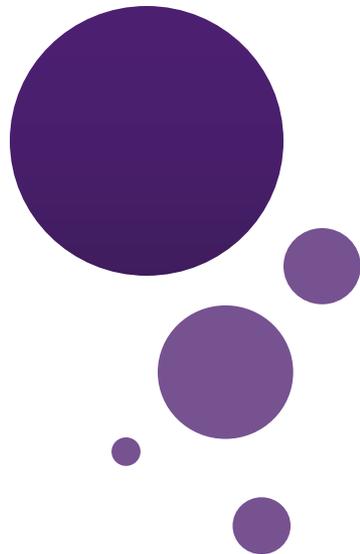




UNIVERSITY
AT ALBANY

State University of New York



Lecture 23: Recursive Algorithms

Dr. Chengjiang Long

Computer Vision Researcher at Kitware Inc.

Adjunct Professor at SUNY at Albany.

Email: clong2@albany.edu

Outline

- Definition of Recursive Algorithms
- Several Recursive Algorithms
- Efficiency of Recursive Algorithms

Outline

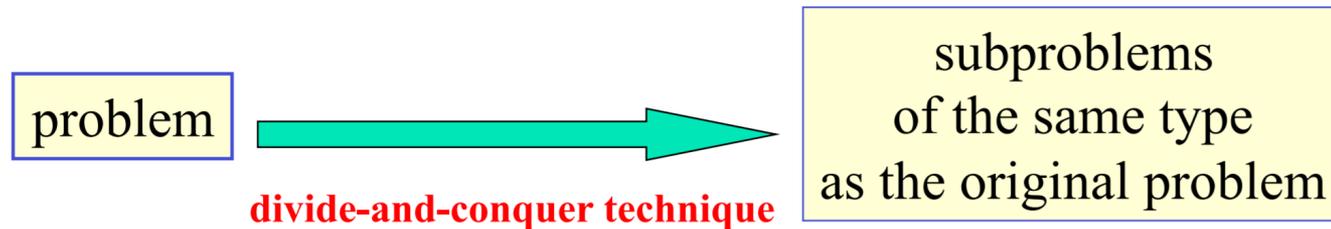
- **Definition of Recursive Algorithms**
- Several Recursive Algorithms
- Efficiency of Recursive Algorithms

Recursive Algorithms

- 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, $n \in \mathbb{N}$)
if $n = 0$ **then return** 1
else return $a \cdot \text{power}(a, n-1)$



- ❑ 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

Outline

- Definition of Recursive Algorithms
- **Several Recursive Algorithms**
- Efficiency of Recursive Algorithms

Recursive Euclid's Algorithm

- $\text{gcd}(a, b) = \text{gcd}((b \bmod a), a)$

procedure $\text{gcd}(a, b \in \mathbb{N}$ with $a < b$)

if $a = 0$ **then return** b

else return $\text{gcd}(b \bmod a, a)$

1220 mod 516 = 188
516 mod 188 = 140
188 mod 140 = 48
140 mod 48 = 44
48 mod 44 = 4
44 mod 4 = 0
4 = GCD

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 .}

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

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

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

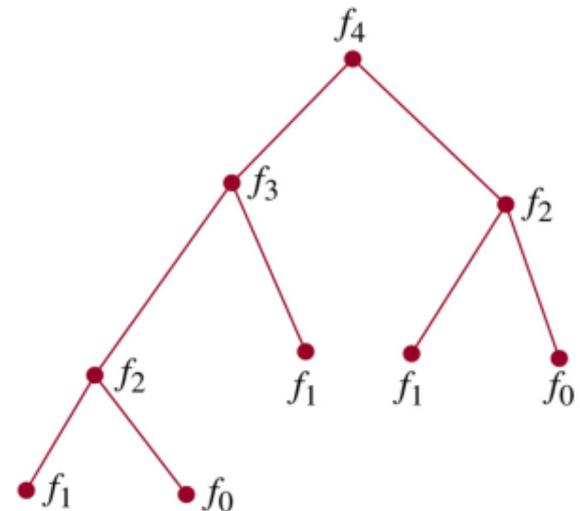
else return 0 {No more items, failure.}

Recursive Fibonacci Algorithm

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

© The McGraw-Hill Companies, Inc. all rights reserved.

- 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

Inductive step:

$$\text{fibonacci}(k+1) = \text{fibonacci}(k) + \text{fibonacci}(k-1)$$

by $P(k)$:

$f_{k+1} - 1$ additions

by $P(k-1)$:

$f_k - 1$ additions

- For $k > 1$, by strong inductive hypothesis, $\text{fibonacci}(k)$ and $\text{fibonacci}(k-1)$ do $f_{k+1} - 1$ and $f_k - 1$ additions respectively.
- $\text{fibonacci}(k+1)$ adds 1 more, for a total of
$$(f_{k+1} - 1) + (f_k - 1) + 1 = f_{k+1} + f_k - 1$$
$$= f_{k+2} - 1.$$

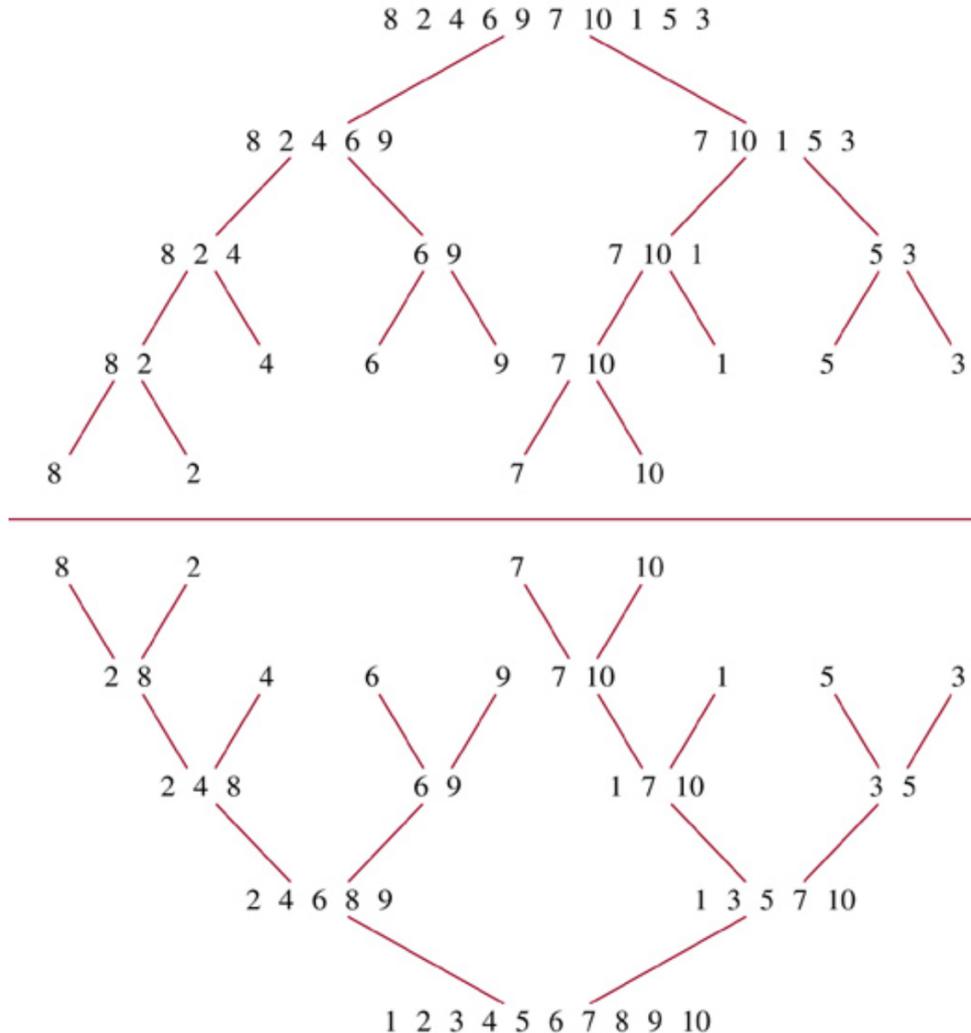
Iterative Fibonacci Algorithm

- **procedure** *iterativeFib*($n \in \mathbf{N}$)
- **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 n th Fibonacci number}

Requires only
 $n - 1$ additions

Recursive Merge Sort Example

© The McGraw-Hill Companies, Inc. all rights reserved.



Split

Merge

Recursive Merge Sort

```
procedure mergesort( $L = 1, \dots, n$ )  
if  $n > 1$  then  
   $m := \lfloor n/2 \rfloor$  {this is rough  $1/2$ -way point}  
   $L1 := 1, \dots, m$   
   $L2 := m+1, \dots, n$   
   $L := \text{merge}(\text{mergesort}(L1), \text{mergesort}(L2))$   
return  $L$ 
```

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

Merging Two Sorted Lists

© The McGraw-Hill Companies, Inc. all rights reserved.

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

<i>First List</i>	<i>Second List</i>	<i>Merged List</i>	<i>Comparison</i>
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

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

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

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

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

if $a_1 < b_1$ **then**

return $(a_1, \text{merge}((a_2, \dots, a_{|A|}), B))$

else

return $(b_1, \text{merge}(A, (b_2, \dots, b_{|B|})))$

Outline

- Definition of Recursive Algorithms
- Several Recursive Algorithms
- **Efficiency of Recursive Algorithms**

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 $b^n \bmod m$, where $m \geq 2$, $n \geq 0$, and $1 \leq b < m$.

Modular Exponentiation #1

- Uses the fact that $b^n = b \cdot b^{n-1}$ and that $x \cdot y \bmod m = x \cdot (y \bmod m) \bmod m$.

(Prove the latter theorem at home.)

{Returns $b^n \bmod m$.}

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 $(b \cdot \text{mpower}(b, n-1, m)) \bmod m$

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

Modular Exponentiation #2

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

```
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·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!

Next class

- Topic: Recursive Algorithms and Basic Counting Rules
- Pre-class reading: Chap 5.4 and Chap 6.1

