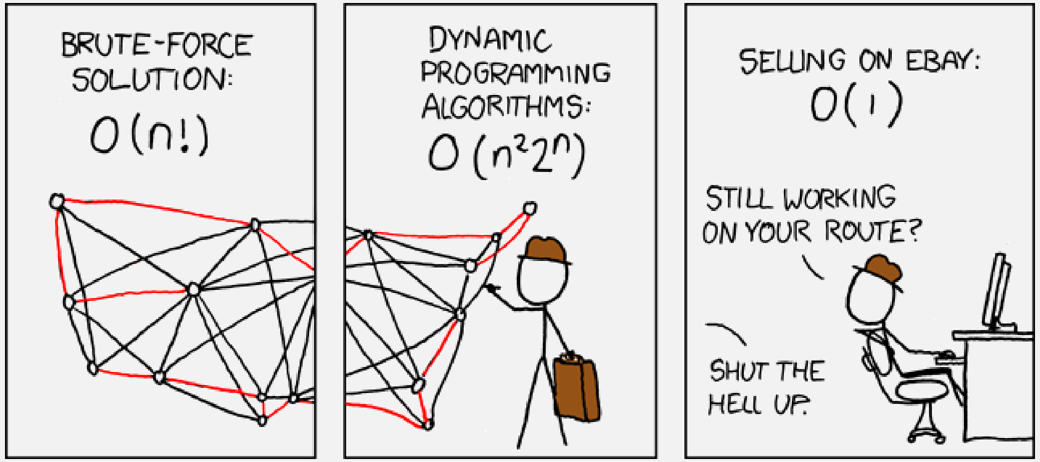
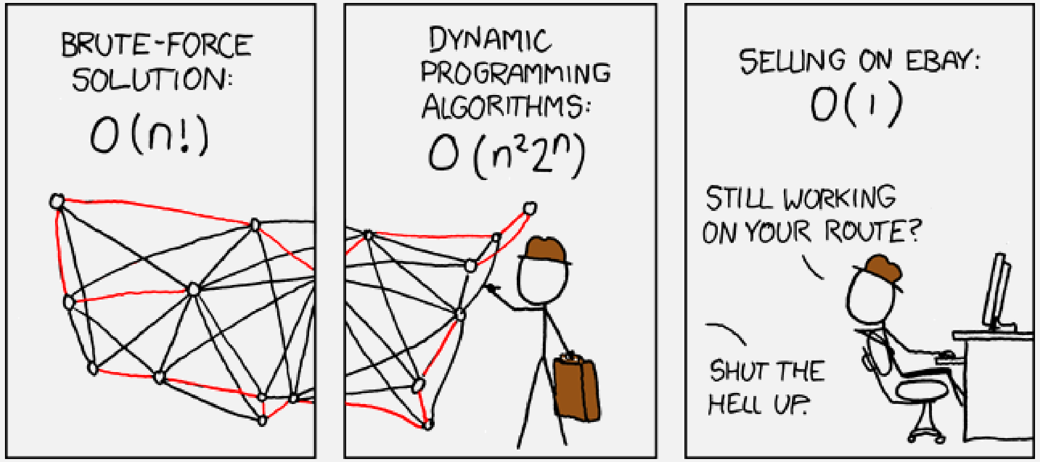


Polynomial Time (P)

Wednesday, April 21, 2021

Announcements



- HW9 past due

- HW10 released
  - Due Tues 4/27 11:59pm EST

- **<u>FAQ</u>**: How can I get better HW scores?
  - To earn more <u>partial credit</u>: show your thought process!
    - Even if you can't figure out the exact answer, show what you do know!
    - Most HW problems simply require basic understanding of class/book concepts
  - **<u>But</u>** … these kinds of answers will receive zero credit:
    - "Throw everything at the wall", i.e., "I will now use every theorem in the book …"
    - Submitting an example copied from the book that is obviously for a different problem

Partial Credit, Concrete Example

<u>Problem</u>: Show that language L is undecidable, where L = ...

<u>A Partial Answer</u> (you can already write most of this *without even reading the rest of the problem!*):

I know:

- To prove undecidability, use <u>proof by contradiction</u>
- A proof by contradiction requires an <u>assumption</u>:
  - Assume language L is decidable
- A decidable language must have a <u>decider</u>, call it R
- Use this decider to create a <u>contradiction</u>:
  - Create a decider for a known undecidable language, A_{TM}
- Decider for A_{TM}, on input <M,w>:
  - We know R distinguishes SOMETHING from SOMETHINGELSE
  - So create M_2, which does SOMETHING if M accepts w, otherwise does SOMETHINGELSE
  - Then give M_2 to R:
    - if R accepts M_2 then M must accept w, so accept, else reject

I couldn't figure out:

- How to make M_2 do SOMETHING if M accepts w
- otherwise do SOMETHINGELSE

This answer would receive almost full credit!

Shows understanding of:
- Decidability and undecidability
- Proper use of proof by contradiction
- Proof techniques used in class examples

Last Time: Time Complexity

DEFINITION 7.1

Let M be a deterministic Turing machine that halts on all inputs. The ***running time*** or ***time complexity*** of M is the function $f\colon \mathcal{N} \longrightarrow \mathcal{N}$, 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.

NOTE: exact units of n not specified, it's only *roughly* "length" of the input

But n can be #characters, #states, #nodes, etc, whatever is more convenient, so long as it's correlated with length of input

It doesn't matter because we only care about large n (so constant factors are ignored)

Last Time: Time Complexity *Classes*

DEFINITION 7.7

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 (ie, running time), **languages** are in a complexity class

The complexity class of a **language** is determined by the time complexity (ie, running time) of their deciding **TMs**

Today: Polynomial Time (**P**) Complexity Class

- Corresponds to **solvable** vs **unsolvable** problems; roughly:
  - Problems in **P** = “solvable”
  - Problems outside **P** = “unsolvable”

- Problems can be “decidable” in theory, but “unsolvable” in practice

Amount of Time to Crack Passwords

“abcdefg” 7 characters	.29 milliseconds
“abcdefgh” 8 characters	5 hours
“abcdefghi” 9 characters	5 days
“abcdefghij” 10 characters	4 months
“abcdefghijk” 11 characters	1 decade
“abcdefghijkl” 12 characters	2 centuries

- Unsolvable problems usually only have “brute force” solutions
  - “try all possible inputs”

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**.

Today: Polynomial Time, Formally

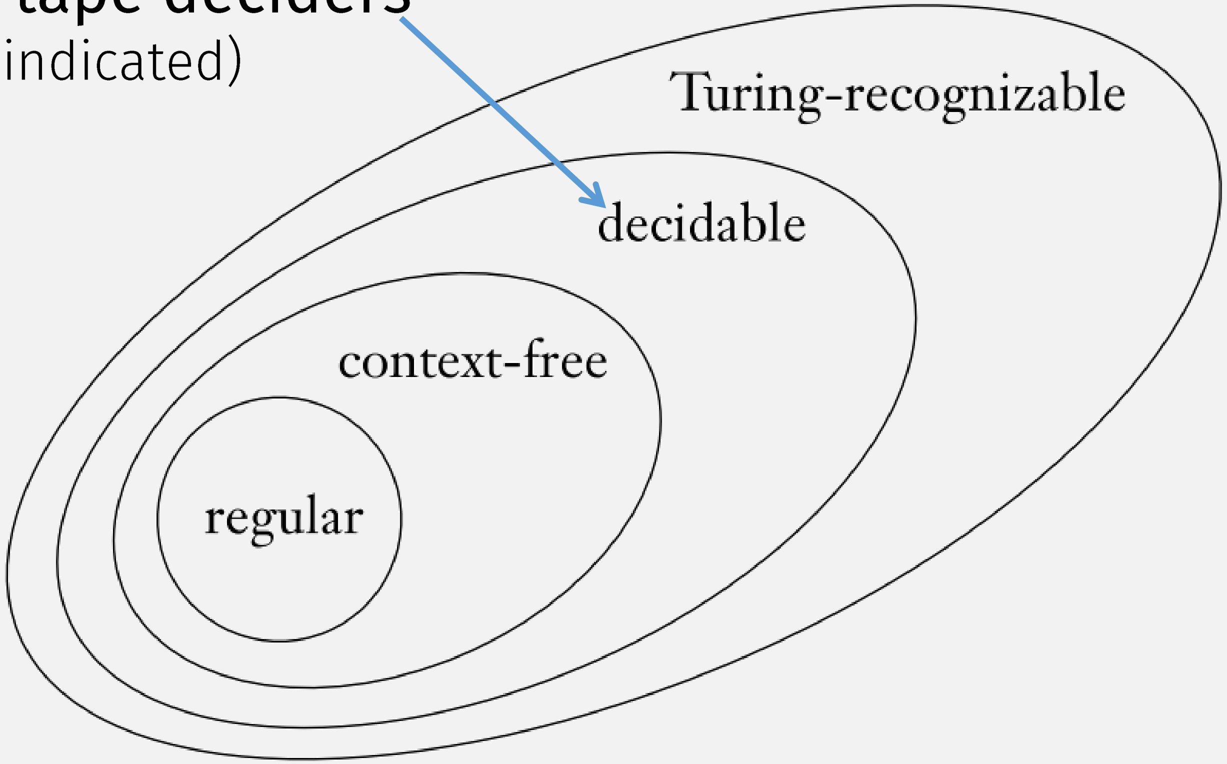
DEFINITION 7.12

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

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

Where Are We Now?

We are back in here now:
deterministic, single-tape deciders
(unless otherwise indicated)



Today: 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 $t\}$

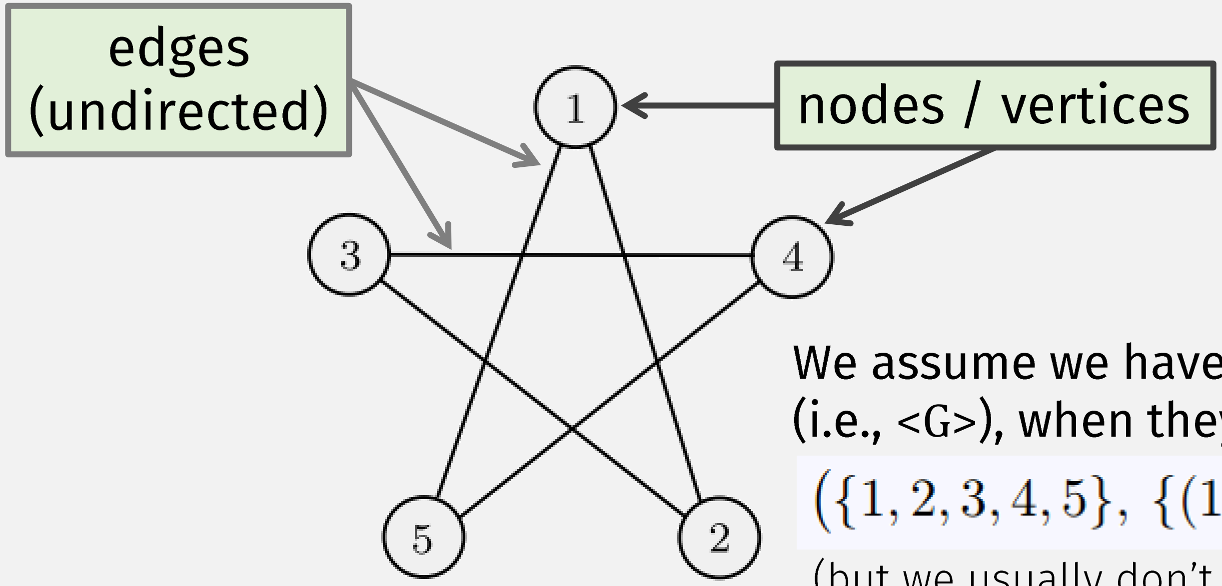
- A Number Problem:

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

- A CFL Problem:

  Every context-free language is a member of P

<u>Interlude</u>: Graphs (see Chapter 0)



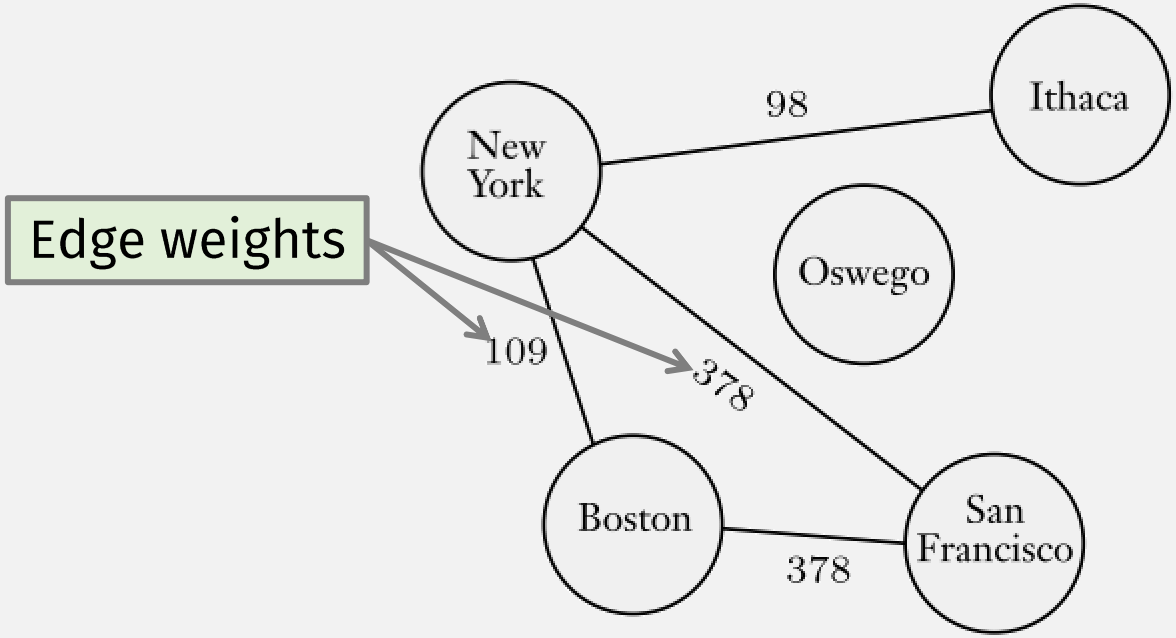
We assume we have *some* **string encoding of a graph** (i.e., <G>), when they are args to TMs, e.g.:

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

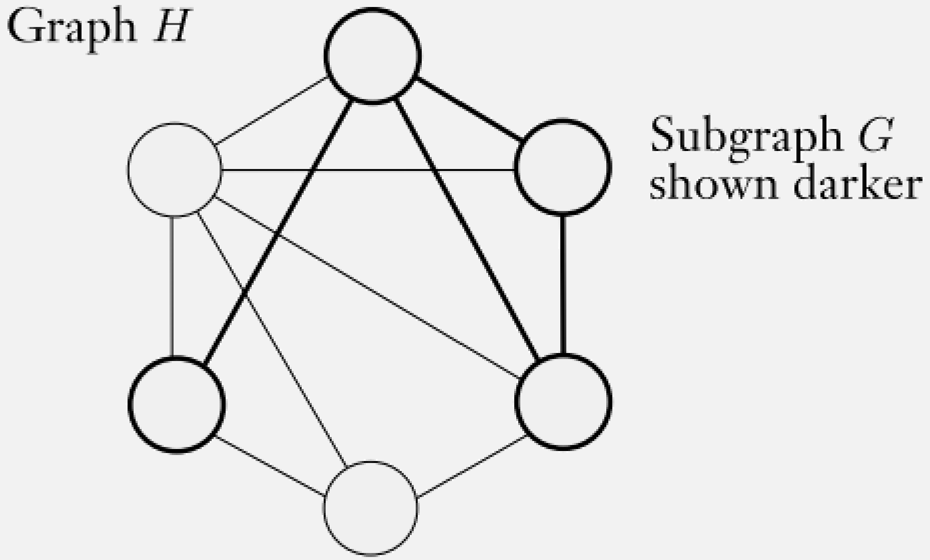
(but we usually don't care about the actual details)

- <u>Edge</u> defined by two <u>nodes</u> (order doesn't matter)
- Formally, a graph = a pair (V, E)
  - Where V = a set of nodes, E = a set of edges

Interlude: Weighted Graphs

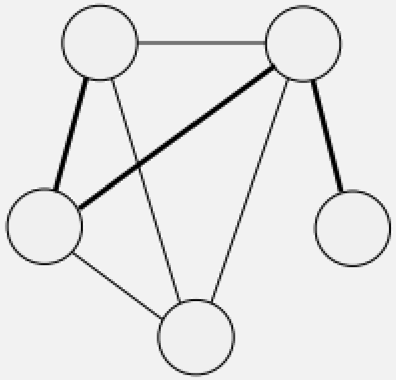


Interlude: Subgraphs

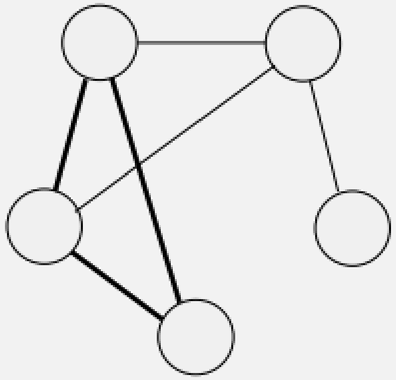


Interlude: Paths and other Graph Things

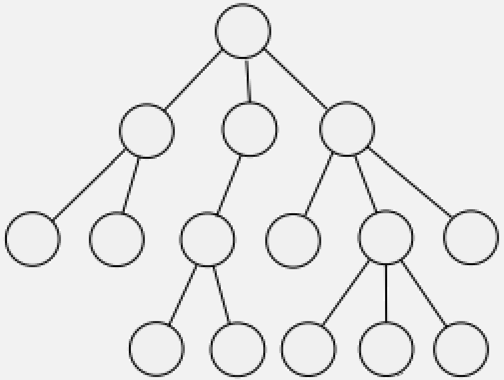
- Path
  - A sequence of nodes connected by edges



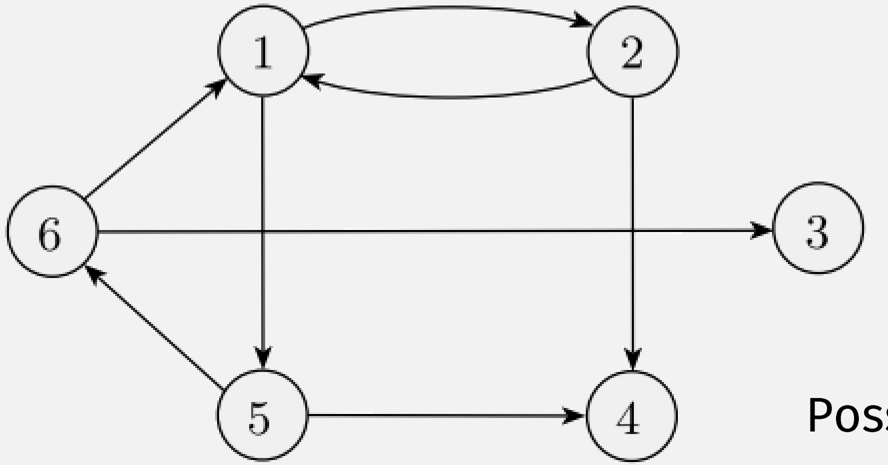
- Cycle
  - A path that starts/ends at the same node



- Connected graph
  - Every two nodes has a path
- Tree
  - A connected graph with no cycles



Interlude: Directed Graphs



Possible **string encoding** given to TMs:

$$(\{1,2,3,4,5,6\},\ \{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\})$$

Each pair of nodes included twice

- Directed graph = (V, E)
  - V = set of nodes, E = set of edges
- An edge is a pair of nodes (u,v), **order now matters**
  - u = “from” node, v = “to” node
- “degree” of a node: number of edges connected to the node
  - Nodes in a directed graph have both indegree and outdegree

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 is usually # of vertices
  - (Not number of chars in the encoding)
- So given graph $G = (V, E)$, $n = |V|$
- Max edges?
  - $= O(|V|^2) = O(\mathbf{n}^2)$
- 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
  - Then L is in $\mathbf{P}$

Today: 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:

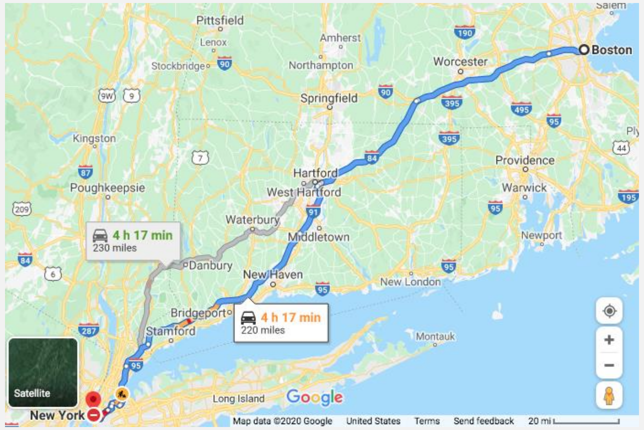
  $RELPRIME = \{\langle x, y\rangle \mid x \text{ and } y \text{ are relatively prime}\}$

- A CFL Problem:

  Every context-free language is a member of P

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

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



- To prove that a language is in **P** …
- … we must construct a <u>polynomial</u> time algorithm deciding the lang
- A <u>non-polynomial</u> (i.e., exponential,"brute force") algorithm:
  - check all possible paths, and see if any connect s to t
  - If n = # vertices, then # paths $\approx n^n$

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**

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**
- Lines 2, 3 (loop):
  - Steps per loop: 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 per loop: max # steps = max # edges = $O(n^2)$
  - ➢ # loops: loop runs at most n times

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**
- Lines 2, 3 (loop):
  - Steps per loop: max # steps = max # edges = $O(n^2)$
  - # loops: loop runs at most n times
  - ➢ Total: $O(n^3)$

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 per loop: max # steps = max # edges = $O(n^2)$
  - # loops: loop runs at most n times
  - Total: $O(n^3)$
- ➢ Line 4: **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 per loop: max # steps = max # edges = $O(n^2)$
  - # loops: loop runs at most n times
  - Total: $O(n^3)$
- Line 4: **1 step**

➢ Total = 1 + 1 + $O(n^3)$ = $O(n^3)$

DEFINITION 7.12

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

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

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

- 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

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

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

- 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) = 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?
  - HINT: What is a typical "representation" of numbers?
  - Answer: binary numbers
- Need gcd algorithm that runs in poly time
  - E.g., Euclid's algorithm

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

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

Modulo (i.e., remainder) cuts x at least in half, e.g.,

- *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."

Each number is cut in half every *other* iteration

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

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

Today: 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

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

- Given a CFL A, can we decide membership in poly time?
- I.e., given grammar G and program w is there a poly time parsing algo?
- Decider for A:

From **Theorem 4.9**

Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

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*."

From **Thm 4.7**

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**

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

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

<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	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

<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:
  - 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:
  - 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:
  - 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	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: **$O(\mathbf{n})$** [[examine each substring of length 1]]
3. For each variable A: **#vars**
4. Test whether $A \to \mathtt{b}$ is a rule, where $\mathtt{b} = w_i$. **#vars * n = $O(\mathbf{n})$**
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: **$O(\mathbf{n})$** [[l is the length of the substring]]
7. For $i = 1$ to $n - l + 1$: **$O(\mathbf{n})$** [[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 $j - 1$: **$O(\mathbf{n})$** [[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 $table(i, j)$. **#rules * $O(\mathbf{n})$ * $O(\mathbf{n})$ * $O(\mathbf{n})$ = $O(\mathbf{n^3})$**
12. If S is in $table(1, n)$, *accept*; else, *reject*."

- Total: $O(\mathbf{n^3})$
- This is also known as the Earley parsing algorithm

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

- 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

Check-in Quiz 4/21

On gradescope