Recursion

Reading

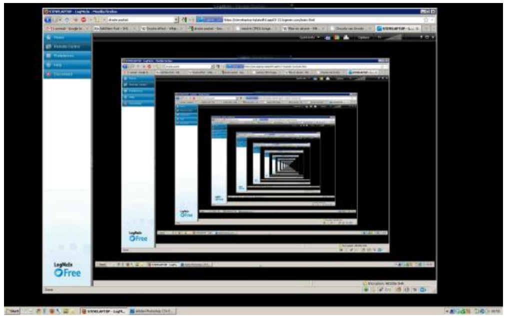
- Malik Magdon-Ismail. Discrete Mathematics and Computing.
  - Chapter 7

Overview

- Recursive functions
  - Analysis using induction
  - Recurrences
  - Recursive programs
- Recursive sets
  - Formal Definition of $\mathbb{N}$
  - The Finite Binary Strings Σ^*
- 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
              - What do you see on the TV on 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, `look-up(x)`
  - Get definition; if a word y in the definition is unknown, `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 $f(2)$?

$$\begin{aligned} f(2) &= f(1) + 3 = \\ &= f(0) + 4 = \\ &= f(-1) + 3 \ = \cdots \end{aligned}$$

  - 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

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

$$\begin{aligned} f(2) &= f(1) + 3 \\ &= f(0) + 4 = 4 \end{aligned}$$

- 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

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

$$f(0) \to f(1) \to f(2) \to \cdots$$

- **Induction**
  - Start with $P(0)$. Show $P(0)$ is T.
  - Then show $P(n) \to P(n+1)$
    - You can conclude $P(n+1)$ is T if $P(n)$ is T
    - $P(0) \to P(1) \to P(2) \to \cdots$
    - $P(n)$ is T $\forall n \ge 0$
- **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
    - $f(0) \to f(1) \to f(2) \to f(3) \to \cdots$
    - We can compute $f(n)$ for all $n \ge 0$

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

$$f(0) = 1$$
$$f(1) = ?$$
$$f(2) = 3$$
$$f(3) = ?$$
$$f(4) = 5$$

- How do we fix f?
  - Hint: leaping induction!
- Practice: Exercise 7.4

Using Induction to Analyze a Recursion

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

- What is $f(1), f(2), f(3), f(4), \ldots$?

$$f(1) = 1$$
$$f(2) = 4$$
$$f(3) = 9$$
$$f(4) = 16$$

  – Hm, could this actually be $f(n) = n^2$???

  – Let's unfold the recursion:

$$\begin{aligned} f(n) &= f(n-1) + 2n - 1 \\ f(n-1) &= f(n-2) + 2n - 3 \\ f(n-2) &= f(n-3) + 2n - 5 \\ &\cdots \\ f(2) &= f(1) + 3 \\ f(1) &= f(0) + 1 \end{aligned}$$

Using Induction to Analyze a Recursion, cont'd

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

- Let's unfold the recursion:

$$\begin{aligned} f(n) &= f(n-1) + 2n - 1 \\ f(n-1) &= f(n-2) + 2n - 3 \\ f(n-2) &= f(n-3) + 2n - 5 \\ &\cdots \\ f(2) &= f(1) + 3 \\ f(1) &= f(0) + 1 \end{aligned}$$

- Let's add them up: ($f(n-1)$'s cancel, $f(n-2)$'s cancel, etc.)

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

- 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 $P(n) \to P(n+1)$.
   – Assume $P(n)$: $f(n) = n^2$.

$$\begin{aligned} f(n+1) &= f(n) + 2(n+1) - 1 && \textbf{[recursion]} \\ &= n^2 + 2n + 1 && \textbf{[induction hypothesis]} \\ &= (n+1)^2 && [P(n+1) \text{ is T}] \end{aligned}$$

3. By induction, $P(n+1)$ is T.

Using Induction to Analyze a Recursion, cont'd

- Hard example:

$$f(n) = \begin{cases} 1 & n = 1 \\ f\left(\frac{n}{2}\right) + 1 & n > 1, even \\ f(n+1) & n > 1, odd \end{cases}$$

- 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 $\lceil x \rceil$ means the smallest integer greater than or equal to x
- **Practice.** Exercise 7.5

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

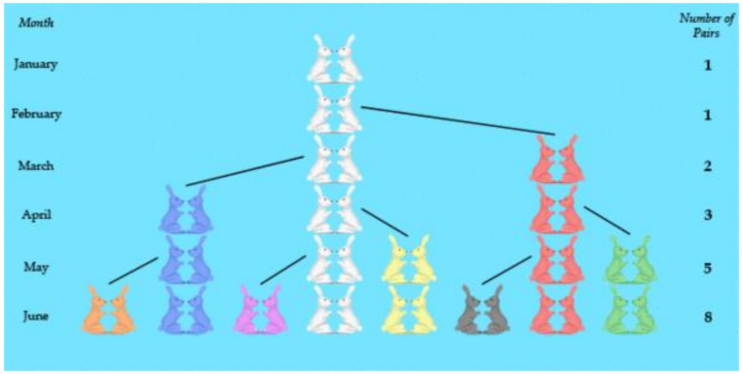
Rensselaer

- 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?

1. [**Induction step**] Prove $P(1) \wedge P(2) \wedge \cdots \wedge P(n) \rightarrow P(n+1)$ for $n \geq 2$.
   - Assume: $P(1) \wedge P(2) \wedge \cdots \wedge P(n)$: $F_i \leq 2^i$ for $1 \leq i \leq n$

$$\begin{aligned} F_{n+1} &= F_n + F_{n-1} && [\textbf{definition for } \boldsymbol{n \geq 2}] \\ &\leq 2^n + 2^{n-1} && [\textbf{strong induction hypothesis}] \\ &\leq 2 \cdot 2^n = 2^{n+1} \end{aligned}$$

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

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

- Proving correctness: let's prove $\texttt{Big}(n) = 2^n$ for $n \geq 1$
- **Induction**.
  - When $n = 0$, $\texttt{Big}(n) = 1 = 2^0$. Check.
  - Assume $\texttt{Big}(n) = 2^n$ for $n \geq 0$.

$$\begin{aligned}\texttt{Big}(n+1) &= 2 \times \texttt{Big}(n) \\ &= 2 \times 2^n = 2^{n+1}\end{aligned}$$

- Proving code correctness has 2 parts (why?)
  - Prove algorithm is correct AND implementation is correct

Recursive Programs, cont'd

- Look at the following program

```
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

$$T_0 = 2 \qquad \textbf{[2 operations]}$$
$$\begin{aligned} T_n &= T_{n-1} + (check\ n == 0) + (multiply\ by\ 2) + (assign\ to\ out) \\ &= T_{n-1} + 3 \end{aligned}$$

- **Exercise.** Prove by induction that $T_n = 3n + 2$

Recursive Sets: $\mathbb{N}$

- Recursive definition of the natural numbers $\mathbb{N}$

$1 \in \mathbb{N}$	[**basis**]
$x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$	[**constructor**]
Nothing else is in $\mathbb{N}$	[**minimality**]

- $\mathbb{N} = \{1,2,3,4, \dots\}$
- Technically, by bullet 3, we mean that $\mathbb{N}$ is the *smallest* set satisfying bullets 1 and 2.
- Minimality is essential in order to define our set without ambiguity

Recursive Sets: Finite Binary Strings, Σ^*

- Let ε be empty string (similar to the empty set)
- Recursive definition of Σ^* (finite binary strings):

$$\varepsilon \in \Sigma^* \qquad\qquad [\textbf{basis}]$$

$$x \in \Sigma^* \rightarrow x \bullet 0 \in \Sigma^* \text{ AND } x \bullet 1 \in \Sigma^* \qquad\qquad [\textbf{constructor}]$$

  - where • means concatenation
- Minimality is there by default: nothing else is in Σ^*

$$\varepsilon \rightarrow 0{,}1 \rightarrow 00{,}01{,}10{,}11 \rightarrow 000{,}001{,}010{,}011{,}100{,}101{,}110{,}111 \rightarrow \cdots$$

- And so finally

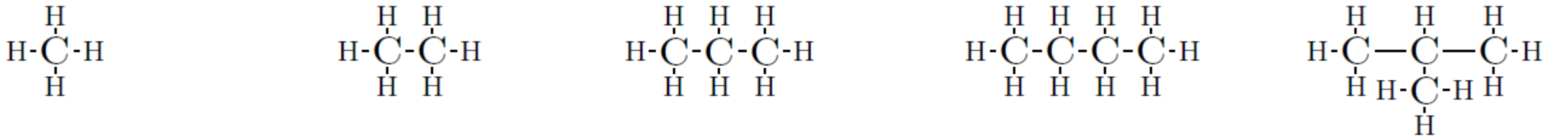
$$\Sigma^* = \{0{,}1{,}00{,}01{,}10{,}11{,}000{,}001{,}010{,}011{,}100{,}101{,}110{,}111, \ldots\}$$

- **Practice.** Exercise 7.12

Recursive Structures: Trees

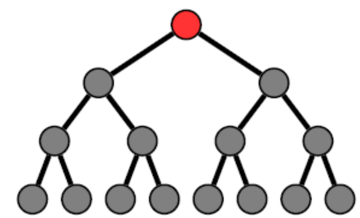
- Arthur Cayley discovered trees when modeling hydrocarbons

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

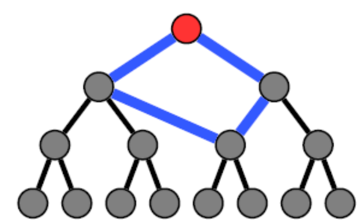


- Trees have many uses in computer science
  - Search trees
  - Game trees
  - Decision trees
  - Compression trees
  - Multi-processor trees
  - Parse trees
  - Expression trees
  - Ancestry trees
  - Organizational trees

Example Tree

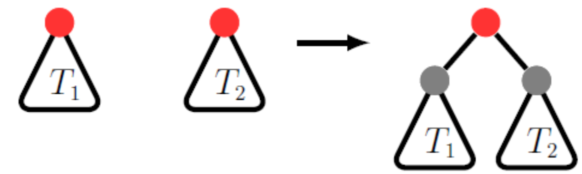


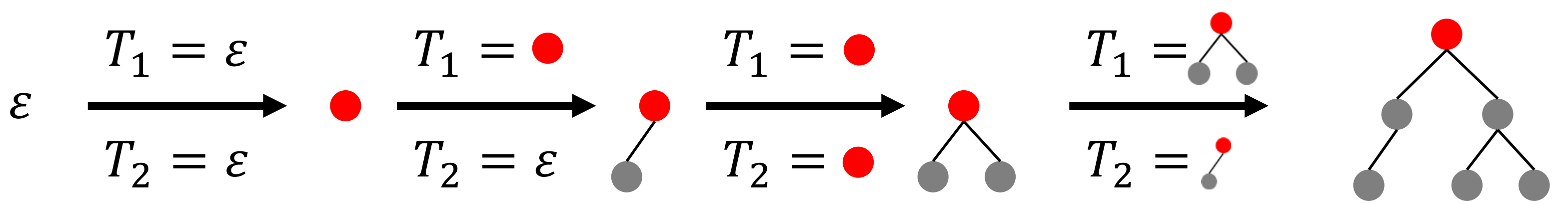
Not a Tree

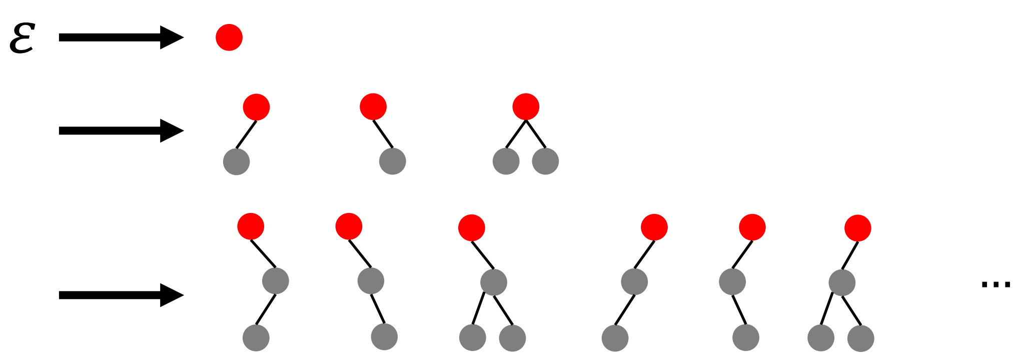


Rooted Binary Trees (RBT)

- **Recursive definition of Rooted Binary Trees (RBT).**
  - The empty tree ε 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



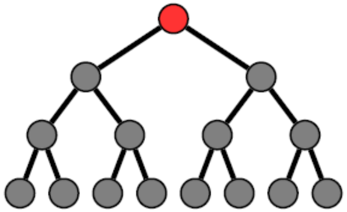




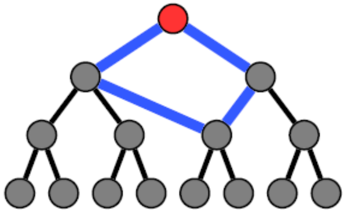
Trees Are Important: Food for Thought

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

Example Tree



Not a Tree



- 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?