MAT 305: Mathematical Computing

Recursion

John Perry

University of Southern Mississippi

Fall 2011

Outline

Outline

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 with no recursion
- All recursive chains terminate at base case

Proof by induction

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

Inductive Base: Show $P(1)$

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

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

Fibonacci's Bunnies

Leonardo da Pisa, called *Fibonacci*, describes in *Liber Abaci* a population of bunnies:

- first month: one pair of bunnies;

Fibonacci's Bunnies

Leonardo da Pisa, called *Fibonacci*, 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

Leonardo da Pisa, called *Fibonacci*, 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

Leonardo da Pisa, called *Fibonacci*, 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?

month	1	2	3	4	5	6	7	...
mature pairs								
immature pairs								
new pairs	1							
total pairs	1							

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs								
immature pairs		1						
new pairs	1							
total pairs	1	1						

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs			1					
immature pairs		1						
new pairs	1		1					
total pairs	1	1	2					

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs			1	1				
immature pairs		1		1				
new pairs	1		1	1				
total pairs	**1**	**1**	**2**	**3**				

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2			
immature pairs		1		1	1			
new pairs	1		1	1	2			
total pairs	**1**	**1**	**2**	**3**	**5**			

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3		
immature pairs		1		1	1	2		
new pairs	1		1	1	2	3		
total pairs	**1**	**1**	**2**	**3**	**5**	**8**		

How many pairs?

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**...**

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**...**

- total = (# mature + # immature) + # new

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**...**

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

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**1**	**1**	**2**	**3**	**5**	**8**	**13**	**...**

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

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**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

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**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

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

Describing it

month	1	2	3	4	5	6	7	...
mature pairs			1	1	2	3	5	
immature pairs		1		1	1	2	3	
new pairs	1		1	1	2	3	5	
total pairs	**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

$$\therefore F_{\text{now}} = F_{\text{one month ago}} + F_{\text{two months ago}}, \text{ or}$$
$$F_i = F_{i-1} + F_{i-2}$$

$\therefore$ 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

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

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

Pseudocode

Easy to implement recursion:

algorithm Fibonacci
inputs
  $n \in \mathbb{N}$
outputs
  the nth Fibonacci number
do
  if $n > 2$
    return $\text{Fibonacci}(n-2) + \text{Fibonacci}(n-1)$
  else
    return 1

Implementation

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

Implementation

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

sage:  fibonacci(5)
5

sage:  fibonacci(20)
6765

sage:  fibonacci(30)
832040
```

Outline

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 > 2:
           return fibonacci(n-2) + fibonacci(n-1)
         else:
           return 1
```

Example

Modify program:

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

$\ldots F_3$ 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!
- To "remember" computations after function ends, make list **global**

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!
- To "remember" computations after function ends, make list **global**

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 nth 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]
```

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 > 2:
           return fibonacci(n-2) + fibonacci(n-1)
         else:
           return 1
```

But... no need to implement!

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

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

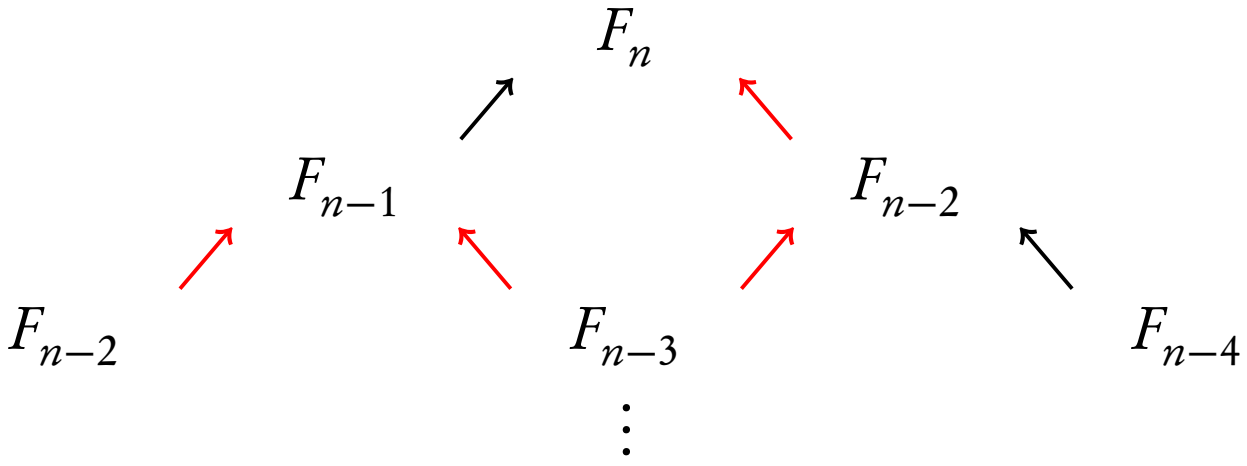
Coincidence? I think not...

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

Looped Fibonacci: How?

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

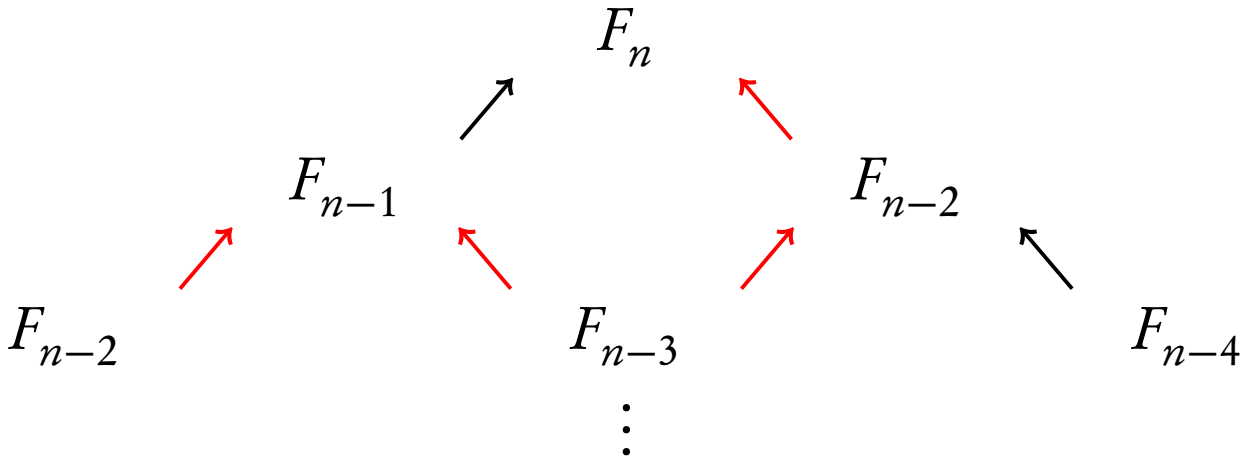
- Recursive: backwards, then forwards again



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? $\Longrightarrow$ variables

Looped Fibonacci: Pseudocode

algorithm Looped Fibonacci

inputs

$n \in \mathbb{N}$

outputs

the nth Fibonacci number

do

— Define the base case

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

— Use the formula to move forward to F_n

Let $i = 2$

while $i < n$

— Compute next element, then move forward

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

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

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

Increment i

return F_{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

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