Izmir Institute of Technology

CENG 115
Discrete Structures

Slides are based on the Text
***Discrete Mathematics & Its Applications* (6th Edition)**
by Kenneth H. Rosen

Module #13: **Recursion**

Rosen 6th ed., §§4.3-4.4

Recursion

- *Recursion* is a general term for 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 $P(n)$.
- We can use recursion to define *algorithms*, *functions*, *sequences*, and *sets*.

§4.3: Recursive Definitions

- In mathematical induction, we prove *P* for all members of a set by proving the truth of larger members in terms of that of smaller members.
- A *recursive definition* is writing the function/predicate value of larger elements in terms of that of smaller ones provided that a starting point/element is given.

Recursively Defined Functions

- A recursive way to define a function $f:\mathbf{N}\to S$ (for any set S) or series $a_n=f(n)$ is to:
  - Specify $f(0)$.
  - 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 $a_n :\equiv 2a_{n-1}$.

Recursively Defined Functions

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?
  - $f(1)=9$ $f(2)=21$ $f(3)=45$ $f(4)=93$

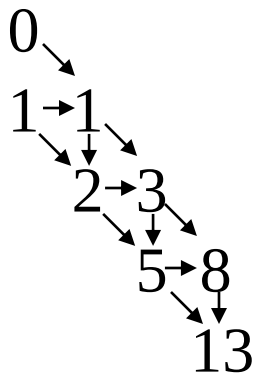
Recursive definition of Factorial

- Give a recursive definition of the factorial function $F(n) :\equiv n! :\equiv 2 \cdot 3 \cdot \ldots \cdot n$.
  - Basis step: $F(0) :\equiv 1$
  - Recursive step: $F(n) :\equiv n \cdot F(n\text{-}1)$.
    - $F(1)=1$
    - $F(2)=2$
    - $F(3)=6$
    - $F(4)=24$

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

Inductive Proof about Fib. Series

- **Theorem:** $\forall n \in \mathbf{N} \quad f_n < 2^n$
- **Proof** by induction:

Basis step: $f_0 = 0 < 2^0 = 1$

$f_1 = 1 < 2^1 = 2$

Inductive step: Use 2nd principle of induction

Assume $\forall k<n$, $f_k < 2^k$ (ind. hypothesis). Then

$$f_n = \underbrace{f_{n-1} + f_{n-2} < 2^{n-1} + 2^{n-2}}_{\text{inductive hypothesis}} < 2^{n-1} + 2^{n-1} = 2^n.$$

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.
- **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 Σ, the set Σ^* of all strings over Σ can be recursively defined as:

  Basis step:

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

  Recursive step:

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

§4.4: Recursive Algorithms

- A *recursive algorithm* solves a problem by reducing it to an instance of the same problem with smaller input.
- Example: A recursive alg. to compute a^n.

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

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

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

Efficiency of Recursive Algorithms

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

Modular Exponentiation Alg. #1

Uses the fact that

b^n **mod** $m = (b \cdot (b^{n-1}$ **mod** $m))$ **mod** m.

procedure *mpower* (integers $b \geq 1, n \geq 0, m \geq 2$)

if $n=0$ **then** *mpower*$(b,n,m)=1$

else

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

{*mpower*$(b,n,m) = b^n$ **mod** m}

Note: This algorithm's time complexity is $\Theta(n)$
(assume each *mpower* call takes a constant time)

Modular Exponentiation Alg. #2

Uses the fact that

b^n **mod** $m = (b^{n/2}$ **mod** $m)^2$ **mod** m.

procedure *mpower*(b,n,m)

if $n=0$ **then** *mpower*(b,n,m)$=1$

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

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

else

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

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

Recursive Euclid's Algorithm

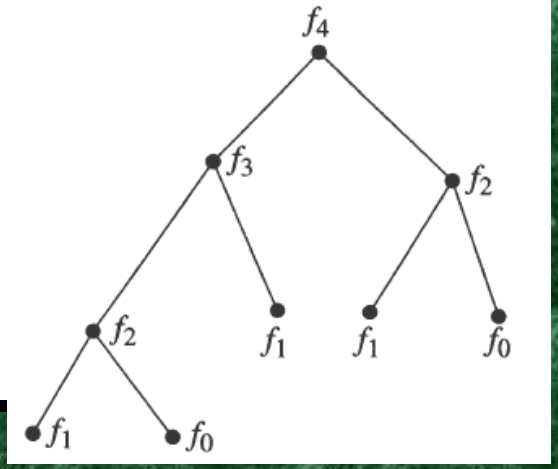
procedure $gcd(a,b \in \mathbf{N})$
  if $a = 0$ **then**
    $gcd(a,b) := b$
  else
    $gcd(a,b) := gcd(b \textbf{ mod } a, a)$

Note that recursive algorithms are often simpler to code than iterative ones, but they may require more computation (next slide).

Recursive Algorithm for Fib. Series

procedure *fibonacci* ($n \in \mathbf{N}$)
  if $n = 0$ **then** *fibonacci*(n):= 0
  else if $n = 1$ **then** *fibonacci*(n):= 1
  else
    fibonacci(n):= *fibonacci*(n-1)+*fibonacci*(n-2)

The tree (recursive calls) ⟶
causes f_{n+1}-1 addition operations



Iterative Algorithm for Fib. Series

procedure *iterative fibonacci* ($n \in \mathbf{N}$)
  if $n = 0$ **then** $y := 0$
  else begin
    $x := 0$, $y := 1$
    for $i := 1$ **to** $n-1$
      $z := x+y$, $x := y$, $y := z$
  end {*y is the* n^{th} *fibonacci number*}

How many addition operations it takes to find f_n ?

n-1 operations, more efficient than recursive way.