Suppose that 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

3x3 x 3x3 Sudoku

4x4 x 4x4 Sudoku
No-Promises Sudoku

5 × 5

This one has no solution.

4 × 4 × 4

This one has multiple solutions.

n × n × n × n

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

Naive decision algorithm:
For each empty cell (≤ n^4), try each possible digit.
Check if that's a valid solution. Overall time = n^4.

Smart decision algorithm: ???

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

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

For n = 100 (meaning 10,000 100 × 100 grids):
Verifying a solution: ≈ 100M steps.
Your cell phone can do this in 1 second.

Naive algorithm: a number with ≈ 200M digits.
Insanely larger than # of quarks in the universe.

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!

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. “LENP”
Is there always also an efficient algorithm for finding solutions to L? “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 L be an algorithmic task.
Suppose there is an efficient algorithm for verifying solutions to L. “LENP”
Is there always also an efficient algorithm for finding solutions to L? “LEP”

Let’s develop these notions formally...

We’ll start by describing some sample algorithmic problems.

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.
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)

Decision problems as languages

<table>
<thead>
<tr>
<th>Decision problems</th>
<th>Languages in ${0,1}^*$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Given $G$, does it have a Hamiltonian cycle?</td>
<td>$\text{HAM-CYCLE} = \langle G \rangle : G$ contains a Hamiltonian cycle</td>
</tr>
<tr>
<td>Given bipartite $G$, does it have a perfect matching?</td>
<td>$\text{PMATCH} = \langle G \rangle : G$ is bipartite, has perfect matching</td>
</tr>
<tr>
<td>Given circuit $C$, does it have a “satisfying” input string?</td>
<td>$\text{CIRCUIT-SAT} = \langle C \rangle : C$ has a satisfying input</td>
</tr>
<tr>
<td>Given graph $G$, is it 3-colorable?</td>
<td>$\text{3-COL} = \langle G \rangle : G$ is 3-colorable</td>
</tr>
<tr>
<td>Given partially filled Sudoku grid $S$, can it be completed?</td>
<td>$\text{SUDOKU} = \langle S \rangle : S$ can be validly completed</td>
</tr>
<tr>
<td>Given TM $M$ and input $x$, does $M(x)$ halt?</td>
<td>$\text{HALTS} = \langle M, x \rangle : M(x)$ halts</td>
</tr>
</tbody>
</table>

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 $x_2$ to 1, and running the decision alg. in both cases.

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 $3^{|V|}$ time.

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

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

$\text{HAM} \times \text{PMATCH} \times \text{CIRCUIT-SAT} \times \text{3-COL} \times \text{SUDOKU}$

$\langle G, x \rangle$: $G$ contains a Hamiltonian cycle

$\langle G \rangle$: $G$ is bipartite, has perfect matching

$\langle C \rangle$: $C$ has a satisfying input

$\langle S \rangle$: $S$ can be validly completed

$\langle M, x \rangle$: $M(x)$ halts

$\langle G \rangle$: $G$ is 3-colorable

$\langle M \rangle$: $M(x)$ halts
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.
- $O(n^{log_2{n}})$? Barely...
- $O(2^n)$? Not really...
- $O(n!)$? Please. Internet $\approx 22!$ 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 = $\{(G) : G$ is a connected graph $\}$ \(\in P\).

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

CONN = \{ (G) : G is a connected graph \} ∈ P.

PMATCH = \{ (G) : G is a bipartite graph with a perfect matching \} ∈ P.

Why?

We sort of described an \(O(n^3)\) time algorithm in the Graphs II lecture.

Examples

Let \(\text{SAME-REG} = \{ (R_1, R_2) : R_1, R_2\text{ are reg. exprs. using } \cup, \cdot, \text{squaring, such that } L(R_1) = L(R_2) \} \)

\( ( a(aUb)^2, aaaUaabaUaabb ) ∈ \text{SAME-REG} \)

\( ( a^2(aUb), aaaUabb ) ∉ \text{SAME-REG} \)

Theorem (Meyer–Stockmeyer 1972):

\(\text{SAME-REG} ∈ P\)

Examples

CONN = \{ (G) : G is a connected graph \} ∈ P.

PMATCH = \{ (G) : G is bip., has perf. matching \} ∈ P.

2-COL ∈ P.

3-COL: Probably not in \(P\), but no one knows.

CIRCUIT-SAT, HAM-CYCLE, SUDOKU: also unknown if they are in \(P\).

So we understand \(P\).

Great, we’re halfway there!

Now what is \(NP\)?

NP: poly-time verifiability

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

Remark: The ‘N’ in \(NP\) stands for ‘nondeterministic’. It does not stand for not!!

Reason for terminology is that \(NP\) can also be defined as languages decided by a “nondeterministic” version of poly-time TMs (similar to NFAs)

Verifying solutions

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

3-COL, CIRCUIT-SAT, HAM-CYCLE: similarly easy to verify solutions.

PMATCH: similarly easy to verify a solution.
**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,

$\forall x \in L$, $\forall y$, $V(x,y)$ 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 $\text{3-COL} = \{(G) : G \text{ is NOT 3-colorable}\}$ 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:** $\text{P} \subseteq \text{NP}$.
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.”

**The P vs. NP problem**

We know that \( P \subseteq NP \).

**Does \( P = NP \)?**

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

**Cook–Levin Theorem**

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

In particular, if \( P \neq NP \) then \( 3 \text{- SAT} \notin P \).
“3-SAT is the hardest problem in NP”

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

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

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

So Cook-Levin Theorem is:

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

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

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

"CIRCUIT-SAT is the hardest problem in \( NP \)"

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

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

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

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

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

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

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

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

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

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

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

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

No-Promises Sudoku

\[ n \times n \times n \times n \]

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!

Recap: Karp Reductions

Language \( A \) has a polynomial-time mapping reduction (aka Karp reduction) to language \( B \) (denoted \( A \leq_P^m B \)):

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

Last lecture we saw Karp reductions (3-COL to/from CIRCUIT-SAT, INDEP-SET to/from CLIQUE)

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

Poll

Which of the following are true?

- 3COLOR \( \leq_P^m \) 2COLOR is known to be true.
- 3COLOR \( \leq_P^m \) 2COLOR is known to be false.
- 3COLOR \( \leq_P^m \) 2COLOR is open.
- 2COLOR \( \leq_P^m \) 3COLOR is known to be true.
- 2COLOR \( \leq_P^m \) 3COLOR is known to be false.
- 2COLOR \( \leq_P^m \) 3COLOR is open.
- if \( \Lambda \leq_P^m B \) and \( B \notin NP \), then \( \Lambda \notin NP \).
**Cook–Levin Theorem revisited**

Actual theorem statement:

Let $L$ be any language in $\textbf{NP}$.
Then there is a poly-time mapping reduction from $L$ to $\textsc{Circuit-Sat}$.

$\therefore \text{Circuit-Sat} \in \textbf{P} \Rightarrow \textbf{NP} \subseteq \textbf{P} \Rightarrow \textbf{NP} = \textbf{P}$.

And $\textbf{NP} = \textbf{P} \Rightarrow \text{Circuit-Sat} \in \textbf{P}$

because $\text{Circuit-Sat} \in \textbf{NP}$.

$\textbf{P} = \textbf{NP}$ if and only if $\text{Circuit-Sat} \in \textbf{P}$

**NP-completeness**

Definition:

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

Note: Cook–Levin $\Rightarrow$ “$\text{Circuit-Sat}$ is $\textbf{NP}$-hard”.

Definition: A language $L$ is $\textbf{NP}$-complete if:

a) $L$ is $\textbf{NP}$-hard; and b) $L \in \textbf{NP}$.

$\textbf{NP}$-complete $= \text{"hardest problem in $\textbf{NP}$"}$. E.g.: $\text{Circuit-Sat}$.

**IMPORANT: Recipe for $\textbf{NP}$-completeness**

To prove a decision problem (language) is $\textbf{NP}$-complete:

Step 1: Prove it is in $\textbf{NP}$.
Step 2: Prove that some known $\textbf{NP}$-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**

All languages in $\textbf{NP}$

Cook–Levin

- Circuit-Sat
- 3-COL
- 3-SAT
- Sudoku
- Independ-Set
- Ham-Cycle
NP-completeness via reductions

- CIRCUIT-SAT ➔ 3-SAT
- 3-COL ➔ INDEP-SET ➔ SUDOKU ➔ HAM-CYCLE

Each of these is a "hardest problem in NP".

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

Many more important algorithmic problems have been proven NP-complete:

- Finding optimal schedules
- Packing objects into bins optimally
- Traveling Salesperson Problem
- Allocating variables to registers optimally
- Laying out circuits optimally
- ....

How many algorithms 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≠NP.

Here are some random problems also known to be NP-complete:

Given a, b, c: is there 0 ≤ x ≤ c such that x² = a (mod b)?

(Oct 2002): Given a sequence of Tetris pieces and a number k, can you clear ≥ k lines?

(Nov. 2011): Given a stack of pancakes and a number k, can you sort the stack using ≤ k flips?

(March 2012): Given a Super Mario Bros. level, is it completable?

Proving Cook-Levin theorem

- Recall: we want to reduce an arbitrary language A ∈ 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 ∈ A ⇔ ∃ y, |y| ≤ |x|⁴ such that V(x,y)=YES
  - Main idea: For a fixed x, can build a circuit C_x of polynomial size that simulates V with first input hardened to x:
  - C_x(y) = V(x,y)
  - Telling if x ∈ A amounts to telling if C_x is satisfiable
  - x → C_x is a poly-time mapping reduction from A to Circuit-SAT.
Appendix: $3SAT \leq^P_m CLIQUE$

1. Define a map $f: \Sigma^* \rightarrow \Sigma^*$

not valid encoding of a 3SAT formula $\mapsto \epsilon$

otherwise we have valid 3SAT formula $\varphi$ (with $m$ clauses).

$\varphi \mapsto \{G, k\}$ (we set $k = m$)

Construction demonstrated with an example.

$3SAT \leq^P_m CLIQUE$: Defining the map

$\varphi = (x_1 \lor \neg x_2 \lor x_3) \land (\neg x_1 \lor x_2 \lor x_3) \land (x_1 \lor x_1 \lor \neg x_1)$

The construction:

- A vertex for each literal in each clause.
- No edges between two literals in the same clause.
- No edges between $x_i$ and $\neg x_i$ for any $i$.
- All other possible edges present.
- Set $k$ to be # clauses in $\varphi$.

$3SAT \leq^P_m CLIQUE$: Why it works

If $\varphi$ is satisfiable, then $G_\varphi$ contains an m-clique:

$\varphi$ is satisfiable

$\implies$

can pick $m$ literals, one from each clause, such that we don’t pick a variable and its negation.

$\implies$

by construction of $G_\varphi$, vertices corresponding to those literals are all connected (by an edge).

$\implies$

$G_\varphi$ contains an m-clique.

$3SAT \leq^P_m CLIQUE$: Why it works

If $G_\varphi$ contains an m-clique, then $\varphi$ is satisfiable:

$G_\varphi$ has a clique $K$ of size $m$

$\implies$

by construction of $G_\varphi$:

- $K$ must contain exactly one literal from each clause.
- $K$ cannot contain a variable and its negation.

$\implies$

$\varphi$ is satisfiable.

$3SAT \leq^P_m CLIQUE$: Poly-time reduction?

Creation of $G_\varphi$ is poly-time:

Creating the vertex set:
- there is just one vertex for each literal in each clause
- scan input formula and create the vertex set.

Creating the edge set:
- there are at most $O(m^2)$ possible edges.
- scan input formula to determine if an edge should be present.
Definitions:
- Decision/search problems
- \( P, \) \( NP, \) NP-hard, NP-complete
- Polytime mapping (Karp) reduction

Theorems:
- Cook–Levin Theorem

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