

LECTURE 11

Mutual Recursion
Tree Recursion
Complexity

MUTUAL RECURSION
(WATCH TEXTBOOK
VIDEO!)

TYPES OF RECURSIONS

Direct Recursion

```
def function(x):
```

```
    ...
```

```
    function(x-1)
```

```
    ...
```

TYPES OF RECURSIONS

Direct Recursion

```
def function(x):  
    ...  
  
    function(x-1)  
    ...
```

Indirect (or Mutual) Recursion

```
def function_a(x):  
    ...  
    function_b(x-1)  
    ...
```

```
def function_b(y):  
    ...  
    function_a(y-2)  
    ...
```

CLASSIC EXAMPLE: EVEN - ODD (TEXTBOOK)

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):
```

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True
```

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True  
    else:  
        return is_odd(n-1)
```


CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True  
    else:  
        return is_odd(n-1)
```

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True  
    else:  
        return is_odd(n-1)
```

```
def is_odd(n):  
    if n == 0:
```

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True  
    else:  
        return is_odd(n-1)
```

```
def is_odd(n):  
    if n == 0:  
        return False  
    else:
```

CLASSIC EXAMPLE: EVEN - ODD

- a number is **even** if it is one more than an odd number
- a number is **odd** if it is one more than an even number
- 0 is even

```
def is_even(n):  
    if n == 0:  
        return True  
    else:  
        return is_odd(n-1)
```

```
def is_odd(n):  
    if n == 0:  
        return False  
    else:  
        return is_even(n-1)
```

LET'S PRACTICE

Write two methods that calculate the following functions that are defined in terms of each other:

$$A(x) = \begin{cases} 1 & , x \leq 1 \\ B(x + 2) & , x > 1 \end{cases}$$

$$B(x) = \begin{cases} A(x - 3) + 4 \end{cases}$$

LET'S PRACTICE

Write two methods that calculate the following functions that are defined in terms of each other:

$$A(x) = \begin{cases} 1 & , x \leq 1 \\ B(x+2) & , x > 1 \end{cases}$$
$$B(x) = A(x-3) + 4$$

```
def A(x) :  
    if x <= 1 :  
        return 1  
    else :  
        return B(x+2)  
  
def B(x) :  
    return A(x-3) + 4
```

Tree recursion

1



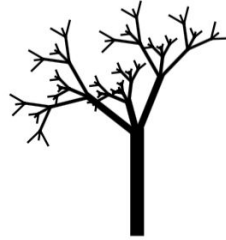
2



3



4



8



Tree recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

```
n: 0, 1, 2, 3, 4, 5, 6, 7, 8, ... , 35
fib(n): 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , 9,227,465
```


Tree recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

`n:` 0, 1, 2, 3, 4, 5, 6, 7, 8, ... , 35

`fib(n):` 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , 9,227,465

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

Tree recursion

Tree-shaped processes arise whenever executing the body of a recursive function makes more than one recursive call

`n:` 0, 1, 2, 3, 4, 5, 6, 7, 8, ... , 35

`fib(n):` 0, 1, 1, 2, 3, 5, 8, 13, 21, ... , 9,227,465

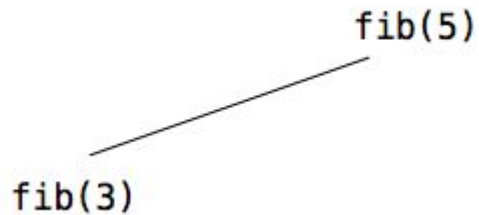
```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```

Tree structure

fib(5)

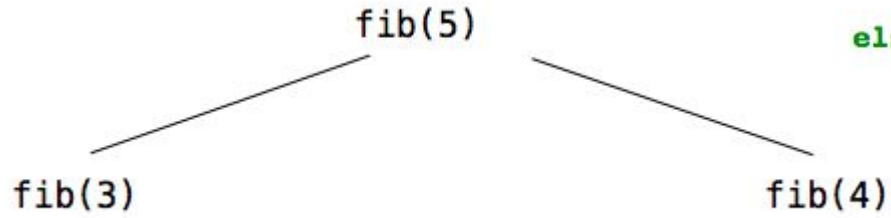
```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```

Tree structure



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

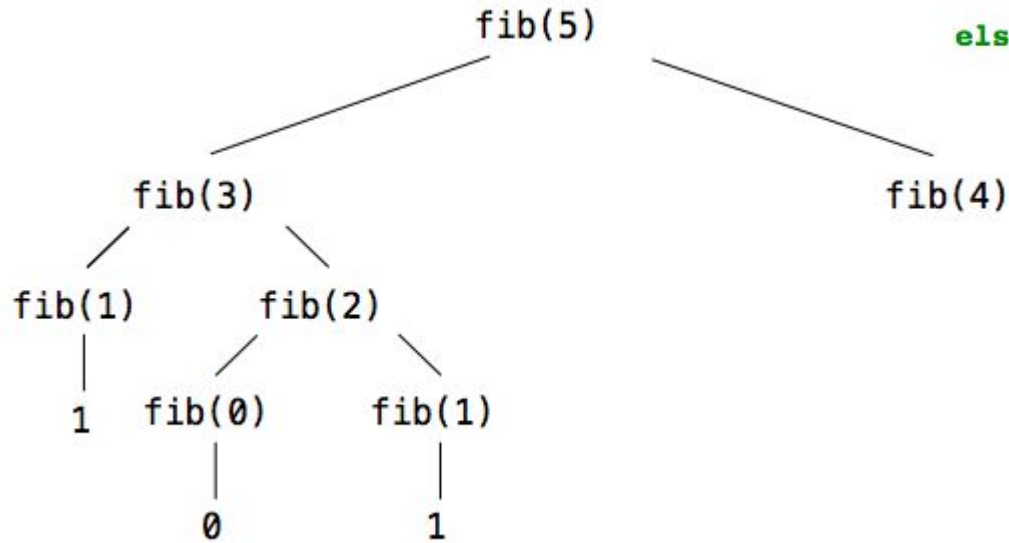
Tree structure



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

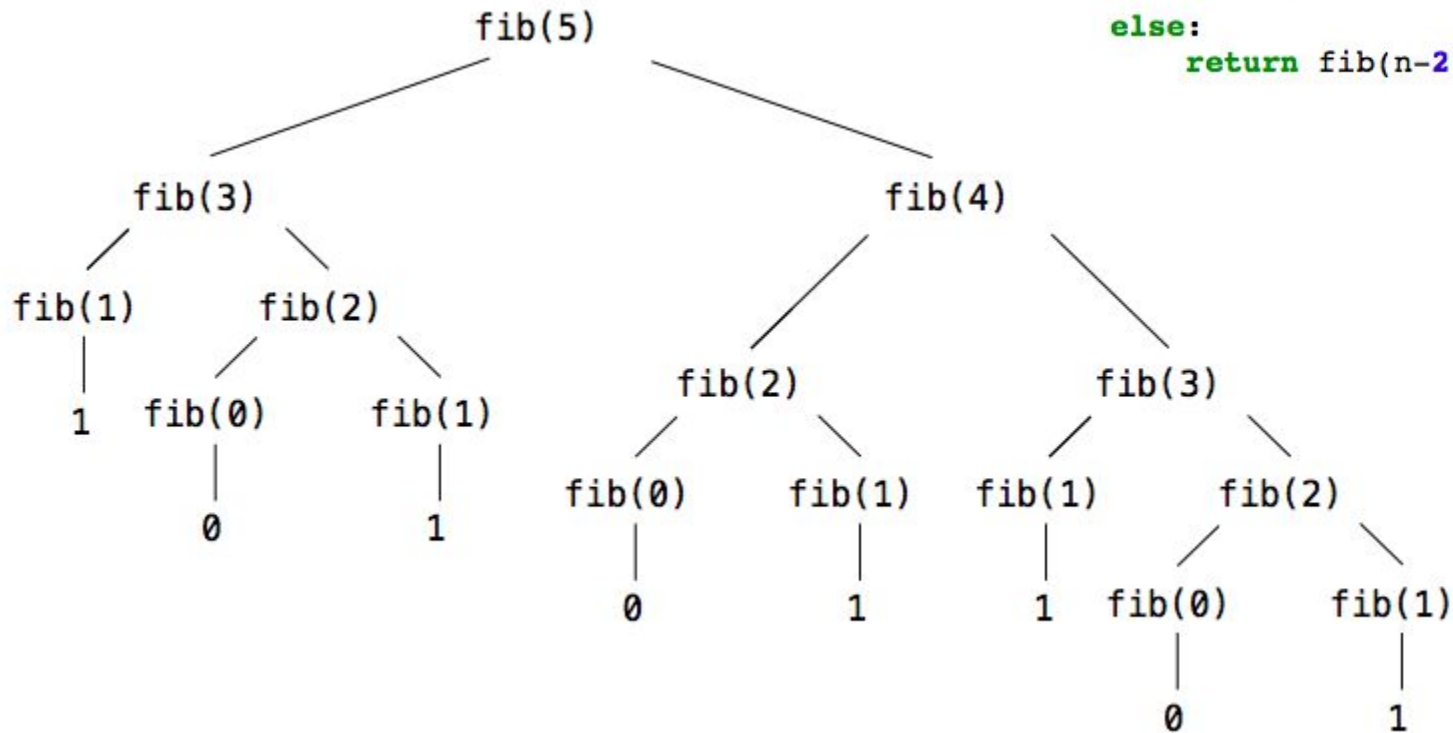
Tree structure

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



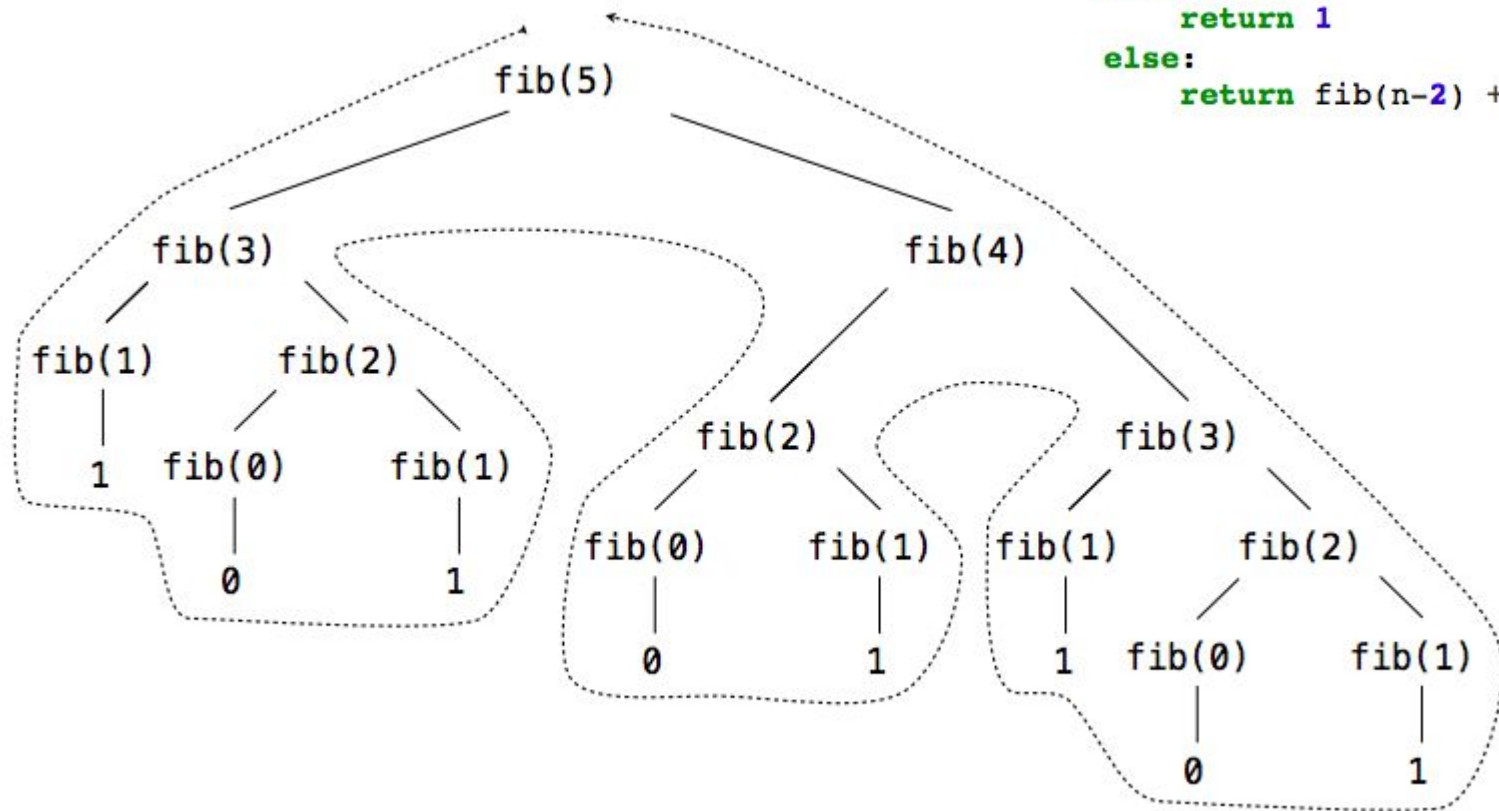
Tree structure

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



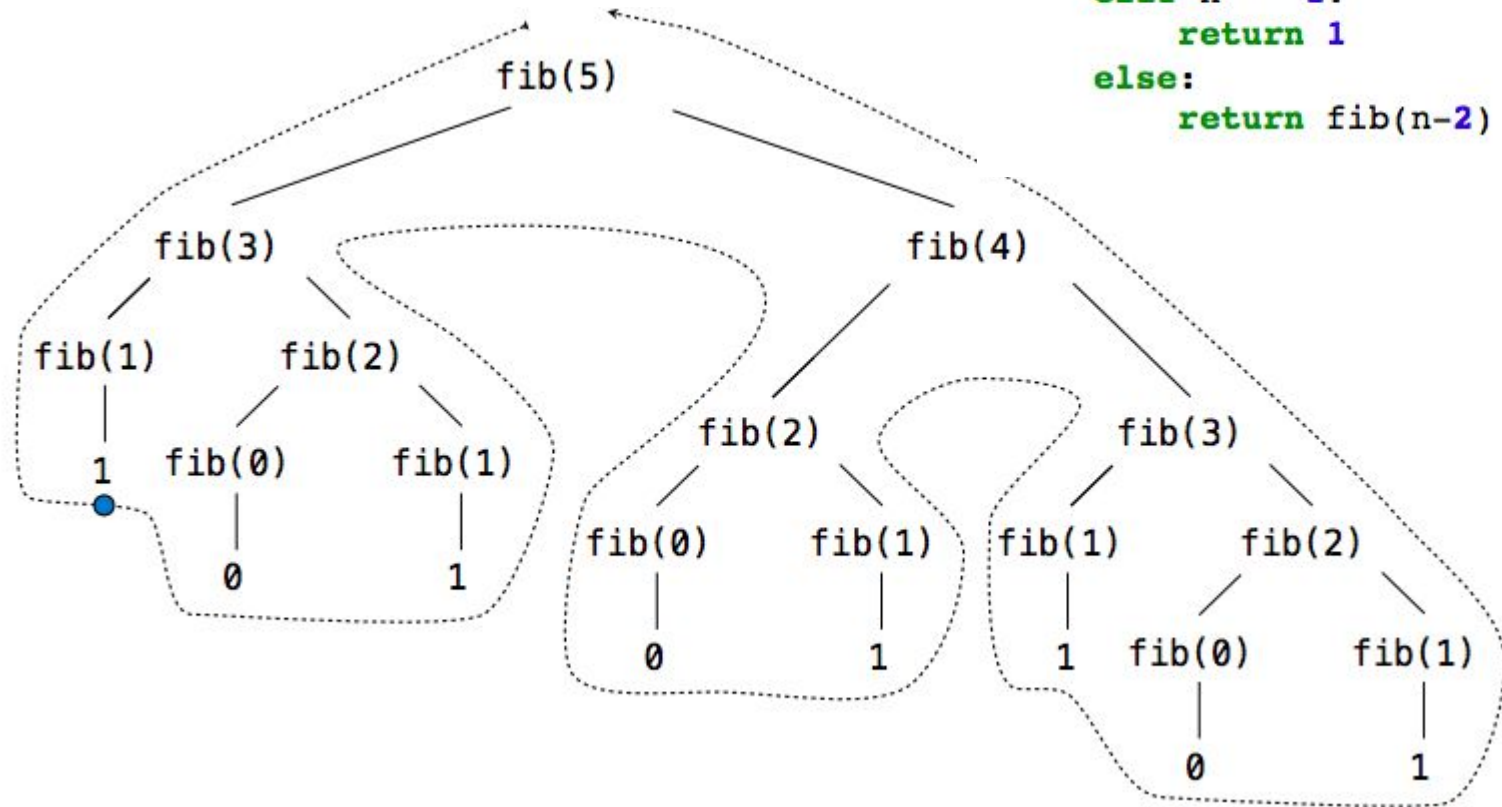
Tree structure

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



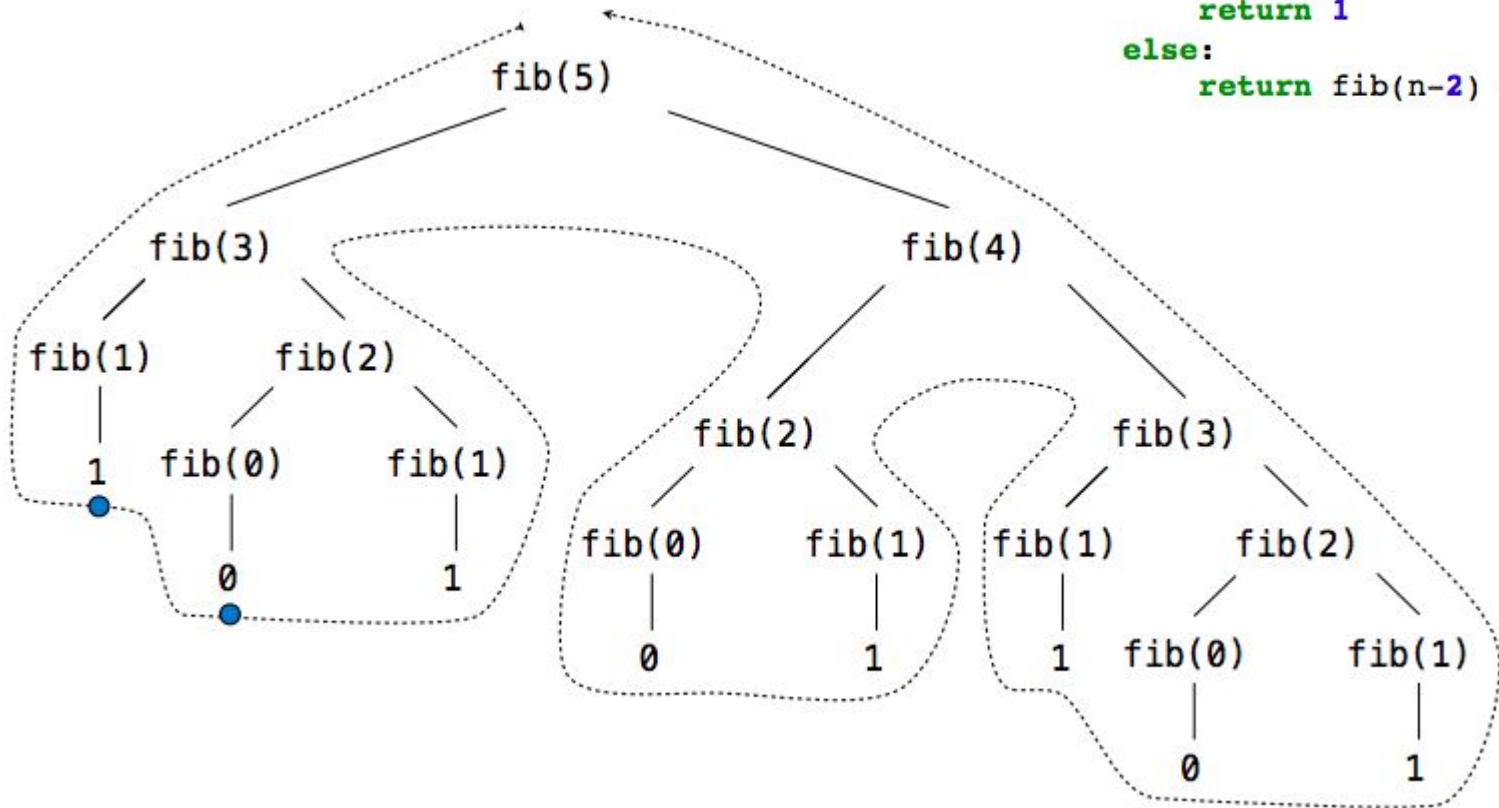
Tree structure

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



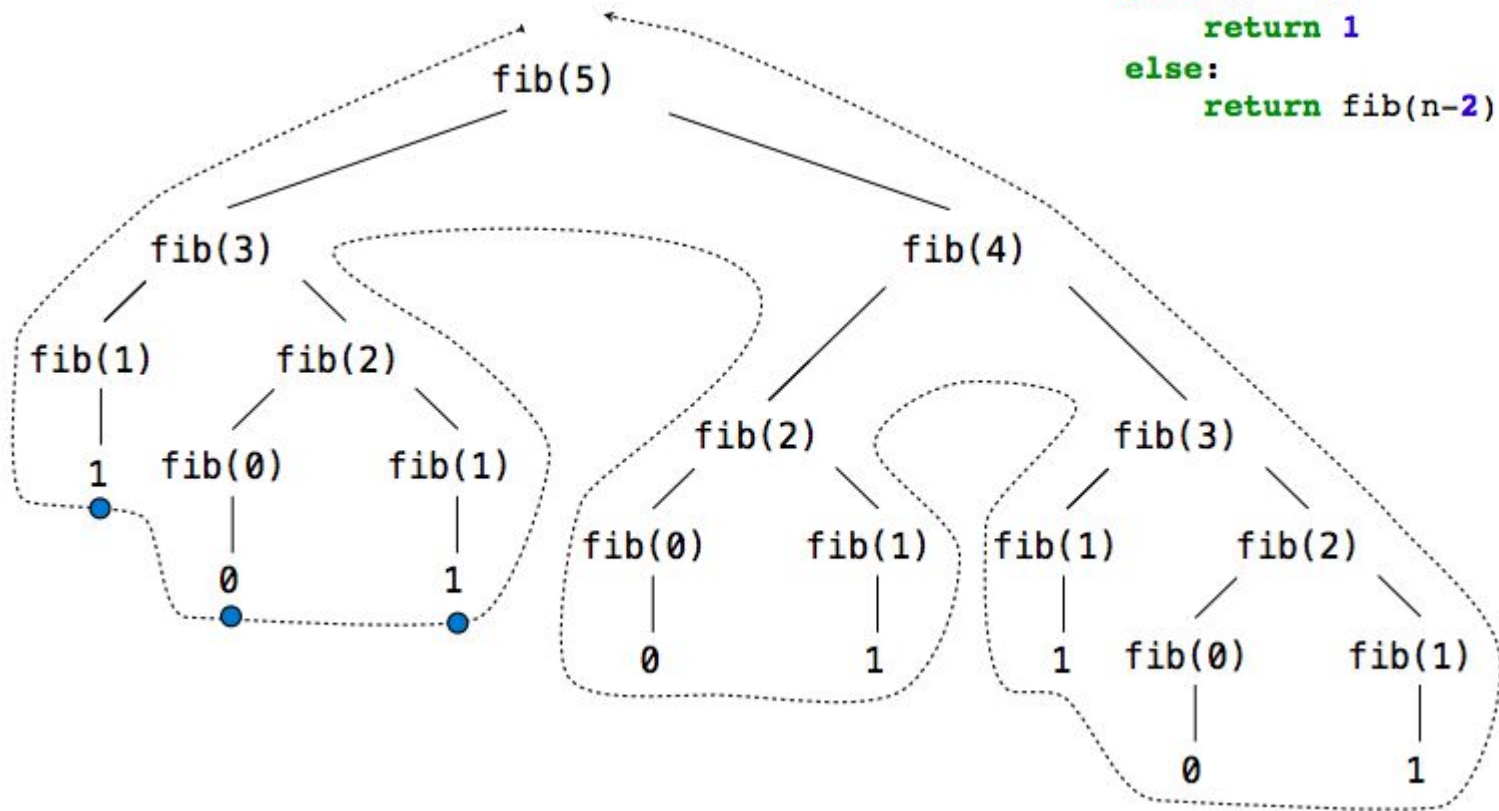
Tree structure

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



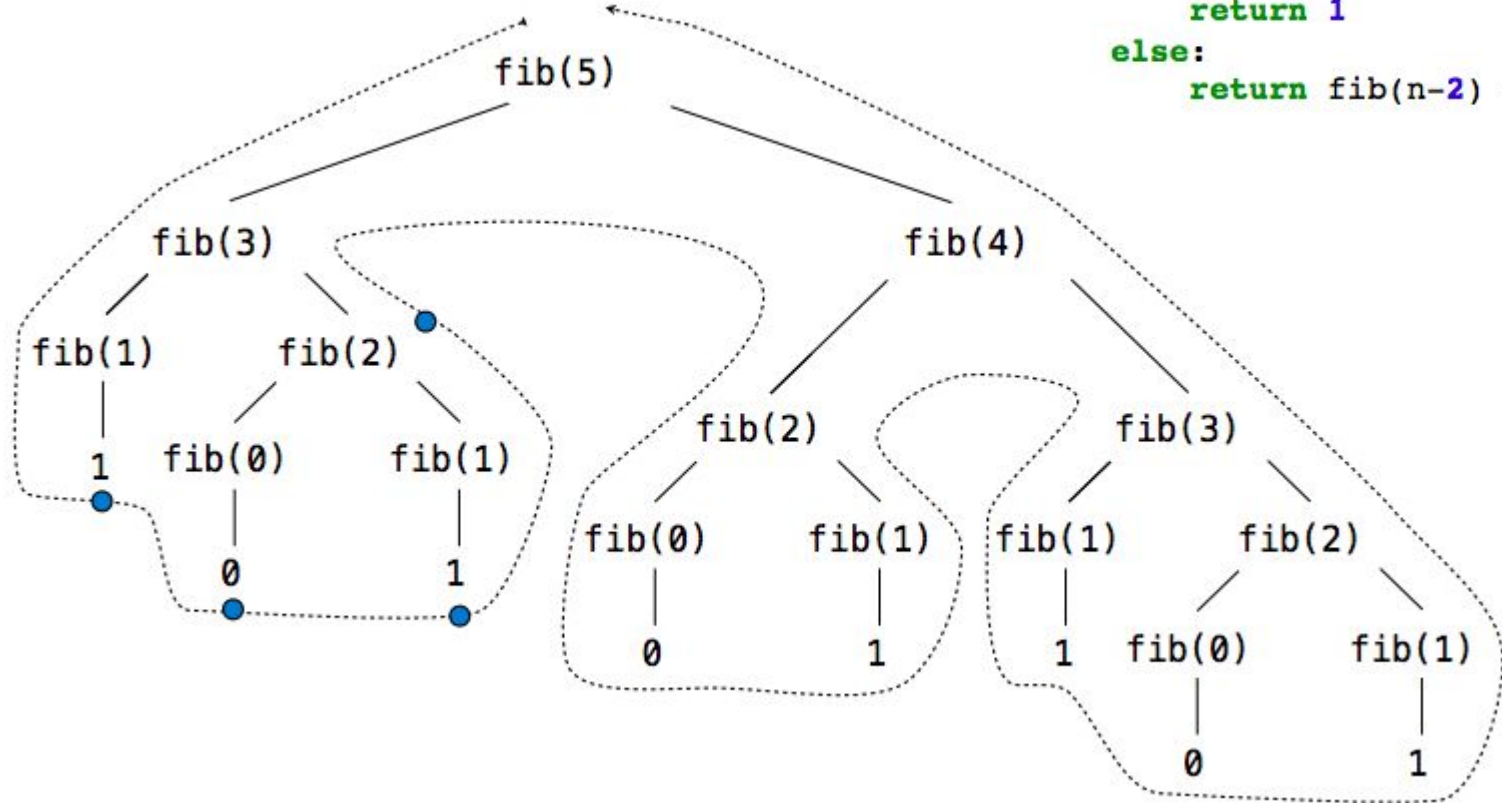
Tree structure

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



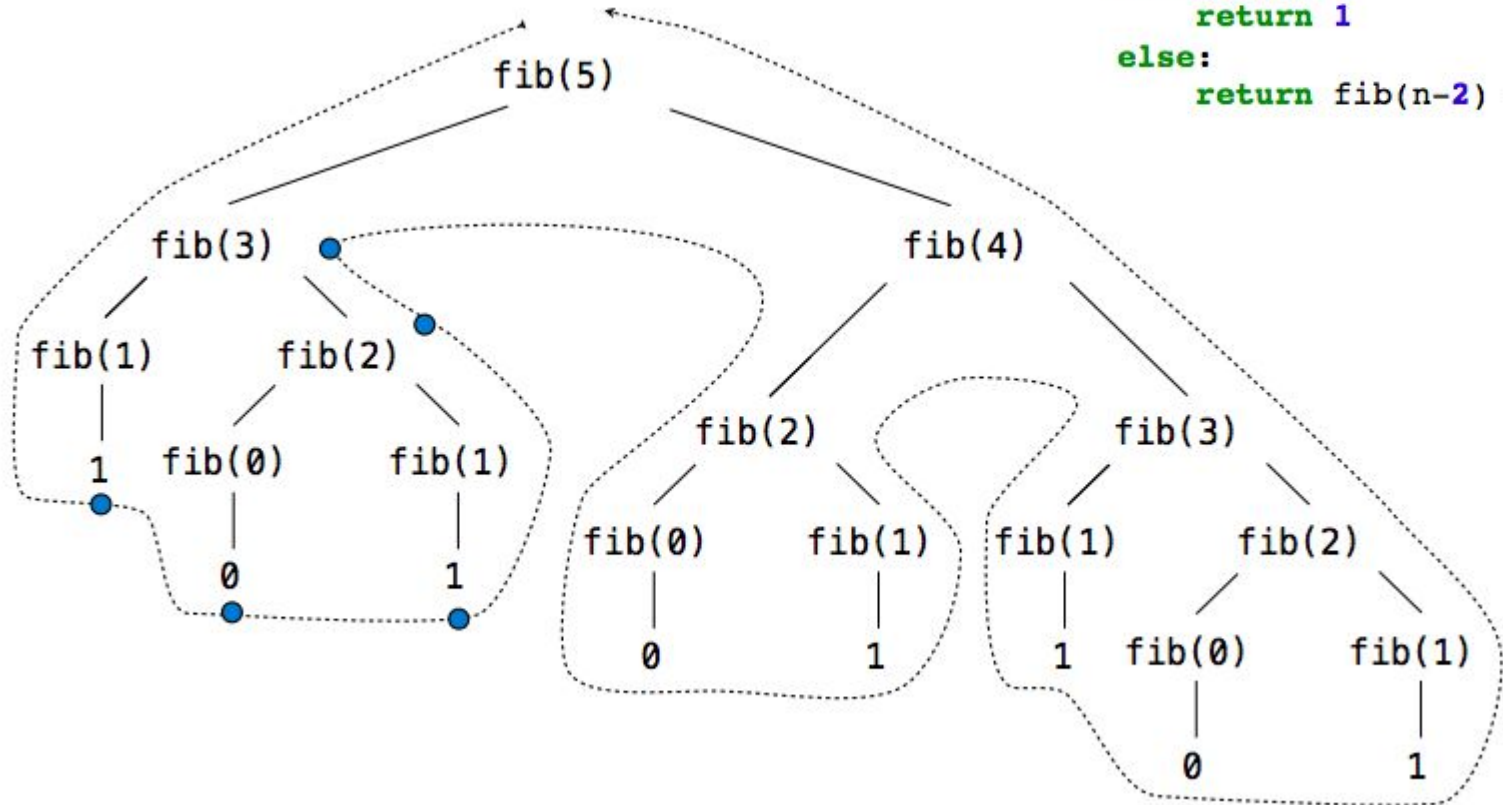
Tree structure

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



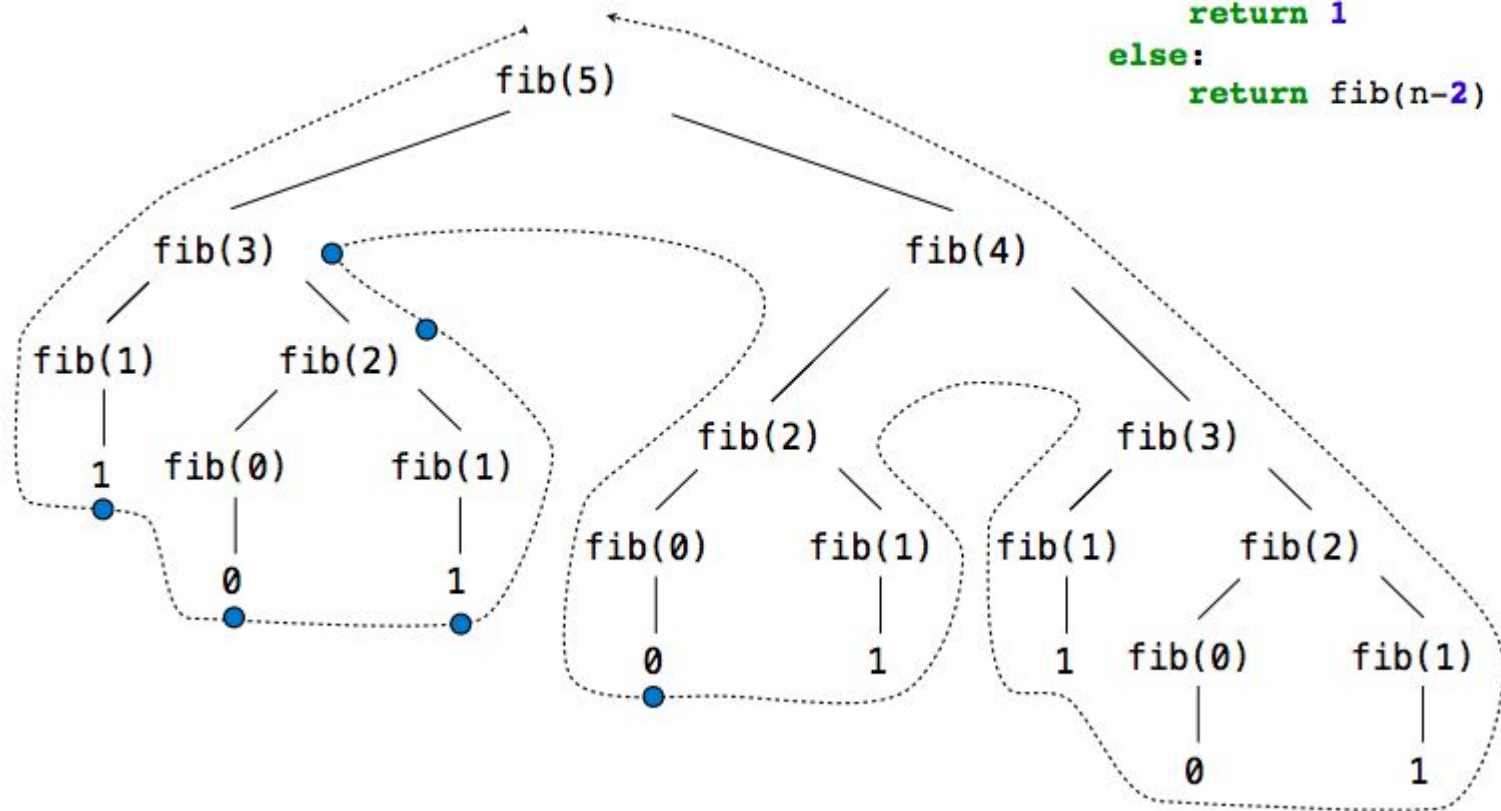
Tree structure

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

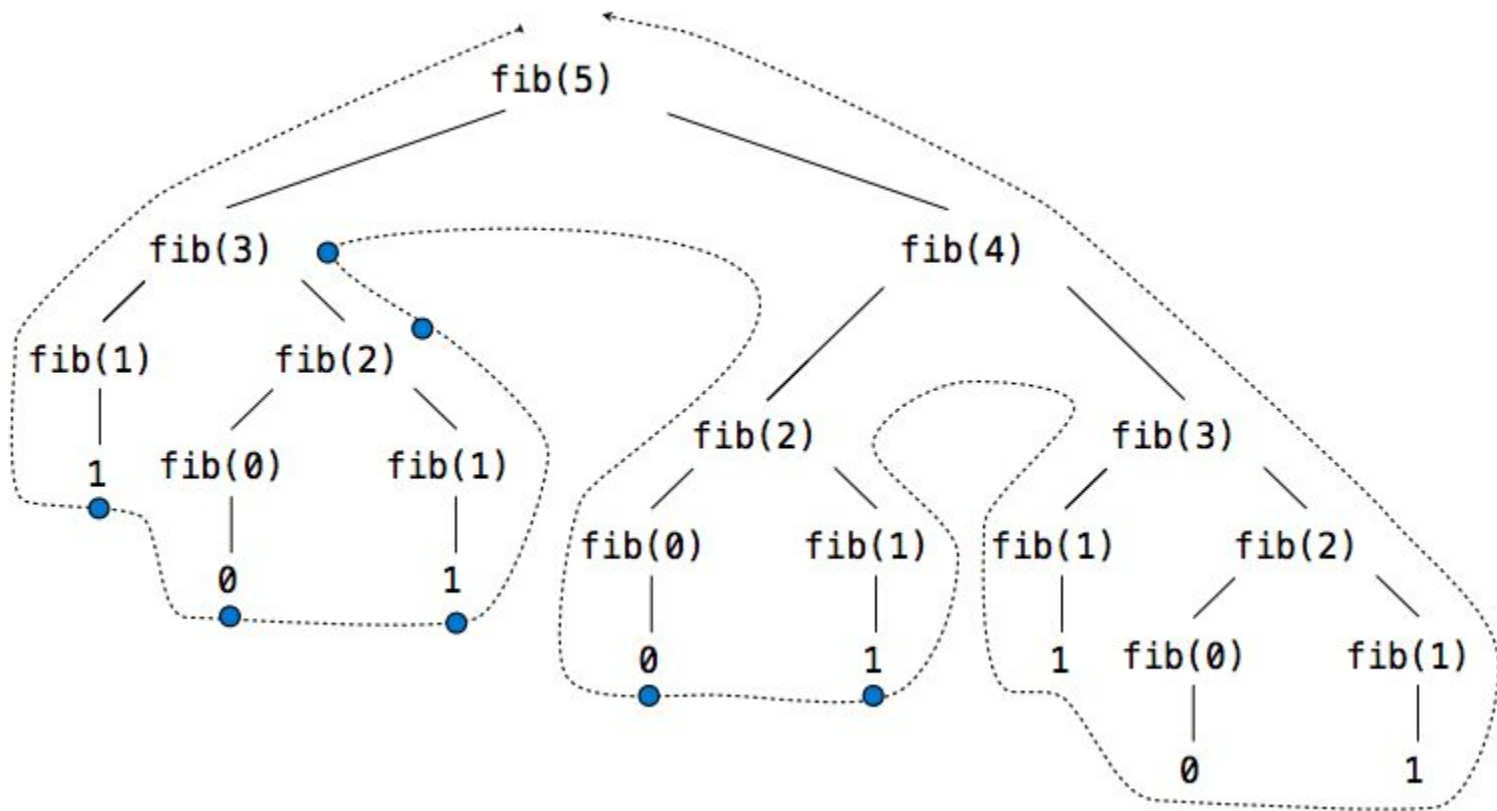


Tree structure

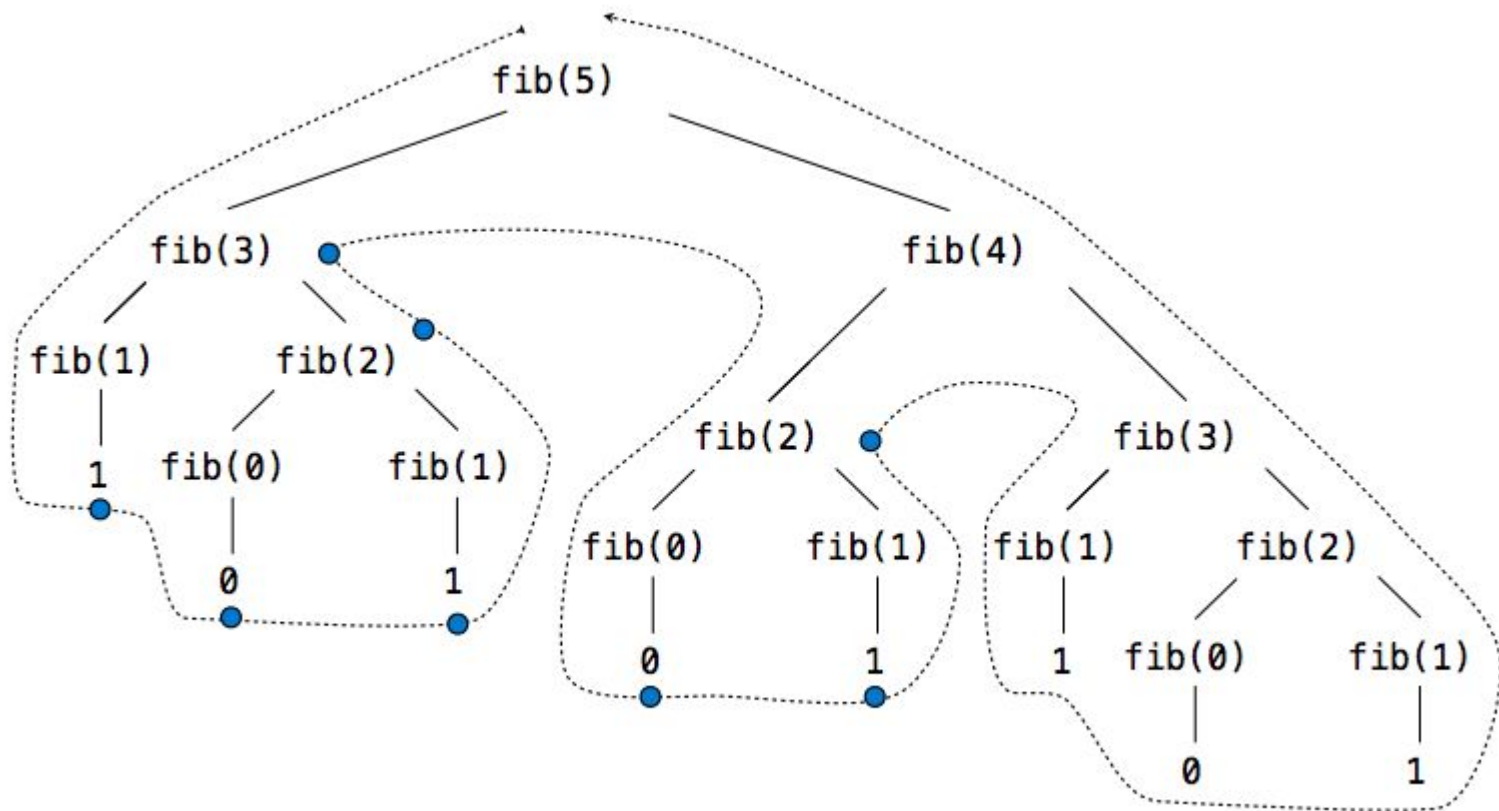
```
def fib(n):  
    if n == 0:  
        return 0  
    elif n == 1:  
        return 1  
    else:  
        return fib(n-2) + fib(n-1)
```



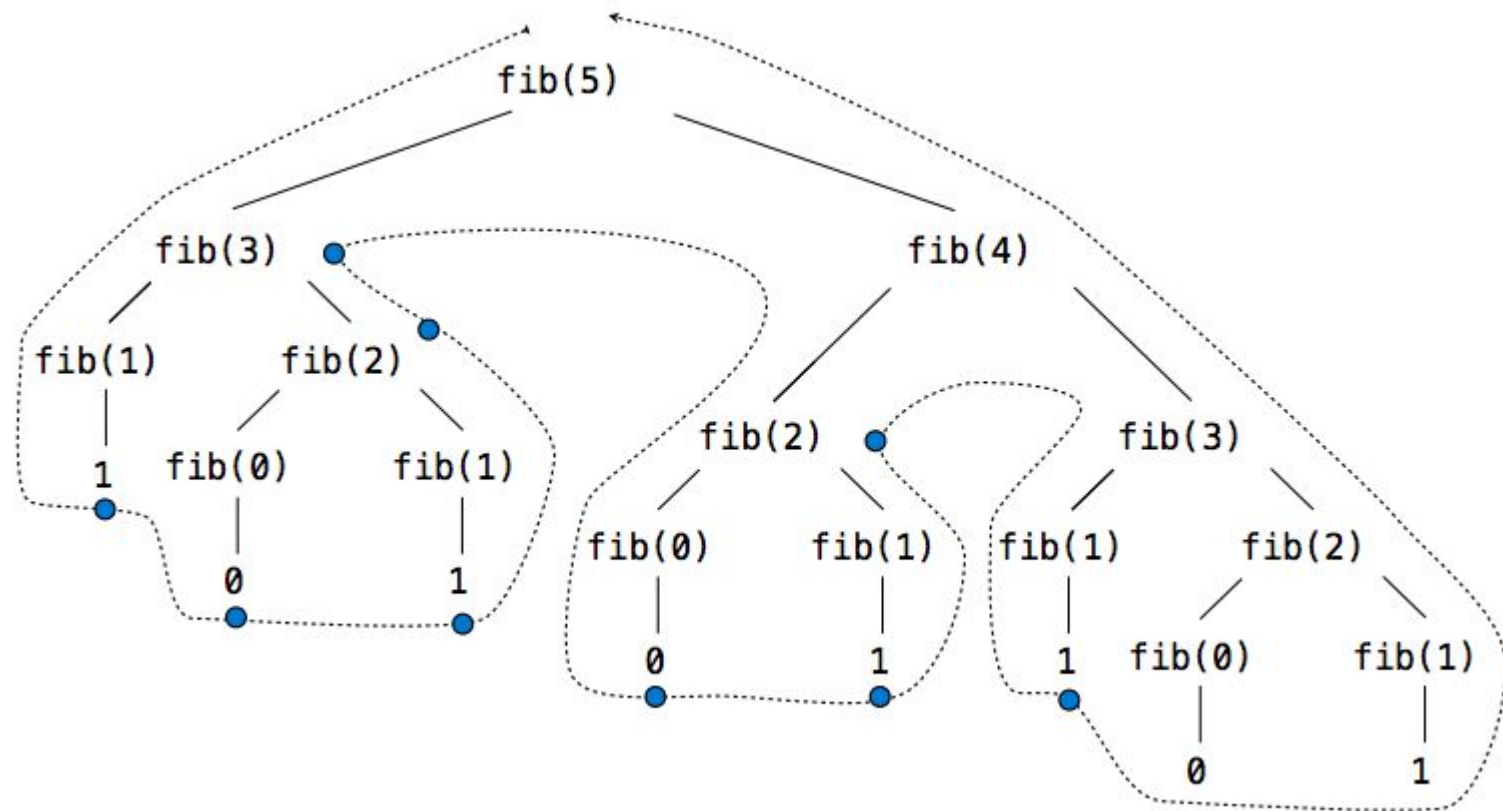
Tree structure



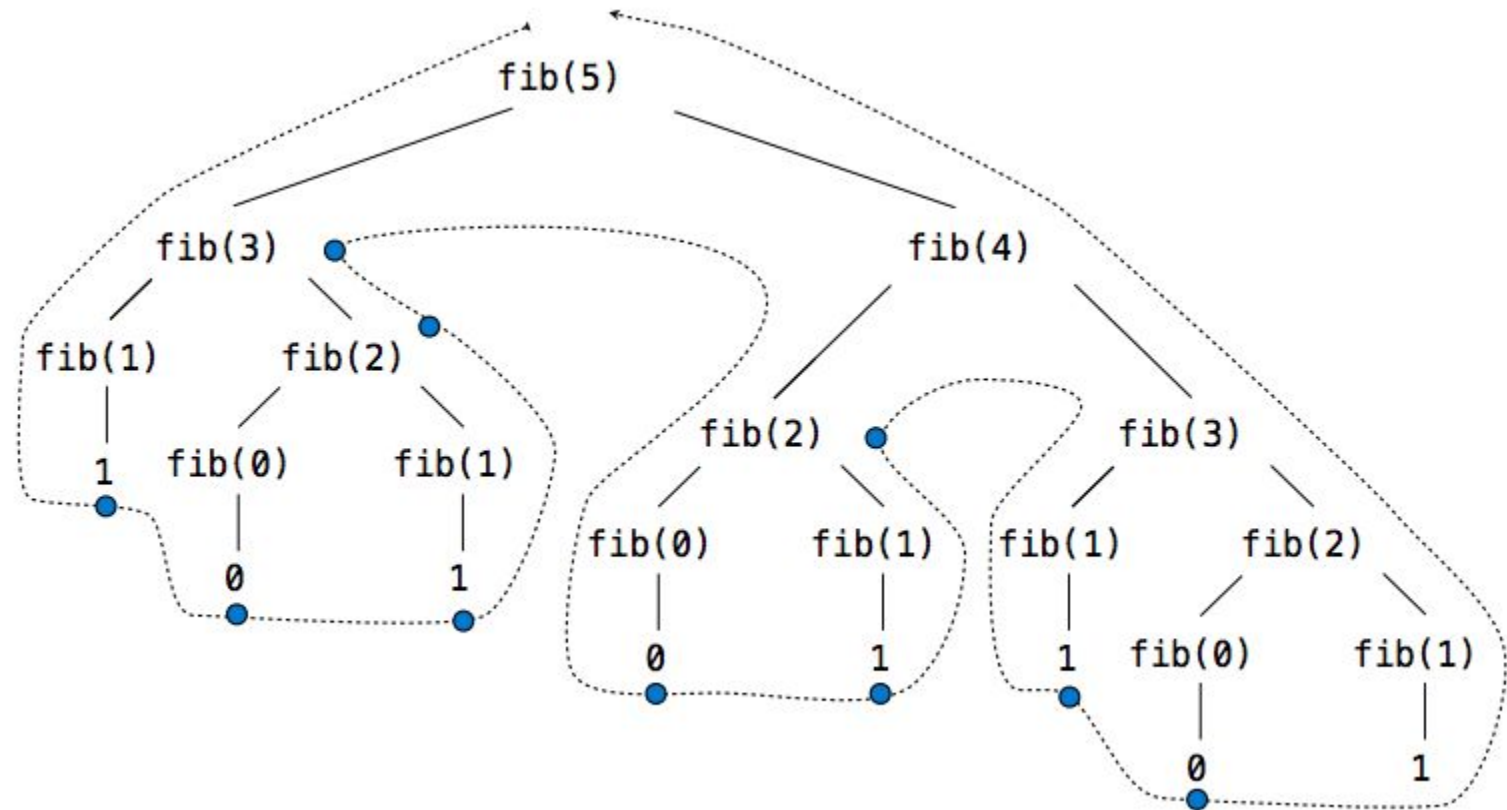
Tree structure



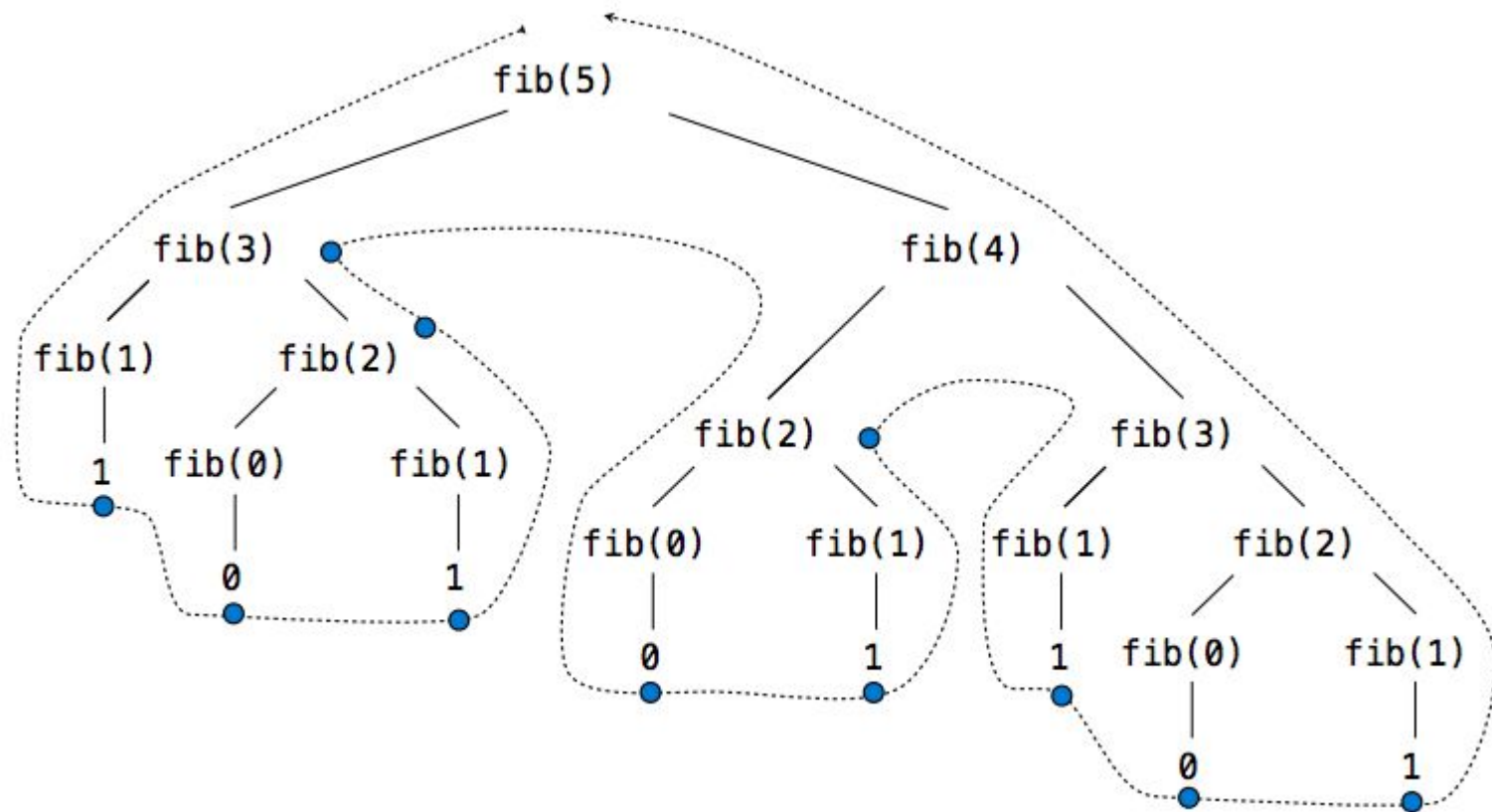
Tree structure



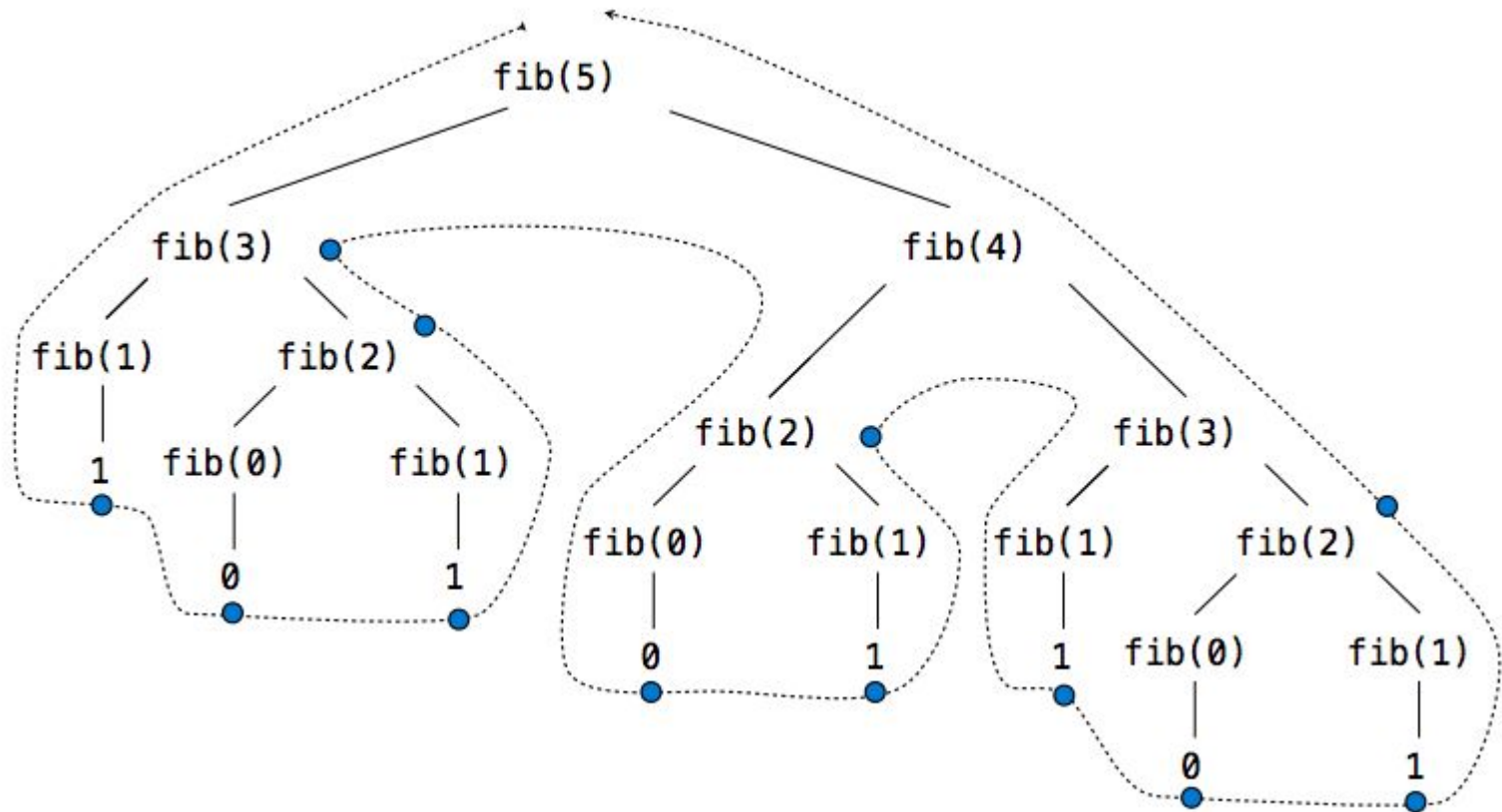
Tree structure



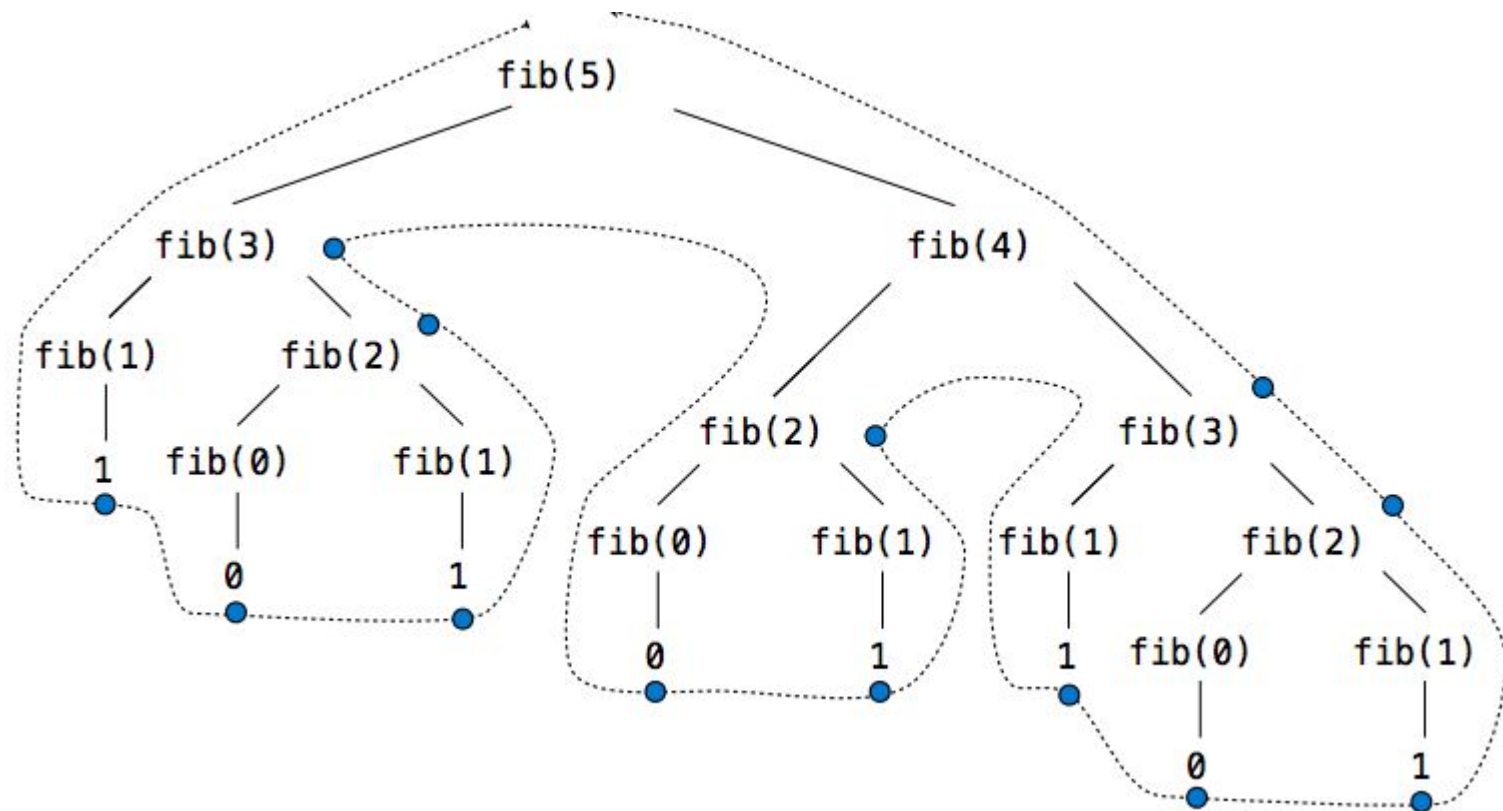
Tree structure



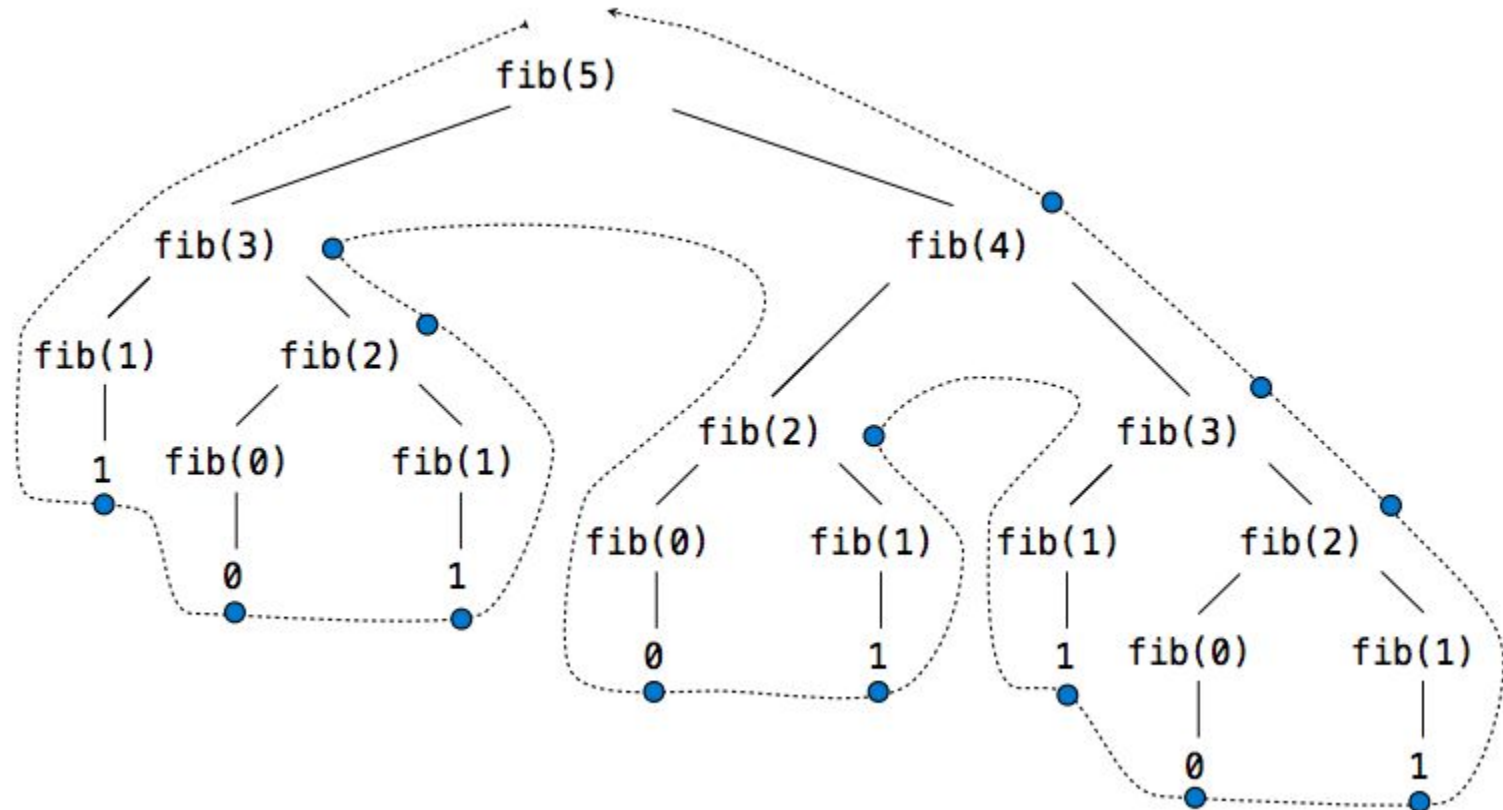
Tree structure



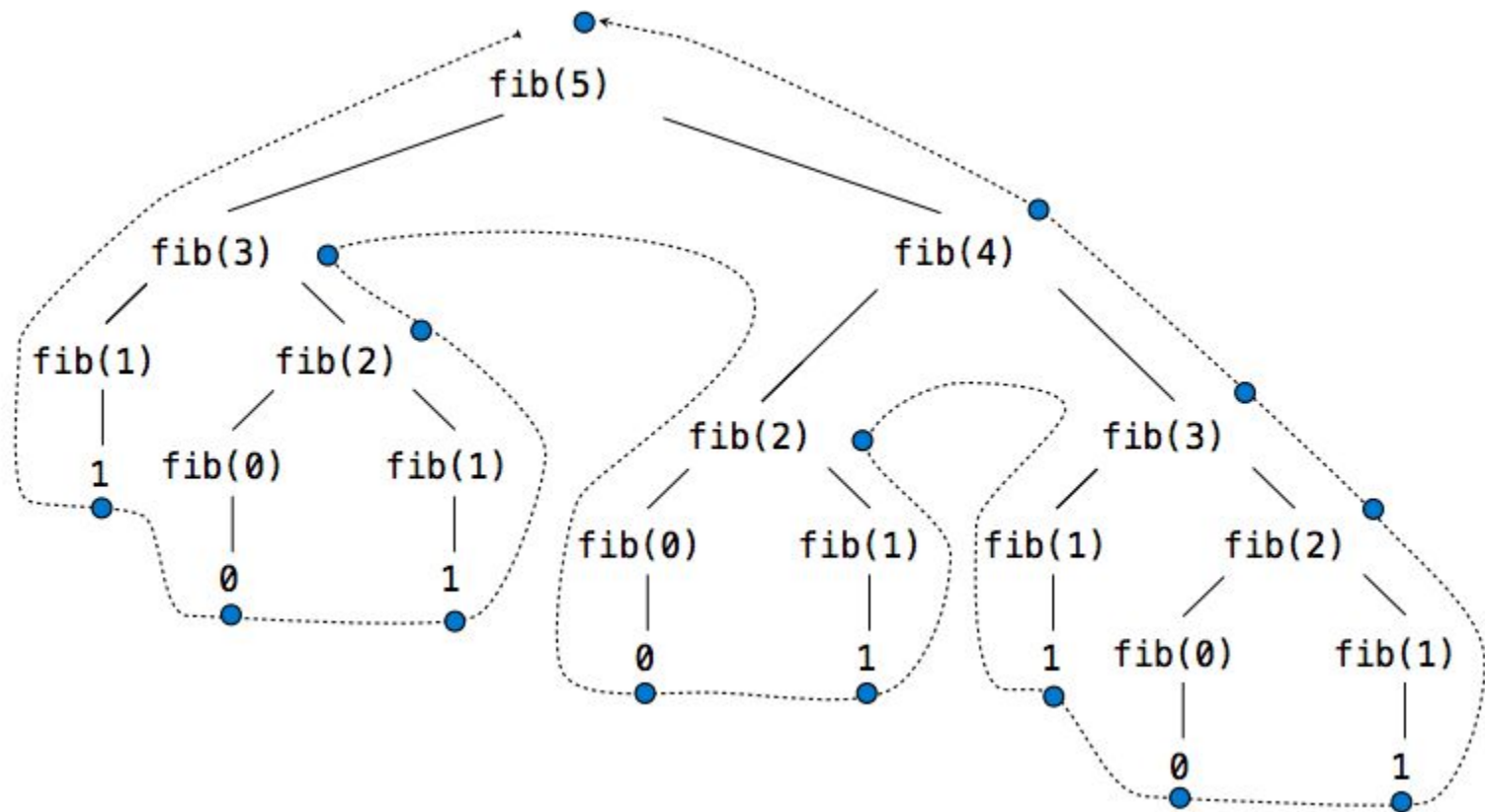
Tree structure



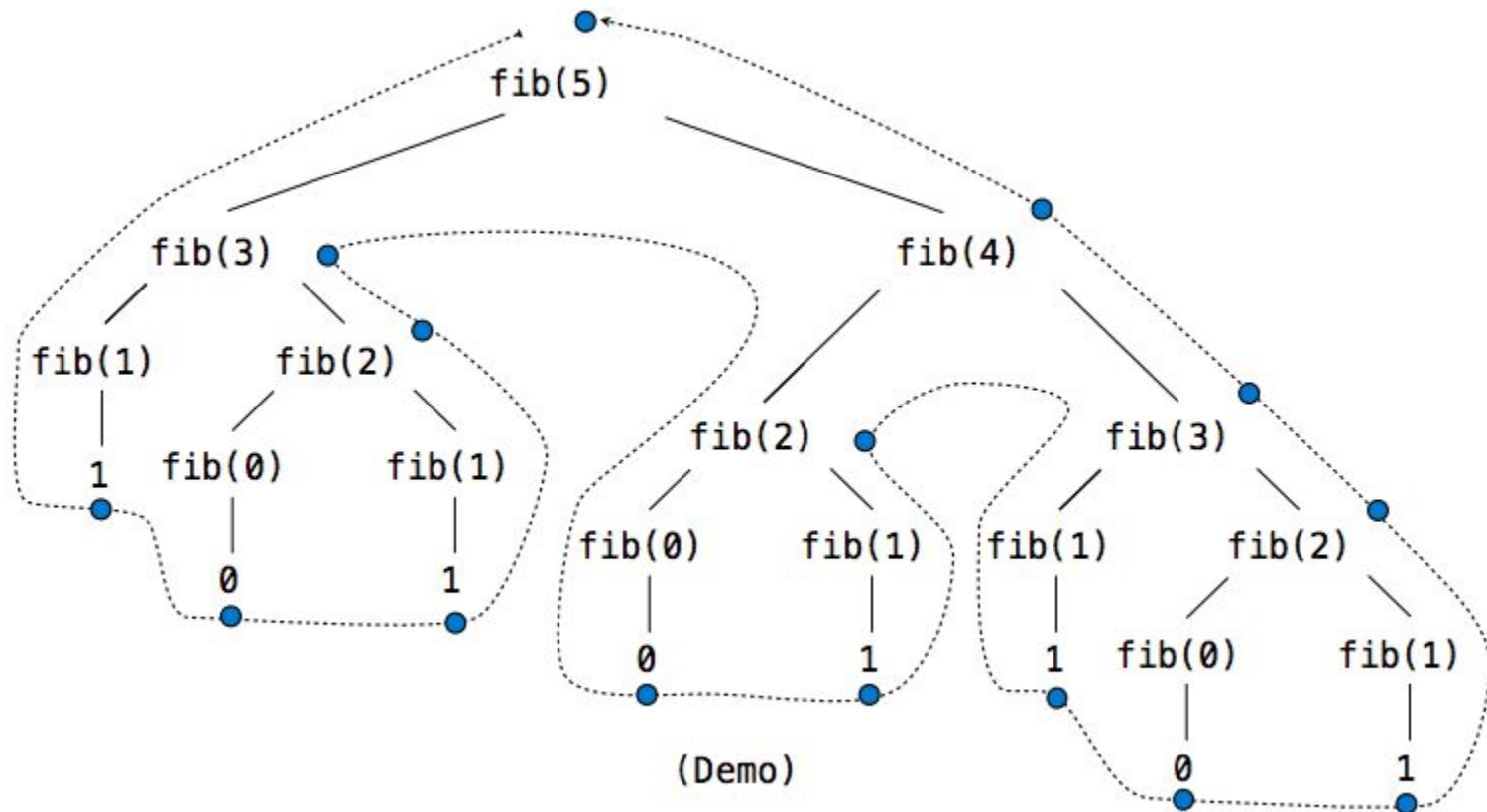
Tree structure



Tree structure



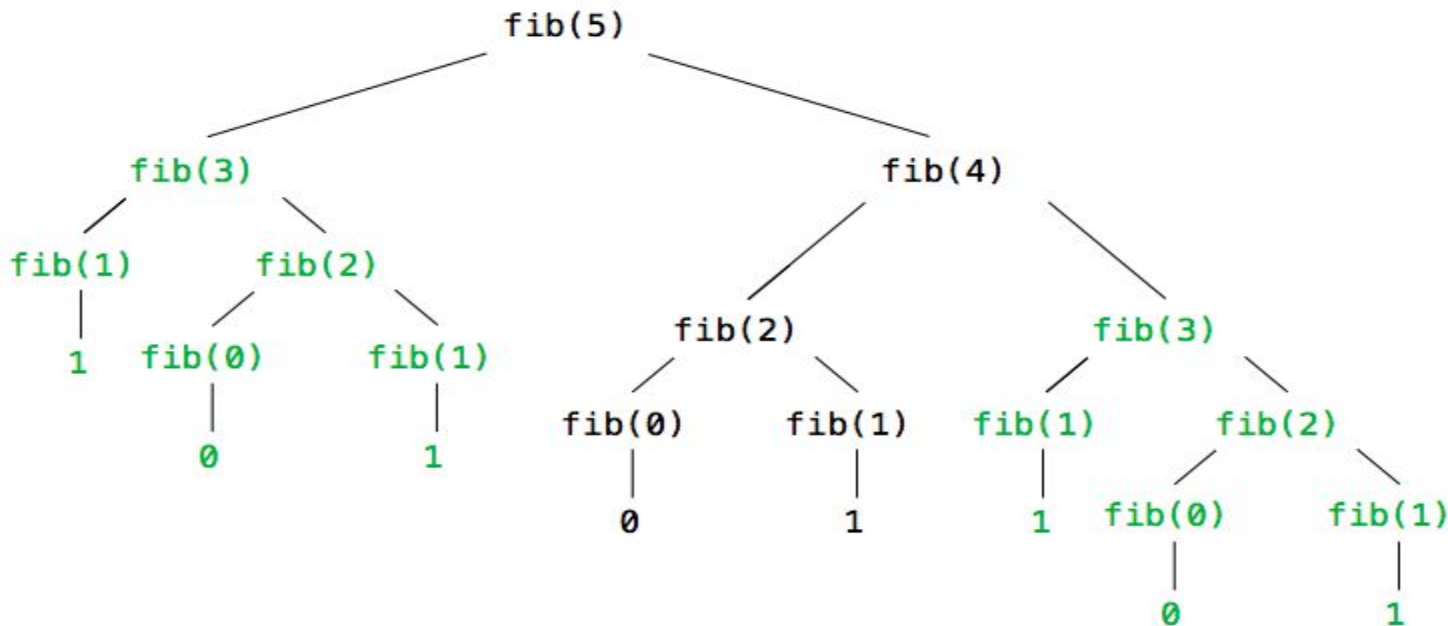
Tree structure



Repetition in Tree-Recursive Computation

Repetition in Tree-Recursive Computation

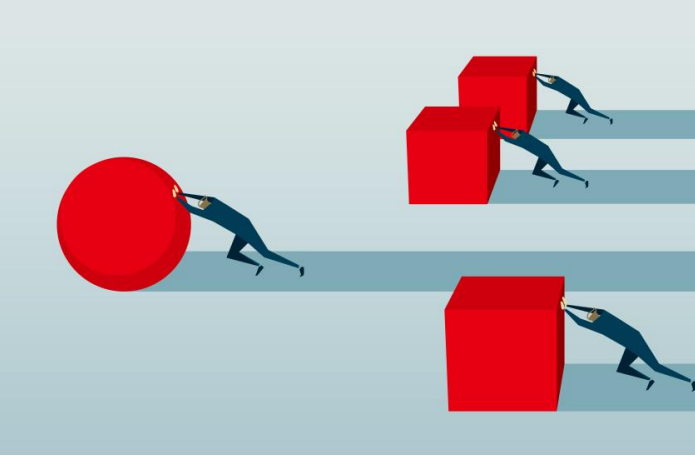
This process is highly repetitive; fib is called on the same argument multiple times



(We will speed up this computation dramatically in a few weeks by remembering results)



Efficiency



Measuring efficiency
(how long it takes your program to run)







Cars

Automobiles are divided by **size** into several categories:

- ☐ subcompacts,
- ☐ compacts,
- ☐ midsize,
- ☐ SUV and so on.

These categories provide a **quick** idea what size car you're talking about, without needing to mention actual dimensions.

☐ Similarly, it's useful to have a shorthand way to say how efficient a computer algorithm is

	Intermediate SUV Ford Escape or similar See All Details [+]	\$884.00 PAY AT COUNTER	\$751.40 PAY NOW Save \$132.60 --
	Compact Ford Focus or similar See All Details [+]	\$995.48 PAY AT COUNTER	\$846.09 PAY NOW Save \$149.31 --
	Economy Ford Fiesta or similar See All Details [+]	\$1,084.55 PAY AT COUNTER	\$921.87 PAY NOW Save \$162.68 --
	Full Size Ford Fusion or similar See All Details [+]	\$1,359.40 PAY AT COUNTER	\$1,155.49 PAY NOW Save \$203.91 --
	Intermediate Chevrolet Cruze or similar See All Details [+]	\$1,378.00 PAY AT COUNTER	\$1,171.30 PAY NOW Save \$206.70 --
	Standard Buick Verano or similar See All Details [+]	\$1,381.70 PAY AT COUNTER	\$1,174.44 PAY NOW Save \$207.26 --

Algorithmic complexity

Algorithmic complexity is a **very important topic** in computer science. Knowing the complexity of algorithms allows you to answer questions such as:

- ☐ How long will a program run on an input?
- ☐ How much space will it take?
- ☐ Is the problem even solvable?
- ☐ What data structure (ADT) to choose.

Complexity

- How well a computer algorithm scales as the amount of data increases.
- □ We need a way to compare algorithms. So we will learn how to create special categories (runtime classes), add algorithms to these categories and compare them instead.

Lists: One application

playlist

Uptown funk

Shake It Off

All About
that Bass

Shut Up and
Dance

...

Thinking Out
Loud

List: one application



Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist. How many places do I have to look to determine whether “All About that Bass” is in my playlist?

- A. 1
- B. 3
- C. 10
- D. 20
- E. Other

Lists: Running time



Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.

How many places do I have to look to determine whether "Riptide" is in my playlist in the BEST case?

- A. 1
- B. 10
- C. 15
- D. 20
- E. Other

Lists: Running time



Imagine that this list is implemented as an array and that there are currently 20 songs in my playlist, but now you can't see what any of them are.

How many places do I have to look to determine whether "Riptide" is in my playlist in the WORST case?

- A. 1
- B. 10
- C. 15
- D. 20
- E. Other

Running time: What version of the problem are you analyzing

- One part of figuring out how long a program takes to run is figuring out how “lucky” you got in your input.
 - You might get lucky (**best case**), and require the least amount of time possible
 - You might get unlucky (**worst case**) and require the most amount of time possible
 - Or you might want to know “on average” (**average case**) if you are neither lucky or unlucky, how long does an algorithm take.

Almost always, what we care about is the **WORST CASE** or the **AVERAGE CASE**.
Best case is usually not that interesting.

when we do analysis, we are doing **WORST CASE** analysis unless
otherwise specified.

Analyzing the worst case

```
def find(lst, toFind):  
    for i in lst:  
        if (i == toFind):  
            return True  
    return False
```

How many instructions do you have to execute to find out if the element is in the list in the worst case, if n represents the length of the list?

Analyzing the worst case

```
def find (lst, toFind):  
    for i in lst:  
        if (i == toFind):  
            return True  
    return False
```

```
def slowFind (lst, toFind):  
    for i in lst:  
        print("looking for: " + str(toFind))  
        if (i == toFind):  
            return True  
    return False
```

Which method is faster?

A: find

B: slowFind

C: They are about the same

Analyzing the worst case

```
def find (lst, toFind):  
    for i in lst:  
        if (i == toFind):  
            return True  
    return False
```


```
def fastFind(lst, toFind):  
    return False
```

Which method is faster?

A: find

B: fastFind

C: They are about the same



(Play)lists typically have 1000s (or more!) elements, so we only care about very large n .

`find` and `slowFind` are “more similar” than `find` and `fastFind`.

We use Big-O notation to represent “classes” of running time—e.g. those that have similar behavior as N gets large, give or take a constant factor.

Time: Comparing Implementations

- Implementations of the same functional abstraction can require different resources
- **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!
```

Time (number of divisions)

Question: How many time does each implementation use division?




Def factors(n)

1: Test each k from 1 through n

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

Time (number of divisions)

- | | |
|----------------------|-------------------|
| A: n for (1) | n for (2) |
| B: n^2 for (1) | $n^{0.5}$ for (2) |
| C: $n^{0.5}$ for (1) | n for (2) |
| D: n for (1) | $n^{0.5}$ for (2) |
| E: None of the above | |



Orders 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
- **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 k_1 and k_2 such that

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

Order of Growth

A method for bounding the resources used by a function 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 k

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 k

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)

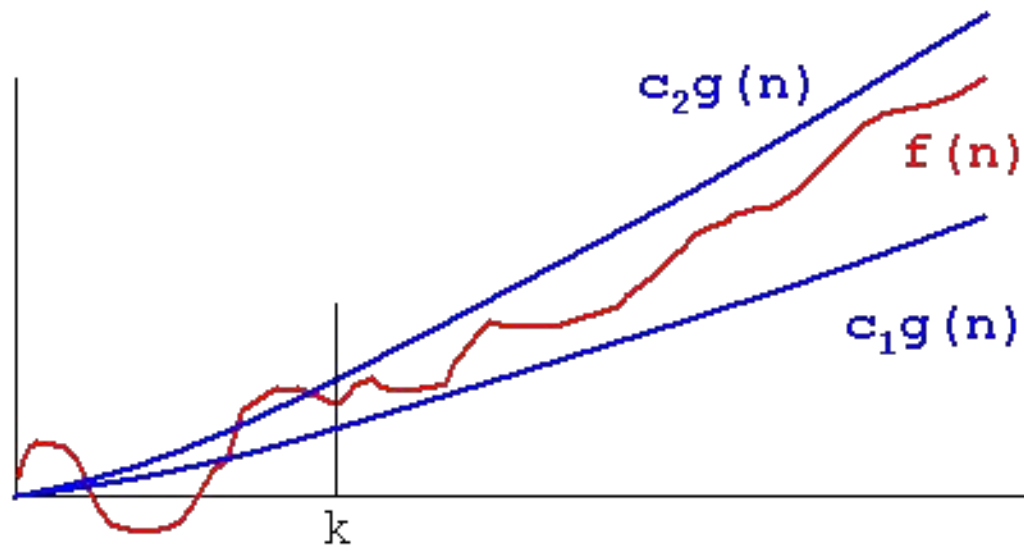
$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

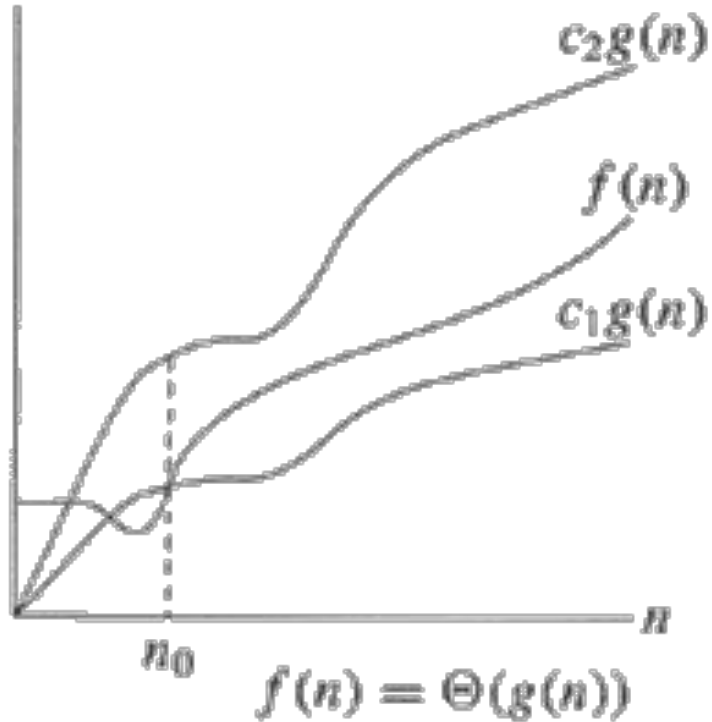
$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for all n larger than some minimum k

Visual



Visual (n_0 instead of k)



Order of Growth of Counting Factors

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

Time

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

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

Time

Slow: Test each k from 1 through n

$\Theta(n)$

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

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

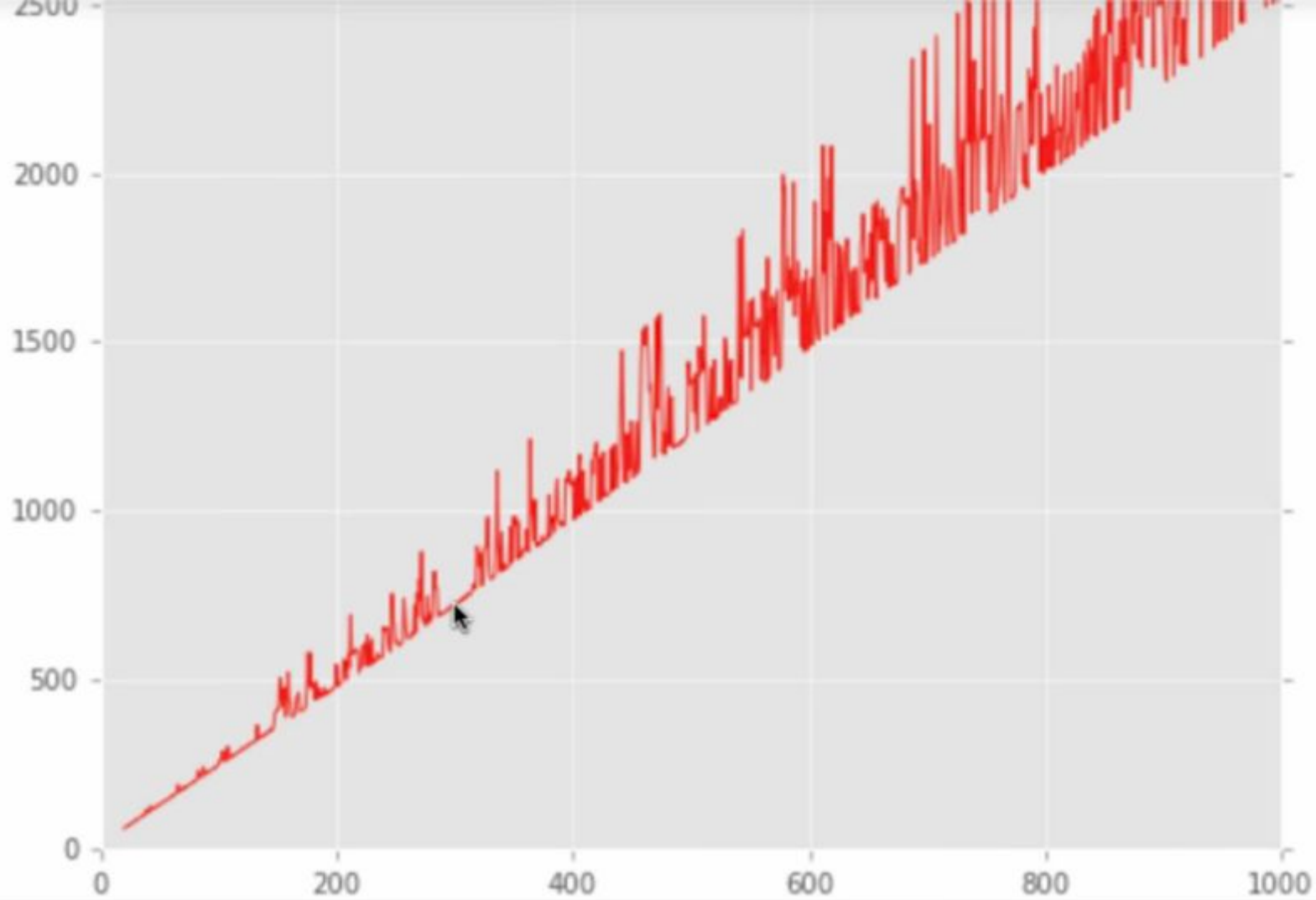
Time

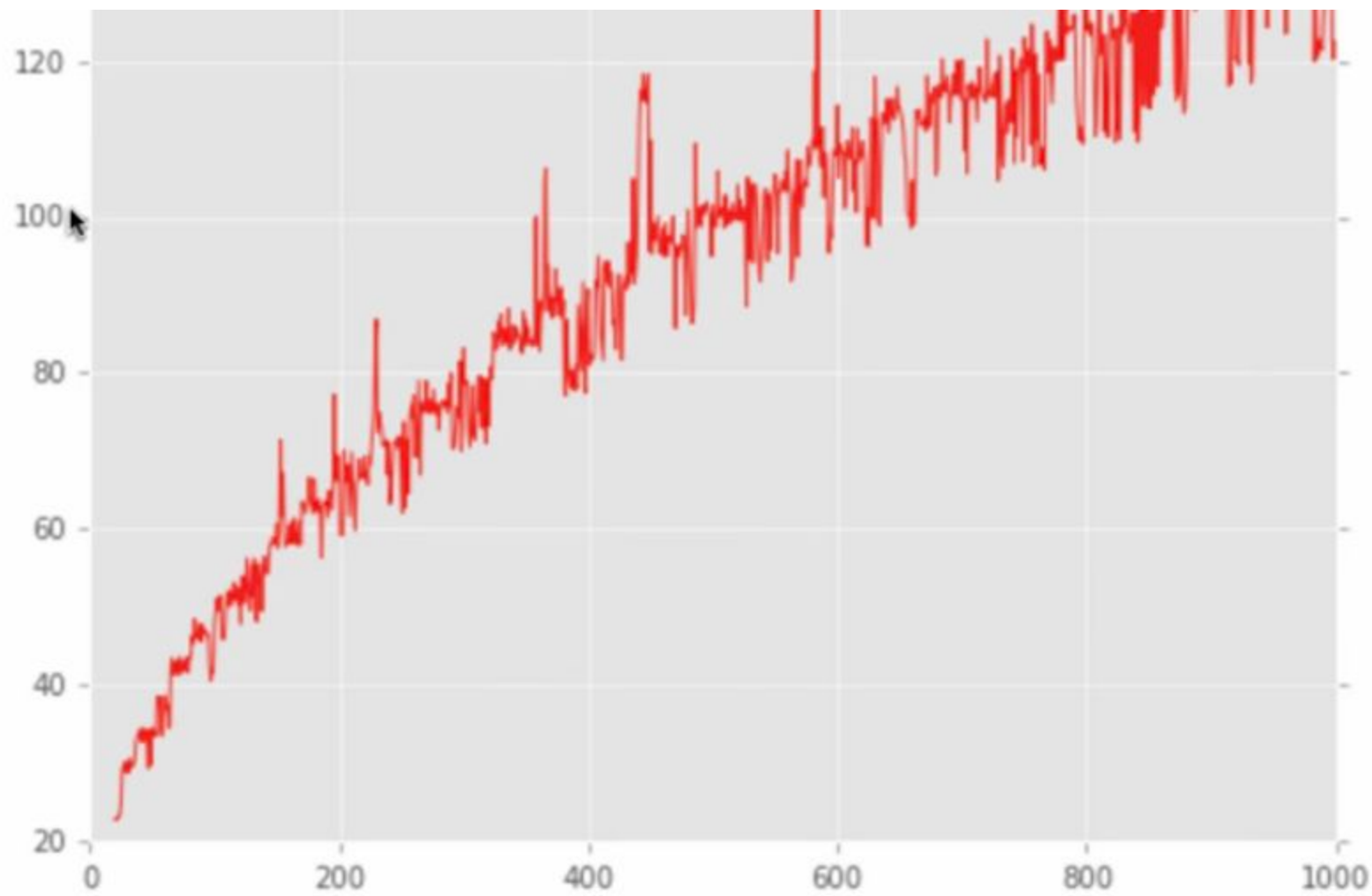
Slow: Test each k from 1 through n

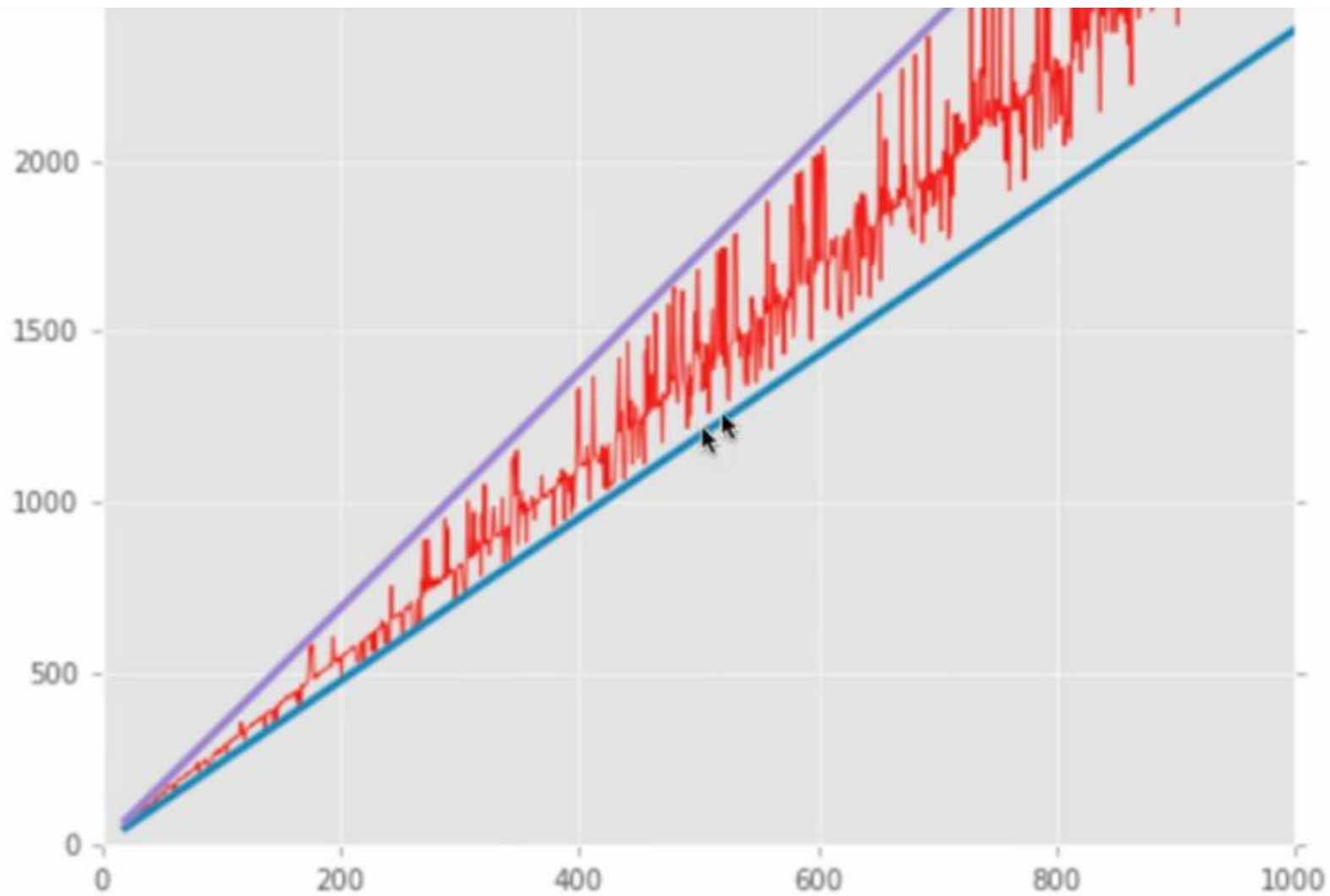
$$\Theta(n)$$

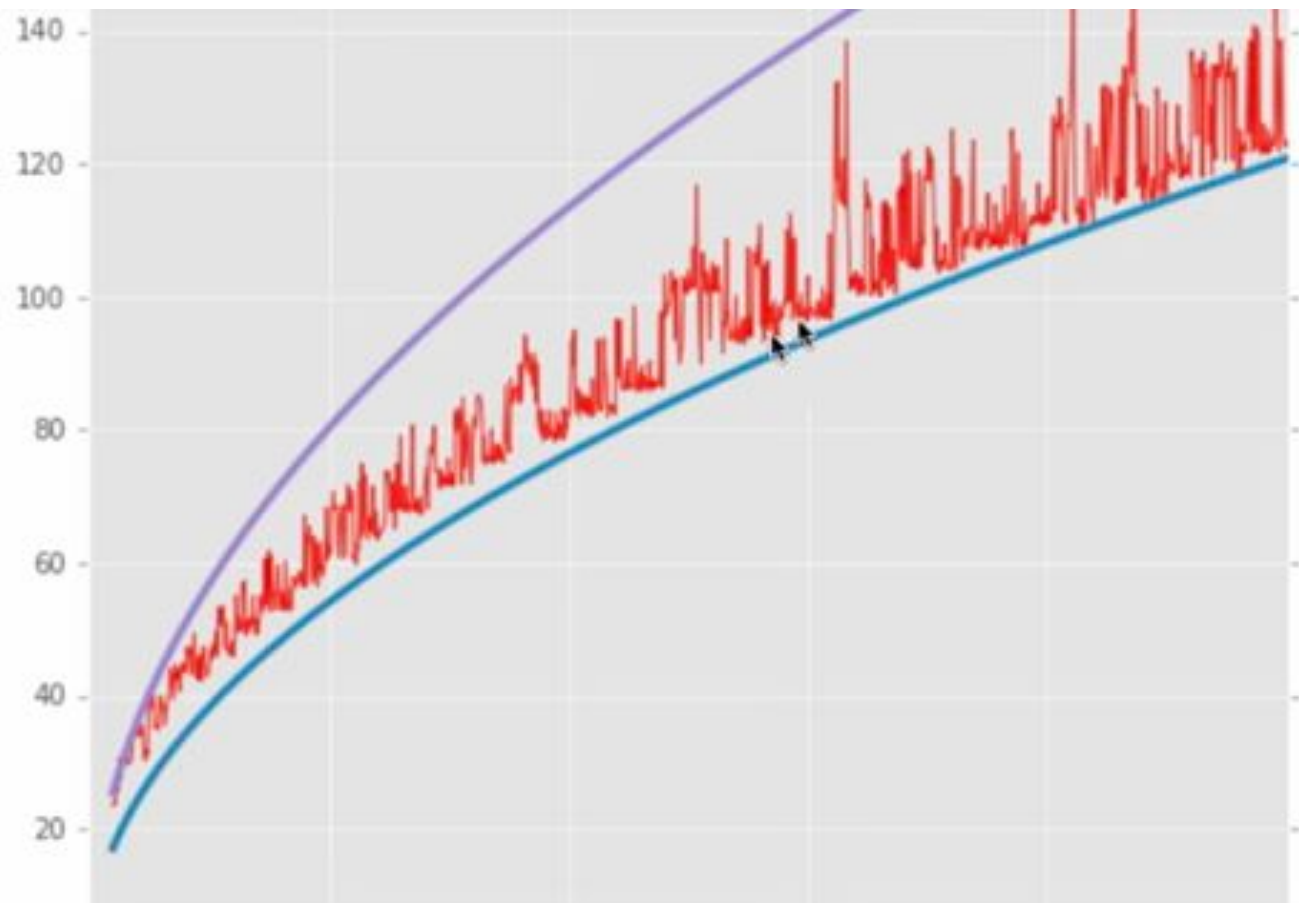
Fast: Test each k from 1 to square root n
For every k , n/k is also a factor!

$$\Theta(\sqrt{n})$$









Practice (board)



Comparing Orders of Growth



Properties of Orders of Growth

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

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

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

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

- **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)

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

$\Theta(\log n)$ Logarithmic growth.

$\Theta(\sqrt{n})$ Square root growth.

$\Theta(n)$ Linear growth.

$\Theta(n^2)$ Quadratic growth.

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

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



Practice

What is the growth rate?

Let $R(n) = 100000$.

What is the growth rate?

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$

What is the growth rate?

Let $R(n) = 100000n + n^2$

What is the growth rate?

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$

What is the growth rate?

Let $R(n) = 100000n + n^2 + n \log n$

What is the growth rate?

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$



What is the growth rate?

Let $R(n) = (1000000n + n^2 + n \log n) / n$

What is the growth rate?

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$

What is the growth rate?

```
sum = 0
for i in range(1, n+1):
    sum = sum * i
print(i)
```

A: $\Theta(n)$

B: $\Theta(\sqrt{n})$

C: $\Theta(1)$

D: $\Theta(n^2)$

E: More than one possible option

What is the growth rate?

```
sum = 0
for i in range(1, n+1):
    if (i%3 == 0):
        sum = sum * i
    print(i)
```

A: $\Theta(n)$

B: $\Theta(1)$

C: $\Theta(\log n)$

C: $\Theta(n^2)$

E: Impossible to calculate

What is the growth rate?

```
sum = 0
for i in range(1, n+1):
    for j in range(1, n+1):
        sum = sum * i + j
```

A: $\Theta(n^2)$

B: $\Theta(2^n)$

C: $\Theta(n)$

D: $\Theta(\log n)$

E: More than one possible option

Useful formula

$1 + 2 + 3 + 4 + \dots + n =$

$$\sum_{k=1}^n k = \frac{n(n+1)}{2},$$

What is the growth rate?

```
sum = 0
```

```
for i in range(1, n+1):
```

```
    j = 1
```

```
    while j <= i:
```

```
        sum = sum * i + j
```

```
        j = j + 1
```

A: $\Theta(n^2)$

B: $\Theta(1)$

C: $\Theta(n)$

D: $\Theta(\log n)$

E: More than one possible option

What is the growth rate?

```
sum = 0
j = n
while j > 0:
    print(j)
    j = j // 2
```

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$



What is the growth rate?

```
sum = 0
for i in range(1, n+1):
    sum = sum * i
for j in range(1, n+1):
    sum = sum + j
```

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$

What is the growth rate?

```
sum = 0
j = 1
for i in range(1, n+1):

    while j <= i:
        sum = sum * i + j
        j = j + 1
```

A: $\Theta(n^2)$

B: $\Theta(1)$

C: $\Theta(n)$

D: $\Theta(\log n)$

E: More than one possible option



What is the growth rate?

```
sum = 0
j = n
while j <= n:
    print(j)
    j = j*2
```

A: $\Theta(1)$

B: $\Theta(n)$

C: $\Theta(\log n)$

D: $\Theta(n \log n)$

E: $\Theta(n^2)$