, v_n, v_0) : v_i \in \mathbb{N}^d \wedge \exists I \subseteq [n] \text{ s.t. } \sum_{i \in I} v_i = v_0\right\}$$

Theorem 19. *$\textsc{Subset Vec Sum}$ is NP-complete.*

Proof. We reduce from $\textsc{Independent Set}$. Given a graph G and k, let $n = |V|, m = |E|$. Let $d = m + 1$. We will write down an $(m+n) \times (m+1)$ matrix M as the vertical concatenation of two matrices M_1, M_2. M_1 is an $n \times (m+1)$ matrix which is exactly the incidence matrix of G with an extra column of 1's appended on the right. M_2 will be the $m \times m$ identity matrix with an extra column of 0's appended on the right.

Our $v_1, \ldots, v_{m+n}$ will be the rows of this matrix and our target vector v_0 will be $(k, 1, \ldots, 1)$.

To see that this reduction is correct, observe:

$\Longrightarrow$ If there is an independent set S of size k, we pick the k row vectors in M_1 corresponding to the vertices in s, which will get us a count of k in the last column and counts of 0 or 1 in every other column (since a count of ≥ 2 would imply 2 vertices incident to the same edge, which you can't have in an independent set). For every 0, we pick the corresponding vector from the identity matrix in M_2 to fill it out.

$\Longleftarrow$ If we have a subset vector sum I, it must have exactly k of the rows from M_1. We ignore the rows from M_2 and take the vertices corresponding to the k selected rows. Since the sum of the rows has a 1 in each column, we cannot have 2 vertices which are incident to the same edge (since the column corresponding to that edge would have a sum ≥ 2 in that case).

□

Definition 25 (Subset sum)**.** We define

$$\textsc{Subset Sum} = \left\{(\ell_1, \ldots, \ell_n, \ell_0) : \ell_i \in \mathbb{N} \wedge \exists I \subseteq [n] \text{ s.t. } \sum_{i \in I} \ell_i = \ell_0\right\}$$

Theorem 20. *$\textsc{Subset Sum}$ is NP-complete.*

Proof. We reduce from SUBSET VEC SUM. While we cannot have an injective homomorphism $\mathbb{N}^d \to \mathbb{N}$, we can use instead what in arithmetic combinatorics is called a "Freiman isomorphism", i.e. a homomorphism that is as good as injective for our current purposes. Namely, we choose K "sufficiently large" and map $(a_1, \ldots, a_d) \mapsto^{\varphi} (a_1 + a_2K + \cdots + a_dK^{d-1})$. In other words, we have effectively written down our vectors as numbers in some enormous base K.

More specifically, let $K > n \cdot \max_{i,j} v_{i,j}$. Then the addition of vectors never has any "carry" (in our base K representation) and so is performed "digit-by-digit." But this is exactly vector addition, giving us the desired reduction. □

Scheduling Problems

We have a bunch of tasks of lengths $\ell_1, \ldots, \ell_m$ and a bunch of processors, and dependency relations between tasks. This yields various families of problems for optimal scheduling.

Definition 26 (Most basic scheduling)**.** A list of m independent tasks of lengths $\ell_1, \ldots, \ell_m$ and a time budget t belongs to SCHEDULING if it is possible to schedule the tasks $\ell_1, \ldots, \ell_m$ on two processors to complete all tasks in time t.

Mathematically, does there exist $I \subseteq [n]$ such that $\sum_{i \in I} \ell_i \leq t$ and $\sum_{i \notin I} \ell_i \leq t$.

Theorem 21. SCHEDULING *is* NP*-complete.*

Bibliography

[1] J. Hartmanis and R. E. Stearns. On the computational complexity of algorithms. *Trans. Amer. Math. Soc.*, 117:285–306, 1965.

[2] M. Blum. A machine-independent theory of the complexity of recursive functions. *Journal of the ACM*, 14:322–336, 1967.

[3] B. A. Trakhtenbrot. The complexity of algorithms and computations. Lecture Notes (Novosibirsk University), 1967.

[4] A Borodin. Complexity classes of recursive functions and the existence of complexity gaps. In *Proceedings of the* 1*st ACM Symposium on the Theory of Computing*, pages 67–78, 1969.

[5] S. Arora and B. Barak. *Computational Complexity: a Modern Approach.* Cambridge University Press, 2009.

[6] M. J. Fischer and M. O. Rabin. Super-exponential complexity of presburger arithmetic. Technical Report 43, MIT, February 1974.

[7] K. L. Manders and L. Adleman. Np-complete decision problems for binary quadratics. *Journal of Computer and System Sciences*, 16:168–184, 1979.