Recursion
  – Chapter 7
Overview

- Recursive functions
  - Analysis using induction
  - Recurrences
  - Recursive programs
- Recursive sets
  - Formal Definition of $\mathbb{N}$
  - The Finite Binary Strings $\Sigma^*$
- Recursive structures
  - Rooted binary trees (RBT)
Fantastic Recursion

- Suppose you’re talking to a friend on Zoom
  - Your friend’s laptop is also projecting on their TV
    - The TV is behind your friend’s back, so you can see it through their camera stream
    - What do you see on the TV?
    - Your friend’s Zoom, which contains your camera stream and your friend’s camera stream
  - What do you see on the TV on your friend’s camera stream?
    » Your friend’s Zoom, which contains your camera stream and your friend’s camera stream
    • What do you see on the TV on your friend’s camera stream?
      • Your friend’s Zoom, which contains your camera stream and your friend’s camera stream
        • What do you see on the TV on your friend’s camera stream?
          • Your friend’s Zoom, which contains your camera stream and your friend’s camera stream
Examples of Recursion: Self Reference

• The TV shows your friend’s Zoom, which has your friend’s camera stream, which has your friend’s TV
  – The TV shows what the TV showed. – self reference

• look-up(word): Get definition; if a word $x$ in the definition is unknown, $\text{look-up}(x)$
  – Get definition; if a word $y$ in the definition is unknown, $\text{look-up}(y)$
  • Eventually you’ll end up in a cycle
    – An unknown word appears in the definition of another word, which appears in the definition of the first, etc.

• $f(n) = f(n - 1) + 2n - 1$
  – What is $f(2)$?
    $$
    f(2) = f(1) + 3 = \\
    = f(0) + 4 = \\
    = f(-1) + 3 = \cdots
    $$
  – WHEN DOES THIS END???
Recursion Must Have Base Cases: Partial Self Reference

- *look-up*(word) works if there are some known words to which everything reduces
  - This way you won’t recurse forever
- Similarly with recursive functions
  \[
  f(n) = \begin{cases} 
  0 & n \leq 0 \\
  f(n - 1) + 2n - 1 & n > 0
  \end{cases}
  \]
  \[
  f(2) = f(1) + 3 \\
  = f(0) + 4 = 4
  \]
- 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

- **Recursion**
  - Start with the base case:
    \[ f(0) = 0 \]
  - Then compute the recursive step: \( f(n + 1) = f(n) + 2n - 1 \)
    - We can compute \( f(n + 1) \) if \( f(n) \) is known
    - \( f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow \cdots \)
    - We can compute \( f(n) \) for all \( n \geq 0 \)

- **Induction**
  - Start with \( P(0) \). Show \( P(0) \) is T.
  - Then show \( P(n) \rightarrow P(n + 1) \)
    - You can conclude \( P(n + 1) \) is T if \( P(n) \) is T
    - \( P(0) \rightarrow P(1) \rightarrow P(2) \rightarrow \cdots \)
    - \( P(n) \) is T \( \forall n \geq 0 \)

- **Recursion**
  - Start with the base case:
    \[ f(0) = 0 \]
  - Then compute the recursive step: \( f(n + 1) = f(n) + 2n - 1 \)
    - We can compute \( f(n + 1) \) if \( f(n) \) is known
    - \( f(0) \rightarrow f(1) \rightarrow f(2) \rightarrow f(3) \rightarrow \cdots \)
    - We can compute \( f(n) \) for all \( n \geq 0 \)
Recursion and Induction, cont’d

• Example: more base cases

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

• Let’s look at some values of \( f \)

\[
\begin{align*}
  f(0) &= 1 \\
  f(1) &= ? \\
  f(2) &= 3 \\
  f(3) &= ? \\
  f(4) &= 5
\end{align*}
\]

• How do we fix \( f \)?
  – Hint: leaping induction!

• Practice: Exercise 7.4
Using Induction to Analyze a Recursion

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

- What is \( f(1), f(2), f(3), f(4), \ldots \)?
  
  \[
  \begin{align*}
  f(1) & = 1 \\
  f(2) & = 4 \\
  f(3) & = 9 \\
  f(4) & = 16 
  \end{align*}
  \]

  - Hm, could this actually be \( f(n) = n^2 \)???
  - Let’s unfold 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, cont’d

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

- Let’s unfold the recursion:
  \[ f(n) = f(n-1) + 2n - 1 \]
  \[ f(n-1) = f(n-2) + 2n - 3 \]
  \[ f(n-2) = f(n-3) + 2n - 5 \]
  \[ \ldots \]
  \[ f(2) = f(1) + 3 \]
  \[ f(1) = f(0) + 1 \]

- Let’s add them up: \((f(n - 1)’s\ cancel, f(n - 2)’s\ cancel, \ etc.)\)
  \[ f(n) = f(0) + 1 + 3 + \ldots + 2n - 1 \]

- Can use Gauss’s idea here also to derive \(f(n) = n^2:\)
  \[ 2f(n) = 2n \cdot n \]
Using Induction to Analyze a Recursion, cont’d

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

- Proof that \( f(n) = n^2 \). [By induction]

1. [Base case] \( P(0) = 0 \). Clearly T.

2. [Induction step] Show \( P(n) \rightarrow P(n+1) \).
   
   - Assume \( P(n): f(n) = n^2 \).
   
   \[
   f(n+1) = f(n) + 2(n+1) - 1 \quad \text{[recursion]}
   \]
   
   \[
   = n^2 + 2n + 1 \quad \text{[induction hypothesis]}
   \]
   
   \[
   = (n + 1)^2 \quad \text{[\( P(n+1) \) is T]}
   \]

3. By induction, \( P(n+1) \) is T.
Using Induction to Analyze a Recursion, cont’d

• Hard example:

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

• A halving recursion!
  – Discussed in the book
  – (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 \(f(n) = 1 + \lceil \log_2 n \rceil\)
  – The notation \([x]\) means the smallest integer greater than or equal to \(x\)

• Practice. Exercise 7.5
• 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

- Fibonacci sequences appear frequently in nature
  - Growth rate of rabbits, family trees of bees, Sanskrit poetry
- Defined formally as:
  \[
  F_1 = 1 \\
  F_2 = 1 \\
  F_n = F_{n-1} + F_{n-2} \text{ for } n > 2
  \]
- Let us prove \( P(n) : F_n \leq 2^n \) by strong induction.
- What do we do first?
  - TINKER!

\[
F_3 = 2 \\
F_4 = 3 \\
F_5 = 5 \\
F_6 = 8 \\
F_7 = 13
\]

Source: https://mathcenter.oxford.emory.edu/site/math125/fibonacciRabbits/
Recurrences: Fibonacci Numbers, cont’d

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

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

1. **[Base cases]**
   
   \[
   F_1 = 1 \leq 2 \\
   F_2 = 1 \leq 2^2
   \]

   - Clearly T.

   - Why two base cases?

2. **[Induction step]** 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 \( 1 \leq i \leq n \)

   \[
   F_{n+1} = F_n + F_{n-1} \quad \quad \quad \quad \quad \quad \quad [\text{definition for } n \geq 2] \\
   \leq 2^n + 2^{n-1} \quad \quad \quad \quad \quad \quad \quad \quad [\text{strong induction hypothesis}] \\
   \leq 2 \cdot 2^n = 2^{n+1}
   \]

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

- **Practice**: Prove \( F_n \geq \left(\frac{3}{2}\right)^n \) for \( n \geq 11 \)
Recursive Programs

• Look at the following program

```python
def Big(n):
    if(n==0): out=1
    else: out=2*Big(n-1)
```

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

• **Induction.**
  – When $n = 0$, $\text{Big}(n) = 1 = 2^0$. Check.
  – Assume $\text{Big}(n) = 2^n$ for $n \geq 0$.
    
    $\text{Big}(n + 1) = 2 \times \text{Big}(n)$
    
    $= 2 \times 2^n = 2^{n+1}$

• Proving code correctness has 2 parts (why?)
  – Prove algorithm is correct AND implementation is correct
Recursive Programs, cont’d

• Look at the following program

```python
def Big(n):
    if(n==0): out=1
    else: out=2*Big(n-1)
```

• What is the runtime?
• Define $T_n =$ runtime of `Big` for input $n$

\[
T_0 = 2 \quad \text{[2 operations]}
\]

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

• Exercise. Prove by induction that $T_n = 3n + 2$
Recursive Sets: \( \mathbb{N} \)

- Recursive definition of the natural numbers \( \mathbb{N} \)
  
  \[
  \begin{align*}
  1 & \in \mathbb{N} \quad \text{[basis]} \\
  x \in \mathbb{N} \to x + 1 & \in \mathbb{N} \quad \text{[constructor]}
  \end{align*}
  \]

  Nothing else is in \( \mathbb{N} \) \quad \text{[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.

- Minimality is essential in order to define our set without ambiguity
• Let \( \varepsilon \) be empty string (similar to the empty set)

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

\[
\varepsilon \in \Sigma^* \quad \text{[basis]}
\]

\[
x \in \Sigma^* \rightarrow x \cdot 0 \in \Sigma^* \text{ AND } x \cdot 1 \in \Sigma^* \quad \text{[constructor]}
\]
– where \( \cdot \) means concatenation

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

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

• And so finally

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

• **Practice.** Exercise 7.12
Recursive Structures: Trees

- Arthur Cayley discovered trees when modeling hydrocarbons

\[
\begin{align*}
\text{methane, } & CH_4 \\
\text{H - C - H} & \\
\text{H - H} & \\
\text{H - C - H} & \\
\text{H - C - C - H} & \\
\text{H - C - C - C - H} & \\
\text{H - C - C - C - C - H} & \\
\text{H - C - C - C - H} & \\
\end{align*}
\]

- 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

Example Tree

Not a Tree
Rooted Binary Trees (RBT)

- 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 $r$
Trees Are Important: Food for Thought

- 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 where any two nodes are connected by exactly one path

- Can we be sure every RBT has these properties?