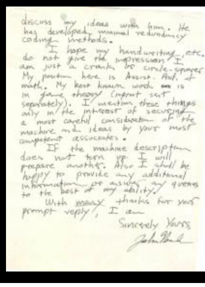
15-251: Great Theoretical Ideas in Computer Science

Lecture 24

November 20, 2014

P vs. NP



$1,000,000

— the prize for solving any of the *Millennium Prize Problems*

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

---

www.claymath.org/millennium/P_vs_NP/

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**.

Keith Devlin in his book "The Millennium Problems: The Seven Greatest Unsolved Mathematical Puzzles of Our Time"

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

The only one with philosophical & metamathematical implications.

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

Solved in 2003
by Grisha Perelman.

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

Sudoku

2			3		8		5	
		3		4	5	9	8	
		8			9	7	3	4
6		7		9				
9	8						1	7
				5		6		9
3	1	9	7			2		
	4	6	5	2		8		
	2		9		3			1

3×3 × 3×3 Sudoku

2	9	4	3	7	8	1	5	6
1	7	3	6	4	5	9	8	2
5	6	8	2	1	9	7	3	4
6	5	7	1	9	2	3	4	8
9	8	2	4	3	6	5	1	7
4	3	1	8	5	7	6	2	9
3	1	9	7	8	4	2	6	5
7	4	6	5	2	1	8	9	3
8	2	5	9	6	3	4	7	1

4×4 × 4×4 Sudoku

	F		2						6			C	B	3	
	C				4	8	E	A			0		D		
D	A	8			3		2	7	F			6		5	
6			E	D	F		C		8						7
	9	3		7					A						2
E						6	F	5		8	4		3		1
C	8		1	3	9	D		0	2		E				
	D		6		5	E	B		1					0	4
9	6					1		F	3	2		0		A	
				4		A	8		D	0	9	B		2	5
2		A		0	D		5	6	C						F
5						2					A		4	8	
B						4		1		A	2	F			0
	0		7			F	3	C		D			2	9	B
		5		1			A	9	0	B				D	
	2	D	A			9						1		4	

4×4 × 4×4 Sudoku

0	F	9	2	A	7	5	1	4	6	E	D	C	B	3	8
7	C	1	3	6	4	8	E	A	B	5	0	2	D	F	9
D	A	8	4	9	3	B	2	7	F	C	1	6	0	5	E
6	5	B	E	D	F	0	C	2	8	9	3	4	A	1	7
4	9	3	5	7	1	C	0	D	A	F	B	8	E	6	2
E	B	7	0	2	A	6	F	5	9	8	4	D	3	C	1
C	8	F	1	3	9	D	4	0	2	6	E	5	7	B	A
A	D	2	6	8	5	E	B	3	1	7	C	9	F	0	4
9	6	4	8	E	B	1	7	F	3	2	5	0	C	A	D
3	7	C	F	4	6	A	8	E	D	0	9	B	1	2	5
2	1	A	B	0	D	3	5	6	C	4	8	7	9	E	F
5	E	0	D	F	C	2	9	B	7	1	A	3	4	8	6
B	3	6	9	C	E	4	D	1	5	A	2	F	8	7	0
1	0	E	7	5	8	F	3	C	4	D	6	A	2	9	B
8	4	5	C	1	2	7	A	9	0	B	F	E	6	D	3
F	2	D	A	B	0	9	6	8	E	3	7	1	5	4	C

4×4 × 4×4 No-Promises Sudoku

	F		2						6			C	B	3	
	C				4	8	E	A			0		D		
D	A	8			3		2	7	F			6		5	
6			E	D	F		C		8						7
	9	3		7					A						2
E						6	F	5		8	4		3		1
C	8		1	3	9	D		0	2		E				
	D		6		5	E	B		1					0	4
9	6					1		F	3	2		0		A	
				4		A	8		D	0	9	B		2	5
2		A		0	D		5	6	C						F
5						2					A		4	8	
B				5		4		1		A	2	F			0
	0		7			F	3	C		D			2	9	B
		5		1			A	9	0	B				D	
	2	D	A			9						1		4	

This one has no solution.

4×4 × 4×4 No-Promises Sudoku

	F		2						6			C	B	3	
	C				4	8	E	A			0		D		
D	A	8			3		2	7	F			6		5	
6			E	D	F		C		8						7
	9	3		7					A						2
E						6	F	5		8	4		3		1
C	8		1	3	9	D		0			E				
	D		6		5	E	B		1					0	4
9	6					1		F	3	2		0		A	
				4		A	8			0	9	B		2	5
2		A		0	D		5	6	C						F
5						2					A		4	8	
B				5		4		1		A		F			0
	0		7			F	3	C		D			2	9	B
		5		1			A	9	0	B				D	
	2	D	A			9						1		4	

This one has multiple solutions.

n×n × n×n No-Promises Sudoku

Given a partially filled n×n×n×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 ≈ n^{n^4}.

Smart decision algorithm: ???

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

n×n × n×n No-Promises Sudoku

Naive decision algorithm: Time ≈ n^{n^4}.

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

For n = 100 (meaning 10,000 100 x 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.

n×n × n×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!

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. "L∈**NP**"

Is there always also an efficient algorithm
for finding solutions to L? "L∈**P**"

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.

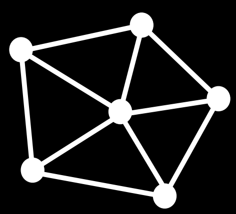
Suppose there is an efficient algorithm for verifying solutions to L. "L∈**NP**"

Is there always also an efficient algorithm for finding solutions to L? "L∈**P**"

Let's develop these notions formally…

We'll start by describing some sample algorithmic problems.

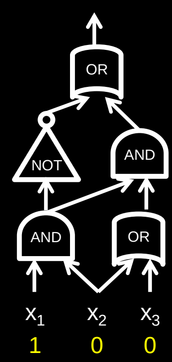
3-Coloring



Input: A graph

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

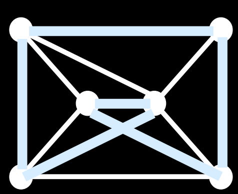
Circuit-Sat



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.

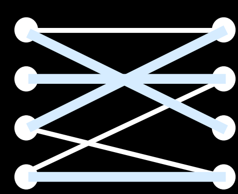
Hamiltonian Cycle



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.

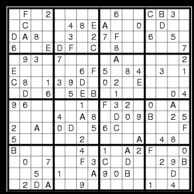
Bipartite Perfect Matching



Input: A bipartite graph

Task: Decide if there is a *perfect matching.* If so, find one.

n×n × n×n (No-Promises) Sudoku



Input: A partially filled $n^2 \times n^2$ Sudoku grid

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

Decision vs. Search

Each of these problems was of the form,

"Does a solution exist? If so, find one."

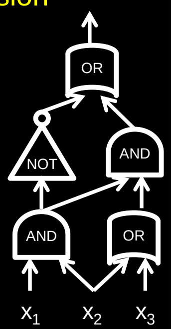
Decision problem — **Search** problem

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	Languages in $\{0,1\}^*$
Given G, does it have a Hamiltonian cycle?	HAM-CYCLE = $\{\langle G\rangle$: G contains a Hamiltonian cycle$\}$
Given bipartite G, does it have a perfect matching?	PMATCH = $\{\langle G\rangle$: G is bipartite, has perfect matching$\}$
Given circuit C, does it have a "satisfying" input string?	CIRCUIT-SAT = $\{\langle C\rangle$: C has a satisfying input$\}$
Given graph G, is it 3-colorable?	3-COL = $\{\langle G\rangle$: G is 3-colorable$\}$
Given partially filled Sudoku grid S, can it be completed?	SUDOKU = $\{\langle S\rangle$: S can be validly completed$\}$
Given TM M and input x, does M(x) halt?	HALTS = $\{\langle M,x\rangle$: 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.

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

HAM-CYCLE = $\{\langle G\rangle$: G contains a Hamiltonian cycle$\}$

PMATCH = $\{\langle G\rangle$: G is bipartite, has perfect matching$\}$

CIRCUIT-SAT = $\{\langle C\rangle$: C has a satisfying input$\}$

3-COL = $\{\langle G\rangle$: G is 3-colorable$\}$

SUDOKU = $\{\langle S\rangle$: S can be validly completed$\}$

HALTS = $\{\langle M,x\rangle$: M(x) halts$\}$

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 $\approx 3^{|V|}$ 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^6) ?	Barely…
O($n^{\log n}$) ?	Not really…
O(2^n) ?	No.
O(n!) ?	Please. Internet ≈ 22! bytes.

What is 'efficient'?

Is your algorithm for deciding L *'efficient'* if
on input strings of length **n** it runs in time…

O(n) ?
O(n log n) ?
O(n^2) ?
O(n^6) ?

Polynomial time:
O(n^c) for some fixed c.

Arguably efficient

O($n^{\log n}$) ?
O(2^n) ?
O(n!) ?

Not efficient

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.

Polynomial time

50 years of computer science experience shows it's a very compelling definition:

- It's independent of the "machine model":
  poly-time on a TM = poly-time on a RAM
  = poly-time in Java = poly-time in Python
- It's "robust": plug a poly-time subroutine into a poly-time algorithm: still poly-time.
- Empirically, it seems that most natural problems with poly-time algorithms also have efficient-in-practice algorithms.

Polynomial time

P = The set of all languages L such that
there is a constant c
and an algorithm (TM) A
such that A decides L and
A runs in time **O($|x|^c$)** on all inputs x.

Examples

CONN = {⟨G⟩ : G is a connected graph} ∈ **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 |⟨G⟩| is ≥ 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 described an $O(n^3)$ time algorithm in Lecture 12 (Graphs II).

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**.

Examples

Let SAME-REG = { $\langle R_1, R_2\rangle$: R_1, R_2 are reg. exprs. using U,·, squaring, such that $L(R_1) = L(R_2)$ }

$\langle$ a(aUb)^2, aaaUaabUabaUabb $\rangle$ ∈ SAME-REG

$\langle$ a^2(aUb), aaaUabb $\rangle$ ∉ SAME-REG

Theorem (Meyer–Stockmeyer 1972): SAME-REG ∉ **P**

So we understand **P**.

Great, we're halfway there!

Now what is **NP**?

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: 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)

NP: formal definition

Let L be a language. We say L∈**NP** iff…

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

V takes **two** inputs, x and y, where $|y| \le O(|x|^c)$.
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,
$\forall x \notin L$, $\forall y$, V(x,y) outputs NO.

Examples

SUDOKU ∈ **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 $|y| \le O(|x|)$.
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

SUDOKU ∈ **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.

For all x∈SUDOKU, there must be a valid completion y.
If magically given this y, V(x,y) will output YES.

For all x∉SUDOKU, there is no valid completion y.
So whatever y is given, V(x,y) will output NO.

Examples

3-COL ∈ **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 ∈ **NP**. (Why?)

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

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

Probably not, but no one knows.

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

Examples

PMATCH ∈ **NP**.

One reason:
Verifying a given perfect matching is easy.

Another reason:
Because PMATCH ∈ **P**!

Fact: **P** ⊆ **NP**.

P ⊆ NP

Proof:

Suppose L∈**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 $\overline{\text{PMATCH}}$ = {⟨G⟩ : G is bipartite, does NOT have a perfect matching }.

Is $\overline{\text{PMATCH}}$ in **NP**?

Yes! Clearly $\overline{\text{PMATCH}}$ ∈ **P** because PMATCH ∈ **P**.

(Just run the PMATCH algorithm, reverse the answer.)

∴ $\overline{\text{PMATCH}}$ ∈ **NP** because **P** ⊆ **NP**.

The P vs. NP problem

We know that **P** ⊆ **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!!

The P vs. NP problem

We know that **P** ⊆ **NP**.

Does **P** = **NP**?

If **P** ≠ **NP** then…
There is *some particular* L∈**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 = **NP** if and only if 3-SAT ∈ **P**

In particular, if **P** ≠ **NP** then 3-SAT ∉ **P**.

“3-SAT is the hardest problem in **NP**”

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

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

∴ Thus 3-SAT∈**P** if and only if CIRCUIT-SAT∈**P**.

So Cook-Levin Theorem is:

P = **NP** if and only if CIRCUIT-SAT∈**P**

Cook–Levin Theorem

P = **NP** if and only if CIRCUIT-SAT∈**P**

In particular, if **P** ≠ **NP** then CIRCUIT-SAT ∉ **P**.

"CIRCUIT-SAT is the hardest problem in **NP**"

The hardest problem(s) in NP

P = **NP** if and only if CIRCUIT-SAT∈**P**

If CIRCUIT-SAT is in **P**, then *all of* **NP** is in **P**.

Last lecture: There is a polynomial-time **reduction from** CIRCUIT-SAT **to** 3-COL.

∴ If 3-COL∈**P** then CIRCUIT-SAT∈**P**.
And hence *all of* **NP** is in **P**.

∴ **P** = **NP** if and only if 3-COL∈**P**.

The hardest problem(s) in NP

P = **NP** if and only if 3-COL∈**P**

Fact (Yato–Seta 2002): There's is a poly-time **reduction from** 3-COL **to** SUDOKU.

∴ If SUDOKU∈**P** then 3-COL∈**P**.
And hence *all of* **NP** is in **P**.

∴ **P** = **NP** if and only if SUDOKU∈**P**.

n×n × n×n No-Promises Sudoku

Question:

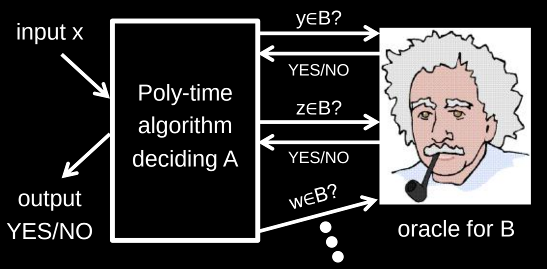
Is there a fixed constant c and an algorithm A such that A solves the decision problem in time $O(n^c)$?

This is **equivalent** to the **P** vs. **NP** problem!

Reductions

Definition 1:

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

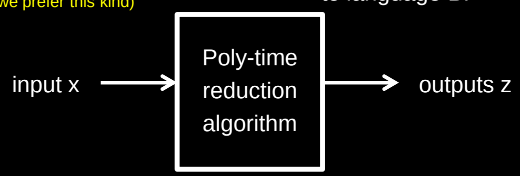


Reductions

Definition 2:

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

(we prefer this kind)



… such that x∈A if and only if z∈B.

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∈**P**, then A∈**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.

∴ CIRCUIT-SAT∈**P** ⇒ **NP** ⊆ **P** ⇒ **NP** = **P**.

And **NP** = **P** ⇒ CIRCUIT-SAT∈**P** because CIRCUIT-SAT∈**NP**.

P = NP if and only if CIRCUIT-SAT∈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.

The proof of the Cook–Levin Theorem is not too hard. We'll mention the high level idea later.

P = NP if and only if CIRCUIT-SAT∈P

NP-completeness

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

Note: Cook–Levin ⇒ "CIRCUIT-SAT is **NP**-hard".

Definition: A language L is **NP**-complete if:
a) L is **NP**-hard; and b) L∈**NP**.

NP-complete = "hardest problem in NP".
E.g.: CIRCUIT-SAT.

NP-completeness

Theorem: 3-COL is **NP**-complete.

Proof:

3-COL ∈ **NP** ✔

3-COL **NP**-hard because…

CIRCUIT-SAT $\leq_m$ 3-COL (last class)
All languages in **NP** (mapping) reduce to CIRCUIT-SAT.
∴ all languages in **NP** (mapping) reduce to 3-COL (by composing the two reductions).

■

IMPORTANT: Recipe for NP-completeness

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

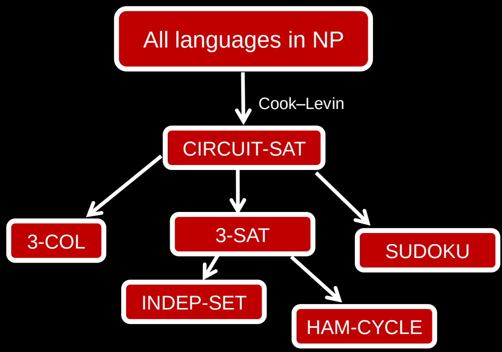
Step 1: Prove it is in **NP**.

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

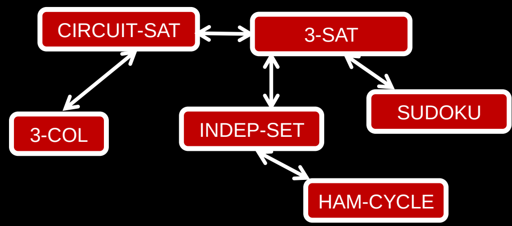
Be sure the reduction goes in the right direction!
To show B is hard, mapping reduce **some other hard problem A to it**, i.e, $A \leq_m B$.

Remember: reducing B to a hard problem does *not* show that B is hard. (Eg. 2-COLOR reduces to HALT)

NP-completeness via reductions

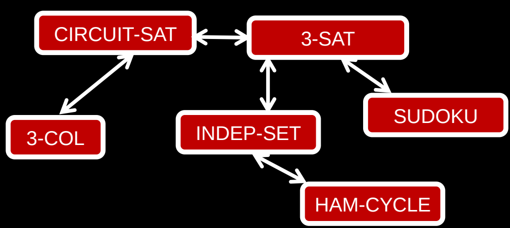


NP-completeness via reductions



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

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



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 $0 \leq x \leq c$ such that $x^2 = a \pmod{b}$?

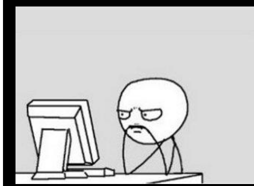
(Oct 2002): Given a sequence of Tetris pieces
and a number k, can you clear ≥ k lines?

(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
  - $x \in A \Leftrightarrow \exists\, y,\ |y| \le |x|^c$ 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 hardwired to x:
  - $C_x(y) = V(x,y)$
  - Telling if $x \in A$ amounts to telling if C_x is satisfiable
- $x \to C_x$ is a poly-time mapping reduction from A to Circuit-SAT.



Study Guide

Definitions:

Decision/search problems
P, **NP**, **NP**-hard, **NP**-complete
Poly-time (mapping) reduction

Theorems:

Cook–Levin Theorem

How-to:

Prove languages in **P**
Prove languages in **NP**
Show NP-completeness
Prove languages **NP**-hard by reduction