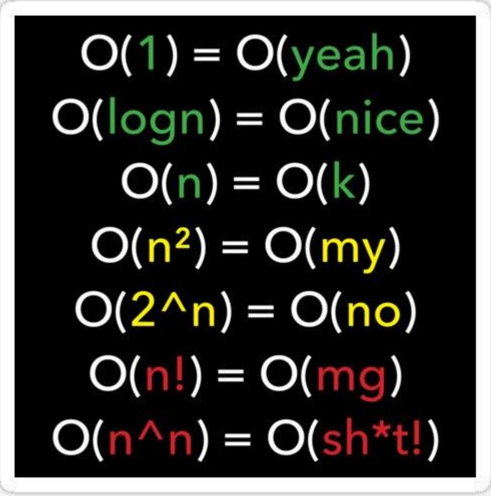
UMB CS622

Polynomial Time (P)

Wednesday November 10, 2021



Announcements

- HW7 due tonight 11:59pm EST
- HW8 out tonight
- FYI: School holiday tomorrow (Thurs)

Last Time: 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,

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

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

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

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:

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

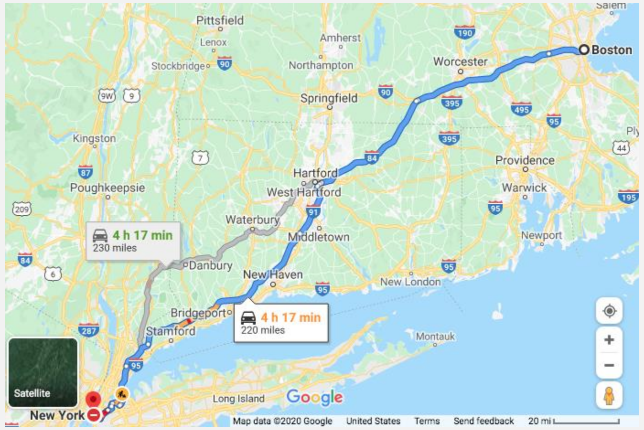
- A CFL Problem:

  Every context-free language is a member of P

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

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

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



- To prove that a language is in **P** …

- … we must construct a <u>polynomial</u> time algorithm deciding the lang

- Languages in **P** can still have <u>non-polynomial</u> (i.e., ”brute force”) algorithms:
  - check all possible paths, and see if any connect s to t
  - If n = # vertices, then # paths ≈ n^n

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

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

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: $PATH \in \mathrm{P}$

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

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

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

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

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: $O(n^3)$

A Graph Theorem: $PATH \in$ 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
  - 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,

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

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

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

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*."

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

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

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

3 Problems in **P**

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

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

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

  Every context-free language is a member of P

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

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

- Two numbers are **relatively prime** if their <u>gcd</u> = 1
  - $\gcd(x, y)$ = largest number that divides both x and y
  - E.g., $\gcd(8, 12) = 4$

- Brute force exponential algorithm deciding *RELPRIME*:
  - Try all of numbers (up to x or y), see if it can divide both numbers
  - Why is this exponential?
  - <u>HINT</u>: What is a typical "representation" of numbers?
  - <u>Answer</u>: binary numbers

- A $\gcd$ algorithm that runs in poly 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**)
cuts x (at least) **in half**

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."

$O(n)$

Each number is cut in half every *other* iteration

Cutting x in half every step requires: **log x** steps

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

3 Problems in $\mathbf{P}$

- A Graph Problem:

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

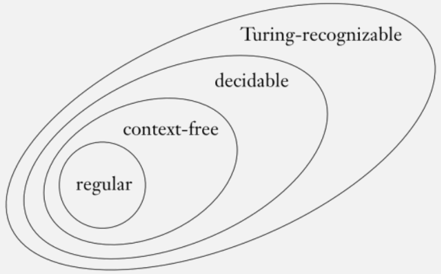
- A Number Problem:

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

- A CFL Problem:

  Every context-free language is a member of P

Review: HW5, Problem 4-1



Prove: the context-free oval is completely contained inside the decidable oval

- I.e., Every context-free language (CFL) is also a decidable language

Proof Plan:

- To prove that a language is decidable … we must construct a decider for it
- To show that every CFL is decidable, we show how to construct a decider for any CFL

To construct our decider, we use the following things learned in this course:

- A language is a set of strings
- A CFL L is a language that … has a CFG (G) and a PDA (P), where:
    - $w \in L \Leftrightarrow G$ generates w, or
    - $w \in L \Leftrightarrow P$ accepts w
- A decider (M) for a CFL L is a TM such that, on input w:
    - M accepts $w \Leftrightarrow G$ generates w, or
    - M accepts $w \Leftrightarrow P$ accepts w

Review: A Decider for Any CFL (HW5)

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 = "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*."

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

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, *rejec*

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

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 algorithm runs in exponential time

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

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

- For CFG problem, instead of re-generating entire string …
  - … keep track of <u>substrings</u> generated by each variable

Dynamic programming

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 <u>duplicates a lot of work</u> because many strings might have have the same first few derivations 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 for "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 for "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

<u>Algo</u>:
- 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

Algo:
- 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[nerating]

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 ge[nerating]

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

Substring end char

Substring start char

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

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

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 char, var …

- 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 generates x and C generates y

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 → S,A,C

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: $O(\boldsymbol{n})$ [[examine each substring of length 1]]
3. For each variable A: *#vars*
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: $O(\boldsymbol{n})$ [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: $O(\boldsymbol{n})$ [[i is the start position of the substring]]
8. Let $j = i + l - 1$. [[j is the end position of the substring]]
9. For $k = i$ to $j - 1$: $O(\boldsymbol{n})$ [[k is the split position]]
10. For each rule $A \to BC$: *#rules*
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

#vars * *n* = $O(\boldsymbol{n})$

For each:
- substring
- split
- rule

#rules * $O(\boldsymbol{n})$ * $O(\boldsymbol{n})$ * $O(\boldsymbol{n})$ = $O(\boldsymbol{n}^3)$

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

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

Summary: 3 Problems in **P**

- A Graph Problem:

  "search" 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

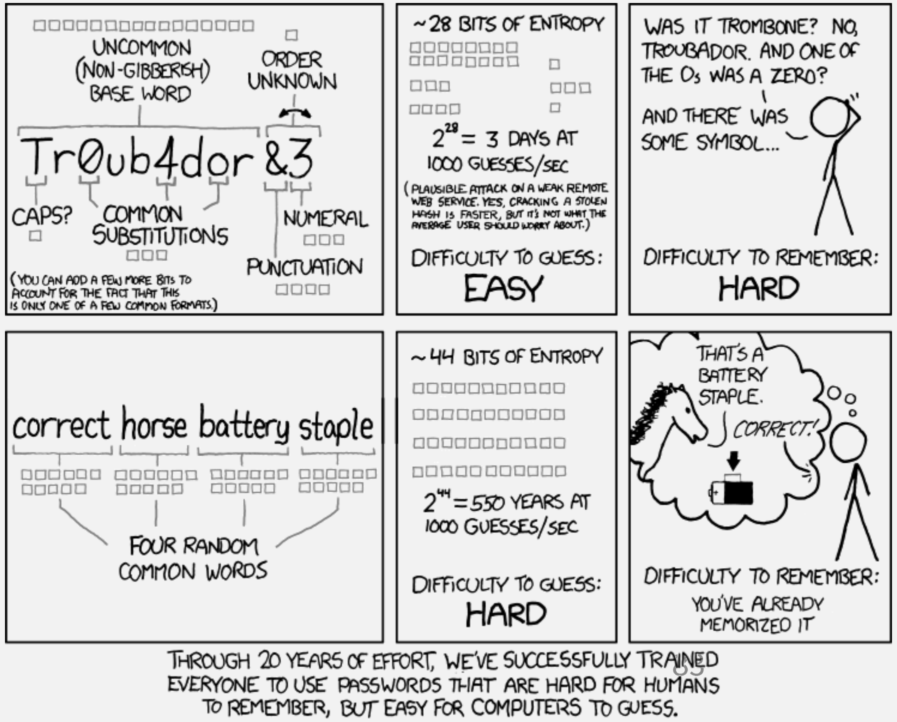
Search vs Verification

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

EXAMPLES

- Factoring
  - **Unsolvable**: <u>Find</u> factors of 8633
  - **Solvable**: <u>Verify</u> 89 and 97 are factors of 8633

- 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 possible paths and see if any connects s and t
  - Polynomial time algorithm:
    - Do a breadth-first search (roughly), marking "seen" nodes as we go

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*."

Verifying a *PATH*

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

The **verification** problem:

- Given some path p in G, check that it is a path from s to t

- Let $\boldsymbol{m}$ = longest possible path = # edges in G

NOTE: extra argument p

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 = $O(\boldsymbol{m}) + O(\boldsymbol{m}^2) = O(\boldsymbol{m}^2)$ = polynomial in m

PATH can be verified in polynomial time

Verifiers, Formally

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

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.

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

certificate, or ***proof***

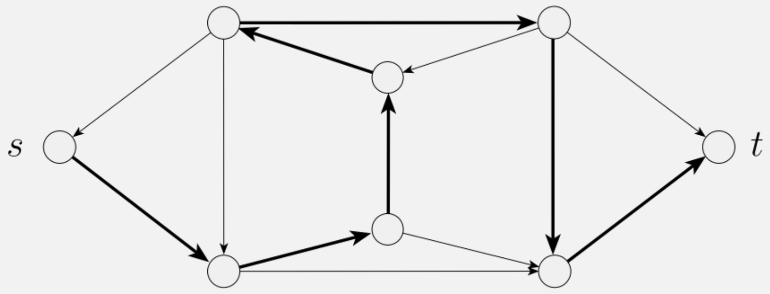
- NOTE: a cert c must be at most length n^k, where n = length of w
  - Why?

So *PATH* is polynomially verifiable

The *HAMPATH* Problem

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

- A Hamiltonian path goes through <u>every</u> 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 $O(\boldsymbol{m^2})$!
  - *HAMPATH* is polynomially verifiable, but <u>not</u> 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 not known 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 =

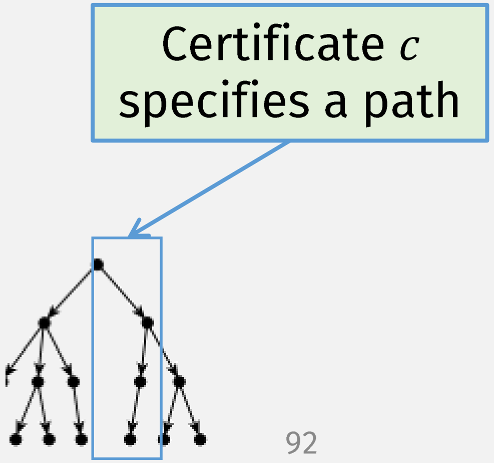
- Nondeterministically run V with w and all possible poly length certificates c

⇐ 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:

On input <w, c> =

- 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 = Nondeterministic polynomial time

NP vs **P**

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

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

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

P = Deterministic polynomial time

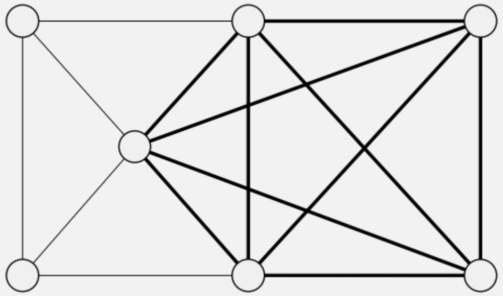
$$\textbf{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}.$$

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

NP = Nondeterministic polynomial time

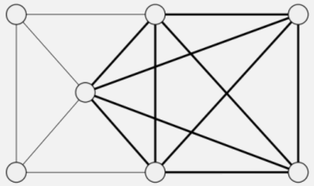
More **NP** Problems

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



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

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 n = # nodes in G

c is at most 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: $O(n^2)$

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

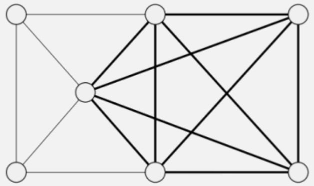
A ***verifier*** for a language A is an algorithm V, where

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

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.

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"

$O(n^2)$

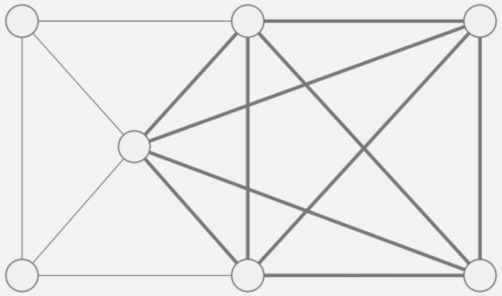
To prove a lang L is in **NP**, create either a:
- **Deterministic** poly time **verifier**
- **Nondeterministic** poly time **decider**

THEOREM

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

More **NP** Problems

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



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

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 subset is the certificate.

To prove a lang is in **NP**, create either:
- **Deterministic** poly time **verifier**
- **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*."

Runtime?

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:

- **Deterministic** poly time **verifier**
- **Nondeterministic** poly time **decider**

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*."

Runtime?

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

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

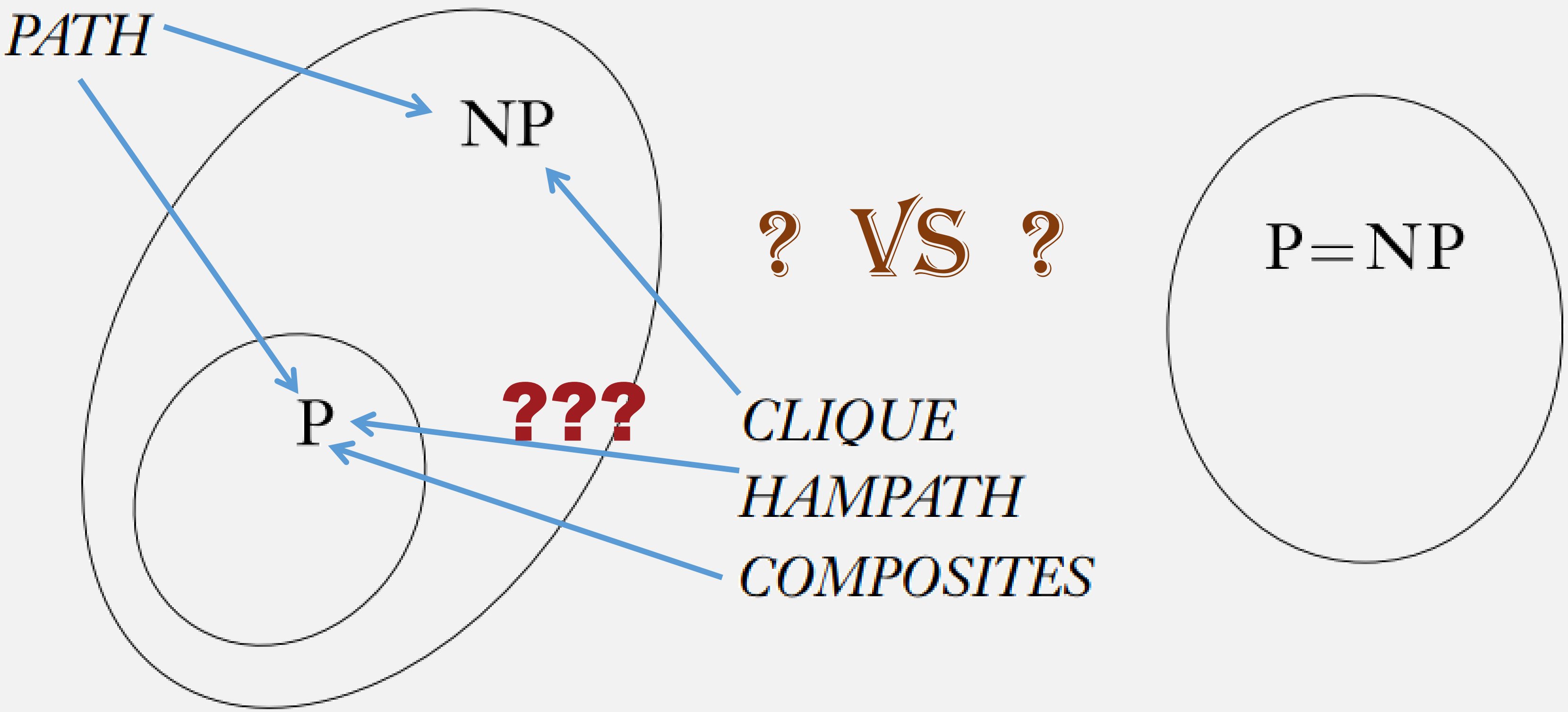
- *COMPOSITES* is polynomially verifiable
  - i.e., it's in **NP**
  - i.e., factorability is in **NP**

- 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 recently (2002)!

One of the Greatest unsolved

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



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**

- Every problem with a "brute force" solution also has an efficient solution
- 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** ?

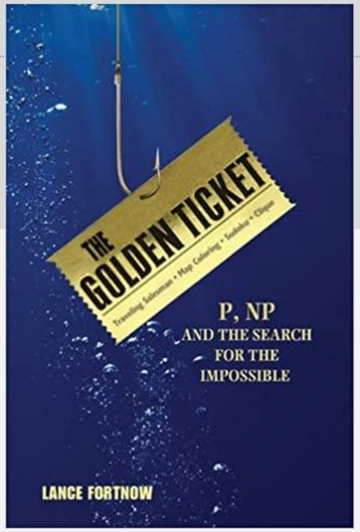
- Some, but still not close

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

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 $B \in$ P, then P $=$ NP.

Check-in Quiz 11/10

On gradescope