UMB CS420
Polynomial Time (P)
Thursday, December 1, 2022

O(1) = O(yeah)
O(logn) = O(nice)
O(n) = O(k)
O(n²) = O(my)
O(2^n) = O(no)
O(n!) = O(mg)
O(n^n) = O(sh*t!)

Announcements

- HW 10 out
  - Due Monday 12/5 11:59pm

- HW 11
  - Out Tuesday 12/6
  - Due Monday 12/12 11:59pm

- HW 12
  - Out Tuesday 12/13
  - Due Monday 12/20 11:59pm

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 $f: \mathcal{N} \longrightarrow \mathcal{N}$, 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 <u>large</u> n"

Let $t: \mathcal{N} \longrightarrow \mathcal{R}^+$ 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.

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

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

A <u>language</u> can have <u>multiple</u> deciding TMs, so could be in <u>multiple</u> **time complexity classes**

40

The Polynomial Time Complexity Class ($\mathbf{P}$)

$\mathbf{P}$ 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 $\mathbf{P}$
    - = "**solvable**" or "**tractable**"
  - Problems outside $\mathbf{P}$
    - = "**unsolvable**" or "**intractable**"

"Unsolvable" Problems

Mathematicians are weird.

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

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

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**.

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**)?

42

3 Problems in **P**

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

  $PATH = \{\langle G, s, t \rangle | \ G \text{ is a directed graph that has a directed path from } s \text{ to } t\}$
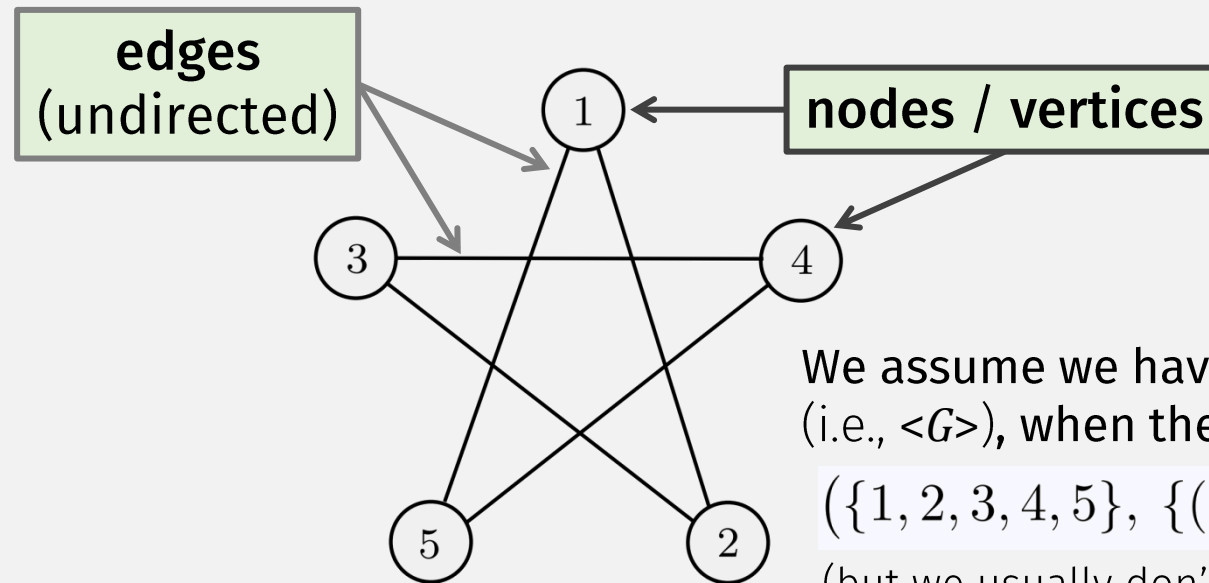
- A <u>Number</u> Problem:

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

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

  Every context-free language is a member of P

- 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 …
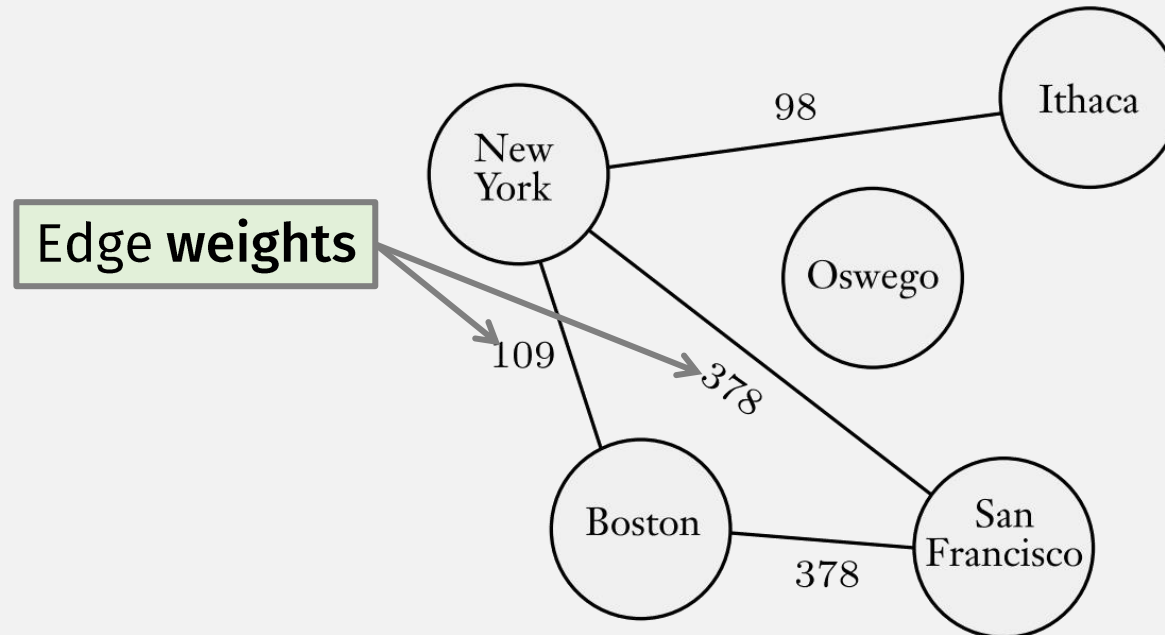
Interlude: Graphs (see Sipser Chapter 0)

edges
(undirected)

nodes / vertices

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

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

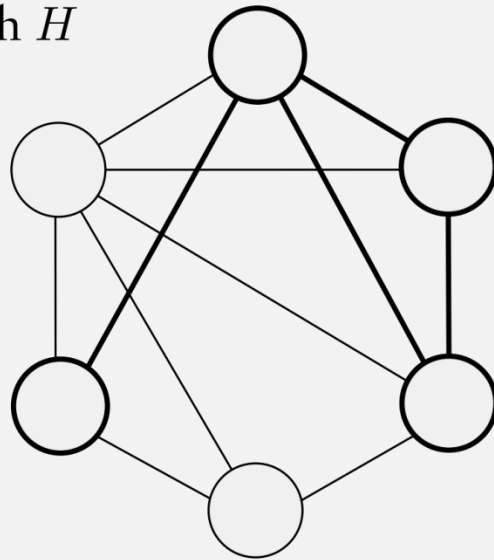(but we usually don't care about the actual details)

- **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
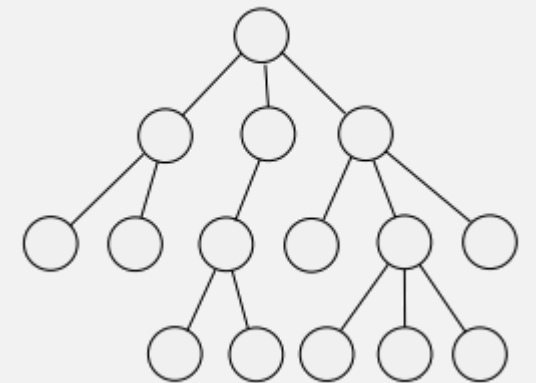
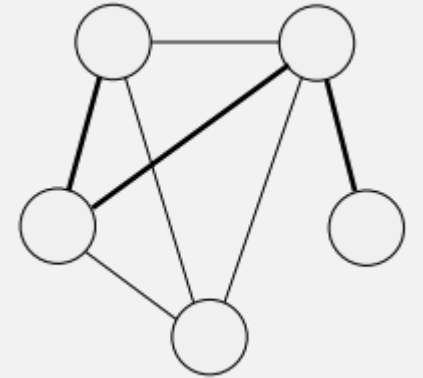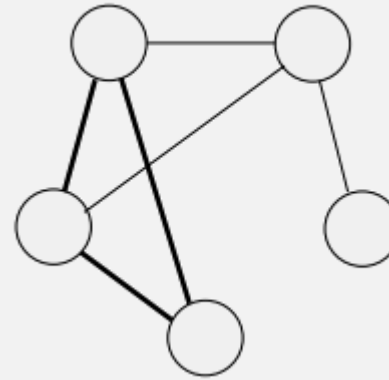# Interlude: Weighted Graphs

Interlude: Subgraphs
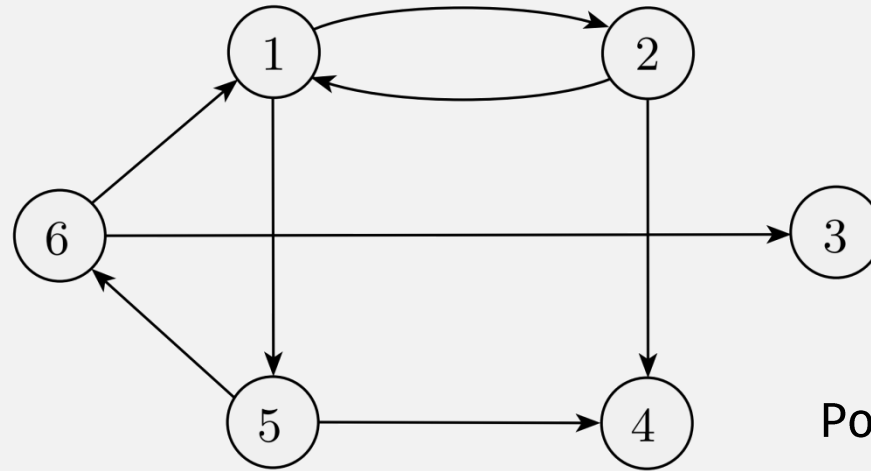

Graph H

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 <u>string encoding</u> 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), <u>*order now matters*</u>
  - u = "from" node, v = "to" node

Each pair of nodes included twice

- "degree" of a node: number of edges connected to the node
  - Nodes in a directed graph have both indegree and outdegree

50

<u>Interlude</u>: Graph Encodings

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

- For <u>graph algorithms</u>, "length of input" n usually = # of vertices
  - (<u>Not</u> number of chars in the encoding)
- So **given graph** $G = (V, E)$, $n = |V|$
- **Max edges?**
  - $= O(|\boldsymbol{V}|^2) = O(\boldsymbol{n}^2)$
- 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 <u>Graph</u> Problem:

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

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

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

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

Every context-free language is a member of P

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

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

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

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

53

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*."

# of steps (worst case) (**n** = # nodes):
➢ Line 1: **1 step**

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*."

# 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: $PATH \in \mathrm{P}$

$PATH = \{\langle G, s, t\rangle |\ G$ 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 = $O(\boldsymbol{n^2})$
  - ➢ # iterations (line 2): loop runs at most **n** times

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

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

$PATH = \{\langle G, s, t \rangle | \ G$ 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*."

# 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 **n** times
  - Total: $O(\boldsymbol{n^3})$
- ➢ Line 4: **1 step**

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*."

$O(n^3)$

(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 = $O(n^2)$
    - # 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 <u>Graph</u> Problem:

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

• A <u>Number</u> Problem:

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

• A <u>CFL</u> 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) = **??**

- 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: $RELPRIME \in P$

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

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

$O(n)$

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 $x \leftarrow x \mod y$.
3.   Exchange x and y.
4. Output x."

15 mod 8 = 7
17 mod 8 = 1

Each number is
cut in half every
other iteration

Cutting x in half
every step requires:
log x steps

Total run time (assume $x > y$): 2log x = 2log2^n = $O(n)$,
where n = number of binary digits in (ie length of) x

66

3 Problems in **P**

☑ • A <u>Graph</u> Problem:

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

☑ • A <u>Number</u> Problem:

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

• A <u>CFL</u> 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: $A_{\mathsf{CFG}} = \{\langle G, w \rangle \mid G$ 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

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*."

S is a decider for: $A_{\mathsf{CFG}} = \{\langle G, w \rangle \mid G$ 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*

This algorithm runs in exponential time

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

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 <u>substrings</u> 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 <u>duplicates a lot of work</u> because many strings might have the <u>same beginning derivation steps</u>

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

74

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 <u>end</u> char

Substring <u>start</u> 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					

75

CFL Dynamic Programming Example

- **Chomsky Grammar** G:
  - $S \to AB \mid BC$
  - $A \to BA \mid a$
  - $B \to CC \mid b$
  - $C \to AB \mid a$

- 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

Substring end char

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

Substring start char

CFL Dynamic Programming Example

- **Chomsky Grammar** G:
  - $S \to AB \mid BC$
  - $A \to BA \mid a$
  - $B \to CC \mid b$
  - $C \to AB \mid a$

- Example string: **baaba**

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

Algo:
- **For** each single char c and var A:
  - **If** $A \to c$ 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 $A \to BC$:
      - 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:
  - $S \rightarrow AB \mid BC$
  - $A \rightarrow BA \mid a$
  - $B \rightarrow CC \mid b$
  - $C \rightarrow AB \mid a$

- Example string: **baaba**

- Store every partial string and their generating

Substring end char

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

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

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

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

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

- 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
  - **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: 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: $O(n)$ ⟦examine each substring of length 1⟧
3. For each variable A: #vars
4. Test whether $A \to b$ is a rule, where $b = w_i$. #vars * n = $O(n)$
5. If so, place A in $table(i, i)$.
6. For $l = 2$ to n: $O(n)$ ⟦l is the length of the substring⟧
7. For $i = 1$ to $n - l + 1$: $O(n)$ ⟦he start position of the substring⟧
8. Let $j = i + l - 1$. ⟦j is the end position of the substring⟧
9. For $k = i$ to $j - 1$: $O(n)$ ⟦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 * $O(n)$ * $O(n)$ * $O(n)$ = $O(n^3)$
12. If S is in $table(1, n)$, *accept*; else, *reject*.

For each:
- char
- var

For each:
- substring
- split
- rule

Total: $O(n^3)$

(This is also known as the Earley parsing algorithm)

Summary: 3 Problems in **P**

☑ • A <u>Graph</u> Problem:

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

☑ • A <u>Number</u> Problem:

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

☑ • A <u>CFL</u> Problem:

Every context-free language is a member of P

Check-in Quiz 12/1

On gradescope