

# MAT 305: Mathematical Computing

## Recursion

John Perry

University of Southern Mississippi

Fall 2011

# Outline

- ① Recursion?
- ② Issues in recursion
- ③ Summary

# Outline

- 1 Recursion?
- 2 Issues in recursion
- 3 Summary

# Recursion?

*re + cursum*: return, travel the path again (Latin)

Two (similar) views:

- **mathematical**: a function defined using itself;
- **computational**: an algorithm that invokes itself.

# When recursion?

- At least one “base” case w/closed form
  - (“closed” = no recursion)
- All recursive chains terminate at base case

# Example: Proof by induction

Recursion?

Issues in  
recursion

Summary

Prove  $P(n)$  for all  $n \in \mathbb{N}$ :

*Inductive Base:* Show  $P(1)$

*Inductive Hypothesis:* Assume  $P(i)$  for  $1 \leq i \leq n$

*Inductive Step:* Show  $P(n+1)$  using  $P(i)$  for  $1 \leq i \leq n$

## Fibonacci's Bunnies

Fibonacci (Leonardo da Pisa) describes in *Liber Abaci* a population of bunnies:

- first month: one pair of bunnies;

## Fibonacci's Bunnies

Fibonacci (Leonardo da Pisa) describes in *Liber Abaci* a population of bunnies:

- first month: one pair of bunnies;
- second month: pair matures;
- third month: mature pair produces new pair;



## Fibonacci's Bunnies

Fibonacci (Leonardo da Pisa) describes in *Liber Abaci* a population of bunnies:

- first month: one pair of bunnies;
- second month: pair matures;
- third month: mature pair produces new pair;
- fourth month: second pair matures, first pair produces new pair;

## Fibonacci's Bunnies

Fibonacci (Leonardo da Pisa) describes in *Liber Abaci* a population of bunnies:

- first month: one pair of bunnies;
- second month: pair matures;
- third month: mature pair produces new pair;
- fourth month: second pair matures, first pair produces new pair;
- fifth month: third pair matures, two mature pairs produce new pairs;
- ...

# How many pairs?

Recursion?

Issues in  
recursion

Summary

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>								
<b>immature pairs</b>								
<b>new pairs</b>	1							
<b>total pairs</b>	1							

# How many pairs?

<b>month</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>...</b>
<b>mature pairs</b>								
<b>immature pairs</b>		1						
<b>new pairs</b>	1							
<b>total pairs</b>	1	1						

Recursion?

Issues in  
recursion

Summary

# How many pairs?

Recursion?

Issues in  
recursion

Summary

<b>month</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>...</b>
<b>mature pairs</b>			1					
<b>immature pairs</b>		1						
<b>new pairs</b>	1		1					
<b>total pairs</b>	1	1	2					

# How many pairs?

Recursion?

Issues in  
recursion

Summary

<b>month</b>	<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>...</b>
<b>mature pairs</b>			1	1				
<b>immature pairs</b>		1		1				
<b>new pairs</b>	1		1	1				
<b>total pairs</b>	1	1	2	3				

## How many pairs?

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2			
<b>immature pairs</b>		1		1	1			
<b>new pairs</b>	1		1	1	2			
<b>total pairs</b>	1	1	2	3	5			

Recursion?

Issues in  
recursion

Summary

## How many pairs?

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3		
<b>immature pairs</b>		1		1	1	2		
<b>new pairs</b>	1		1	1	2	3		
<b>total pairs</b>	1	1	2	3	5	8		

Recursion?

Issues in  
recursion

Summary



## How many pairs?

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

Recursion?

Issues in  
recursion

Summary

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- $\text{total} = (\# \text{ mature} + \# \text{ immature}) + \# \text{ new}$

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- $\text{total} = (\# \text{ mature} + \# \text{ immature}) + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ new}$

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- $\text{total} = (\# \text{ mature} + \# \text{ immature}) + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ mature now}$

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- $\text{total} = (\# \text{ mature} + \# \text{ immature}) + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ mature now}$
- $\text{total} = \# \text{ one month ago} + \# \text{ two months ago}$

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- $\text{total} = (\# \text{ mature} + \# \text{ immature}) + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ new}$
- $\text{total} = \# \text{ one month ago} + \# \text{ mature now}$
- $\text{total} = \# \text{ one month ago} + \# \text{ two months ago}$

$$\therefore F_{\text{now}} = F_{\text{one month ago}} + F_{\text{two months ago}}, \text{ or}$$

## Describing it

<b>month</b>	1	2	3	4	5	6	7	...
<b>mature pairs</b>			1	1	2	3	5	
<b>immature pairs</b>		1		1	1	2	3	
<b>new pairs</b>	1		1	1	2	3	5	
<b>total pairs</b>	1	1	2	3	5	8	13	...

- total = (# mature + # immature) + # new
- total = # one month ago + # new
- total = # one month ago + # mature now
- total = # one month ago + # two months ago

$$\therefore F_{\text{now}} = F_{\text{one month ago}} + F_{\text{two months ago}}, \text{ or}$$

$$F_i = F_{i-1} + F_{i-2}$$

## ∴ Fibonacci Sequence

$$F_i = \begin{cases} 1, & i = 1, 2; \\ F_{i-1} + F_{i-2}, & i \geq 3. \end{cases}$$



## ∴ Fibonacci Sequence

$$F_i = \begin{cases} 1, & i = 1, 2; \\ F_{i-1} + F_{i-2}, & i \geq 3. \end{cases}$$

### Example

$$\begin{aligned} F_5 &= F_4 + F_3 \\ &= (F_3 + F_2) + (F_2 + F_1) \\ &= [(F_2 + F_1) + F_2] + (F_2 + F_1) \\ &= 3F_2 + 2F_1 \\ &= 5. \end{aligned}$$

## ∴ Fibonacci Sequence

$$F_i = \begin{cases} 1, & i = 1, 2; \\ F_{i-1} + F_{i-2}, & i \geq 3. \end{cases}$$

### Example

$$\begin{aligned} F_5 &= F_4 + F_3 \\ &= (F_3 + F_2) + (F_2 + F_1) \\ &= [(F_2 + F_1) + F_2] + (F_2 + F_1) \\ &= 3F_2 + 2F_1 \\ &= 5. \end{aligned}$$

$$\begin{aligned} F_{100} &= F_{99} + F_{98} \\ &= \dots \\ &= 218922995834555169026 \cdot F_2 + 135301852344706746049 \cdot F_1 \\ &= 354224848179261915075 \end{aligned}$$

# Pseudocode

Easy to implement recursion:

**algorithm** *Fibonacci*

**inputs**

$n \in \mathbb{N}$

**outputs**

the  $n$ th Fibonacci number

**do**

**if**  $n = 1$  **or**  $n = 2$

**return** 1

**else**

**return**  $Fibonacci(n - 2) + Fibonacci(n - 1)$

# Implementation

Recursion?

Issues in  
recursion

Summary

```
sage: def fibonacci(n):  
        if n == 1 or n == 2:  
            return 1  
        else:  
            return fibonacci(n-2) + fibonacci(n-1)
```

# Implementation

```
sage: def fibonacci(n):  
        if n == 1 or n == 2:  
            return 1  
        else:  
            return fibonacci(n-2) + fibonacci(n-1)
```

```
sage: fibonacci(5)  
5
```

```
sage: fibonacci(20)  
6765
```

```
sage: fibonacci(30)  
832040
```

# Outline

① Recursion?

② Issues in recursion

③ Summary

# Issues in recursion

- Infinite loops
  - recursion must stop eventually
  - must ensure reach base case

## Issues in recursion

- Infinite loops
  - recursion must stop eventually
  - must ensure reach base case
- Wasted computation
  - `fibonacci(20)` requires `fibonacci(19)` and `fibonacci(18)`
  - `fibonacci(19)` *also* requires `fibonacci(18)`
  - $\therefore$  `fibonacci(18)` computed twice!



## Example

Modify program:

```
sage: def fibonacci(n):  
        print 'computing fibonacci #', n,  
        if n == 1 or n == 2:  
            return 1  
        else:  
            return fibonacci(n-2) + fibonacci(n-1)
```

## Example

Modify program:

```
sage: def fibonacci(n):  
        print 'computing fibonacci #', n,  
        if n == 1 or n == 2:  
            return 1  
        else:  
            return fibonacci(n-2) + fibonacci(n-1)
```

```
sage: fibonacci(5)  
computing fibonacci # 5 computing fibonacci # 3  
computing fibonacci # 1 computing fibonacci # 2  
computing fibonacci # 4 computing fibonacci # 2  
computing fibonacci # 3 computing fibonacci # 1  
computing fibonacci # 2  
5
```

## Example

Modify program:

```
sage: def fibonacci(n):
        print 'computing fibonacci #', n,
        if n == 1 or n == 2:
            return 1
        else:
            return fibonacci(n-2) + fibonacci(n-1)
```

```
sage: fibonacci(5)
computing fibonacci # 5 computing fibonacci # 3
computing fibonacci # 1 computing fibonacci # 2
computing fibonacci # 4 computing fibonacci # 2
computing fibonacci # 3 computing fibonacci # 1
computing fibonacci # 2
5
```

... $F_3$  computed 2 times;  $F_2$ , 3 times;  $F_1$ , 2 times

## Workaround

Can we tell Sage to “remember” pre-computed values?

- Need a list
- Compute  $F_i$ ? add value to list
- Apply formula *only* if  $F_i$  not in list!
- “Remember” computation after function ends: **global list**
  - (called a **cache**)

## Workaround

Can we tell Sage to “remember” pre-computed values?

- Need a list
- Compute  $F_i$ ? add value to list
- Apply formula *only* if  $F_i$  not in list!
- “Remember” computation after function ends: **global** list
  - (called a **cache**)

### Definition

- **global** variables available to all functions in system
- **cache** makes information quickly accessible

# Pseudocode

**algorithm** *Fibonacci\_with\_table*

**globals**  $F$ , a list of integers, initially  $[1, 1]$

**inputs**

$n \in \mathbb{N}$

**outputs**

the  $n$ th Fibonacci number

**do**

**if**  $n > \#F$

Let  $a = \text{Fibonacci\_with\_table}(n - 1)$

Let  $b = \text{Fibonacci\_with\_table}(n - 2)$

Append  $a + b$  to  $F$

**return**  $F_n$

# Hand implementation

```
sage: F = [1,1]
sage: def fibonacci_with_table(n):
    global F
    if n > len(F):
        print 'computing fibonacci #', n,
        a = fibonacci_with_table(n-2)
        b = fibonacci_with_table(n-1)
        F.append(a + b)
    return F[n-1]
```

Recursion?

Issues in  
recursion

Summary

# Hand implementation

```
sage: F = [1,1]
sage: def fibonacci_with_table(n):
    global F
    if n > len(F):
        print 'computing fibonacci #', n,
        a = fibonacci_with_table(n-2)
        b = fibonacci_with_table(n-1)
        F.append(a + b)
    return F[n-1]
```

## Example

```
sage: fibonacci_with_table(5)
computing fibonacci # 5 computing fibonacci # 4
computing fibonacci # 3
5
```



## But... no need to implement!

```
sage: @cached_function
def fibonacci(n):
    print 'computing fibonacci #', n,
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

## But... no need to implement!

```
sage: @cached_function
def fibonacci(n):
    print 'computing fibonacci #', n,
    if n == 1 or n == 2:
        return 1
    else:
        return fibonacci(n-2) + fibonacci(n-1)
```

### Example

```
sage: fibonacci(5)
computing fibonacci # 5 computing fibonacci # 3
computing fibonacci # 1 computing fibonacci # 2
computing fibonacci # 4
5
```

However...

Avoid recursion when possible

- can often rewrite as a loop
- can sometimes rewrite in “closed form”

However...

Avoid recursion when possible

- can often rewrite as a loop
- can sometimes rewrite in “closed form”

Example

“Closed form” for Fibonacci sequence:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

However...

Avoid recursion when possible

- can often rewrite as a loop
- can sometimes rewrite in “closed form”

Example

“Closed form” for Fibonacci sequence:

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

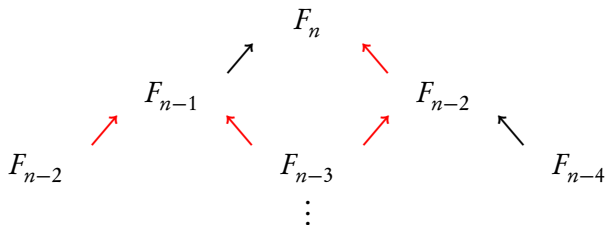
Coincidence? I think not...

$$\frac{1 + \sqrt{5}}{2} = \textit{golden ratio}$$

## Looped Fibonacci: How?

We will *not* use the closed form, but a loop

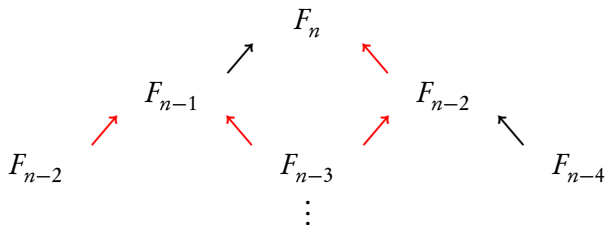
- Recursive: backwards, then forwards again



## Looped Fibonacci: How?

We will *not* use the closed form, but a loop

- Recursive: backwards, then forwards again



- Looped: direct
  - $F_2 \xrightarrow{+F_1} F_3 \xrightarrow{+F_2} \cdots \xrightarrow{+F_{n-2}} F_n$
  - remember two previous computations
  - remember?  $\implies$  variables

# Looped Fibonacci: Pseudocode

**algorithm** Looped Fibonacci

**inputs**

$$n \in \mathbb{N}$$

**outputs**

the  $n$ th Fibonacci number

**do**

— Define the base case

$$\text{Let } F_{\text{prev}} = 1, F_{\text{curr}} = 1$$

— Use the formula to move forward to  $F_n$

$$\text{Let } i = 2$$

**while**  $i < n$

— Compute next element, then move forward

$$\text{Let } F_{\text{next}} = F_{\text{prev}} + F_{\text{curr}}$$

$$\text{Let } F_{\text{prev}} = F_{\text{curr}}$$

$$\text{Let } F_{\text{curr}} = F_{\text{next}}$$

Increment  $i$

**return**  $F_{\text{curr}}$



# Looped Fibonacci: Implementation

```
sage: def looped_Fibonacci(n):  
    Fprev = 1  
    Fcurr = 1  
    i = 2  
    while (i < n):  
        Fnext = Fprev + Fcurr  
        Fprev = Fcurr  
        Fcurr = Fnext  
        i = i + 1  
    return Fcurr
```

## Looped Fibonacci: Implementation

```
sage: def looped_Fibonacci(n):  
        Fprev = 1  
        Fcurr = 1  
        i = 2  
        while (i < n):  
            Fnext = Fprev + Fcurr  
            Fprev = Fcurr  
            Fcurr = Fnext  
            i = i + 1  
        return Fcurr
```

```
sage: looped_Fibonacci(100)  
354224848179261915075
```

*(Much faster than recursive version)*

# Recursive vs. Looped vs. Closed-form

- Recursive

**pros:** simpler to write

**cons:** slower, memory intensive, *indefinite loop w/out loop structure*

- Looped

**pros:** not too slow, not too complicated, loop can be definite

**cons:** not as simple as recursive, sometime not obvious

- Closed-form

**pros:** one step (no loop)

**cons:** finding it often requires *significant* effort

# Outline

① Recursion?

② Issues in recursion

③ Summary

# Summary

- Recursion: function defined using other values of function
- Issues
  - can waste computation
  - can lead to infinite loops (bad design)
- Use when
  - closed/loop form too complicated
  - chains not too long
  - “memory table” feasible