Foundations of Computer Science
Lecture 7

Recursion

Powerful but Dangerous
Recursion and Induction
Recursive Sets and Structures
With induction, it may be easier to prove a stronger claim.

Leaping induction.
- \( n^3 < 2^n \) for \( n \geq 10 \).
- Postage.

Strong induction.
- Representation theorems: \textbf{FTA}, binary expansion.
- Games: Nim with 2 equal piles.
Today: Recursion

1 Recursive functions
   - Analysis using induction
   - Recurrences
   - Recursive programs

2 Recursive sets
   - Formal Definition of $\mathbb{N}$
   - The Finite Binary Strings $\Sigma^*$

3 Recursive structures
   - Rooted binary trees (RBT)
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.

PROFESSOR
Online lecture tool “Demo”: allows lecturer to see screen of remote student.

PROFESSOR

STUDENT
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
Online lecture tool “Demo”: allows lecturer to see screen of remote student.
A Fantastic Recursion

Online lecture tool “Demo”: allows lecturer to see screen of remote student.

PROFESSOR

HANG!, CRASH!, BANG!, reboot required

*/%&# 😞@$#!
The tool shows the student’s screen, i.e. my previous screen, which is what the tool showed,

The tool *shows* what the tool *showed.*

– self reference
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e. my previous screen, which is what the tool showed,

The tool \textit{shows} what the tool \textit{showed}. – self reference

\textit{look-up} (word): Get definition; if a word \( x \) in the definition is unknown, \( \text{look-up}(x) \).
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e my previous screen, which is what the tool showed,

The tool \textit{shows} what the tool \textit{showed}. \hspace{3cm} - self reference

\textit{look-up} (word): Get definition; if a word \( x \) in the definition is unknown, \textit{look-up} \((x)\).

\[
f(n) = f(n - 1) + 2n - 1.
\]

What is \( f(2) \)?
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e my previous screen, which is what the tool showed,

The tool shows what the tool showed.  

– self reference

look-up(word): Get definition; if a word x in the definition is unknown, look-up(x).

\[ f(n) = f(n - 1) + 2n - 1. \]

What is \( f(2) \)?

\[ f(2) = f(1) + 3 \]
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e my previous screen, which is what the tool showed,

The tool *shows* what the tool *showed*. – *self reference*

*look-up* (word): Get definition; if a word *x* in the definition is unknown, *look-up* (*x*).

\[
f(n) = f(n - 1) + 2n - 1.\]

What is \( f(2) \)?

\[
f(2) = f(1) + 3 = f(0) + 4
\]
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e my previous screen, which is what the tool showed,

The tool shows what the tool showed. – self reference

look-up (word): Get definition; if a word $x$ in the definition is unknown, look-up ($x$).

$$f(n) = f(n - 1) + 2n - 1.$$  What is $f(2)$?

$$f(2) = f(1) + 3 = f(0) + 4 = f(-1) + 3$$
Examples of Recursion: Self Reference

The tool shows the student’s screen, i.e. my previous screen, which is what the tool showed,

The tool *shows* what the tool *showed*.

– *self reference*

*look-up* (word): Get definition; if a word $x$ in the definition is unknown, $\text{look-up}(x)$.

$$f(n) = f(n - 1) + 2n - 1.$$

What is $f(2)$?

$$f(2) = f(1) + 3 = f(0) + 4 = f(-1) + 3 = \cdots$$

*/?%&# 😞@$#!"
Recursion Must Have Base Cases: *Partial* Self Reference.

*look-up* (word) works if there are some known words to which everything reduces.

Similarly with recursive functions,

$$f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases}$$

$$f(2) = f(1) + 3$$
Recursion Must Have Base Cases: *Partial Self Reference.*

*look-up (word)* works if there are some known words to which everything reduces.

Similarly with recursive functions,

\[
f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases}
\]

\[f(2) = f(1) + 3 = f(0) + 4\]
Recursion Must Have Base Cases: \textit{Partial} Self Reference.

\textit{look-up (word)} works if there are some known words to which everything reduces.

Similarly with recursive functions,

\[
f(n) = \begin{cases} 
0 & n \leq 0; \\
f(n - 1) + 2n - 1 & n > 0. 
\end{cases}
\]

\[f(2) = f(1) + 3 = f(0) + 4 = 0 + 4 = 4.\] (ends at a base case)
Recursion Must Have Base Cases: *Partial Self Reference.*

*look-up (word)* works if there are some known words to which everything reduces.

Similarly with recursive functions,

\[
f(n) = \begin{cases} 
0 & n \leq 0; \\
f(n - 1) + 2n - 1 & n > 0.
\end{cases}
\]

\[f(2) = f(1) + 3 = f(0) + 4 = 0 + 4 = 4.\]

(ends at a base case)

Must have **base cases:**

In this case \( f(0) \).
Recursion Must Have Base Cases: *Partial* Self Reference.

*look-up (word)* works if there are some known words to which everything reduces.

Similarly with recursive functions,

\[
  f(n) = \begin{cases} 
    0 & n \leq 0; \\
    f(n - 1) + 2n - 1 & n > 0. 
  \end{cases}
\]

\[
  f(2) = f(1) + 3 = f(0) + 4 = 0 + 4 = 4. 
\]

(ends at a base case)

Must have **base cases:**

In this case \(f(0)\).

Must make **recursive progress:**

To compute \(f(n)\) you must move *closer* to the base case \(f(0)\).
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & \text{if } n \leq 0; \\
 f(n-1) + 2n - 1 & \text{if } n > 0. 
\end{cases} \]

\[ f(0) \rightarrow f(1) \]
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
\ f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

\[
\begin{align*}
\text{f(0)} & \rightarrow f(1) \rightarrow f(2)
\end{align*}
\]
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

\begin{align*}
\boxed{f(0)} \rightarrow f(1) & \rightarrow f(2) & \rightarrow f(3) & \rightarrow f(4) & \rightarrow \cdots
\end{align*}
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n-1) + 2n - 1) & n > 0.
\end{cases} \]

**Induction**

\( P(0) \) is T; \( P(n) \rightarrow P(n+1) \)
(you can conclude \( P(n+1) \) if \( P(n) \) is T)

\[ P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \cdots \]

\( P(n) \) is T for all \( n \geq 0 \).

**Recursion**

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
& \text{Induction} \\
& \text{Recursion} \\
f(n - 1) + 2n - 1 & n > 0.
\end{cases} \]

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]

**Induction**

\( P(0) \) is \( T \); \( P(n) \rightarrow P(n + 1) \)
(\( P(n) \) is \( T \) for all \( n \geq 0 \).

\[ P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \cdots \]

**Recursion**

\( f(0) = 0; f(n + 1) = f(n) + 2n + 1 \)
(\( f(n) \) is known)

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]

We can compute \( f(n) \) for all \( n \geq 0 \).
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
\ f(n - 1) + 2n - 1 & n > 0.
\end{cases} \]

**Induction**

\( P(0) \) is \( T \); \( P(n) \rightarrow P(n + 1) \)

(you can conclude \( P(n + 1) \) if \( P(n) \) is \( T \))

\[ P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \cdots \]

\( P(n) \) is \( T \) for all \( n \geq 0 \).

**Recursion**

\( f(0) = 0; \ f(n + 1) = f(n) + 2n + 1 \)

(we can compute \( f(n + 1) \) if \( f(n) \) is known)

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]

We can compute \( f(n) \) for all \( n \geq 0 \).

**Example: More Base Cases**

\[ f(n) = \begin{cases} 
1 & n = 0; \\
\ f(n - 2) + 2 & n > 0.
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>1</td>
<td>( \times )</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n-1) + 2n - 1 & n > 0. 
\end{cases} \]

**Induction**

\( P(0) \) is \( \top \); \( P(n) \rightarrow P(n + 1) \)

(you can conclude \( P(n + 1) \) if \( P(n) \) is \( \top \))

\[ P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \cdots \]

\( P(n) \) is \( \top \) for all \( n \geq 0 \).

**Recursion**

\( f(0) = 0; f(n + 1) = f(n) + 2n + 1 \)

(we can compute \( f(n + 1) \) if \( f(n) \) is known)

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]

We can compute \( f(n) \) for all \( n \geq 0 \).

**Example: More Base Cases**

\[ f(n) = \begin{cases} 
1 & n = 0; \\
f(n - 2) + 2 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>1</td>
<td>( \times )</td>
<td>3</td>
<td>( \times )</td>
<td>5</td>
<td>( \times )</td>
<td>7</td>
<td>( \times )</td>
<td>9</td>
</tr>
</tbody>
</table>

Creator: Malik Magdon-Ismail
Recursion: 7/16
Analysing Recursion →
Recursion and Induction

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

**Induction**

\[ P(0) \text{ is } T; \ P(n) \rightarrow P(n + 1) \]

(you can conclude \( P(n + 1) \) if \( P(n) \) is \( T \))

\[ P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow P(3) \rightarrow P(4) \rightarrow \cdots \]

\( P(n) \) is \( T \) for all \( n \geq 0 \).

**Recursion**

\[ f(0) = 0; \ f(n + 1) = f(n) + 2n + 1 \]

(we can compute \( f(n + 1) \) if \( f(n) \) is known)

\[ f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow f(4) \rightarrow \cdots \]

We can compute \( f(n) \) for all \( n \geq 0 \).

**Example: More Base Cases**

\[ f(n) = \begin{cases} 
1 & n = 0; \\
 f(n - 2) + 2 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>1</td>
<td>✘</td>
<td>3</td>
<td>✘</td>
<td>5</td>
<td>✘</td>
<td>7</td>
<td>✘</td>
<td>9</td>
</tr>
</tbody>
</table>

How to fix \( f(n) \)? Hint: leaping induction.

**Practice.** Exercise 7.4
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n - 1) + 2n - 1) & n > 0. 
\end{cases} \]

\[
\begin{array}{c|cccccccccc}
  n & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \cdots \\
  f(n) & 0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & \cdots 
\end{array}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & \text{if } n \leq 0; \\
 f(n - 1) + 2n - 1 & \text{if } n > 0.
\end{cases} \]

Unfolding the Recursion

\[ f(n) = f(n - 1) + 2n - 1 \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\ldots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\ldots</td>
</tr>
</tbody>
</table>

Creator: Malik Magdon-Ismail

Recursion: 8 / 16

Checklist for Analyzing Recursion →
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & \text{if } n \leq 0; \\
 f(n - 1) + 2n - 1 & \text{if } n > 0. 
\end{cases} \]

Unfolding the Recursion

\[
\begin{align*}
 f(n) &= f(n - 1) + 2n - 1 \\
f(n - 1) &= f(n - 2) + 2n - 3 
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(n)</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
 f(n) &= f(n - 1) + 2n - 1 \\
 f(n - 1) &= f(n - 2) + 2n - 3 \\
 f(n - 2) &= f(n - 3) + 2n - 5
\end{align*}
\]
Using Induction to Analyze a Recursion

\[
f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n - 1) + 2n - 1) & n > 0. 
\end{cases}
\]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
  f(n) &= f(n - 1) + 2n - 1 \\
  f(n - 1) &= f(n - 2) + 2n - 3 \\
  f(n - 2) &= f(n - 3) + 2n - 5 \\
  & \vdots \\
  f(2) &= f(1) + 3 \\
  f(1) &= f(0) + 1
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
\frac{f(n-1) + 2n - 1}{n > 0}. 
\end{cases} \]

\[
\begin{array}{c|cccccccc}
\text{n} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & \cdots \\
\hline
f(n) & 0 & 1 & 4 & 9 & 16 & 25 & 36 & 49 & 64 & \cdots \\
\end{array}
\]

Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
& \vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n-1) + 2n - 1 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
  f(n) &= f(n-1) + 2n - 1 \\
  f(n-1) &= f(n-2) + 2n - 3 \\
  f(n-2) &= f(n-3) + 2n - 5 \\
  & \vdots \\
  f(2) &= f(1) + 3 \\
  f(1) &= f(0) + 1 \\
  \hline
  f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0.
\end{cases} \]

Unfolding the Recursion

\[
\begin{align*}
 f(n) &= f(n-1) + 2n - 1 \\
 f(n-1) &= f(n-2) + 2n - 3 \\
 f(n-2) &= f(n-3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\end{align*}
\]

\[ + \quad f(n) = 1 + 3 + \cdots + 2n - 1 \]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
<th>15</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>81</td>
<td>100</td>
<td>121</td>
<td>144</td>
<td>169</td>
<td>196</td>
<td>225</td>
<td>256</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
  f(n) &= f(n-1) + 2n - 1 \\
  f(n-1) &= f(n-2) + 2n - 3 \\
  f(n-2) &= f(n-3) + 2n - 5 \\
  &\vdots \\
  f(2) &= f(1) + 3 \\
  f(1) &= f(0) + 1 \\
  f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\( P(n) : f(n) = n^2 \)

[Base case] \( P(0) : f(0) = 0^2 \) (clearly T).
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n-1) + 2n - 1) & n > 0. 
\end{cases} \]

### Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
& \vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\end{align*}
\]

\[ f(n) = 1 + 3 + \cdots + 2n - 1 \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>\cdots</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

### Proof by induction that \( f(n) = n^2 \).

**Base case** \( P(0) : f(0) = 0^2 \) (clearly true).

**Induction** Show \( P(n) \rightarrow P(n+1) \) for \( n \geq 0 \).
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n - 1) + 2n - 1) & n > 0. 
\end{cases} \]

<table>
<thead>
<tr>
<th>n</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>⋯</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(n)</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>⋯</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
  f(n) &= f(n-1) + 2n - 1 \\
  f(n-1) &= f(n-2) + 2n - 3 \\
  f(n-2) &= f(n-3) + 2n - 5 \\
  &\vdots \\
  f(2) &= f(1) + 3 \\
  f(1) &= f(0) + 1 \\
  f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

[Base case] \( P(0) : f(0) = 0^2 \) (clearly true).

[Induction] Show \( P(n) \rightarrow P(n+1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n - 1) + 2n - 1) & n > 0.
\end{cases} \]

Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\hline
f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

[Base case] \( P(0) : f(0) = 0^2 \) (clearly T).

[Induction] Show \( P(n) \rightarrow P(n + 1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[ f(n + 1) \]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n - 1) + 2n - 1) & n > 0.
\end{cases} \]

Unfolding the Recursion

\[
\begin{align*}
    f(n) &= f(n - 1) + 2n - 1 \\
    f(n - 1) &= f(n - 2) + 2n - 3 \\
    f(n - 2) &= f(n - 3) + 2n - 5 \\
    &\vdots \\
    f(2) &= f(1) + 3 \\
    f(1) &= f(0) + 1 \\
    f(0) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

[Base case] \( P(0) : f(0) = 0^2 \) (clearly true).

[Induction] Show \( P(n) \rightarrow P(n + 1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[
\begin{align*}
    f(n + 1) &= f(n) + 2(n + 1) - 1 \\
    &= f(n) + 2n + 1 \quad \text{(recursion)}
\end{align*}
\]
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & \text{if } n \leq 0; \\
(f(n-1) + 2n - 1) & \text{if } n > 0.
\end{cases} \]

---

**Unfolding the Recursion**

\[
\begin{align*}
f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
& \vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\hline
f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

---

**Proof by induction that** \( f(n) = n^2 \).

**Base case** \( P(0) : f(0) = 0^2 \) (clearly T).

**Induction** Show \( P(n) \implies P(n+1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[
\begin{align*}
f(n+1) &= f(n) + 2(n+1) - 1 \\
&= n^2 + 2n + 1
\end{align*}
\] (recursion) \( (f(n) = n^2) \)
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
(f(n-1) + 2n - 1) & n > 0.
\end{cases} \]

| \( n \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | \cdots \\
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>\cdots</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
 f(n) &= f(n-1) + 2n - 1 \\
f(n-1) &= f(n-2) + 2n - 3 \\
f(n-2) &= f(n-3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\Rightarrow f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

[Base case] \( P(0) : f(0) = 0^2 \) (clearly T).

[Induction] Show \( P(n) \rightarrow P(n+1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[
\begin{align*}
 f(n+1) &= f(n) + 2(n+1) - 1 \\
&= n^2 + 2n + 1 \\
&= (n+1)^2
\end{align*}
\] (recursion) \( (f(n) = n^2) \) \( (P(n+1) \text{ is T}) \)
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0.
\end{cases} \]

<table>
<thead>
<tr>
<th>( n )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(n) )</td>
<td>0</td>
<td>1</td>
<td>4</td>
<td>9</td>
<td>16</td>
<td>25</td>
<td>36</td>
<td>49</td>
<td>64</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n - 1) + 2n - 1 \\
f(n - 1) &= f(n - 2) + 2n - 3 \\
f(n - 2) &= f(n - 3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
\end{align*}
\]

\[
\begin{align*}
f(n) &= 1 + 3 + \cdots + 2n - 1 \\
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\( P(n) : f(n) = n^2 \)

[Base case] \( P(0) : f(0) = 0^2 \) (clearly T).

[Induction] Show \( P(n) \rightarrow P(n + 1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[
\begin{align*}
f(n + 1) &= f(n) + 2(n + 1) - 1 \\
&= n^2 + 2n + 1 \\
&= (n + 1)^2 \\
\end{align*}
\]

So, \( P(n + 1) \) is T.
Using Induction to Analyze a Recursion

\[ f(n) = \begin{cases} 
0 & n \leq 0; \\
 f(n - 1) + 2n - 1 & n > 0.
\end{cases} \]

Unfolding the Recursion

\[
\begin{align*}
f(n) &= f(n - 1) + 2n - 1 \\
f(n - 1) &= f(n - 2) + 2n - 3 \\
f(n - 2) &= f(n - 3) + 2n - 5 \\
&\vdots \\
f(2) &= f(1) + 3 \\
f(1) &= f(0) + 1 \\
+ f(n) &= 1 + 3 + \cdots + 2n - 1
\end{align*}
\]

Proof by induction that \( f(n) = n^2 \).

\[ P(n) : f(n) = n^2 \]

[Base case] \( P(0) : f(0) = 0^2 \) (clearly T).

[Induction] Show \( P(n) \rightarrow P(n + 1) \) for \( n \geq 0 \).

Assume \( P(n) : f(n) = n^2 \).

\[
\begin{align*}
f(n + 1) &= f(n) + 2(n + 1) - 1 \\
&= n^2 + 2n + 1 \\
&= (n + 1)^2 \\
&= (P(n + 1) \text{ is } T)
\end{align*}
\]

So, \( P(n + 1) \) is T.

Hard Example: A halving recursion (see text)

\[ f(n) = \begin{cases} 
1 & n = 1; \\
 f(\frac{n}{2}) + 1 & n > 1, \text{ even;} \\
 f(n + 1) & n > 1, \text{ odd;} 
\end{cases} \]

(Looks esoteric? Often, you halve a problem (if it is even) or pad it by one to make it even, and then halve it.)

Prove \( f(n) = 1 + \lceil \log_2 n \rceil \).

Practice. Exercise 7.5
Tinker. Draw the implication arrows. Is the function well defined?
Checklist for Analyzing Recursion

- Tinker. Draw the implication arrows. Is the function well defined?
- Tinker. Compute $f(n)$ for small values of $n$. 
Checklist for Analyzing Recursion

- Tinker. Draw the implication arrows. Is the function well defined?
- Tinker. Compute $f(n)$ for small values of $n$.
- Make a guess for $f(n)$. “Unfolding” the recursion can be helpful here.
Checklist for Analyzing Recursion

- Tinker. Draw the implication arrows. Is the function well defined?
- Tinker. Compute $f(n)$ for small values of $n$.
- Make a guess for $f(n)$. “Unfolding” the recursion can be helpful here.
- Prove your conjecture for $f(n)$ by induction.
Checklist for Analyzing Recursion

- Tinker. Draw the implication arrows. Is the function well defined?
- Tinker. Compute $f(n)$ for small values of $n$.
- Make a guess for $f(n)$. “Unfolding” the recursion can be helpful here.
- Prove your conjecture for $f(n)$ by induction.
  - The type of induction to use will often be related to the type of recursion.
  - In the induction step, use the recursion to relate the claim for $n+1$ to lower values.
Checklist for Analyzing Recursion

- Tinker. Draw the implication arrows. Is the function well defined?
- Tinker. Compute $f(n)$ for small values of $n$.
- Make a guess for $f(n)$. “Unfolding” the recursion can be helpful here.
- Prove your conjecture for $f(n)$ by induction.
  - The type of induction to use will often be related to the type of recursion.
  - In the induction step, use the recursion to relate the claim for $n + 1$ to lower values.

Practice. Exercise 7.6
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]
Growth rate of rabbits, Sanskrit poetry, family trees of bees, ….

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<table>
<thead>
<tr>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>( F_3 )</th>
<th>( F_4 )</th>
<th>( F_5 )</th>
<th>( F_6 )</th>
<th>( F_7 )</th>
<th>( F_8 )</th>
<th>( F_9 )</th>
<th>( F_{10} )</th>
<th>( F_{11} )</th>
<th>( F_{12} )</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<table>
<thead>
<tr>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>( F_3 )</th>
<th>( F_4 )</th>
<th>( F_5 )</th>
<th>( F_6 )</th>
<th>( F_7 )</th>
<th>( F_8 )</th>
<th>( F_9 )</th>
<th>( F_{10} )</th>
<th>( F_{11} )</th>
<th>( F_{12} )</th>
<th>. . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>. . .</td>
</tr>
</tbody>
</table>

Let us prove \( P(n) : F_n \leq 2^n \) by \textbf{strong induction}. 

Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<table>
<thead>
<tr>
<th>( F_1 )</th>
<th>( F_2 )</th>
<th>( F_3 )</th>
<th>( F_4 )</th>
<th>( F_5 )</th>
<th>( F_6 )</th>
<th>( F_7 )</th>
<th>( F_8 )</th>
<th>( F_9 )</th>
<th>( F_{10} )</th>
<th>( F_{11} )</th>
<th>( F_{12} )</th>
<th>( \cdots )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>( \cdots )</td>
</tr>
</tbody>
</table>

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \checkmark \) and \( F_2 = 1 \leq 2^2 \checkmark \)

(why 2 base cases?)
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

\[
\begin{array}{cccccccccccc}
F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} & F_{12} & \ldots \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & \ldots \\
\end{array}
\]

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \checkmark \) and \( F_2 = 1 \leq 2^2 \checkmark \)

(why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n+1) \) for \( n \geq 2. \)
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \ F_2 = 1; \ F_n = F_{n-1} + F_{n-2} \text{ for } n > 2. \]

\[
\begin{array}{cccccccccccc}
F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} & F_{12} & \cdots \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & \cdots \\
\end{array}
\]

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \checkmark \text{ and } F_2 = 1 \leq 2^2 \checkmark \) (why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n + 1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n \).
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_1 )</td>
<td>( F_2 )</td>
<td>( F_3 )</td>
<td>( F_4 )</td>
<td>( F_5 )</td>
<td>( F_6 )</td>
<td>( F_7 )</td>
<td>( F_8 )</td>
<td>( F_9 )</td>
<td>( F_{10} )</td>
<td>( F_{11} )</td>
<td>( F_{12} )</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
</tr>
</tbody>
</table>

Let us prove \( P(n) : F_n \leq 2^n \) by **strong induction**.

**Base Cases:** \( F_1 = 1 \leq 2^1 \✓ \) and \( F_2 = 1 \leq 2^2 \✓ \)

(why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \implies P(n + 1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n \).

\[ F_{n+1} \]
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, ... .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( F_1 )</td>
<td>( F_2 )</td>
<td>( F_3 )</td>
<td>( F_4 )</td>
<td>( F_5 )</td>
<td>( F_6 )</td>
<td>( F_7 )</td>
<td>( F_8 )</td>
<td>( F_9 )</td>
<td>( F_{10} )</td>
<td>( F_{11} )</td>
<td>( F_{12} )</td>
<td>...</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
<td>...</td>
</tr>
</tbody>
</table>

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \) ✓ and \( F_2 = 1 \leq 2^2 \) ✓

(why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n+1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n \).

\[ F_{n+1} = F_n + F_{n-1} \quad \text{(needs } n \geq 2) \]
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

\[
\begin{array}{cccccccccccccc}
F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} & F_{12} & \cdots \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 & \cdots \\
\end{array}
\]

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \) ✔ and \( F_2 = 1 \leq 2^2 \) ✔ (why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n + 1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n \).

\[
F_{n+1} = F_n + F_{n-1} \leq 2^n + 2^{n-1} \quad \text{(needs } n \geq 2) \quad \text{(strong induction hypothesis)}
\]
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \; F_2 = 1; \; F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

<p>| | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>F_1</td>
<td>F_2</td>
<td>F_3</td>
<td>F_4</td>
<td>F_5</td>
<td>F_6</td>
<td>F_7</td>
<td>F_8</td>
<td>F_9</td>
<td>F_{10}</td>
<td>F_{11}</td>
<td>F_{12}</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>8</td>
<td>13</td>
<td>21</td>
<td>34</td>
<td>55</td>
<td>89</td>
<td>144</td>
</tr>
</tbody>
</table>

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \) ✔ and \( F_2 = 1 \leq 2^2 \) ✔

(why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n+1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n \).

\[
F_{n+1} = F_n + F_{n-1} \\
\leq 2^n + 2^{n-1} \quad \text{(needs } n \geq 2) \\
\leq 2 \times 2^n = 2^{n+1} \quad \text{(strong induction hypothesis)}
\]

So, \( F_{n+1} \leq 2^{n+1} \), concluding the proof.
Recurrences: Fibonacci Numbers

Growth rate of rabbits, Sanskrit poetry, family trees of bees, . . . .

\[ F_1 = 1; \quad F_2 = 1; \quad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 2. \]

\[
\begin{array}{cccccccccccc}
F_1 & F_2 & F_3 & F_4 & F_5 & F_6 & F_7 & F_8 & F_9 & F_{10} & F_{11} & F_{12} \\
1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 & 144 \\
\end{array}
\]

Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.

**Base Cases:** \( F_1 = 1 \leq 2^1 \checkmark \) and \( F_2 = 1 \leq 2^2 \checkmark \) (why 2 base cases?)

**Strong Induction:** Prove \( P(1) \land P(2) \land \cdots \land P(n) \rightarrow P(n+1) \) for \( n \geq 2 \).

**Assume:** \( P(1) \land P(2) \land \cdots \land P(n) : F_i \leq 2^i \) for \( 1 \leq i \leq n. \)

\[
F_{n+1} = F_n + F_{n-1} \quad \text{(needs } n \geq 2) \\
\leq 2^n + 2^{n-1} \quad \text{(strong induction hypothesis)} \\
\leq 2 \times 2^n = 2^{n+1}
\]

So, \( F_{n+1} \leq 2^{n+1} \), concluding the proof.

**Practice.** Prove \( F_n \geq (\frac{3}{2})^n \) for \( n \geq 11. \)
Recursive Programs

out=Big(n)
if(n==0) out=1;
else out=2*Big(n-1);

Does this function compute $2^n$?
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

```
out=Big(n)
if(n==0) out=1;
else out=2*Big(n-1);
```

Does this function compute $2^n$?
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**

```plaintext
out=Big(n)
if(n==0) out=1;
else out=2*Big(n-1);
```

Does this function compute $2^n$?
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**

When $n = 0$, $\text{Big}(0) = 1 = 2^0$ ✓

Does this function compute $2^n$?
Recursive Programs

Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**
When $n = 0$, $\text{Big}(0) = 1 = 2^0$ ✓
Assume $\text{Big}(n) = 2^n$ for $n \geq 0$

\[
\text{out=Big(n)}
\]
\[
\text{if(n==0) out=1; else out=2*Big(n-1);}
\]

Does this function compute $2^n$?
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**
When $n = 0$, $\text{Big}(0) = 1 = 2^0 \checkmark$
Assume $\text{Big}(n) = 2^n$ for $n \geq 0$

$$\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}.$$
Recursive Programs

Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**

When $n = 0$, $\text{Big}(0) = 1 = 2^0$ ✓

Assume $\text{Big}(n) = 2^n$ for $n \geq 0$

$$\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}. $$

What is the runtime?

Let $T_n =$ runtime of $\text{Big}$ for input $n$. 

```
out=Big(n)
if(n==0) out=1;
else out=2*Big(n-1);
```

Does this function compute $2^n$?
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**

When $n = 0$, $\text{Big}(0) = 1 = 2^0$ ✓

Assume $\text{Big}(n) = 2^n$ for $n \geq 0$

$$\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}.$$ 

### What is the runtime?

Let $T_n =$ runtime of $\text{Big}$ for input $n$.

$$T_0 = 2$$"
Proving correctness: let’s prove \( \text{Big}(n) = 2^n \) for \( n \geq 1 \)

**Induction.**
When \( n = 0 \), \( \text{Big}(0) = 1 = 2^0 \) √
Assume \( \text{Big}(n) = 2^n \) for \( n \geq 0 \)
\[
\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}.
\]

What is the runtime?
Let \( T_n = \) runtime of \( \text{Big} \) for input \( n \).
\[
T_0 = 2 \\
T_n = T_{n-1} + (\text{check } n==0) + (\text{multiply by 2}) + (\text{assign to } \text{out})
\]

```c
out=Big(n)
if(n==0) out=1;
else out=2*Big(n-1);
Does this function compute \( 2^n \)?
```
Proving correctness: let’s prove $\text{Big}(n) = 2^n$ for $n \geq 1$

**Induction.**
When $n = 0$, $\text{Big}(0) = 1 = 2^0$ ✓
Assume $\text{Big}(n) = 2^n$ for $n \geq 0$

$$\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}.$$ 

What is the runtime?
Let $T_n =$ runtime of $\text{Big}$ for input $n$.

$$T_0 = 2$$
$$T_n = T_{n-1} + (\text{check } n==0) + (\text{multiply by 2}) + (\text{assign to out})$$
$$= T_{n-1} + 3$$

out=$\text{Big}(n)$
if(n==0) out=1;
else out=2*$\text{Big}(n-1)$;

Does this function compute $2^n$?
Proving correctness: let’s prove \( \text{Big}(n) = 2^n \) for \( n \geq 1 \)

**Induction.**

When \( n = 0 \), \( \text{Big}(0) = 1 = 2^0 \) ✓

Assume \( \text{Big}(n) = 2^n \) for \( n \geq 0 \)

\[
\text{Big}(n + 1) = 2 \times \text{Big}(n) = 2 \times 2^n = 2^{n+1}.
\]

What is the runtime?

Let \( T_n = \) runtime of \( \text{Big} \) for input \( n \).

\[
T_0 = 2 \\
T_n = T_{n-1} + \text{(check } n==0) + \text{(multiply by 2)} + \text{(assign to } \text{out}) \\
= T_{n-1} + 3
\]

**Exercise.** Prove by induction that \( T_n = 3n + 2 \).
Recursive definition of the natural numbers $\mathbb{N}$.

1. $1 \in \mathbb{N}$.  
   [basis]

\[ \mathbb{N} = \{ 1, \} \]
Recursive definition of the natural numbers $\mathbb{N}$.

1. $1 \in \mathbb{N}$. \[\text{[basis]}\]
2. $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$. \[\text{[constructor]}\]

$\mathbb{N} = \{1, 2, \}$
Recursive definition of the natural numbers $\mathbb{N}$.

1. $1 \in \mathbb{N}$. [basis]
2. $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$. [constructor]

$$\mathbb{N} = \{1, 2, 3, \ldots\}$$
Recursive definition of the natural numbers $\mathbb{N}$.

1. $1 \in \mathbb{N}$. [basis]
2. $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$. [constructor]

$\mathbb{N} = \{1, 2, 3, 4, \ldots\}$
Recursive definition of the natural numbers $\mathbb{N}$.

1. $1 \in \mathbb{N}$. \hspace{1cm} [basis]
2. $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$. \hspace{1cm} [constructor]
3. Nothing else is in $\mathbb{N}$. \hspace{1cm} [minimality]

$\mathbb{N} = \{1, 2, 3, 4, \ldots\}$

Technically, by bullet 3, we mean that $\mathbb{N}$ is the *smallest* set satisfying bullets 1 and 2.
Recursive Sets: \( \mathbb{N} \)

Recursive definition of the natural numbers \( \mathbb{N} \).

1. \( 1 \in \mathbb{N} \).  \hspace{1cm} \textbf{[basis]}
2. \( x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N} \).  \hspace{1cm} \textbf{[constructor]}
3. Nothing else is in \( \mathbb{N} \).  \hspace{1cm} \textbf{[minimality]}

\[ \mathbb{N} = \{1, 2, 3, 4, \ldots \} \]

Technically, by bullet 3, we mean that \( \mathbb{N} \) is the \textit{smallest} set satisfying bullets 1 and 2.

**Pop Quiz.** Is \( \mathbb{R} \) a set that satisfies bullets 1 and 2 alone? Is it the smallest?
Let $\varepsilon$ be the \textit{empty string} (similar to the empty set).
Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$. 
   
   [basis]
Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$.
2. $x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^*$ AND $x \cdot 1 \in \Sigma^*$.

[basis] [constructor]
Recursive Sets: Finite Binary Strings, $\Sigma^*$

Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$.
2. $x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^*$ AND $x \cdot 1 \in \Sigma^*$.

[basis] [constructor]

Minimality is there by default: nothing else is in $\Sigma^*$. 
Let \( \varepsilon \) be the *empty string* (similar to the empty set).

---

**Recursive definition of \( \Sigma^* \) (finite binary strings).**

1. \( \varepsilon \in \Sigma^* \). [basis]
2. \( x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^* \) AND \( x \cdot 1 \in \Sigma^* \). [constructor]

---

Minimality is there by default: nothing else is in \( \Sigma^* \).
Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$. [basis]
2. $x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^*$ AND $x \cdot 1 \in \Sigma^*$. [constructor]

Minimality is there by default: nothing else is in $\Sigma^*$.

$$\varepsilon \rightarrow 0, 1$$
Let \( \varepsilon \) be the \emph{empty string} (similar to the empty set).

**Recursive definition of \( \Sigma^* \) (finite binary strings).**

1. \( \varepsilon \in \Sigma^* \).  
   **[basis]**

2. \( x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^* \) AND \( x \cdot 1 \in \Sigma^* \).  
   **[constructor]**

Minimality is there by default: nothing else is in \( \Sigma^* \).

\[
\varepsilon \rightarrow 0, 1 \rightarrow 00, 01, 10, 11
\]
Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$.
2. $x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^*$ AND $x \cdot 1 \in \Sigma^*$.  

Minimality is there by default: nothing else is in $\Sigma^*$.

\[\varepsilon \rightarrow 0, 1 \rightarrow 00, 01, 10, 11 \rightarrow 000, 001, 010, 011, 100, 101, 110, 111 \rightarrow \cdots.\]
Recursive Sets: Finite Binary Strings, $\Sigma^*$

Let $\varepsilon$ be the *empty string* (similar to the empty set).

**Recursive definition of $\Sigma^*$ (finite binary strings).**

1. $\varepsilon \in \Sigma^*$. [basis]
2. $x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^*$ AND $x \cdot 1 \in \Sigma^*$. [constructor]

Minimality is there by default: nothing else is in $\Sigma^*$.

\[
\varepsilon \rightarrow 0, 1 \rightarrow 00, 01, 10, 11 \rightarrow 000, 001, 010, 011, 100, 101, 110, 111 \rightarrow \cdots.
\]

\[
\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, 010, 011, 100, 101, 110, 111, \ldots \}
\]

**Practice.** Exercise 7.12
Sir Arthur Cayley discovered trees when modeling chemical hydrocarbons,

\[
\text{methane, } CH_4
\]

\[
\begin{array}{c}
\text{H} \\
\text{H} \cdot \text{C} \cdot \text{H} \\
\text{H} \\
\end{array}
\]
Sir Aurthur Cayley discovered trees when modeling chemical hydrocarbons,

\[
\begin{align*}
\text{methane, } CH_4 & & \text{ethane, } C_2H_6 \\
\text{H} & & \text{H} \\
\text{H-CH} & & \text{H-H} \\
\text{H} & & \text{H} \\
\text{H} & & \text{H} \\
\end{align*}
\]
Sir Arthur Cayley discovered trees when modeling chemical hydrocarbons,
Sir Aurthur Cayley discovered trees when modeling chemical hydrocarbons,

<table>
<thead>
<tr>
<th>Methane, $CH_4$</th>
<th>Ethane, $C_2H_6$</th>
<th>Propane, $C_3H_8$</th>
<th>Butane, $C_4H_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>H  [beam] C  [beam] H</td>
<td>H  [beam] H  [beam] C  [beam] H</td>
<td>H  [beam] H  [beam] H  [beam] C  [beam] H</td>
<td>H  [beam] H  [beam] H  [beam] C  [beam] C  [beam] C  [beam] C  [beam] H</td>
</tr>
<tr>
<td>H  [beam] H  [beam]</td>
<td>H  [beam] H  [beam] C  [beam] C  [beam] H</td>
<td>H  [beam] H  [beam] H  [beam] C  [beam] C  [beam] C  [beam] C  [beam] H</td>
<td>H  [beam] H  [beam] H  [beam] H  [beam] H  [beam] C  [beam] C  [beam] C  [beam] C  [beam] C  [beam] H</td>
</tr>
</tbody>
</table>
Sir Arthur Cayley discovered trees when modeling chemical hydrocarbons, 

- methane, $CH_4$
- ethane, $C_2H_6$
- propane, $C_3H_8$
- butane, $C_4H_{10}$
- iso-butane, $C_4H_{10}$
## Recursive Structures: Trees

Sir Aurthur Cayley discovered trees when modeling chemical hydrocarbons,

<table>
<thead>
<tr>
<th>Chemical</th>
<th>Formula</th>
<th>Structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>Methane</td>
<td>$CH_4$</td>
<td><img src="image" alt="Methane Structure" /></td>
</tr>
<tr>
<td>Ethane</td>
<td>$C_2H_6$</td>
<td><img src="image" alt="Ethane Structure" /></td>
</tr>
<tr>
<td>Propane</td>
<td>$C_3H_8$</td>
<td><img src="image" alt="Propane Structure" /></td>
</tr>
<tr>
<td>Butane</td>
<td>$C_4H_{10}$</td>
<td><img src="image" alt="Butane Structure" /></td>
</tr>
<tr>
<td>Iso-butane</td>
<td>$C_4H_{10}$</td>
<td><img src="image" alt="Iso-butane Structure" /></td>
</tr>
</tbody>
</table>

Trees have many uses in computer science

- Search trees.
- Game trees.
- Decision trees.
- Compression trees.
- Multi-processor trees.
- Parse trees.
- Expression trees.
- Ancestry trees.
- Organizational trees.
- ...
Sir Arthur Cayley discovered trees when modeling chemical hydrocarbons,

- **methane,** $CH_4$
- **ethane,** $C_2H_6$
- **propane,** $C_3H_8$
- **butane,** $C_4H_{10}$
- **iso-butane,** $C_4H_{10}$

Trees have many uses in computer science:

- Search trees.
- Game trees.
- Decision trees.
- Compression trees.
- Multi-processor trees.
- Parse trees.
- Expression trees.
- Ancestry trees.
- Organizational trees.
- ...
Recursive Structures: Trees

Sir Aurthur Cayley discovered trees when modeling chemical hydrocarbons,

- Methane, $\text{CH}_4$
  - $\text{H} \cdot \text{C} \cdot \text{H}$
  - $\text{H} \cdot \text{H}$
  - $\text{H}$

- Ethane, $\text{C}_2\text{H}_6$
  - $\text{H} \cdot \text{C} \cdot \text{H}$
  - $\text{H} \cdot \text{H}$
  - $\text{H}$

- Propane, $\text{C}_3\text{H}_8$
  - $\text{H} \cdot \text{C} \cdot \text{C} \cdot \text{H}$
  - $\text{H} \cdot \text{H}$
  - $\text{H}$

- Butane, $\text{C}_4\text{H}_{10}$
  - $\text{H} \cdot \text{C} \cdot \text{C} \cdot \text{C} \cdot \text{H}$
  - $\text{H} \cdot \text{H}$
  - $\text{H}$

- Iso-butane, $\text{C}_4\text{H}_{10}$
  - $\text{H} \cdot \text{C} \cdot \text{C} \cdot \text{C} \cdot \text{H}$
  - $\text{H} \cdot \text{H}$
  - $\text{H}$

Trees have many uses in computer science

- Search trees.
- Game trees.
- Decision trees.
- Compression trees.
- Multi-processor trees.
- Parse trees.
- Expression trees.
- Ancestry trees.
- Organizational trees.
- ...

Tree.

Not a tree.
Recursive definition of Rooted Binary Trees (RBT).

- The empty tree $\varepsilon$ is an RBT.
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

\[ T_1 \quad T_2 \]
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

![Diagram of recursive definition of RBTs]

- $T_1$ and $T_2$ are disjoint RBTs with roots $r_1$ and $r_2$.
- Linking $r_1$ and $r_2$ to a new root $r$ forms a new RBT.
The empty tree $\varepsilon$ is an RBT.

If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

$\varepsilon$
Recursive definition of Rooted Binary Trees (RBT).
1. The empty tree \( \varepsilon \) is an RBT.
2. If \( T_1, T_2 \) are disjoint RBTs with roots \( r_1 \) and \( r_2 \), then linking \( r_1 \) and \( r_2 \) to a \textit{new} root \( r \) gives a new RBT with root \( r \).

\[
\begin{align*}
\varepsilon & \quad T_1 = \varepsilon \\
T_2 & = \varepsilon
\end{align*}
\]
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

\[
\begin{align*}
\varepsilon & \rightarrow T_1 = \varepsilon \\
T_2 & = \varepsilon \\
\end{align*}
\]
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

\[
\begin{align*}
\varepsilon & \quad T_1 = \varepsilon \\
T_2 & \quad T_2 = \varepsilon
\end{align*}
\]
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

$\begin{align*}
\varepsilon & \quad T_1 = \varepsilon \\
T_2 = \varepsilon \\
\end{align*}$

$\begin{align*}
T_1 = & \quad T_2 = \varepsilon \\
T_1 = & \quad T_2 = \\
\end{align*}$
Recursive definition of Rooted Binary Trees (RBT).

1. The empty tree $\varepsilon$ is an RBT.
2. If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

\[ 
\begin{align*}
\varepsilon \quad T_1 & = \varepsilon \\
T_2 & = \varepsilon
\end{align*}
\]
Recursive definition of Rooted Binary Trees (RBT).

- The empty tree \( \varepsilon \) is an RBT.
- If \( T_1, T_2 \) are disjoint RBTs with roots \( r_1 \) and \( r_2 \), then linking \( r_1 \) and \( r_2 \) to a new root \( r \) gives a new RBT with root \( r \).
Recursive definition of Rooted Binary Trees (RBT).

- The empty tree $\varepsilon$ is an RBT.
- If $T_1, T_2$ are disjoint RBTs with roots $r_1$ and $r_2$, then linking $r_1$ and $r_2$ to a new root $r$ gives a new RBT with root $r$. 

\[
\begin{align*}
\varepsilon &= T_1 = \varepsilon \\
T_2 &= T_2 = \varepsilon \\
T_1 &= T_1 = . \\
T_2 &= T_2 = . \\
T_1 &= T_1 = . \\
T_2 &= T_2 = . \\
\end{align*}
\]
Trees Are Important: Food for Thought

- Tree.
- Not a tree.
- Do we *know* the right structure is not a tree?
Trees Are Important: Food for Thought

- Tree.
- Not a tree.

Do we *know* the right structure is not a tree?

Are we *sure* it can’t be derived?
Trees Are Important: Food for Thought

- Tree.
- Not a tree.

Do we know the right structure is not a tree?

Are we sure it can’t be derived?

Is there only one way to derive a tree?
Trees Are Important: Food for Thought

- Tree.
- Not a tree.
- Do we *know* the right structure is not a tree?
- Are we *sure* it can’t be derived?

Is there only one way to derive a tree?

- Trees are more general than just RBT and have many interesting properties.
  - A tree is a connected graph with $n$ nodes and $n - 1$ edges.
  - A tree is a connected graph with no cycles.
  - A tree is a graph in which any two nodes are connected by exactly one path.
Trees Are Important: Food for Thought

- Tree. Not a tree.

Do we know the right structure is not a tree?
Are we sure it can’t be derived?

- Is there only one way to derive a tree?

- Trees are more general than just RBT and have many interesting properties.
  - A tree is a connected graph with \( n \) nodes and \( n - 1 \) edges.
  - A tree is a connected graph with no cycles.
  - A tree is a graph in which any two nodes are connected by exactly one path.

Can we be sure every RBT has these properties?