MAT 305: Mathematical Computing

Recursion

John Perry

University of Southern Mississippi

Spring 2017

Outline

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 w/closed form
  - ("closed" = no recursion)
- All recursive chains terminate at base case

Example: Proof by induction

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

Inductive Base: Show $P(1)$

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

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

Outline

Example: Pascal's triangle

$$
\begin{array}{ccccccccccc}
 & & & & & 1 & & & & & \\
 & & & & 1 & & 1 & & & & \\
 & & & 1 & & 2 & & 1 & & & \\
 & & 1 & & 3 & & 3 & & 1 & & \\
 & 1 & & 4 & & 6 & & 4 & & 1 & \\
\iddots & & & & & \vdots & & & & & \ddots
\end{array}
$$

Pascal's triangle ⇒ binomial expansion

$$
\begin{array}{rcccccccccc}
(x+1)^0 & = & & & & & 1 & & & & \\
(x+1)^1 & = & & & & 1x & + & 1 & & & \\
(x+1)^2 & = & & & 1x^2 & + & 2x & + & 1 & & \\
(x+1)^3 & = & & 1x^3 & + & 3x^2 & + & 3x & + & 1 & \\
(x+1)^4 & = & 1x^4 & + & 4x^3 & + & 6x^2 & + & 4x & + & 1 \\
\vdots & & \iddots & & & & & & & & \ddots
\end{array}
$$

Do you notice a pattern?

				1				
			1		1			
		1		2		1		
	1		3		3		1	
1		4		6		4		1

Do you notice a pattern?

1

1 1

1 2 1

1 3 3 1

1 4 6 4 1

Classic example of recursion.

Formulating it

$$
\begin{tikzcd}
 & 1 \arrow[dr] & & p_2 \arrow[dl] \arrow[dr] & & p_3 \arrow[dl] \arrow[dr] & \cdots \\
1 & & r_2 & & r_3 & & \cdots
\end{tikzcd}
$$

$P =$ previous row, $R =$ current row

- $r_{\text{first}}, r_{\text{last}}$ both 1
- $r_i = p_{i-1} + p_1$

Pseudocode

algorithm *pascals_row*
inputs

- $i \in \mathbb{N}$, the desired row of Pascal's triangle

outputs

- the sequence of numbers in row i of Pascal's triangle

Pseudocode

algorithm *pascals_row*
inputs

- $i \in \mathbb{N}$, the desired row of Pascal's triangle

outputs

- the sequence of numbers in row i of Pascal's triangle

do
  if $i = 1$
    $R = [1]$
  else if $i = 2$
    $R = [1,1]$

Pseudocode

algorithm *pascals_row*
inputs

- $i \in \mathbb{N}$, the desired row of Pascal's triangle

outputs

- the sequence of numbers in row i of Pascal's triangle

do
  if $i = 1$
    $R = [1]$
  else if $i = 2$
    $R = [1, 1]$
  else
    $P = pascals\_row(i - 1)$
    $R = [1]$
    for $j \in (2, 3, \ldots, i - 1)$
      append $P_{j-1} + P_j$ to R
    append 1 to R
  return R

Sage code

```
def pascals_row(i):
    if i == 1:
        R = [1]
    elif i == 2:
        R = [1, 1]
    else:
        # compute previous row first
        P = pascals_row(i - 1)
        # this row starts with 1...
        R = [1]
        # ...adds two above next in this row...
        for j in xrange(1, i - 1):
            R.append(P[j-1] + P[j])
        # ...  and ends with 1
        R.append(1)
    return R
```

Example

```
sage: pascals_row(3)
[1, 2, 1]
sage: pascals_row(5)
[1, 4, 6, 4, 1]
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

What happened there?

```
if i == 1:
    R = [1]
elif i == 2:
    R = [1, 1]
else:
    P = pascals_row(i - 1)
    R = [1]
    for j in xrange(1, i - 1):
        R.append(P[j-1] + P[j])
    R.append(1)
return R
```

```
pascals_row(5)
pascals_row(4)
pascals_row(3)
pascals_row(2)
```

Outline

Example: Fibonacci's Bunnies

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

- first month: one pair of bunnies;

Example: 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;

Example: 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;

Example: 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?

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

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

∴ 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 w/recursion:

algorithm *Fibonacci*

inputs

  $n \in \mathbb{N}$

outputs

  the nth Fibonacci number

do

  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)
```

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

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!

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!
- Limit to recursion
  - `pascals_row(1000)`

Example

Modify program:

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

Example

Modify program:

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

Outline

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 nth Fibonacci number

do

  if $n > \#F$

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

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

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

But... no need to implement!

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

Example

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

Outline

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...

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

Outline

Looped Fibonacci: How?

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

- Recursive: down, then up, then down, then up…

$$\begin{tikzcd}
 & & F_n & & \\
 & F_{n-1} \arrow[ur] & & F_{n-2} \arrow[ul] & \\
F_{n-2} \arrow[ur] & & F_{n-3} \arrow[ul] \arrow[ur] & & F_{n-4} \arrow[ul] \\
 & & \vdots & &
\end{tikzcd}$$

Looped Fibonacci: How?

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

- Recursive: down, then up, then down, then up…

$$\begin{tikzcd}
 & & F_n & & \\
 & F_{n-1} \arrow[ur] & & F_{n-2} \arrow[ul] & \\
F_{n-2} \arrow[ur] & & F_{n-3} \arrow[ul] \arrow[ur] & & F_{n-4} \arrow[ul] \\
 & & \vdots & &
\end{tikzcd}$$

- Looped: only up, directly!
  - $F_2 \xrightarrow[+F_1]{} F_3 \xrightarrow[+F_2]{} \cdots \xrightarrow[+F_{n-2}]{} F_n$
  - 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

for $i \in \{3, \ldots, n\}$

— Compute next element, then move forward

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

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

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

return F_{curr}

Looped Fibonacci: Implementation

```
sage:  def looped_Fibonacci(n):
         Fprev = 1
         Fcurr = 1
         for i in xrange(3,n+1):
           Fnext = Fprev + Fcurr
           Fprev = Fcurr
           Fcurr = Fnext
         return Fcurr
```

Looped Fibonacci: Implementation

```
sage: def looped_Fibonacci(n):
        Fprev = 1
        Fcurr = 1
        for i in xrange(3,n+1):
          Fnext = Fprev + Fcurr
          Fprev = Fcurr
          Fcurr = Fnext
        return Fcurr
sage: looped_Fibonacci(100)
354224848179261915075
```

(Much faster than recursive version)

Faster, too

```
sage: %time a = looped_Fibonacci(30000)
CPU time: 0.01 s, Wall time: 0.01 s
sage: %time a = Fibonacci_with_table(30000)
CPU time: probably crashes, Wall time: if not, get
some coffee
sage: %time a = Fibonacci(10000)
CPU time: probably crashes, Wall time: if not,
come back tomorrow
```

Recursive vs. Looped vs. Closed-form

- Recursive

  pros: simple to write, "naïve" approach

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

Recursive vs. Looped vs. Closed-form

- Recursive
  - pros: simple to write, "naïve" approach
  - cons: slower, memory intensive, *indefinite loop w/out loop structure*
- Looped (also called **dynamic programming**)
  - pros: not too slow, not too complicated, loop can be definite
  - cons: not as simple as recursive, sometime not obvious

Recursive vs. Looped vs. Closed-form

- Recursive

  pros: simple to write, "naïve" approach
  cons: slower, memory intensive, *indefinite loop w/out loop structure*

- Looped (also called **dynamic programming**)

  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

Neat fact of eigenvectors

Theorem (Eigendecomposition)

Let M be an $n \times n$ matrix with

- *independent eigenvectors $\mathbf{e}_1, \ldots, \mathbf{e}_n$*
- *corresponding to eigenvalues $\lambda_1, \ldots, \lambda_n$.*

We can rewrite M as $M = Q\Lambda Q^{-1}$ where

$$Q = (\mathbf{e}_1 | \mathbf{e}_2 | \cdots | \mathbf{e}_n) \qquad \Lambda = \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix}.$$

Example

With M as defined,

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \Lambda = \begin{pmatrix} 2 & \\ & -2 \end{pmatrix}$$

Verify in Sage that $M = Q\Lambda Q^{-1}$

Example

With M as defined,

$$Q = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \qquad \Lambda = \begin{pmatrix} 2 & \\ & -2 \end{pmatrix}$$

Verify in Sage that $M = Q\Lambda Q^{-1}$

```
sage: Q = matrix(2,2,[1,1,1,-1])
sage: L = matrix(2,2,[2,0,0,-2])
sage: Q*L*Q**(-1)
[0 2]
[2 0]
```

$$\ldots\text{recall } M = \begin{pmatrix} 0 & 2 \\ 2 & 0 \end{pmatrix}$$

But how is this useful?

Consider the numbers

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

But how is this useful?

Consider the numbers

$$1, 1, 2, 3, 5, 8, 13, \ldots$$

This is the well-known Fibonacci sequence:

$$f_1 = 1 \quad f_2 = 1 \quad f_n = f_{n-1} + f_{n-2}$$

Can we get a "non-recursive" formula?

Fibonacci matrix

As a matrix equation,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

Let's try rewriting the matrix

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Fibonacci matrix

As a matrix equation,

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} f_{n-1} \\ f_{n-2} \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

Let's try rewriting the matrix

$$F = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}.$$

Iterative multiplication generates the sequence

```
sage:  F = matrix(2,2,[1,1,1,0])
sage:  f12 = vector([1,1])
sage:  F*f12
[2, 1]
sage:  F^2*f12
[3, 2]
sage:  F^3*f12
[5, 3]
...
```

In short,

$$F^{n-2}\begin{pmatrix} f_2 \\ f_1 \end{pmatrix} = \begin{pmatrix} f_n \\ f_{n-1} \end{pmatrix}$$

and

$$\begin{aligned} F^{n-2} &= \left(Q\Lambda Q^{-1}\right)^{n-2} \\ &= \underbrace{\left(Q\Lambda Q^{-1}\right)\left(Q\Lambda Q^{-1}\right)\cdots\left(Q\Lambda Q^{-1}\right)}_{n-2} \\ &= \underbrace{Q\Lambda\left(Q^{-1}Q\right)\Lambda\left(Q^{-1}Q\right)\cdots\left(Q^{-1}Q\right)\Lambda Q^{-1}}_{n-2} \\ &= Q\Lambda^{n-2}Q^{-1} \end{aligned}$$

Since Λ is diagonal, it is easy to compute Λ^n

What to do?

General outline:

- Compute eigenvectors and eigenvalues
```
sage:  F.eigenvectors_right()
```
- Construct $Q\Lambda^n Q^{-1}$
```
sage:  Q = matrix(2,2,[...])
sage:  L = matrix(2,2,[...])
```
- Analyze the equation

One "drawback"

- eigenvectors, eigenvalues look inexact

```
sage:  F.eigenvectors_right()
[(-0.618033988749895?,
    [(1, -1.618033988749895?)], 1),
 (1.618033988749895?,
    [(1, 0.618033988749895?)], 1)]
```

One “drawback”

- eigenvectors, eigenvalues look inexact

```
sage:  F.eigenvectors_right()
[(-0.618033988749895?,
    [(1, -1.618033988749895?)], 1),
 (1.618033988749895?,
    [(1, 0.618033988749895?)], 1)]
```

- In fact, we can determine their exact values

```
sage:  edata = F.eigenvectors_right()
sage:  lam1, lam2 = edata[0][0], edata[1][0]
sage:  lam1 = lam1.radical_expression(); lam1
-1/2*sqrt(5) + 1/2
sage:  lam2 = lam2.radical_expression(); lam2
1/2*sqrt(5) + 1/2
```

Put it together

```
[(-1/2*sqrt(5) + 1/2, [(1, -1/2*sqrt(5) - 1/2)], 1),
  (1/2*sqrt(5) + 1/2, [(1, 1/2*sqrt(5) - 1/2)], 1)]
```

```
sage:  Q = matrix(
            [1, -1/2*sqrt(5) - 1/2],
            [1,1/2*sqrt(5) - 1/2]
           )
sage:  var('n')
sage:  L = matrix(2,2,[
            (-1/2*sqrt(5) + 1/2)^(n-2), 0,
            0, (1/2*sqrt(5) + 1/2)^(n-2)
           ])
sage:  Q*L*Q**(-1)
```

…very unpleasant

…or is it?

Let $M = \left(Q\Lambda^n Q^{-1}\right)\binom{1}{1}$, and let $f_n = M_{1,1}$ (the top entry).

An "algebraic massage" (`.full_simplify()`) gives

$$f_n = \frac{\sqrt{5}}{10}\left[\left(3+\sqrt{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(3-\sqrt{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right],$$

already a "pleasant" closed form, and thus what we wanted.

…or is it?

Let $M = (Q\Lambda^n Q^{-1})\binom{1}{1}$, and let $f_n = M_{1,1}$ (the top entry).

An "algebraic massage" (`.full_simplify()`) gives

$$f_n = \frac{\sqrt{5}}{10}\left[\left(3+\sqrt{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{n-2} - \left(3-\sqrt{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right],$$

already a "pleasant" closed form, and thus what we wanted.

But we can do better!

More algebraic massage...

Use Sage (in particular, `expand()`) to verify that

$$3+\sqrt{5}=2\left(\frac{1+\sqrt{5}}{2}\right)^2 \quad \text{and} \quad 3-\sqrt{5}=2\left(\frac{1-\sqrt{5}}{2}\right)^2.$$

More algebraic massage...

Use Sage (in particular, `expand()`) to verify that

$$3+\sqrt{5}=2\left(\frac{1+\sqrt{5}}{2}\right)^2 \quad \text{and} \quad 3-\sqrt{5}=2\left(\frac{1-\sqrt{5}}{2}\right)^2 .$$

We can use this fact to rewrite

$$f_n=\frac{\sqrt{5}}{10}\left[\left(3+\sqrt{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^{n-2}-\left(3-\sqrt{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^{n-2}\right]$$

as...

Binet's Formula

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

golden ratio

(kindly observe a moment of reverent awe)

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