CS 420 / CS 620

NP

Wednesday, December 3, 2025

UMass Boston Computer Science

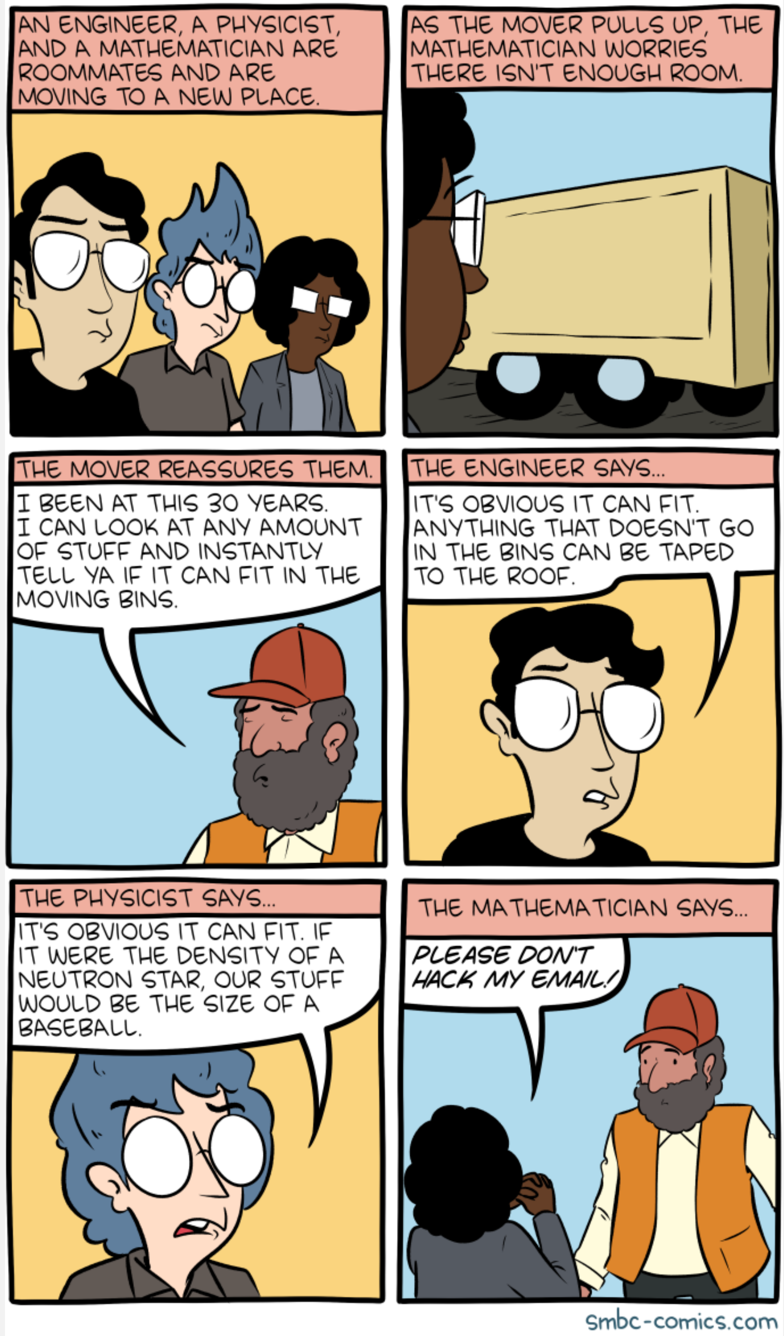
Announcements

- HW 12
    - ~~Out: **Mon 11/24 12pm (noon)**~~
    - ~~Thanksgiving: 11/26-11/30~~
    - Due: **Fri 12/5 12pm (noon)**

Last HW

- HW 13
    - Out: **Fri 12/5 12pm (noon)**
    - Due: **Fri 12/12 12pm (noon)** (classes end)
    - Late due: **Mon 12/15 12pm (noon)** (exams start)
        - **Nothing accepted after this** (please don't ask)

Who doesn't like niche NP jokes?



Class participation question (in Gradescope)

Q1 Which of the following are ways to show that a language is in NP?
1 Point

(select all that apply)

- [ ] create a deterministic poly time decider
- [ ] create a non-deterministic poly time decider
- [ ] create a deterministic poly time verifier
- [ ] create a non-deterministic poly time verifier

Previously: Poly Time Complexity Class (**P**)

P is the class of languages that are decidable in $\boxed{\text{polynomial time}}$ on a $\boxed{\text{deterministic single-tape Turing machine}}$. In other words,

$$P = \bigcup_k \mathrm{TIME}(n^k).$$

- Corresponds to "realistically" solvable problems:
  - Problems in **P**
    - = "solvable" or "tractable"
  - Problems outside **P**
    - = "unsolvable" or "intractable"

Previously: 3 Problems in $\mathbf{P}$

“search” problem

(to accept the string, decider must find a path)

- A Graph Problem:

$PATH = \{\langle G, s, t\rangle |\ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

- A Number Problem:

$RELPRIME = \{\langle x, y\rangle |\ x \text{ and } y \text{ are relatively prime}\}$

- A CFL Problem:

Every context-free language is a member of P

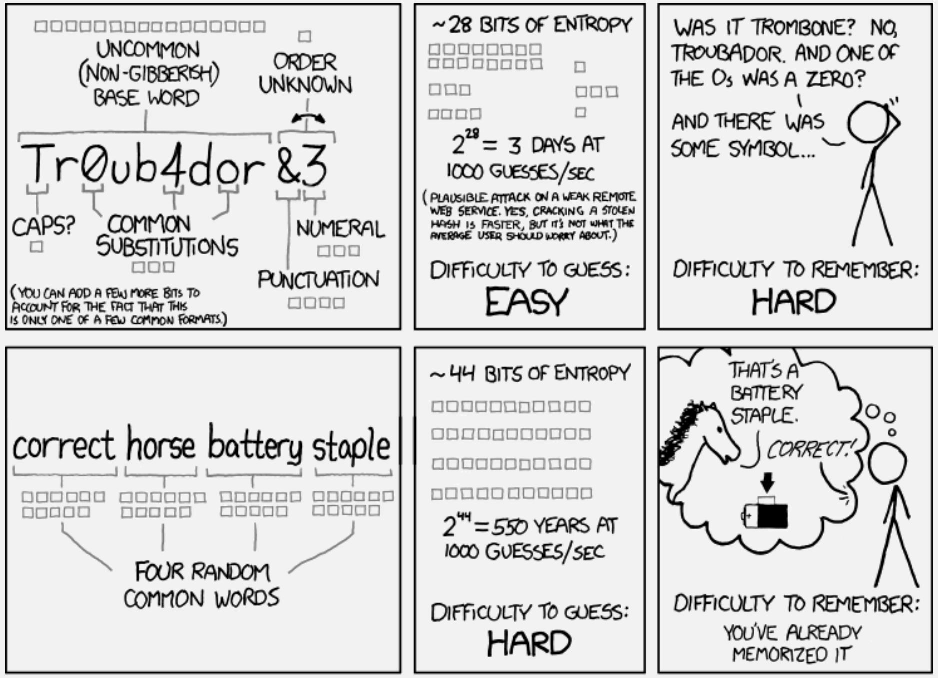
Search vs Verification

- **<u>Search</u>** problems are **often** **unsolvable**
- But, **<u>verification</u>** of a **search result** is **usually** **solvable**

EXAMPLES

- FACTORING
  - **Unsolvable**: <u>Find</u> factors of 8633
    - Must “try all” possibilities
  - **Solvable**: <u>Verify</u> 89 and 97 are factors of 8633
    - Just do multiplication

- PASSWORDS
  - **Unsolvable**: <u>Find</u> my `umb.edu` password
  - **Solvable** : <u>Verify</u> whether my `umb.edu` password is …
    - “`correct horse battery staple`”



THROUGH 20 YEARS OF EFFORT, WE'VE SUCCESSFULLY TRAINED EVERYONE TO USE PASSWORDS THAT ARE HARD FOR HUMANS TO REMEMBER, BUT EASY FOR COMPUTERS TO GUESS.

The *PATH* Problem

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

- It's a **search problem:**
  - Exponential time (brute force) **algorithm (n^n):**
    - Check all n^n possible paths and see if any connect s and t
  - Polynomial time **algorithm:**
    - Do a breadth-first search (roughly), marking "seen" nodes as we go ($\boldsymbol{n}$ = # nodes)

PROOF A polynomial time algorithm M for *PATH* operates as follows.

M = "On input $\langle G, s, t\rangle$, where G is a directed graph with nodes s and t:

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

$\boxed{O(\boldsymbol{n}^3)}$

Verifying a *PATH*

$PATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$

The **verification** problem:

- Given some path p in G, check that it is a **path from** s **to** t
- Let m = length of longest possible path = # ed

NOTE: extra argument p,
"**Verifying**" an answer requires
having a **potential answer to check!**

Verifier V = On input $<G, s, t, p>$, where p is some set of edges:

1. Check some edge in p has "from" node s; mark and set it as "current" edge
   - Max steps = $O(m)$
2. **Loop**: While there remains unmarked edges in p:
   1. Find the "next" edge in p, whose "from" node is the "to" node of "current" edge
   2. If found, then mark that edge and set it as "current", else reject
   - Each loop iteration: $O(m)$
   - # loops: $O(m)$
   - Total looping time = $O(m^2)$
3. Check "current" edge has "to" node t; if yes accept, else reject

☑

- Total time = $O(m) + O(m^2) = \boxed{O(m^2)}$ = polynomial in m

PATH can be **verified**
in **polynomial time**

Verifiers, Formally

$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a $\boxed{\text{directed path}}$ from s to $t\}$

A possible

... with **extra argument**: can be **any string** that **helps to find a result in poly time** (is often just a potential result itself)

Decider ...

A ***verifier*** for a language A is an algorithm V, where

$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

certificate, or ***proof***

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

- NOTE: a certificate c must be at most length n^k, where n = length of w
  - Why? Because it takes time n^k to read it

So *PATH* is polynomially verifiable

The class **NP**

DEFINITION

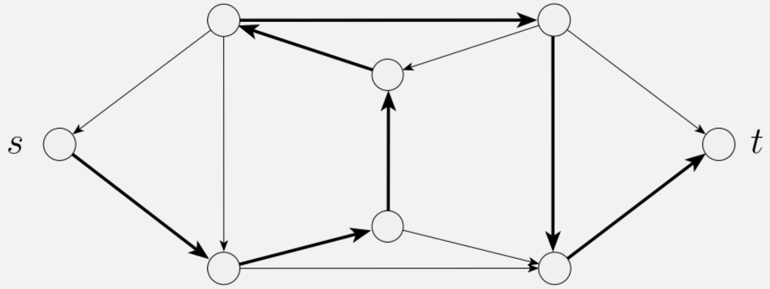
NP is the class of languages that have polynomial time verifiers.

- *PATH* is **in NP**, and **P**

The *HAMPATH* Problem

$$HAMPATH = \{\langle G, s, t\rangle \mid G \text{ is a directed graph with a Hamiltonian path from } s \text{ to } t\}$$

- A **Hamiltonian path** goes through every node in the graph



- The **Search problem:**
  - Exponential time (brute force) **algorithm:**
    - Check all possible paths and see if any connect s and t using all nodes
  - Polynomial time **algorithm:** ???
    - We don't know if there is one!!!
- The **Verification problem:**
  - Still $O(m^2)$! (same verifier for *PATH*)
  - *HAMPATH* is polynomially verifiable, but not polynomially decidable

The class **NP**

DEFINITION

NP is the class of languages that have polynomial time verifiers.

- *PATH* is **in NP**, and **P**
- *HAMPATH* is **in NP**, but it's <u>unknown</u> whether it's **in P**

NP = Nondeterministic Polynomial time

Definition: **NP** is the class of languages that have polynomial time verifiers.

THEOREM

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

⇒ If a language is in **NP**, then it has a non-deterministic poly time decider

- We know: If a lang L is in **NP**, then it has a poly time verifier V
- Need to: create NTM deciding L:

NTM definition needs to say what happens in each branch

(can't "do" anything with branch results)

On input w =

NTM runtime = slowest branch

- Nondeterministically run V with w and all possible poly length certificates c (and accept if it accepts)

NOTE: a verifier **cert** is usually a potential "answer", but does not have to be (like here)

Certificate c specifies a path

⇐ If a language has a non-deterministic poly time decider, then it is in **NP**

- We know: L has NTM decider N,
- Need to: show L is in **NP**, i.e., create polytime verifier V:

Deterministic (verifier) TMs cannot "call" non-deterministic TMs

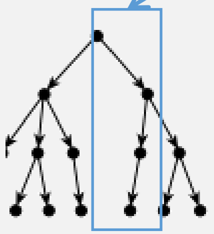
Because Converting NTM to deterministic is exponentially slower!

On input <w, c> =

Potentially exponential slowdown?

But which path to take?

- Convert N to deterministic TM, and run it on w, but take only one computation path
- Let certificate c dictate which computation path to follow



NP

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$$

NP = <u>Nondeterministic</u> polynomial time

NP vs **P**

P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

$$\mathrm{P} = \bigcup_k \mathrm{TIME}(n^k).$$

P = <u>Deterministic</u> polynomial time

$$\textbf{NTIME}(\boldsymbol{t(n)}) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

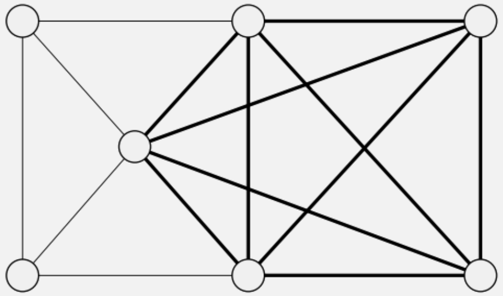
$$\mathrm{NP} \;= \bigcup\nolimits_k \mathrm{NTIME}(n^k)$$

Also, **NP** = <u>Deterministic</u> polynomial time **verification**

NP = <u>Nondeterministic</u> polynomial time

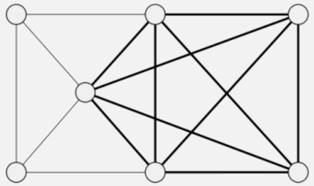
More **NP** Problems

- $\mathit{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
  - A **clique** is a **subgraph** where **every two nodes are connected**
  - A k**-clique contains** k **nodes**



- $$\begin{aligned}\mathit{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid\ & S = \{x_1, \ldots, x_k\}, \text{ and for some} \\ & \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}\end{aligned}$$

Theorem: *CLIQUE* is in NP



$CLIQUE = \{\langle G, k\rangle |\ G \text{ is an undirected graph with a } k\text{-clique}\}$

PROOF IDEA The possible clique is the certificate.

Let n = # nodes in G

PROOF The following is a verifier V for *CLIQUE*.

Cert c has at most n nodes

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

For each: node in cert c, check whether it's in G,

runtime: $O(n)$

For each: pair of nodes in cert c, check whether there's an edge in G,

runtime: $O(n^2)$

A ***verifier*** for a language A is an algorithm V, where

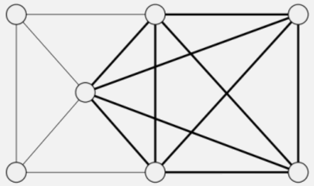
$$A = \{w|\ V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

We measure the time of a verifier only in terms of the length of w, so a ***polynomial time verifier*** runs in polynomial time in the length of w. A language A is ***polynomially verifiable*** if it has a polynomial time verifier.

How to prove a language is in **NP**:
Proof technique #1: **create a poly time verifier**

NP is the class of languages that have polynomial time verifiers.

Proof 2: *CLIQUE* is in NP



$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

$N =$ "On input $\langle G, k\rangle$, where G is a graph:
1. Nondeterministically select a subset c of k nodes of G.
2. Test whether G contains all edges connecting nodes in c.
3. If yes, *accept*; otherwise, *reject*."

"try all subgraphs"

Check whether a subgraph is clique:

Runtime: $O(n^2)$

To prove a lang L is in **NP**, create either a:
1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

Don't forget to count the steps

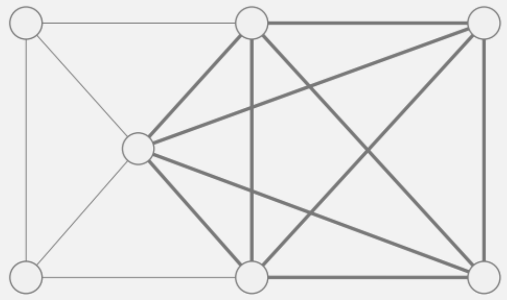
How to prove a language is in **NP**:
Proof technique #2: **create an NTM**

THEOREM

A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.

More **NP** Problems

- $\mathit{CLIQUE} = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$
  - A **clique** is a **subgraph** where **every two nodes are connected**
  - A k**-clique contains** k **nodes**



set sum

- $\mathit{SUBSET\text{-}SUM} = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some}$

  subset $\{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$ sum

  - Some **subset** of a **set of numbers** S must **sum to some total** t
  - e.g., $\langle\{\boxed{4}, 11, 16, \boxed{21}, 27\}, 25\rangle \in \mathit{SUBSET\text{-}SUM}$

Theorem: *SUBSET-SUM* is in NP

$$SUBSET\text{-}SUM = \{\langle S, t\rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = t\}$$

PROOF IDEA The possible subset is the certificate.

To prove a lang is in **NP**, create either:
1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

PROOF The following is a $\boxed{\text{verifier } V}$ for *SUBSET-SUM*.

V = "On input $\langle\langle S, t\rangle, c\rangle$:

1. Test whether c is a collection of numbers that sum to t.
2. Test whether S contains all the numbers in c.
3. If both pass, *accept*; otherwise, *reject*."

Don't forget to compute run time!
Does this run in poly time?

Proof 2: *SUBSET-SUM* is in NP

$$\mathit{SUBSET\text{-}SUM} = \{\langle S,t\rangle \mid S = \{x_1,\ldots,x_k\},\ \text{and for some } \{y_1,\ldots,y_l\} \subseteq \{x_1,\ldots,x_k\},\ \text{we have } \Sigma y_i = t\}$$

To prove a lang is in **NP**, create either:

1. **Deterministic** poly time **verifier**
2. **Nondeterministic** poly time **decider**

Don't forget to compute run time!
Does this run in poly time?

ALTERNATIVE PROOF We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for *SUBSET-SUM* as follows.

N = "On input $\langle S, t\rangle$:

1. Nondeterministically select a subset c of the numbers in S.
2. Test whether c is a collection of numbers that sum to t.
3. If the test passes, *accept*; otherwise, *reject*."

Nondeterministically runs the verifier on each possible subset "in parallel"

$$COMPOSITES = \{x \mid x = pq, \text{ for integers } p, q > 1\}$$

- A **composite number** is <u>not</u> **prime**

- *COMPOSITES* is **polynomially verifiable**
  - i.e., it's **in NP**
  - i.e., **factorability is in NP**

- A **certificate** could be:
  - Some **factor** that **is not 1**

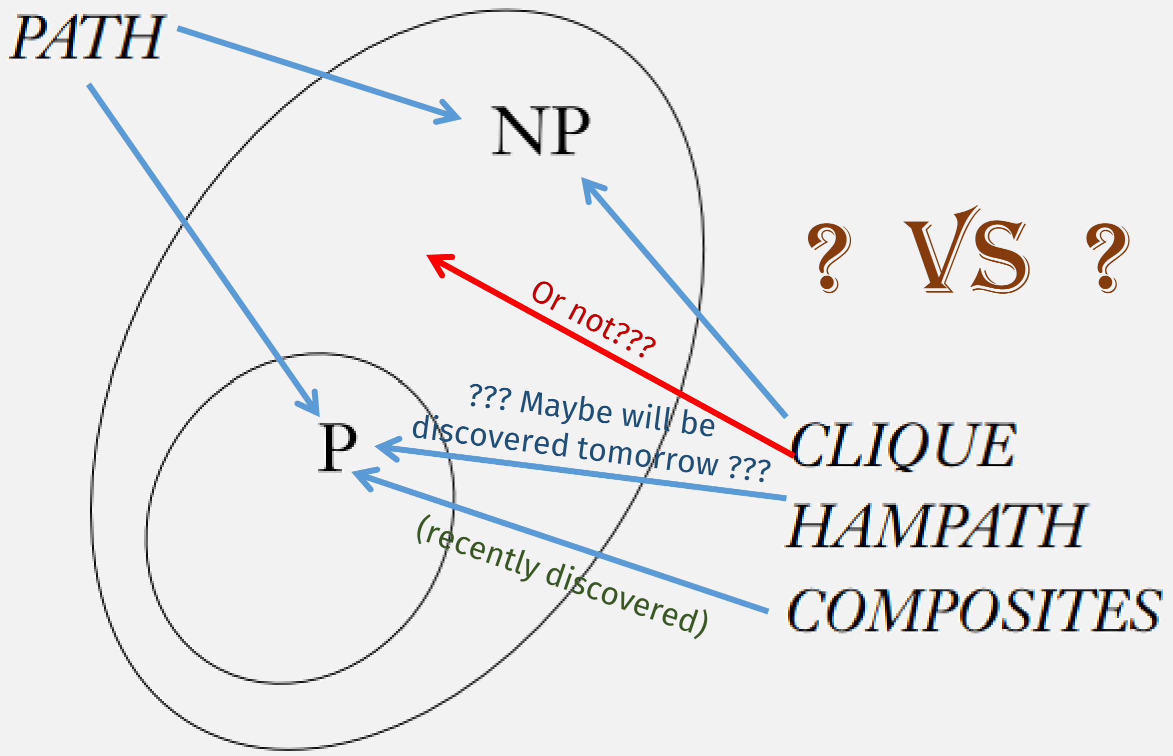
- **Checking existence of factors** (or not, i.e., **testing primality**) …
  - … is also **poly time**
  - But **only discovered** <u>recently</u> (2002)!

One of the Greatest unsolved

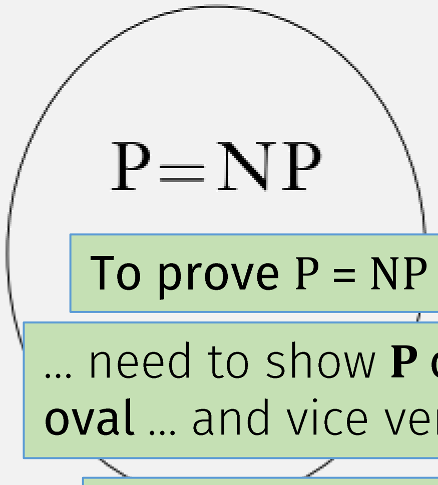
~~Bonus~~ ~~HW~~ Question: Does P = NP?

To prove P ≠ NP … … need to find a **language in NP** but **not in P!**

(you know how to do it!)



? VS ?



To prove P = NP …

(you also know how to do it!)

… need to show **P oval overlaps with NP oval** … and vice versa**!**

… need need to show every **language in NP** is also **in P**, and vice versa**!**

BUT … **How to prove an algorithm <u>doesn't</u> have poly time algorithm?**
(in general it's hard to prove that something <u>doesn't</u> exist)

Not this course, see Sipser Ch8-9

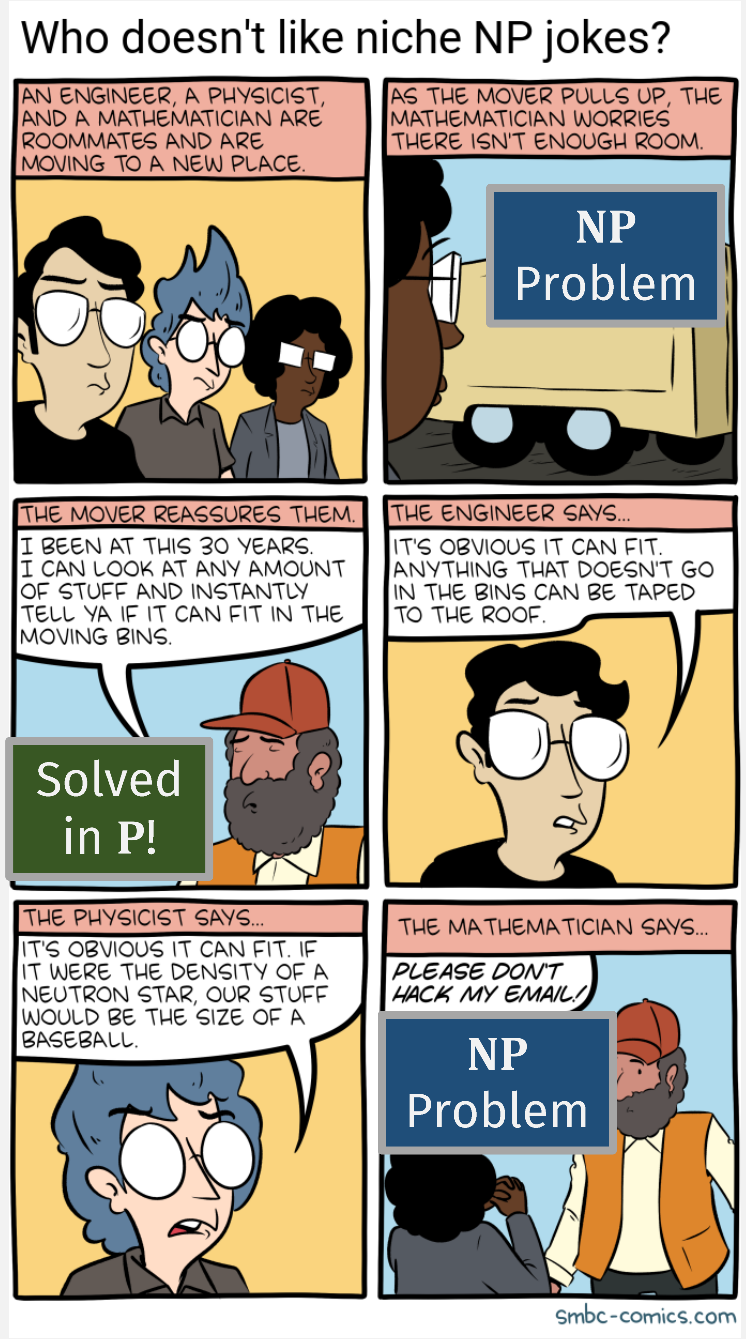
Implications if **P = NP**

- **Problems with "brute force"** ("try all") **solutions** now have **efficient solutions**

- I.e., **"unsolvable" problems are "solvable"**

- <u>BAD</u>:
  - **Cryptography** needs **unsolvable problems**
  - **perfect AI learning, recognition** (maybe good?)

- <u>GOOD</u>: **Optimization problems are solved**
  - **Optimal resource allocation** could **fix all the world's** (food, energy, space …) **problems?**



Progress on whether **P = NP** ?

- Some, but still not close

$P \stackrel{?}{=} NP$

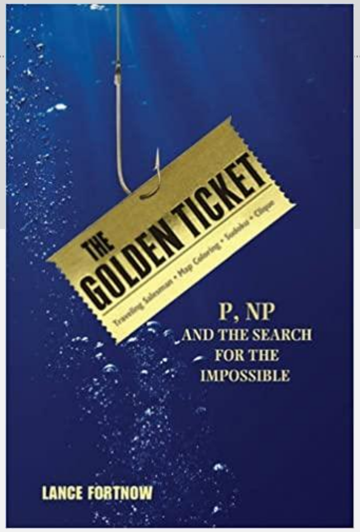
Scott Aaronson*

The Status of the P Versus NP Problem

By Lance Fortnow

Communications of the ACM, September 2009, Vol. 52 No. 9, Pages 78-86

10.1145/1562164.1562186



- One important concept discovered:
  - **NP**-Completeness

NP-Completeness

Must prove for **all langs**, not just a single lang

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and

easy

2. every A in NP is polynomial time reducible to B.

hard????

What's this?

Flashback: Mapping Reducibility

Language A is ***mapping reducible*** to language B, written $\boxed{A \leq_{\mathrm{m}} B}$, if there is a $\boxed{\text{computable function } f\colon \Sigma^* \longrightarrow \Sigma^*}$, where for every w,

$$\boxed{w \in A \iff f(w) \in B.}$$

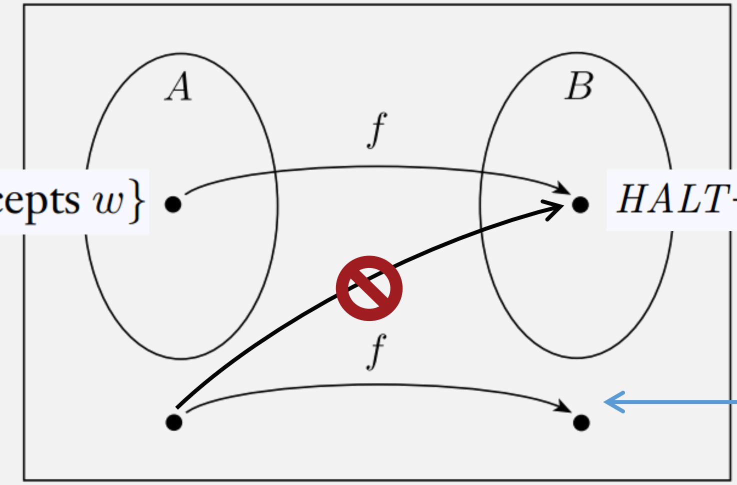
IMPORTANT: "if and only if" ...

The function f is called the ***reduction*** from A to B.

To show mapping reducibility:

1. create **computable fn**
2. and then show **forward direction**
3. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

$A_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$



$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}$

... means $\boxed{\overline{A} \leq_{\mathrm{m}} \overline{B}}$

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Polynomial Time Mapping Reducibility

Language A is ***mapping reducible*** to language
if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A t

To show poly time mapping reducibility:

1. create **computable fn**
2. **show computable fn runs in poly time**
3. then show **forward direction**
4. and show **reverse direction** (or **contrapositive** of **reverse direction**)

Language A is ***polynomial time mapping reducible***, or simply ***polynomial time reducible***, to language B, written $\boxed{A \leq_{\mathrm{P}} B}$, if a polynomial time computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$ exists, where for every w,

$$\boxed{w \in A \iff f(w) \in B.}$$

Don't forget: "if and only if" ...

The function f is called the ***polynomial time reduction*** of A to B.

poly time poly time

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

Flashback: If $A \leq_m B$ and B is decidable, then A is decidable.

Has a decider

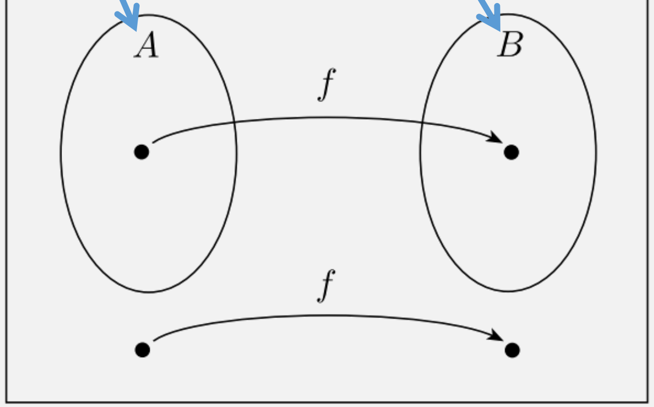
PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

decides

decides



This proof only works because of the if-and-only-if requirement

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$\boxed{w \in A \iff f(w) \in B.}$$

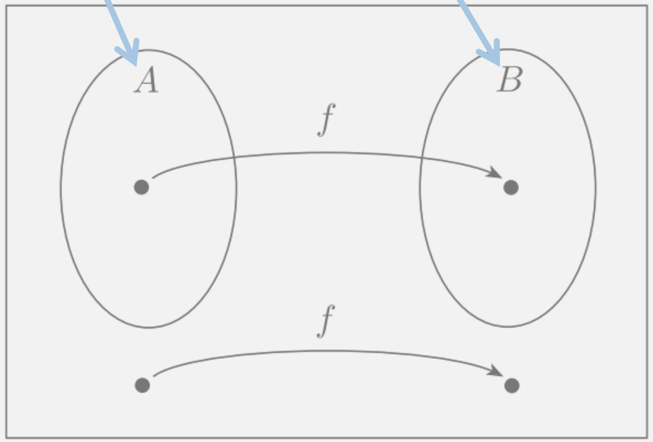
The function f is called the ***reduction*** from A to B.

Thm: If $A \leq_{\cancel{\mathrm{m}}\,\mathbf{P}} B$ and B $\in \mathrm{P}$ ~~is decidable,~~ then A $\in \mathrm{P}$ ~~is decidable.~~

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is ***mapping reducible*** to language B, written $A \leq_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

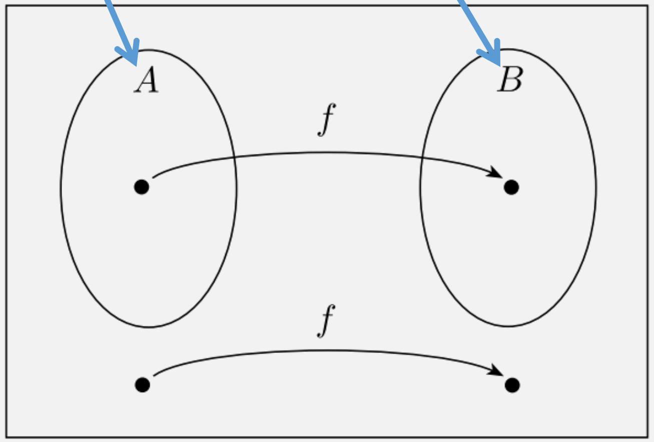
The function f is called the ***reduction*** from A to B.

Thm: If $A \le_{\cancel{\mathrm{m}}\,\mathbf{P}} B$ and B ~~is decidable,~~ $\in \mathrm{P}$ then A ~~is decidable.~~ $\in \mathrm{P}$

PROOF We let M be the poly time decider for B and f be the poly time reduction from A to B. We describe a poly time decider N for A as follows.

$N =$ "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."



Language A is poly time ***mapping reducible*** to language B, written $A \le_{\mathrm{m}} B$, if there is a computable function $f\colon \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** from A to B.

NP-Completeness

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

- How does this help the **P** = **NP** problem?

THEOREM

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

THEOREM

If B is NP-complete and $B \in \mathrm{P}$, then $\mathrm{P} = \mathrm{NP}$.

To prove **P** = **NP**, must show:

1. every language in **P** is in **NP**
   - Trivially true (why?) Convert deci

2. every language in **NP** is in **P**
   - Given a language $A \in$ **NP** ...
   - ... can **poly time mapping reduce** A to B $A \leq_{\mathrm{P}} B$
     - because B is **NP**-Complete
   - Then A also $\in$ **P** ...
     - Because $A \leq_{\mathrm{P}} B$ and $B \in \mathrm{P}$, then $A \in \mathrm{P}$

(prev slide)

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So to prove **P** = **NP**, we only need to **find a poly-time algorithm** for <u>one</u> **NP**-Complete problem!

Thus, **if a language** B **is NP**-complete and **in P**, **then P** = **NP**

An **NP**-Complete Language?

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

So to prove **P** = **NP**, we only need to **find a poly-time algorithm for** **one** **NP-Complete problem!**

Thus, **if a language** B **is NP-complete** and **in P**, **then P = NP**

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: *SAT* **NP**-complete

??

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z
Operation	Combines Boolean **variables**	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)

Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean **value**	x, y, z
Operation	Combines Boolean **variables**	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines **vars** and **operations**	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$

Boolean Satisfiability

- A **Boolean formula** is **satisfiable** if …

- … there is **some assignment** of TRUE or FALSE $(1$ or $0)$ to its **variables** that **makes the entire formula** TRUE

- Is $(\overline{x} \wedge y) \vee (x \wedge \overline{z})$ satisfiable?
  - Yes
  - $x = \mathrm{FALSE}$,
    $y = \mathrm{TRUE}$,
    $z = \mathrm{FALSE}$

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: SAT is **NP**-complete

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

The Boolean Satisfiability Problem

$SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: SAT is in **NP**:

- Let $\boldsymbol{n}$ = the number of variables in the formula

Verifier:

On input <ϕ, c>, where c is a possible assignment of variables in ϕ to values:

- Plug values from c into ϕ, **Accept** if result is TRUE

Running Time: $O(\boldsymbol{n})$

Non-deterministic Decider:

On input <ϕ>, where ϕ is a boolean formula:

- Non-deterministically try all possible assignments in parallel
- **Accept** if any satisfy ϕ

Running Time: Checking each assignment takes time $O(\boldsymbol{n})$

The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Theorem: *SAT* **NP**-complete

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

☑ **1.** B is in NP, and

?? → **2.** every A in NP is polynomial time reducible to B.

Proving ^the first!^ **NP**-Completeness ^problem^ is hard!

But after we find one, then we can use that problem to prove other problems **NP**-Complete!

(Just like figuring out the first undecidable problem was hard!)

THEOREM

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

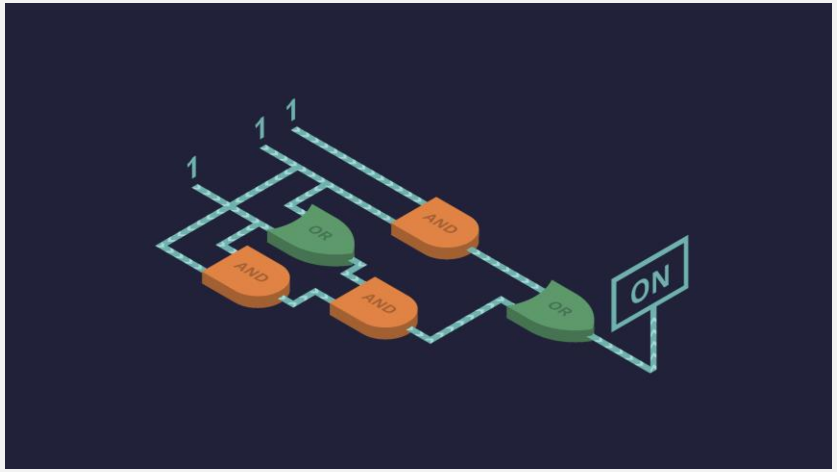
The Boolean Satisfiability Problem

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

<u>Theorem</u>**:** *SAT* **NP**-complete

The first **NP**-Complete problem

It sort of makes sense that every problem can be reduced to it …



PROOF: The Cook-Levin Theorem

(complicated proof
--- defer explaining for now, assume it's true)

The Boolean Satisfiability Problem

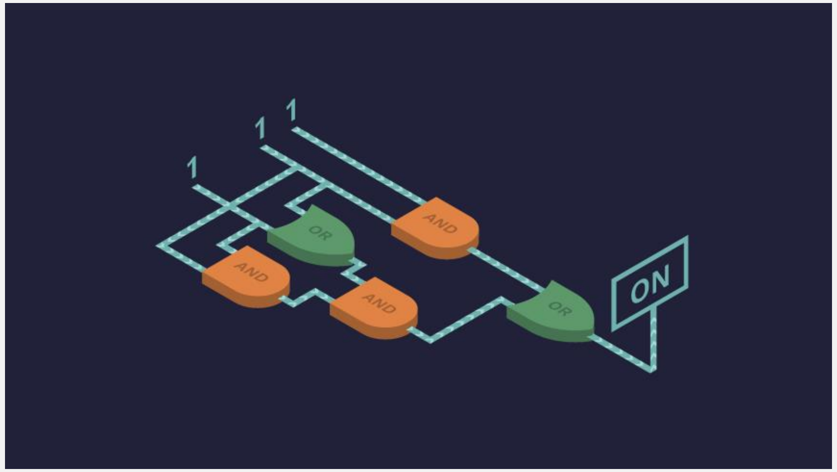
$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

<u>Theorem:</u> *SAT* **NP**-complete

PROOF: The Cook-Levin Theorem

(complicated proof
--- defer explaining for now, assume it's true)

Then **we can use** *SAT* **to prove other problems**
NP-Complete!



THEOREM

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

The *3SAT* Problem

$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

??

<u>Theorem:</u> $3SAT$ is **NP**-complete

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$

More Boolean Formulas

A Boolean _____	Is ...	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.

More Boolean Formulas

A Boolean _____	Is ...	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
Conjunctive Normal Form (**CNF**)	**Clauses** ANDed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6)$

∧ = AND = "Conjunction"
∨ = OR = "Disjunction"
¬ = NOT = "Negation"

More Boolean Formulas

A Boolean _____	Is …	Example:
Value	TRUE or FALSE (or 1 or 0)	TRUE, FALSE
Variable	Represents a Boolean value	x, y, z
Operation	Combines Boolean variables	AND, OR, NOT ($\wedge$, $\vee$, and $\neg$)
Formula ϕ	Combines vars and operations	$(\overline{x} \wedge y) \vee (x \wedge \overline{z})$
Literal	A var or a negated var	x or $\overline{x}$.
Clause	**Literals** ORed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$
Conjunctive Normal Form (**CNF**)	**Clauses** ANDed together	$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6)$
3CNF Formula	Three **literals** in each **clause**	$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4)$

∧ = AND = "Conjunction"
∨ = OR = "Disjunction"
¬ = NOT = "Negation"

Key thm:
Let's prove it so we can use it

THEOREM

known

unknown

If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

To use this theorem, C must be in **NP**

DEFINITION

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Proof:

- Need to show: C is **NP**-complete:
  - it's in **NP** (given), and
  - every lang A in **NP** reduces to C in poly time (must show)
- For every language A in **NP**, reduce $A \rightarrow C$ by:
  - First reduce $A \rightarrow B$ in poly time
    - Can do this because B is **NP**-Complete
  - Then reduce $B \rightarrow C$ in poly time
    - This is given
- Total run time: Poly time + poly time = poly time

THEOREM

<u>Using</u>: If $\boxed{B \text{ is NP-complete}}$ and $\boxed{B \leq_{\mathrm{P}} C}$ for $\boxed{C \text{ in NP}}$, then $\boxed{C \text{ is NP-complete.}}$

<u>**3 steps**</u> to prove a language $\boxed{C \text{ is } \mathbf{NP}\text{-complete:}}$

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

To show poly time <u>mapping reducibility</u>:
1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language C is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let $C = 3SAT$, to prove $3SAT$ is **NP**-Complete:

1. Show $3SAT$ is in **NP**

Flashback: $3SAT$ is in **NP**

$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$

Let $\boldsymbol{n}$ = the number of variables in the formula

Verifier:

On input $<\phi, c>$, where c is a possible assignment of variables in ϕ to values:

- Accept if c satisfies ϕ

Running Time: $O(\boldsymbol{n})$

Non-deterministic Decider:

On input $<\phi>$, where ϕ is a boolean formula:

- Non-deterministically try all possible assignments in parallel
- Accept if any satisfy ϕ

Running Time: Checking each assignment takes time $O(\boldsymbol{n})$

THEOREM

<u>Using</u>: If $\boxed{B \text{ is NP-complete}}$ and $\boxed{B \leq_P C}$ for C in NP, then C is NP-complete.

<u>**3 steps**</u> to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

<u>**Example**</u>:

Let $C = 3SAT$, to prove $3SAT$ is **NP**-Complete:

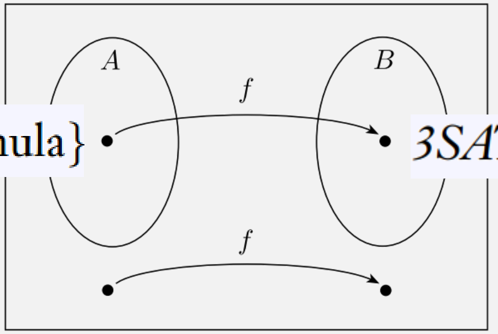
☑ 1. Show $3SAT$ is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: SAT

3. Show a poly time mapping reduction from SAT to $3SAT$

Theorem: *SAT* is Poly Time Reducible to *3SAT*

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$



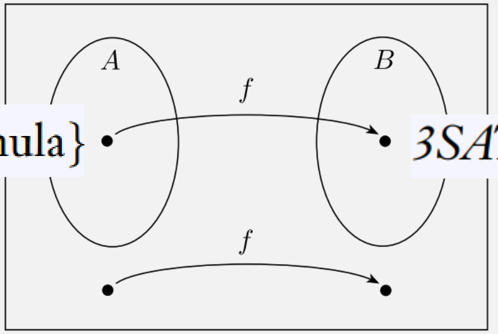
$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

To show poly time mapping reducibility:

1. create **computable fn** f,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
   ⇒ if $\phi \in SAT$, then $f(\phi) \in 3SAT$
4. and **reverse direction**
   ⇐ if $f(\phi) \in 3SAT$, then $\phi \in SAT$
   (or **contrapositive** of **reverse direction**)
   ⇐ (alternative) if $\phi \notin SAT$, then $f(\phi) \notin 3SAT$

Theorem: *SAT* is Poly Time Reducible to *3SAT*

$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$



$3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

Want: poly time computable fn converting a Boolean formula ϕ to 3CNF:

1. Convert ϕ to CNF (an AND of OR clauses)
   a) Use DeMorgan's Law to push negations onto literals

      $\neg(P \vee Q) \iff (\neg P) \wedge (\neg Q)$ $\quad\quad$ $\neg(P \wedge Q) \iff (\neg P) \vee (\neg Q)$ $\quad$ $O(n)$

   b) Distribute ORs to get ANDs outside of parens

      $(P \vee (Q \wedge R)) \Leftrightarrow ((P \vee Q) \wedge (P \vee R))$ $\quad$ $O(n)$

2. Convert to 3CNF by adding new variables

   $(a_1 \vee a_2 \vee a_3 \vee a_4) \Leftrightarrow (a_1 \vee a_2 \vee z) \wedge (\overline{z} \vee a_3 \vee a_4)$ $\quad$ $O(n)$

Remaining step: show iff relation holds …

… this thm is a special case, don't need to separate forward/reverse dir bc each step is already a known "law"

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let $C = 3SAT$, to prove $3SAT$ is **NP**-Complete:

☑ 1. Show $3SAT$ is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: SAT

☑ 3. Show a poly time mapping reduction from SAT to $3SAT$

Each **NP**-complete problem we prove makes it easier to prove the next one!

NP-Complete problems, so far

- $SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$ (havent proven yet)

- $3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$ (reduced *SAT* to *3SAT*)

- $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$ (reduce *???* to *CLIQUE*)?

Each **NP**-complete problem we prove makes it easier to prove the next one!

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let $C = 3SAT$, to prove $3SAT$ is **NP**-Complete:

☑ 1. Show $3SAT$ is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: SAT

☑ 3. Show a poly time mapping reduction from SAT to $3SAT$

Each **NP**-complete problem we prove makes it easier to prove the next one!

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

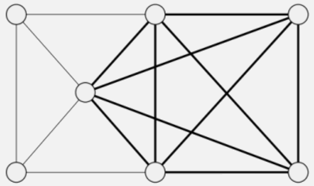
Let C = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

?1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

?2. Choose B, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

?3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

Flashback: *CLIQUE* is in NP



$CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

PROOF IDEA The clique is the certificate.

Let $\boldsymbol{n}$ = # nodes in G

c is at most $\boldsymbol{n}$

PROOF The following is a verifier V for *CLIQUE*.

V = "On input $\langle\langle G, k\rangle, c\rangle$:

1. Test whether c is a subgraph with k nodes in G.
2. Test whether G contains all edges connecting nodes in c.
3. If both pass, *accept*; otherwise, *reject*."

For each node in c, check whether it's in G: $O(\boldsymbol{n})$

For each pair of nodes in c, check whether there's an edge in G: $O(\boldsymbol{n}^2)$

THEOREM

<u>Using</u>: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

<u>**3 steps**</u> to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

<u>**Example**</u>**:**

Let C = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

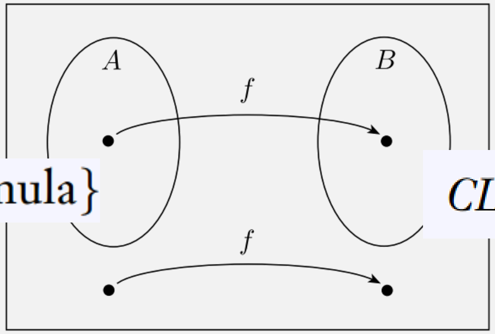
☑ 1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

☑ 2. Choose B, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

? 3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

Theorem: $3SAT$ is polynomial time reducible to $CLIQUE$.

$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

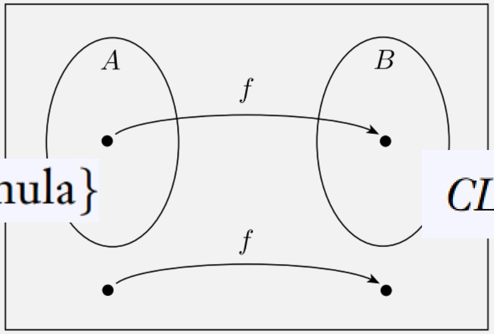


$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

To show poly time mapping reducibility:

1. create **computable fn**,
2. show that it **runs in poly time**,
3. then show **forward direction** of mapping red.,
4. and **reverse direction**
   (or **contrapositive** of **reverse direction**)

Theorem: *3SAT* is polynomial time reducible to *CLIQUE*.



$3SAT = \{\langle \phi \rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$

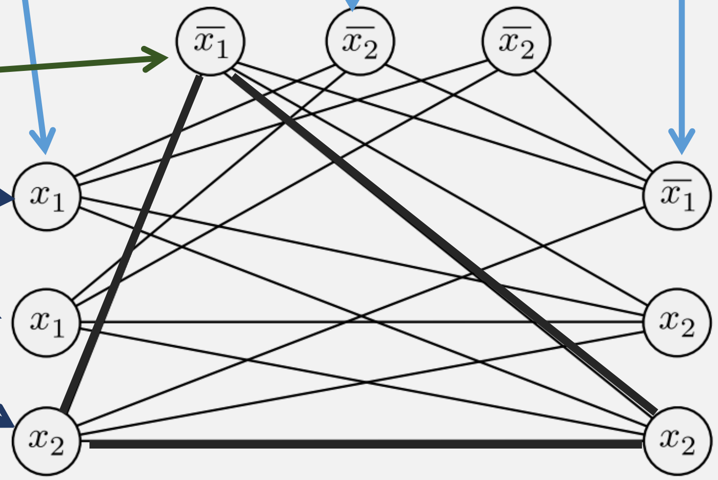
$CLIQUE = \{\langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$

Need: poly time computable fn converting a 3cnf-formula …

Example:

$$\phi = (x_1 \vee x_1 \vee \boxed{x_2}) \wedge (\boxed{\overline{x_1}} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee \boxed{x_2})$$

- … to a graph containing a clique:
  - Each clause maps to a group of 3 nodes
  - Connect all nodes except:
    - Contradictory nodes
    - Nodes in the same group



Don't forget iff

Runs in **poly time:**

- \# literals = # nodes $O(\boldsymbol{n})$
- \# edges poly in # nodes $O(\boldsymbol{n^2})$

$\Rightarrow$ If $\phi \in 3SAT$

- Then each clause has a TRUE literal
  - Those are nodes in the 3-clique!
  - E.g., $x_1 = 0$, $x_2 = 1$

$\Leftarrow$ If $\phi \notin 3SAT$

- Then for any assignment, some clause must have a contradiction with another clause
- Then in the graph, some clause's group of nodes won't be connected to another group, preventing the clique

THEOREM

Using: If B is NP-complete and $B \leq_P C$ for C in NP, then C is NP-complete.

3 steps to prove a language is **NP**-complete:

1. Show C is in **NP**
2. Choose B, the **NP**-complete problem to reduce from
3. Show a poly time mapping reduction from B to C

Example:

Let C = ~~*3SAT*~~ ***CLIQUE***, to prove ~~*3SAT*~~ ***CLIQUE*** is **NP**-Complete:

☑1. Show ~~*3SAT*~~ ***CLIQUE*** is in **NP**

☑2. Choose B, the **NP**-complete problem to reduce from: ~~*SAT*~~ ***3SAT***

☑3. Show a poly time mapping reduction from ***3SAT*** to ~~*3SAT*~~ ***CLIQUE***

NP-Complete problems, so far

- $SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}$ (havent proven yet)

- $3SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable 3cnf-formula}\}$ (reduced *SAT* to *3SAT*)

- $CLIQUE = \{\langle G, k\rangle \mid G \text{ is an undirected graph with a } k\text{-clique}\}$ (reduced *3SAT* to *CLIQUE*)

Each **NP**-complete problem we prove makes it easier to prove the next one!

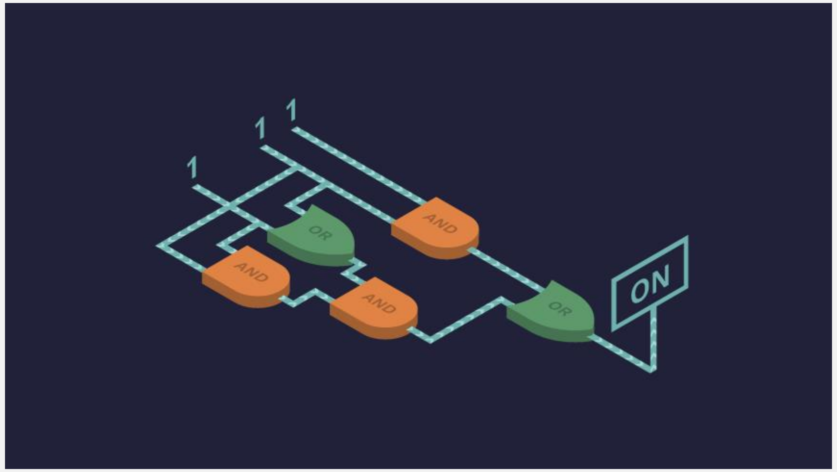
Next Time: The Cook-Levin Theorem

The first **NP**-Complete problem

THEOREM

SAT is NP-complete.

It sort of makes sense that every problem can be reduced to it …



After this, it'll be much easier to find other **NP**-Complete problems!

THEOREM

If B is NP-complete and $B \leq_{\mathrm{P}} C$ for C in NP, then C is NP-complete.