UMB CS622

P (and NP?)

Monday, May 6, 2024
Announcements

• HW 11
  • Due Wed 5/8 12pm noon

• HW 12
  • Release Wed 5/8 12pm noon
  • Due Wed 5/15 12pm noon (no late days, no exceptions)
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 $\mathbf{P}$

IF-THEN Statement to Prove:

IF a language $L$ is a CFL, THEN $L$ is in $\mathbf{P}$
Review: A Decider for Any CFL

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

$L = \text{“On input } w:\n1. \text{ Run TM } S \text{ on input } \langle G, w \rangle.\n2. \text{ If this machine accepts, } accept; \text{ if it rejects, } reject.$

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

$S = \text{“On input } \langle G, w \rangle, \text{ where } G \text{ is a CFG and } w \text{ is a string:}\n1. \text{ Convert } G \text{ to an equivalent grammar in Chomsky normal form.}\n2. \text{ List all derivations with } 2n - 1 \text{ steps, where } n \text{ is the length of } w; \text{ except if } n = 0, \text{ then instead list all derivations with one step.}\n3. \text{ If any of these derivations generate } w, \text{ accept; if not, reject.”}$

$M_G$ is a decider, bc $S$ is a decider

$M_G$ accepts all $w \in L$, for any CFL $L$ (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 $L$:

$$M_G = \text{“On input } w:\n1. \text{ Run TM } S \text{ on input } \langle G, w \rangle.\n2. \text{ If this machine accepts, accept; if it rejects, reject.”}$$

$S$ 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. 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?

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

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

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>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>
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>vars generating “b”</td>
<td>vars for “ba”</td>
<td>vars for “baa”</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>a</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>
</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>
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>vars generating “b”</td>
<td>vars for “ba”</td>
<td>vars for “baa”</td>
<td>...</td>
<td></td>
</tr>
<tr>
<td>a</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>
</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>
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>b</td>
<td></td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>A</td>
<td></td>
<td></td>
<td>A,C</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A,C</td>
</tr>
<tr>
<td>a</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>A,C</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 variables in a table:

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

\textbf{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 \( y \)
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 rule:

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

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 → AB | BC
  - A → BA | a
  - B → CC | b
  - C → AB | a

- **Example string:** **baaba**

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

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

**Algorithm:**
- 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
CFL Dynamic Programming Example

- Chomsky Grammar $G$:
  - For each:
    - char
    - var
  - $B \rightarrow \varepsilon | b$
  - $C \rightarrow AB | a$

- For each: substring
  - split of substring
  - rule

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 $y$

<table>
<thead>
<tr>
<th>Substring start char</th>
<th>Substring end char</th>
<th>If $S$ is here, accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>b</td>
<td>B</td>
<td>$S,A,C$</td>
</tr>
<tr>
<td>a</td>
<td>A,C</td>
<td>B</td>
</tr>
<tr>
<td>a</td>
<td>A,C</td>
<td>$S,C$</td>
</tr>
<tr>
<td>b</td>
<td>B</td>
<td>$S,A$</td>
</tr>
<tr>
<td>a</td>
<td>A,C</td>
<td>$A,C$</td>
</tr>
</tbody>
</table>
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 $n$: \hspace{2cm} \text{O(n) chars}\hspace{1cm} \text{[examine each substring of length 1]}$
3. For each variable $A$: \hspace{3cm} \text{#vars = constant = O(1)}$
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 $n$: \hspace{2cm} \text{O(n) diff lengths}\hspace{1cm} \text{[ $l$ is the length of the substring]}$
7. For $i = 1$ to $n - l + 1$: \hspace{3cm} \text{O(n) strings of each length}\hspace{1cm} \text{[substrings]}$
8. Let $j = i + l - 1$. \hspace{2cm} \text{[ $j$ is the end position of the substring]}$
9. For $k = i$ to $j - 1$: \hspace{4cm} \text{O(n) ways to split a string into two pieces}$
10. For each rule $A \rightarrow BC$: \hspace{4cm} \text{#vars = constant = O(1)}$
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: \text{O(n$^3$)}$

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

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

- **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 \( \mathbf{P} \)
Who doesn't like niche NP jokes?

AN ENGINEER, A PHYSICIST, AND A MATHEMATICIAN ARE ROOMMATES AND ARE MOVING TO A NEW PLACE.

THE MOVEMBER REASSURES THEM I SEEN AT THIS 30 YEARS, I CAN LOOK AT ANY AMOUNT OF STUFF AND INSTANTLY TELL YA IF IT CAN FIT IN THE MOVING BINS.

THE ENGINEER SAYS... IT'S OBVIOUS IT CAN FIT. ANYTHING THAT DOESN'T GO IN THE BINS CAN BE TAPED TO THE ROOF.

THE PHYSICIST SAYS... IT'S OBVIOUS IT CAN FIT, IF IT WERE THE DENSITY OF A NEUTRON STAR, OUR STUFF WOULD BE THE SIZE OF A BASEBALL.

THE MATHEMATICIAN SAYS... PLEASE DON'T HACK MY EMAIL.

NP
Last Time: Poly 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,

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

- Corresponds to “realistically” solvable problems:
  - Problems in **P**
    - = “solvable” or “tractable”
  - Problems outside **P**
    - = “unsolvable” or “intractable”
Last Time: 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 \( \mathbf{P} \)
Search vs Verification

- **Search** problems are often **unsolvable**
- But, **verification** of a search result is usually **solvable**

**Examples**

- **Factoring**
  - **Unsolvable:** Find factors of 8633
    - Must “try all” possibilities
  - **Solvable:** Verify 89 and 97 are factors of 8633
    - Just do multiplication

- **Passwords**
  - **Unsolvable:** Find my umb.edu password
  - **Solvable:** Verify whether my umb.edu password is ...  
    - “correct horse battery staple”
The *PATH* Problem

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

- It’s a **search** problem:
  - **Exponential time** (brute force) algorithm \((n^n)\):
    - Check all \(n^n\) possible paths and see if any connects \(s\) and \(t\)
  - **Polynomial time** algorithm:
    - Do a breadth-first search (roughly), marking “seen” nodes as we go \((n = \# \text{ nodes})\)

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

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

\(O(n^3)\)
Verifying a **PATH**

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

The **verification** problem:

- **Given** some path \( p \) in \( G \), check that it is a path from \( s \) to \( t \)
- Let \( m = \) length of longest possible path = \# edges

Verifier \( V \) = On input \( \langle G, s, t, p \rangle \), where \( p \) is some set of edges:

1. Check some edge in \( p \) has “from” node \( s \); mark and set it as “current” edge
   - Max steps = \( O(m) \)
2. **Loop**: While there remains unmarked edges in \( p \):
   1. Find the “next” edge in \( p \), whose “from” node is the “to” node of “current” edge
   2. If found, then mark that edge and set it as “current”, else reject
   - Each loop iteration: \( O(m) \)
   - \# loops: \( O(m) \)
   - Total looping time = \( O(m^2) \)
3. Check “current” edge has “to” node \( t \); if yes accept, else reject

- Total time = \( O(m^2) = O(m^2) = \text{polynomial in } m \)

**NOTE**: extra argument \( p \), “Verifying” an answer requires having a potential answer to check!

**PATH** can be verified in polynomial time
Verifiers, Formally

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

A verifier for a language \( A \) is an algorithm \( V \), where
\[ A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \} \]

We measure the time of a verifier only in terms of the length of \( w \), so a polynomial time verifier runs in polynomial time in the length of \( w \). A language \( A \) is polynomially verifiable if it has a polynomial time verifier.

- **NOTE**: a certificate \( c \) must be at most length \( n^k \), where \( n = \text{length of } w \)

  - Why? Because it takes time \( n^k \) to read it

So \( \text{PATH} \) is polynomially verifiable
The **HAMPATH** Problem

- A Hamiltonian path goes through every node in the graph

- The **Search** problem:
  - Exponential time (brute force) algorithm:
    - Check all possible paths and see if any connect $s$ and $t$ using all nodes
  - Polynomial time algorithm: **???**
    - We don’t know if there is one!!!

- The **Verification** problem:
  - Still $O(m^2)!$ (same verifier for **PATH**)
  - **HAMPATH** is polynomially verifiable, but not polynomially decidable
The class **NP**

**DEFINITION**

*N*P* is the class of languages that have polynomial time verifiers.

- *PATH* is in *NP*, and *P*
- *HAMPATH* is in *NP*, but it’s **unknown** whether it’s in *P*
NP = **Non-deterministic polynomial time**

NP is the class of languages that have polynomial time verifiers.

---

**Theorem**

A language is in NP iff it is decided by some non-deterministic polynomial time Turing machine.

⇒ If a language is in NP, then it has a non-deterministic poly time decider

- We know: If a lang $L$ is in NP, then it has a poly time verifier $V$
- Need to: create NTM deciding $L$

On input $w =$

- Nondeterministically run $V$ with $w$ and all possible poly length certificates $c$

⇐ If a language has a non-deterministic poly time decider, then it is in NP

- We know: $L$ has NTM decider $N$,
- Need to: show $L$ is in NP, i.e., create polytime verifier $V$:

On input $<w, c> =$ Potentially exponential slowdown?

- Convert $N$ to deterministic TM, and run it on $w$, but take only one computation path
- Let certificate $c$ dictate which computation path to follow

---

**NOTE:** a verifier cert is usually a potential “answer”, but does not have to be (like here)

Certificate $c$ specifies a path

Deterministic (verifier) TMs cannot “call” non-deterministic TMs

Converting NTM to deterministic TM is exponentially slower!
$\text{NP} = \bigcup_k \text{NTIME}(n^k)$

$\text{NTIME}(t(n)) = \{L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine}\}$. 

$\text{NP} = \text{Nondeterministic polynomial time}$
**NP vs P**

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

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

**P = Deterministic polynomial time**

\[ \text{NTIME}(t(n)) = \{ L \mid L \text{ is a language decided by an } O(t(n)) \text{ time nondeterministic Turing machine} \}. \]

**NP** = \( \bigcup_k \text{NTIME}(n^k) \)

**NP = Nondeterministic polynomial time**

Also, **NP** = Deterministic polynomial time verification
More **NP** Problems

- **CLIQUE** = \{\langle G, k \rangle | G \text{ is an undirected graph with a } k\text{-clique} \}
  - A clique is a subgraph where every two nodes are connected
  - A \(k\)-clique contains \(k\) nodes

- **SUBSET-SUM** = \{\langle S, t \rangle | S = \{x_1, \ldots, x_k\}, and for some \( \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\} \), we have \( \Sigma y_i = t \) \}
Theorem: **CLIQUE** is in NP

\[ \text{CLIQUE} = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \} \]

**Proof Idea**  The clique is the certificate.

**Proof**  The following is a verifier \( V \) for **CLIQUE**.

\[ V = \text{“On input } \langle \langle G, k \rangle, c \rangle \text{, check whether:} \]

1. Test whether \( c \) is a subgraph with \( k \) nodes in \( G \).
2. Test whether \( G \) contains all edges connecting nodes in \( c \).
3. If both pass, accept; otherwise, reject.”

A verifier for a language \( A \) is an algorithm \( V \), where

\[ A = \{ w \mid V \text{ accepts } \langle w, c \rangle \text{ for some string } c \}. \]

We measure the time of a verifier only in terms of the length of \( w \), so a polynomial time verifier runs in polynomial time in the length of \( w \). A language \( A \) is polynomially verifiable if it has a polynomial time verifier.

NP is the class of languages that have polynomial time verifiers.
Proof 2: CLIQUE is in NP

CLIQUE = \{ \langle G, k \rangle \mid G \text{ is an undirected graph with a } k\text{-clique} \}

N = “On input \langle G, k \rangle, where G is a graph:
1. Nondeterministically select a subset \( c \) of \( k \) nodes of \( G \).
2. Test whether \( G \) contains all edges connecting nodes in \( c \).
3. If yes, accept; otherwise, reject.”

Checking whether a subgraph is clique: \( O(n^2) \)

To prove a lang \( L \) is in NP, create either a:
1. Deterministic poly time verifier
2. Nondeterministic poly time decider

Don’t forget to count the steps

How to prove a language is in NP:
Proof technique #2: create an NTM

Theorem
A language is in NP iff it is decided by some nondeterministic polynomial time Turing machine.
More **NP** Problems

- **CLIQUE** = \{⟨G, k⟩ | G is an undirected graph with a k-clique\}
  - A clique is a subgraph where every two nodes are connected
  - A k-clique contains k nodes

- **SUBSET-SUM** = \{⟨S, t⟩ | S = \{x₁, ..., xₖ\}, and for some \(y₁, ..., yₙ \subseteq \{x₁, ..., xₖ\}\), we have \(\sum y_i = t\)\}
  - Some subset of a set of numbers S must sum to some total t
  - e.g., \{\{4, 11, 16, 21, 27\}, 25\} \in **SUBSET-SUM**
Theorem: **SUBSET-SUM** is in NP

\[ \text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t \} \]

**Proof Idea**

The subset is the certificate.

**Proof**

The following is a verifier \( V \) for **SUBSET-SUM**.

\[ V = \text{“On input } \langle \langle S, t \rangle, c \rangle \text{:} \]

1. Test whether \( c \) is a collection of numbers that sum to \( t \).
2. Test whether \( S \) contains all the numbers in \( c \).
3. If both pass, accept; otherwise, reject.”

To prove a lang is in **NP**, create either:

1. Deterministic poly time verifier
2. Nondeterministic poly time decider

Don’t forget to compute run time! Does this run in poly time?
Proof 2: \textit{SUBSET-SUM} is in NP

\[ \text{SUBSET-SUM} = \{ \langle S, t \rangle \mid S = \{x_1, \ldots, x_k\}, \text{ and for some } \{y_1, \ldots, y_l\} \subseteq \{x_1, \ldots, x_k\}, \text{ we have } \sum y_i = t \} \]

To prove a lang is in \textbf{NP}, create either:
1. Deterministic poly time verifier
2. Nondeterministic poly time decider

Don’t forget to compute run time! Does this run in poly time?

\textbf{ALTERNATIVE PROOF} We can also prove this theorem by giving a nondeterministic polynomial time Turing machine for \textit{SUBSET-SUM} as follows.

\[ N = \text{“On input } \langle S, t \rangle: \]

1. Nondeterministically select a subset \( c \) of the numbers in \( S \).
2. Test whether \( c \) is a collection of numbers that sum to \( t \).
3. If the test passes, \textit{accept}; otherwise, \textit{reject}.”
COMPOSITES = \{ x \mid x = pq, \text{ for integers } p, q > 1 \}

• A composite number is not prime

• COMPOSITES is polynomially verifiable
  • i.e., it’s in \textbf{NP}
  • i.e., factorability is in \textbf{NP}

• A certificate could be:
  • Some factor that is not 1

• Checking existence of factors (or not, i.e., testing primality) ...
  • ... is also poly time
  • But only discovered recently (2002)!
**HW Question:** Does $P = NP$?

How do you prove an algorithm doesn’t have a poly time algorithm? (in general it’s hard to prove that something doesn’t exist)
Implications if $P = NP$

- Every problem with a “brute force” solution also has an efficient solution
- I.e., “unsolvable” problems are “solvable”
- **BAD:**
  - Cryptography needs unsolvable problems
  - Near perfect AI learning, recognition
- **GOOD:** Optimization problems are solved
  - Optimal resource allocation could fix all the world’s (food, energy, space ...) problems?
Progress on whether $P = NP$?

- Some, but still not close

The Status of the P Versus NP Problem

By Lance Fortnow
Communications of the ACM, September 2009, Vol. 52 No. 9, Pages 78-86
10.1145/1562164.1562186

- One important concept discovered:
  - NP-Completeness
**NP-Completeness**

**DEFINITION**

A language $B$ is **NP-complete** if it satisfies two conditions:

1. $B$ is in NP, and
2. every $A$ in NP is polynomial time reducible to $B$.

- How does this help the $P = NP$ problem?

**THEOREM**

If $B$ is NP-complete and $B \in P$, then $P = NP$. 
**Flashback: Mapping Reducibility**

Language $A$ is **mapping reducible** to language $B$, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the **reduction** from $A$ to $B$.

**IMPORTANT:** “if and only if” ...

To show mapping reducibility:
1. create **computable fn**
2. and then show **forward direction**
3. and **reverse direction**
   (or **contrapositive of forward direction**)

$A_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ accepts } w\}$

$HALT_{TM} = \{\langle M, w \rangle | M \text{ is a TM and } M \text{ halts on input } w\}$

... means $\overline{A} \leq_m \overline{B}$

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a **computable function** if some Turing machine $M$, on every input $w$, halts with just $f(w)$ on its tape.
Polynomial Time Mapping Reducibility

Language $A$ is **mapping reducible** to language $B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, such that:

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the **reduction** from $A$ to $B$.

Language $A$ is **polynomial time mapping reducible**, or simply **polynomial time reducible**, to language $B$, written $A \leq \text{P} B$, if a polynomial time computable function $f: \Sigma^* \rightarrow \Sigma^*$ exists, where for every $w$,

$$w \in A \iff f(w) \in B.$$ 

The function $f$ is called the **polynomial time reduction** of $A$ to $B$.

- **To show poly time mapping reducibility:**
  1. create *computable fn*
  2. show computable fn runs in *poly time*
  3. then show **forward direction**
  4. and show **reverse direction**
     (or **contra-positive** of reverse direction)

**Don’t forget:** “if and only if” ...

A function $f: \Sigma^* \rightarrow \Sigma^*$ is a **computable function** if some Turing machine $M$, on every input $w$, halts with just $f(w)$ on its tape.
Flashback: If $A \leq_m B$ and $B$ is decidable, then $A$ is decidable.

PROOF: We let $M$ be the decider for $B$ and $f$ be the reduction from $A$ to $B$. We describe a decider $N$ for $A$ as follows.

$N =$ “On input $w$:
1. Compute $f(w)$.
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs.”

This proof only works because of the if-and-only-if requirement.

Language $A$ is mapping reducible to language $B$, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$  

The function $f$ is called the reduction from $A$ to $B$. 
**Thm:** If $A \leq_m B$ and $B \in \mathsf{P}$, then $A \in \mathsf{P}$.

**Proof** We let $M$ be the decider for $B$ and $f$ be the reduction from $A$ to $B$. We describe a decider $N$ for $A$ as follows.

$N =$ “On input $w$:
1. Compute $f(w)$.
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs.”

Language $A$ is *mapping reducible* to language $B$, written $A \leq_m B$, if there is a computable function $f : \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$  

The function $f$ is called the *reduction* from $A$ to $B$. 

![Diagram](image)
Thm: If $A \leq_m B$ and $B \in \mathbb{P}$ is decidable, then $A \in \mathbb{P}$.

**Proof**

We let $M$ be the decider for $B$ and $f$ be the reduction from $A$ to $B$. We describe a decider $N$ for $A$ as follows.

$N =$ “On input $w$:

1. Compute $f(w)$.
2. Run $M$ on input $f(w)$ and output whatever $M$ outputs.”

Language $A$ is **mapping reducible** to language $B$, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \rightarrow \Sigma^*$, where for every $w$,

$$w \in A \iff f(w) \in B.$$  

The function $f$ is called the **reduction** from $A$ to $B$. 

**poly time**

**poly time**

**poly time**

**poly time**
To prove $\mathbf{P} = \mathbf{NP}$, must show:

1. every language in $\mathbf{P}$ is in $\mathbf{NP}$
   - Trivially true (why?)

2. every language in $\mathbf{NP}$ is in $\mathbf{P}$
   - Given a language $A \in \mathbf{NP}$ ...
   - ... can poly time mapping reduce $A$ to $B$
     - because $B$ is $\mathbf{NP}$-Complete
   - Then $A$ also $\in \mathbf{P}$ ...
     - Because $A \leq_{\mathbf{P}} B$ and $B \in \mathbf{P}$, then $A \in \mathbf{P}$

Thus, if a language $B$ is $\mathbf{NP}$-complete and in $\mathbf{P}$, then $\mathbf{P} = \mathbf{NP}$
Next Time: 3SAT is polynomial time reducible to CLIQUE.