Previously on CS103…
The Complexity Class $\mathbf{P}$

- The complexity class $\mathbf{P}$ contains all problems that can be solved in polynomial time.

- Formally:

$$
\mathbf{P} = \bigcup_{k=0}^{\infty} \text{TIME}(n^k)
$$

- The Cobham-Edmonds Thesis: A decision problem can be solved efficiently iff it is in $\mathbf{P}$. 
Examples of Problems in \( \mathbf{P} \)

- All regular languages are in \( \mathbf{P} \).
  - Can be decided in \( O(n) \).
- All DCFLs are in \( \mathbf{P} \).
  - Can simulate a DPDA in \( O(n) \) on a two-tape TM, so can simulate DPDA on a single-tape TM in \( O(n^2) \).
- All CFLs are in \( \mathbf{P} \).
  - Using more advanced techniques, can solve in polynomial time.
- Many other problems are in \( \mathbf{P} \):
  - Linear programming.
  - String searching.
  - Maximum matching.
Undecidable Languages

- Regular Languages
- CFLs
- DCFLs

Decidable Languages

- P

Undecidable Languages

Decidable Languages
Proving Languages are in $\text{P}$

• **Directly prove the language is in $\text{P}$**.
  - Build a decider for the language $L$.
  - Prove that the decider runs in time $O(n^k)$.

• **Use closure properties**.
  - Prove that the language can be formed by appropriate transformations of languages in $\text{P}$.

• **Reduce the language to a language in $\text{P}$**.
  - Show how a polynomial-time decider for some language $L'$ can be used to decide $L$. 
Polynomial-Time Reductions

- Let $A \subseteq \Sigma_1^*$ and $B \subseteq \Sigma_2^*$ be languages.

- A **polynomial-time reduction** is a function $f : \Sigma_1^* \rightarrow \Sigma_2^*$ with the following properties:
  - $f(w)$ can be computed in polynomial time.
  - $w \in A$ iff $f(w) \in B$.

- Notation: $A \leq_p B$.

- Informally:
  - A way of turning inputs to $A$ into inputs to $B$
  - that can be computed in polynomial time
  - that preserves the correct answer.
Polynomial-Time Reductions

- Suppose that we know that \( B \in P \).
- Suppose that \( f \) is a polynomial-time reduction from \( A \) to \( B \).
Polynomial-Time Reductions

- Suppose that we know that $B \in \text{P}$. 
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$. 

Input size: $n$

- $A$
  - Solvable?
- $B$
  - Solvable in $O(n')$
Polynomial-Time Reductions

• Suppose that we know that $B \in \mathbf{P}$.
• Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$
Polynomial-Time Reductions

• Suppose that we know that $B \in \mathbb{P}$.
• Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

Time required: $O(n^k)$

A reduces to B

A Solvable?

B Solvable in $O(n^r)$
Polynomial-Time Reductions

- Suppose that we know that $B \in \text{P}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$. 

![Diagram](image)

Input size: $n$

**A**

Solvable? 

Time required: $O(n^k)$

A reduces to B

Input size: $O(n^k)$

**B**

Solvable in $O(n^r)$
Polynomial-Time Reductions

- Suppose that we know that $B \in \mathbf{P}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

<table>
<thead>
<tr>
<th>\hspace{1cm} A \hspace{1cm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\hspace{1cm} Solvable? \hspace{1cm}</td>
</tr>
</tbody>
</table>

Time required: $O(n^k)$

---

Input size: $O(n^k)$

<table>
<thead>
<tr>
<th>\hspace{1cm} B \hspace{1cm}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\hspace{1cm} Solvable in $O(n^r)$ \hspace{1cm}</td>
</tr>
</tbody>
</table>

A reduces to B

B decides A
Polynomial-Time Reductions

- Suppose that we know that \( B \in P \).
- Suppose that \( f \) is a polynomial-time reduction from \( A \) to \( B \).

\[
\begin{align*}
\text{Input size: } n & \quad & \text{Time required: } O(n^k) \\
A & \quad & \text{A reduces to } B \\
\text{Solvable?} & \quad & \text{B decides } A \\
\text{Input size: } O(n^k) & \quad & \text{Time required: } O(n^k + n^{kr}) \\
B & \quad & \text{Solvable in } O(n^r)
\end{align*}
\]
Polynomial-Time Reductions

- Suppose that we know that $B \in \mathbb{P}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

A

Solvable in $O(n^k + n^{kr})$

Time required: $O(n^k)$

A reduces to B

B

Solvable in $O(n^r)$

Time required: $O(n^k + n^{kr})$
Summary of $\mathbf{P}$

- $\mathbf{P}$ is the complexity class of yes/no questions that can be solved in polynomial time.
- $\mathbf{P}$ is closed under many operations, such as union and intersection.
- $\mathbf{P}$ is closed under polynomial-time reductions.
What *can't* you do in polynomial time?
How many simple paths are in this graph from the green node to the red node?
How many simple paths are in this graph from the green node to the red node?
How many simple paths are in this graph from the green node to the red node?
How many simple paths are in this graph from the green node to the red node?
How many subsets of this set are there?
How many binary search trees can you make for these numbers?
An Interesting Observation

- There are (at least) an exponential number of choices for each of the preceding objects.
- We cannot list all of them in polynomial time.
- However, each individual object being listed is quite short.
- This leads us to our next topic...
NTMs

- A **nondeterministic Turing machine** (NTM) is a generalization of the Turing machine.
- An NTM may have multiple transitions defined for a given state/symbol combination.
- The NTM accepts if **any** choice of transitions enters an accepting state.
- The NTM rejects if **all** choices of transitions enter a rejecting state.
- Otherwise, the NTM loops.
Nondeterminism Revisited

- If we add nondeterminism to the DFA, we get the NFA.
  - NFAs are no more powerful than DFAs.
- If we add nondeterminism to the DPDA, we get the PDA.
  - PDAs are more powerful than DPDAs.
- Adding nondeterminism to a TM produces the equivalently powerful NTM.
  - NTMs are no more powerful than TMs.
Nondeterminism Revisited

- Converting an NFA to a DFA might introduce exponentially more space.
- It is sometimes impossible to convert an NPDA to a DPDA.
- Converting an NTM to a TM might dramatically slow down the TM.
Designing NTMs

- Nondeterminism is a **very** powerful tool for solving problems.
- Many problems can be solved simply with nondeterminism using the following template:
  - **Nondeterministically** guess some important piece of information.
  - **Deterministically** check that the guess was correct.
Nondeterministic Algorithms

- Recall: a number is composite if it is not prime.
- Let $\Sigma = \{ 1 \}$ and consider the language

$$COMPOSITE = \{ 1^n \mid n \text{ is composite} \}$$
Nondeterministic Algorithms

- Recall: a number is composite if it is not prime.
- Let $\Sigma = \{1\}$ and consider the language $COMPOSITE = \{1^n \mid n \text{ is composite}\}$
- We can build a multitape, nondeterministic TM for $COMPOSITE$ as follows:
  - $M = \text{“On input } 1^n:\$
    - Nondeterministically write out $q$ 1s on a second tape ($2 \leq q < n$)
    - Nondeterministically write out $r$ 1s on a third tape ($2 \leq r < n$)
    - Deterministically check if $qr = n$.
    - If so, accept.
    - Otherwise, reject"
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

...
Nondeterministic Algorithms

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

...
Nondeterministic Algorithms

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

...
Nondeterministic Algorithms

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

- Compute $qr$ (Deterministic)
- Check if $n = qr$ (Deterministic)
- Guess $q$ and $r$ (Nondeterministic)
- Check if $n = qr$ (Deterministic)

```
1 1 1 1 1 1 1 1 1 1 1 1 ...
```

```
1 1 ...
```

```
1 1 1 1 1 1 ...
```

```
...
```
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 ...

1 1 ...

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

- Guess q and r (Nondeterministic)
- Compute qr (Deterministic)
- Check if n = qr (Deterministic)
Nondeterministic Algorithms

Compute \( qr \) (Deterministic)

Guess \( q \) and \( r \) (Nondeterministic)

Compute \( qr \) (Deterministic)

Check if \( n = qr \) (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 ....
### Nondeterministic Algorithms

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>…</td>
</tr>
</tbody>
</table>

1. **Guess q and r** (Nondeterministic)
2. **Compute qr** (Deterministic)
3. **Check if n = qr** (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

```
1 1 1 1 1 1 1 1 1 1 1 1 ...
```

```
1 1 ...
```

```
1 1 1 1 1 1 1 ...
```

```
1 1 1 1 1 1 1 ...
```

```
1 1 ...
```

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

Compute $qr$ (Deterministic)
Check if $n = qr$ (Deterministic)

Guess $q$ and $r$ (Nondeterministic)
Compute $qr$ (Deterministic)
Check if $n = qr$ (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1
### Nondeterministic Algorithms

| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ...
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 |   |   |   |   |   |   |   |   |   |   | ...
| 1 | 1 | 1 | 1 | 1 | 1 |   |   |   |   |   |   | ...
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | ...

- **Guess q and r (Nondeterministic)**
- **Compute qr (Deterministic)**
- **Check if n = qr (Deterministic)**
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

1 1 ...

Compute qr (Deterministic)

1 1 1 1 1 1 ...

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

…
Nondeterministic Algorithms

Compute $qr$ (Deterministic)

Check if $n = qr$ (Deterministic)

Guess $q$ and $r$ (Nondeterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1

1 1

1 1 1 1 1 1 1 1 1 1 1 1

1 1 1 1 1 1 1 1 1 1 1 1
Nondeterministic Algorithms

Guess $q$ and $r$ (Nondeterministic)

Compute $qr$ (Deterministic)

Check if $n = qr$ (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ... 

Guess q and r  
(Nondeterministic)

Compute qr  
(Deterministic)

Check if n = qr  
(Deterministic)

1 1 1 1 1 1 1 ... 

1 1 1 1 1 1 1 1 1 1 1 1 ... 

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1

1 1 1 1 1 1 1 ...  

1 1 1 1 1 1 1 ...  

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r  
(Nondeterministic)

Compute qr  
(Deterministic)

Check if n = qr  
(Deterministic)
Nondeterministic Algorithms

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

Compute qr (Deterministic)
Check if n = qr (Deterministic)

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

- Compute $qr$ (Deterministic)
- Check if $n = qr$ (Deterministic)
- Guess $q$ and $r$ (Nondeterministic)
- Compute $qr$ (Deterministic)
- Check if $n = qr$ (Deterministic)

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 …

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 …

1 1 1 1 1 1 1 …

1 1 1 1 1 1 1 …
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r
(Nondeterministic)

Compute qr
(Deterministic)

Check if n = qr
(Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...  

1 1 1 1 1 1 1 1 1 1 1 1 1 ...  

1 1 1 1 1 1 1 1 1 1 1 1 1 ...  

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)
Nondeterministic Algorithms

Compute $qr$ (Deterministic)

Check if $n = qr$ (Deterministic)

Guess $q$ and $r$ (Nondeterministic)

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\end{array}
\]

\[
\begin{array}{cccccccccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots \\
\end{array}
\]
Nondeterministic Algorithms

- Guess q and r (Nondeterministic)
- Compute qr (Deterministic)
- Check if n = qr (Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 ...

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...

Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)
Compute qr (Deterministic)
Check if n = qr (Deterministic)
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 ...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 ...

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...

...
Nondeterministic Algorithms

1 1 1 1 1 1 1 1 1 1 1 1 …

1 1 …

1 1 1 1 1 1 …

Guess q and r (Nondeterministic)

Compute qr (Deterministic)

Check if n = qr (Deterministic)

1 1 1 1 1 1 1 1 1 1 1 1 …
Analyzing NTMs

- When discussing deterministic TMs, time complexity was defined as the length of the longest computation on a string of length $n$.

- **Recall**: One way of thinking about nondeterminism is as a tree.

- In a **deterministic** computation, the tree linear; time complexity is the height of that line.
Analyzing NTMs

- When discussing deterministic TMs, time complexity was defined as the length of the longest computation on a string of length $n$.

- **Recall**: One way of thinking about nondeterminism is as a tree.

- The time complexity is the height of the tree (the length of the *longest* possible choice we could make).
Analyzing NTMs

• M = “On input $1^n$:
  • **Nondeterministically** write out q 1s on a second tape ($2 \leq q < n$)
  • **Nondeterministically** write out r 1s on a third tape ($2 \leq r < n$)
  • **Deterministically** check if $qr = n$.
  • If so, accept.
  • Otherwise, reject”
Analyzing NTMs

M = “On input $1^n$:

- **Nondeterministically** write out q 1s on a second tape ($2 \leq q < n$)
- **Nondeterministically** write out r 1s on a third tape ($2 \leq r < n$)
- **Deterministically** check if $qr = n$.
- If so, accept.
- Otherwise, reject”
Analyzing NTMs

• M = “On input $1^n$:
  • **Nondeterministically** write out q 1s on a second tape ($2 \leq q < n$)
  • **Nondeterministically** write out r 1s on a third tape ($2 \leq r < n$)
  • **Deterministically** check if $qr = n$.
  • If so, accept.
  • Otherwise, reject”
Analyzing NTMs

• M = “On input $1^n$:
  • Nondeterministically write out q 1s on a second tape ($2 \leq q < n$) 
  • Nondeterministically write out r 1s on a third tape ($2 \leq r < n$) 
  • Deterministically check if $qr = n$. 
  • If so, accept. 
  • Otherwise, reject”
Analyzing NTMs

- $M = \text{“On input } 1^n:\text{“}$
  - **Nondeterministically** write out $q$ 1s on a second tape ($2 \leq q < n$) \(O(n)\) steps
  - **Nondeterministically** write out $r$ 1s on a third tape ($2 \leq r < n$) \(O(n)\) steps
  - **Deterministically** check if $qr = n.$ \(O(n^2)\) steps
  - If so, accept.
  - Otherwise, reject"
Analyzing NTMs

- M = “On input $1^n$:
  - **Nondeterministically** write out q 1s on a second tape ($2 \leq q < n$) \(O(n)\) steps
  - **Nondeterministically** write out r 1s on a third tape ($2 \leq r < n$) \(O(n)\) steps
  - **Deterministically** check if $qr = n.$ \(O(n^2)\) steps
  - If so, accept.
  - Otherwise, reject” \(O(1)\) steps
Analyzing NTMs

• M = “On input $1^n$:
  • **Nondeterministically** write out $q$ 1s on a second tape ($2 \leq q < n$) \( O(n) \text{ steps} \)
  • **Nondeterministically** write out $r$ 1s on a third tape ($2 \leq r < n$) \( O(n) \text{ steps} \)
  • **Deterministically** check if $qr = n$. \( O(n^2) \text{ steps} \)
  • If so, accept.
  • Otherwise, reject” \( O(1) \text{ steps} \)
Analyzing NTMs

- M = “On input $1^n$:
  - Non-deterministically write out q 1s on a second tape ($2 \leq q < n$) \(O(n)\) steps
  - Non-deterministically write out r 1s on a third tape ($2 \leq r < n$) \(O(n)\) steps
  - Deterministically check if qr = n. \(O(n^2)\) steps
  - If so, accept.
  - Otherwise, reject” \(O(1)\) steps
Analyzing NTMs

- Our multitape NTM can decide \textit{COMPOSITE} in time $O(n^2)$.
- Using a similar construction to the deterministic case, a single-tape NTM can decide \textit{COMPOSITE} in $O(n^4)$.
- The best known deterministic algorithm for deciding \textit{COMPOSITE} runs \textit{much} more slowly.
  - Runs no faster than $O(n^{12})$.
- Just how much more powerful are NTMs?
From NTMs to TMs

- NTMs are at least as powerful as TMs.
  - Just don't use any nondeterminism!
- TMs are at least as powerful as NTMs.
  - Idea: Simulate the NTM with a multitape TM.
  - Run a breadth-first search on possible options.
From NTMs to TMs

- NTMs are at least as powerful as TMs.
  - Just don't use any nondeterminism!
- TMs are at least as powerful as NTMs.
  - Idea: Simulate the NTM with a multitape TM.
  - Run a breadth-first search on possible options.
From NTMs to TMs
From NTMs to TMs

- **Theorem**: For any NTM with time complexity $f(n)$, there is a TM with time complexity $2^{\Theta(f(n))}$.
- It is **unknown** whether it is possible to do any better than this.
- NTMs are capable of exploring multiple options in parallel; this “seems” inherently faster than deterministic computation.
TIME and NTIME

- **Recall**: TIME($f(n)$) is the class of languages that can be decided in $O(f(n))$ time by a single-tape TM.
- **NTIME($f(n)$)** is the class of languages that can be decided in $O(f(n))$ time by a single-tape NTM.
  - All possible options terminate in $O(f(n))$ steps.
  - For any $f(n)$, TIME($f(n)$) $\subseteq$ NTIME($f(n)$).
  - Can always convert a TM to an NTM.
The Complexity Class **NP**

- The complexity class **NP** (nondeterministic polynomial time) contains all problems that can be solved in polynomial time by a single-tape NTM.

- Formally:

\[
\text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k)
\]

- What types of problems are in **NP**?
A Problem in NP

• Does a Sudoku grid have a solution?
  • $M = \text{“On input } \langle S \rangle, \text{ an encoding of a Sudoku puzzle:} \quad$
    - Nondeterministically guess how to fill in all the squares.
    - Deterministically check whether the guess is correct.
    - If so, accept; if not, reject.”
A Problem in \textbf{NP}

- Does a Sudoku grid have a solution?
  - \textit{M = “On input \langle S \rangle, an encoding of a Sudoku puzzle:}
    - \textit{Nondeterministically} guess how to fill in all the squares.
    - \textit{Deterministically} check whether the guess is correct.
    - If so, accept; if not, reject.”

\begin{tabular}{cccc|cccc}
2 & 5 & 7 & 9 & 6 & 4 & 1 & 8 & 3 \\
4 & 9 & 1 & 8 & 7 & 3 & 6 & 5 & 2 \\
3 & 8 & 6 & 1 & 2 & 5 & 9 & 4 & 7 \\
\hline
6 & 4 & 5 & 7 & 3 & 2 & 8 & 1 & 9 \\
7 & 1 & 9 & 5 & 4 & 8 & 3 & 2 & 6 \\
8 & 3 & 2 & 6 & 1 & 9 & 5 & 7 & 4 \\
\hline
1 & 6 & 3 & 2 & 5 & 7 & 4 & 9 & 8 \\
5 & 7 & 8 & 4 & 9 & 6 & 2 & 3 & 1 \\
9 & 2 & 4 & 3 & 8 & 1 & 7 & 6 & 5 \\
\end{tabular}
A Problem in **NP**

- Does a Sudoku grid have a solution?
  - **M** = “On input \(\langle S\rangle\), an encoding of a Sudoku puzzle:
    - **Nondeterministically** guess how to fill in all the squares.
    - **Deterministically** check whether the guess is correct.
    - If so, accept; if not, reject.”

If we allow for a generalized Sudoku board of arbitrary size:

There are polynomially many grid cells to fill in.

Checking the grid takes polynomial time.

Overall algorithm takes polynomial time.
A Problem in $\textbf{NP}$

- A graph coloring is a way of assigning colors to nodes in an undirected graph such that no two nodes joined by an edge have the same color.
  - Applications in compilers, cell phone towers, etc.
- Question: Can graph $G$ be colored with at most $k$ colors?
A Problem in NP

- A **graph coloring** is a way of assigning colors to nodes in an undirected graph such that no two nodes joined by an edge have the same color.
  - Applications in compilers, cell phone towers, etc.
- Question: Can graph $G$ be colored with at most $k$ colors?
A Problem in \textbf{NP}

- A **graph coloring** is a way of assigning colors to nodes in an undirected graph such that no two nodes joined by an edge have the same color.
  - Applications in compilers, cell phone towers, etc.
- Question: Can graph $G$ be colored with at most $k$ colors?
- M = “On input $\langle G, k \rangle$:
  - \textbf{Nondeterministically} guess a coloring of the nodes.
  - \textbf{Deterministically} check whether it is legal.
  - If so, accept; if not, reject.”
A Problem in \textbf{NP}

- Suppose you want to start a delivery service.
- You want to place depots such that each customer is within some distance of the depot.
- Given a set of candidate locations for depots, can you place $k$ depots and guarantee that each customer is covered?

\begin{itemize}
  \item \texttt{M} = "On input $D$, $C$, $\delta$, $k$ (depot locations, customer locations, minimum distance required, and number of depots desired):
    \begin{itemize}
      \item Nondeterministically guess $k$ depots.
      \item Deterministically verify that each customer is within $\delta$ distance of some depot.
      \item If so, accept; otherwise reject."
\end{itemize}
\end{itemize}
A Problem in NP

- Suppose you want to start a delivery service.
- You want to place depots such that each customer is within some distance of the depot.
- Given a set of candidate locations for depots, can you place k depots and guarantee that each customer is covered?
A Problem in NP

- Suppose you want to start a delivery service.
- You want to place depots such that each customer is within some distance of the depot.
- Given a set of candidate locations for depots, can you place $k$ depots and guarantee that each customer is covered?
A Problem in \textbf{NP}

- Suppose you want to start a delivery service.
- You want to place depots such that each customer is within some distance of the depot.
- Given a set of candidate locations for depots, can you place \( k \) depots and guarantee that each customer is covered?
A Problem in NP

- Suppose you want to start a delivery service.
- You want to place depots such that each customer is within some distance of the depot.
- Given a set of candidate locations for depots, can you place k depots and guarantee that each customer is covered?
- M = “On input \( \langle D, C, \delta, k \rangle \) (depot locations, customer locations, minimum distance required, and number of depots desired):
  - **Nondeterministically** guess \( k \) depots.
  - **Deterministically** verify that each customer is within \( \delta \) distance of some depot.
  - If so, accept; otherwise reject.”
A General Pattern

• The NTMs we have seen so far always follow this pattern:
  • M = “On input $w$:
    – **Nondeterministically** guess some string $x$.
    – **Deterministically** check whether $x$ solves $w$.
    – If so, accept; otherwise, reject.”

• We use nondeterminism purely to guess the answer, and determinism purely to verify it.

• Is there a different way of characterizing $\text{NP}$?
Verification
Verification
Verification

Question: Can this lock be opened?
Verifiers

- Formally, a **verifier** is a TM $V$ such that
  $$w \in L \iff V \text{ accepts } \langle w, x \rangle \text{ for some } x \in \Sigma^*$$

- In other words
  $$L = \{ w \mid \exists x. V \text{ accepts } \langle w, x \rangle \}$$

- The string $x$ is called a **certificate** or **witness** for $w$, since it helps prove that $w \in L$.

- If $w \in L$, the verifier can check this if we know the right $x$.

- If $w \in L$ and we have the wrong $x$, the verifier does not help.
  - Even if $V$ rejects $\langle w, x \rangle$, it is still possible that $w \in L$.

- If $w \notin L$, the verifier does not help us.
  - We would have to check all possible $x$ before confirming $w \notin L$. 
Verification is Powerful

- Many undecidable languages can still be verified.
- Here is a verifier for $A_{\text{TM}}$:
  - $V = \text{“On input } \langle M, w, n \rangle, \text{ where } M \text{ is a TM, } w \text{ is a string, and } n \text{ is a natural number:} $
    - \text{Run } M \text{ on } w \text{ for } n \text{ steps.}
    - \text{If } M \text{ accepts } w \text{ within that time, accept; otherwise reject.”}
- $V$ always halts on all inputs (even if $M$ loops on $w$).
- If $M$ accepts $w$, there is some choice of $n$ for which $H$ accepts $\langle M, w, n \rangle$ (namely, the number of steps $M$ takes before it accepts $w$).
- Thus $A_{\text{TM}}$ can be verified but not decided.
Polynomial-Time Verifiers

• A **polynomial-time verifier** is a TM $V$ for a language $L$ such that

  $$w \in L \iff \exists x. \text{ } V \text{ accepts } \langle w, x \rangle$$

  and $V$ runs in time polynomial in $w$.

• That is, a normal verifier that runs in polynomial time.
A Problem in \textbf{NP}

• Does a Sudoku grid have a solution?

\[ V = \text{"On input } \langle S, A \rangle \text{, an encoding of a Sudoku puzzle and an alleged solution to it:}\]

– Deterministically check whether the solution is correct.
– If so, accept; if not, reject."
Question: Can graph $G$ be colored with at most $k$ colors?

$V$ = “On input $\langle G, k, C \rangle$, where $C$ is an alleged $k$-coloring of graph $G$:

- **Deterministically** check whether $C$ is a legal $k$-coloring of $G$.
- If so, accept; if not, reject.”
Two Equivalent Formulations of $\textbf{NP}$

• **Theorem**: A language $L$ has a polynomial-time verifier iff $L \in \textbf{NP}$.

• **Proof sketch**:
  
  • Any polynomial-time verifier can be turned into a polynomial-time NTM by having the NTM nondeterministically guess the certificate for $w$, then check it (deterministically) by running the verifier.
  
  • If an NTM can decide $L$ in polynomial time, a verifier could work by having a certificate saying which nondeterministic choices the original NTM made, then simulating those choices of the NTM to check it.
A Problem in \textbf{NP}

- Given an undirected graph $G = (V, E)$, a \textit{k-clique} in $G$ is a set of $k$ nodes where each node has an edge to each other node in the clique.

- \textbf{Question}: Does $G$ contain a $k$-clique?
A Problem in NP

- Given an undirected graph $G = (V, E)$, a k-clique in $G$ is a set of $k$ nodes where each node has an edge to each other node in the clique.

- Question: Does $G$ contain a $k$-clique?
A Problem in NP

• Given an undirected graph $G = (V, E)$, a \textbf{k-clique} in $G$ is a set of $k$ nodes where each node has an edge to each other node in the clique.

• \textbf{Question}: Does $G$ contain a $k$-clique?
A Problem in \textbf{NP}

- Given an undirected graph $G = (V, E)$, a \textbf{k-clique} in $G$ is a set of $k$ nodes where each node has an edge to each other node in the clique.

- \textbf{Question}: Does $G$ contain a $k$-clique?

- $V =$ “On input $\langle G, k, C \rangle$:
  - \textbf{Deterministically} verify that $|C| = k$.
  - \textbf{Deterministically} verify that each node in $C$ has an edge to each other node in $C$.
  - If so, accept; otherwise reject.”
A Problem in \textbf{NP}

- Given a propositional logic formula $\varphi$, is $\varphi$ satisfiable?
  - This is the SAT problem.
A Problem in **NP**

• Given a propositional logic formula \( \varphi \), is \( \varphi \) satisfiable?
  • This is the SAT problem.
• \( \mathcal{V} = \) “On input \( \langle \varphi, A \rangle \), where \( \varphi \) is a propositional logic formula and \( A \) is an alleged satisfying assignment:
  **Deterministically** evaluate \( \varphi \) using \( A \).
  If \( \varphi \) evaluates to true, accept.
  Otherwise, reject.”
NP
Proving Languages are in $\textbf{NP}$

- **Build a polynomial-time NTM for $L$.**
  - Build an NTM for the language $L$.
  - Prove that it runs in nondeterministic time $O(n^k)$.

- **Build a polynomial-time verifier for $L$.**
  - Build a TM that verifies a string, given a certificate.
  - Prove that it runs in deterministic time $O(n^k)$.

- **Use closure properties.**
  - Prove that the language can be formed by appropriate transformations of languages in $\textbf{NP}$.

- **Reduce the language to a language in $\textbf{NP}$.**
  - Show how a polynomial-time verifier or polynomial-time NTM for some language $L'$ can be used to decide $L$. 
Proving Languages are in \textbf{NP}

Build a polynomial-time NTM for \(L\).
- Build an NTM for the language \(L\).
- Prove that it runs in nondeterministic time \(O(n^k)\).

Build a polynomial-time verifier for \(L\).
- Build a TM that verifies a string, given a certificate.
- Prove that it runs in deterministic time \(O(n^k)\).

Use closure properties.
- Prove that the language can be formed by appropriate transformations of languages in \textbf{NP}.

- \textbf{Reduce the language to a language in NP.}
  - Show how a polynomial-time verifier or polynomial-time NTM for some language \(L'\) can be used to decide \(L\).
Polynomial-Time Reductions

- Suppose that we know that $B \in \text{NP}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$. 

A

Solvable?
Polynomial-Time Reductions

- Suppose that we know that \( B \in \textbf{NP} \).
- Suppose that \( f \) is a polynomial-time reduction from \( A \) to \( B \).
Polynomial-Time Reductions

• Suppose that we know that $B \in \textbf{NP}$.
• Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

\begin{itemize}
  \item \textbf{A}
  \begin{itemize}
    \item Solvable?
  \end{itemize}
  \item \textbf{B}
  \begin{itemize}
    \item Solvable in $O(n^r)$ by NTM
  \end{itemize}
\end{itemize}
Polynomial-Time Reductions

- Suppose that we know that $B \in \text{NP}$.  
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.  

Input size: $n$

A (Solvable?) reduces to B (Solvable in $O(n^r)$ by NTM)
Polynomial-Time Reductions

• Suppose that we know that $B \in \textbf{NP}$.
• Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

Time required: $O(n^k)$

A reduces to B

A

Solvable?

B

Solvable in $O(n^r)$ by NTM
Polynomial-Time Reductions

- Suppose that we know that $B \in \text{NP}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

Input size: $O(n^k)$

Time required: $O(n^k)$

A reduces to B

A Solvable?

B Solvable in $O(n^r)$ by NTM
Polynomial-Time Reductions

- Suppose that we know that $B \in \textbf{NP}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Input size: $n$

- Solvable?

Time required: $O(n^k)$

Input size: $O(n^k)$

- Solvable in $O(n^r)$ by NTM

A reduces to B

B decides A
Polynomial-Time Reductions

- Suppose that we know that $B \in \textbf{NP}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

$A$ reduces to $B$

$B$ decides $A$

\begin{align*}
\text{Input size: } n \\
\text{Time required: } O(n^k)
\end{align*}

\begin{align*}
\text{Time required: } O(n^k + n^{kr})
\end{align*}
Polynomial-Time Reductions

- Suppose that we know that $B \in \textbf{NP}$.
- Suppose that $f$ is a polynomial-time reduction from $A$ to $B$.

Inputs:
- $A$: Solvable in $O(n^k)$ by NTM
- $B$: Solvable in $O(n^r)$ by NTM

Time required:
- $O(n^k + n^{kr})$
Trust me, these reductions matter.

We'll see why later.
And now...
The Most Important Question in Theoretical Computer Science
What is the connection between \( P \) and \( NP \)?
\[ P = \bigcup_{k=0}^{\infty} \text{TIME}(n^k) \]

\[ NP = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k) \]
\[ P = \bigcup_{k=0}^{\infty} \text{TIME}(n^k) \]

\[ \text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k) \]

\[ \text{TIME}(n^k) \subseteq \text{NTIME}(n^k) \]
\[ P = \bigcup_{k=0}^{\infty} \text{TIME}(n^k) \]

\[ \text{NP} = \bigcup_{k=0}^{\infty} \text{NTIME}(n^k) \]

\[ \text{TIME}(n^k) \subseteq \text{NTIME}(n^k) \]

\[ P \subseteq \text{NP} \]
Which Picture is Correct?
Which Picture is Correct?

P

NP
Does $P = NP$?
The question of whether $P \cong NP$ is the most important question in all of theoretical computer science.

With the verifier definition of NP, one way of phrasing this question is

If a problem can be verified efficiently, can it be solved efficiently?

An answer either way will give fundamental insights into the nature of computation.
Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem)
  - Determining whether a simple DNA strand exists that two gene sequences could be a part of (shortest common supersequence)
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation)
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling)
  - And many more.

- If $P = NP$, all of these problems have efficient solutions.
- If $P \neq NP$, none of these problems have efficient solutions.
- These are problems we want to solve on a daily basis.
Why This Matters

- The following problems are known to be efficiently verifiable, but have no known efficient solutions:
  - Determining whether an electrical grid can be built to link up some number of houses for some price (Steiner tree problem)
  - Determining whether a simple DNA strand exists that two gene sequences could be a part of (shortest common supersequence)
  - Determining the best way to assign hardware resources in a compiler (optimal register allocation)
  - Determining the best way to distribute tasks to multiple workers to minimize completion time (job scheduling)
  - And many more.
- If $P = NP$, all of these problems have efficient solutions.
- If $P \neq NP$, none of these problems have efficient solutions.
- These are problems we need to solve on a daily basis.
Why This Matters

• If $P = NP$:  
  • A huge number of seemingly difficult problems could be solved efficiently.
  • Our capacity to solve many problems will scale well with the size of the problems we want to solve.

• If $P \neq NP$:
  • Enormous computational power would be required to solve many seemingly easy tasks.
  • Our capacity to solve many problems will fail to keep up with our curiosity.
What We Know

- Trying to prove $P \neq NP$ has been extremely difficult.
- In the past 35 years:
  - Not a single correct proof either way has been found.
  - Many types of proofs have been shown to be insufficiently powerful to determine whether $P = NP$.
  - It is commonly believed that $P \neq NP$, but no one knows for sure.
- Interesting read: Interviews with leading computer scientists about $P \neq NP$:
The Million-Dollar Question

The Clay Mathematics Institute has offered a $1,000,000 prize to anyone who proves or disproves $P = NP$. 
The Clay Mathematics Institute has offered a $1,000,000 prize to anyone who proves or disproves $P = NP$. 
NP-Completeness
Polynomial-Time Reductions

- If $L_1 \leq_p L_2$ and $L_2 \in \text{P}$, then $L_1 \in \text{P}$ as well.
Polynomial-Time Reductions

- If $L_1 \leq_p L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$ as well.
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{P}$, then $L_1 \in \mathbf{P}$ as well.

- If $L_1 \leq_P L_2$ and $L_2 \in \mathbf{NP}$, then $L_1 \in \mathbf{NP}$ as well.
Polynomial-Time Reductions

- If $L_1 \leq_p L_2$ and $L_2 \in \text{P}$, then $L_1 \in \text{P}$ as well.
- If $L_1 \leq_p L_2$ and $L_2 \in \text{NP}$, then $L_1 \in \text{NP}$ as well.
Polynomial-Time Reductions

- If $L_1 \leq_p L_2$ and $L_2 \in \textbf{P}$, then $L_1 \in \textbf{P}$ as well.
- If $L_1 \leq_p L_2$ and $L_2 \in \textbf{NP}$, then $L_1 \in \textbf{NP}$ as well.
Polynomial-Time Reductions

- If $L_1 \leq_P L_2$ and $L_2 \in \textbf{P}$, then $L_1 \in \textbf{P}$ as well.
- If $L_1 \leq_P L_2$ and $L_2 \in \textbf{NP}$, then $L_1 \in \textbf{NP}$ as well.
NP-Hardness

- A language $L$ is called **NP-hard** iff for every language $L' \in \textbf{NP}$, we have that $L' \leq_p L$. 

\[ P \subseteq \textbf{NP} \]

\[ \textbf{NP} \cap \textbf{NP-Hard} \]
A language $L$ is called $\textbf{NP-hard}$ iff for every language $L' \in \textbf{NP}$, we have that $L' \leq_p L$.
A language $L$ is called **NP-hard** iff for every language $L' \in \text{NP}$, we have that $L' \leq_P L$.

The class of NP-complete problems is called $\text{NPC}$. 

**NP-Hardness**
A language $L$ is called **NP-hard** iff for every language $L' \in \text{NP}$, we have that $L' \leq_p L$.

A language $L$ is called **NP-complete** iff $L$ is NP-hard and $L \in \text{NP}$.

The class of NP-complete problems is called **NPC**.
A language $L$ is called **NP-hard** iff for every language $L' \in \text{NP}$, we have that $L' \leq_p L$.

A language $L$ is called **NP-complete** iff $L$ is NP-hard and $L \in \text{NP}$.

The class of NP-complete problems is called **NPC**.
A language $L$ is called **NP-hard** iff for every language $L' \in \textbf{NP}$, we have that $L' \leq_p L$.

A language $L$ is called **NP-complete** iff $L$ is NP-hard and $L \in \textbf{NP}$.

The class of NP-complete problems is called **NPC**.
NP-Hardness

- A language $L$ is called **NP-hard** iff for every language $L' \in \text{NP}$, we have that $L' \leq_p L$.

- A language $L$ is called **NP-complete** iff $L$ is NP-hard and $L \in \text{NP}$.

- The class of NP-complete problems is called **NPC**.
The Tantalizing Truth

- **Theorem**: If any NP-complete language is in P, then $P = NP$. 
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in P, then P = NP.
- **Proof**: If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_P L \) because \( L \) is NP-complete.
The Tantalizing Truth

- **Theorem**: If any NP-complete language is in $P$, then $P = NP$.
- **Proof**: If $L \in NPC$ and $L \in P$, we know that for any $L' \in NP$, that $L' \leq_P L$ because $L$ is NP-complete. Since $L' \leq_P L$ and $L \in P$, this means that $L' \in P$. 
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in P, then P = NP.
- **Proof**: If \( L \in \text{NPC} \) and \( L \in \text{P} \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_P L \) because \( L \) is NP-complete. Since \( L' \leq_P L \) and \( L \in \text{P} \), this means that \( L' \in \text{P} \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in \text{P} \), so \( \text{NP} \subseteq \text{P} \).
The Tantalizing Truth

Theorem: If any NP-complete language is in P, then P = NP.

Proof: If \( L \in \text{NPC} \) and \( L \in \text{P} \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in \text{P} \), this means that \( L' \in \text{P} \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in \text{P} \), so \( \text{NP} \subseteq \text{P} \). Since \( \text{P} \subseteq \text{NP} \), this means \( \text{P} = \text{NP} \).
The Tantalizing Truth

- **Theorem**: If any NP-complete language is in P, then P = NP.

- **Proof**: If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_P L \) because \( L \) is NP-complete. Since \( L' \leq_P L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in P \), so \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \), this means \( P = \text{NP} \). ■
The Tantalizing Truth

- **Theorem:** If any NP-complete language is in P, then P = NP.

- **Proof:** If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in P \), so \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \), this means \( P = \text{NP} \). ■
The Tantalizing Truth

- **Theorem:** If *any* NP-complete language is in \( \mathbf{P} \), then \( \mathbf{P} = \mathbf{NP} \).
- **Proof:** If \( L \in \mathbf{NPC} \) and \( L \in \mathbf{P} \), we know that for any \( L' \in \mathbf{NP} \), that \( L' \leq_p L \) because \( L \) is \( \mathbf{NP} \)-complete. Since \( L' \leq_p L \) and \( L \in \mathbf{P} \), this means that \( L' \in \mathbf{P} \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \mathbf{NP} \) satisfies \( L' \in \mathbf{P} \), so \( \mathbf{NP} \subseteq \mathbf{P} \). Since \( \mathbf{P} \subseteq \mathbf{NP} \), this means \( \mathbf{P} = \mathbf{NP} \). ■
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in $P$, then $P = NP$.

- **Proof**: If $L \in NPC$ and $L \in P$, we know that for any $L' \in NP$, that $L' \leq_p L$ because $L$ is NP-complete. Since $L' \leq_p L$ and $L \in P$, this means that $L' \in P$. Since our choice of $L'$ was arbitrary, any language $L' \in NP$ satisfies $L' \in P$, so $NP \subseteq P$. Since $P \subseteq NP$, this means $P = NP$. ■
The Tantalizing Truth

- **Theorem**: If *any* NP-complete language is in P, then P = NP.
- **Proof**: If \( L \in \text{NPC} \) and \( L \in P \), we know that for any \( L' \in \text{NP} \), that \( L' \leq_p L \) because \( L \) is NP-complete. Since \( L' \leq_p L \) and \( L \in P \), this means that \( L' \in P \). Since our choice of \( L' \) was arbitrary, any language \( L' \in \text{NP} \) satisfies \( L' \in P \), so \( \text{NP} \subseteq P \). Since \( P \subseteq \text{NP} \), this means \( P = \text{NP} \). ■
The Tantalizing Truth

- **Theorem:** If *any* NP-complete language is not in $P$, then $P \neq NP$.

- **Proof:** If $L$ is NP-complete, it is in NP. If it is in NP but not P, then $P \neq NP$. ■
An Feel for NP-Completeness

• If a problem is \textbf{NP}-complete, under the (commonly-held) assumption that \( P \neq \text{NP} \), then there cannot be an efficient algorithm for it.

• In a sense, \textbf{NP}-complete problems are the hardest problems in \textbf{NP}.

• Because it is conjectured that \( P \neq \text{NP} \), \textbf{NP}-complete problems tend to be enormously difficult to solve.
  • All known algorithms for \textbf{NP}-complete problems run in exponential time.
  • All known algorithms for \textbf{NP}-complete problems are infeasible for any reasonably-sized inputs.
What Problems are NP-Complete?

• NP-complete problems give a promising approach for proving or disproving that $P = NP$:
  • If any \textbf{NPC} problem is in $P$, then $P = NP$.
  • If any \textbf{NPC} problem is not in $P$, then $P \neq NP$.
• However, we haven't shown that any problems are NP-complete in the first place!
• How do we even know they exist?
Why This Matters

- When discussing computability theory, we saw that these questions are undecidable:
  - Does $M$ halt on $w$?
  - Is $\mathcal{L}(M)$ regular?
  - Is $\mathcal{L}(M)$ infinite?
  - Does $M$ accept a string of length at most three?
  - Does $M$ accept every string?
- These are all questions about TMs.
- Though these problems sometimes arise in practice, it's unlikely that you will bump into an undecidable problem unless you're asking questions about what a program does.
Why This Matters

• In contrast, NP-complete problems are everywhere.
  • Can you buy five cell phone towers to guarantee everyone in a city has service?
  • Is there a sequence of 100 genes that is common to some group of DNA strands?
  • Given a list of prices and a budget, can you buy over fifteen items?
  • Given a city layout, can you use 100,000 feet of piping to supply water to each building?

• These are questions we want to solve on a daily basis.
Next Time

• NP-Complete Problems
  • What problems are known to be NP-complete?
  • How do you prove that a problem is NP-complete?