Suppose you are organizing housing accommodations for a group of four hundred university students. Space is limited and only one hundred of the students will receive places in the dormitory. To complicate matters, the Dean has provided you with a list of pairs of incompatible students, and requested that no pair from this list appear in your final choice. This is an example of what computer scientists call an NP problem, since it is easy to check if a given choice of one hundred students proposed by a coworker is satisfactory (i.e., no pair taken from your coworker's list also appears on the list from the Dean's office), however the task of generating such a list from scratch seems to be so hard as to be completely impractical. Indeed, the total number of ways of choosing one hundred students from the four hundred applicants is greater than the number of atoms in the known universe! Thus no future civilization could ever hope to build a supercomputer capable of solving the problem by brute force; that is, by checking every possible combination of 100 students. However, this apparent difficulty may only reflect the lack of ingenuity of your programmer. In fact, one of the outstanding problems in computer science is determining whether questions exist whose answer can be quickly checked, but which require an impossibly long time to solve by any direct procedure. Problems like the one listed above certainly seem to be of this kind, but so far no one has managed to prove that any of them really are so hard as they appear, i.e., that there really is no feasible way to generate an answer with the help of a computer. Stephen Cook and Leonid Levin formulated the P (i.e., easy to find) versus NP (i.e., easy to check) problem independently in 1971.

If one of them is solved in the next few years, it'll probably be P vs. NP.

If, in the year 3000, exactly one of them is unsolved, it'll unquestionably be P vs. NP.

Why did Lovász say that?
Millennium Prize Problems

1. Birch and Swinnerton-Dyer Conjecture
2. Hodge Conjecture
3. Existence & smoothness for Navier-Stokes
4. Poincaré Conjecture
5. **P vs. NP**
6. Riemann Hypothesis
7. Yang-Mills existence and mass gap

What is the **P vs. NP** problem?

Solved in 2003 by Grisha Perelman.

**Sudoku**

3×3 × 3×3 Sudoku

4×4 × 4×4 Sudoku

4×4 × 4×4 Sudoku
No-Promises Sudoku

Given a partially filled $n \times n \times n \times n$ Sudoku grid, output YES or NO: can it be validly completed?

Naive decision algorithm:
For each empty cell ($\leq n^4$), try each possible digit.
Check if that's a valid solution. Overall time $\approx n^{4n}$.

Smart decision algorithm: ???

Verifying a proposed solution: Time $O(n^4)$.

$n \times n \times n \times n$ No-Promises Sudoku

Naive decision algorithm: Time $= n^{4n}$.
Verifying a proposed solution: Time $O(n^4)$.

For $n = 100$ (meaning $10,000$ $100 \times 100$ grids):
Verifying a solution: $\approx 100M$ steps.
Your cell phone can do this in 1 second.
Naive algorithm: a number with $\approx 200M$ digits.
Insanely larger than # of quarks in the universe.

$n \times n \times n \times n$ No-Promises Sudoku

Question:
Is there a fixed constant $c$ and an algorithm $A$ such that $A$ solves the decision problem in time $O(n^c)$?

This is equivalent to the P vs. NP problem!

$n \times n \times n \times n$ No-Promises Sudoku

Is this famous $1,000,000$ problem really about Sudoku?? Yes and no.

Here's how P vs. NP is usually (informally) stated:

Let $L$ be an algorithmic task.
Suppose there is an efficient algorithm for verifying solutions to $L$, $\text{"LeNP"}$
Is there always also an efficient algorithm for finding solutions to $L$, $\text{"LeP"}$
Isn’t Sudoku just one particular instance of this question?
We’ll see: It’s true for all problems if and only if it is true for Sudoku!

Let’s develop these notions formally...
We’ll start by describing some sample algorithmic problems.

Let L be an algorithmic task.
Suppose there is an efficient algorithm for verifying solutions to L. “$L \in NP$”
Is there always also an efficient algorithm for finding solutions to L? “$L \in P$”

We’ll see: It’s true for all problems if and only if it is true for Sudoku!

Let’s develop these notions formally...
We’ll start by describing some sample algorithmic problems.

Let L be an algorithmic task.
Suppose there is an efficient algorithm for verifying solutions to L. “$L \in NP$”
Is there always also an efficient algorithm for finding solutions to L? “$L \in P$”

3-Coloring
Input: A graph
Task: Decide if there is a 3-coloring. If so, find one.

Circuit-Sat
Input: A boolean circuit C
Task: Decide if there is a 0/1 setting to the input wires which “satisfies” C (makes output wire 1). If so, find such a setting.

Hamiltonian Cycle
Input: A graph
Task: Decide if there is a Hamiltonian Cycle in it, meaning a cycle that visits each vertex exactly once. If so, find one.

Bipartite Perfect Matching
Input: A bipartite graph
Task: Decide if there is a perfect matching. If so, find one.
A partially filled $n^2 \times n^2$ Sudoku grid

Task: Decide if there is a valid Sudoku completion. If so, find one.

Decision vs. Search
Each of these problems was of the form, “Does a solution exist? If so, find one.”

Decision problem
Search problem

For simplicity, we focus on decision problems.

(Given a decision algorithm, it’s usually easy to use it to solve the search problem. We saw this for 3-coloring in last lecture)

Reducing search to decision

Example: Circuit-Sat

Suppose you have a good decision alg. for Circuit-Sat.

How can you get a good alg. for solving the search problem?

Hint:
Try fixing $x_1$ to 0, fixing $x_1$ to 1, and running the decision alg. in both cases.

Decision problems as languages

Decision problems

- Given $G$, does it have a Hamiltonian cycle?
- Given bipartite $G$, does it have a perfect matching?
- Given circuit $C$, does it have a “satisfying” input string?
- Given graph $G$, is it 3-colorable?
- Given partially filled Sudoku grid $S$, can it be completed?
- Given TM $M$ and input $x$, does $M(x)$ halt?

Languages in $\{0,1\}^*$

- HAM-CYCLE = $\{(G) : G$ contains a Hamiltonian cycle$\}$
- PMATCH = $\{(G) : G$ is bipartite, has perfect matching$\}$
- CIRCUIT-SAT = $\{(C) : C$ has a satisfying input$\}$
- 3-COL = $\{(G) : G$ is 3-colorable$\}$
- SUDOKU = $\{(S) : S$ can be validly completed$\}$
- HALTS = $\{(M,x) : M(x)$ halts$\}$

Decision problems as languages

Is there a TM (or Java algorithm) which decides the others?

Of course!

No Turing Machine (or Java algorithm) can decide this one.

Efficiency

HAM, PMATCH, CIRCUIT-SAT, 3-COL, SUDOKU can all be decided by “trying all possibilities.”

E.g., there is a naive algorithm for deciding 3-COL which runs in $3^{\Omega(n)}$ time.

We care about more than just “Is there an algorithm?”

We care about “Is there a reasonably ‘efficient’ algorithm?”
What is ‘efficient’?

Is your algorithm for deciding L ‘efficient’ if on input strings of length $n$ it runs in time...

- $O(n)$? Sure (unless the constant is huge...)
- $O(n \log n)$? Sure.
- $O(n^2)$? Kind of efficient.
- $O(n^{log n})$? Barely...
- $O(2^n)$? Not really...
- $O(n!)$? Please. Internet ≈ 22! bytes.

**Polynomial time**

Polynomial time is the standard ‘theoretical’ definition of ‘efficient’.

It is a very “low bar” for efficiency:
if it’s not poly-time, it’s really not efficient.

Yes, yes, yes, an algorithm running in time $O(n^{100})$ is not actually efficient in practice.

It’s a low bar: a polynomial time solution is a **necessary** first step towards a truly efficient one.

**Examples**

$CONN = \{ \langle G \rangle : G$ is a connected graph $\} \in \mathbf{P}.$

**Why?**

Given graph $G$ with $n$ nodes, can correctly decide connectivity by doing breadth-first-search, counting the number of nodes seen, checking if the count equals $n.$

Running time is $O(|V| + |E|) = O(n^2)$ in most reasonable models (maybe $O(n^4)$ on a poor Turing Machine).

(Input size $|\langle G \rangle|$ is $\geq n$ for most reasonable encodings.)
Examples

$\text{CONN} = \{ (G) : G \text{ is a connected graph} \} \in \mathcal{P}$.

$\text{PMATCH} = \{ (G) : G \text{ is a bipartite graph with a perfect matching} \} \in \mathcal{P}$.

**Why?**

We described an $O(n^3)$ time algorithm in Lecture 12 (Graphs II).

Examples

Let $\text{SAME-REG} = \{ (R_1, R_2) :$

$R_1, R_2$ are reg. exprs.

using $\cup$, squaring,

such that $L(R_1) = L(R_2)$

$\langle a(a\cup b)^2, \text{aaaUaabUabaUabb} \rangle \in \text{SAME-REG}$

$\langle a^2(a\cup b), \text{aaaUabb} \rangle \notin \text{SAME-REG}$

Theorem (Meyer–Stockmeyer 1972):

$\text{SAME-REG} \notin \mathcal{P}$

Verifying solutions

$\text{SUDOKU}$: Filling in the grid may be tough, but if someone gives you a solution, **verifying** it is easy (poly-time).

$3\text{-COL, CIRCUIT-SAT, HAM-CYCLE}$: similarly easy to **verify** solutions.

$\text{PMATCH}$: similarly easy to **verify** a solution.

Examples

$\text{CONN} = \{ (G) : G \text{ is a connected graph} \} \in \mathcal{P}$.

$\text{PMATCH} = \{ (G) : G \text{ is bip., has perf. matching} \} \in \mathcal{P}$.

$2\text{-COL} \in \mathcal{P}$.

$3\text{-COL}$: Probably not in $\mathcal{P}$, but no one knows.

$\text{CIRCUIT-SAT, HAM-CYCLE, SUDOKU}$: also unknown if they are in $\mathcal{P}$.

**NP**: poly-time verifiability

Informally, $\mathcal{NP}$ is the set of all languages $L$ such that there is a poly-time algorithm $V$ which can **verify** that $x \in L$ if it is (magically) given a valid certificate (aka proof, witness) that $x \in L$.

Remark: The ‘$N$’ in $\mathcal{NP}$ stands for ‘nondeterministic’. It does not stand for not !!

Reason for terminology is that $\mathcal{NP}$ can also be defined as languages decided by a “nondeterministic” version of poly-time TMs (similar to NFAs).
NP: formal definition

Let $L$ be a language. We say $L \in \text{NP}$ iff...

There are constants $c, d$ and an algorithm $V$ called the "verifier" such that:

$V$ takes two inputs, $x$ and $y$, where $|y| \leq O(|x|^c)$.
$x$ is called the "real input"; $y$ is called the "certificate".
$V(x,y)$ runs in time $O((|x|+|y|)^d)$.

$\forall x \in L$, $\exists y$ such that $V(x,y)$ outputs YES,
$\forall x \not\in L$, $\forall y$, $V(x,y)$ outputs NO.

Examples

SUDOKU $\in \text{NP}$. Why?

The verifying algorithm $V$ takes as input:
- $x$: a partially filled $n^2 \times n^2$ Sudoku grid;
- $y$: supposed to be a valid completion of $x$.

Note that the "certificate" $y$ satisfies $|y| \leq O(|x|)$.
Now $V$ just checks two things:
- on all non-blank cells in $x$, same value appears in $y$;
- $y$ is a valid Sudoku solution.

$V$ runs in polynomial time: in fact, $O(|x|+|y|)$ time.

Examples

3-COL $\in \text{NP}$. Why?

Briefly:

The verifying algorithm takes graph $x$ and expects $y$ to be a valid 3-coloring.

In polynomial time, can check that $y$ is indeed a valid 3-coloring of $x$.

REMINDER: Verifier $V$ does not need to find the certificate.

Examples

HAM-CYCLE, CIRCUIT-SAT $\in \text{NP}$. (Why?)

Is 3-COL $= \{G : G$ is NOT 3-colorable$\}$ in $\text{NP}$?

Informally, is there an easy-to-check certificate that a graph is NOT 3-colorable?

Probably not, but no one knows.

HAM-CYCLE, CIRCUIT-SAT, SUDOKU; not known if in $\text{NP}$.

Examples

PMATCH $\in \text{NP}$.

One reason:

Verifying a given perfect matching is easy.

Another reason:

Because PMATCH $\in \text{P}$!

Fact: $\text{P} \subseteq \text{NP}$. 

Examples

HAM-CYCLE, CIRCUIT-SAT $\in \text{NP}$.
**P ⊆ NP**

Proof:
Suppose L \(\in\) P.
Let A be a poly-time alg. which decides L.
Let V be the following verifier algorithm:
V takes as input:
real input x, “certificate” y of length 0.
V(x,y) just runs A(x) and gives its output.

“Verifier doesn’t need a certificate: it can check membership in L itself.”

Proofs that a language is NP are almost quite easy. But...

Let \(\text{PMATCH} = \{(G) : G \text{ is bipartite, does NOT have a perfect matching} \}\).

Is \(\text{PMATCH} \in \text{NP}\)?

Yes! Clearly \(\text{PMATCH} \in \text{P}\) because \(\text{PMATCH} \in \text{P}\).
(Just run the \(\text{PMATCH}\) algorithm, reverse the answer.)

\(\therefore\) \(\text{PMATCH} \in \text{NP}\) because \(P \subseteq \text{NP}\).

---

**The P vs. NP problem**

We know that \(P \subseteq \text{NP}\).

*Does P = NP?*

If \(P = \text{NP}\) then there is an efficient (polynomial-time) algorithm for SUDOKU, 3-COL, CIRCUIT-SAT, HAM-CYCLE, ...
That would be awesome!!

---

**The P vs. NP problem**

We know that \(P \subseteq \text{NP}\).

*Does P = NP?*

If \(P \neq \text{NP}\) then...
There is *some particular* \(L \in \text{NP}\) which is not in \(P\).

Doesn’t sound like a big deal.
Maybe it’s just some uninteresting, obscure \(L\).

---

**Cook–Levin Theorem**

\(P = \text{NP}\) if and only if 3-SAT \(\in\) P

In particular, if \(P \neq \text{NP}\) then 3-SAT \(\not\in\) P.

“3-SAT is the hardest problem in \(\text{NP}\)”

**The hardest problem(s) in NP**

Last lecture: There is a polynomial-time reduction from CIRCUIT-SAT to 3SAT (and vice versa).

\(\therefore\) Thus 3-SAT \(\in\) P if and only if CIRCUIT-SAT \(\in\) P.

So Cook-Levin Theorem is:

\(P = \text{NP}\) if and only if CIRCUIT-SAT \(\in\) P
Cook–Levin Theorem

\[ P = \text{NP} \text{ if and only if } \text{CIRCUIT-SAT} \in \text{P} \]

In particular, if \( P \neq \text{NP} \) then \( \text{CIRCUIT-SAT} \notin \text{P} \).

“\text{CIRCUIT-SAT} is the hardest problem in \( \text{NP} \)”

The hardest problem(s) in \( \text{NP} \)

\[ P = \text{NP} \text{ if and only if } \text{CIRCUIT-SAT} \in \text{P} \]

If \( \text{CIRCUIT-SAT} \) is in \( P \), then all of \( \text{NP} \) is in \( P \).

Last lecture: There is a polynomial-time reduction from \( \text{CIRCUIT-SAT} \) to \( 3\text{-COL} \).

\[ \therefore \text{If } 3\text{-COL} \in \text{P} \text{ then } \text{CIRCUIT-SAT} \in \text{P} \]

And hence all of \( \text{NP} \) is in \( P \).

\[ \therefore P = \text{NP} \text{ if and only if } 3\text{-COL} \in \text{P} \]

The hardest problem(s) in \( \text{NP} \)

\[ P = \text{NP} \text{ if and only if } 3\text{-COL} \in \text{P} \]

Fact (Yato–Seta 2002): There's a poly-time reduction from \( 3\text{-COL} \) to \( \text{SUDOKU} \).

\[ \therefore \text{If } \text{SUDOKU} \in \text{P} \text{ then } 3\text{-COL} \in \text{P} \]

And hence all of \( \text{NP} \) is in \( P \).

\[ \therefore P = \text{NP} \text{ if and only if } \text{SUDOKU} \in \text{P} \]

\( n \times n \times n \times n \) No-Promises Sudoku

Question:

Is there a fixed constant \( c \) and an algorithm \( A \) such that \( A \) solves the decision problem in time \( O(n^c) \)?

This is equivalent to the \( P \) vs. \( \text{NP} \) problem!

Definitions:

**Definition 1:** Language \( A \) has a polynomial-time oracle reduction (also called Cook reduction) to language \( B \):

- input \( x \)
- Poly-time algorithm deciding \( A \)
- output YES/NO

\[ y \in B? \]

- YES/NO

- \( z \in B? \)

- YES/NO

- \( w \in B? \)

- YES/NO

Also called Cook reduction

**Definition 2:** Language \( A \) has a polynomial-time mapping reduction (also called Karp reduction) to language \( B \):

- input \( x \)
- Poly-time reduction algorithm
- outputs \( z \)

\( x \in A \) if and only if \( z \in B \).

Also called Karp reduction

We prefer this kind.
**Reductions**

Language A has a polynomial-time mapping reduction to language B (denoted $A \leq_m B$):

... means there is a poly-time computable function $f$ such that $x \in A$ if and only if $f(x) \in B$.

Reductions from last lecture (3-COL to/from CIRCUIT-SAT, INDEP-SET to/from CLIQUE) were mapping reductions.

Fact: If A has a (mapping) reduction to B, and $B \in P$, then $A \in P$.

### Cook–Levin Theorem revisited

Actual theorem statement:

Let $L$ be any language in $NP$.
Then there is a poly-time mapping reduction from $L$ to CIRCUIT-SAT.

\[ \therefore \text{CIRCUIT-SAT} \in P \Rightarrow \text{NP} \subseteq P \Rightarrow \text{NP} = P. \]

And $\text{NP} = P \Rightarrow \text{CIRCUIT-SAT} \in P$ because $\text{CIRCUIT-SAT} \in NP$.

**NP-completeness**

Definition:

A language L is $NP$-hard if every language in $NP$ has a mapping reduction to L.

Note: Cook–Levin $\Rightarrow$ “CIRCUIT-SAT is $NP$-hard”.

Definition: A language L is $NP$-complete if:

a) $L \in NP$-hard; and
b) $L \in NP$.

$NP$-complete = “hardest problem in $NP$”.

E.g.: CIRCUIT-SAT.

### IMPORTANT: Recipe for $NP$-completeness

To prove a decision problem (language) is $NP$-complete:

Step 1: Prove it is in $NP$.

Step 2: Prove that some known $NP$-complete language mapping reduces to it.

Be sure the reduction goes in the right direction!
To show B is hard, mapping reduce some other hard problem A to it, i.e., $A \leq_m B$.

Remember: reducing B to a hard problem does not show that B is hard. (Eg, 2-COLOR reduces to HALT)
NP-completeness via reductions

Each of these is a “hardest problem in NP”.

Either ALL of them are in P,
or NONE OF THEM is in P.

How many algorithm problems have been proven to be NP-complete?

My guess is that 10,000 is probably the right order of magnitude.
Problems in every branch of science.

Remember: if even a single one of them is shown to be in P, then all of them are in P!

The fact that this hasn’t happened is the reason 99.9% of people believe \( P \neq NP \).
Proving Cook-Levin theorem

- Recall: we want to reduce an arbitrary language \( A \in \text{NP} \) to Circuit-SAT
- What do we know about \( A \) due to its being in NP?
  - Membership in \( A \) has a poly-time verifier \( V \)
  - \( x \in A \iff \exists y, |y| \leq |x|^c \) such that \( V(x,y) = \text{YES} \)
- Main idea: For a fixed \( x \), can build a circuit \( C_x \), of polynomial size that simulates \( V \) with first input hardwired to \( x \):
  - \( C_x(y) = V(x,y) \)
  - Telling if \( x \in A \) amounts to telling if \( C_x \) is satisfiable
- \( x \rightarrow C_x \) is a poly-time mapping reduction from \( A \) to Circuit-SAT.

Definitions:
- Decision/search problems
- \( \text{P, NP, NP-hard, NP-complete} \)
- Poly-time (mapping) reduction

Theorems:
- Cook-Levin Theorem

How-to:
- Prove languages in \( \text{P} \)
- Prove languages in \( \text{NP} \)
- Show NP-completeness
- Prove languages \( \text{NP-hard} \) by reduction