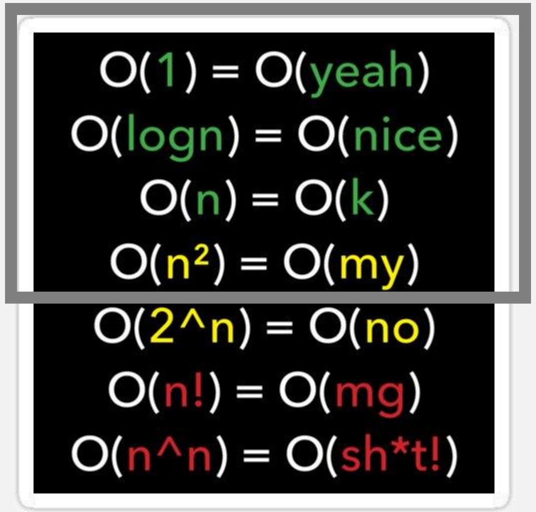
CS 420 / CS 620

Polynomial Time (P)

Monday, December 1, 2025

UMass Boston Computer Science



Caveat:
This class: **polynomial time = "good"** (won't take forever)
Real programmers: **polynomial time = "eh"** (pretty slow)

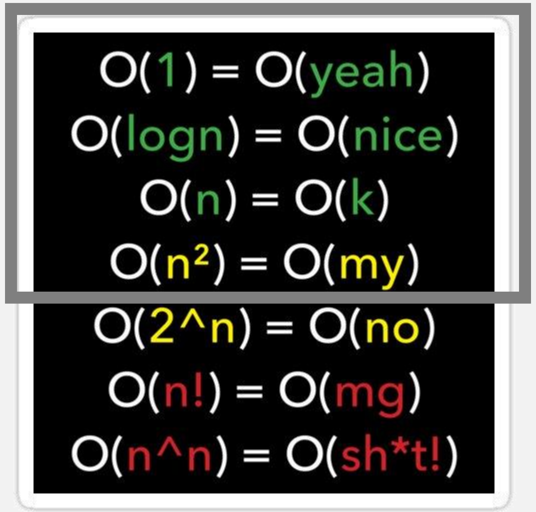
Announcements

- 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)

Caveat:
This class: **polynomial time = "good"** (won't take forever)
Real programmers: **polynomial time = "eh"** (pretty slow)



Class participation question (in Gradescope)

Q1 The time complexity class P represents what kind of problems ?
1 Point

(select all that apply)

- [ ] realistically solvable problems
- [ ] tractable problems
- [ ] problems that have a polynomial time algorithm
- [ ] languages decided by Turing-machines that run in a worst case polynomial number of steps

Previously: Time Complexity

Running Time or **Time Complexity** is a property of decider TMs (algorithms)

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f: \mathcal{N} \longrightarrow \mathcal{N}$, where $f(n)$ is the maximum number of steps that M uses on any input of length n. If $f(n)$ is the running time of M, we say that M runs in time $f(n)$ and that M is an $f(n)$ time Turing machine. Customarily we use n to represent the length of the input.

Depends on size of input

Worst case

Last Time: Time Complexity Classes

Big-O = asymptotic upper bound,
i.e., "only care about large n"

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ be a function. Define the ***time complexity class***, $\mathbf{TIME}(t(n))$, to be the collection of all languages that are decidable by an $O(t(n))$ time Turing machine.

Remember:
- TMs: have **a time complexity** (i.e., a running time),
- languages: are in **a time complexity class**

The **time complexity class** of a language is determined by the **time complexity** (running time) of its deciding TM

But: a language can have multiple **TMs deciding it,** so could be in multiple **time complexity classes**

The Polynomial Time Complexity Class (**P**)

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

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

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

“Unsolvable” Problems

- **Unsolvable problems** (those **outside P**):
  - **usually only have “brute force” solutions**
  - i.e., **“try all possible inputs”**
  - **“unsolvable”** applies **only to large *n***

Time it takes a hacker to brute force your password in 2025

Hardware: 12 x RTX 5090 | Password hash: bcrypt (10)

Number of Characters	Numbers Only	Lowercase Letters	Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters	Numbers, Upper and Lowercase Letters, Symbols
4	Instantly	Instantly	Instantly	Instantly	Instantly
5	Instantly	Instantly	57 minutes	2 hours	4 hours
6	Instantly	46 minutes	2 days	6 days	2 weeks
7	Instantly	20 hours	4 months	1 year	2 years
8	Instantly	3 weeks	15 years	62 years	164 years
9	2 hours	2 years	791 years	3k years	11k years
10	1 day	40 years	41k years	238k years	803k years
11	1 weeks	1k years	2m years	14m years	56m years
12	3 months	27k years	111m years	917m years	3bn years
13	3 years	705k years	5bn years	56bn years	275bn years
14	28 years	18m years	300bn years	3tn years	19tn years
15	284 years	477m years	15tn years	218tn years	1qd years
16	2k years	12bn years	812tn years	13qd years	94qd years
17	28k years	322bn years	42qd years	840qd years	6qn years
18	284k years	8tn years	2qn years	52qn years	463qn years

Mathematicians are weird.

Brute-force attack

From Wikipedia, the free encyclopedia

In cryptography, a **brute-force attack** consists of an attacker submitting many passwords or passphrases with the hope of eventually guessing a combination correctly. The attacker systematically checks all possible passwords and passphrases until the correct one is found. Alternatively, the attacker can attempt to guess the key which is typically created from the password using a key derivation function. This is known as an **exhaustive key search**.

As usual, in this class we’re interested in questions like:

today

How to prove something is **“solvable”** (in **P**)?

How to prove something is **“unsolvable”** (not in **P**)?

(much harder)

3 Problems in **P**

- A <u>Graph</u> Problem:

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

- A <u>Number</u> Problem:

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

- A <u>CFL</u> Problem:

  Every context-free language is a member of P

A decider!

- **To prove** that a **language** is **"solvable"**, i.e., **in P ...**
  - **... construct** a polynomial **time algorithm deciding the language**
- **(These may also have** nonpolynomial, i.e., **brute force, algorithms)**
  - Check all possible ... paths/numbers/strings ...

Interlude: Graph Encodings

$$(\{1,2,3,4,5\},\ \{(1,2),\ (2,3),\ (3,4),\ (4,5),\ (5,1)\})$$

- For graph algorithms, "length of input" n usually = # of vertices
  - (Not number of chars in the encoding)
- So given graph $G = (V, E)$, $n = |V|$
- Max edges?
  - $= O(|V|^2) = O(n^2)$
- So if a set of graphs (call it lang L) is decided by a TM where
  - # steps of the TM = polynomial in the # of vertices

    Or polynomial in the # of edges

- Then L is in $\mathbf{P}$

3 Problems in $\mathbf{P}$

- A Graph Problem:

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

- A Number Problem:

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

- A CFL Problem:

Every context-free language is a member of P

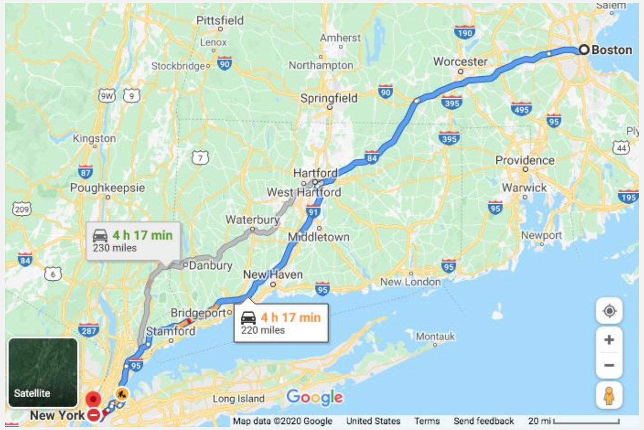
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

A Graph Theorem: $PATH \in \mathrm{P}$

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

(A **path** is a sequence of nodes connected by edges)



- To prove that a language is in **P** …
- … we must construct a polynomial time algorithm deciding the lang

A decider!

- A non-polynomial (i.e., "brute force") algorithm:
  - check all possible combination (ordering) of all vertices,
  - see if any connect s to t
  - If n = # vertices, then # paths $\approx n^n$ or $n!$ (worse than $2^{O(n)}$)

A Graph Theorem: $\mathit{PATH} \in \mathrm{P}$

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

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*."

# of steps (worst case) **(*n* = # nodes):**

- Line 1: **1 step**

A Graph Theorem: $PATH \in \mathrm{P}$

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

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*."

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$

A Graph Theorem: $PATH \in \mathrm{P}$

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

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*."

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$
  - ➢ # iterations (line 2): loop runs at most n times

A Graph Theorem: $PATH \in \mathrm{P}$

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

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*."

(Breadth-first search)

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$
  - # iterations (line 2): loop runs at most n times
  - ➢ Total: $\boxed{O(n^3)}$

A Graph Theorem: $\mathit{PATH} \in \mathrm{P}$

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

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*."

# of steps (worst case) **(n = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(n^2)$
  - # iterations (line 2): loop runs at most n times
  - Total: $O(n^3)$

➢ Line 4: **1 step**

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

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

A Graph Theorem: $PATH \in \mathrm{P}$

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

$PATH \in \mathrm{TIME}(\boldsymbol{n}^3)$

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*."

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

(For practical purposes, not a great algorithm, but it's **in P!** i.e., **"solvable"**)

# of steps (worst case) **($\boldsymbol{n}$ = # nodes):**

- Line 1: **1 step**
- Lines 2-3 (loop):
  - Steps/iteration (line 3): max # steps = max # edges = $O(\boldsymbol{n}^2)$
  - # iterations (line 2): loop runs at most $\boldsymbol{n}$ times
  - Total: $O(\boldsymbol{n}^3)$
- Line 4: **1 step**

➢ Total = $1 + 1 + O(\boldsymbol{n}^3) = \boxed{O(\boldsymbol{n}^3)}$

3 Problems in **P**

☑ • A Graph Problem:

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

• A Number Problem:

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

• A CFL Problem:

Every context-free language is a member of P

A Number Theorem: $RELPRIME \in \mathrm{P}$

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

- Two numbers are **relatively prime:** if their gcd = 1
  - gcd(x, y) = largest number that divides both x and y
  - E.g., gcd(8, 12) = $\boxed{}$??
- Brute force (exponential) algorithm deciding $RELPRIME$:
  - Try all of numbers (up to x or y), see if it can divide both numbers

  Q: Why is this exponential?

  HINT: What is a typical "representation" of numbers?

  A: binary numbers

  (if $x = 2^n$, then trying x numbers is exponential in n = the number of digits)
- A gcd algorithm that runs in polynomial time:
  - Euclid's algorithm

A GCD Algorithm for: $RELPRIME \in \mathrm{P}$

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

Modulo
(i.e., **remainder**)

15 mod 8 = 7
17 mod 8 = 1

The Euclidean algorithm E is as follows.

E = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:
2. Assign $x \leftarrow x \bmod y$.
3. Exchange x and y.
4. Output x."

O(n)

cuts x (at least) **in half every loop,** requires: **log** x **loops**

Each number is **cut in half every** *other* **iteration**

Total run time (assume $x > y$): $2\log x = 2\log 2^n = \boldsymbol{O(n)}$,
where n = **number of binary digits in** (ie **length of**) x

3 Problems in **P**

☑ • A Graph Problem:

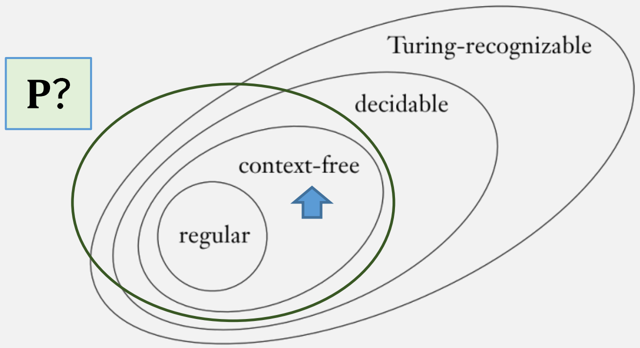
$PATH = \{\langle G, s, t\rangle \mid G$ is a directed graph that has a directed path from s to $t\}$

☑ • A Number Problem:

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

• A CFL Problem:

Every context-free language is a member of P



IF-THEN Statement to Prove:

IF a language L is a CFL,
THEN L is in **P**

Review: A (Decider) TM for Any CFL (hw10 sol)

Given any **CFL** L, with **CFG** G, the following **decider** M_G **decides** L:

$M_G =$ "On input w:
1. Run TM S on input $\langle G, w\rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

S is a decider for: $A_{\text{CFG}} = \{\langle G, w\rangle |\ G \text{ is a CFG that generates string } w\}$

$S =$ "On input $\langle G, w\rangle$, where G is a CFG and w is a string:
1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

M_G is a decider, bc S is a decider

M_G accepts all $w \in L$, for any CFL L (with CFL G)

Therefore, **every CFL is decidable**

But, **is every CFL decidable in poly time?**

A Decider for Any CFL: Running Time

Given any CFL L**,** with **CFG** G**,** the following **decider** M_G **decides** L**:**

$M_G =$ "On input w:
1. Run TM S on input $\langle G, w \rangle$.
2. If this machine accepts, *accept*; if it rejects, *reject*."

$A \to 0A1$
$A \to B$
$B \to \#$

S is a decider for: $A_{\text{CFG}} = \{\langle G, w \rangle | \, G \text{ is a CFG that generates string } w\}$

$S =$ "On input $\langle G, w \rangle$, where G is a CFG and w is a string:
1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n-1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

How many different possibilities at *each* derivation step?

$A \Rightarrow 0A1 \Rightarrow \quad \ldots$

Worst case:
$|R|^{2n-1}$ steps = $O(2^n)$
(R = set of rules)

This algorithm runs in exponential time

A CFL Theorem: Every context-free language is a member of P

- Given a CFL, we must construct a decider for it …
- … that runs in polynomial time

Dynamic Programming

- **Keep track of partial solutions,** and **re-use them**
  - Start with smallest and build up
- **For CFG problem,** instead **of re-generating entire string …**
  - **… keep track of substrings generated by each variable**

$S =$ "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w; except if $n = 0$, then instead list all derivations with one step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

This **duplicates a lot of work** because **many strings** might have the **same beginning derivation steps**

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

Substring end char

Substring start char

	b	a	a	b	a
b					
a					
a					
b					
a					

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

Substring end char

Substring start char

	b	a	a	b	a
b	vars generating "b"	vars for "ba"	vars for "baa"	…	
a		vars for "a"	vars for "aa"	vars for "aab"	
a			…		
b					
a					

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

Algo:
- **For each single char** c **and var** A:
  - **If** A → c **is a rule, add** A **to table**

Substring end char

Substring start char

	b	**a**	**a**	**b**	**a**
b	vars generating "b"	vars for "ba"	vars for "baa"	…	
a		vars for "a"	vars for "aa"	vars for "aab"	
a			…		
b					
a					

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

Algo:
- For each single char c and var A:
  - If A → c is a rule, add A to table

Substring end char

Substring start char

	b	**a**	**a**	**b**	**a**
b	B				
a		A,C			
a			A,C		
b				B	
a					A,C

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating variables in a table

<u>Algo</u>:
- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring s (len > 1):
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B generates x and C generates y

Substring end char

Substring start char

	b	**a**	**a**	**b**	**a**
b	B				
a		A,C			
a			A,C		
b				B	
a					A,C

CFL Dynamic Programming Example

- Chomsky Grammar *G*:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their ge…

Algo:
- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring s:
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B …

For substring "**ba**", split into "**b**" and "**a**":
- For rule S → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO
- For rule S → BC
  - Does B generate "**b**" and C generate "**a**"?
  - **YES**
- For rule A → BA
  - Does B generate "**b**" and A generate "**a**"?
  - **YES**
- For rule B → CC
  - Does C generate "**b**" and C generate "**a**"?
  - NO
- For rule C → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO

Substring end char

Substring start char

	b	**a**	**a**
b	B		
a		A,C	
a			A,C
b			
a			

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their generating

Algo:
- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring s:
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B

For substring "**ba**", split into "**b**" and "**a**":
- For rule S → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO
- For rule S → BC
  - Does B generate "**b**" and C generate "**a**"?
  - **YES**
- For rule A → BA
  - Does B generate "**b**" and A generate "**a**"?
  - **YES**
- For rule B → CC
  - Does C generate "**b**" and C generate "**a**"?
  - NO
- For rule C → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO

Substring end char

Substring start char

	b	**a**	**a**
b	B	S,A	
a		A,C	
a			A,C
b			
a			

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - C
  -
  - B → CC | b
  - C → AB | a
- ing: **baaba**
- partial string and their ge

For each:
- char
- var

For each:
- substring
- split of substring
- rule

Algo:
- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B generates x and C generates y

For each: char, var ...

For each: substring, split, rule ...

Substring end char

Substring start char

	b	a	a	b	a
b	B	S,A			S,A,C
a		A,C	B	B	S,A,C
a			A,C	S,C	B
b				B	S,A
a					A,C

If S is here, accept

A CFG Theorem: Every context-free language is a member of P

$D =$ "On input $w = w_1 \cdots w_n$:

1. For $w = \varepsilon$, if $S \to \varepsilon$ is a rule, *accept*; else, *reject*. [[$w = \varepsilon$ case]]
2. For $i = 1$ to n: [[examine each substring of length 1]]
3. For each variable A:
4. Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$.
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: substring]]
8. Let $j = i + l - 1$. [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$:
10. For each rule $A \to BC$:
11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C, put A in $table(i, j)$.
12. If S is in $table(1, n)$, *accept*; else, *reject*."

For each:
- char
- var

$O(n)$ chars

#vars = constant = $O(1)$

$O(1) * O(n) = O(n)$

For each:
- substring
- split of substring
- rule

$O(n)$ diff lengths

$O(n)$ strings of each length

$O(n)$ ways to split a string into two pieces

#vars = constant = $O(1)$

$O(1) * O(n) * O(n) * O(n) = O(n^3)$

<u>Total</u>: $\boxed{O(n^3)}$

(This is also known as the <u>Earley parsing algorithm</u>)

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

☑ • A Graph Problem:

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

☑ • A Number Problem:

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

☑ • A CFL Problem:

Every context-free language is a member of P

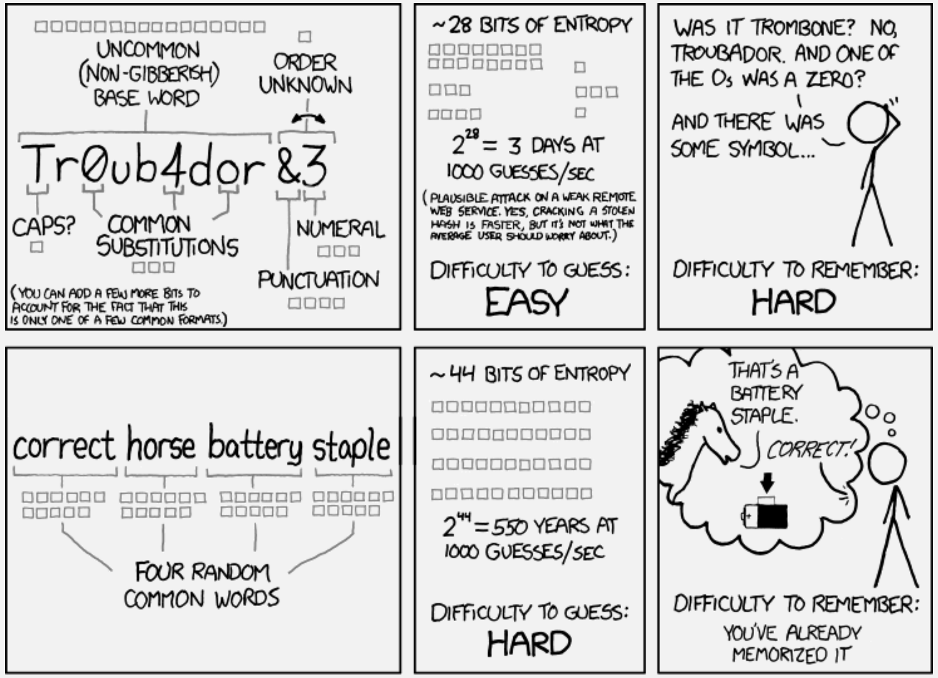
NP

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



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.

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

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 $\boldsymbol{m}$ = 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 = $O(\boldsymbol{m})$
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: $O(\boldsymbol{m})$
   - # loops: $O(\boldsymbol{m})$
   - Total looping time = $O(\boldsymbol{m}^2)$
3. Check "current" edge has "to" node t; if yes accept, else reject ☑

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

Decider …

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)

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.

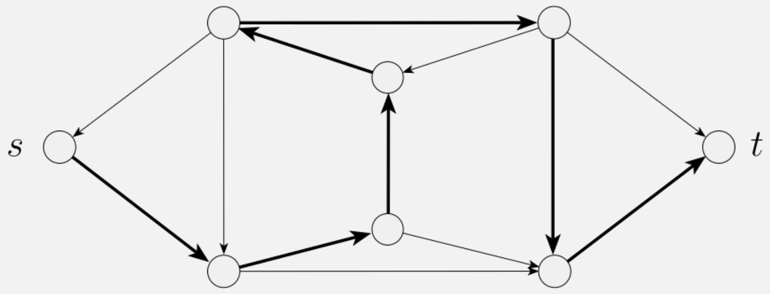
- NOTE: a certificate c must be at most length n^k, where n = length of w
  - Why? Because it takes time n^k to read it

So *PATH* is polynomially verifiable

The *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

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:

On input w =

- Nondeterministically run V with w and all possible poly length certificates c

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

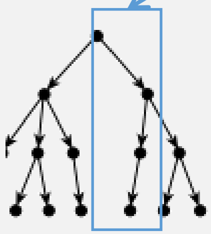
Because Converting NTM to deterministic is exponentially slower!

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

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

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

NP = Nondeterministic 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 = Deterministic polynomial time

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

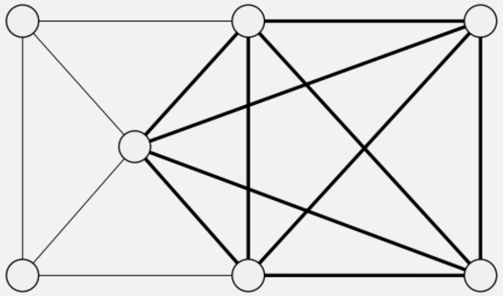
$$\mathrm{NP} = \bigcup_k \mathrm{NTIME}(n^k)$$

Also, **NP** = Deterministic polynomial time **verification**

NP = Nondeterministic polynomial time

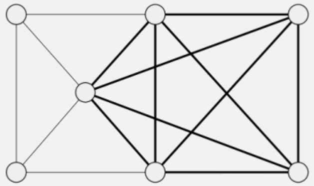
More **NP** Problems

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



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

Theorem: *CLIQUE* is in NP



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

PROOF IDEA The clique is the certificate.

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

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*."

c is at most $\boldsymbol{n}$

`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: $O(\boldsymbol{n}^2)$

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

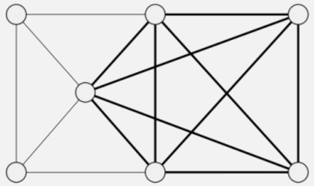
$$A = \{w \mid V \text{ accepts } \langle w, c\rangle \text{ for some string } c\}.$$

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 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"

Checking whether a subgraph is clique: $O(\boldsymbol{n^2})$

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

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

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

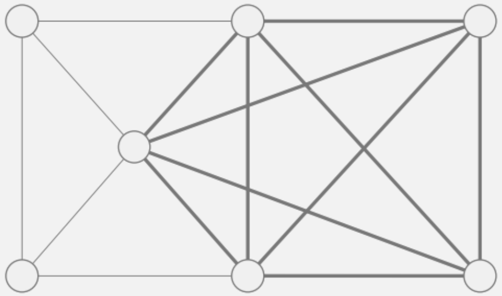
Don't forget to count the steps

THEOREM

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

More **NP** Problems

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



- $SUBSET\text{-}SUM = \{\langle \overset{\text{set}}{S}, \overset{\text{sum}}{t} \rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \overset{\text{subset}}{\{y_1, \ldots, y_l\}} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \Sigma y_i = \overset{\text{sum}}{t}\}$
  - Some **subset** of a **set of numbers** S must **sum to some total** t
  - e.g., $\langle\{\boxed{4}, 11, 16, \boxed{21}, 27\}, 25\rangle \in SUBSET\text{-}SUM$

Theorem: *SUBSET-SUM* is in NP

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

PROOF IDEA The subset is the certificate.

To prove a lang is in **NP**, create <u>either</u>:

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

PROOF The following is a 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

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

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**

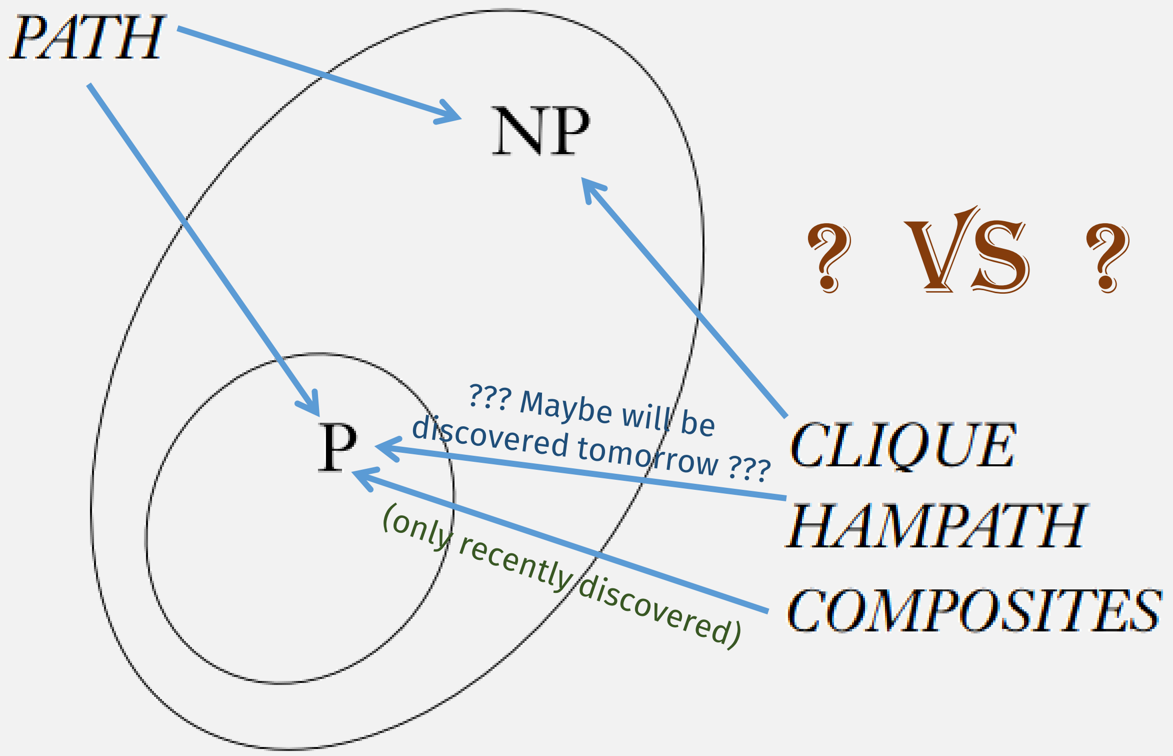
- A **certificate** could be:
  - Some **factor** that **is not 1**

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

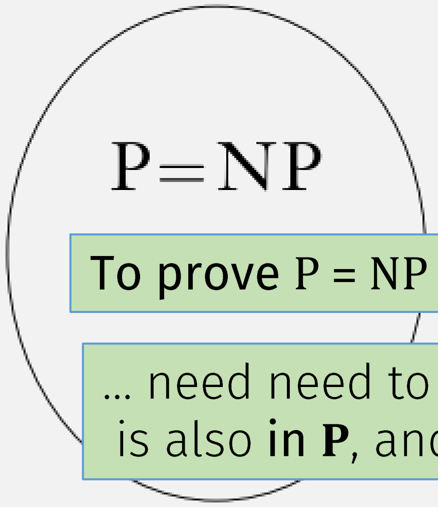
One of the Greatest unsolved

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

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



? VS ?



To prove P = NP …

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

How do you prove an algorithm doesn't have a poly time algorithm?
(in general it's hard to prove that something doesn't exist)

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**
  - **Near perfect AI learning, recognition**
- **<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** ?

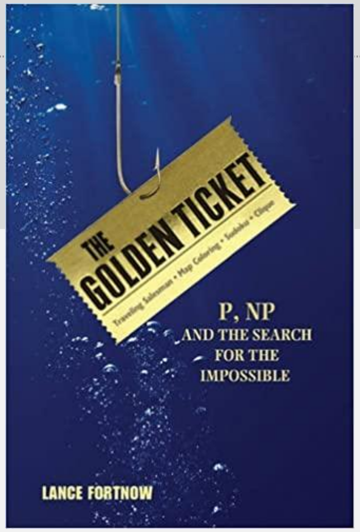
- Some, but still not close

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

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 look at all langs, can't just look at 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?

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

THEOREM

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

Flashback: Mapping Reducibility

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

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

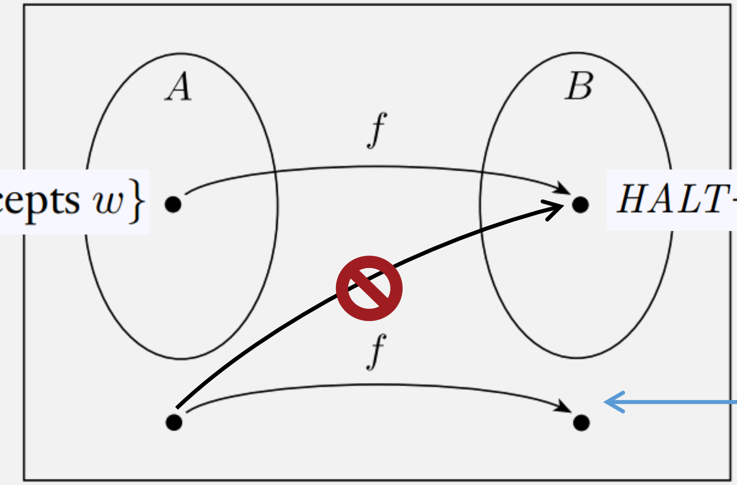
IMPORTANT: "if and only if" ...

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 forward 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 $\boxed{\overline{A} \leq_m \overline{B}}$

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

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 ***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,

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

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

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.

A function $f\colon \Sigma^* \longrightarrow \Sigma^*$ is a *poly time* ***computable function*** if some *poly time* 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

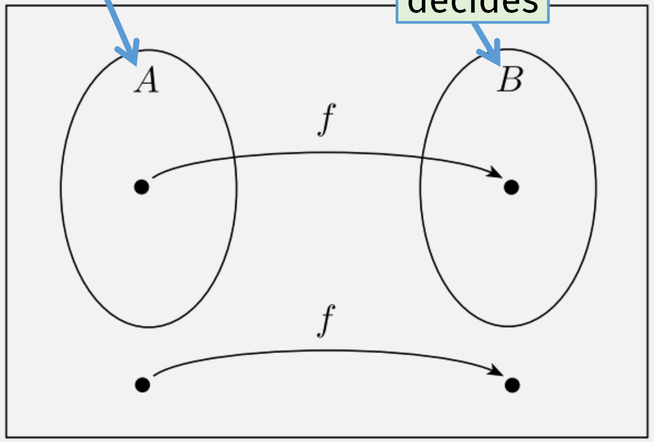
PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

$N =$ "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,

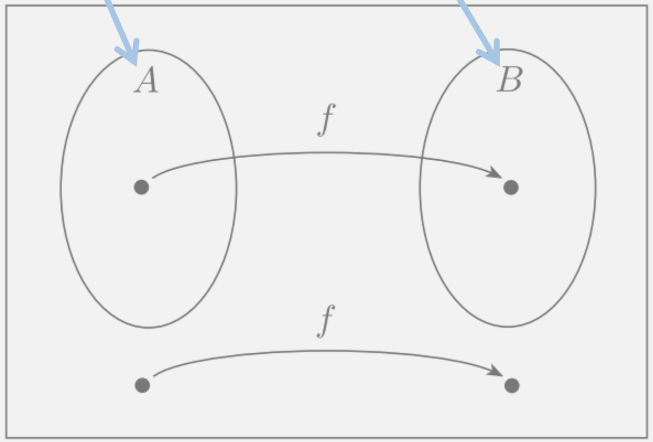
$$\boxed{w \in A \iff f(w) \in B.}$$

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,

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

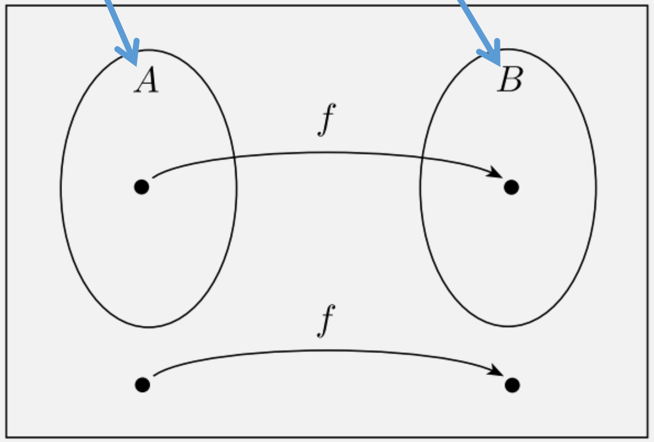
The function f is called the ***reduction*** from A to B.

Thm: If $A \leq_{\cancel{\mathrm{m}}\,\mathbf{P}} B$ 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 $A \leq_{\mathrm{m}} B$, 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.

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

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.

Next: How to do poly time mapping reducibility

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

Next Time: *3SAT* is polynomial time reducible to *CLIQUE*.