University of Hawaii

ICS141: Discrete Mathematics for Computer Science I

Dept. Information & Computer Sci., University of Hawaii

Jan Stelovsky

based on slides by Dr. Baek and Dr. Still

Originals by Dr. M. P. Frank and Dr. J.L. Gross

Provided by McGraw-Hill

Lecture 22

Chapter 4. Induction and Recursion

4.3 Recursive Definitions and Structural Induction

4.4 Recursive Algorithms

Review: Recursive Definitions

- ***Recursion*** is the general term for the practice of defining an object in terms of *itself* or of part of itself.

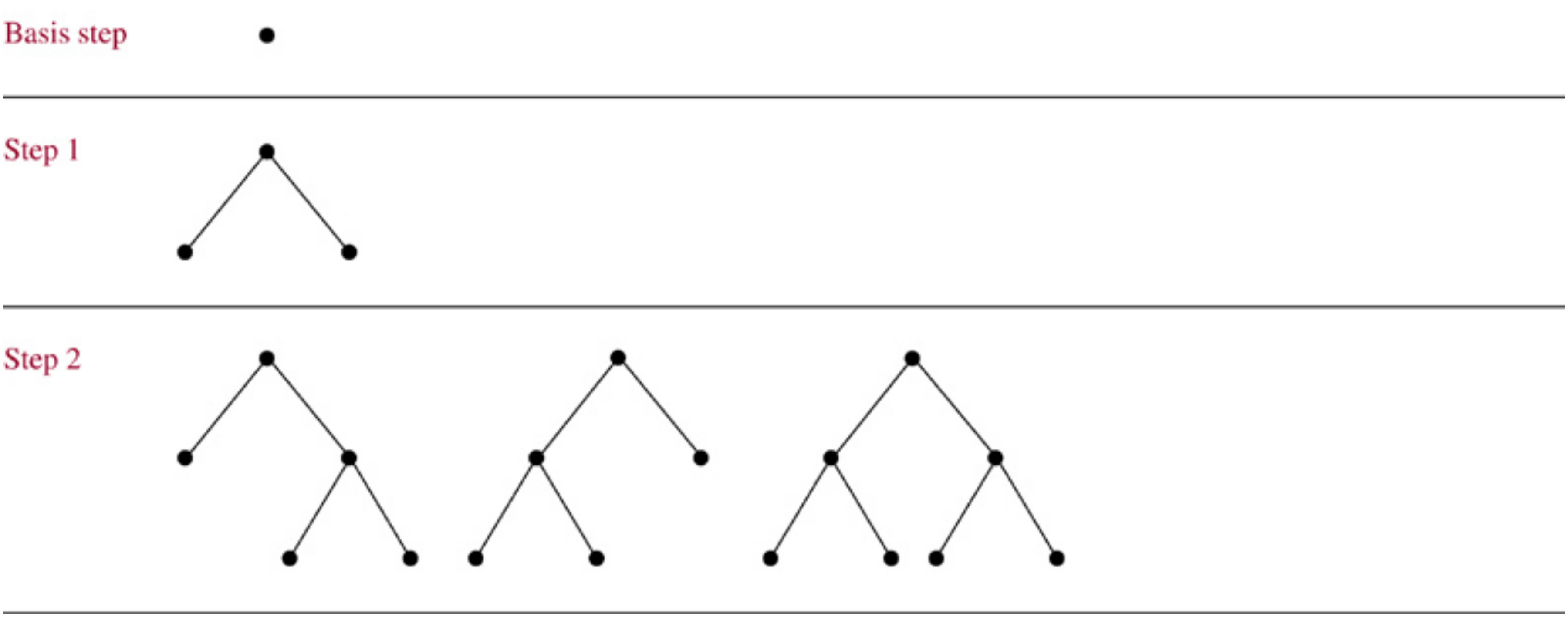
- In ***recursive definitions***, we similarly *define* a function, a predicate, a set, or a more complex structure over an infinite domain (universe of discourse) by:
  - defining the function, predicate value, set membership, or structure of larger elements in terms of those of smaller ones.

Full Binary Trees

- A special case of extended binary trees.
- Recursive definition of full binary trees:
  - **Basis step**: A single node r is a full binary tree.
    - Note this is different from the extended binary tree base case.
  - **Recursive step**: If T_1, T_2 are disjoint full binary trees with roots r_1 and r_2, then $\{(r, r_1), (r, r_2)\} \cup T_1 \cup T_2$ is an full binary tree.

Building Up Full Binary Trees

University of Hawaii



Structural Induction

- Proving something about a recursively defined object using an inductive proof whose structure mirrors the object's definition.
  - **Basis step**: Show that the result holds for all elements in the set specified in the basis step of the recursive definition
  - **Recursive step**: Show that if the statement is true for each of the elements in the new set constructed in the recursive step of the definition, the result holds for these new elements.

Structural Induction: Example

- Let $3 \in S$, and let $x+y \in S$ if $x,y \in S$.
  Show that S is the set of positive multiples of 3.
- Let $A = \{n \in \mathbf{Z}^+ \mid (3|n)\}$. We'll show that $A = S$.
  - **Proof:** We show that $A \subseteq S$ and $S \subseteq A$.
  - To show $A \subseteq S$, show $[n \in \mathbf{Z}^+ \wedge (3|n)] \rightarrow n \in S$.
    - **Inductive proof.** Let $n \in \mathbf{Z}^+$ and $P(n) = 3n \in S$.
      Induction over positive multiples of 3.

      Basis case: $n = 1$, thus $3 \in S$ by definition of S.

      Inductive step: Given $P(k)$, prove $P(k+1)$.
      By inductive hypothesis $3k \in S$, and $3 \in S$,
      so by definition of S, $3(k + 1) = 3k + 3 \in S$.

Example *cont.*

- To show $S \subseteq A$: let $n \in S$, show $n \in A$.
  - **Structural inductive proof.** Let $P(n) = n \in A$.

    Two cases: $n = 3$ (basis case), which is in A, or $n = x + y$ for $x, y \in S$ (recursive step).

    We know x and y are positive, since neither rule generates negative numbers.

    So, $x < n$ and $y < n$, and so we know x and y are in A, by strong inductive hypothesis.

    Since $3|x$ and $3|y$, we have $3|(x+y)$, thus $x + y = n \in A$. ■

Recursive Algorithms

University of Hawaii

- Recursive definitions can be used to describe functions and sets as well as *algorithms*.
- A *recursive procedure* is a procedure that invokes itself.
- A *recursive algorithm* is an algorithm that contains a recursive procedure.
- *An algorithm is called recursive if it solves a problem by reducing it to an instance of the same problem with smaller input.*

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

problem

divide-and-conquer technique

subproblems
of the same type
as the original problem

Recursive Euclid's Algorithm

University of Hawaii

- gcd(a, b) = gcd((b mod a), a)

  procedure *gcd*(a,$b$$\in$**N** with $a < b$)

  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

Recursive Linear Search

{Finds x in series a at a location $\geq i$ and $\leq j$}

procedure *search*

  (a: series; i, j: integer; x: item to find)

 if $a_i = x$ **return** i {At the right item? Return it!}

 if $i = j$ **return** 0 {No locations in range? Failure!}

 return *search*(a, i +1, j, x) {Try rest of range}

- Note there is no real advantage to using recursion here over just looping

  for *loc* := i to j...

  recursion is slower because procedure call costs

Recursive Binary Search

{Find location of x in a, $\geq i$ and $\leq j$}

procedure *binarySearch*(a, x, i, j)

$m := \lfloor (i + j)/2 \rfloor$ {Go to halfway point}

if $x = a_m$ **return** m {Did we luck out?}

if $x < a_m \wedge i < m$ {If it's to the left, check that ½}

return *binarySearch*(a, x, i, $m-1$)

else if $x > a_m \wedge j > m$ {If it's to right, check that ½}

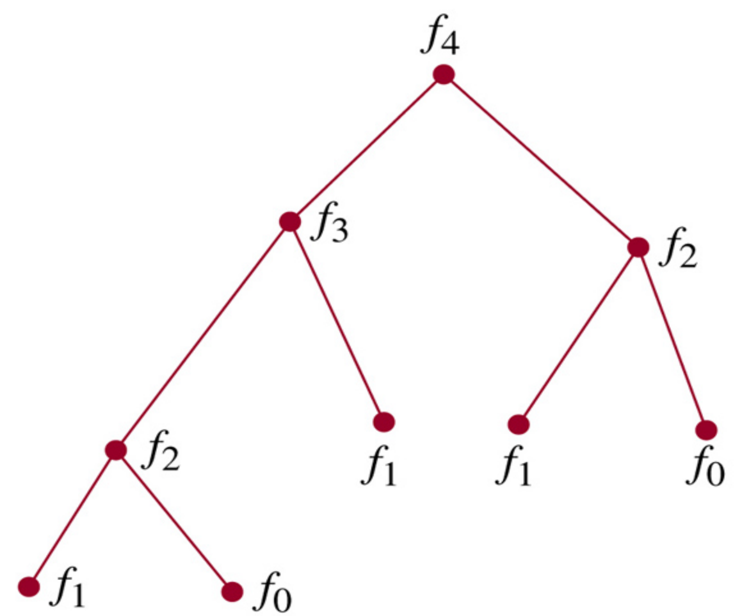
return *binarySearch*(a, x, $m+1$, j)

else return 0 {No more items, failure.}

Recursive Fibonacci Algorithm

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

- Is this an efficient algorithm?
- How many additions are performed?



Analysis of Fibonacci Procedure

- **Theorem:** The recursive procedure *fibonacci*(n) performs $f_{n+1} - 1$ additions.
  - **Proof:** By strong structural induction over n, based on the procedure's own recursive definition.
    - **Basis step:**
      - *fibonacci*(0) performs 0 additions, and $f_{0+1} - 1 = f_1 - 1 = 1 - 1 = 0$.
      - Likewise, *fibonacci*(1) performs 0 additions, and $f_{1+1} - 1 = f_2 - 1 = 1 - 1 = 0$.

Analysis of Fibonacci Procedure

University of Hawaii

- **Inductive step**:

  $$\mathit{fibonacci}(k+1) = \underbrace{\mathit{fibonacci}(k)}_{\substack{\text{by } P(k): \\ f_{k+1} - 1 \text{ additions}}} + \underbrace{\mathit{fibonacci}(k-1)}_{\substack{\text{by } P(k-1): \\ f_k - 1 \text{ additions}}}$$

  - For $k > 1$, by strong inductive hypothesis, *fibonacci*(k) and *fibonacci*($k-1$) do $f_{k+1} - 1$ and $f_k - 1$ additions respectively.
  - *fibonacci*(k+1) adds 1 more, for a total of

    $$(f_{k+1} - 1) + (f_k - 1) + 1 = f_{k+1} + f_k - 1$$
    $$= f_{k+2} - 1. \blacksquare$$

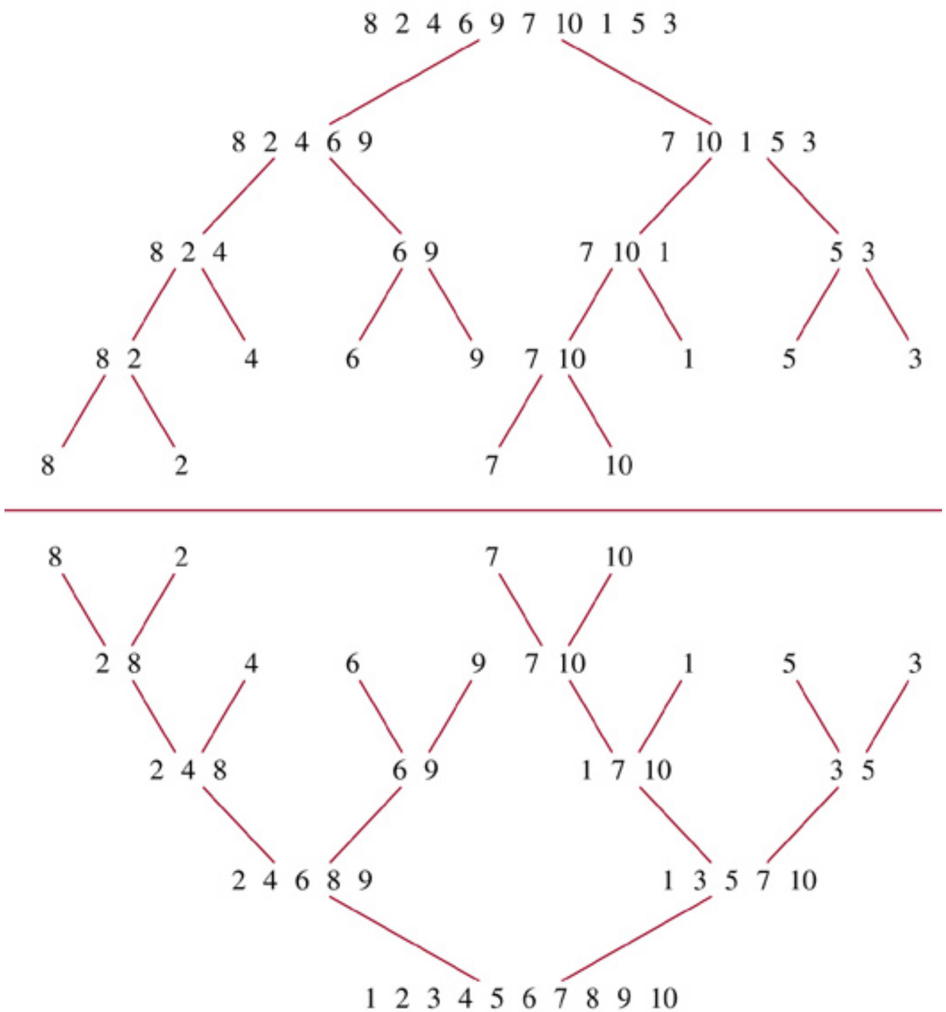
Iterative Fibonacci Algorithm

University of Hawaii

```
procedure iterativeFib(n ∈ N)
   if n = 0 then
        return 0
   else begin
        x := 0
        y := 1
        for i := 1 to n – 1 begin
             z := x + y
             x := y
             y := z
        end
   end
   return y    {the nth Fibonacci number}
```

Requires only $n-1$ additions

Recursive Merge Sort Example



Split

Merge

Recursive Merge Sort

University of Hawaii

procedure *mergesort*($L = \ell_1, \ldots, \ell_n$)

  if $n > 1$ **then**

    $m := \lfloor n/2 \rfloor$ {this is rough ½-way point}

    $L_1 := \ell_1, \ldots, \ell_m$

    $L_2 := \ell_{m+1}, \ldots, \ell_n$

    $L := merge(mergesort(L_1), mergesort(L_2))$

  return L

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

Merging Two Sorted Lists

TABLE 1 Merging the Two Sorted Lists 2, 3, 5, 6 and 1, 4.

First List	*Second List*	*Merged List*	*Comparison*
2 3 5 6	1 4		1 < 2
2 3 5 6	4	1	2 < 4
3 5 6	4	1 2	3 < 4
5 6	4	1 2 3	4 < 5
5 6		1 2 3 4	
		1 2 3 4 5 6	

Recursive Merge Method

University of Hawaii

{Given two sorted lists $A = (a_1, \ldots, a_{|A|})$, $B = (b_1, \ldots, b_{|B|})$, returns a sorted list of all.}

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

  if A = empty **return** B {If A is empty, it's B.}

  if B = empty **return** A {If B is empty, it's A.}

  if $a_1 < b_1$ **then**

    return (a_1, *merge*($(a_2, \ldots, a_{|A|})$, B))

  else

    return (b_1, *merge*(A, $(b_2, \ldots, b_{|B|})$))

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 $1 \leq b < m$.

Modular Exponentiation #1

University of Hawaii

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

  {Returns b^n **mod** m.}
  procedure *mpower*
  (b, n, m: integers with $m \geq 2$, $n \geq 0$, and $1 \leq b < m$)
  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 #2

- Uses the fact that $b^{2k} = b^{k\cdot 2} = (b^k)^2$.
- Then, b^{2k} **mod** $m = (b^k$ **mod** $m)^2$ **mod** m.

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 ($b\cdot$*mpower*(b,$n-1$,m)) **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)·

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!