# Module #14: **Recursion**

Rosen 5th ed., §§3.4-3.5
~18 slides, ~1 lecture


(c)2001-2003, Michael P. Frank

# §3.4: Recursive Definitions

- In induction, we *prove* all members of an infinite set have some property $P$ by proving the truth for larger members in terms of that of smaller members.
- In *recursive definitions*, we similarly *define* a function, a predicate or a set over an infinite number of elements by defining the function or predicate value or set-membership of larger elements in terms of that of smaller ones.


(c)2001-2003, Michael P. Frank

# Recursion

- *Recursion* is a general term for the practice of defining an object in terms of *itself* (or of part of itself).
- An inductive proof establishes the truth of $P(n+1)$ *recursively* in terms of $P(n)$.
- There are also recursive *algorithms*, *definitions*, *functions*, *sequences*, and *sets*.

# Recursively Defined Functions

- Simplest case: One way to define a function $f$:**N**→$S$ (for any set $S$) or series $a_n=f(n)$ is to:
  - Define $f(0)$.
  - For $n>0$, define $f(n)$ in terms of $f(0),\ldots,f(n-1)$.
- *E.g.*: Define the series $a_n :\equiv 2^n$ recursively:
  - Let $a_0 :\equiv 1$.
  - For $n>0$, let $a_n :\equiv 2a_{n-1}$.

# Another Example

- Suppose we define $f(n)$ for all $n \in \mathbf{N}$ recursively by:
  - Let $f(0)=3$
  - For all $n \in \mathbf{N}$, let $f(n+1)=2f(n)+3$
- What are the values of the following?
  - $f(1)=9$ $f(2)=21$ $f(3)=45$ $f(4)=93$

# Recursive definition of Factorial

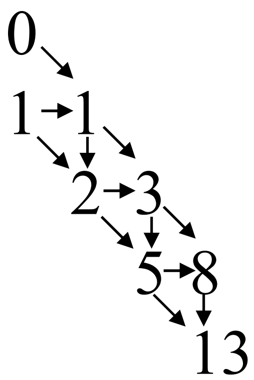
- Give an inductive definition of the factorial function $F(n) :\equiv n! :\equiv 2\cdot3\cdot\ldots\cdot n$.
  - Base case: $F(0) :\equiv 1$
  - Recursive part: $F(n) :\equiv n \cdot F(n\text{-}1)$.
    - $F(1)=1$
    - $F(2)=2$
    - $F(3)=6$


(c)2001-2003, Michael P. Frank

# The Fibonacci Series

- The *Fibonacci series* $f_{n\geq 0}$ is a famous series defined by:

$$f_0 :\equiv 0, \quad f_1 :\equiv 1, \quad f_{n\geq 2} :\equiv f_{n-1} + f_{n-2}$$




Leonardo Fibonacci
1170-1250


(c)2001-2003, Michael P. Frank

# Inductive Proof about Fib. series

- **Theorem:** $f_n < 2^n$. ⟵ Implicitly for all $n \in \mathbf{N}$
- **Proof:** By induction.

Base cases: $f_0 = 0 < 2^0 = 1$, $f_1 = 1 < 2^1 = 2$ — Note use of base cases of recursive def'n.

Inductive step: Use 2nd principle of induction (strong induction). Assume $\forall k<n,\ f_k < 2^k$. Then $f_n = f_{n-1} + f_{n-2}$ is

$$< 2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2^n. \quad \blacksquare$$

# Recursively Defined Sets

- An infinite set $S$ may be defined recursively, by giving:
  - A small finite set of *base* elements of $S$.
  - A rule for constructing new elements of $S$ from previously-established elements.
  - Implicitly, $S$ has no other elements but these.
- **Example:** Let $3 \in S$, and let $x+y \in S$ if $x,y \in S$. What is $S$?

# The Set of All Strings

- Given an alphabet $\Sigma$, the set $\Sigma^*$ of all strings over $\Sigma$ can be recursively defined as:

  $\varepsilon \in \Sigma^*$ ($\varepsilon :\equiv$ “”, the empty string)    Book uses $\lambda$

  $w \in \Sigma^* \wedge x \in \Sigma \rightarrow wx \in \Sigma^*$

- Exercise: Prove that this definition is equivalent to our old one:

$$\Sigma^* :\equiv \bigcup_{n \in \mathbf{N}} \Sigma^n$$


(c)2001-2003, Michael P. Frank

# Recursive Algorithms (§3.5)

- Recursive definitions can be used to describe *algorithms* as well as functions and sets.
- Example: A procedure to compute $a^n$.

**procedure** *power*($a \neq 0$: real, $n \in \mathbf{N}$)

**if** $n = 0$ **then return** 1

**else return** $a \cdot power(a, n-1)$



(c)2001-2003, Michael P. Frank

# Efficiency of Recursive Algorithms

- The time complexity of a recursive algorithm may depend critically on the number of recursive calls it makes.
- Example: *Modular exponentiation* to a power $n$ can take $\log(n)$ time if done right, but linear time if done slightly differently.
  - Task: Compute $b^n$ **mod** $m$, where $m \geq 2$, $n \geq 0$, and $1 \leq b < m$.


(c)2001-2003, Michael P. Frank

# Modular Exponentiation Alg. #1

Uses the fact that $b^n = b \cdot b^{n-1}$ and that $x \cdot y$ **mod** $m = x \cdot (y$ **mod** $m)$ **mod** $m$. (Prove the latter theorem at home.)

**procedure** *mpower*($b \geq 1, n \geq 0, m > b \in \mathbf{N}$)

{Returns $b^n$ **mod** $m$.}
**if** $n=0$ **then return** 1 **else**
**return** ($b \cdot$*mpower*($b, n-1, m$)) **mod** $m$

Note this algorithm takes $\Theta(n)$ steps!

# Modular Exponentiation Alg. #2

- Uses the fact that $b^{2k} = b^{k\cdot 2} = (b^k)^2$.

**procedure** *mpower*($b,n,m$) {same signature}

**if** $n=0$ **then return** 1

**else if** $2|n$ **then**

**return** *mpower*$(b,n/2,m)^2$ **mod** $m$

**else return** (*mpower*$(b,n-1,m)\cdot b$) **mod** $m$

What is its time complexity? $\Theta(\log n)$ steps

# A Slight Variation

Nearly identical but takes $\Theta(n)$ time instead!

**procedure** *mpower*$(b,n,m)$ {same signature}

**if** $n=0$ **then return** 1

**else if** $2|n$ **then**

**return** $(mpower(b,n/2,m)\cdot$

$mpower(b,n/2,m))$ **mod** $m$

**else return** $(mpower(b,n-1,m)\cdot b)$ **mod** $m$

The number of recursive calls made is critical.

# Recursive Euclid's Algorithm

**procedure** $gcd(a,b\in\mathbf{N})$
  **if** $a = 0$ **then return** $b$
  **else return** $gcd(b$ **mod** $a, a)$

- Note recursive algorithms are often simpler to code than iterative ones…
- However, they can consume more stack space, if your compiler is not smart enough.


(c)2001-2003, Michael P. Frank

# Merge Sort

**procedure** *sort*$(L = \ell_1,\ldots, \ell_n)$
  **if** $n>1$ **then**
    $m := \lfloor n/2 \rfloor$  {this is rough ½-way point}
    $L := merge(sort(\ell_1,\ldots, \ell_m),$
        $sort(\ell_{m+1},\ldots, \ell_n))$
  **return** $L$

- The merge takes $\Theta(n)$ steps, and merge-sort takes $\Theta(n \log n)$.

# Merge Routine

**procedure** *merge*($A$, $B$: sorted lists)
  $L$ = empty list
  $i$:=0, $j$:=0, $k$:=0
  **while** $i<|A| \wedge j<|B|$  {$|A|$ is length of $A$}
    **if** $i=|A|$ **then** $L_k := B_j$; $j := j + 1$
    **else if** $j=|B|$ **then** $L_k := A_i$; $i := i + 1$
    **else if** $A_i < B_j$ **then** $L_k := A_i$; $i := i + 1$
    **else** $L_k := B_j$; $j := j + 1$
    $k := k+1$
  **return** $L$

Takes $\Theta(|A|+|B|)$ time


(c)2001-2003, Michael P. Frank