Polynomial Time (P)

Wednesday, April 21, 2021
Announcements

• HW9 past due

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

• **FAQ**: How can I get better HW scores?
  • To earn more partial credit: 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
  • **But** ... 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

Problem: Show that language $L$ is undecidable, where $L = ...$

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

I know:
- To prove undecidability, use proof by contradiction
- A proof by contradiction requires an assumption:
  - Assume language $L$ is decidable
- A decidable language must have a decider, call it $R$
- Use this decider to create a contradiction:
  - 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 $f : \mathbb{N} \rightarrow \mathbb{N}$, 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.

**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 $t: \mathbb{N} \rightarrow \mathbb{R}^+$ 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 (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

• Unsolvable problems usually only have “brute force” solutions
  • “try all possible inputs”
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, $P = \bigcup_{k} \text{TIME}(n^k)$. 
Where Are We Now?

We are back in here now: deterministic, single-tape deciders (unless otherwise indicated)
Today: 3 Problems in \( \mathbf{P} \)

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

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

- **A CFL Problem:**
  
  Every context-free language is a member of \( \mathbf{P} \).
Interlude: 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)

- **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

Edge weights

New York

Ithaca

Oswego

Boston

San Francisco

109

98

378

378
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:

\[
\{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\}
\]
Interlude: Graph Encodings

\(G = (V, E), n = |V|\)

- 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?
  - \(= O(|V|^2) = O(n^2)\)

- 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 \(P\)
Today: 3 Problems in P

• A Graph Problem:

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

• A Number Problem:

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

• A CFL Problem:

Every context-free language is a member of P
A Graph Theorem: $PATH \in \mathbf{P}$

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

- To prove that a language is in $\mathbf{P}$ ...

- ... we must construct a polynomial time algorithm deciding the lang

- A non-polynomial (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: \( \text{PATH} \in \mathbf{P} \)

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

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

\( M = \text{“On input } \langle G, s, t \rangle, \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t:\)

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: $\text{PATH} \in \mathbf{P}$

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

**Proof**

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

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

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

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

**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, accept. Otherwise, reject.”

# of steps (worst case) $(n = \# \text{ nodes})$: 
- **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: \( \text{PATH} \in P \)

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

**Proof**

A polynomial time algorithm \( M \) for \( \text{PATH} \) operates as follows.

\( M = \) “On input \( (G, s, t) \), 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:
   2.1. 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 per loop:** max # steps = max # edges = \( O(n^2) \)
  - **# loops:** loop runs at most \( n \) times
    - **Total:** \( O(n^3) \)
A Graph Theorem: $PATH \in P$

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

**Proof**

A polynomial time algorithm $M$ for $PATH$ operates as follows.

$M = \text{"On input } (G, s, t), \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t:\}
\begin{itemize}
  \item 1. Place a mark on node $s$.
  \item 2. Repeat the following until no additional nodes are marked:
  \item 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 $b$.
  \item 4. If $t$ is marked, $accept$. Otherwise, $reject$.
\end{itemize}

# of steps (worst case) ($n = \# \text{ nodes}$):
- **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: $\textbf{PATH} \in \textbf{P}$

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

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

$M = \text{"On input } \langle G, s, t \rangle, \text{ where } G \text{ is a directed graph with nodes } s \text{ and } t:}$

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
- **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 = $1 + 1 + O(n^2) = O(n^3)$

**Definition 7.12**

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

\[ P = \bigcup_k \text{TIME}(n^k) \]
Today: 3 Problems in $\textbf{P}$

• 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:

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

• A CFL Problem:

Every context-free language is a member of $\textbf{P}$
A Number Theorem: \( RELPRIME \in \mathbb{P} \)

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

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

- 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: \( \text{RELPRIME} \in \mathbb{P} \)

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

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 \( x \).”

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

Each number is cut in half every other iteration.

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

Total run time (assume \( x > y \)): \( 2 \log x = 2 \log 2^n = O(n) \), where \( n \) = number of binary digits in (ie length of) \( x \)
Today: 3 Problems in 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 P
A CFG Theorem: Every context-free language is a member of \( \mathbb{P} \)

- 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 \( A \):

  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.

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

\[ S = \text{"On input } (G, w), \text{ where } G \text{ is a CFG and } w \text{ is a string:} \]
  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 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 substrings generated by each variable
CFL Dynamic Programming Example

- Chomsky Grammar $G$:
  - $S \rightarrow AB | BC$
  - $A \rightarrow BA | a$
  - $B \rightarrow CC | b$
  - $C \rightarrow AB | a$

- 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>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>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td></td>
<td>vars for “b”</td>
<td>vars for “ba”</td>
<td>vars for “baa”</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>vars for “a”</td>
<td>vars for “aa”</td>
<td>vars for “aab”</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></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>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>a</td>
<td>a</td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</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>
<th>b</th>
<th>a</th>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td>a</td>
<td>A,C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td>A,C</td>
<td></td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
CFL Dynamic Programming Example

• Chomsky Grammar $G$:
  • $S \rightarrow AB | BC$
  • $A \rightarrow BA | a$
  • $B \rightarrow CC | b$
  • $C \rightarrow AB | a$

• 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>
<th>a</th>
<th>b</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>b</td>
<td>B</td>
<td></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>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>B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A,C</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: $baaba$
- Store every partial string and their generating grammatical rules

**Algo:**
- 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 $B$

```
<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>A,C</td>
<td></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>
```

For substring “$ba$”, split into “$b$” and “$a$”:
- For rule $S \to AB$
  - Does $A$ generate “$b$” and $B$ generate “$a$”?  
    - NO
  - For rule $S \to BC$
    - Does $B$ generate “$b$” and $C$ generate “$a$”?  
      - YES
  - For rule $A \to BA$
    - Does $B$ generate “$b$” and $A$ generate “$a$”?  
      - YES
  - For rule $B \to CC$
    - Does $C$ generate “$b$” and $C$ generate “$a$”?  
      - NO
  - For rule $C \to AB$
    - 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 rules:

Substrings:

<table>
<thead>
<tr>
<th>Start char</th>
<th>End char</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
</tr>
<tr>
<td>a</td>
<td>A,C</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
- 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>Substring start char</th>
<th>b</th>
<th>a</th>
<th>Substring end char</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
<td>S,A</td>
<td>S,A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td>S,A,C</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td>A,C</td>
<td>B</td>
</tr>
<tr>
<td>b</td>
<td></td>
<td></td>
<td>S,A</td>
</tr>
<tr>
<td>a</td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
</tbody>
</table>

**Algo:**
- For each single char $c$ and var $A$:
  - If $A \rightarrow c$ is a rule, add $A$ to table
- For each substring $s$:
  - 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
A CFG Theorem: Every context-free language is a member of P

\[ D = \text{"On input } w = w_1 \cdots w_n:\]
\[ \begin{align*}
1. & \text{ For } w = \varepsilon, \text{ if } S \rightarrow \varepsilon \text{ is a rule, accept; else, reject. [ } w = \varepsilon \text{ case]}
2. & \text{ For } i = 1 \text{ to } n: \quad O(n) \quad \#\text{vars} \quad \text{[examine each substring of length 1]}
3. & \text{ For each variable } A: \quad O(n) \quad \#\text{vars} \quad \text{[start position of the substring]}
4. & \text{ Test whether } A \rightarrow b \text{ is a rule, where } b = w_i.
5. & \text{ If so, place } A \text{ in } \text{table}(i, i).
6. & \text{ For } i = 2 \text{ to } n: \quad O(n) \quad \#\text{rules} \quad \text{[end position of the substring]}
7. & \text{ For } j = i + l - 1: \quad O(n) \quad \#\text{rules} \quad \text{[split position]}
8. & \text{ For } k = i \text{ to } j - 1: \quad O(n) \quad \#\text{rules} \quad \text{[start position of the substring]}
9. & \text{ For each rule } A \rightarrow BC: \quad \#\text{rules} \quad \text{[contains B and } table(k + 1, j) \text{ contains C]}
10. & \text{ If } \text{table}(i, k) \text{ contains } B \text{ and } \text{table}(k + 1, j) \text{ contains C, put } A \text{ in } \text{table}(i, j).
11. & \text{ If } S \text{ is in } \text{table}(1, n), \text{ accept; else, reject.}
\end{align*} \]

\[ \#\text{vars} \times n = O(n) \]
\[ \#\text{rules} \times O(n) \times O(n) \times O(n) = O(n^3) \]

- **Total:** \( O(n^3) \)
- This is also known as the Earley parsing algorithm
Summary: 3 Problems in $\textbf{P}$

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

- A Number Problem:
  \[ \text{RELPRIME} = \{ \langle x, y \rangle \mid x \text{ and } y \text{ are relatively prime} \} \]

- A CFL Problem:
  Every context-free language is a member of $\textbf{P}$
Check-in Quiz 4/21

On gradescope