

UMB CS420

Polynomial Time (P)

Monday, May 1, 2023

$O(1) = O(\text{yeah})$
 $O(\log n) = O(\text{nice})$
 $O(n) = O(k)$
 $O(n^2) = O(\text{my})$
 $O(2^n) = O(\text{no})$
 $O(n!) = O(\text{mg})$
 $O(n^n) = O(\text{sh*t!})$

Announcements

- HW 10 in
 - ~~Due Sunday 4/30 11:59pm~~
- HW 11 out
 - Due Sunday 5/7 11:59pm

Quiz Preview

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

Last Time: 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 $f: \mathcal{N} \rightarrow \mathcal{N}$, 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 $t: \mathcal{N} \rightarrow \mathcal{R}^+$ 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

A language can have multiple deciding TMs, 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,

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

- 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

Brute-force attack

From Wikipedia, the free encyclopedia

In **cryptology**, 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**.

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

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**)?

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:

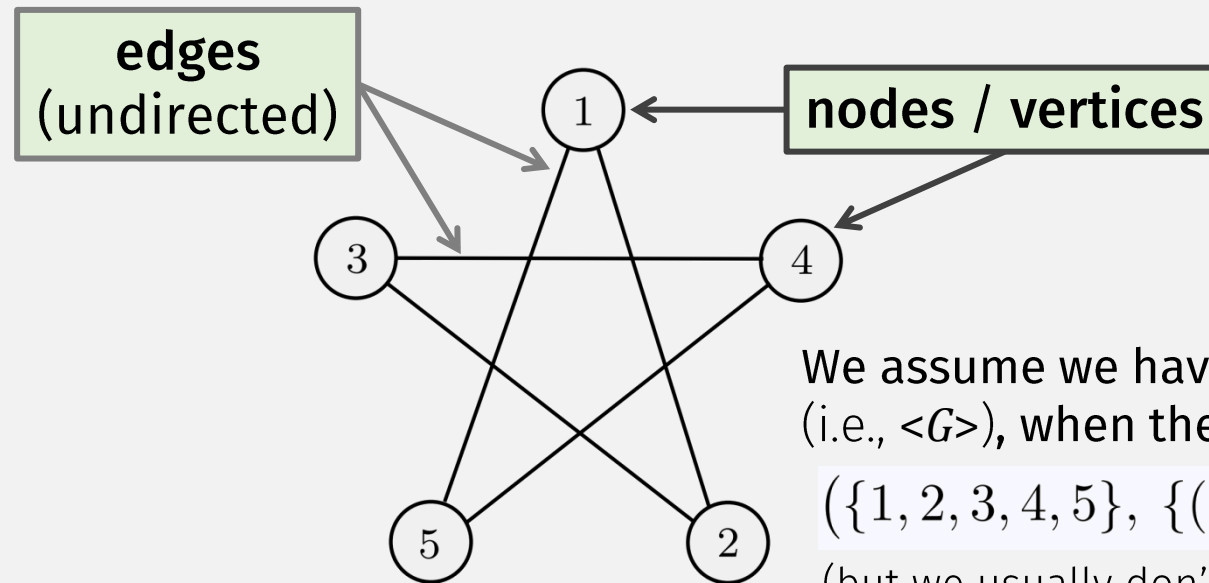
$RELPRIME = \{\langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime}\}$

- A CFL Problem:

Every context-free language is a member of P

- 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: Graphs (see Sipser Chapter 0)



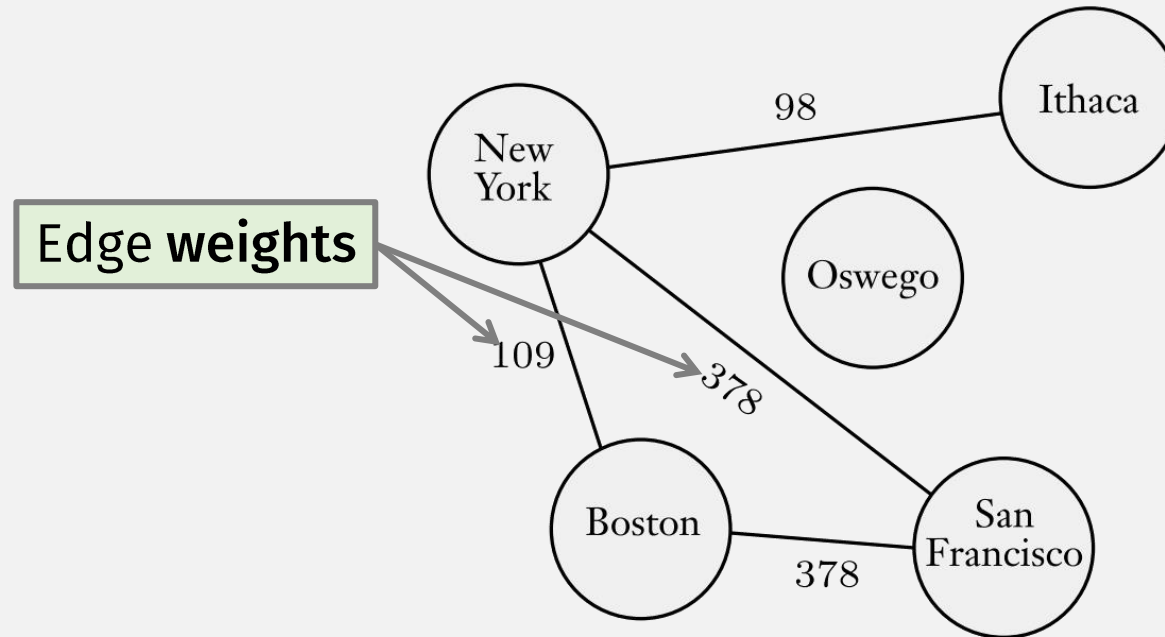
We assume we have some string encoding of a graph (i.e., $\langle G \rangle$), when they are args to TMs, e.g.:

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

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

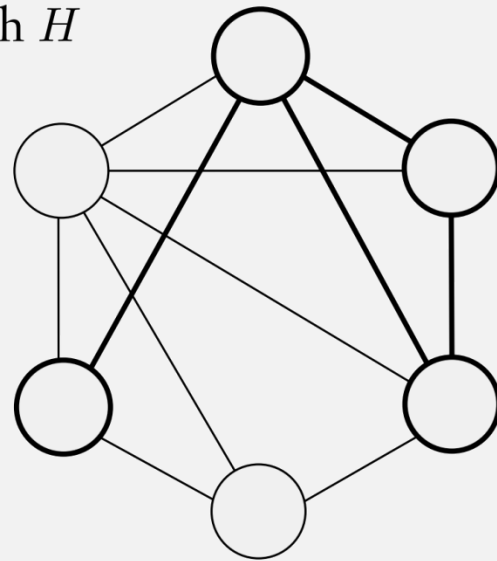
- **Edge** defined by two **nodes** (order doesn't matter)
- Formally, a **graph** = a pair (V, E)
 - Where V = a set of nodes, E = a set of edges

Interlude: Weighted Graphs



Interlude: Subgraphs

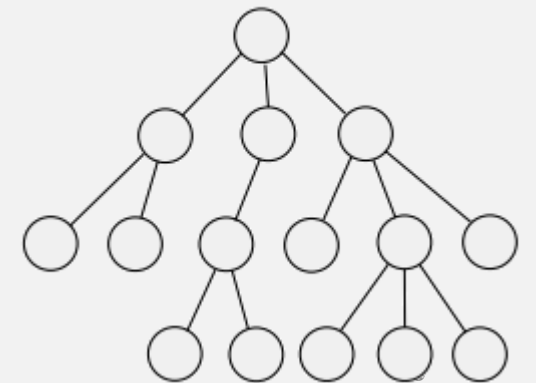
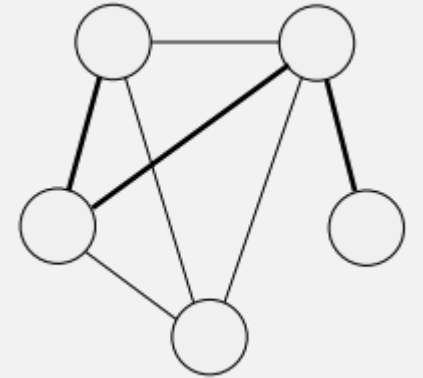
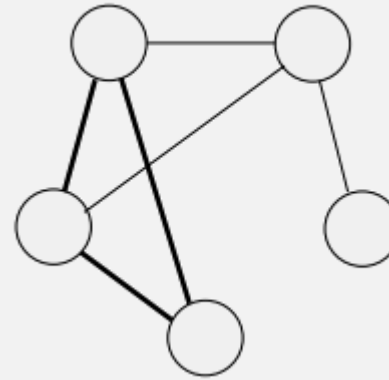
Graph H



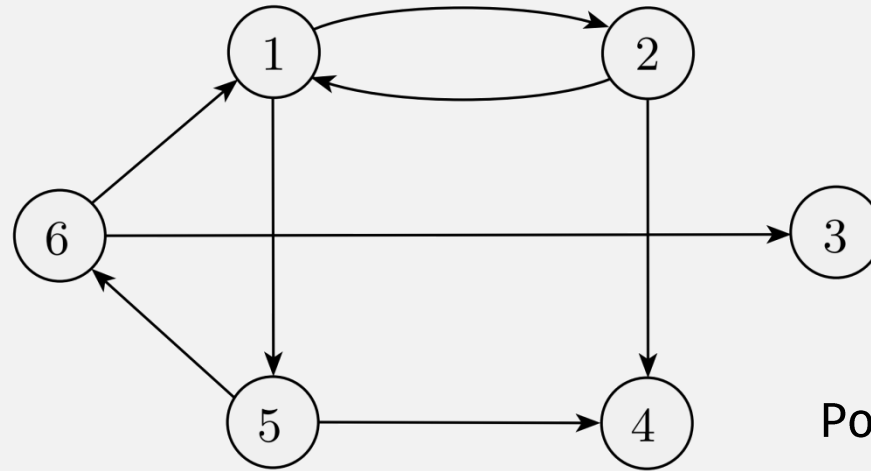
Subgraph G
shown darker

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:

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

- **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

Each pair of nodes
included twice

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 **P**

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

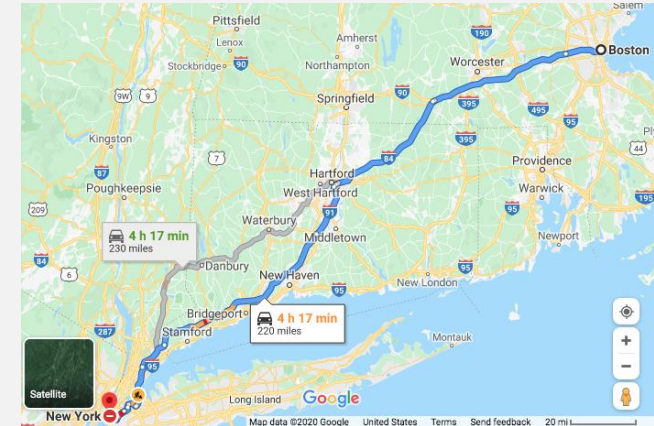
P is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,

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

A Graph Theorem: $PATH \in P$

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

(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 non-polynomial (i.e., "brute force") algorithm:
 - check all possible paths,
 - see if any connect s to t
 - If $n = \#$ vertices, then $\#$ paths $\approx n^n$

A Graph Theorem: $PATH \in P$

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

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: $PATH \in P$

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

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 = $O(n^2)$

A Graph Theorem: $PATH \in P$

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

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: $PATH \in P$

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

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*.”

(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: $O(n^3)$

A Graph Theorem: $PATH \in P$

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

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: $O(n^3)$
- 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: $PATH \in P$

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

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*.”

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

$O(n^3)$

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
- Total = $1 + 1 + O(n^3) = O(n^3)$

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: $RELPRIME \in P$

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

- Two numbers are **relatively prime**: if their $\text{gcd} = 1$
 - $\text{gcd}(x, y)$ = largest number that divides both x and y
 - E.g., $\text{gcd}(8, 12) = ??$
- 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: $RELPRIME \in P$

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

Modulo
(i.e., remainder)

$15 \bmod 8 = 7$
 $17 \bmod 8 = 1$

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

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

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 **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

Review: A Decider for Any CFL

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: $A_{CFG} = \{\langle G, w \rangle \mid G \text{ is a CFG that generates string } w\}$

$S =$ “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*.”

$A \rightarrow 0A1$
 $A \rightarrow B$
 $B \rightarrow \#$

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

$S =$ “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?

$A \Rightarrow 0A1 \Rightarrow \dots$

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

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating variables in a table

Substring end char

	b	a	a	b	a
b					
a					
a					
b					
a					

Substring start char

CFL Dynamic Programming Example

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

Substring end 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					

Substring start char

CFL Dynamic Programming Example

- Chomsky Grammar G :

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating variables in a table

Algo:

- For each single char c and var A :
 - If $A \rightarrow c$ is a rule, add A to table

Substring end char

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

Substring start char

CFL Dynamic Programming Example

- Chomsky Grammar G :

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating variables in a table

Algo:

- For each single char c and var A :
 - If $A \rightarrow c$ is a rule, add A to table

Substring end char

	b	a	a	b	a
b	B				
a		A,C			
a			A,C		
b				B	
a					A,C ₇₈

Substring start char

CFL Dynamic Programming Example

- Chomsky Grammar G :

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating variables in a table

Algo:

- For each single char c and var A :
 - If $A \rightarrow c$ is a rule, add A to table
- For each substring s ($\text{len} > 1$):
 - For each split of substring s into x,y :
 - For each rule of shape $A \rightarrow BC$:
 - Use table to check if B generates x and C generates y

Substring end char

	b	a	a	b	a
b	B				
a		A,C			
a			A,C		
b				B	
a					A,C ₉

Substring start char

CFL Dynamic Programming Example

- Chomsky Grammar G :

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating

Substring end char

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

Substring start char

Algo:

- For each single char c and var A :
 - If $A \rightarrow c$ 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 $A \rightarrow BC$:
 - use table to check if B

For substring "ba", split into "b" and "a":

- For rule $S \rightarrow AB$
 - Does A generate "b" and B generate "a"?
 - NO
- For rule $S \rightarrow BC$
 - Does B generate "b" and C generate "a"?
 - YES
- For rule $A \rightarrow BA$
 - Does B generate "b" and A generate "a"?
 - YES
- For rule $B \rightarrow CC$
 - Does C generate "b" and C generate "a"?
 - NO
- For rule $C \rightarrow AB$
 - Does A generate "b" and B generate "a"?
 - NO

CFL Dynamic Programming Example

- Chomsky Grammar G :

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating

Algo:

- For each single char c and var A :
 - If $A \rightarrow c$ 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 $A \rightarrow BC$:
 - use table to check if B

Substring end char

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

Substring start char

For substring "ba", split into "b" and "a":

- For rule $S \rightarrow AB$
 - Does A generate "b" and B generate "a"?
 - NO
- For rule $S \rightarrow BC$
 - Does B generate "b" and C generate "a"?
 - YES
- For rule $A \rightarrow BA$
 - Does B generate "b" and A generate "a"?
 - YES
- For rule $B \rightarrow CC$
 - Does C generate "b" and C generate "a"?
 - NO
- For rule $C \rightarrow AB$
 - Does A generate "b" and B generate "a"?
 - NO

CFL Dynamic Programming Example

- Chomsky Grammar G :

- For each: C
- char
- var
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

Algo:

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

For each:

- substring
- split of substring
- rule

ing: **baaba**

partial string and their generating variables in a table

Substring end 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

Substring start char

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

$D =$ “On input $w = w_1 \cdots w_n$:

- For each:
- char
- var
1. For $w = \varepsilon$, if $S \rightarrow \varepsilon$ is a rule, *accept*; else, *reject*. $\llbracket w = \varepsilon$ case \rrbracket
 2. \rightarrow For $i = 1$ to n : $O(n)$ chars \llbracket examine each substring of length 1 \rrbracket
 3. \rightarrow For each variable A : #vars = constant = $O(1)$
 4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$. $O(1) * O(n) = O(n)$
 5. If so, place A in $table(i, i)$.
 6. \rightarrow For $l = 2$ to n : $O(n)$ diff lengths $\llbracket l$ is the length of the substring \rrbracket
 7. \rightarrow For $i = 1$ to $n - l + 1$: $O(n)$ strings of each length substring
 8. Let $j = i + l - 1$. $\llbracket j$ is the end position of the substring \rrbracket
 9. \rightarrow For $k = i$ to $j - 1$: $O(n)$ ways to split a string into two pieces
 10. \rightarrow For each rule $A \rightarrow BC$: #vars = constant = $O(1)$
 11. If $table(i, k)$ contains B and $table(k + 1, j)$ contains C , put A in $table(i, j)$. $O(1) * O(n) * O(n) * O(n) = O(n^3)$
 12. If S is in $table(1, n)$, *accept*; else, *reject*.

Total: $O(n^3)$

(This is also known as the Earley parsing algorithm)

Summary: 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

Check-in Quiz 5/1

On gradescope