Efficiency
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)
```

```python
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:

```
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)
```

[Diagram of recursive computation of Fibonacci sequence]

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

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

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

```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:

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

[Diagram showing the recursive computation of the Fibonacci sequence]
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:

\[
\text{fib}(5) \rightarrow \text{fib}(4) \rightarrow \text{fib}(3) \rightarrow \text{fib}(2) \rightarrow \text{fib}(1) \rightarrow \text{fib}(0)
\]

```
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)
```

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)
```

![Diagram of the Fibonacci sequence computation](http://en.wikipedia.org/wiki/File:Fibonacci.jpg)
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
    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
```

- Keys are arguments that map to return values
- Same behavior as f, if f is a pure function

(Demo)
Memoized Tree Recursion
Memoized Tree Recursion

Call to fib

fib(5)

fib(3)

fib(1)

1

fib(0)

0

1

fib(2)

fib(1)

1

fib(0)

0

1

fib(4)

fib(2)

fib(1)

0

1

fib(3)

fib(1)

1

fib(0)

0

1

fib(2)

fib(1)

0

1
Memoized Tree Recursion

Call to fib

Found in cache
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

```
Call to fib
fib(5)
  fib(3)
    fib(1)
      1
    fib(2)
      fib(0)
      0
      fib(1)
      1
  fib(4)
    fib(2)
      fib(0)
      0
      fib(1)
      1
    fib(3)
      fib(1)
      fib(2)
      fib(0)
      fib(1)
      0
      1
```
Memoized Tree Recursion

Call to fib
- Found in cache
- Skipped

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

fib(5) → fib(3) → fib(1) → fib(0) → 0 → fib(1) → 1 → fib(2) → fib(0) → 0 → fib(1) → 1 → fib(3) → fib(1) → fib(2) → fib(0) → 0 → fib(1) → 1 → fib(4) → fib(2) → fib(0) → 0 → fib(1) → 1 → fib(2) → fib(0) → 0 → fib(1) → 1

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

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

- 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

Diagram showing the memoized tree recursion for fib(5), fib(4), and fib(3), with nodes indicating calls to fib, found values in the cache, and skipped computations.
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)
```

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

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

\[
b^n = \begin{cases} 
1 & \text{if } n = 0 \\
(b \cdot b^{n-1})^2 & \text{if } n \text{ is even} \\
(b \cdot b^{n-1}) & \text{if } n \text{ is odd}
\end{cases}
\]
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
```
(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
- $1024x$ the input takes $1024x$ as much time

Logarithmic time:
- Doubling the input **increases** the time by one step
- $1024x$ the input increases the time by only 10 steps
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])
```

```
  3  5  7  6
4  0  0  0  0
5  0  1  0  0
6  0  0  0  1
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])
```

```plaintext
  3  5  7  6
4  0  0  0  0
5  0  1  0  0
6  0  0  0  1

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

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)
```

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 \( \text{fib} \)

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

Quadratic growth. E.g., \( \text{overlap} \)

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

Linear growth. E.g., slow \( \text{exp} \)

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

Logarithmic growth. E.g., \( \text{exp\_fast} \)

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

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

\[
\text{Time for input } n+1 = (a \cdot b^n) \cdot b
\]

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

\[
\text{Time for input } n+1 = (a \cdot n^2) + a \cdot (2n + 1)
\]

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

\[
\text{Time for input } n+1 = (a \cdot n) + a
\]

**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 \textit{time} by a constant

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

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

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

**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 \textit{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

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

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

Quadratic growth. E.g., $\text{overlap}$
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 $\text{exp}$
Incrementing $n$ increases time by a constant

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

Logarithmic growth. E.g., $\text{exp\_fast}$

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

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

**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

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exponential growth</td>
<td>$\text{Recursive fib}$</td>
<td>$\Theta(b^n)$</td>
</tr>
<tr>
<td></td>
<td>Incrementing $n$ multiplies time by a constant</td>
<td></td>
</tr>
<tr>
<td>Quadratic growth</td>
<td>$\text{Overlap}$</td>
<td>$\Theta(n^2)$</td>
</tr>
<tr>
<td></td>
<td>Incrementing $n$ increases time by $n$ times a constant</td>
<td></td>
</tr>
<tr>
<td>Linear growth</td>
<td>$\text{Slow exp}$</td>
<td>$\Theta(n)$</td>
</tr>
<tr>
<td></td>
<td>Incrementing $n$ increases time by a constant</td>
<td></td>
</tr>
<tr>
<td>Logarithmic growth</td>
<td>$\text{Exp_fast}$</td>
<td>$\Theta(\log n)$</td>
</tr>
<tr>
<td></td>
<td>Doubling $n$ only increments time by a constant</td>
<td></td>
</tr>
<tr>
<td>Constant growth</td>
<td>Increasing $n$ doesn't affect time</td>
<td>$\Theta(1)$</td>
</tr>
</tbody>
</table>
### Big Theta and Big O Notation for Orders of Growth

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>( \Theta(b^n) )</th>
<th>( O(b^n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Exponential growth.</strong></td>
<td>E.g., recursive <code>fib</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing ( n ) multiplies <code>time</code> by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>( \Theta(n^2) )</th>
<th>( O(n^2) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Quadratic growth.</strong></td>
<td>E.g., <code>overlap</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing ( n ) increases <code>time</code> by ( n ) times a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>( \Theta(n) )</th>
<th>( O(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Linear growth.</strong></td>
<td>E.g., slow <code>exp</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Incrementing ( n ) increases <code>time</code> by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>( \Theta(\log n) )</th>
<th>( O(\log n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Logarithmic growth.</strong></td>
<td>E.g., <code>exp_fast</code></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Doubling ( n ) only increments <code>time</code> by a constant</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Growth Type</th>
<th>Example</th>
<th>( \Theta(1) )</th>
<th>( O(1) )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Constant growth.</strong></td>
<td>Increasing ( n ) doesn't affect <code>time</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Space
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

fib(5)

fib(3)
Fibonacci Space Consumption

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

\[ \text{fib}(3) \quad \text{fib}(4) \]
Fibonacci Space Consumption

```
fib(5)

  fib(3)

    fib(1)  fib(2)

        1

    fib(0)  fib(1)

        0  1
```
Fibonacci Space Consumption

```
Fibonacci Space Consumption

```

```
19

```

```
fib(5)

```

```
fib(3)

```

```
fib(1)

```

```
fib(0)

```

```
fib(1)

```

```
fib(2)

```

```
fib(0)

```

```
fib(1)

```

```
fib(3)

```

```
fib(2)

```

```
fib(0)

```

```
fib(1)

```

```
fib(2)

```

```
fib(0)

```

```
fib(1)

```

```
fib(1)

```

```
fib(0)

```

```
fib(1)

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```

```
0 1

```
Fibonacci Space Consumption

Assume we have reached this step

```
fib(5)
  └── fib(4)
      └── fib(3)
          └── fib(1)
              1
          └── fib(2)
              └── fib(0)
                  |   0
              └── fib(1)
                  1
```

```
fib(2)
  └── fib(0)
      0
  └── fib(1)
      1
```

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

```
0 1
```

```
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)

Has an active environment
Fibonacci Space Consumption

Assume we have reached this step:

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

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