Reading: Sipser §7.1, §7.2.
Objective of Complexity Theory

- To move the focus:
  - from what it is possible in principle to compute
  - to what is feasible to compute given “reasonable” resources

- For us the principle “resource” is time, though it could also be memory (“space”) or hardware (switches)
What is the “speed” of an algorithm?

Def: A TM $M$ has **running time** $t : \mathcal{N} \to \mathcal{N}$ iff for all $n$, $t(n)$ is the maximum number of steps taken by $M$ over all inputs of length $n$.

→ implies that $M$ halts on every input

→ in particular, every decision procedure has a running time

→ time used as a function of size $n$

→ worst-case analysis
Example running times

Running times are generally increasing functions of $n$

\[ t(n) = 4n \]
\[ t(n) = 2n \cdot \lceil \log n \rceil \]
\[ \lceil x \rceil = \text{least integer} \geq x \text{ (running times must be integers)} \]
\[ t(n) = 17n^2 + 33 \]
\[ t(n) = 2^n + n \]
\[ t(n) = 2^{2n} \]
“Table lookup” provides a speedup for finitely many inputs

**Claim:** For every decidable language $L$ and every constant $k$, there is a TM $M$ that decides $L$ with running time $t(n) = n$ for all $n \leq k$.

**Proof:**
“Table lookup” provides a speedup for finitely many inputs

**Claim:** For every decidable language $L$ and every constant $k$, there is a TM $M$ that decides $L$ with running time $t(n) = n$ for all $n \leq k$.

**Proof:**

- Answers to any finite number of “questions” can be built into the finite control of a TM, so that it takes no more time to answer these questions than the time needed to read the input.

- So any **finite** language can be decided in the time needed to read the input.

- (Though size of TM grows in proportion to the number of hard-wired questions!)

  ⇒ study behavior only of Turing machines $M$ deciding infinite languages, and only by analyzing the running time $t(n)$ as $n \to \infty$. 
Why bother measuring TM time, when TMs are so miserably inefficient?

- **Answer:** Within limits, multitape TMs are a reasonable model for measuring computational speed.

- The trick is to specify the right amount of “slop” when stating that two algorithms are “roughly equivalent”.

- Even coarse distinctions can be very informative.
**Def:** Let $t : \mathbb{N} \rightarrow \mathbb{R}^+$. Then \(\text{TIME}(t)\) is the class of languages \(L\) that can be decided by some **multitape** TM with running time \(\leq t(n)\).

\[\text{e.g. } \text{TIME}(10^{10} \cdot n), \text{TIME}(n \cdot 2^n)\]

\(\mathbb{R}^+ = \text{positive real numbers}\)

**Q:** Is it true that with more time you can solve more problems? i.e., if \(g(n) < f(n)\) for all \(n\), is \(\text{TIME}(g) \subsetneq \text{TIME}(f)\)?

**A:** Not exactly . . .
Linear Speedup Theorem

Let \( t : \mathbb{N} \rightarrow \mathbb{R}^+ \) be any function s.t. \( t(n) \geq n \) and \( 0 < \varepsilon < 1 \), then for every \( L \in \text{TIME}(t) \), we also have \( L \in \text{TIME}(\varepsilon \cdot t(n) + n) \)

- \( n = \) time to read input
- Note implied quantification:
  \[ (\forall \text{ TM } M)(\forall \varepsilon > 0)(\exists \text{ TM } M') \; M' \text{ is equivalent to } M \text{ but runs in fraction } \varepsilon \text{ of the time.} \]
- “Given any TM we can make it run, say, 1,000,000 times faster on all inputs.”
Proof of Linear Speedup

Let $M$ be a TM deciding $L$ in time $T$.

A new, faster machine $M'$:

1. Copies its input to a second tape, in compressed form.

2. Moves head to beginning of compressed input.

3. Simulates the operation of $M$ treating all tapes as compressed versions of $M$’s tapes.

(Compression factor $= 3$ in this example—actual value TBD at end of proof)
Analysis of linear speedup

- Let the “compression factor” be $c$ ($c = 3$ here), and let $n$ be the length of the input.
- Running time of $M'$:
  1. $n$ steps.
  2. $\lceil n/c \rceil$ steps
     - $\lceil x \rceil = \text{smallest integer } \geq x$
  3. takes ?? steps.
How long does the simulation (3) take?

- $M'$ remembers in its finite control which of the $c$ “subcells” $M$ is scanning.

- $M'$ keeps simulating $c$ steps of $M$ by 8 steps of $M'$:
  1. Look at current cell on either side.
     (4 steps to read $3c$ symbols)
  2. Figure out the next $c$ steps of $M$.
     (can’t depend on anything outside these $3c$ subcells)
  3. Update these 3 cells and reposition the head.
     (4 steps)
End of simulation analysis

- It must do this $\lceil t(n)/c \rceil$ times, for a total of $8 \cdot \lceil t(n)/c \rceil$ steps.
- Total of $\leq (10/c) \cdot t(n) + n$ steps of $M'$ for sufficiently large $n$.
- If $c$ is chosen so that $c \geq 10/\varepsilon$ then $M'$ runs in time $\varepsilon \cdot t(n) + n$. 
“Throwing hardware at a problem” can speed up any algorithm by any desired constant factor

E.g. moving from 8 bit → 16 bit → 32 bit → 64 bit parallelism

Our theory does not “charge” for huge capital expenditures to build big machines, since they can be used for infinitely many problems of unbounded size.

This complexity theory is too weak to be sensitive to multiplicative constants—so we study growth rate
Growth Rates of Functions

- We need a way to compare functions according to how **fast** they increase, not just how **large** their values are.

- Intuitively, $f(n) = n^2$ grows faster than $g(n) = 10^{10} \cdot n$, even though for many values of $n$, $g(n) > f(n)$.

- **Def:** For $f : \mathbb{N} \to \mathbb{R}^+$, $g : \mathbb{N} \to \mathbb{R}^+$, we write $g = \mathcal{O}(f)$ if there exists $c, n_0 \in \mathbb{N}$ such that $g(n) \leq c \cdot f(n)$ for all $n \geq n_0$.
  - Binary relation: we could write $g = \mathcal{O}(f)$ as $g \preceq f$.
  - “If $f$ is scaled up uniformly, it will be above $g$ at all but finitely many points.”
  - “$g$ grows no faster than $f$.”
  - Also write $f = \Omega(g)$. 
Examples of Big-$\mathcal{O}$ notation

- If $f(n) = n^2$ and $g(n) = 10^{10} \cdot n$
  
  $g = \mathcal{O}(f)$ since $g(n) \leq 10^{10} \cdot f(n)$ for all $n \geq 0$
  
  where $c = 10^{10}$ and $n_0 = 0$

- Usually we would write: “$10^{10} \cdot n = \mathcal{O}(n^2)$”
  
  i.e. use an expression to name a function

- By Linear Speedup Theorem, $\text{TIME}(t)$ is the class of languages $L$ that can be decided by some multitape TM with running time $\mathcal{O}(t(n))$ (provided $t(n) \geq 1.01n$)
Examples

- $10^{10} \cdot n = \mathcal{O}(n^2)$.
- $1764 = \mathcal{O}(1)$.
  1: The constant function $1(n) = 1$ for all $n$.
- $n^3 \neq \mathcal{O}(n^2)$.
- Time $\mathcal{O}(n^k)$ for fixed $k$ is considered “fast” (“polynomial time”)
- Time $\Omega(k^n)$ is considered “slow” (“exponential time”)
- Does this really make sense?
More Relations

**Def:** We say that \( g = o(f) \) iff for every \( \varepsilon > 0 \), \( \exists n_0 \) such that 
\[
g(n) \leq \varepsilon \cdot f(n) \quad \text{for all } n \geq n_0.
\]
- Equivalently, \( \lim_{n \to \infty} \frac{g(n)}{f(n)} = 0 \)
- “\( g \) grows more slowly than \( f \).”
- Also write \( f = \omega(g) \).

**Def:** We say that \( f = \Theta(g) \) iff \( f = O(g) \) and \( g = O(f) \).
- “\( g \) grows at the same rate as \( f \)”
- An equivalence relation between functions.
- The equivalence classes are called **growth rates**.
- Because of linear speed up, \( \text{TIME}(t) \) is really the union of all growth rate classes \( \preceq \Theta(t) \).
More Examples

- **Polynomials (of degree $d$):**
  \[ f(n) = a_d n^d + a_{d-1} n^{d-1} + \cdots + a_1 n + a_0, \text{ where } a_d > 0. \]
  - $f(n) = \mathcal{O}(n^c)$ for $c \geq d$.
  - $f(n) = \Theta(n^d)$
    - “If $f$ is a polynomial, then lower order terms don’t matter to the growth rate of $f$”
  - $f(n) = o(n^c)$ for $c > d$.
  - $f(n) = n^{\mathcal{O}(1)}$. 
More Examples

- **Exponential Functions:** \( g(n) = 2^n \Theta(1) \).
  
  - Then \( f = o(g) \) for any polynomial \( f \).
  
  - \( 2^n \alpha = o(2^n \beta) \) if \( \alpha < \beta \).

- **What about** \( n^{\log n} = 2^{\log^2 n} \)?

  Here \( \log x = \log_2 x \)

- **Logarithmic Functions:**

  \( \log_a x = \Theta(\log_b x) \) for any \( a, b > 1 \)
Polynomial Time

Reading: Sipser §7.2.
Q: How quickly can a 1-tape TM $M_2$ simulate a multitape TM $M_1$?

- If $M_1$ uses $f(n)$ time, then it uses $\leq f(n)$ tape cells.
- $M_2$ simulates one step of $M_1$ by a complete sweep of its tape. This takes $\leq k \cdot f(n)$ steps.

\[ \therefore M_2 \text{ uses } \leq f(n) \cdot O(f(n)) = O(f^2(n)) \text{ steps in all.} \]

So $L \in \text{TIME}_{\text{multitape TM}}(f) \Rightarrow L \in \text{TIME}_{\text{1-tape TM}}(O(f^2))$

- Similarly for
  - 2-D Tapes
  - Random Access TMs . . .
Basic thesis of complexity theory

- **Extended Church-Turing Thesis**: Every “reasonable” model of computation can be simulated on a Turing machine with only a polynomial slowdown.

- Counterexamples?
  - Randomized computation.
  - Parallel computation.
  - Analog computers.
  - DNA computers.
  - Quantum computers.
Definition: Let \( P = \bigcup_{p} \text{TIME}(p) \), where \( p \) is a polynomial

\[
= \bigcup_{k \geq 0} \text{TIME}(n^k)
\]

- \( P \) is also known as PTIME or \( \mathcal{P} \)
- *Coarse* approximation to “efficient algorithm”
Model-Independence of P

Although P is defined in terms of TM time, P is a stable class, independent of the computational model.

(Provided the model is reasonable)

Justification:

- If A and B are different models of computation,
  \[ L \in \text{TIME}_A(p_1(n)), \]
  and B can simulate a time \( t \) computation of A in time \( p_2(t) \),
  then \( L \in \text{TIME}_B(p_2(p_1(n))) \).

- Polynomials are closed under composition, e.g.
  \[ f(n) = n^2, \quad g(n) = n^3 + 1 \Rightarrow f(g(n)) = (n^3 + 1)^2 = n^6 + 2n^3 + 1. \]
How much does representation matter?

- How big is the representation of an $n$-node directed graph?
  - ... as a list of edges?
  - ... as an adjacency matrix?
- How big is the representation of a natural number $n$?
  - ... in binary?
  - ... in decimal?
  - ... in unary?
For which of the following do we know polynomial-time algorithms?

- Given a DFA $M$ and a string $w$, decide whether $M$ accepts $w$.
  - What is the “size” of a DFA?

- Given an NFA $N$, construct an equivalent DFA $M$. 

More computational problems: are they in P?

- Given an NFA $N$ and a string $w$, decide whether $N$ accepts $w$.

- Given a regular expression $R$, construct an equivalent NFA $N$.

- Given a CFG $G$ and a string $w$, decide whether $G$ generates $w$. 
And more computational problems: are they in P?

- Given two numbers $n$, $m$, compute their product.
  - What is the “size” of the numbers?

- Given a number $n$, decide if $n$ is prime.

- Given a number $n$, compute $n$’s prime factorization.
Another way of looking at $P$

- Multiplicative increases in time or computing power yield multiplicative increases in the size of problems that can be solved.

- If $L$ is in $P$, then there is a constant factor $k$ such that:
  - If you can solve problems of size $s$ within a given amount of time,
  - and you are given a computer that runs twice as fast, then
  - you can solve problems of size $k \cdot s$ on the new machine in the same amount of time.

- E.g. if $L$ is decidable in $\mathcal{O}(n^d)$ time, then with twice as much time you can solve problems $2^{\frac{1}{d}}$ as large.
Exponential time

- \( E = \bigcup_{c > 0} \text{TIME}(c^n) \)

- For problems in \( E \), a multiplicative increase in computing power yields only an additive increase in the size of problems that can be solved.

- If \( L \) is in \( E \), then there is a constant \( k \) such that
  - If you can solve problems of size \( s \) within a given amount of time
  - and you are given a computer that runs twice as fast, then
  - you can solve problems only of size \( k + s \) on the new machine using the same amount of time.
NP

Reading: Sipser §7.3
“Nondeterministic Time”

We say that a nondeterministic TM $M$ decides a language $L$ iff for every $w \in \Sigma^*$,

1. Every computation by $M$ on input $w$ halts (in state $q_{\text{accept}}$ or state $q_{\text{reject}}$);
2. $w \in L$ iff there exists at least one accepting computation by $M$ on $w$.
3. $w \notin L$ iff every computation by $M$ on $w$ rejects.

$M$ decides $L$ in **nondeterministic time** $t(\cdot)$ iff for every $w$, every computation by $M$ on $w$ is of length at most $t(|w|)$.
More on Nondeterministic Time

1. Linear speedup holds.

2. “Polynomial equivalence” holds among nondeterministic models

   e.g. \( L \) decided in time \( T \) by a nondeterministic multitape TM

   \[ \Rightarrow L \text{ decided in time } \mathcal{O}(T^2) \text{ by a nondeterministic 1-tape TM} \]

Definition:

\[
\text{NTIME}(t(n)) = \{ L : L \text{ is decided in time } t(n) \text{ by some nondet. multitape TM} \}
\]

\[
\text{NP} = \bigcup_{p \text{ polynomial}} \text{NTIME}(p) = \bigcup_{k \geq 0} \text{NTIME}(n^k).
\]
Clearly $P \subseteq \text{NP}$. But there are problems in NP that are not obviously in $P$ (≠ “obviously not”)

**TSP = TRAVELLING SALESMAN PROBLEM**

- Let $m > 0$ be the number of cities,
- $D : \{1, \ldots, m\}^2 \rightarrow \mathbb{N}$ give the distance $D(i, j)$ between city $i$ and city $j$, and
- $B$ be a distance bound

Then $\text{TSP} = \{\langle m, D, B \rangle : \exists \text{ tour of all cities of length } \leq B\}$. 
Traveling Salesman Problem: Example

\[ n = 4 \]

\[ B \geq 15 \Rightarrow \langle m, D, B \rangle \in \text{TSP} \]
\[ B \leq 14 \Rightarrow \langle m, D, B \rangle \notin \text{TSP} \]

“tour” = visits every city and returns to starting point

There are many variants of TSP, e.g. require visiting every city exactly once, triangle inequality on distances...
TSP $\in$ NP

Why is TSP $\in$ NP?

Because if $\langle m, D, B \rangle \in$ TSP, the following nondeterministic strategy will accept in time $O(n^3)$, where $n =$ length of representation of $\langle m, D, B \rangle$.

- **nondeterministically** write down a sequence of cities $c_1, \ldots, c_m$, for $m \leq n^2$. (“guess”)

- trace through that circuit and verify that the length is $\leq B$. If so, halt in $q_{\text{accept}}$. If not, halt in $q_{\text{reject}}$. (and “check”)

But any obvious deterministic version of this algorithm takes exponential time.
A useful characterization of NP

- A verifier for a language $L$ is an algorithm $V$ such that

$$L = \{x : V \text{ accepts } \langle x, y \rangle \text{ for some string } y\}.$$

- A polynomial-time verifier is one that runs in time polynomial in $|x|$ on input $\langle x, y \rangle$.

- A string $y$ that makes $V(\langle x, y \rangle)$ accept is a “proof” or “certificate” that $x \in L$.

- Example: TSP

  certificate $y = ?$

  $$V(\langle x, y \rangle) = ?$$

- Without loss of generality, $|y|$ is at most polynomial in $|x|$.
**Theorem:** NP equals the class of languages with polynomial-time verifiers

**Proof:**

$\Rightarrow$

$\Leftarrow$

"$L$ is in NP iff members of $L$ have short, efficiently verifiable certificates"
NP is the class of easily verified languages

**Theorem:** NP equals the class of languages with polynomial-time verifiers

**Proof:**

⇒ The $y$ in $\langle x, y \rangle$ is a record of a computation that accepts $x$.

⇐ Guess $y$ and use the Verifier to check that $V$ accepts $\langle x, y \rangle$

“$L$ is in NP iff members of $L$ have short, efficiently verifiable certificates”
More problems in NP

- **HAMILTONIAN CIRCUIT**

\[ \text{HC} = \{ G : G \text{ is an undirected graph with a circuit that touches each node just once} \}. \]

Really just a special case of TSP. (why?)

- We are not fussy about the precise method of representing a graph as a string, because all reasonable methods are within a polynomial of each other in length.
A “similar” problem that is in P

- **Eulerian Circuit**

\[ EC = \{ G : G \text{ is an undirected graph with a circuit} \]
\[ \text{that passes through each edge exactly once} \}. \]

It is easy to check if \( G \) is Eulerian.

So \( EC \in P \).
Composite Numbers

- \textbf{COMPOSITES} = \{ w : w \text{ is a composite number in binary}\}.

\textbf{COMPOSITES} \in \textbf{NP}

Not obviously in \textbf{P}, since an exhaustive search for factors would take at least proportional to the value of \( w \), which grows as \( 2^n = \text{exponential in the size of } w \).

Only recently (2002), it was shown that \textbf{COMPOSITES} \in \textbf{P} (equivalently, \textbf{PRIMES} \in \textbf{P}).
Boolean logic

- **Boolean formulas**

**Def:** A **Boolean formula** (B.F.) is either:

- a “Boolean variable” \( x, y, z, \ldots \)
- \((\alpha \lor \beta)\) where \(\alpha, \beta\) are B.F.’s.
- \((\alpha \land \beta)\) where \(\alpha, \beta\) are B.F.’s
- \(\neg \alpha\) where \(\alpha\) is a B.F.

**e.g.** \((x \lor y \lor z) \land (\neg x \lor \neg y \lor \neg z)\)

[Omitting redundant parentheses]
Boolean satisfiability

**Def:** A truth-assignment is a mapping $\mathcal{A} : \text{Boolean variables} \rightarrow \{0, 1\}$. [$0 = \text{false}, 1 = \text{true}$]

A T-A is extended to all B.F.’s by the rules:

- $\mathcal{A}(\alpha \lor \beta) = 1$ iff $\mathcal{A}(\alpha) = 1$ or $\mathcal{A}(\beta) = 1$
- $\mathcal{A}(\alpha \land \beta) = 1$ iff $\mathcal{A}(\alpha) = 1$ and $\mathcal{A}(\beta) = 1$
- $\mathcal{A}(-\alpha) = 1$ iff $\mathcal{A}(\alpha) = 0$

$\mathcal{A}$ satisfies $\alpha$ (written $\mathcal{A} \models \alpha$) iff $\mathcal{A}(\alpha) = 1$

In this case, $\alpha$ is **satisfiable**. If no $\mathcal{A}$ satisfies $\alpha$, then $\alpha$ is **unsatisfiable**.

$\text{SAT} = \{\alpha : \alpha \text{ is a satisfiable Boolean formula}\}$.

**Prop:** $\text{SAT} \in \text{NP}$
A “similar” problem in P: 2-SAT

A 2-CNF formula is one that looks like

\[(x \lor y) \land (\neg y \lor z) \land (\neg y \lor \neg x)\]

i.e., a conjunction of **clauses**, each of which is the disjunction of 2 literals (or 1 literal, since \((x) \equiv (x \lor x)\))

2-SAT = the set of satisfiable 2-CNF formulas.

**e.g.** \((x \lor y) \land (\neg x \lor \neg y) \land (\neg x \lor y) \land (x \lor \neg y) \notin \text{SAT}\)
2-SAT $\in P$

**Method** (resolution):

1. If $x$ and $\neg x$ are both clauses, then *not* satisfiable
   
   e.g. $(x) \land (z \lor y) \land (\neg x)$

2. If $(x \lor y) \land (\neg y \lor z)$ are both clauses, add clause $(x \lor z)$ (which is implied).

3. Repeat. If no contradiction emerges $\Rightarrow$ satisfiable.

$O(n^2)$ repetitions of step 2 since only 2 literals/clause.
P vs. NP

- We would like to solve problems in NP efficiently.
- We know $P \subseteq NP$.
- Problems in $P$ can be solved “fairly” quickly.
- What is the relationship between $P$ and $NP$?
NP and Exponential Time

**Claim:** $\text{NP} \subseteq \bigcup_{k} \text{TIME}(2^{n^k})$

Of course, this gets us nowhere near $P$.

Is $P = \text{NP}$?

i.e., do all the NP problems have polynomial time algorithms?

It doesn’t “feel” that way but as of today there is no NP problem that has been **proven** to require exponential time!
The Strange, Strange World if $P = NP$

- Thousands of important languages can be decided in polynomial time, e.g.
  - SATISFIABILITY
  - TRAVELING SALESMAN
  - HAMILTONIAN CIRCUIT
  - MAP COLORING
If $P = NP$, then searching becomes easy

- Every “reasonable” search problem could be solved in polynomial time.
  - “reasonable” $\equiv$ solutions can be recognized in polynomial time (and are of polynomial length)
  - **SAT SEARCH**: Given a satisfiable boolean formula, find a satisfying assignment.
  - **FACTORIZATION**: Given a natural number (in binary), find its prime factorization.
  - **NASH EQUILIBRIUM**: Given a two-player “game”, find a Nash equilibrium.
If $P = NP$, Optimization becomes easy

- Every “reasonable” optimization problem can be solved in polynomial time.
  - Optimization problem $\equiv$ “maximize (or minimize) $f(x)$ subject to certain constraints on $x$”
  - “Reasonable” $\equiv$ “$f$ and constraints are poly-time”
  - MIN-TSP: Given a TSP instance, find the shortest tour.
  - SCHEDULING: Given a list of assembly-line tasks and dependencies, find the maximum-throughput scheduling.
  - PROTEIN FOLDING: Given a protein, find the minimum-energy folding.
  - CIRCUIT MINIMIZATION: Given a digital circuit, find the smallest equivalent circuit.
If $P = NP$, Secure Cryptography becomes impossible

**Cryptography:** Every encryption algorithm can be “broken” in polynomial time.

- “Given an encryption $z$, find the corresponding decryption key $K$ and message $m$” is an NP search problem.
If $P = NP$, Artificial Intelligence becomes easy

- **Artificial Intelligence**: “Learning” is easy.
  - Given many examples of some concept (e.g. pairs (image1, “dog”), (image2, “person”), …), classify new examples correctly.
  - Turns out to be equivalent to finding a short “classification rule” consistent with examples.
If P = NP, Even Mathematics Becomes Easy!

- **Mathematical Proofs**: Can always be found in polynomial time (in their length).
  - **Short Proof**: Given a mathematical statement $S$ and a number $n$ (in unary), decide if $S$ has a proof of length at most $n$ (and, if so, find one).
  - An NP problem!
Gödel’s Letter to von Neumann, 50 years ago

$[\phi(n) = \text{time required for a TM to determine whether a formula has a proof of length } n]$

... If there really were a machine with $\phi(n) \sim k \cdot n$ (or even $\sim k \cdot n^2$) this would have consequences of the greatest importance. Namely, it would obviously mean that in spite of the undecidability of the Entscheidungsproblem, the mental work of a mathematician concerning Yes-or-No questions could be completely replaced by a machine. ...

It would be interesting to know, for instance, the situation concerning the determination of primality of a number and how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search. ...
The World if $P \neq NP$?

- **Q:** If $P \neq NP$, can we conclude anything about any *specific* problems?

- **Idea:** Try to find the “hardest” NP language.
  
  - Just like $A_{TM}$ was the “hardest” Turing-recognizable language.
  
  - Want $L \in NP$ such that $L \in P$ iff *every* NP language is in $P$. 

---

Dr Russ Ross  (Dixie State University)  CS 3530  Fall 2016  56 / 98
Polynomial-time reducibility

- **Def:** \( L_1 \leq_P L_2 \) iff there is a **polynomial-time** computable function \( f : \Sigma_1^* \rightarrow \Sigma_2^* \) such that for every \( x \in \Sigma_1^* \), \( x \in L_1 \) iff \( f(x) \in L_2 \).

- **Proposition:** If \( L_1 \leq_P L_2 \) and \( L_2 \in P \), then \( L_1 \in P \).

- **Proof:**
$L_1 \leq_P L_2$

$x \in L_1 \Rightarrow f(x) \in L_2$

$x \notin L_1 \Rightarrow f(x) \notin L_2$

$f$ computable in polynomial time

$L_2 \in P \Rightarrow L_1 \in P.$
NP-Completeness

- **Def:** $L$ is **NP-complete** iff
  
  1. $L \in \text{NP}$ and
  2. Every language in NP is reducible to $L$ in polynomial time.
    ("$L$ is **NP-hard**")

- **Prop:** Let $L$ be any NP-complete language. Then $P = \text{NP}$ if and only if $L \in P$. 
Cook-Levin Theorem

- Stephen Cook 1971, Leonid Levin 1973
- **Theorem:** SAT (Boolean satisfiability) is NP-complete
- **Proof:** Need to show that *every* language in NP reduces to SAT (!) Proof later.
NP-Completeness

- **Reading**: Sipser §7.4, §7.5.
More NP-complete problems

From now on we prove NP-completeness using:

**Lemma**: If we have the following

- $L$ is in NP
- $L_0 \leq_P L$ for some NP-complete $L_0$

Then $L$ is NP-complete

**Proof:**
More NP-complete problems

From now on we prove NP-completeness using:

Lemma: If we have the following

- $L$ is in NP
- $L_0 \leq_P L$ for some NP-complete $L_0$

Then $L$ is NP-complete

Proof: Since by hypothesis $L \in \text{NP}$, it suffices to show that every $L' \in \text{NP}$ reduces to $L$.

- $L' \leq_P L_0$ since $L_0$ is NP-complete;
- $L_0 \leq_P L$ by hypothesis; and so
- $L' \leq_P L$ by transitivity.

Thus, $L$ is NP-complete.
3-SAT

Def: A Boolean formula is in 3-CNF if it is of the form:

\[ C_1 \land C_2 \land \ldots \land C_n \]

where each clause \( C_i \) is a disjunction ("or") of 3 literals:

\[ C_i = (C_{i1} \lor C_{i2} \lor C_{i3}) \]

where each literal \( C_{ij} \) is either

- a variable \( x \), or
- the negation of a variable, \( \neg x \).

e.g. \((x \lor y \lor z) \land (\neg x \lor \neg u \lor w) \land (u \lor u \lor u)\)

3-SAT is the set of \textbf{satisfiable} 3-CNF formulas.
3-SAT is NP-complete

Proof: Show that SAT $\leq_P$ 3-SAT.

1. Given an arbitrary Boolean formula, e.g.

$$F = \neg(((x \lor \neg y) \land (z \lor w)) \lor \neg x).$$

2. Number the operators.

3. Select a new variable $a_i$ for each operator.
   The variable $a_i$ is supposed to mean “the subformula rooted at operator $i$ is true.”

4. Write a formula stating the relation between each subformula and its children subformulas.
Reduction of SAT to 3-SAT, continued

For example, where

\[ F = \neg((x \lor \neg y) \land (z \lor w)) \lor \neg x, \]

\[ F_1 = \left( \begin{array}{c}
(a_3 \equiv \neg y) \\
\land (a_2 \equiv x \lor a_3) \\
\land (a_5 \equiv z \lor w) \\
\land (a_4 \equiv a_2 \land a_5)
\end{array} \right) \land (a_7 \equiv \neg x) \land (a_1 \equiv \neg a_4) \land (a_6 \equiv a_1 \lor a_7) \]

5. Let \( k \) be the number of the main operator/subformula of \( F \).
   (Note: \( k = 6 \) in the example)
Write $F_1$ in 3-CNFind to obtain $F_2$

- $\alpha \equiv \beta$ is the same as $(\alpha \Rightarrow \beta) \land (\beta \Rightarrow \alpha)$, so to get rid of the $\alpha \equiv \beta$ clauses in $F_1$,

<table>
<thead>
<tr>
<th>Replace</th>
<th>By</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a \equiv \neg b$</td>
<td>$(a \lor a \lor b) \land (\neg a \lor \neg a \lor \neg b)$</td>
</tr>
<tr>
<td>$a \equiv b \lor c$</td>
<td>$(\neg a \lor b \lor c) \land (\neg b \lor \neg b \lor a) \land (\neg c \lor \neg c \lor a)$</td>
</tr>
<tr>
<td>$a \equiv b \land c$</td>
<td>$(\neg a \lor \neg a \lor b) \land (\neg a \lor \neg a \lor c) \land (\neg b \lor \neg c \lor a)$</td>
</tr>
</tbody>
</table>

Output of the reduction: $a_k \land F_2$. 

Dr Russ Ross  (Dixie State University)
Make sure this reduction is polynomial-time

**Q:** Does this prove that every Boolean formula can be converted to 3-CNF?

► Note that transforming boolean formulas to CNF by “multiplying out” does *not* show that the satisfiable CNF formulas are an NP-complete set.

(Much less that 3-SAT is NP-complete.)

Non-polynomial transformation:

\[x_1 y_1 \lor x_2 y_2 \lor \ldots \lor x_n y_n\] (which has size \(O(n)\)) becomes

\[(x_1 \lor \ldots \lor x_n) \land (x_1 \lor \ldots \lor x_{n-1} \lor y_n) \land \ldots \land (y_1 \lor \ldots \lor y_n)\] (size \(O(2^n)\))
**Vertex Cover (VC)**

- **Instance:**
  - a graph
  - a number $k$, (e.g. 4)

- **Question:** Is there a set of $k$ vertices that “cover” the graph, i.e., include at least one endpoint of every edge?
VC is NP-complete

- VC is in NP:

- 3-SAT \( \leq_P \) VC:
  - Let \( F \) be a 3-CNF formula with clauses \( C_1, \ldots, C_m \), variables \( x_1, \ldots, x_n \).
  - We construct a graph \( G_F \) and a number \( N_F \) such that:
    \[
    G_F \text{ has a size } N_F \text{ vertex cover iff } F \text{ is satisfiable}
    \]
Construction of $G_F$ and $N_F$ from $F$

E.g. $F = (x_1 \lor x_2 \lor \neg x_3) \land (\neg x_1 \lor \neg x_2 \lor x_3) \land (x_1 \lor \neg x_2 \lor x_3)$

$G_F = \text{one dumbbell for each variable, one triangle for each clause, and corner } j \text{ of triangle } i \text{ is connected to the vertex representing the } j\text{th literal in } C_i.$

$N_F = 2m + n = 2(\text{# clauses}) + (\text{# variables})$
$\Rightarrow 1 \text{ vertex from each dumbbell and 2 from each triangle.}$
If $F$ is satisfiable then there is an $N_F$ cover

**Proof:** Choose a node from each dumbbell according to the truth-assignment.

Choose the other 2 corners from each triangle.
If $G_F$ has a $N_F$ cover then $F$ is satisfiable

Proof: The cover must include

- At least one vertex from each
- At least two vertices from each

But this totals $N_F$, and so there must be

- Exactly one vertex from each
- Exactly two vertices from each
Getting a satisfying truth-assignment from a vertex cover

The selections from the $x_1 \quad \overline{x_1}$ are a truth-assignment.

Of the 3 “cross-edges” from each triangle:

Exactly 2 are covered by the choices from the triangle, so one must be covered by the choice from the dumbbell:

This means that the truth-assignment from the dumbbell satisfies at least one literal from each clause.
Clique

Instance:

- a graph, e.g.
- a number $k$ (e.g. 4)

Question: Is there a clique of size $k$, i.e., a set of $k$ vertices such that there is an edge between each pair?

Easy to see that $\text{Clique} \in \text{NP}$. 
If $G$ is any graph, let $G^c$ be the graph with the same vertices such that:

there is an edge between $x$ and $y$ in $G^c$

iff

there is no edge between $x$ and $y$ in $G$

e.g.

\[
\begin{align*}
G &= \begin{array}{c}
\text{\includegraphics[width=0.3\textwidth]{graph1.png}}
\end{array} \\
G^c &= \begin{array}{c}
\text{\includegraphics[width=0.3\textwidth]{graph2.png}}
\end{array}
\end{align*}
\]
Let \((G, k)\) be an instance of VC.

**Claim:** \(G\) has a \(k\)-cover iff \(G^c\) has a \(|G| - k\) clique, where \(|G|\) is the number of vertices in \(G\).

(So the mapping \((G, k) \mapsto (G^c, |G| - k)\)
is a reduction of VC to CLIQUE.)

**Proof** (by example):

\[
G = \begin{array}{c}
\text{vertex 1} \\
\text{vertex 2} \\
\text{vertex 3} \\
\text{vertex 4} \\
\end{array}
\begin{array}{c}
\text{vertex 5} \\
\text{vertex 6} \\
\text{vertex 7} \\
\end{array}
\begin{array}{c}
\text{vertex 8} \\
\text{vertex 9} \\
\text{vertex 10} \\
\end{array}
\end{array}
\]

\[
G^c = \begin{array}{c}
\text{vertex 1} \\
\text{vertex 2} \\
\text{vertex 3} \\
\text{vertex 4} \\
\end{array}
\begin{array}{c}
\text{vertex 5} \\
\text{vertex 6} \\
\text{vertex 7} \\
\end{array}
\begin{array}{c}
\text{vertex 8} \\
\text{vertex 9} \\
\text{vertex 10} \\
\end{array}
\end{array}
\]

\[
G = \begin{array}{c}
\text{vertex 1} \\
\text{vertex 2} \\
\text{vertex 3} \\
\text{vertex 4} \\
\end{array}
\begin{array}{c}
\text{vertex 5} \\
\text{vertex 6} \\
\text{vertex 7} \\
\end{array}
\begin{array}{c}
\text{vertex 8} \\
\text{vertex 9} \\
\text{vertex 10} \\
\end{array}
\end{array}
\]

\[
G^c = \begin{array}{c}
\text{vertex 1} \\
\text{vertex 2} \\
\text{vertex 3} \\
\text{vertex 4} \\
\end{array}
\begin{array}{c}
\text{vertex 5} \\
\text{vertex 6} \\
\text{vertex 7} \\
\end{array}
\begin{array}{c}
\text{vertex 8} \\
\text{vertex 9} \\
\text{vertex 10} \\
\end{array}
\end{array}
\]
An integer linear program is

- A set of variables $x_1, \ldots, x_n$ which must take integer values.
- A set of linear inequalities:
  \[ a_{i1}x_1 + a_{i2}x_2 + \ldots + a_{in}x_n \leq c_i \quad [i = 1, \ldots, m] \]

  e.g. $x_1 - 2x_2 + x_4 \leq 7$
  $x_1 \geq 0$ \hspace{1cm} [−$x_1 \leq 0$]
  $x_4 + x_1 \leq 3$

ILP = the set of integer linear programs for which there are values for the variables that simultaneously satisfy all the inequalities.
INTEGER LINEAR PROGRAMMING ∈ NP. (Not obvious! Need a little math to prove it. Proof omitted.)

INTEGER LINEAR PROGRAMMING is NP-hard: by reduction from 3-SAT (3-SAT ≤p ILP). Given 3-CNF Formula $F$, construct following ILP $P$ as follows:
Example of 3-SAT $\leq_p$ INTEGER LINEAR PROGRAMMING

- Suppose
  \[ F = (x \lor y \lor \neg z) \land (\neg x \lor \neg y \lor w) \land (\neg y \lor \neg w \lor \neg z) \land (x \lor w \lor z) \]

  Then ILP $P$ can be:
  \[
  \begin{align*}
  x \geq 0 & \quad x \leq 1 \\
  y \geq 0 & \quad y \leq 1 \\
  x + \neg x & = 1 \\
  y + \neg y & = 1 \\
  x + y + \neg z & \geq 1 \\
  \neg x + \neg y + w & \geq 1 \\
  \neg y + \neg w + \neg z & \geq 1 \\
  x + w + z & \geq 1
  \end{align*}
  \]

  The integer variables are $x, y, z, w, \neg x, \neg y, \neg z, \neg w$.

  $F$ is satisfiable iff
  
  $P$ can be satisfied (with values of variables $\in \{0, 1\}$)

Note: LINEAR PROGRAMMING where the variables can take real values is known to be in $P$. 
More NP-complete/NP-hard Problems

- **HAMILTONIAN CIRCUIT** (and hence TSP) (see Sipser for related problems)
- **SCHEDULING**
- **CIRCUIT MINIMIZATION**
- **SHORT PROOF**
- **NASH EQUILIBRIUM WITH MAXIMUM PAYOFF**
- **PROTEIN FOLDING**
- See Gary & Johnson for hundreds more.
Cook-Levin Theorem and Beyond

Reading: (none)
Cook-Levin Theorem: SAT is NP-complete

Proof:

- Already know SAT ∈ NP, so only need to show SAT is NP-hard.
- Let \( L \) be any language in NP. Let \( M \) be a NTM that decides \( L \) in time \( n^k \).

We define a polynomial-time reduction

\[ f_L : \text{inputs} \mapsto \text{formulas} \]

such that for every \( w \),

\[ M \text{ accepts input } w \text{ iff } f_L(w) \text{ is satisfiable} \]
Reduction via “computation histories”

Proof idea: satisfying assignments of $f_L(w) \leftrightarrow$ accepting computations of $M$ on $w$

Describing computations of $M$ by boolean variables:

- If $n = |w|$, then any computation of $M$ on $w$ has at most $n^k$ configurations.
- Each configuration is an element of $C^{n^k}$, where $C = Q \cup \Gamma \cup \{\#\}$ (mark left and right ends with $\{\#\}$).

$\Rightarrow$ computation depicted by $n^k \times n^k$ “tableau” of members of $C$.

- Represent contents of cell $(i, j)$ by $|C|$ boolean variables $\{x_{i,j,s} : s \in C\}$, where $x_{i,j,s} = 1$ means “cell $(i, j)$ contains $s$”.
- $0 \leq i, j \leq n^k$, so $|C| \cdot n^{2k}$ boolean variables in all
Subformulas that verify the computation

Express conditions for an accepting computation on $w$ by boolean formulas:

- $\phi_{\text{cell}} = \text{“for each } (i, j), \text{ there is exactly one } s \in C \text{ such that } x_{i,j,s} = 1\text{”}$.
- $\phi_{\text{start}} = \text{“first row equals start configuration on } w\text{”}$
- $\phi_{\text{accept}} = \text{“last row is an accept configuration on } w\text{”}$
- $\phi_{\text{move}} = \text{“every } 2 \times 3 \text{ window is consistent with the transition function of } M\text{”}$
Completing the proof

**Claim:** Each of above can be expressed by a formula of size \( O((n^k)^2) = O(n^{2k}) \), and can be constructed in polynomial time from \( w \).

**Claim:** \( M \) has an accepting computation on \( w \) if and only if 
\[ f_L(w) = \phi_{\text{cell}} \land \phi_{\text{start}} \land \phi_{\text{accept}} \land \phi_{\text{move}} \] has a satisfying assignment.

Thus \( w \mapsto f_L(w) \) is a polynomial-time reduction from \( L \) to SAT.

Since above holds for every \( L \in \text{NP} \), SAT is NP-hard, as desired. ■
Glimpses Beyond: co-NP

Recall that co-NP = \{ \overline{L} : L \in \text{NP} \}.

Some co-NP-complete problems:

- Complement of any NP-complete problem.
- \text{TAUTOLOGY} = \{ \varphi : \forall a \varphi(a) = 1 \} (even for 3-DNF formulas \varphi).

Believed that NP \neq \text{co-NP}, \text{P} \neq \text{NP} \cap \text{co-NP}.
“Between” P and NP-complete

**Theorem:** If $P \neq NP$, then there are NP languages that are neither in P nor NP-complete.

Some natural candidates:

- FACTORING (when described as a language)
- Nash Equilibrium
- Graph Isomorphism
- Any problem in $NP \cap co-NP$ for which we don’t know a poly-time algorithm.

Despite decades of effort, we don’t know ANY nontrivial lower bound on the complexity of any NP-complete problem!
The World If $P \neq NP$
The World If $P = NP$
Beyond NP

- “Space” as a resource = number of TM squares used in a computation
- A reasonable proxy for memory on other computational models
- \( \text{PSPACE} = \) languages decidable on TMs using polynomial space
- \( \text{P} \subseteq \text{PSPACE} \) (why?)
- \( \text{NP} \subseteq \text{PSPACE} \) (why?)
- \( \text{PSPACE} \subseteq \text{NPSPACE} \) (why?)
Examples of problems in PSPACE or NPSPACE but probably not in NP

- Determining whether $w \in L(G)$, where $G$ is a context-sensitive grammar, is in NPSPACE
  - Recall that $G$ is a CSG if the right-hand side of every rule is at least as long as its left-hand side
  - I.e. if $u \rightarrow v$ is a rule then $|u| \leq |v|$

- Determining whether a quantified boolean expression is true is in PSPACE
  - E.g. $\forall x \exists y \forall z [(x \land y) \lor (\neg x \land \neg z)]$
Savitch’s Theorem

NPSPACE = PSPACE

- How much time can a computation take if it uses $O(n^k)$ space and does not loop? $O(2^{n^k})$

- To check deterministically if there exists a computation from TM configuration $C_1$ to TM configuration $C_2$ in $T$ steps,
  - for all configurations $C$, check if
    - there is a computation from $C_1$ to $C$ of $T/2$ steps and
    - there is a computation from $C$ to $C_2$ in $T/2$ steps

- To check whether the initial configuration yields the final configuration takes $O(\log(2^{n^k})) = O(n^k)$ recursion levels and $O(n^k)$ space at each level $\Rightarrow O(n^{2k})$ space
Conclusions

Reading: (none)
Reprise: Models of computation and formal systems

- DFAs, NFAs, REs, CFGs, PDAs, TMs, NTMs, ...
- How to formally **model computation**
- **Asymptotic perspective** (fixed program for all input lengths)
- Design your own models as circumstances demand (e.g. interactive/distributed computation, randomized computation, biological systems, economic systems)
Classification of computational problems

- **Positive results:** regular, context-free, polynomial-time, decidable, Turing-recognizable languages
- **Negative results:** non-regular, non-CF, NP-complete(?), undecidable, non-recognizable languages
- Notion of reduction between problems
- The systematic methodology for proving things impossible is one of the most important achievements of computer science
- NP-completeness is one of the most important “exports” of computer science to the rest of science
Understanding Intractability

▶ Many important problems are NP-complete (or even undecidable)
▶ But also some great positive results in algorithms design
  ▶ E.g. poly-time algorithms for Linear Programming, Primality Testing, Polynomial Factorization, Network Flows, ...
▶ What does NP-completeness mean? (assuming P ≠ NP)
  ▶ No algorithm can be guaranteed to solve the problem perfectly in polynomial time on all instances
  ▶ Exhaustive search is often unavoidable
  ▶ Mathematical nastiness: no nice, closed form solutions
Coping with Intractability

What if you need to solve an NP-complete (or undecidable) problem?

- Ask your boss for a new assignment. :-)
- Simplify the problem, you may not need to solve it in full generality
- Identify additional constraints that make the problem easier (e.g. bounded-degree graphs, ILP with fixed number of variables, 2-SAT)
- Approximation algorithms, e.g., find a TSP tour of length at most 1.01 times the shortest
- Average-case analysis—analyse running time or correctness on “random” inputs. (Often hard to find distribution that models “real-life” inputs well.)
More attacks on intractable problems

- **Heuristics**—techniques that seem to work well in practice but do not have rigorous performance guarantees.

- **Change the problem**
  - Instead of verifying that general programs satisfy desired security properties (undecidable), ask programmers to supply programs with (easily verifiable) “proofs” that the properties are satisfied.
  - Change the programming language