UMB CS420

Polynomial Time (P)

Wednesday, April 20, 2022
Announcements

• HW 9 in
  • Mapping Reducibility and Unrecognizability
  • Due Tues 4/19 11:59pm EST

• HW 10 out
  • Time Complexity and Poly time
  • Due Tues 4/26 11:59pm EST
Last Time: Time Complexity

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

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

Depends on size of input

Worst case
Last Time: Time Complexity Classes

Big-$O = \text{asymptotic upper bound, i.e., “only care about large } n\text{“}$

Let $t : \mathbb{N} \to \mathbb{R}^+$ be a function. Define the time complexity class, $\text{TIME}(t(n))$, 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)

\[ P = \bigcup_{k} \text{TIME}(n^k). \]

- Corresponds to “realistically” solvable problems:
  - Problems in \( P \)
    - = “solvable” or “tractable”
  - Problems outside \( P \)
    - = “unsolvable” or “intractable”
“Unsolvable” Problems

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

---

**Brute-force attack**

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.

**Amount of Time to Crack Passwords**

<table>
<thead>
<tr>
<th>Password Length</th>
<th>Time to Crack</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;abcdefg&quot; 7 chars</td>
<td>0.29 milliseconds</td>
</tr>
<tr>
<td>&quot;abcdefgh&quot; 8 chars</td>
<td>5 hours</td>
</tr>
<tr>
<td>&quot;abcdefghi&quot; 9 chars</td>
<td>5 days</td>
</tr>
<tr>
<td>&quot;abcdefghij&quot; 10 chars</td>
<td>4 months</td>
</tr>
<tr>
<td>&quot;abcdefghijkl&quot; 11 chars</td>
<td>1 decade</td>
</tr>
<tr>
<td>&quot;abcdefghijklm&quot; 12 chars</td>
<td>2 centuries</td>
</tr>
</tbody>
</table>

**How to prove something is “solvable”?**

**How to prove something is “unsolvable”?**
3 Problems in $\mathbf{P}$

- **A Graph Problem:**
  \[ \text{PATH} = \{ (G, s, t) | \text{G is a directed graph that has a directed path from s to t} \} \]

- **A Number Problem:**
  \[ \text{RELPRIME} = \{ (x, y) | \text{x and y are relatively prime} \} \]

- **A CFL Problem:**
  Every context-free language is a member of $\mathbf{P}$

- To prove that a language is “solvable”, i.e., in $\mathbf{P}$...
  - ... 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

- **New York**
- **Oswego**
- **Boston**
- **San Francisco**
- **Ithaca**

Edge weights:
- New York to Oswego: 109
- Oswego to Boston: 378
- Boston to San Francisco: 378
- New York to Ithaca: 98
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

- **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

Possible string encoding given to TMs:
\[
\{1, 2, 3, 4, 5, 6\}, \{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\}
\]
Interlude: Graph Encodings

- 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 $\mathbf{P}$

- **A Graph Problem:**
  \[ \text{PATH} = \{(G, s, t) | G \text{ is a directed graph that has a directed path from } s \text{ to } t \} \]

- **A Number Problem:**
  \[ \text{RELPRIME} = \{(x, y) | x \text{ and } y \text{ are relatively prime} \} \]

- **A CFL Problem:**
  Every context-free language is a member of $\mathbf{P}$
A Graph Theorem: \( \text{PATH} \in \mathbb{P} \)

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

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

- To prove that a language is in \( \mathbb{P} \) ...
- ... 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 \( n = \# \text{ vertices} \), then \( \# \text{ paths} \approx n^n \)
A Graph Theorem: $\text{PATH} \in \mathbf{P}$

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

**Proof** A polynomial time algorithm $M$ for $\text{PATH}$ operates as follows.

$M = \text{“On input } (G, s, t), \text{ where } G$ 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) ($n = \# \text{ nodes}$):

- **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.

$M = \text{“On input } \langle G, s, t \rangle, \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t:\n1. \text{ Place a mark on node } s. \n2. \text{ Repeat the following until no additional nodes are marked:} \n3. \text{ Scan all the edges of } G. \text{ If an edge } (a, b) \text{ is found going from a marked node } a \text{ to an unmarked node } b, \text{ mark node } b. \n4. \text{ If } t \text{ is marked, accept. Otherwise, reject.”} \n
# of steps (worst case) ($n = \# \text{ nodes}$):

- **Line 1**: 1 step
- **Lines 2-3 (loop)**:
  - **Steps/iteration (line 3)**: max # steps = max # edges = $O(n^2)$
A Graph Theorem: $\text{PATH} \in \text{P}$

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

**PROOF** A polynomial time algorithm $M$ for $\text{PATH}$ 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: $PATH \in P$

$PATH = \{(G, s, t) | G$ 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 $(G, s, t)$, 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: \( PATH \in P \)

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

**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:
   1. 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 \( b \).
3. If \( t \) is marked, accept. Otherwise, reject.”

# of steps (worst case) \( (n = \# \text{ nodes}) \):

- **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
A Graph Theorem: \( \textit{PATH} \in \mathcal{P} \)

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

PROOF. A polynomial time algorithm \( M \) for \( \text{PATH} \) 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, \textit{accept}. Otherwise, \textit{reject}.”

\# of steps (worst case) \( (n = \# \text{ nodes}) \):
- 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

\( \text{Total} = 1 + 1 + O(n^3) = O(n^3) \)

\( \mathcal{P} \) is the class of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. In other words,
\[ \mathcal{P} = \bigcup_{k} \text{TIME}(n^k). \]
3 Problems in \( \mathbf{P} \)

- **A Graph Problem:**
  \( \text{PATH} = \{ (G, s, t) | \text{G is a directed graph that has a directed path from } s \text{ to } t \} \)

- **A Number Problem:**
  \( \text{RELPRIME} = \{ (x, y) | x \text{ and } y \text{ are relatively prime} \} \)

- **A CFL Problem:**
  Every context-free language is a member of \( \mathbf{P} \)
A Number Theorem: \textit{RELPRIME} $\in$ \textit{P}

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

- Two numbers are \textbf{relatively prime}: if their $\text{gcd} = 1$
  - $\text{gcd}(x, y)$ = largest number that divides both $x$ and $y$
  - E.g., $\text{gcd}(8, 12) = 4$

- \textbf{Brute force (exponential) algorithm deciding RELPRIME:}
  - Try all of numbers (up to $x$ or $y$), see if it can divide both numbers
  - \textbf{Q:} Why is this exponential?
  - \textbf{HINT:} What is a typical “representation” of numbers?
  - \textbf{A:} binary numbers
  - (if $x = 2^n$, then trying $x$ numbers is exponential in $n$, the number of digits)

- \textbf{A gcd algorithm that runs in polynomial time:}
  - Euclid’s algorithm
A GCD Algorithm for: \( RELPRIME \in P \)

\[ RELPRIME = \{ (x, y) \mid x \text{ and } y \text{ are relatively prime} \} \]

**Modulo (i.e., remainder)**
- cuts \( x \) (at least) in half
- 15 mod 8 = 7
- 17 mod 8 = 1

**Cutting \( x \) in half every step requires:**
- \( \log x \) steps

The Euclidean algorithm \( E \) is as follows.

\[ E = \text{“On input } (x, y), \text{ where } x \text{ and } y \text{ are natural numbers in binary:} \]

1. Repeat until \( y = 0 \):
2. Assign \( x \leftarrow x \mod y \).
3. Exchange \( x \) and \( y \).
4. Output \( x \)."

\[ O(n) \]

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 \( n = \text{number of binary digits in (i.e., length of) } x \)
3 Problems in $\mathbf{P}$

• A Graph Problem:
  \[\text{PATH} = \{(G, s, t) \mid G \text{ is a directed graph that has a directed path from } s \text{ to } t\}\]

• A Number Problem:
  \[\text{RELPRIME} = \{(x, y) \mid x \text{ and } y \text{ are relatively prime}\}\]

• A CFL Problem:
  Every context-free language is a member of $\mathbf{P}$
Review: A Decider for Any CFL

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

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

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

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

$M_G$ 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 $L$:

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

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

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

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 $\mathbb{P}$

- 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

\[ S = \text{“On input } \langle G, w \rangle, \text{ where } G \text{ is a CFG and } w \text{ is a string:} \]

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

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Substring end 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.

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>Substring end char</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>vars generating “b”</td>
<td>vars for “ba”</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vars for “ba”</td>
<td>vars for “baa”</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vars for “baa”</td>
<td>b</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vars for “aab”</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>vars for “aab”</td>
<td>a</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Substrings end char

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

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>Substring end char</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>A,C</td>
</tr>
<tr>
<td>b</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>Substring end char</th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>A,C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:** \textbf{baaba}

- **Store every partial string and their generating rules:**

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Algorithm:**

- For each single char \( c \) and var \( A \):
  - If \( A \to c \) 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 \( A \to BC \):
      - Use table to check if \( B \)
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 rules in a table.

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>b</th>
<th>a</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Sub algorithm:

- 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 variables in a table.**

#### Substring end char

<table>
<thead>
<tr>
<th></th>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td>B</td>
<td>S,A</td>
<td></td>
<td>S,A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td>B</td>
<td>B</td>
<td>S,A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td>S,C</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>B</td>
<td>S,A</td>
<td></td>
<td>A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
</tbody>
</table>

#### 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 \( B \) generates \( x \) and \( C \) generates \( y \)

- If \( S \) is here, accept \( S,A,C \)
A CFG Theorem: Every context-free language is a member of $P$

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

1. For $w = \varepsilon$, if $S \rightarrow \varepsilon$ is a rule, accept; else, reject. [ $w = \varepsilon$ case ]
2. For $i = 1$ to $n$: \[ O(n) \] \[ \text{[Examine each substring of length 1]} \]
3. For each variable $A$: \[ \#\text{vars} \]
4. Test whether $A \rightarrow b$ is a rule, where $b = w_i$.
5. If so, place $A$ in $table(i,i)$.
6. For $l = 2$ to $n$: \[ O(n) \] \[ \text{[ } l \text{ is the length of the substring} \]
7. For $i = 1$ to $n - l + 1$: \[ O(n) \] \[ \text{[ } i \text{ is the start position of the substring} \]
8. Let $j = i + l - 1$. \[ \text{[ } j \text{ is the end position of the substring} \]
9. For $k = i$ to $j - 1$: \[ O(n) \] \[ \text{[ } k \text{ is the split position} \]
10. For each rule $A \rightarrow BC$: \[ \#\text{rules} \]
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.

Total: \[ O(n^3) \]

(This is also known as the Earley parsing algorithm)
Summary: 3 Problems in \( \mathbf{P} \)

- **A Graph Problem:**
  \( \text{PATH} = \{ (G, s, t) | \text{G is a directed graph that has a directed path from } s \text{ to } t \} \)

- **A Number Problem:**
  \( \text{RELPRIME} = \{ (x, y) | x \text{ and } y \text{ are relatively prime} \} \)

- **A CFL Problem:**
  Every context-free language is a member of \( \mathbf{P} \)
Check-in Quiz 4/20

On gradescope