CS 61A Discussion 3

Recursion and Tree Recursion

Albert Xu

Attendance: links.cs61a.org/albert-disc
Slides: albertxu.xyz/teaching/cs61a/

Announcements

Wednesday, July 10

- Small Group Tutoring Section Sign-ups are open! See Piazza @220 for enrollment details.
- Project 2, Typing Test is released.
  - Phase 1/2 due Friday, July 12th at 11:59 PM PDT.
  - Entire project is due Tuesday, July 23rd at 11:59 PM PDT. You may complete all phases with *one* partner.
  - Can get 1 point of extra credit by submitting by Monday, July 22nd at 11:59 PM PDT and filling out a feedback survey
- HW3 will be released tonight and will be due Tuesday, July 16th at 11:59 PM PDT.
- Midterm and Final will be held 6-9 PM PDT (except for W61A students the final remotely)
  - Conflicts? Fill out links.cs61a.org/mt-conflict or links.cs61a.org/final-conflict by 7/9 at 11:59 PM PDT

*Albert's extra OH next week 6-7pm
TuTh

Recursion: an example

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

Recursion: an example

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

Here are the two key parts of recursion:

1) Base case
2) Recursive call

Look for them in this example.

Recursion: an example

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

Recursion: an example

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

Recursion: an example

```
1   def factorial(n):
2       if n == 0:
3           return 1
4       return n * factorial(n - 1)
```

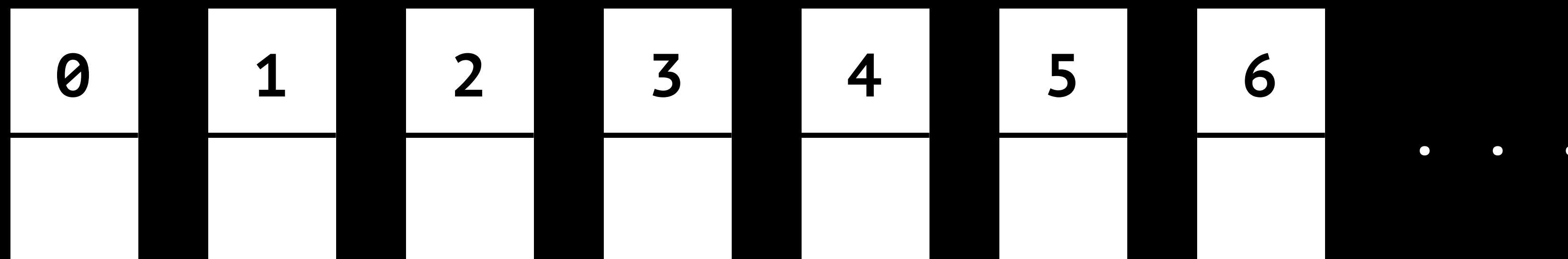
Base case

Recursive call (and multiply it by n)

This function works because n! = n * (n - 1)!,
which is our recursive call.

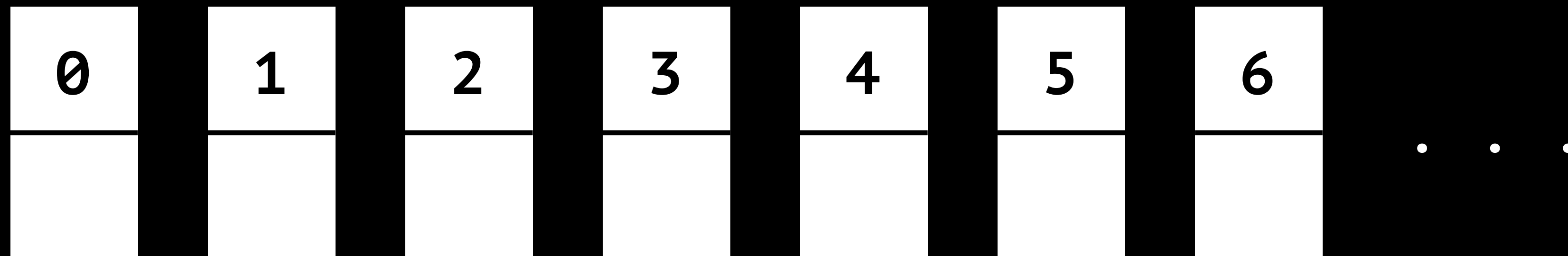
Recursion: dominoes

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```



Recursion: dominoes

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

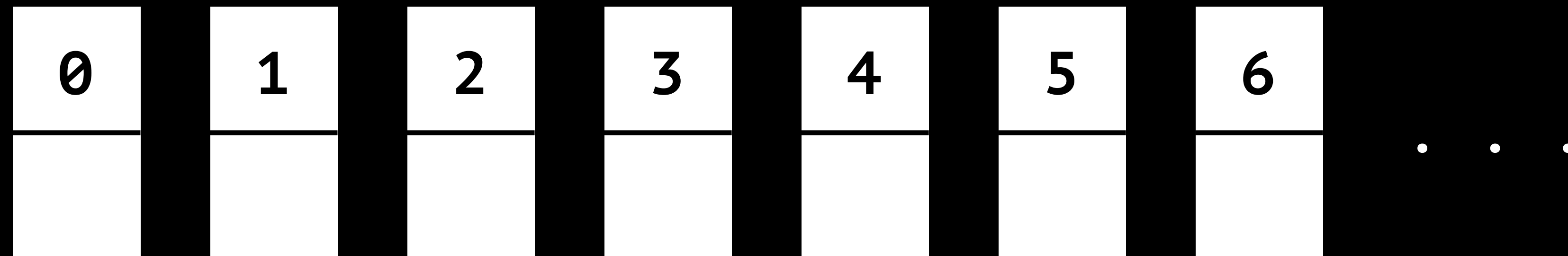


Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

Let's calculate factorial(4)!

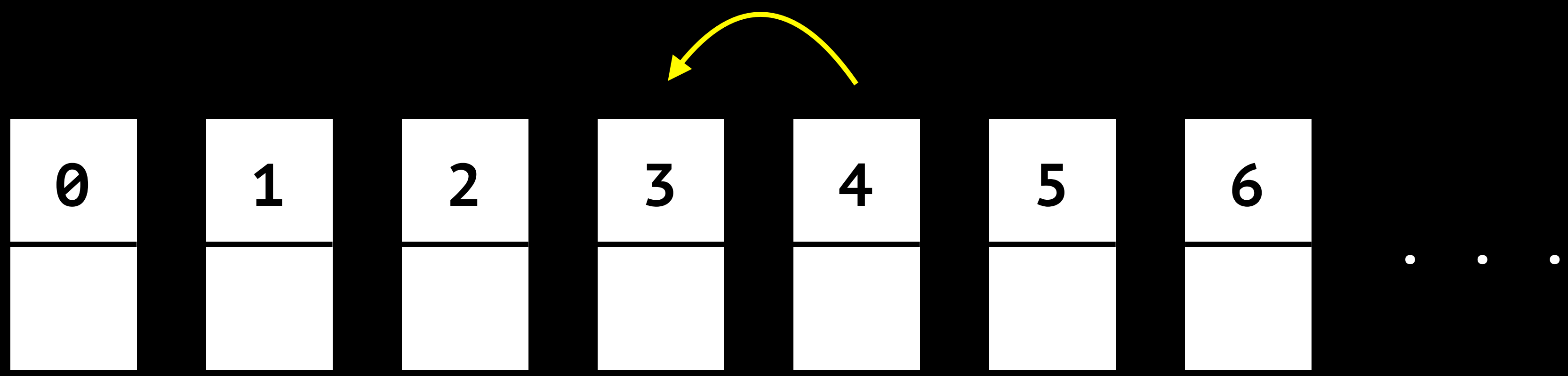


Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```

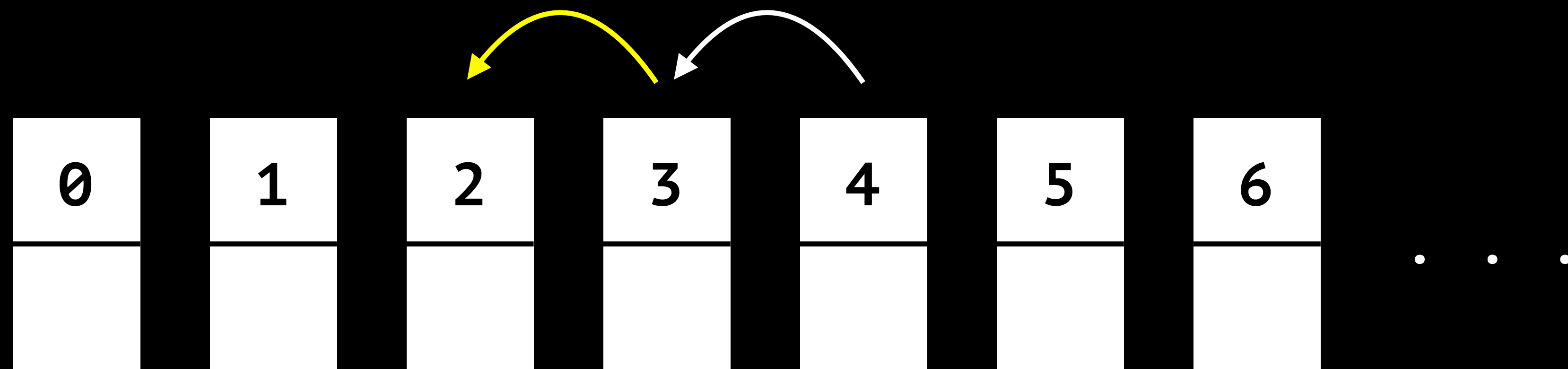
factorial(4) = 4 * factorial(3)



Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

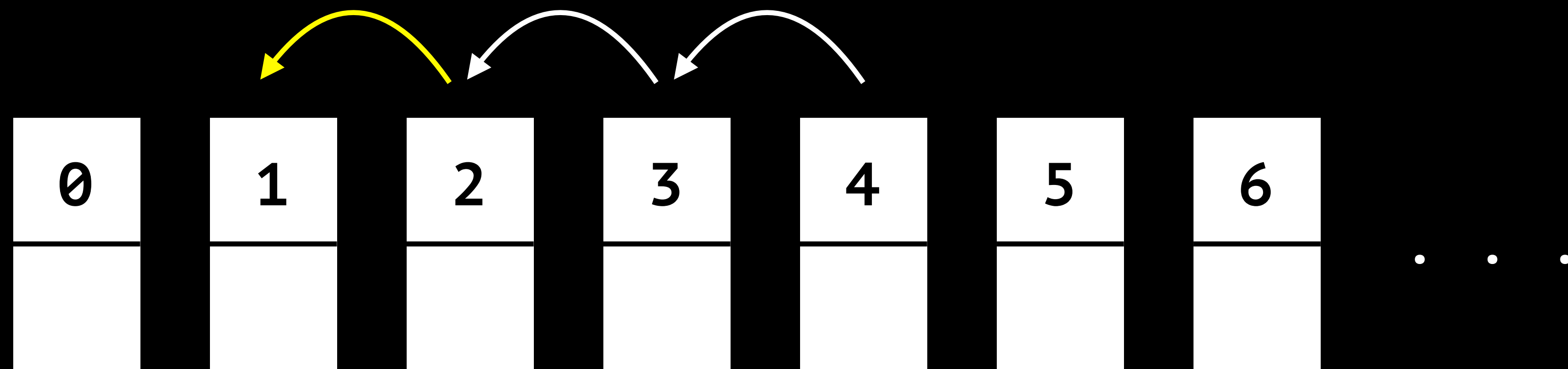
```
1   def factorial(n):
2       if n == 0:
3           return 1
4       return n * factorial(n - 1)
```



Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

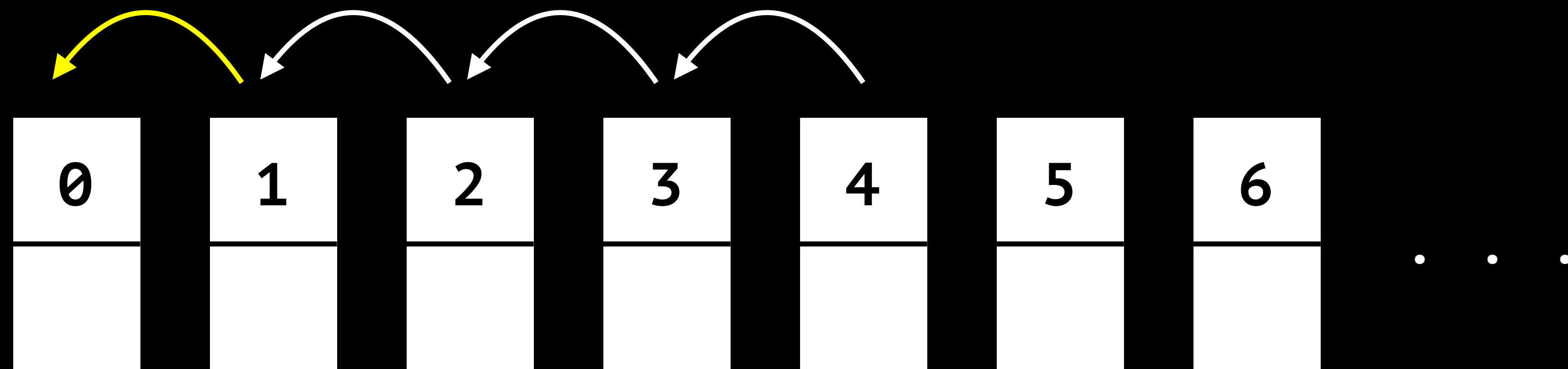
```
1  def factorial(n):
2      if n == 0:
3          return 1
4      return n * factorial(n - 1)
```



Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

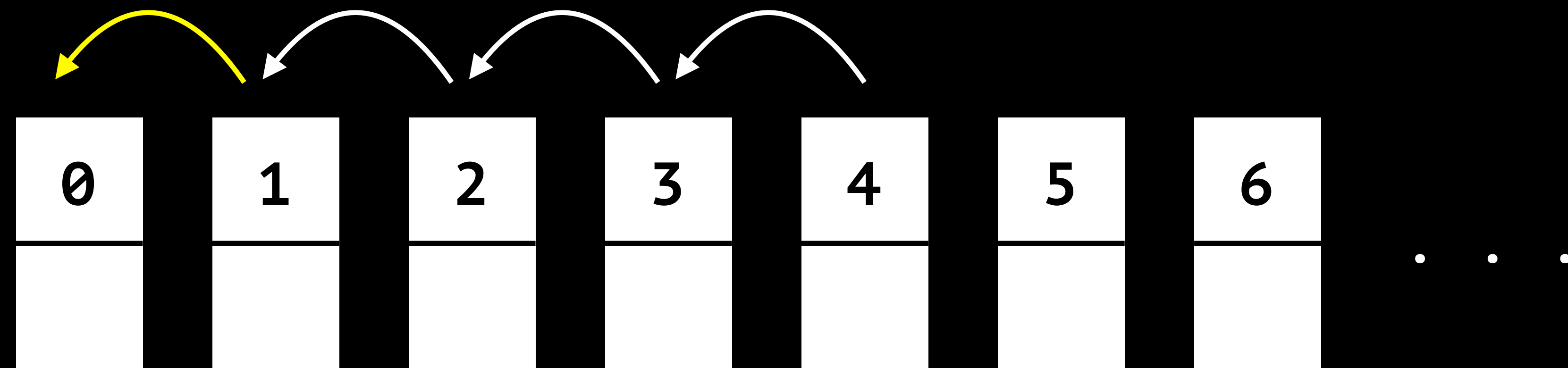
```
1   def factorial(n):
2       if n == 0:
3           return 1
4       return n * factorial(n - 1)
```



Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

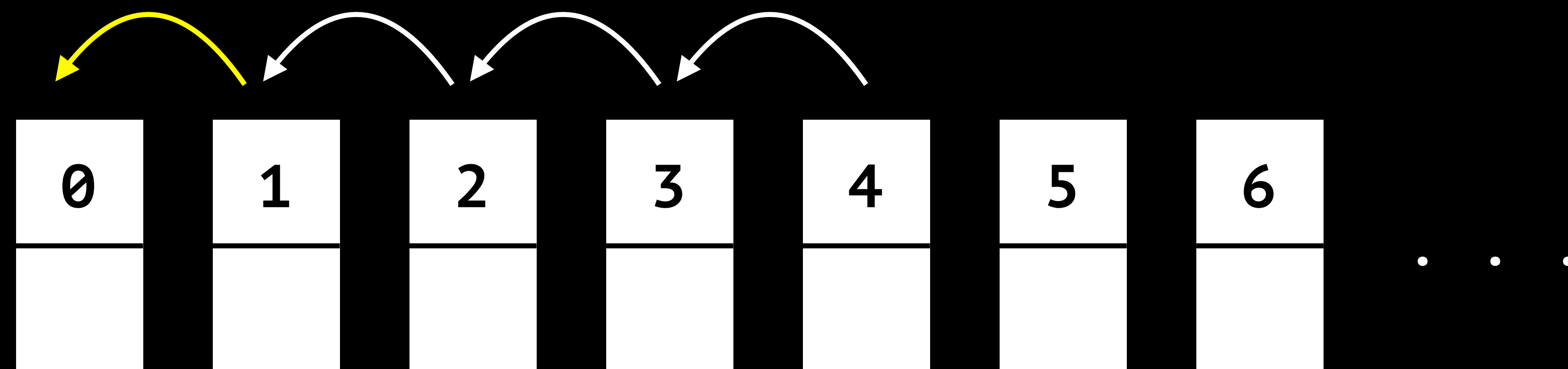
```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```



Dominoes again. Remember induction? Well think of this as induction - but backwards.

Recursion: dominoes

```
1   def factorial(n):
2       if n == 0:
3           return 1
4       return n * factorial(n - 1)
```



Dominoes again. Remember induction? Well think of this as induction - but backwards.

When you're writing code, assume the recursive call works!

Approaching Recursion

hard functions are hard to write.

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Write triangular.
This computes the nth triangular number, which is 0 + 1 + 2 + … + n

```
1    def triangular(n):
2        if ______:
3            return _______
4        return ______________________
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?

Write triangular.
This computes the nth triangular number, which is 0 + 1 + 2 + ... + n

```
1  def triangular(n):
2      if ______:
3          return _______
4      return ______________________
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?

Write triangular.
This computes the nth triangular number, which is 0 + 1 + 2 + … + n

```
1 def triangular(n):
2     if ______:
3         return ________
4     return ___triangular(n - 1)
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?
2) Build on the recursive call - what do I have to add to the result of the recursive call to get the answer to our overall problem

Write `triangular`.
This computes the nth triangular number, which is `0 + 1 + 2 + … + n`

```
1  def triangular(n):
2      if ______:
3          return _______
4      return ___triangular(n - 1)
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?
2) Build on the recursive call - what do I have to add to the result of the recursive call to get the answer to our overall problem

Write `triangular`. This computes the nth triangular number, which is `0 + 1 + 2 + … + n`

```
1 def triangular(n):
2     if ______:
3         return _______
4     return n + triangular(n - 1)
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?
2) Build on the recursive call - what do I have to add to the result of the recursive call to get the answer to our overall problem
3) Base case - when do I stop… i.e. what is the smallest case

Write triangular.
This computes the nth triangular number, which is 0 + 1 + 2 + … + n

```
1  def triangular(n):
2      if ______:
3          return _______
4      return n + triangular(n - 1)
```

Approaching Recursion

hard functions are hard to write.

the first step is often the most difficult

Steps:

1) Recursive call(s) - what is a convenient subproblem that our overall problem shrinks to?
2) Build on the recursive call - what do I have to add to the result of the recursive call to get the answer to our overall problem
3) Base case - when do I stop… i.e. what is the smallest case

Write `triangular`. This computes the nth triangular number, which is `0 + 1 + 2 + … + n`

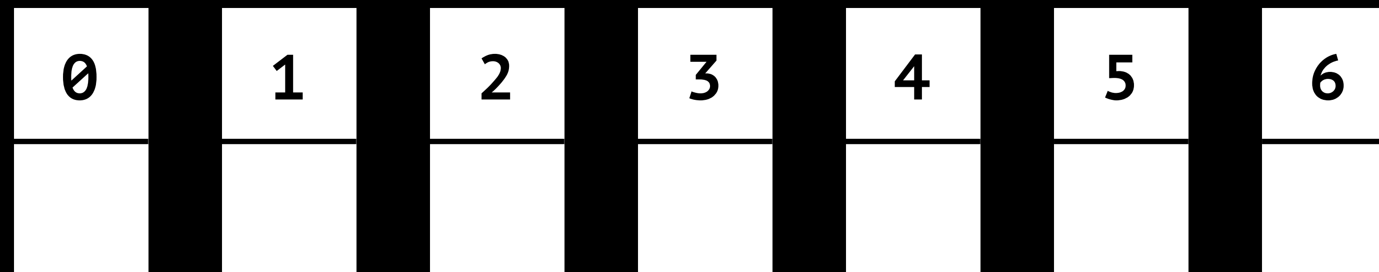
```
1  def triangular(n):
2      if n == 0:
3          return 0
4      return n + triangular(n - 1)
```

Recursion, illustrated

```
1   def triangular(n):
2       if n == 0:
3           return 0
4       return n + triangular(n - 1)
```

triangular(4) = 4 + triangular(3)

Remember this?



Recursion, illustrated

```python
def triangular(n):
    if n == 0:
        return 0
    return n + triangular(n - 1)
```

How else can we draw this?

Recursion, illustrated

```
1  def triangular(n):
2      if n == 0:
3          return 0
4      return n + triangular(n - 1)
```

How else can we draw this?

triangular(4)

Recursion, illustrated

```
1  def triangular(n):
2      if n == 0:
3          return 0
4      return n + triangular(n - 1)
```

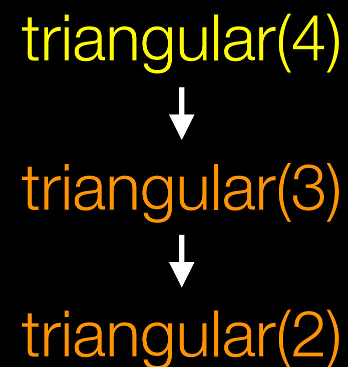
How else can we draw this?

triangular(4)
↓
triangular(3)

Recursion, illustrated

```
1  def triangular(n):
2      if n == 0:
3          return 0
4      return n + triangular(n - 1)
```

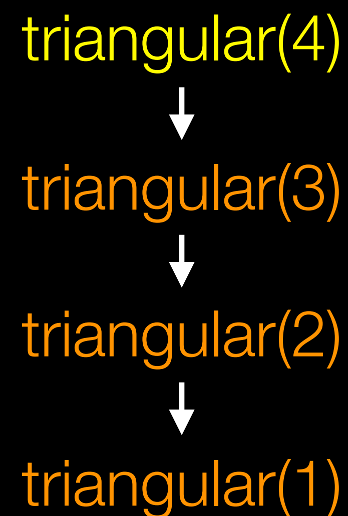
How else can we draw this?



Recursion, illustrated

```
1    def triangular(n):
2        if n == 0:
3            return 0
4        return n + triangular(n - 1)
```

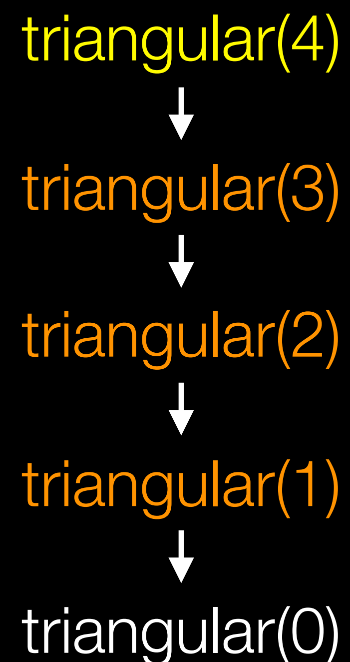
How else can we draw this?



Recursion, illustrated

```
1  def triangular(n):
2      if n == 0:
3          return 0
4      return n + triangular(n - 1)
```

How else can we draw this?



Tree Recursion, illustrated

```
1 def fib(n):
2     if n == 0:
3         return 0
4     if n == 1:
5         return 1
6     return fib(n - 1) + fib(n - 2)
```

How do we draw this function?

Tree Recursion, illustrated

```
1  def fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      return fib(n - 1) + fib(n - 2)
```

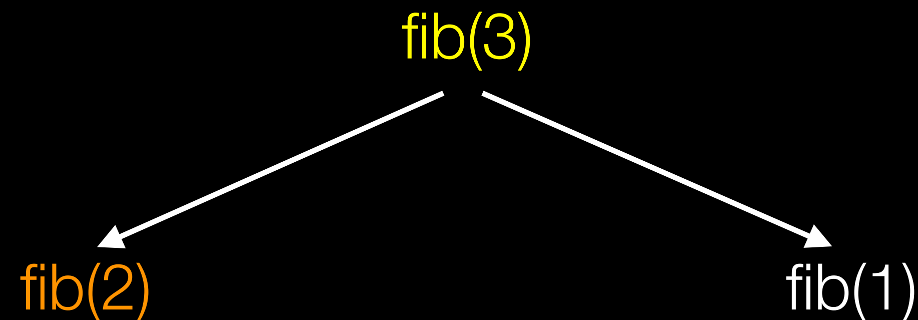
How do we draw this function?

fib(3)

Tree Recursion, illustrated

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

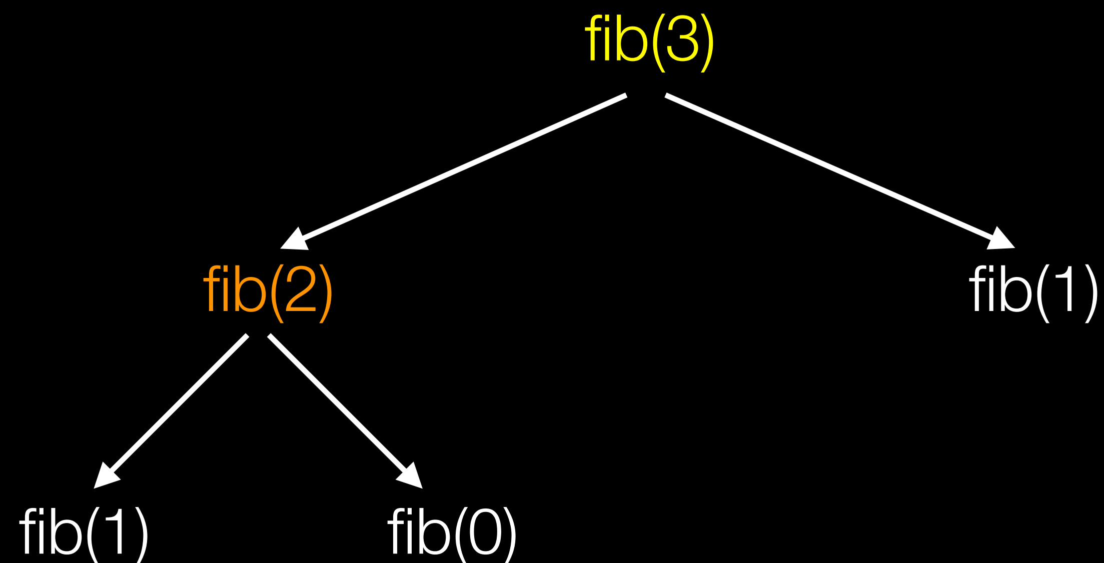
How do we draw this function?



Tree Recursion, illustrated

```
1  def fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      return fib(n - 1) + fib(n - 2)
```

How do we draw this function?



Tree recursion is defined by branching - multiple recursive calls in a frame!

Tree Recursion, illustrated

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

How do we draw this function?

Tree Recursion, illustrated

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

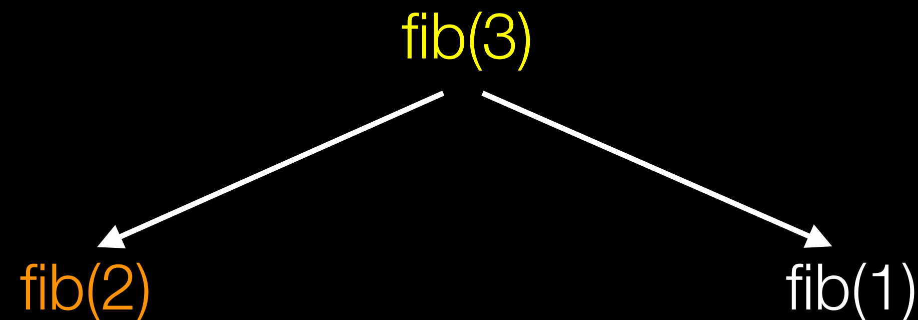
How do we draw this function?

fib(3)

Tree Recursion, illustrated

```
1 def fib(n):
2     if n == 0:
3         return 0
4     if n == 1:
5         return 1
6     return fib(n - 1) + fib(n - 2)
```

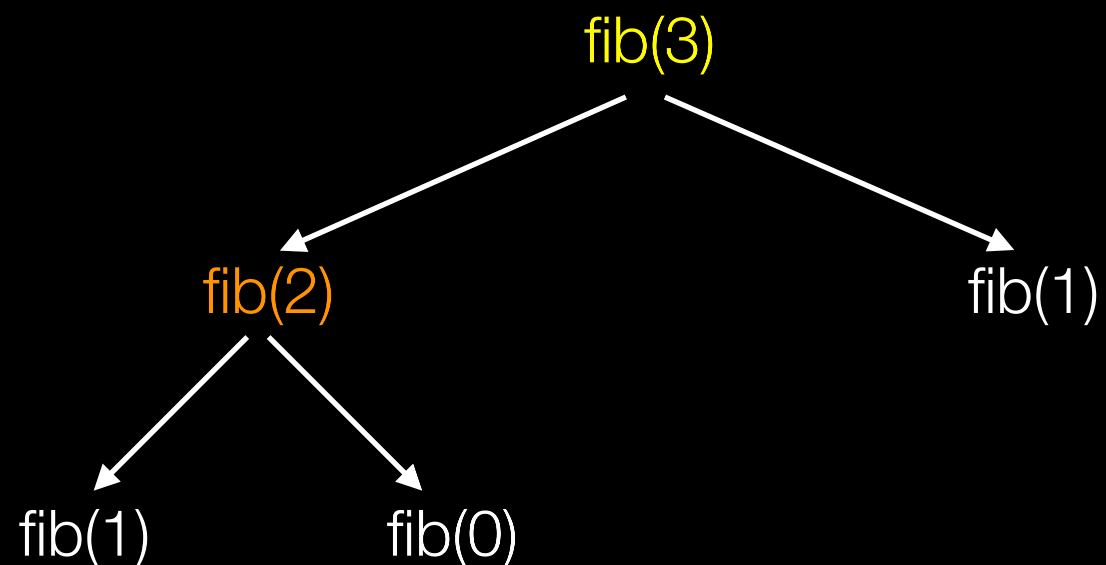
How do we draw this function?



Tree Recursion, illustrated

```
1  def fib(n):
2      if n == 0:
3          return 0
4      if n == 1:
5          return 1
6      return fib(n - 1) + fib(n - 2)
```

How do we draw this function?



Tree recursion is defined by branching - multiple recursive calls in a frame!

What's the difference?

```python
def triangular(n):
    if n == 0:
        return 0
    return n + triangular(n - 1)
```

Recursion

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Tree Recursion

What's the difference?

```python
def triangular(n):
    if n == 0:
        return 0
    return n + triangular(n - 1)
```

Recursion

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Tree Recursion

- more recursive calls

What's the difference?

```python
def triangular(n):
    if n == 0:
        return 0
    return n + triangular(n - 1)
```

Recursion

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Tree Recursion

- more recursive calls
- more base cases

What's the difference?

```python
def triangular(n):
    if n == 0:
        return 0
    return n + triangular(n - 1)
```

Recursion

```python
def fib(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fib(n - 1) + fib(n - 2)
```

Tree Recursion

- more recursive calls
- more base cases

But fundamentally this is the same! We're still dealing with dominoes, except sometimes branching.

Thanks for coming.

Have a great rest of your week! :)

Attendance: links.cs61a.org/albert-disc
Slides: albertxu.xyz/teaching/cs61a/