61A Lecture 20

Monday, March 11

Announcements

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Guerrilla section this weekend on recursive data (linked lists and trees)

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Guerrilla section this weekend on recursive data (linked lists and trees)
- Homework 6 due Monday 3/16 @ 11:59pm

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm–6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Guerrilla section this weekend on recursive data (linked lists and trees)
- Homework 6 due Monday 3/16 @ 11:59pm
- Midterm 2 is on Thursday 3/19 7pm–9pm

Announcements

- Project 3 due Thursday 3/12 @ 11:59pm
  - Project party on Tuesday 3/10 5pm-6:30pm in 2050 VLSB
  - Bonus point for early submission by Wednesday 3/11
- Guerrilla section this weekend on recursive data (linked lists and trees)
- Homework 6 due Monday 3/16 @ 11:59pm
- Midterm 2 is on Thursday 3/19 7pm-9pm
  - Fill out conflict form if you cannot attend due to a course conflict

Time

The Consumption of Time

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Time (number of divisions)

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	Greatest integer less than $\sqrt{n}$

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time (number of divisions)
Slow: Test each k from 1 through n	n
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	Greatest integer less than $\sqrt{n}$

(Demo)

Space

The Consumption of Space

The Consumption of Space

```
Which environment frames do we need to keep during evaluation?
```

The Consumption of Space

Which environment frames do we need to keep during evaluation?

At any moment there is a set of active environments

The Consumption of Space

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

The Consumption of Space

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

The Consumption of Space

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:

The Consumption of Space

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

The Consumption of Space

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

The Consumption of Space

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)

Interactive Diagram

Fibonacci Space Consumption

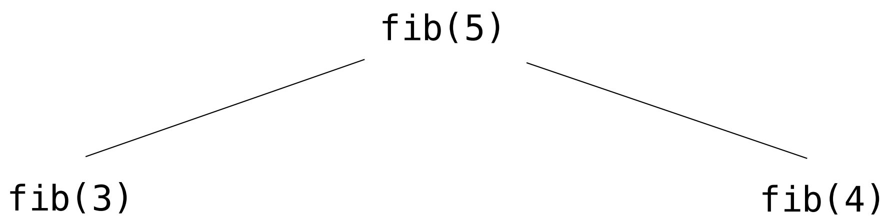
Fibonacci Space Consumption

`fib(5)`

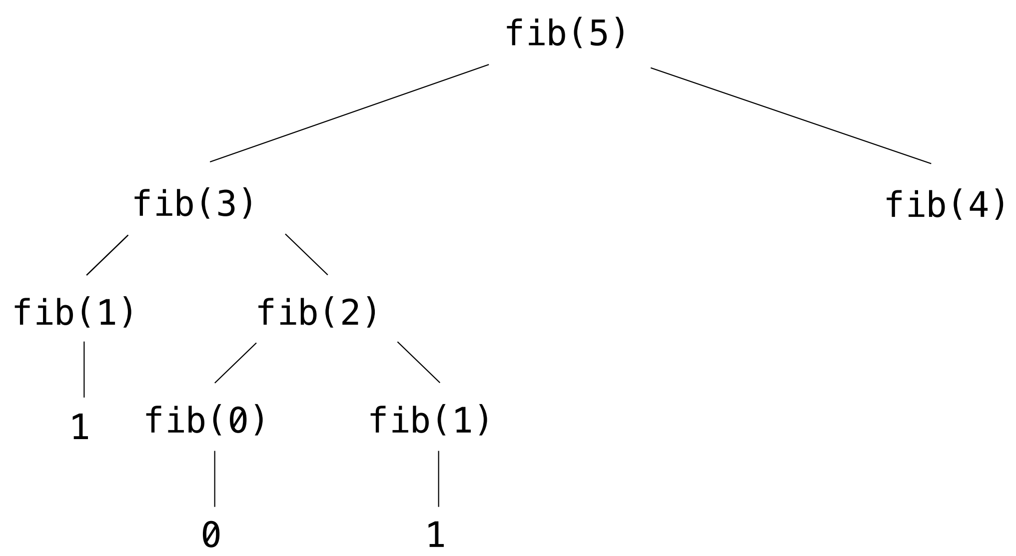
Fibonacci Space Consumption

```
        fib(5)
       /
fib(3)
```

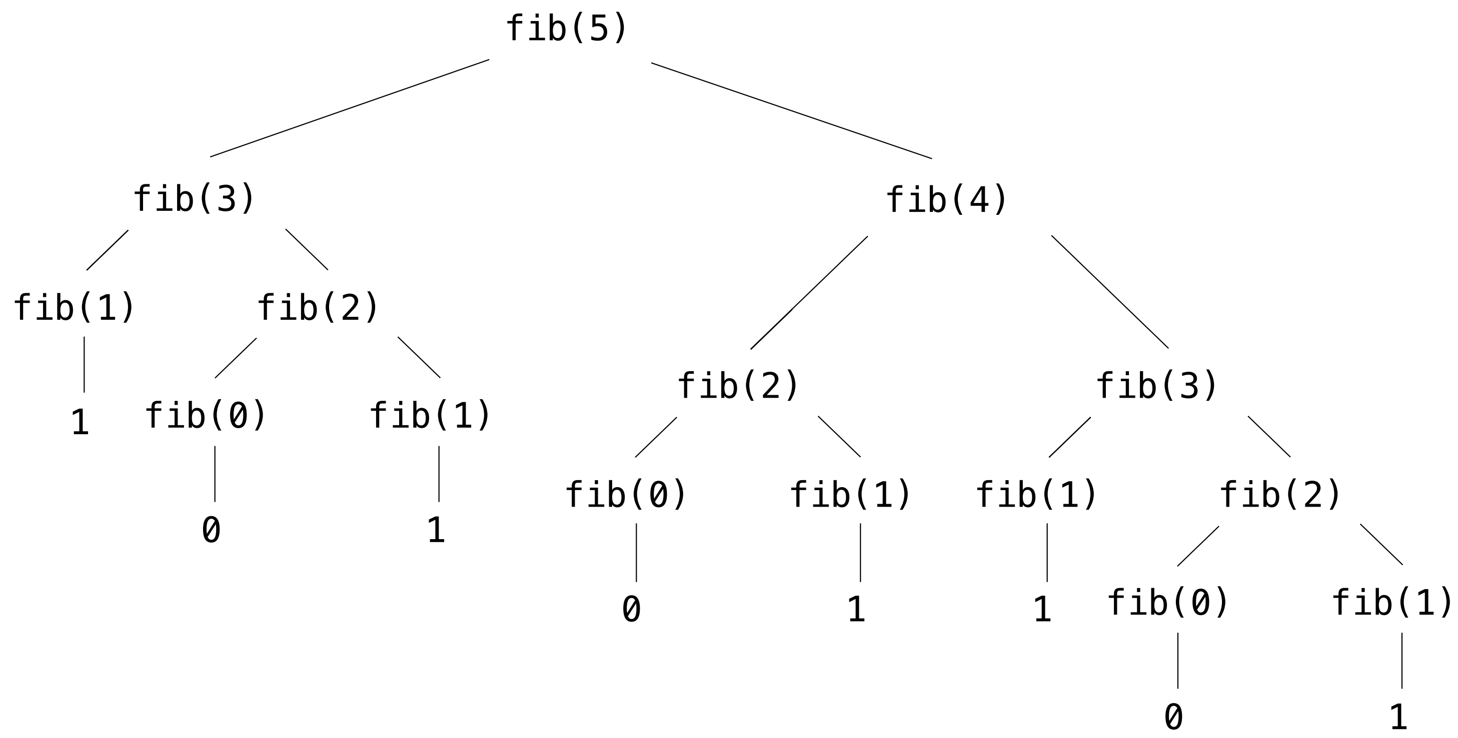
Fibonacci Space Consumption



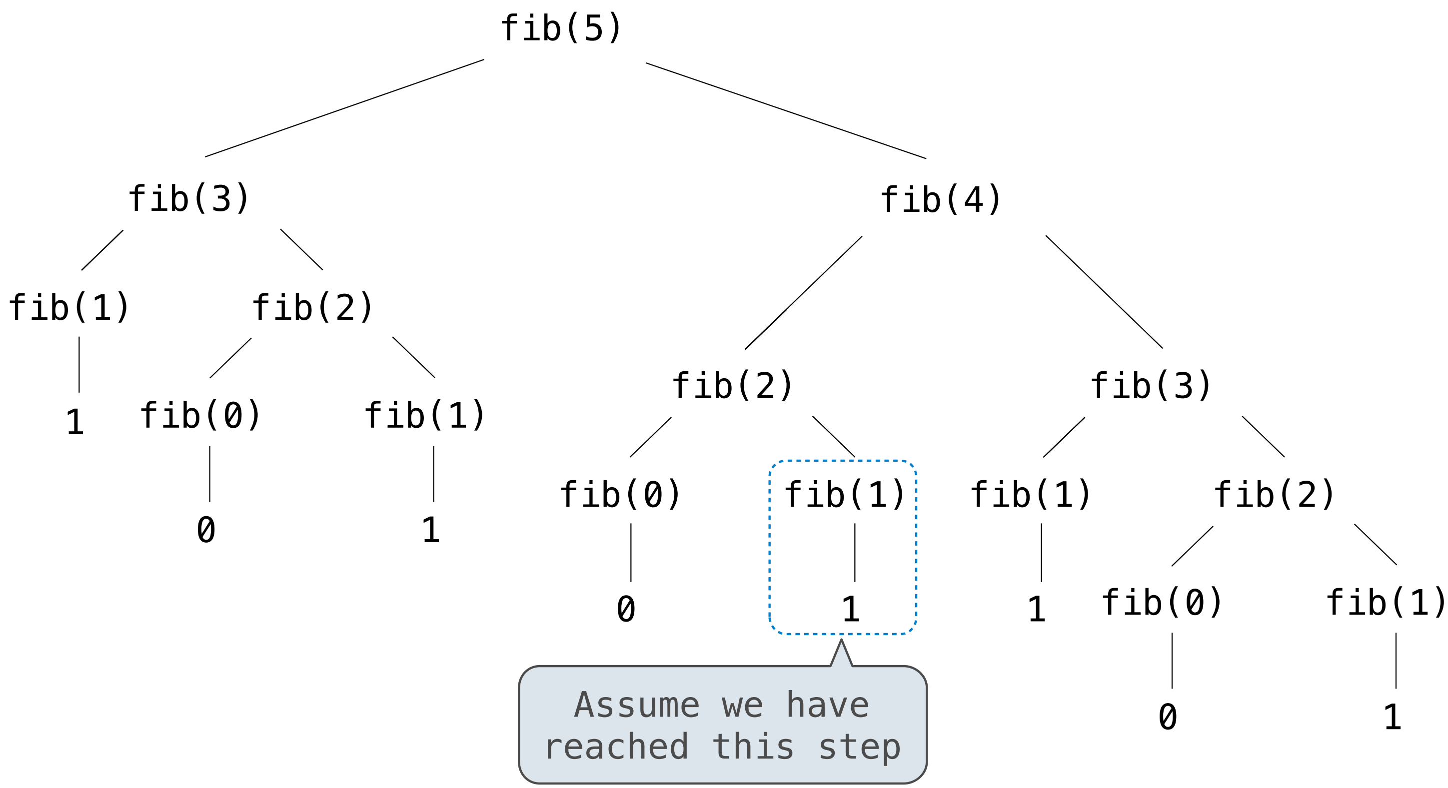
Fibonacci Space Consumption



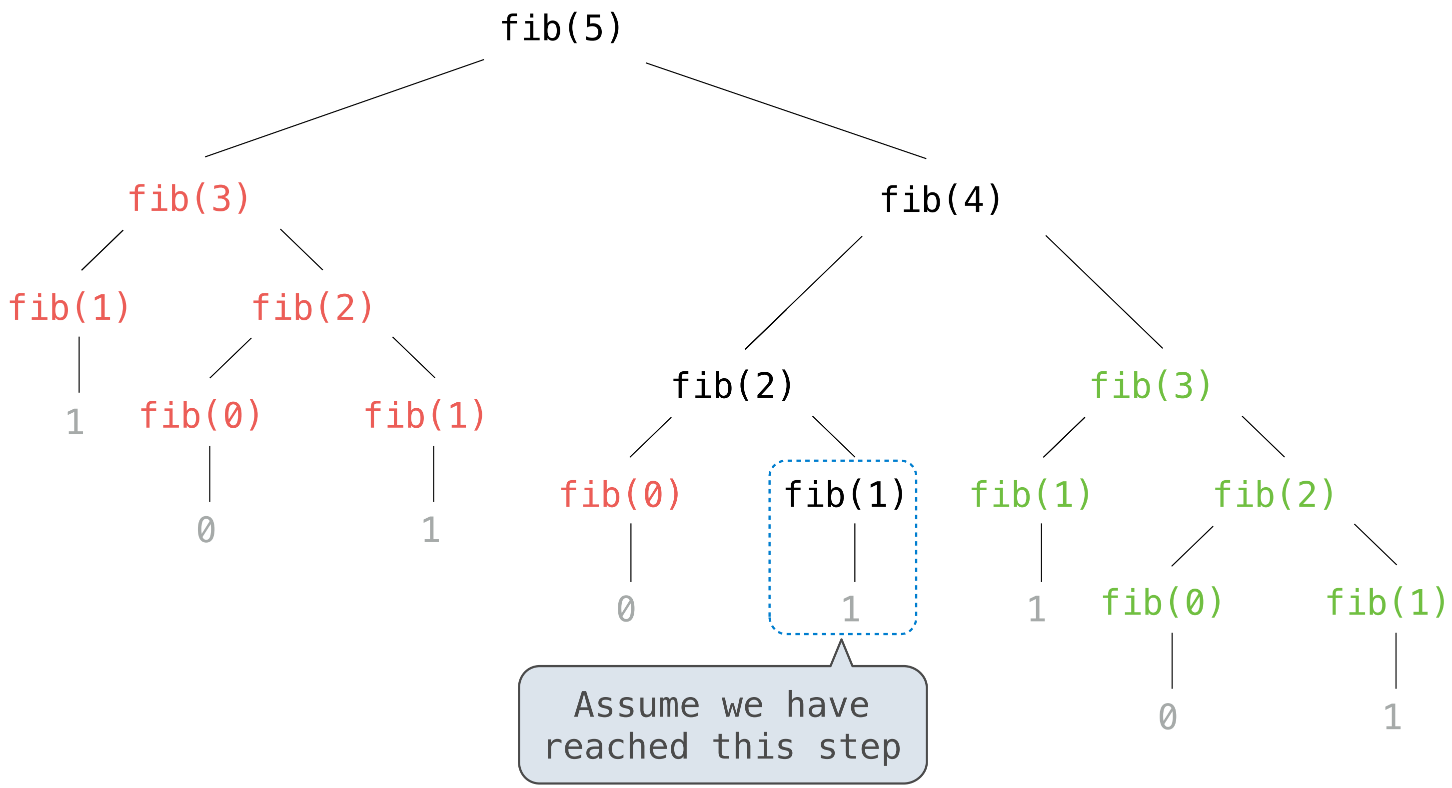
Fibonacci Space Consumption



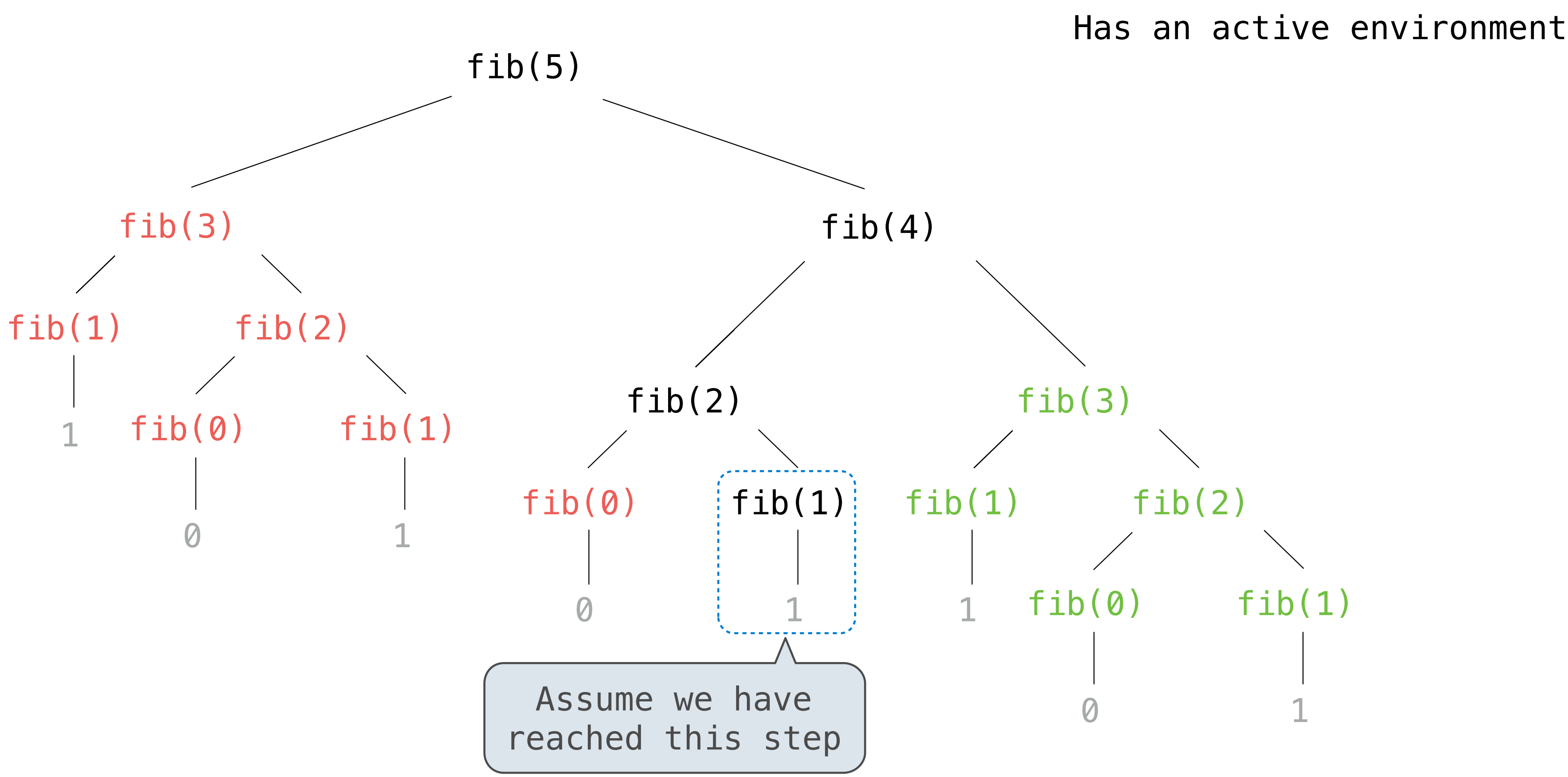
Fibonacci Space Consumption



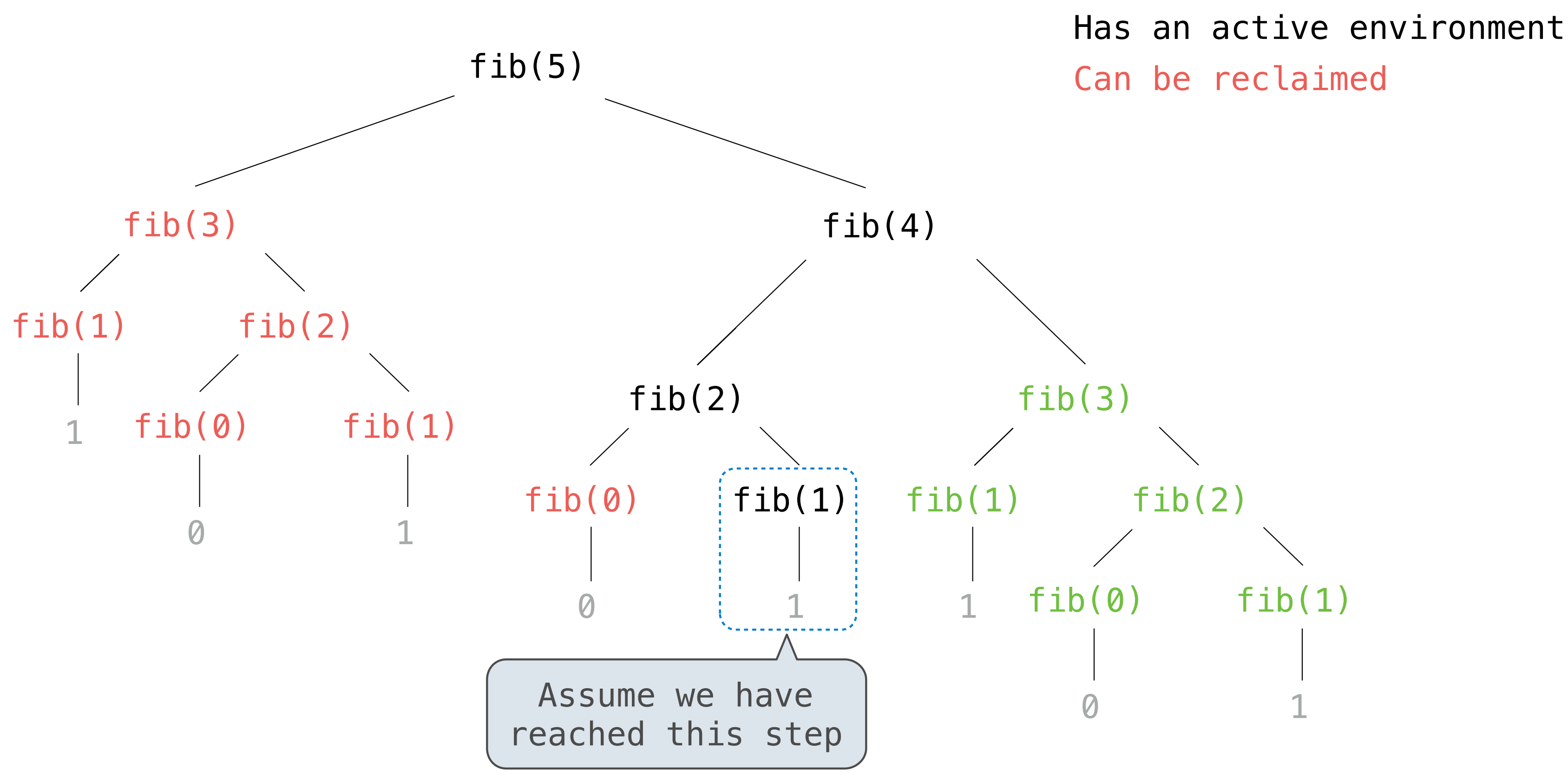
Fibonacci Space Consumption



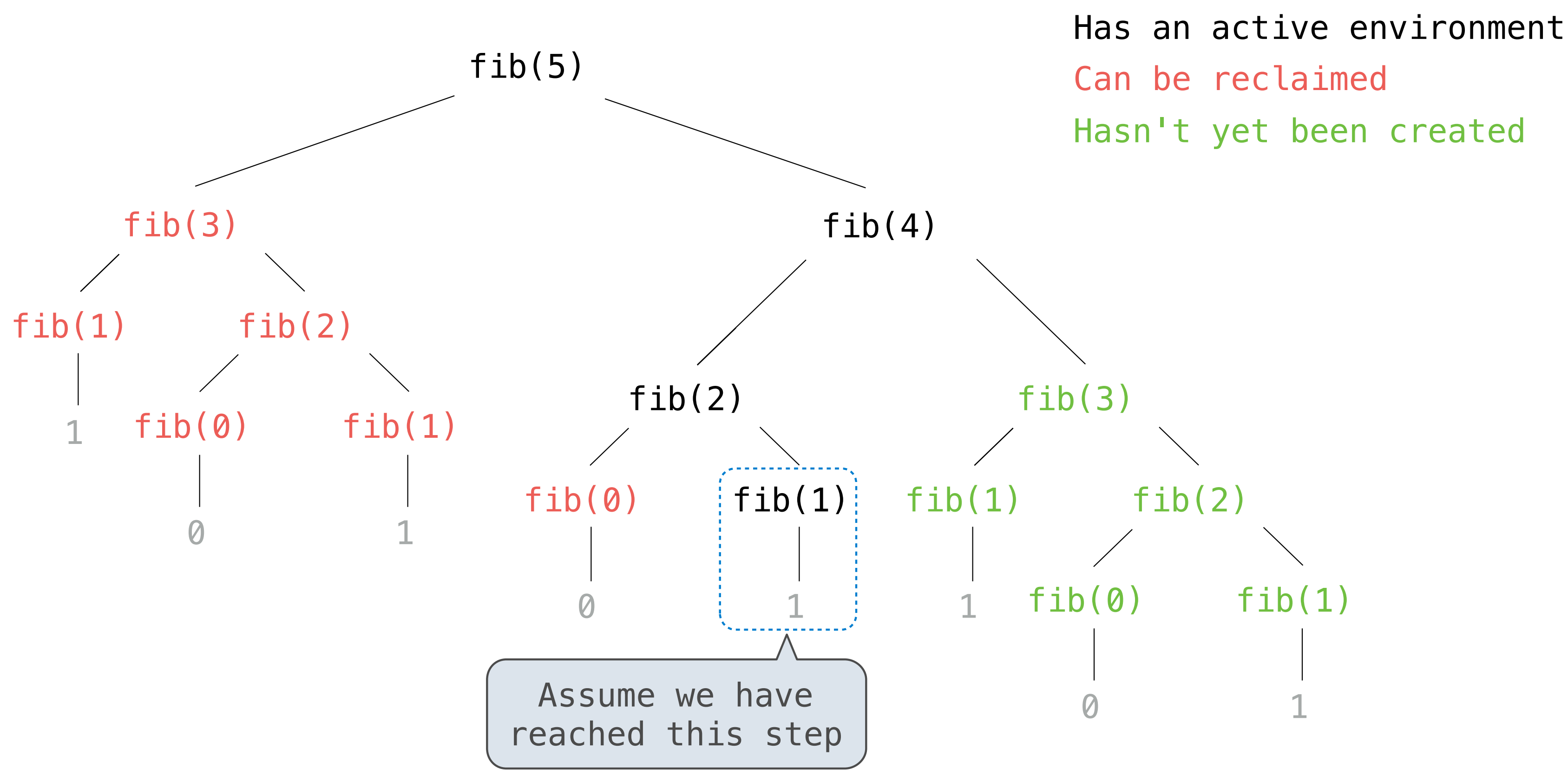
Fibonacci Space Consumption



Fibonacci Space Consumption



Fibonacci Space Consumption



Order of Growth

Order of Growth

Order of Growth

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

Order of Growth

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

n: size of the problem

Order of Growth

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

n: size of the problem

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

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

means that there are positive constants $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

means that there are positive constants $\mathbf{k_1}$ and $\mathbf{k_2}$ such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for all **n** larger than some minimum **m**

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

means that there are positive constants **k_1** and **k_2** such that

$$\boxed{k_1 \cdot f(n) \leq R(n)} \leq k_2 \cdot f(n)$$

for all **n** larger than some minimum **m**

Order of Growth

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

n: size of the problem

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

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq \boxed{R(n) \leq k_2 \cdot f(n)}$$

for all **n** larger than some minimum **m**

Counting Factors

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1} = 1$:

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1} = 1$:

- Statements outside the while: 4 or 5

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose **k_1** = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1}$ = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n} - 1$ and $\sqrt{n}$

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1} = 1$:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n} - 1$ and $\sqrt{n}$
- Total number of statements executed: at least $4 + 3(\sqrt{n} - 1)$

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose **k_1** = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n} - 1$ and $\sqrt{n}$
- Total number of statements executed: at least $4 + 3(\sqrt{n} - 1)$

To check the *upper bound*

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose **k_1** = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n}-1$ and $\sqrt{n}$
- Total number of statements executed: at least $4+3(\sqrt{n}-1)$

To check the *upper bound*

- Maximum statements executed: $5+4\sqrt{n}$

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1}$ = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n}-1$ and $\sqrt{n}$
- Total number of statements executed: at least $4+3(\sqrt{n}-1)$

To check the *upper bound*

- Maximum statements executed: $5+4\sqrt{n}$
- Maximum operations required per statement: some **p**

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose **k_1** = 1:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n} - 1$ and $\sqrt{n}$
- Total number of statements executed: at least $4 + 3(\sqrt{n} - 1)$

To check the *upper bound*

- Maximum statements executed: $5 + 4\sqrt{n}$
- Maximum operations required per statement: some **p**

Assumption: every statement, such as addition-then-assignment using the += operator, takes some fixed number of operations to execute

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Counting Factors

Number of operations required to count the factors of n using factors_fast is $\Theta(\sqrt{n})$

To check the *lower bound*, we choose $\mathbf{k_1} = 1$:

- Statements outside the while: 4 or 5
- Statements within the while (including header): 3 or 4
- while statement iterations: between $\sqrt{n} - 1$ and $\sqrt{n}$
- Total number of statements executed: at least $4 + 3(\sqrt{n} - 1)$

To check the *upper bound*

- Maximum statements executed: $5 + 4\sqrt{n}$
- Maximum operations required per statement: some **p**
- We choose $\mathbf{k_2} = 5\mathbf{p}$ and $\mathbf{m} = 25$

```
def factors_fast(n):
    sqrt_n = sqrt(n)
    k, total = 1, 0
    while k < sqrt_n:
        if divides(k, n):
            total += 2
        k += 1
    if k * k == n:
        total += 1
    return total
```

Assumption: every statement, such as addition-then-assignment using the += operator, takes some fixed number of operations to execute

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

Time	Space

Slow: Test each k from 1 through n

Fast: Test each k from 1 to square root n
For every k, n/k is also a factor!

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!		

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Order of Growth of Counting Factors

Implementations of the same functional abstraction can require different amounts of time

Problem: How many factors does a positive integer n have?

A factor k of n is a positive integer that evenly divides n

```
def factors(n):
```

	Time	Space
Slow: Test each k from 1 through n	$\Theta(n)$	$\Theta(1)$
Fast: Test each k from 1 to square root n For every k, n/k is also a factor!	$\Theta(\sqrt{n})$	$\Theta(1)$

Assumption: integers occupy a fixed amount of space

Exponentiation

Exponentiation

Exponentiation

Goal: one more multiplication lets us double the problem size

Exponentiation

Goal: one more multiplication lets us double the problem size

```
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

```
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

```
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

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

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

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

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)
```

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

Exponentiation

Goal: one more multiplication lets us double the problem size

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

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

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

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)
```

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

(Demo)

Exponentiation

Goal: one more multiplication lets us double the problem size

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


def square(x):
    return x*x

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)
```

Time	Space

Exponentiation

Goal: one more multiplication lets us double the problem size

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


def square(x):
    return x*x

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)
```

Time	Space
$\Theta(n)$	$\Theta(n)$

Exponentiation

Goal: one more multiplication lets us double the problem size

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

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

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)
```

	Time	Space
`exp`	$\Theta(n)$	$\Theta(n)$
`exp_fast`	$\Theta(\log n)$	$\Theta(\log n)$

Comparing Orders of Growth

Properties of Orders of Growth

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

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

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

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

Outer: length of a

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:          Outer: length of a
        if item in b:       Inner: length of b
            count += 1
    return count
```

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

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

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

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

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:          Outer: length of a
        if item in b:       Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

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

Outer: length of a

Inner: length of b

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n)$$

Properties of Orders of Growth

Constants: Constant terms do not affect the order of growth of a process

$$\Theta(n) \qquad \Theta(500 \cdot n) \qquad \Theta(\frac{1}{500} \cdot n)$$

Logarithms: The base of a logarithm does not affect the order of growth of a process

$$\Theta(\log_2 n) \qquad \Theta(\log_{10} n) \qquad \Theta(\ln n)$$

Nesting: When an inner process is repeated for each step in an outer process, multiply the steps in the outer and inner processes to find the total number of steps

```
def overlap(a, b):
    count = 0
    for item in a:        # Outer: length of a
        if item in b:     # Inner: length of b
            count += 1
    return count
```

If a and b are both length **n**, then overlap takes $\Theta(n^2)$ steps

Lower-order terms: The fastest-growing part of the computation dominates the total

$$\Theta(n^2) \qquad \Theta(n^2 + n) \qquad \Theta(n^2 + 500 \cdot n + \log_2 n + 1000)$$

Comparing orders of growth (n is the problem size)

Comparing orders of growth (n is the problem size)

$$\Theta(b^n)$$

Comparing orders of growth (n is the problem size)

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

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

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

Comparing orders of growth (n is the problem size)

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

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

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

$\Theta(n^2)$

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

Comparing orders of growth (n is the problem size)

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

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$

Comparing orders of growth (n is the problem size)

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

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

$\Theta(\log n)$

Comparing orders of growth (n is the problem size)

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

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n).

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n).

$\Theta(1)$

Comparing orders of growth (n is the problem size)

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

$\Theta(\phi^n)$ steps, where $\phi = \dfrac{1+\sqrt{5}}{2} \approx 1.61828$

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., overlap

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

$\Theta(n)$ Linear growth. E.g., slow factors or exp

$\Theta(\sqrt{n})$ Square root growth. E.g., factors_fast

$\Theta(\log n)$ Logarithmic growth. E.g., exp_fast

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter

Comparing orders of growth (n is the problem size)

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

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

Incrementing the problem scales R(n) by a factor

$\Theta(n^2)$ Quadratic growth. E.g., `overlap`

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

$\Theta(n)$ Linear growth. E.g., slow `factors` or `exp`

$\Theta(\sqrt{n})$ Square root growth. E.g., `factors_fast`

$\Theta(\log n)$ Logarithmic growth. E.g., `exp_fast`

Doubling the problem only increments R(n).

$\Theta(1)$ Constant. The problem size doesn't matter