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:

```
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:

```
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:

```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:

```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 showing the recursive computation of 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 showing recursive computation of the Fibonacci sequence]

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)
```

![Diagram of 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:

```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)
```

Memoization
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:
```
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

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
```

(Demo) Keys are arguments that map to return values

Same behavior as f, if f is a pure function
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(1)  fib(2)

fib(3)

fib(1)  fib(2)

1  fib(0)  fib(1)

0  1
Memoized Tree Recursion

Call to fib

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(1)  fib(2)

1  fib(0)  fib(1)

0  1
Memoized Tree Recursion

Call to fib

Found in cache

fib(5)

fib(3)

fib(1)

1

fib(0)

0

fib(2)

fib(1)

1

fib(4)

fib(2)

fib(1)

0

1

fib(3)

fib(1)

1

fib(2)

fib(1)

0

1
Memoized Tree Recursion

- Blue circle: Call to fib
- Red circle: Found in cache
- Circle: Skipped

Diagram showing the recursive calls to the fib function, with some calls marked as found in cache and others marked as 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
Memoized Tree Recursion
Memoized Tree Recursion

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

Call to fib
- Found in cache
- Skipped

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

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

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

Memoized Tree Recursion

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

Call to fib
- Found in cache
- Skipped

```
Call to fib

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)

1  fib(0)  fib(1)

0  1
```
Memoized Tree Recursion

Call to \text{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

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

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

- Call to \text{fib}
- Found in cache
- Skipped
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)
```

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

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

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

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

```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
```

(Demo)
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
- $1024 \times$ the input takes $1024 \times$ as much time

Logarithmic time:
- Doubling the input increases the time by a constant C
- $1024 \times$ 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])
```

```
\begin{array}{cccc}
3 & 5 & 7 & 6 \\
4 & 0 & 0 & 0 & 0 \\
5 & 0 & 1 & 0 & 0 \\
6 & 0 & 0 & 0 & 1 \\
5 & 0 & 1 & 0 & 0 \\
\end{array}
```
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>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
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)
```

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)
```

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)
```

[Image: Fibonacci.jpg]
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

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example Function</th>
<th>Growth Formula</th>
<th>Time for Input $n+1$</th>
<th>Time for Input $n$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponential growth</strong></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>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., overlap</td>
<td>$a \cdot (n+1)^2 = (a \cdot n^2) + a \cdot (2n + 1)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Linear growth</strong></td>
<td>E.g., slow exp</td>
<td>$a \cdot (n+1) = (a \cdot n) + a$</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Logarithmic growth</strong></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><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

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 \texttt{fib}
Incrementing $n$ multiplies time by a constant
\[ a \cdot b^{n+1} = (a \cdot b^n) \cdot b \]

Quadratic growth. E.g., \texttt{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 \texttt{exp}
\[ a \cdot (n + 1) = (a \cdot n) + a \]

Logarithmic growth. E.g., \texttt{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>Time for input n+1</th>
<th>Time for input n</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a \cdot b^{n+1}) = ((a \cdot b^n) \cdot b)</td>
<td></td>
</tr>
<tr>
<td>(a \cdot (n + 1)^2) = ((a \cdot n^2) + a \cdot (2n + 1))</td>
<td></td>
</tr>
<tr>
<td>(a \cdot (n + 1) = (a \cdot n) + a)</td>
<td></td>
</tr>
<tr>
<td>(a \cdot \ln(2 \cdot n) = (a \cdot \ln n) + a \cdot \ln 2)</td>
<td></td>
</tr>
</tbody>
</table>
| Constant growth. Increasing \(n\) doesn't affect time

Exponential growth. E.g., recursive \texttt{fib}
Incrementing \(n\) multiplies time by a constant

Quadratic growth. E.g., \texttt{overlap}
Incrementing \(n\) increases time by \(n\) times a constant

Linear growth. E.g., slow \texttt{exp}
Incrementing \(n\) increases time by a constant

Logarithmic growth. E.g., \texttt{exp\_fast}
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>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>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`
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
Order of Growth Notation
Big Theta and Big O Notation for Orders of Growth

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

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

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

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

**Constant growth.** Increasing \( n \) doesn't affect \( time \)
Big Theta and Big O Notation for Orders of Growth

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

**Quadratic growth.** E.g., `overlap` \( \Theta(n^2) \)
Incrementing \( n \) increases time by \( n \) times a constant

**Linear growth.** E.g., slow `exp` \( \Theta(n) \)
Incrementing \( n \) increases time by a constant

**Logarithmic growth.** E.g., `exp_fast` \( \Theta(\log n) \)
Doubling \( n \) only increments time by a constant

**Constant growth.** Increasing \( n \) doesn't affect time \( \Theta(1) \)
Big Theta and Big O Notation for Orders of Growth

**Exponential growth.** E.g., recursive `fib`  
\[ \Theta(b^n) \quad O(b^n) \]  
Incrementing \( n \) multiplies time by a constant

**Quadratic growth.** E.g., `overlap`  
\[ \Theta(n^2) \quad O(n^2) \]  
Incrementing \( n \) increases time by \( n \) times a constant

**Linear growth.** E.g., slow `exp`  
\[ \Theta(n) \quad O(n) \]  
Incrementing \( n \) increases time by a constant

**Logarithmic growth.** E.g., `exp_fast`  
\[ \Theta(\log n) \quad O(\log n) \]  
Doubling \( n \) only increments time by a constant

**Constant growth.** Increasing \( n \) doesn't affect time  
\[ \Theta(1) \quad O(1) \]
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

```python
fib(5)
```
Fibonacci Space Consumption

fib(5)

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

\[
\begin{align*}
\text{fib}(5) \\
\text{fib}(3) & \quad \text{fib}(4) \\
\text{fib}(1) & \quad \text{fib}(2) & \quad \text{fib}(2) & \quad \text{fib}(3) \\
1 & \quad \text{fib}(0) & \quad \text{fib}(1) & \quad \text{fib}(0) & \quad \text{fib}(1) \quad \text{fib}(1) & \quad \text{fib}(2) \\
0 & \quad 1 & \quad 0 & \quad 1 & \quad 1 & \quad \text{fib}(0) & \quad \text{fib}(1) \\
& & & & & 0 & \quad 1
\end{align*}
\]
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(4)
     /   
    fib(1)  fib(2)
          /   
         fib(0)  fib(1)
```

Has an active environment:

```
fib(2)
   /   
  fib(0)  fib(1)

fib(3)
   /   
  fib(0)  fib(1)
      /   
     fib(0)  fib(1)
```

Assume we have reached this step.
Fibonacci Space Consumption

Assume we have reached this step

Has an active environment
Can be reclaimed
Fibonacci Space Consumption

Assume we have reached this step

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