Space Consumption
Space Consumption

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

Which environment frames do we need to keep during evaluation?

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

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames referenced by active environments are kept.
Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames referenced by active environments are kept.

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

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

Active environments:
Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments. Values and frames referenced by active environments are kept. Memory used for other values and frames can be reclaimed.

Active environments:
• The environment for the current expression being evaluated
Space Consumption

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of active environments.

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

**Active environments:**

- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression
Space Consumption

Which environment frames do we need to keep during evaluation?

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

Values and frames referenced by active environments are kept.

Memory used for other values and frames can be reclaimed.

**Active environments:**

- The environment for the current expression being evaluated
- Environments for calls that depend upon the value of the current expression
- Environments associated with functions referenced by active environments
Fibonacci Environment Diagram

```
fib(3)

if n == 1:
    return 0
if n == 2:
    return 1
return fib(n-2) + fib(n-1)
```
Fibonacci Environment Diagram

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

- **n**: 3
- **fib(3)**
  - if n == 1:
    - return 0
  - if n == 2:
    - return 1
  - return fib(n-2) + fib(n-1)
- **fib(n-2)**
  - if n == 1:
    - return 0
  - if n == 2:
    - return 1
  - return fib(n-2) + fib(n-1)
- **fib(n-1)**
  - if n == 1:
    - return 0
  - if n == 2:
    - return 1
  - return fib(n-2) + fib(n-1)
fib(3)

if n == 1:
    return 0

if n == 2:
    return 1

return fib(n-2) + fib(n-1)
Fibonacci Environment Diagram

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

- `n: 3`
- `fib(3)`
  ```python
  if n == 1:
      return 0
  if n == 2:
      return 1
  return fib(n-2) + fib(n-1)
  ```
- `fib(2)`
  ```python
  if n == 1:
      return 0
  if n == 2:
      return 1
  return fib(n-2) + fib(n-1)
  ```
- `fib(1)`
  ```python
  if n == 1:
      return 0
  if n == 2:
      return 1
  return fib(n-2) + fib(n-1)
  ```
Fibonacci Environment Diagram

fib(3):

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

fib:

fib(n):
...

n: 1
fib

n: 2
fib

n: 3
fib
Fibonacci Memory Consumption

fib(6)
  /   
fib(4)  fib(5)
  /   /   
/ fib(2) fib(3) / fib(3) / fib(4)
  |       |       |       |
1 fib(1) fib(2) fib(1) fib(2) fib(1) fib(2)
Fibonacci Memory Consumption

Assume we have reached this step

Monday, October 17, 2011
Assume we have reached this step
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Fibonacci Memory Consumption

Has an active environment
Can be reclaimed

Assume we have reached this step
Fibonacci Memory Consumption

Assume we have reached this step

Has an active environment
Can be reclaimed
Hasn't yet been created

Monday, October 17, 2011
Active Environments for Returned Functions

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

Monday, October 17, 2011
Active Environments for Returned Functions

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

Monday, October 17, 2011
Active Environments for Returned Functions

```
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```
Active Environments for Returned Functions

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```
Active Environments for Returned Functions

```python
make_adder:
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
```

Monday, October 17, 2011
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)

Associated with an environment
Active Environments for Returned Functions

```python
def make_adder(n):
    def adder(k):
        return k + n
    return adder

add1 = make_adder(1)
def make_adder(n):
    def adder(k):
        return k + n
    return adder

Therefore, all frames in this environment must be kept

Associated with an environment
```
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

\[ n: \text{size of the problem} \]

\[ R(n): \text{Measurement of some resource used (time or space)} \]

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

means that there are 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 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 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 \( n \).
Iteration vs Memoized Tree Recursion

Iterative and memoized 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

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

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Time</strong></td>
<td><strong>Space</strong></td>
</tr>
<tr>
<td>-----</td>
<td>-----</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Monday, October 17, 2011
Iteration vs Memoized Tree Recursion

Iterative and memoized 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

@memo
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>$\Theta(n)$</td>
</tr>
</tbody>
</table>
Iteration vs Memoized Tree Recursion

Iterative and memoized 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

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

Iterative and memoized 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

@memo
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 Memoized Tree Recursion

Iterative and memoized 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

@memo
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>
Comparing orders of growth
Comparing orders of growth

$\Theta(b^n)$
Comparing orders of growth

$\Theta(b^n)$  Exponential growth! Recursive fib takes
Comparing orders of growth

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1 + \sqrt{5}}{2} \approx 1.61828$
Comparing orders of growth

$\Theta(b^n)$ 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

$\Theta(b^n)$  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.

$\Theta(n)$
Comparing orders of growth

\( \Theta(b^n) \)  
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.

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

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

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

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

\[ \Theta(n) \quad \text{Linear growth. Resources scale with the problem.} \]

\[ \Theta(\log n) \]
Comparing orders of growth

$\Theta(b^n)$ 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.

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

$\Theta(\log n)$ Logarithmic growth. These functions scale well.
Comparing orders of growth

$\Theta(b^n)$ 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.

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

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

Doubling the problem increments resources needed.
Comparing orders of growth

\[ \Theta(b^n) \] 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.

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

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

Doubling the problem increments resources needed.

\[ \Theta(1) \]
Comparing orders of growth

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

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

Θ(n) Linear growth. Resources scale with the problem.

Θ(log n) Logarithmic growth. These functions scale well.

Doubling the problem increments resources needed.

Θ(1) Constant. The problem size doesn't matter.
Exponentiation
Exponentiation

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

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

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

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.

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

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

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

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

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

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

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

```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):
```

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

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

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

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

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

```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 square(fast_exp(b, n//2))
```

Monday, October 17, 2011
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 % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

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

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
 (b^{1/2})^2 & \text{if } n \text{ is even} \\
 b \cdot b^{n-1} & \text{if } n \text{ is odd}
\end{cases}
\]
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
    if n % 2 == 0:
        return square(fast_exp(b, n//2))
    else:
        return b * fast_exp(b, n-1)
```

Time | Space
--- | ---

Monday, October 17, 2011
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 % 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>Θ(n)</td>
<td>Θ(n)</td>
</tr>
</tbody>
</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
    if 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>
<tr>
<td>$\Theta(\log n)$</td>
<td>$\Theta(\log n)$</td>
</tr>
</tbody>
</table>