Announcements

☐ Ants project due tonight

☐ HW8 due Wednesday at 7pm

☐ Midterm 2 Thursday at 7pm
  ☐ See course website for more information
A tuple can contain another tuple as an element.

Pairs are sufficient to represent sequences.

Recursive list representation of the sequence 1, 2, 3, 4:

Recursive lists are recursive: the rest of the list is a list.

Nested pairs (old): \((1, (2, (3, (4, \text{None}))))\)

Rlist class (new): \(\text{Rlist}(1, \text{Rlist}(2, \text{Rlist}(3, \text{Rlist}(4))))\)
Recursive List Class

Methods can be recursive as well!

class Rlist(object):
    class EmptyList(object):
        def __len__(self):
            return 0
    empty = EmptyList()

def __init__(self, first, rest=empty):
    self.first = first
    self.rest = rest

def __len__(self):
    return 1 + len(self.rest)

def __getitem__(self, i):
    if i == 0:
        return self.first
    return self.rest[i - 1]

There's the base case!

Yes, this call is recursive
Recursive Operations on Rlists
Recursive list processing almost always involves a recursive call on the rest of the list.
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
>>> s.rest
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))
```

```python
>>> s.rest
Rlist(2, Rlist(3))
```

```python
>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```
Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3))))))

def extend_rlist(s1, s2):
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))
```

```python
def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
```

Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
given s = Rlist(1, Rlist(2, Rlist(3)))

>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3))))))

def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
```
Recursive Operations on Rlists

Recursive list processing almost always involves a recursive call on the rest of the list.

```python
>>> s = Rlist(1, Rlist(2, Rlist(3)))

>>> s.rest
Rlist(2, Rlist(3))

>>> extend_rlist(s.rest, s)
Rlist(2, Rlist(3, Rlist(1, Rlist(2, Rlist(3)))))

def extend_rlist(s1, s2):
    if s1 is Rlist.empty:
        return s2
    return Rlist(s1.first, extend_rlist(s1.rest, s2))
```
Map and Filter on Rlists
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
```
We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
```
Map and Filter on Rlists

We want operations on a whole list, not an element at a time.

```python
def map_rlist(s, fn):
    if s is Rlist.empty:
        return s
    return Rlist(fn(s.first), map_rlist(s.rest, fn))

def filter_rlist(s, fn):
    if s is Rlist.empty:
        return s
    rest = filter_rlist(s.rest, fn)
    if fn(s.first):
        return Rlist(s.first, rest)
    return rest
```
Tree Structured Data
Tree Structured Data

Nested Sequences are Hierarchical Structures.
Tree Structured Data

Nested Sequences are Hierarchical Structures.

((1, 2), (3, 4), 5)
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$$((1, 2), (3, 4), 5)$$
Tree Structured Data

Nested Sequences are Hierarchical Structures.

\[((1, 2), (3, 4), 5)\]
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$((1,\ 2),\ (3,\ 4),\ 5)$
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$$((1, 2), (3, 4), 5)$$
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$$((1, 2), (3, 4), 5)$$
Tree Structured Data

Nested Sequences are Hierarchical Structures.

$$((1, 2), (3, 4), 5)$$
Tree Structured Data

Nested Sequences are Hierarchical Structures.

\[
((1, 2), (3, 4), 5)
\]
Tree Structured Data

Nested Sequences are Hierarchical Structures.

\(((1, 2), (3, 4), 5)\)
Tree Structured Data

Nested Sequences are Hierarchical Structures.

\[((1, 2), (3, 4), 5)\]
Nested Sequences are Hierarchical Structures.

\(((1, 2), (3, 4), 5)\)
Tree Structured Data

Nested Sequences are Hierarchical Structures.

\(((1, 2), (3, 4), 5)\)

In every tree, a vast forest
Nested Sequences are Hierarchical Structures.

$$\left( \left( 1, 2 \right), \left( 3, 4 \right), 5 \right)$$

In every tree, a vast forest

Example: [http://goo.gl/0h6n5](http://goo.gl/0h6n5)
Recursive Tree Processing
Recursive Tree Processing

Tree operations typically make recursive calls on branches
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))
```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
```

```python```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
Recursive Tree Processing

Tree operations typically make recursive calls on branches

def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn)
```
Recursive Tree Processing

Tree operations typically make recursive calls on branches

```python
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))

def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn) for branch in tree)
```
Trees with Internal Node Values
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

```
  fib(6)
     /   \\    
  fib(4)  fib(5)
   / \
fib(2) fib(3)   fib(3)
  /   \\    / \
fib(1) fib(2) fib(1) fib(2)
 /   \\    / \
0     1  0   1
```

fib(4):
- fib(2): 1
- fib(1): 0
- fib(3): 1

fib(5):
- fib(3): 0
- fib(2): 1
- fib(1): 0
- fib(2): 1
- fib(3): 0

fib(6):
- fib(4): fib(2): 1
- fib(3): 0
- fib(2): 1
- fib(1): 0
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

def count_factors(n):

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
```

Time (remainders)
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

The Consumption of Time
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

Time (remainders)
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        if k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
return factors
```

**Time (remainders)**
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        if k * k == n:
            factors += 1
        factors += 2
    k += 1
return factors
```

Time (remainders)

\[ n \quad \text{and} \quad \left\lfloor \sqrt{n} \right\rfloor \]
Order of Growth
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)

\[
R(n) = \Theta(f(n))
\]
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)

\[ R(n) = \Theta(f(n)) \]

means that there are positive constants \( k_1 \) and \( k_2 \) such that
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)

\[ R(n) = \Theta(f(n)) \]

means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]
Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

\( n \): size of the problem

\( R(n) \): Measurement of some resource used (time or space)

\[ R(n) = \Theta(f(n)) \]

means that there are positive constants \( k_1 \) and \( k_2 \) such that

\[ k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n) \]

for sufficiently large values of \( n \).
Constant Time: $\Theta(1)$
Constant Time: $\Theta(1)$

Time does not depend on input size.
Constant Time: $\Theta(1)$

Time does **not** depend on input size.

```python
def g(n):
    return 42
```
Time does not depend on input size.

```python
def g(n):
    return 42

def foo(n):
    baz = 7
    if n > 5:
        baz += 5
    return baz
```
Time does not depend on input size.

```
def g(n):
    return 42

def foo(n):
    baz = 7
    if n > 5:
        baz += 5
    return baz

def is_even(n):
    return n % 2 == 0
```
Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
```
Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

\[
\begin{array}{c}
\text{Time} \\
\Theta(n)
\end{array}
\]

def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

<table>
<thead>
<tr>
<th></th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n))</td>
<td>(\Theta(\phi^n))</td>
</tr>
</tbody>
</table>

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
```
Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr
```

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

Next time, we will see how to make recursive version faster.
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        factors += 2
        k += 1
    if k * k == n:
        factors += 1
return factors
```
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

```
sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        if k * k == n:
            factors += 1
        k += 1
    factors += 2
return factors
```

\[
\Theta(n)
\]
The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        factors += 2
        k += 1
    if k * k == n:
        factors += 1
return factors
```

<table>
<thead>
<tr>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td>$\Theta(\sqrt{n})$</td>
</tr>
</tbody>
</table>
Exponentiation
Exponentiation

**Goal:** one more multiplication lets us double the problem size.
Goal: one more multiplication lets us double the problem size.

def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise} 
\end{cases} \]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{1/2})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

```python
def square(x):
\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
(b^{\frac{1}{2}})^2 & \text{if } n \text{ is even} \\
b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
```
**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x
```

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise}
\end{cases} \]

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{\frac{1}{2}})^{2n} & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases} \]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):

\[
\begin{align*}
    b^n &= \begin{cases}
        1 & \text{if } n = 0 \\
        b \cdot b^{n-1} & \text{otherwise}
    \end{cases} \\
    b^n &= \begin{cases}
        1 & \text{if } n = 0 \\
        (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\
        b \cdot b^{n-1} & \text{if } n \text{ is odd}
    \end{cases}
\end{align*}
\]
```
Exponentiation

Goal: one more multiplication lets us double the problem size.

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    if n % 2 == 0:
        return (b**(1/2))**2
    else:
        return b * exp(b, n - 1)
```
Exponentiation

Goal: one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    if n is even:
        return (b^(1/2))^2
    return b * b^(n-1)
```

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise} 
\end{cases} \]

\[ b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{1/2})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd} 
\end{cases} \]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

\[
\begin{align*}
\text{def } \exp(b, n) : & \\
\text{if } n == 0 : & \quad \text{return 1} \\
\text{return } b \times \exp(b, n - 1) & \\

\text{def } \text{square}(x) : & \\
\text{return } x \times x & \\

\text{def } \text{fast}\_\exp(b, n) : & \\
\text{if } n == 0 : & \quad \text{return 1} \\
\text{elif } n \% 2 == 0 : & \quad \text{return } (b^{1/2})^2 \\
\end{align*}
\]

\[
\begin{align*}
b^n &= \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise}
\end{cases} \\

b^n &= \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{1/2})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    return b * exp(b, n - 1)
```

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
(b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\
b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise}
\end{cases}
\]

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{\frac{n}{2}})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 b \cdot b^{n-1} & \text{otherwise} 
\end{cases}
\]

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{1/2})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd} 
\end{cases}
\]

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

```python
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)

def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Exponentiation

**Goal:** one more multiplication lets us double the problem size.

\[
\begin{array}{c|c|c}
\text{def } \text{exp}(b, n): \\
\quad \text{if } n == 0: \\
\quad \quad \text{return } 1 \\
\quad \text{return } b \times \text{exp}(b, n - 1) \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{def } \text{square}(x): \\
\quad \text{return } x \times x \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{def } \text{fast_exp}(b, n): \\
\quad \text{if } n == 0: \\
\quad \quad \text{return } 1 \\
\quad \text{elif } n \% 2 == 0: \\
\quad \quad \text{return } \text{square}(\text{fast_exp}(b, n \div 2)) \\
\quad \text{else:} \\
\quad \quad \text{return } b \times \text{fast_exp}(b, n - 1) \\
\end{array}
\]

\[
\begin{array}{c|c|c}
\text{Time} & \text{Space} \\
\Theta(n) & \Theta(n) \\
\Theta(\log n) & \Theta(\log n)
\end{array}
\]
The Consumption of Space
The Consumption of Space

Which environment frames do we need to keep during evaluation?
The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.
The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames in active environments consume memory.
The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.
The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.

Active environments:

• Environments for any statements currently being executed

• Parent environments of functions named in active environments
The Consumption of Space

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        if k * k == n:
            factors += 1
        factors += 2
        k += 1
    return factors
```
Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        factors += 2
        k += 1
    if k * k == n:
        factors += 1
return factors
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
The Consumption of Space

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```python
def count_factors(n):
    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors

sqrt_n = sqrt(n)
k, factors = 1, 0
while k < sqrt_n:
    if n % k == 0:
        factors += 2
    k += 1
if k * k == n:
    factors += 1
return factors
```

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>

$\Theta(\sqrt{n})$ $\Theta(1)$
Fibonacci Memory Consumption

```
fib(6)
fib(4)
fib(2)  fib(3)
  1  fib(1)  fib(2)
    0 1
```

```
fib(5)
fib(3)
fib(1)  fib(2)
    0 1
```

```
fib(4)
fib(2)  fib(3)
    1  fib(1)  fib(2)
      0 1
```

```
fib(3)
fib(1)  fib(2)
    0 1
```

```
fib(2)
    0
```

```
fib(1)
    1
```

```
fib(0)
    0
```

```
fib(1)
    1
```

```
fib(2)
    1
```

```
fib(3)
    1
```

```
fib(4)
    1
```

```
fib(5)
    1
```

```
fib(6)
    1```
Fibonacci Memory Consumption

Assume we have reached this step
Fibonacci Memory Consumption

Assume we have reached this step.
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Can be reclaimed
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Can be reclaimed
Hasn't yet been created
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
</table>

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
```
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td></td>
</tr>
</tbody>
</table>

def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

\[
\begin{array}{c|c|c}
\text{Time} & \text{Space} \\
\hline
\Theta(n) & \Theta(1) \\
\end{array}
\]

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
```
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

<table>
<thead>
<tr>
<th>Time</th>
<th>Space</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Theta(n)$</td>
<td>$\Theta(1)$</td>
</tr>
<tr>
<td>$\Theta(\phi^n)$</td>
<td>$\Theta(n)$</td>
</tr>
</tbody>
</table>

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

def fib(n):
    if n == 1:
        return 0
    if n == 2:
        return 1
    return fib(n - 2) + fib(n - 1)
```
Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

\[
\begin{array}{c|c|c}
\text{Time} & \Theta(n) & \Theta(n) \\
\hline
\text{Space} & \Theta(1) & \Theta(n) \\
\end{array}
\]

```python
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr

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

Next time, we will see how to make recursive version faster.
Comparing Orders of Growth \( (n \text{ is problem size}) \)
Comparing Orders of Growth \((n \text{ is problem size})\)

\[\Theta(b^n)\]
Comparing Orders of Growth \((n \text{ is problem size})\)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes 
\[ \Theta(\phi^n) \] steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Comparing Orders of Growth \((n\text{ is problem size})\)

\(\Theta(b^n)\)  Exponential growth! Recursive fib takes
\(\Theta(\phi^n)\) steps, where  \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\)

Incrementing the problem scales \(R(n)\) by a factor.
Comparing Orders of Growth \((n \text{ is problem size})\)

\(\Theta(b^n)\) \hspace{1cm} \text{Exponential growth! Recursive } \text{fib takes }
\[\Theta(\phi^n)\text{ steps, where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\]

Incrementing the problem scales \(R(n)\) by a factor.

\(\Theta(n^2)\)
Comparing Orders of Growth \( (n \text{ is problem size}) \)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes \( \Theta(\phi^n) \) steps, where \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)

Incrementing the problem scales \( R(n) \) by a factor.

\[ \Theta(n^2) \] Quadratic growth. E.g., operations on all pairs.
Comparing Orders of Growth \((n \text{ is problem size})\)

\[\Theta(b^n)\] Exponential growth! Recursive fib takes \(\Theta(\phi^n)\) steps, where \[\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\]

Incrementing the problem scales \(R(n)\) by a factor.

\[\Theta(n^2)\] Quadratic growth. E.g., operations on all pairs.

Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).
Comparing Orders of Growth \((n\) is problem size\)

\[ \Theta(b^n) \quad \text{Exponential growth! Recursive fib takes} \]

\[ \Theta(\phi^n) \quad \text{steps, where} \quad \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \]

Incrementing the problem scales \(R(n)\) by a factor.

\[ \Theta(n^2) \quad \text{Quadratic growth. E.g., operations on all pairs.} \]

Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).

\[ \Theta(n) \]
Comparing Orders of Growth (n is problem size)

\( \Theta(b^n) \)  \hspace{1em} \text{Exponential growth! Recursive fib takes}  
\( \Theta(\phi^n) \) steps, where  \( \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828 \)
Incrementing the problem scales \( R(n) \) by a factor.

\( \Theta(n^2) \)  \hspace{1em} \text{Quadratic growth. E.g., operations on all pairs.}  
Incrementing \( n \) increases \( R(n) \) by the problem size \( n \).

\( \Theta(n) \)  \hspace{1em} \text{Linear growth. Resources scale with the problem.}
Comparing Orders of Growth \((n \text{ is problem size})\)

\(\Theta(b^n)\) Exponential growth! Recursive `fib` takes \(\Theta(\phi^n)\) steps, where \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\)

Incrementing the problem scales \(R(n)\) by a factor.

\(\Theta(n^2)\) Quadratic growth. E.g., operations on all pairs.

Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).

\(\Theta(n)\) Linear growth. Resources scale with the problem.

\(\Theta(\log n)\)
Comparing Orders of Growth \((n\) is problem size)\\

\(\Theta(b^n)\) Exponential growth! Recursive fib takes \\
\(\Theta(\phi^n)\) steps, where \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\) \\
Incrementing the problem scales \(R(n)\) by a factor.\\

\(\Theta(n^2)\) Quadratic growth. E.g., operations on all pairs. \\
Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).\\

\(\Theta(n)\) Linear growth. Resources scale with the problem.\\

\(\Theta(\log n)\) Logarithmic growth. These processes scale well.
Comparing Orders of Growth \((n \text{ is problem size})\)

\[ \Theta(b^n) \] Exponential growth! Recursive fib takes \(\Theta(\phi^n)\) steps, where \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\)

Incrementing the problem scales \(R(n)\) by a factor.

\[ \Theta(n^2) \] Quadratic growth. E.g., operations on all pairs.

Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).

\[ \Theta(n) \] Linear growth. Resources scale with the problem.

\[ \Theta(\log n) \] Logarithmic growth. These processes scale well.

Doubling the problem only increments \(R(n)\).
Comparing Orders of Growth \((n \text{ is problem size})\)

<table>
<thead>
<tr>
<th>(\Theta(b^n))</th>
<th>Exponential growth! Recursive fib takes (\Theta(\phi^n)) steps, where (\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828) Incrementing the problem scales (R(n)) by a factor.</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\Theta(n^2))</td>
<td>Quadratic growth. E.g., operations on all pairs. Incrementing (n) increases (R(n)) by the problem size (n).</td>
</tr>
<tr>
<td>(\Theta(n))</td>
<td>Linear growth. Resources scale with the problem.</td>
</tr>
<tr>
<td>(\Theta(\log n))</td>
<td>Logarithmic growth. These processes scale well. Doubling the problem only increments (R(n)).</td>
</tr>
<tr>
<td>(\Theta(1))</td>
<td></td>
</tr>
</tbody>
</table>
Comparing Orders of Growth ($n$ is problem size)

$\Theta(b^n)$  Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales $R(n)$ by a factor.

$\Theta(n^2)$  Quadratic growth. E.g., operations on all pairs.

Incrementing $n$ increases $R(n)$ by the problem size $n$.

$\Theta(n)$  Linear growth. Resources scale with the problem.

$\Theta(\log n)$  Logarithmic growth. These processes scale well.

Doubling the problem only increments $R(n)$.

$\Theta(1)$  Constant. The problem size doesn't matter.
Comparing Orders of Growth \((n\) is problem size\)

- \(\Theta(b^n)\): Exponential growth! Recursive fib takes \(\Theta(\phi^n)\) steps, where \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\). Incrementing the problem scales \(R(n)\) by a factor.

- \(\Theta(n^2)\): Quadratic growth. E.g., operations on all pairs. Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).

- \(\Theta(n)\): Linear growth. Resources scale with the problem.

- \(\Theta(\log n)\): Logarithmic growth. These processes scale well. Doubling the problem only increments \(R(n)\).

- \(\Theta(1)\): Constant. The problem size doesn't matter.
Comparing Orders of Growth \((n \text{ is problem size})\)

\[\Theta(\phi^n)\text{ steps, where } \phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\]

**Exponential growth!** Recursive fib takes \(\Theta(\phi^n)\) steps, where \(\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828\).

Incrementing the problem scales \(R(n)\) by a factor.

**Quadratic growth.** E.g., operations on all pairs.

Incrementing \(n\) increases \(R(n)\) by the problem size \(n\).

**Linear growth.** Resources scale with the problem.

**Logarithmic growth.** These processes scale well.

Doubling the problem only increments \(R(n)\).

**Constant.** The problem size doesn't matter.
Comparing Orders of Growth ($n$ is problem size)

- **$\Theta(b^n)$**: Exponential growth! Recursive fib takes $\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$
- **$\Theta(n^6)$**: Incrementing the problem scales $R(n)$ by a factor.
- **$\Theta(n^2)$**: Quadratic growth. E.g., operations on all pairs. Incrementing $n$ increases $R(n)$ by the problem size $n$.
- **$\Theta(n)$**: Linear growth. Resources scale with the problem.
- **$\Theta(\sqrt{n})$**: Logarithmic growth. These processes scale well. Doubling the problem only increments $R(n)$.
- **$\Theta(\log n)$**
- **$\Theta(1)$**: Constant. The problem size doesn't matter.