Efficiency
Announcements
Measuring Efficiency
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

fib(5)
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

[Diagram of recursive computation]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[
\text{def } \text{fib}(n):
\begin{align*}
\text{if } n &= 0: \\
& \text{return } 0 \\
\text{elif } n &= 1: \\
& \text{return } 1 \\
\text{else:} \\
& \text{return } \text{fib}(n-2) + \text{fib}(n-1)
\end{align*}
\]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

\[
\text{def } \text{fib}(n):
\]
\[
\quad \text{if } n == 0:
\quad \quad \text{return } 0
\]
\[
\quad \text{elif } n == 1:
\quad \quad \text{return } 1
\]
\[
\quad \text{else:}
\quad \quad \text{return } \text{fib}(n-2) + \text{fib}(n-1)
\]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

![Recursive Computation of the Fibonacci Sequence](http://en.wikipedia.org/wiki/File:Fibonacci.jpg)
Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

Memoization

**Idea:** Remember the results that have been computed before
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
```
Memoization

Idea: Remember the results that have been computed before

```python
def memo(f):
    cache = {}
```
Memoization

Idea: Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            # Put code here to compute f(n)
            cache[n] = compute_result
        return cache[n]

# Example usage
result = memo(lambda n: 2 ** n)(3)
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
```
Memoization

Idea: Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as \( f \), if \( f \) is a pure function
Memoization

**Idea:** Remember the results that have been computed before

```python
def memo(f):
    cache = {}

def memoized(n):
    if n not in cache:
        cache[n] = f(n)
    return cache[n]
return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)
Memoized Tree Recursion

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

fib(4)
fib(2)
fib(0)  fib(1)
0   1

fib(3)
fib(1)  fib(2)
fib(0)  fib(1)
0   1
```
Memoized Tree Recursion

Call to fib

- fib(5)
  - fib(3)
    - fib(1)  
      - fib(0) 1
      - 0 1
  - fib(2)
    - fib(0)  
      - 0 1
    - fib(1)  
      - fib(0) 1
      - 1 fib(1)  
        - fib(0) 1
        - 0 1
Memoized Tree Recursion

Call to fib
Found in cache
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
• Found in cache
○ Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- **Call to fib**
- **Found in cache**
- **Skipped**
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to \texttt{fib}
- \textcolor{red}{Found in cache}
- \textcolor{red}{Skipped}
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

Example:
- fib(5)
- fib(4)
- fib(3)
- fib(2)
- fib(1)
- fib(0)

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

Values:
- fib(5) = 5
- fib(4) = 3
- fib(3) = 2
- fib(2) = 1
- fib(1) = 1
- fib(0) = 0
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

```
Call to fib(5)

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

fib(4)

fib(2)
  fib(0) fib(1)
    0 1

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

fib(2)
  fib(0) fib(1)
    0 1

fib(1)

fib(0)
  fib(1)
    0 1
```
Memoized Tree Recursion

Call to fib
Found in cache
Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped
Memoized Tree Recursion

- **Call to fib**
- **Found in cache**
- **Skipped**

The diagram illustrates the memoized tree recursion for the Fibonacci function. Each call to `fib` is depicted with a blue circle, denoting a call, a red circle for a cached value, and a gray circle for a skipped call. The tree structure shows how repeated calculations are avoided by caching results in a cache tree.
Memoized Tree Recursion

- Call to fib
- Found in cache
- Skipped

Diagram showing recursive calls to fib with caching.
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

fib(5)
- fib(3)
  - fib(1)
    - fib(0)
      - 0
      - 1
    - fib(1)
      - 1
- fib(2)
  - fib(0)
    - 0
    - 1
- fib(4)
  - fib(2)
    - fib(0)
      - 0
      - 1
    - fib(1)
      - 1
  - fib(3)
    - fib(0)
      - 0
      - 1
    - fib(1)
      - 1
Exponentiation
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
    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
    else:
        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
    else:
        return b * exp(b, n-1)
```

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

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

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

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

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

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

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

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

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

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

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

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

**Linear time:**
- Doubling the input **doubles** the time
- $1024x$ the input takes $1024x$ as much time

**Logarithmic time:**
- Doubling the input **increases** the time by a constant $C$
- $1024x$ the input increases the time by only 10 times $C$
Orders of Growth
Quadratic Time

Functions that process all pairs of values in a sequence of length $n$ take quadratic time
Quadratic Time

Functions that process all pairs of values in a sequence of length $n$ take quadratic time.

```python
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```
Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time.

```python
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

5

0

1

0

0
Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time.

```python
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

```
<table>
<thead>
<tr>
<th></th>
<th>3</th>
<th>5</th>
<th>7</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
```

```python
overlap([3, 5, 7, 6], [4, 5, 6, 5])
```
Quadratic Time

Functions that process all pairs of values in a sequence of length n take quadratic time.

```python
def overlap(a, b):
    count = 0
    for item in a:
        for other in b:
            if item == other:
                count += 1
    return count

overlap([3, 5, 7, 6], [4, 5, 6, 5])
```

(Demo)
Exponential Time

Tree-recursive functions can take exponential time

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

Exponential Time

Tree-recursive functions can take exponential time

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

Tree-recursive functions can take exponential time because they involve redundant computations. For example, calculating fib(2) results in a call to fib(0) and fib(1), which themselves are calculated as fib(0) and fib(1). Each subsequent call leads to more calls, creating a tree-like structure of function calls. This results in an exponential number of function calls and computations, making the function inefficient for large inputs.
Exponential Time

Tree-recursive functions can take exponential time

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

Exponential Time

Tree-recursive functions can take exponential time

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

Tree-recursive functions can take exponential time

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

Common Orders of Growth

Exponential growth. E.g., recursive fib

Quadratic growth. E.g., overlap

Linear growth. E.g., slow exp

Logarithmic growth. E.g., exp_fast

Constant growth. Increasing \( n \) doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`

\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing \( n \) doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`

\[ a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`

\[ a \cdot (n+1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing \( n \) doesn't affect time
**Common Orders of Growth**

**Exponential growth.** E.g., recursive `fib`

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`

\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing `n` doesn't affect time
**Common Orders of Growth**

**Exponential growth.** E.g., recursive `fib`  
\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`  
\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`  
\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`  
\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing `n` doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., `slow exp`

\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing `n` doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`
Incrementing $n$ multiplies time by a constant

\[
a \cdot b^{n+1} = (a \cdot b^n) \cdot b
\]

**Quadratic growth.** E.g., `overlap`

\[
a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)
\]

**Linear growth.** E.g., slow `exp`

\[
a \cdot (n + 1) = (a \cdot n) + a
\]

**Logarithmic growth.** E.g., `exp_fast`

\[
a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2
\]

**Constant growth.** Increasing $n$ doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`
Incrementing \( n \) multiplies time by a constant

\[
a \cdot b^{n+1} = (a \cdot b^n) \cdot b
\]

**Quadratic growth.** E.g., `overlap`

\[
a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)
\]

**Linear growth.** E.g., slow `exp`

\[
a \cdot (n + 1) = (a \cdot n) + a
\]

**Logarithmic growth.** E.g., `exp_fast`

\[
a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2
\]

**Constant growth.** Increasing \( n \) doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`
Incrementing $n$ multiplies time by a constant

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`

\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing $n$ doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive fib
Incrementing $n$ multiplies time by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

**Quadratic growth.** E.g., overlap
Incrementing $n$ increases time by $n$ times a constant

$$a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$$

**Linear growth.** E.g., slow exp

$$a \cdot (n + 1) = (a \cdot n) + a$$

**Logarithmic growth.** E.g., exp_fast

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

**Constant growth.** Increasing $n$ doesn't affect time
Common Orders of Growth

**Exponential growth.** E.g., recursive fib
Incrementing $n$ multiplies time by a constant

$$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$$

**Quadratic growth.** E.g., overlap
Incrementing $n$ increases time by $n$ times a constant

$$a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$$

**Linear growth.** E.g., slow exp

$$a \cdot (n + 1) = (a \cdot n) + a$$

**Logarithmic growth.** E.g., exp_fast

$$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$$

**Constant growth.** Increasing $n$ doesn't affect time
Common Orders of Growth

Exponential growth. E.g., recursive fib
Incrementing $n$ multiplies time by a constant

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

Quadratic growth. E.g., overlap
Incrementing $n$ increases time by $n$ times a constant

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

Linear growth. E.g., slow exp

\[ a \cdot (n + 1) = (a \cdot n) + a \]

Logarithmic growth. E.g., exp_fast

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

Constant growth. Increasing $n$ doesn't affect time
Common Orders of Growth

**Exponential growth.**  E.g., recursive `fib`  
Incrementing \( n \) multiplies time by a constant  
\[
a \cdot b^{n+1} = (a \cdot b^n) \cdot b
\]

**Quadratic growth.**  E.g., `overlap`  
Incrementing \( n \) increases time by \( n \) times a constant  
\[
a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)
\]

**Linear growth.**  E.g., slow `exp`  
Incrementing \( n \) increases time by a constant  
\[
a \cdot (n + 1) = (a \cdot n) + a
\]

**Logarithmic growth.**  E.g., `exp_fast`  
\[
a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2
\]

**Constant growth.**  Increasing \( n \) doesn't affect time
### Common Orders of Growth

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Example</th>
<th>Time for Input n+1</th>
<th>Time for n+n</th>
<th>Time for Input n</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponential growth</strong></td>
<td>E.g., recursive <code>fib</code></td>
<td>$a \cdot b^{n+1} = (a \cdot b^n) \cdot b$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing $n$ multiplies time by a constant</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Quadratic growth</strong></td>
<td>E.g., <code>overlap</code></td>
<td>$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing $n$ increases time by $n$ times a constant</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Linear growth</strong></td>
<td>E.g., <code>slow exp</code></td>
<td>$a \cdot (n+1) = (a \cdot n) + a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing $n$ increases time by a constant</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logarithmic growth</strong></td>
<td>E.g., <code>exp_fast</code></td>
<td>$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Constant growth</strong></td>
<td>Increasing $n$ doesn't affect time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Common Orders of Growth

<table>
<thead>
<tr>
<th>Time for n+n</th>
<th>Time for input n+1</th>
<th>Time for input n</th>
</tr>
</thead>
</table>
| Exponential growth. E.g., recursive `fib`  
Incrementing $n$ multiplies time by a constant | $a \cdot b^{n+1} = (a \cdot b^n) \cdot b$ | |
| Quadratic growth. E.g., `overlap`  
Incrementing $n$ increases time by $n$ times a constant | $a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$ | |
| Linear growth. E.g., slow `exp`  
Incrementing $n$ increases time by a constant | $a \cdot (n + 1) = (a \cdot n) + a$ | |
| Logarithmic growth. E.g., `exp_fast`  

$a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2$ | | |
| Constant growth. Increasing $n$ doesn't affect time | | |
### Common Orders of Growth

<table>
<thead>
<tr>
<th>Order of Growth</th>
<th>Example</th>
<th>Time for n+n</th>
<th>Time for input n+1</th>
<th>Time for input n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential growth</td>
<td>E.g., recursive fib</td>
<td>a \cdot b^{n+1} = (a \cdot b^n) \cdot b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quadratic growth</td>
<td>E.g., overlap</td>
<td>a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linear growth</td>
<td>E.g., slow exp</td>
<td>a \cdot (n + 1) = (a \cdot n) + a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Logarithmic growth</td>
<td>E.g., exp_fast</td>
<td>a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant growth</td>
<td>Increasing n doesn't affect time</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Common Orders of Growth

**Exponential growth.** E.g., recursive `fib`
Incrementing $n$ multiplies time by a constant

\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

**Quadratic growth.** E.g., `overlap`
Incrementing $n$ increases time by $n$ times a constant

\[ a \cdot (n + 1)^2 = (a \cdot n^2) + a \cdot (2n + 1) \]

**Linear growth.** E.g., slow `exp`
Incrementing $n$ increases time by a constant

\[ a \cdot (n + 1) = (a \cdot n) + a \]

**Logarithmic growth.** E.g., `exp_fast`
Doubling $n$ only increments time by a constant

\[ a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2 \]

**Constant growth.** Increasing $n$ doesn't affect time
Space and Environments
Space and Environments

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

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

• Environments for any function calls currently being evaluated
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments.

Values and frames in active environments consume memory.

Memory that is used for other values and frames can be recycled.

Active environments:

- Environments for any function calls currently being evaluated.
- Parent environments of functions named in active environments.
Space and Environments

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

Values and frames in active environments consume memory

Memory that is used for other values and frames can be recycled

Active environments:

- Environments for any function calls currently being evaluated
- Parent environments of functions named in active environments

(Demo)
Fibonacci Space Consumption
Fibonacci Space Consumption

fib(5)
Fibonacci Space Consumption

\[ \text{fib}(5) \]

\[ \text{fib}(3) \]
Fibonacci Space Consumption

 fib(5)  
/ | \
|  |
fib(3) fib(4)
Fibonacci Space Consumption

fib(5)
/    \
|     |
fib(3)  fib(4)
/     /
|     |
fib(1)  fib(2)
/     /  
|     |  
1 fib(0) fib(1)
|    |
0    1
Fibonacci Space Consumption

```
    fib(5)
    /   
   /     
  /       
 fib(3)   fib(4)
  / |     /   
 /  |     /     
 fib(1) fib(2) fib(2) fib(3)
  |   |   |   |   |   |   |   |
 1 fib(0) fib(1) fib(0) fib(1) fib(2)
     |       |       |       |
     0       1       0       1
```

fib(2)

fib(1)

fib(0)

fib(1)

fib(0)

fib(1)

fib(0)

fib(1)
Fibonacci Space Consumption

Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step

fib(5)

fib(3)

fib(1)  fib(2)

1  fib(0)  fib(1)

fib(0)  fib(1)

0  1

Has an active environment

Assume we have reached this step

fib(2)

fib(0)  fib(1)

0  1

fib(3)

fib(1)  fib(2)

fib(0)  fib(1)

0  1
Fibonacci Space Consumption

Assume we have reached this step

fib(5)

fib(3)

fib(1) fib(2)

1 fib(0) fib(1)

fib(4)

fib(2)

fib(0) fib(1)

0 1

Has an active environment
Can be reclaimed

Assume we have reached this step
Fibonacci Space Consumption

Assume we have reached this step.

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

```
<table>
<thead>
<tr>
<th>fib(5)</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(3)</td>
</tr>
<tr>
<td>fib(1)     fib(2)</td>
</tr>
<tr>
<td>1   fib(0)     fib(1)</td>
</tr>
<tr>
<td>0   1           0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>fib(0)     fib(1)</td>
</tr>
<tr>
<td>0   1       fib(0)     fib(1)</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>fib(3)</td>
</tr>
<tr>
<td>fib(0)     fib(1)</td>
</tr>
<tr>
<td>0   1       0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>fib(2)</td>
</tr>
<tr>
<td>fib(0)     fib(1)</td>
</tr>
<tr>
<td>0   1       0</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>fib(1)</td>
</tr>
<tr>
<td>1           1       0   1</td>
</tr>
</tbody>
</table>
```