Module #14: Recursion

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

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 $f: \mathbb{N} \to S$ (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 $a_n$ recursively:
  - Let $a_0=1$.
  - For $n>0$, let $a_n=2a_{n-1}$.

Another Example

- Suppose we define $f(n)$ for all $n \in \mathbb{N}$ recursively by:
  - Let $f(0)=3$
  - For all $n \in \mathbb{N}$, let $f(n+1)=2f(n)+3$
- What are the values of the following?
  - $f(1)=9$, $f(2)=21$, $f(3)=45$, $f(4)=93$
Module #14 - Recursion

Recursive definition of Factorial

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

The Fibonacci Series

- The Fibonacci series $f_{n=0}$ is a famous series defined by:
  
  $f_0 : 0, \quad f_1 : 1, \quad f_{n=2} : f_{n-1} + f_{n-2}$

Leonardo Fibonacci
1170-1250
Inductive Proof about Fib. series

**Theorem:** $f_n < 2^n$. — Implicitly for all $n \in \mathbb{N}$

**Proof:** By induction.

Base cases:

- $f_0 = 0 < 2^0 = 1$
- $f_1 = 1 < 2^1 = 2$

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

$< 2^{n-1} + 2^{n-2} < 2^{n-1} + 2^{n-1} = 2^n$. \(\square\)

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 $S$?
Module #14 - Recursion

The Set of All Strings

- Given an alphabet $S$, the set $S^*$ of all strings over $S$ can be recursively defined as:
  
  $e \in S^*$ (e : ∅ “”, the empty string)
  $w \in S^* \land x \in S \implies wx \in S^*$

- Exercise: Prove that this definition is equivalent to our old one: $\Sigma^* := \bigcup_{n \in \mathbb{N}} \Sigma^n$

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 \mathbb{N})
  if n = 0 then return 1
  else return a \cdot power(a, n-1)
  ```
Module #14 - Recursion

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 $m=2$, $n=0$, and $1=b<m$.

Module #14 - Recursion

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=1,n=0,m>b ∈ 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 $T(n)$ steps!
Module #14 - Recursion

**Modular Exponentiation Alg. #2**

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

```plaintext
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)·b) mod m
```

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

---

Module #14 - Recursion

**A Slight Variation**

Nearly identical but takes $T(n)$ time instead!

```plaintext
procedure mpower(b, n, m) {same signature}
  if n=0 then return 1
  else if 2 | n then
    return (mpower(b, n/2, m)·
     mpower(b, n/2, m)) mod m
  else return (mpower(b, n- 1, m)·b) mod m
```

The number of recursive calls made is critical.
Module #14 - Recursion

Recursive Euclid’s Algorithm

procedure $gcd(a,b \in \mathbb{N})$
    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.

Module #14 - Recursion

Merge Sort

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

• The merge takes $T(n)$ steps, and merge-sort takes $T(n \log n)$. 
Module #14 - Recursion

Merge Routine

**procedure** merge($A, B$: sorted lists)

$L = empty$ list

$i := 0, j := 0, k := 0$

**while** $i < |A| \land j < |B|$ \{ $|A|$ 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 $O(|A| + |B|)$ time