Polynomial Time

Monday November 23, 2020

HW Questions?

Announcements

- See piazza for note about grading

- See piazza for note about submitting work that is not yours
  - Tl;dr: Don’t do it!

- Grace period until Nov 30 11:59pm EST
  - Anyone may resubmit hw 7 or hw 8 without penalty

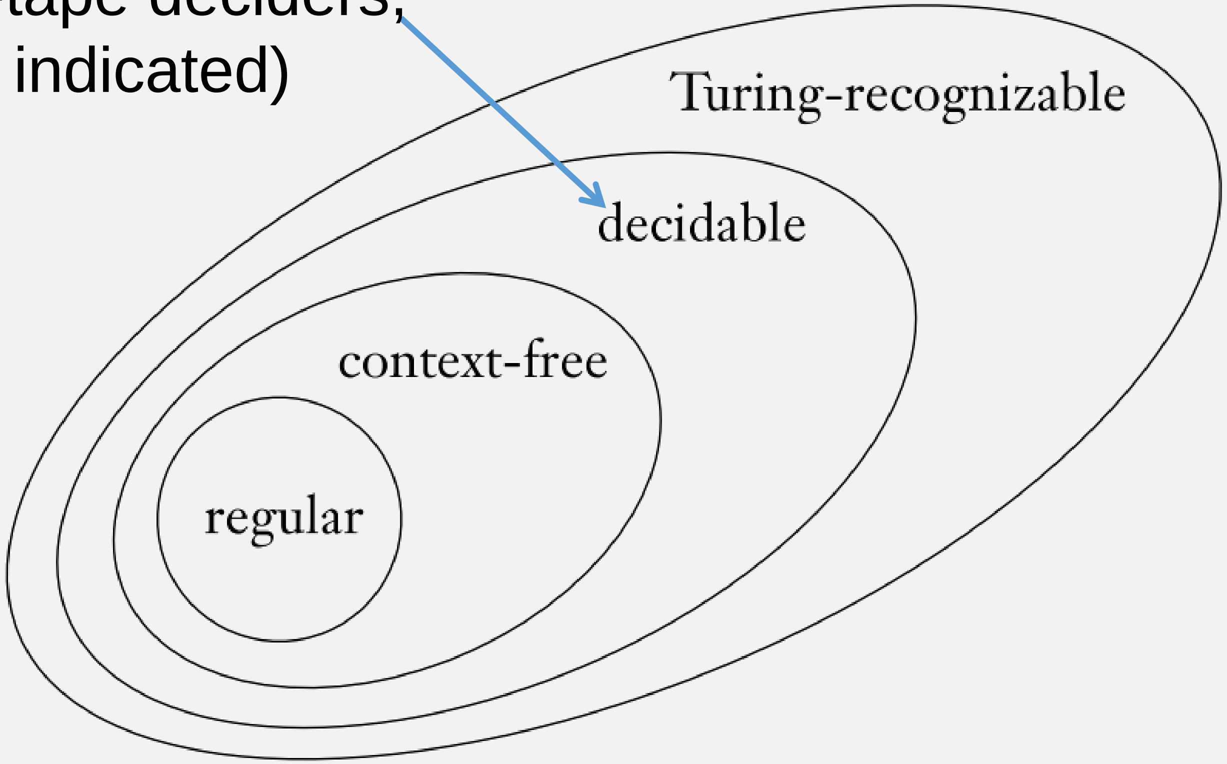
How to Use Answers from the Internet

- Assume course staff sees everything you do

- Changing variable names or using a thesaurus is insufficient
  - Don’t just make local changes
  - If the structure of the answer is unchanged, it’s still copied
- Any “weirdness” in the answer indicates copying
  - Small mistakes, wrong terminology, bad grammar, nonsensical phrases
  - At best it better be unique to your submission

We are here

(deterministic, single-tape deciders, unless otherwise indicated)



P

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,

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

- Corresponds to “realistically” solvable problems
- From now on:
  - Problems in P = “solvable”
  - Problems outside P = “unsolvable”
    - These are usually “brute force” solutions that “try all possible inputs”

Today: 3 Problems in **P**

- A Graph Problem:

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

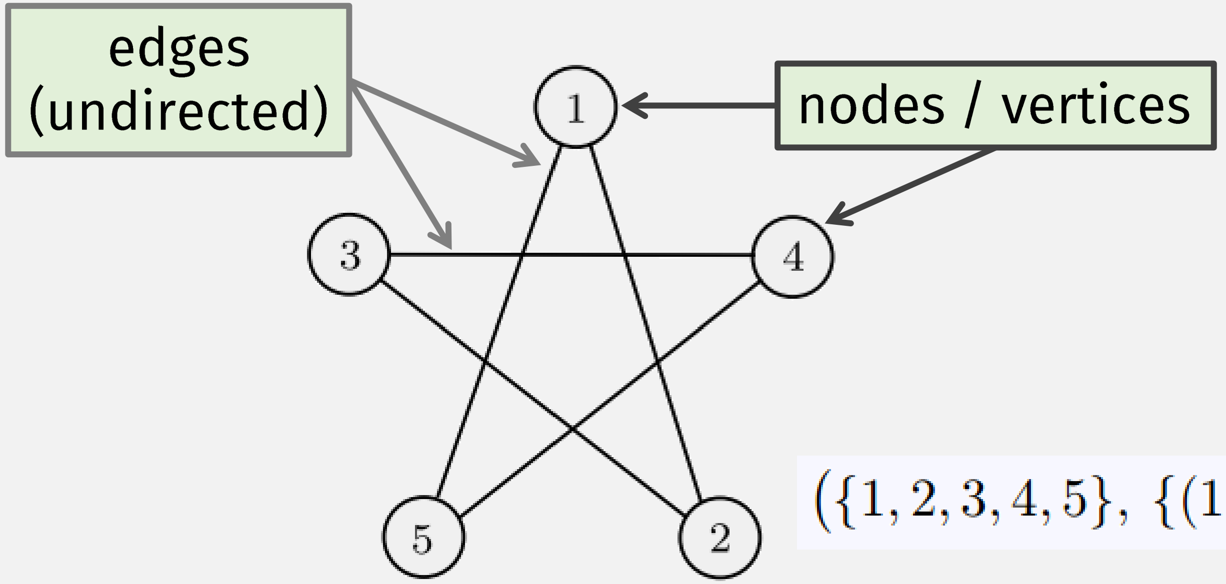
- A Number Problem:

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

- A CFL Problem:

Every context-free language is a member of P

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



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

- Edge defined by two nodes (order doesn't matter)
- Formally, a graph = (V, E)
  - V = set of nodes, E = set of edges
- This will roughly be the string encoding passed to TMs, ie <*G*>

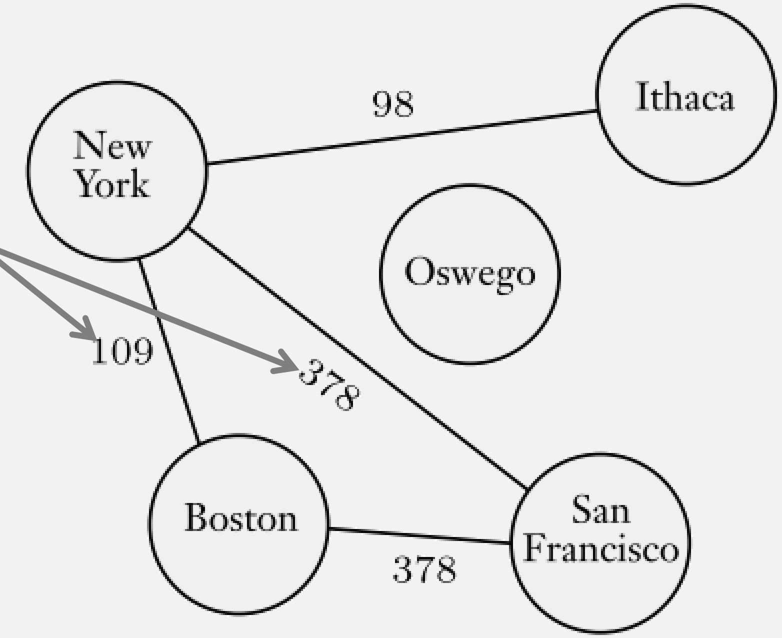
Interlude: Graph Encodings

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

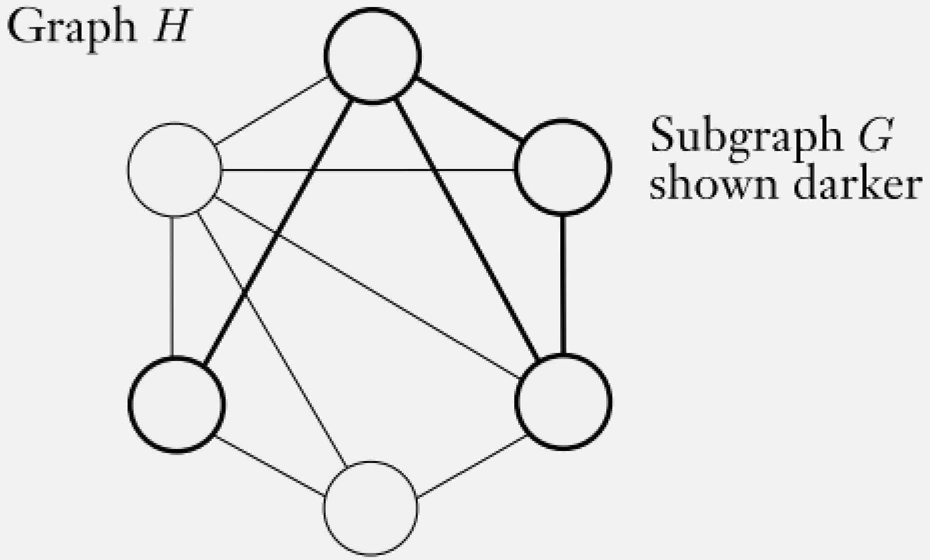
- In graph algorithms, count steps in terms of number of vertices
  - and sometimes number of edges
  - Instead of actual "length of input"
- Given a graph $G = (V, E)$ with $|V|$ vertices
- Max edges =
  - $O(|V|^2)$
- So # vertices + edges is always polynomial in length of input
- Algorithm runs in time polynomial in the number of vertices ⇔ algorithm runs in time polynomial in the length of input

Interlude: Weighted Graphs

Edge weights

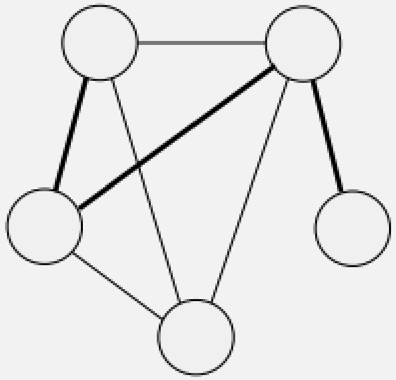


Interlude: Subgraphs

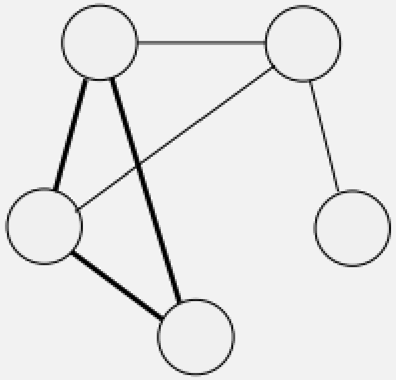


Interlude: Paths and other Graph Things

- Path
  - A sequence of nodes connected by edges

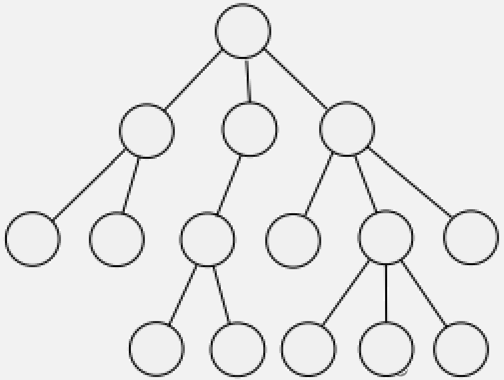


- Cycle
  - A path that starts/ends at the same node

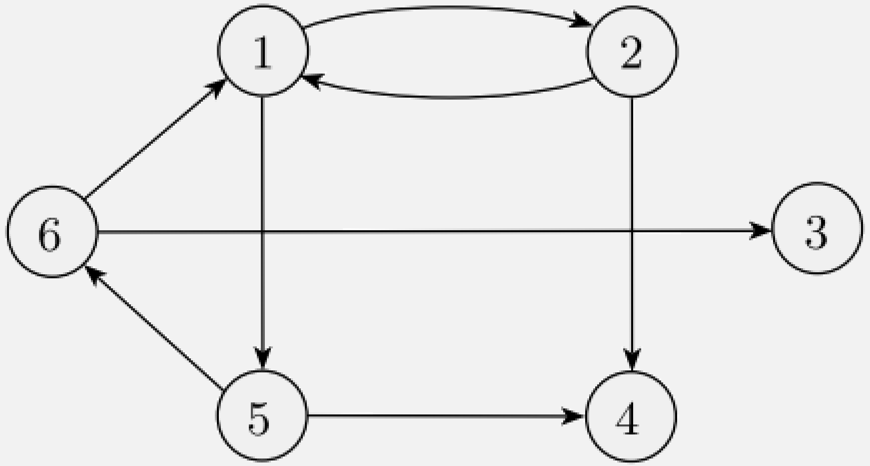


- Connected graph
  - Every two nodes has a path

- Tree
  - A connected graph with no cycles



Interlude: Directed Graphs

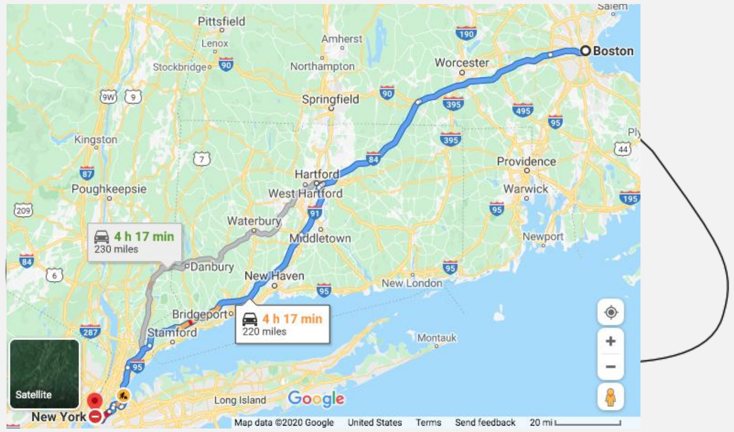


$(\{1,2,3,4,5,6\}, \{(1,2), (1,5), (2,1), (2,4), (5,4), (5,6), (6,1), (6,3)\})$

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

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

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



- To prove that a language is in **P** …

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

- A non-polynomial (i.e., exponential, brute force) algorithm:
  - check all paths, and see if any connect s to t

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

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

PROOF A polynomial time algorithm M for *PATH* operates as follows.

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

1. Place a mark on node s.
2. Repeat the following until no additional nodes are marked:
3. Scan all the edges of G. If an edge (a, b) is found going from a marked node a to an unmarked node b, mark node b.
4. If t is marked, *accept*. Otherwise, *reject*."

- # of steps ($\boldsymbol{n}$ = # nodes):
  - Line 1: ***1*** step
  - Line 3: max # steps = max # edges = $O(\boldsymbol{n}^2)$
  - Line 2: loop runs at most $\boldsymbol{n}$ times
  - Line 4: ***1*** step
  - Total = $O(\boldsymbol{n}^3)$

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

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

- Two numbers are relatively prime if their $\gcd = 1$
  - 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?

- We can prove using a $\gcd$ algorithm that runs in poly time
  - E.g., Euclid's algorithm

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

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

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

- *15 mod 8 = 7*
- *17 mod 8 = 1*

The Euclidean algorithm E is as follows.

E = "On input $\langle x, y\rangle$, where x and y are natural numbers in binary:

1. Repeat until $y = 0$:

2. Assign $x \leftarrow x \bmod y$.

3. Exchange x and y.

4. Output x."

Each number is cut in half every *other* iteration

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

So <u>run time</u> (assume x > y) is 2log x = 2log2^n = **O(n)**, where n = length of x

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

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

From Thm 4.9

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

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

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*."

From Thm 4.7

? ?

- 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 which variables can generate which <u>substrings</u>

CFL Dynamic Programming Example

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

Substring end char

Substring start char

	b	a	a	b	a
b					
a					
a					
b					
a					

CFL Dynamic Programming Example

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

Substring end char

Substring start char

	b	a	a	b	a
b	vars for “b”	vars for “ba”	vars for “baa”	…	
a		vars for “a”	vars for “aa”	vars for “aab”	
a			…		
b					
a					

CFL Dynamic Programming Example

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

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

Substring end char

Substring start char

	b	a	a	b	a
b	vars for “b”	vars for “ba”	vars for “baa”	…	
a		vars for “a”	vars for “aa”	vars for “aab”	
a			…		
b					
a					

CFL Dynamic Programming Example

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

<u>Algo</u>:
- For each single char c and var A:
  - If A → c is a rule, add A to table

Substring end char

Substring start char

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

CFL Dynamic Programming Example

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

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

Substring end char

Substring start char

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

CFL Dynamic Programming Example

- Chomsky Grammar *G*:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their ge[nerating…]

Algo:
- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring s:
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B

Substring end char

Substring start char

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

For substring "**ba**", split into "**b**" and "**a**":
- For rule S → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO
- For rule S → BC
  - Does B generate "**b**" and C generate "**a**"?
  - YES
- For rule A → BA
  - Does B generate "**b**" and A generate "**a**"?
  - YES
- For rule B → CC
  - Does C generate "**b**" and C generate "**a**"?
  - NO
- For rule C → AB
  - Does A generate "**b**" and B generate "**a**"?
  - NO

CFL Dynamic Programming Example

- Chomsky Grammar G:
  - S → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a
- Example string: **baaba**
- Store every partial string and their ge[nerating …]

Algo:

- For each single char c and var A:
  - If A → c is a rule, add A to table
- For each substring s:
  - For each split of substring s into x,y:
    - For each rule of shape A → BC:
      - Use table to check if B …

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

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

Substring end char (columns); Substring start char (rows)

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

CFL Dynamic Programming Example

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

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

Substring end char

Substring start char

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

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

- <u>Total</u>: $O(\mathbf{n^3})$
- A.k.a., Earley parsing algorithm

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

- A Graph Problem:

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

- A Number Problem:

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

- A CFL Problem:

Every context-free language is a member of P

Check-in Quiz 11/23

On gradescope

End of Class Survey 11/23

See course website