Announcements

- **Welcome back!**
- Lecture 23 video should be posted by the end of tonight.
  - Sorry for not getting it up sooner!
- Problem session tonight in 380-380X from 7:00PM – 7:50PM.
  - Optional, but highly recommended.
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
WELCOME TO THEORYLAND
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
It may be that since one is customarily concerned with existence, [...] finiteness, and so forth, one is not inclined to take seriously the question of the existence of a better-than-finite algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
It may be that since one is customarily concerned with existence, [...] decidability, and so forth, one is not inclined to take seriously the question of the existence of a better-than-decidable algorithm.

- Jack Edmonds, “Paths, Trees, and Flowers”
A Decidable Problem

- **Presburger arithmetic** is a logical system for reasoning about arithmetic.
  - \( \forall x. x + 1 \neq 0 \)
  - \( \forall x. \forall y. (x + 1 = y + 1 \rightarrow x = y) \)
  - \( \forall x. x + 0 = x \)
  - \( \forall x. \forall y. (x + y) + 1 = x + (y + 1) \)
  - \( \forall x. ((P(0) \land \forall y. (P(y) \rightarrow P(y + 1)))) \rightarrow \forall x. P(x) \)

- Given a statement, it is decidable whether that statement can be proven from the laws of Presburger arithmetic.

- Any Turing machine that decides whether a statement in Presburger arithmetic is true or false has to move the tape head at least \( 2^{2^{cn}} \) times on some inputs of length \( n \) (for some fixed constant \( c \)).
For Reference

• Assume $c = 1$.

\[ 2^{2^0} = 2 \]
For Reference

- Assume $c = 1$.

\[ 2^{2^0} = 2 \]
\[ 2^{2^1} = 4 \]
For Reference

• Assume $c = 1$.

\[
\begin{align*}
2^0 &= 2 \\
2^1 &= 4 \\
2^2 &= 16
\end{align*}
\]
For Reference

• Assume $c = 1$.

\[
2^0 = 2 \\
2^1 = 4 \\
2^2 = 16 \\
2^3 = 256
\]
For Reference

- Assume $c = 1$.

\[
\begin{align*}
2^0 &= 2 \\
2^1 &= 4 \\
2^2 &= 16 \\
2^3 &= 256 \\
2^4 &= 65536
\end{align*}
\]
For Reference

- Assume $c = 1$.

\[
2^0 = 2 \\
2^1 = 4 \\
2^2 = 16 \\
2^3 = 256 \\
2^4 = 65536 \\
2^5 = 18446744073709551616
\]
For Reference

• Assume $c = 1$.

\[
\begin{align*}
2^0 &= 2 \\
2^1 &= 4 \\
2^2 &= 16 \\
2^3 &= 256 \\
2^4 &= 65536 \\
2^5 &= 18446744073709551616 \\
2^6 &= 340282366920938463463374607431768211456
\end{align*}
\]
The Limits of Decidability

- The fact that a problem is decidable does not mean that it is feasibly decidable.

- In computability theory, we ask the question

  Is it possible to solve problem $L$?

- In complexity theory, we ask the question

  Is it possible to solve problem $L$ efficiently?

- In the remainder of this course, we will explore this question in more detail.
Regular Languages

CFLs

DCFLs

All Languages

R

RE

Regular Languages

DCFLs

CFLs

All Languages
The Setup

• In order to study computability, we needed to answer these questions:
  • What is “computation?”
  • What is a “problem?”
  • What does it mean to “solve” a problem?

• To study complexity, we need to answer these questions:
  • What does “complexity” even mean?
  • What is an “efficient” solution to a problem?
Measuring Complexity

- Suppose that we have a decider $D$ for some language $L$.
- How might we measure the complexity of $D$?
Measuring Complexity

- Suppose that we have a decider $D$ for some language $L$.
- How might we measure the complexity of $D$?
  - Number of states.
  - Size of tape alphabet.
  - Size of input alphabet.
  - Amount of tape required.
  - Number of steps required.
  - Number of times a given state is entered.
  - Number of times a given symbol is printed.
  - Number of times a given transition is taken.
  - (Plus a whole lot more...)
Measuring Complexity

• Suppose that we have a decider $D$ for some language $L$.
• How might we measure the complexity of $D$?
  
  Number of states.
  Size of tape alphabet.
  Size of input alphabet.

  • Amount of tape required.
  • Number of steps required.
  
    Number of times a given state is entered.
    Number of times a given symbol is printed.
    Number of times a given transition is taken.
(Plus a whole lot more...)
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
</tr>
<tr>
<td></td>
<td>reject</td>
<td></td>
<td>accept</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>q_1</td>
<td>reject</td>
<td>1</td>
</tr>
</tbody>
</table>
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>q_1</td>
<td>reject</td>
<td>1</td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>q_1</td>
<td>reject</td>
<td>1</td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

### Step Counter

<table>
<thead>
<tr>
<th>Step Counter</th>
<th>0</th>
</tr>
</thead>
</table>

| 0 | 1 | 0 | 1 | 0 |

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>R</th>
<th>q₁</th>
<th>reject</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
</tr>
</tbody>
</table>

- A Turing machine halts when it reaches the accept state.
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>R</th>
<th>q₁</th>
<th>reject</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>

Step Counter: 0
**Time Complexity**

- A **step** of a Turing machine is one event where the TM takes a transition.

- Running a TM on different inputs might take a different number of steps.

---

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_0</td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td>q_1</td>
<td>reject</td>
<td>1</td>
</tr>
</tbody>
</table>

---

Step Counter

1
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.

- Running a TM on different inputs might take a different number of steps.
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
</tr>
</tbody>
</table>

Step Counter: 2
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>q_0</th>
<th>0</th>
<th>R</th>
<th>q_1</th>
<th>reject</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>q_1</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q_0</td>
<td>accept</td>
</tr>
</tbody>
</table>

Step Counter: 3
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>R</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>

Step Counter: 3
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Step Counter: 4
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>Step Counter</th>
<th>4</th>
</tr>
</thead>
</table>
| \[ \begin{array}{|c|c|c|c|}
| 0 | 1 | 0 | 1 | 0 | \end{array} \] |  |
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

```
0 1 0 1 0

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>
```

Step Counter: 5
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

### Step Counter

<table>
<thead>
<tr>
<th>q₀</th>
<th>0</th>
<th>R</th>
<th>q₁</th>
<th>reject</th>
<th>accept</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

---

<table>
<thead>
<tr>
<th>Step Counter</th>
<th>6</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>Step Counter</th>
<th>6</th>
</tr>
</thead>
</table>

$egin{array}{ccc}
0 & 1 & 0 \\
0 & R & q_1 \\
q_0 & \text{reject} & \text{accept} \\
q_1 & \text{reject} & 1 \ R \ q_0 \\
\end{array}$

Accepting means transitioning into a special state.
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Step</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Counter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q&lt;sub&gt;0&lt;/sub&gt;</td>
<td>0</td>
<td>R</td>
<td>q&lt;sub&gt;1&lt;/sub&gt;</td>
</tr>
<tr>
<td>q&lt;sub&gt;1&lt;/sub&gt;</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>q₀</strong></td>
<td>0</td>
<td>R</td>
</tr>
<tr>
<td><strong>q₁</strong></td>
<td>reject</td>
<td>1</td>
</tr>
</tbody>
</table>

Step Counter: 0
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>R</td>
<td>$q_1$</td>
<td>reject</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>$q_1$</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>$q_0$</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Step Counter: 1
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
</tr>
</tbody>
</table>

Step Counter: 1
Time Complexity

• A **step** of a Turing machine is one event where the TM takes a transition.

• Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>q₀</td>
<td>0</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
</tr>
</tbody>
</table>

Step Counter 2
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
<td>reject</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
<td>q₀</td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>

Step Counter: 2
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>Step Counter</th>
<th>0</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>q_0</strong></td>
<td>0</td>
<td>R</td>
<td><strong>q_1</strong></td>
<td><strong>reject</strong></td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td><strong>q_1</strong></td>
<td><strong>reject</strong></td>
<td>1</td>
<td>R</td>
<td><strong>q_0</strong></td>
<td>accept</td>
<td></td>
</tr>
</tbody>
</table>
Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>q_0</strong></td>
<td>0</td>
<td>R</td>
<td><strong>q_1</strong></td>
</tr>
<tr>
<td><strong>q_1</strong></td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>

Step Counter: 0
Time Complexity

- A step of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>q₀</td>
<td>0</td>
<td>R</td>
<td>q₁</td>
</tr>
<tr>
<td></td>
<td>reject</td>
<td>accept</td>
<td></td>
</tr>
<tr>
<td>q₁</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
<tr>
<td></td>
<td>accept</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Step Counter: 0
### Time Complexity

- A **step** of a Turing machine is one event where the TM takes a transition.
- Running a TM on different inputs might take a different number of steps.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>R</td>
<td>reject</td>
</tr>
<tr>
<td>$q_1$</td>
<td>reject</td>
<td>1</td>
<td>R</td>
</tr>
</tbody>
</table>

Step Counter: 1
Time Complexity

- The number of steps a TM takes on some input is sensitive to
  - The structure of that input.
  - The length of the input.
- How can we come up with a consistent measure of a machine's runtime?
Time Complexity

- The **time complexity** of a TM $M$ is a function (typically denoted $f(n)$) that measures the *worst-case* number of steps $M$ takes on any input of length $n$.
  - By convention, $n$ denotes the length of the input.
  - If $M$ loops on some input of length $k$, then $f(k) = \infty$.
- The previous TM has time complexity $f(n) = n + 1$.
  - Any input of length $n$ of the form $01010...$ halts after $n + 1$ steps.
  - Some inputs may take less time to halt, but time complexity considers the worst-case complexity.
A Slight Problem

• Consider the following TM over $\Sigma = \{0, 1\}$ for the language $BALANCE = \{ w \in \Sigma^* | w$ has the same number of 0s and 1s $\}$:

  • $M = \text{“On input } w:\text{”}$
    - Scan across the tape until a 0 or 1 is found.
    - If none are found, accept.
    - If one is found, continue scanning until a matching 1 or 0 is found.
    - If none is found, reject.
    - Otherwise, cross off that symbol and repeat.”

• What is the time complexity of $M$?
A Loss of Precision

• When considering *computability*, using high-level TM descriptions is perfectly fine.

• When considering *complexity*, high-level TM descriptions make it nearly impossible to precisely reason about the actual time complexity.

• What are we to do about this?
The Best We Can

\[ M = \text{“On input } w:\text{”} \]

- Scan across the tape until a 0 or 1 is found. \text{At most } n \text{ steps.}
- If none are found, accept. \text{At most 1 step.}
- If one is found, continue scanning until a matching 1 or 0 is found. \text{At most } n \text{ more steps.}
- If none are found, reject. \text{At most 1 step}
- Otherwise, cross off that symbol and repeat.” \text{At most } n \text{ steps to get back to the start of the tape.}

\[ + \]
\[ \text{At most } 3n + 2 \text{ steps.} \]
\[ \times \]
\[ \text{At most } n/2 \text{ loops.} \]
\[ \text{At most } 3n^2 / 2 + n \text{ steps.} \]
An Easier Approach

- In complexity theory, we rarely need an exact value for a TM's time complexity.
- Usually, we are curious with the long-term growth rate of the time complexity.
- For example, if the time complexity is $3n + 5$, then doubling the length of the string roughly doubles the worst-case runtime.
- If the time complexity is $2^n - n^2$, since $2^n$ grows much more quickly than $n^2$, for large values of $n$, increasing the size of the input by 1 doubles the worst-case running time.
Big-O Notation

- Ignore *everything* except the dominant growth term, including constant factors.

- Examples:
  - $4n + 4 = \mathcal{O}(n)$
  - $137n + 271 = \mathcal{O}(n)$
  - $n^2 + 3n + 4 = \mathcal{O}(n^2)$
  - $2^n + n^3 = \mathcal{O}(2^n)$
  - $137 = \mathcal{O}(1)$
  - $n^2 \log n + \log^5 n = \mathcal{O}(n^2 \log n)$
Let $f : \mathbb{N} \rightarrow \mathbb{N}$ and $g : \mathbb{N} \rightarrow \mathbb{N}$.

Then $f(n) = O(g(n))$ iff there exist constants $c \in \mathbb{R}$ and $n_0 \in \mathbb{N}$ such that

$$\text{For any } n \geq n_0, f(n) \leq cg(n)$$

Intuitively, as $n$ gets “large” (greater than $n_0$), $f(n)$ is bounded from above by some multiple (determined by $c$) of $g(n)$.  

Big-O Notation, Formally
Properties of Big-O Notation

- **Theorem**: If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \), then \( f_1(n) + f_2(n) = O(g_1(n) + g_2(n)) \).

  - Intuitively: If you run two programs one after another, the big-O of the result is the big-O of the sum of the two runtimes.

- **Theorem**: If \( f_1(n) = O(g_1(n)) \) and \( f_2(n) = O(g_2(n)) \), then \( f_1(n)f_2(n) = O(g_1(n)g_2(n)) \).

  - Intuitively: If you run one program some number of times, the big-O of the result is the big-O of the program times the big-O of the number of iterations.

- This makes it substantially easier to analyze time complexity, though we do lose some precision.
$M = \text{“On input } w:\text{ “}$

- Scan across the tape until a 0 or 1 is found. $\mathcal{O}(n)$ steps
- If none are found, accept. $\mathcal{O}(1)$ steps
- If one is found, continue scanning until a matching 1 or 0 is found. $\mathcal{O}(n)$ steps
- If none is found, reject. $\mathcal{O}(1)$ steps
- Otherwise, cross off that symbol and repeat.” $\mathcal{O}(n)$ loops

$\mathcal{O}(n^2)$ steps
MTTMs

- A **multitape Turing machine** (MTTM) is a Turing machine with multiple tapes.
- The input tape holds the original input.
- Each tape head can move independently of the rest.
- Each tape head can base its transition on the symbols under all tape heads.
An MTTM for BALANCE

$M_2 = "\text{On input } w:\"

- Scan across the tape and copy all $1$s to a secondary tape. $O(n)$ steps
- Move both tape heads back to the start of their tapes. $O(n)$ steps
- Until the end of the input is reached:
  - Scan on the input tape until a $0$ is found. $O(n)$ steps
  - Match the $0$ with a $1$ on the second tape.
- If an imbalance is found, reject. $O(1)$ steps
- If all $0$s and $1$s are matched, accept. $O(1)$ steps

$O(n)$ steps.
A Performance Comparison

- Our original 1-tape TM for $BALANCE$ runs in $O(n^2)$ time.
- Our MTTM can decide $BALANCE$ in $O(n)$ time.
- **Nontrivial result:** There is no single-tape TM that can decide $BALANCE$ in $O(n)$ time.
- **The MTTM is inherently faster than the single-tape TM!**
Complexity is Tricky

- The Church-Turing thesis states that any feasible model of computation is no more powerful than a TM.
- However, some models of computation might be more efficient than the TM.
- When analyzing complexity, the model of computation matters!
Analyzing Efficiency

• We need to reason about the efficiency of our TM equivalents.

• Questions worth considering:
  • If there is a MTTM for $L$ that runs in time $f(n)$, can we find a TM for $L$ that runs in time $f(n)$? $f(n)^2$? $f(n)^3$?
  • If there is a WB program for $L$ that runs in time $f(n)$, can we find a TM for $L$ that runs in time $f(n)$? $f(n)^2$? $f(n)^3$?
Our Line of Reasoning

- To analyze the relative efficiencies of MTTMs, \textbf{WB} programs, and TMs, we will do the following:
  - Show how much slowdown we get when converting a \textbf{WB} program to a TM.
  - Show how much slowdown we get when we convert a multitape \textbf{WB} program to a single-tape \textbf{WB} program.
  - Show how much slowdown we get when we convert a multitape TM to a multitape \textbf{WB} program.
From **WB** to TMs

0: If reading 0, go to 4.
1: If reading 1, go to 5.
2: Move right.
3: Go to 0.
4: Accept.
5: Reject.

0 → 0, R
1 → 1, R
Γ → Γ, L
Γ → Γ, R
B → B, R
0 → 0, R
Γ → Γ, R
1 → 1, R
Γ → Γ, L
Γ → Γ, L
Γ → Γ, L
Γ → Γ, L
Γ → Γ, L
Γ → Γ, R
Γ → Γ, R
Γ → Γ, R
Γ → Γ, L
Γ → Γ, R
0: If reading B, go to 4.
1: If reading 1, go to 5.
2: Move right.
3: Go to 0.
4: Accept.
5: Reject.
Connecting Models of Computation

• **Theorem**: If there is a WB program for $L$ whose time complexity is $f(n)$, there is a TM whose time complexity is at most $2f(n)$.

• **Proof sketch**: Every line in a WB program gets converted into a set of TM states. Executing each line makes at most two transitions. Thus if the WB program takes time $f(n)$, then TM takes time at most $2f(n)$. 
How efficient is a multitape \textbf{WB} program compared to a single-tape \textbf{WB} program?

\textbf{Recall:} We saw how to implement a multitape \textbf{WB} program with a multistack \textbf{WB} program such that each operation on the multitape \textbf{WB} program required \(O(1)\) stack operations.

We can therefore analyze the efficiency of a multitape \textbf{WB} program by analyzing the efficiency of a multistack \textbf{WB} program.
0: Push 1 onto Stack 3.
0: Push 1 onto Stack 3.

0: Write × on track 5.
0: Push 1 onto Stack 3.

0: Write × on track 5.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.
<table>
<thead>
<tr>
<th>Track 4</th>
<th>Track 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt; 1 1 &lt;</td>
<td></td>
</tr>
<tr>
<td>&gt; 0 1 1 &lt;</td>
<td></td>
</tr>
<tr>
<td>&gt; 1 &lt;</td>
<td></td>
</tr>
</tbody>
</table>

0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.
0: Push 1 onto Stack 3.
0: Write × on track 5.
1: Move left until {>} on track 4.
2: Move right until {<} on track 4.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.
| 1 | 1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | … |
| > | 1 | 1 | < |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| > | 0 | 1 | 1 | < |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| > | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| > | 1 | 1 |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |

0: Push 1 onto Stack 3.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

0: Write × on track 5.
0: Push 1 onto Stack 3.

0: Write \( \times \) on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until {>} on track 4.
0: Push 1 onto Stack 3.

0: Write $\times$ on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write $<$ on track 4

6: Move left until {>} on track 4.
0: Push 1 onto Stack 3.

0: Write × on track 5.

1: Move left until {>} on track 4.

2: Move right until {<} on track 4.

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until {>} on track 4.

7: Move right until {×} on track 5.
0: Push 1 onto Stack 3.

0: Write \times\text{ on track 5.}

1: Move left until \{>\text{ on track 4.}

2: Move right until \{<\text{ on track 4.}

3: Write 1 on track 4.

4: Move right.

5: Write < on track 4

6: Move left until \{>\text{ on track 4.

7: Move right until \{\times\text{ on track 5.}
0: Push 1 onto Stack 3.

0: Write × on track 5.
1: Move left until {>} on track 4.
2: Move right until {<} on track 4.
3: Write 1 on track 4.
4: Move right.
5: Write < on track 4
6: Move left until {>} on track 4.
7: Move right until {×} on track 5.
8: Write B on track 5.
0: Push 1 onto Stack 3.

0: Write × on track 5.
1: Move left until {>} on track 4.
2: Move right until {<} on track 4.
3: Write 1 on track 4.
4: Move right.
5: Write < on track 4
6: Move left until {>} on track 4.
7: Move right until {×} on track 5.
8: Write B on track 5.
Multitape TM Efficiency

- Time to push or pop a stack is determined by
  - how many elements are on that stack and
  - where the tape head is on the tape.
- **Important Fact #1:** After running for $n$ steps, a multistack program can have at most $n$ elements on any stack.
- **Important Fact #2:** After running for $n$ steps, the read head of a TM can be at most $n$ cells to the right of where it started.
Multitape TM Efficiency

- **Lemma**: The time required to simulate the $k$th step of a multitape TM is $O(k)$.

- **Proof sketch**: We need to do at most $O(k)$ work to seek back to the start of the tape, at most $O(k)$ work to seek to the end of the stack, at most $O(1)$ work manipulating the stack, and at most $O(k)$ work moving the tape head back to where it started.

- **Theorem**: If there is a multitape TM for $L$ with time complexity $f(n)$, there is a single-tape TM for $L$ with time complexity $O(f(n)^2)$.

- **Proof Sketch**: At most $O(f(n))$ work is required to simulate any move of the multitape TM, because there are at most $f(n)$ moves made. Doing $O(f(n))$ work $f(n)$ times requires time at most $O(f(n)^2)$. ■
What This Result Means

- We have shown that if it's possible to find an $f(n)$-time MTTM for some language $L$, we can also find an $O(f(n)^2)$-time single-tape TM for $L$.
- It might be possible to do better, though there's no guarantee.
More Impressive Results

• What is the connection between the big-O notation we're used to for real computers and the time complexity of Turing machines?

• **Theorem**: Any algorithm written on a standard computer that runs in time $f(n)$ can be simulated by a single-tape TM in time $O(f(n)^6)$.

• Proof involves building up a simulator for standard computers using TMs; talk to me if you'd like a reference.
Why All This Matters

- **Different models of computation have different efficiencies.**
- TMs, MTTMs, \textbf{WB} programs, and computers can all solve the same problems, but may do so at different speeds.
- In many theoretical results, these differences \textbf{do not matter}.
  - We'll see why in a minute.
Time Complexity Classes
Time Complexity

- Armed with big-O notation, we can start to define different complexity classes.

- The **time complexity class** \( \text{TIME}(f(n)) \) is the set of languages decidable by a single-tape TM with runtime \( O(f(n)) \).

- For example:
  - \( \text{TIME}(n) \) is the set of all languages decidable in time \( O(n) \).
  - \( \text{TIME}(2^n) \) is the set of all languages decidable in time \( O(2^n) \).
TIME($n$)

• All regular languages are in TIME($n$)
  • Build a DFA for a regular language.
  • Convert the DFA into a TM.
  • Accepts in time at most $n + 1$.

• Nontrivial result: A language is regular iff it is in TIME($n$).
  • (This is why we can't build a single-tape TM for BALANCE that runs in $O(n)$ time.)
TIME($n^2$)

- The language of palindromes is in TIME($n^2$)
  - Snake back and forth across the tape checking whether the ends match.
- The language of balanced parentheses is in TIME($n^2$).
  - Use an MTTM to track unmatched open parentheses on a second tape.
- All DCFLs are in TIME($n^2$).
  - Simulate a DCFL with a multitape TM in time $O(n)$.
  - Convert to a single-tape TM in $O(n^2)$.
- Any language in TIME($n$) is also in TIME($n^2$).
  - Since it takes at most $O(n)$ time, it also takes at most $O(n^2)$ time as well.
TIME($n^{18}$)

• All CFLs are in TIME($n^{18}$).
  • Given a grammar $G$, there exists an algorithm on a standard computer that can decide whether $G$ generates $w$ in time $O(n^3)$.
  • Since an $f(n)$-time computer program can be simulated in time $O(f(n)^6)$ on a TM, this means all CFLs are in TIME($n^{18}$).
What is Efficiency?
Growth Rates, Part One

- $O(1)$
- $O(\log n)$
- $O(n)$
Growth Rates, Part Two

- $O(n)$
- $O(n \log n)$
- $O(n^2)$
Growth Rates, Part Three

- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

Graph showing the growth of functions $O(n^2)$, $O(n^3)$, and $O(2^n)$.
To Give You A Better Sense...

- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$
Once More with Logarithms

Graph showing the growth rates of different Big O notations:
- $O(1)$
- $O(\log n)$
- $O(n)$
- $O(n \log n)$
- $O(n^2)$
- $O(n^3)$
- $O(2^n)$

The graph compares the growth rates visually, illustrating how each notation's growth rate changes with increasing input size.
## Comparison of Runtimes

(1 operation = 1 microsecond)

<table>
<thead>
<tr>
<th>Size</th>
<th>1</th>
<th>(\lg n)</th>
<th>n</th>
<th>(n \log n)</th>
<th>(n^2)</th>
<th>(n^3)</th>
<th>(2^n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1(\mu s)</td>
<td>7(\mu s)</td>
<td>100(\mu s)</td>
<td>0.7ms</td>
<td>10ms</td>
<td>&lt;1min</td>
<td>40 quadrillion yrs</td>
</tr>
<tr>
<td>200</td>
<td>1(\mu s)</td>
<td>8(\mu s)</td>
<td>200(\mu s)</td>
<td>1.5ms</td>
<td>40ms</td>
<td>&lt;1min</td>
<td>More than that</td>
</tr>
<tr>
<td>300</td>
<td>1(\mu s)</td>
<td>8(\mu s)</td>
<td>300(\mu s)</td>
<td>2.5ms</td>
<td>90ms</td>
<td>1min</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>1(\mu s)</td>
<td>9(\mu s)</td>
<td>400(\mu s)</td>
<td>3.5ms</td>
<td>160ms</td>
<td>2min</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>1(\mu s)</td>
<td>9(\mu s)</td>
<td>500(\mu s)</td>
<td>4.5ms</td>
<td>250ms</td>
<td>4min</td>
<td></td>
</tr>
<tr>
<td>600</td>
<td>1(\mu s)</td>
<td>9(\mu s)</td>
<td>600(\mu s)</td>
<td>5.5ms</td>
<td>360ms</td>
<td>6min</td>
<td></td>
</tr>
<tr>
<td>700</td>
<td>1(\mu s)</td>
<td>9(\mu s)</td>
<td>700(\mu s)</td>
<td>6.6ms</td>
<td>490ms</td>
<td>9min</td>
<td></td>
</tr>
<tr>
<td>800</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>800(\mu s)</td>
<td>7.7ms</td>
<td>640ms</td>
<td>12min</td>
<td></td>
</tr>
<tr>
<td>900</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>900(\mu s)</td>
<td>8.8ms</td>
<td>810ms</td>
<td>17min</td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>1000(\mu s)</td>
<td>10ms</td>
<td>1000ms</td>
<td>22min</td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>1100(\mu s)</td>
<td>11ms</td>
<td>1200ms</td>
<td>29min</td>
<td></td>
</tr>
<tr>
<td>1200</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>1200(\mu s)</td>
<td>12ms</td>
<td>1400ms</td>
<td>37min</td>
<td></td>
</tr>
<tr>
<td>1300</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>1300(\mu s)</td>
<td>13ms</td>
<td>1700ms</td>
<td>45min</td>
<td></td>
</tr>
<tr>
<td>1400</td>
<td>1(\mu s)</td>
<td>10(\mu s)</td>
<td>1400(\mu s)</td>
<td>15ms</td>
<td>2000ms</td>
<td>56min</td>
<td></td>
</tr>
</tbody>
</table>
Polynomials and Exponentials

- Polynomial functions “scale well.”
  - Small changes to the size of the input do not typically induce enormous changes to the overall runtime.

- Exponential functions scale terribly.
  - Small changes to the size of the input induce huge changes in the overall runtime.
The Cobham-Edmonds Thesis

A language $L$ can be **decided efficiently** iff there is a TM that decides it in polynomial time.

Equivalently, $L$ can be decided in time $O(n^k)$ for some $k \in \mathbb{N}$.

Equivalently, $L \in \text{TIME}(n^k)$ for some $k \in \mathbb{N}$.
The Cobham-Edmonds Thesis

- Efficient runtimes:
  - $4n + 13$
  - $n^3 - 2n^2 + 4n$
  - $n \log \log n$
- "Efficient" runtimes:
  - $n^{1,000,000,000,000}$
  - $10^{500}$

- Inefficient runtimes:
  - $2^n$
  - $n!$
  - $n^n$
- "Inefficient" runtimes:
  - $n^{0.0001 \log n}$
  - $1.000000001^n$
The Complexity Class $P$

- The *complexity class* $P$ contains all problems that can be solved in polynomial time.

- Formally:

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

- Using our definition, a problem can be solved efficiently iff it is in $P$. 
Examples of Problems in $\mathbf{P}$

- All regular languages are in $\mathbf{P}$.
  - Contained in $\text{TIME}(n)$.
- All DCFLs are in $\mathbf{P}$.
  - Contained in $\text{TIME}(n^2)$.
- All CFLs are in $\mathbf{P}$.
  - Contained in $\text{TIME}(n^{18})$
- Many other problems are in $\mathbf{P}$.
  - $POWER2$
  - $SEARCH$
Undecidable Languages

Regular Languages

CFLs

DCFLs

Efficiently Decidable Languages

Undecidable Languages
Undecidable Languages

- Regular Languages
- DCFLs
- CFLs
- P
- R

Undecidable Languages
Problems in P

• **Graph connectivity:**
  Given a graph $G$ and nodes $s$ and $t$, is there a path from $s$ to $t$?

• **Primality testing:**
  Given a number $n$, is $n$ prime? (Best known TM for this takes time $O(n^{72})$.)

• **Maximum matching:**
  Given a set of tasks and workers who can perform those tasks, can all of the tasks be completed in under $n$ hours?
Problems in P

- **Remoteness testing:**
  Given a graph $G$, are all of the nodes in $G$ within distance at most $k$ of one another?

- **Linear programming:**
  Given a linear set of constraints and linear objective function, is the optimal solution at least $n$?

- **Edit distance:**
  Given two strings, can the strings be transformed into one another in at most $n$ single-character edits?
Other Models of Computation

- All models of computation that we've talked about so far (except for the nondeterministic TM) can be reduced to a TM in polynomial time.

- **Theorem**: $L \in \textbf{P}$ iff there is a polynomial-time TM, \textit{WBn} program, multitape TM, or normal computer program for it.

- Essentially – a problem is in \textbf{P} iff you could solve it on a normal computer in polynomial time.
A Feel For Polynomial Time

- What can you do in polynomial time?
- What can you not do in polynomial time?
- Let's see some examples.
Closure under Addition

- **Theorem:** \( O(n^k) + O(n^r) = O(n^{\max\{k, r\}}) \).
  - The sum of two polynomial-bounded functions is itself a polynomial-bounded function.
- If you have two programs that each run in polynomial time, running them in sequence still stays within polynomial time.

```plaintext
function newCode() {
    polynomialFunctionOne();
    polynomialFunctionTwo();
}
```
Closure under Multiplication

- **Theorem**: $O(n^k) \cdot O(n^r) = O(n^{k+r})$.
  - The product of two polynomial-bounded functions is itself a polynomial-bounded function.
- Doing polynomial work polynomially many times stays polynomial.

```c
for (int i = 0; i < poly(); i++) {
    polynomialFunction();
}
```
Closure under Composition

- **Theorem**: If \( f(n) = O(n^k) \) and \( g(n) = O(n^r) \), then \( f(g(n)) = O(n^{kr}) \).

  - The composition of polynomials (applying one polynomial to another) is itself a polynomial.

- Calling one polynomial function on the result of another stays polynomial:

  ```
  function newCode() {
      polynomial2(polynomial1());
  }
  ```
Proving Languages are in $\textbf{P}$

• To prove that a language is regular, we could
  • Design a DFA for it.
  • Design an NFA for it.
  • Design a regular expression for it.
  • Use closure properties.

• To prove that a language is a CFL, we could
  • Design a CFG for it.
  • Design a PDA for it.
  • Use closure properties.

• How do we prove that a language is in $\textbf{P}$?
Proving Languages are in $\mathbf{P}$

- **Directly prove the language is in $\mathbf{P}$**.
  - 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 $\mathbf{P}$.

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

Proving Languages are in \( \textbf{P} \)

- **Directly prove the language is in \( \textbf{P} \).**
  - 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 \( \textbf{P} \).

**Reduce the language to a language in \( \textbf{P} \).**

Show how a polynomial-time decider for some language \( L' \) can be used to decide \( L \).
Proving Languages are in P

Directly prove the language is in 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 P.

Reduce the language to a language in P.
- Show how a polynomial-time decider for some language $L'$ can be used to decide $L$. 
Proving Languages are in $\mathbf{P}$

Directly prove the language is in $\mathbf{P}$.

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 $\mathbf{P}$.

Reduce the language to a language in $\mathbf{P}$.

Show how a polynomial-time decider for some language $L'$ can be used to decide $L$. 

Proving Languages are in $\mathbf{P}$

Directly prove the language is in $\mathbf{P}$.
   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 $\mathbf{P}$.

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

If any instance of $A$ can be converted into an instance of $B$, we say that $A$ reduces to $B$. 

Diagram:
- Problem A
- Problem B
  - Can be converted to
  - Can be used to solve
Mapping Reductions and $\mathsf{P}$

- When studying whether problems were in $\mathsf{R}$, $\mathsf{RE}$, or co-$\mathsf{RE}$, we used mapping reductions.
- We cannot use mapping reductions when talking about the class $\mathsf{P}$.
  - The reduction can do more than polynomial work.
- We will need to introduce a new kind of reduction.
Polynomial-Time Reductions

- Let $A \subseteq \Sigma_1^*$ and $B \subseteq \Sigma_2^*$ be languages.
- A **polynomial-time mapping 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 $f(w) \in B$.
- Informally:
  - A way of turning inputs to $A$ into inputs to $B$ that can be computed in polynomial time that preserves the correct answer.
- Notation: $A \leq_p B$ iff there is a polynomial-time mapping reduction from $A$ to $B$. 
Next Time

- **Polynomial-Time Reductions**
  - What do these reductions look like?
- **NP**
  - What can we *verify* quickly?
- **P \neq NP**
  - How are these classes related?