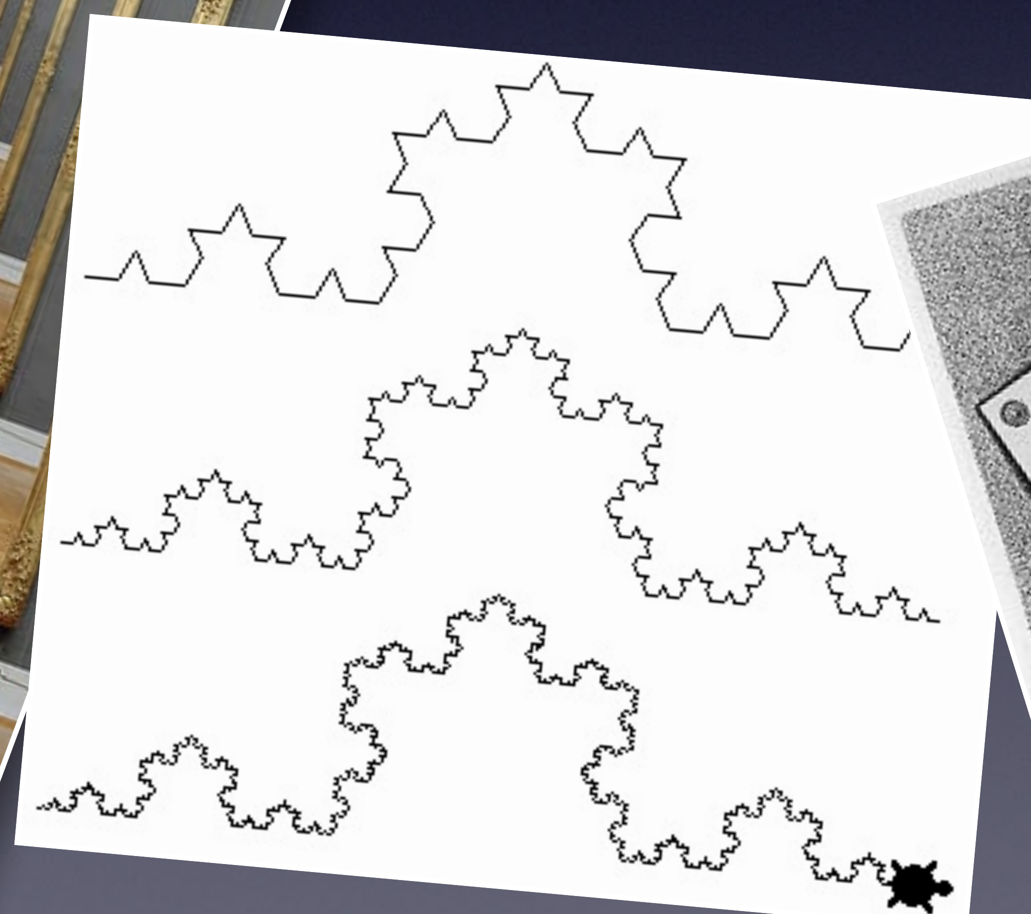
Recursion



code.google.com/p/rzzzwilson/talks/recursion

Recursion

It has been said:

> To understand recursion,
> one must first understand recursion.

Recursion

It has been said:

> To understand recursion,
> one must first understand recursion.

It's actually true!

Recursion

It has been said:

> To understand recursion,
> one must first understand recursion.

It's actually true!

> You write a recursive solution as if
> you already have a solution.

Recursion

It has been said:

> To understand recursion,
> one must first understand recursion.

It's actually true!

> You write a recursive solution as if
> you already have a solution.

It's programming by faith!

Recursion

A solution is recursive if it:

- solves a problem with smaller recursive cases
- has one or more non-recursive base cases

Recursion

A solution is recursive if it:

- solves a problem with smaller recursive cases
- has one or more non-recursive base cases

An example: the Factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

Recursion

A solution is recursive if it:

- solves a problem with smaller recursive cases
- has one or more non-recursive base cases

An example: the Factorial function:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0. \end{cases}$$

recursive case

Recursion

A program is recursive if it:

- solves a problem with smaller recursive cases
- has one or more non-recursive base cases

An example: the Factorial function:

$$n! = \begin{cases} \boxed{1 \qquad \text{if } n = 0,} \\ (n-1)! \times n \qquad \text{if } n > 0. \end{cases}$$

base case

Recursion

The "Tower of Hanoi" puzzle was invented by Édouard Lucas in 1883:

Recursion

How do we solve the puzzle recursively?
The general case is that we want to move N disks from the source to target position.

Recursion

How do we solve the puzzle recursively? The general case is that we want to move N disks from the source to target position.

- Base case (N = 1)

  We know how to move one disk.

Recursion

How do we solve the puzzle recursively? The general case is that we want to move N disks from the source to target position.

- Base case (N = 1)

  We know how to move one disk.

- Recursive case

  We can move the top N-1 disks to the other position, then the single largest disk to the target position, then move the N-1 disks to the target.

Recursion

Let's write some python:

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print('move %s to %s' % (src, dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```

Recursion

Let's write some python:

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print('move %s to %s' % (src, dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```

Recursion

Let's write some python:

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print('move %s to %s' % (src, dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```

Recursion

Let's write some python:

```
def hanoi(n, src, dst, tmp):
    if n == 1:
        print('move %s to %s' % (src, dst))
    else:
        hanoi(n-1, src, tmp, dst)
        hanoi(1, src, dst, tmp)
        hanoi(n-1, tmp, dst, src)
```

This solution is easy to split into base and recursive cases. It fails if N is less than zero.

Recursion

A more robust and more compact solution:

```
def hanoi(n, src, dst, tmp):
    """Move n disks from src to dst"""

    if n > 0:
        hanoi(n-1, src, tmp, dst)
        print('move %s to %s' % (src, dst))
        hanoi(n-1, tmp, dst, src)
```

More efficient than the first solution?

Recursion

```
import sys

def hanoi(n, src, dst, tmp):
    if n > 0:
        hanoi(n-1, src, tmp, dst)
        print('move %s to %s' % (src, dst))
        hanoi(n-1, tmp, dst, src)

number = int(sys.argv[1])
hanoi(number, 'A', 'B', 'C')
```

```
% ./hanoi 3
move A to B
move A to C
move B to C
move A to B
move C to A
move C to B
move A to B
```

Recursion

Why are recursive algorithms useful?

Recursion

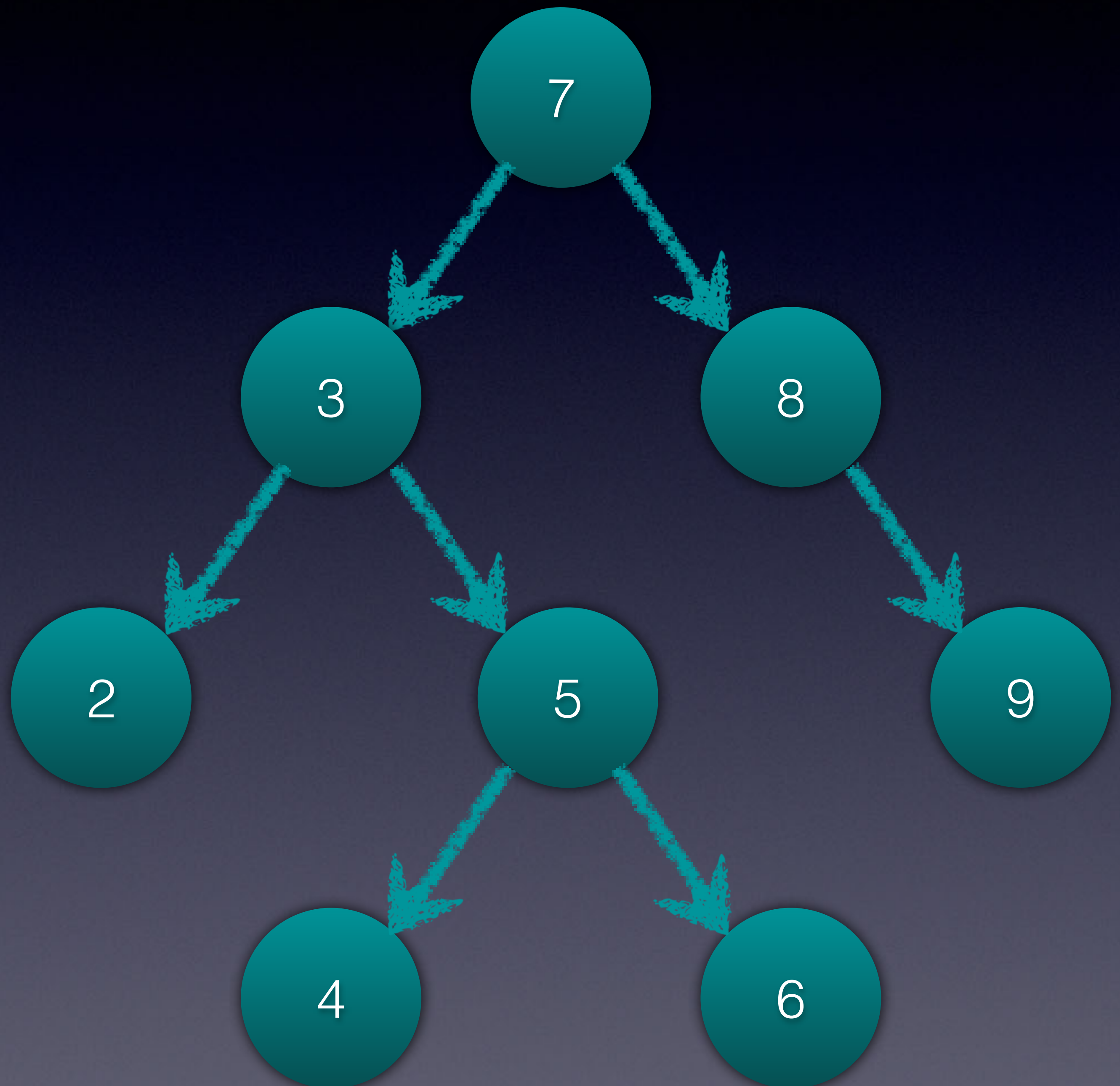
Why are recursive algorithms useful?

Some data structures are recursively defined, so recursive algorithms are a natural way to process them.

For example, binary trees.

Recursion

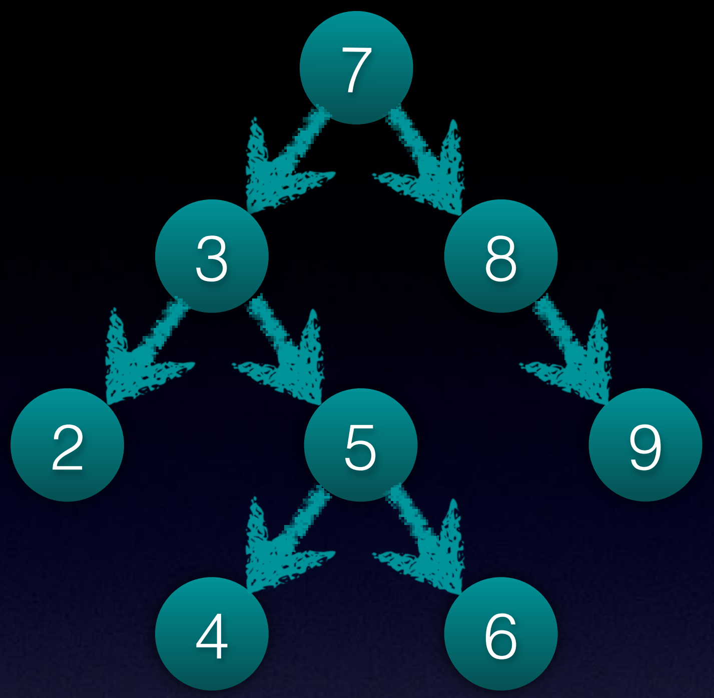
A binary tree looks like this:



A binary tree consists of a *node* which has a *value* and *left* and *right* pointers, which may be *None* or refer to another binary tree.

Recursion

Suppose we want to print out the values in the tree nodes.



One method is, for each node:

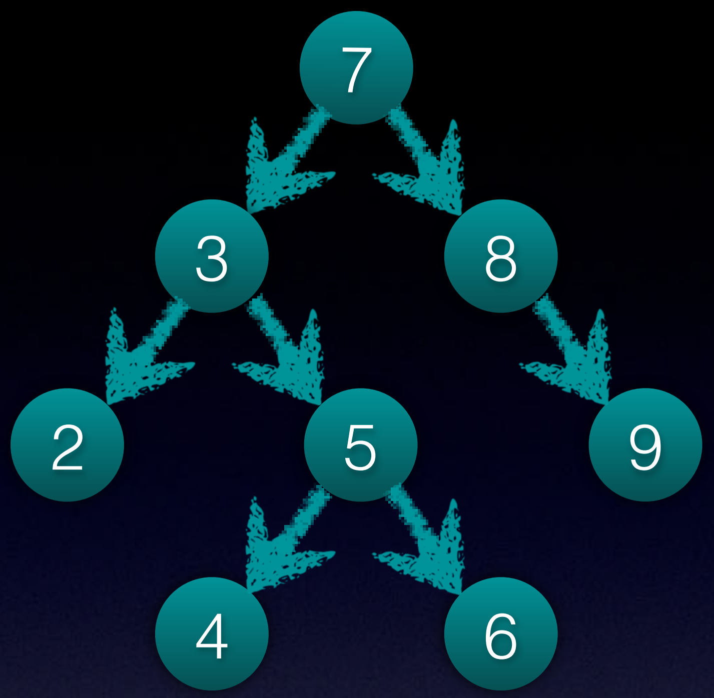
- print the left sub-tree
- print the node value
- print the right sub-tree

This is called an *in-order* walk of the tree. We print the node value in between the left and right sub-tree values.

Recursion

Let's define our tree:

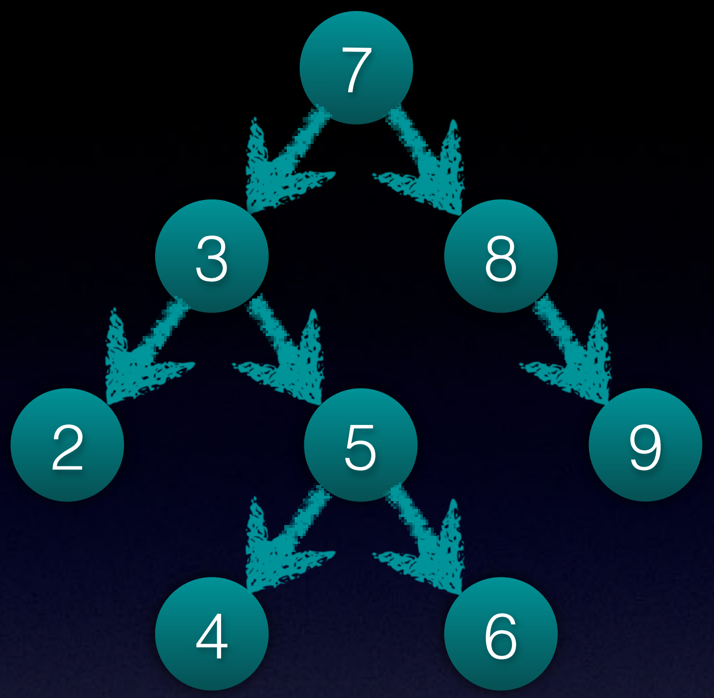
```
class Node(object):
    def __init__(self, value,
                 left=None, right=None):
        self.value = value
        self.left = left
        self.right = right

tree = Node(value=7)
tree.left = Node(value=3)
tree.left.left = Node(value=2)
tree.left.right = Node(value=5)
tree.left.right.left = Node(value=4)
tree.left.right.right = Node(value=6)
tree.right = Node(value=8)
tree.right.right = Node(value=9)
```



Recursion

The code to walk the tree is:

```
def walk_inorder(node):
    if node is not None:
        walk_inorder(node.left)
        print(node.value)
        walk_inorder(node.right)
```

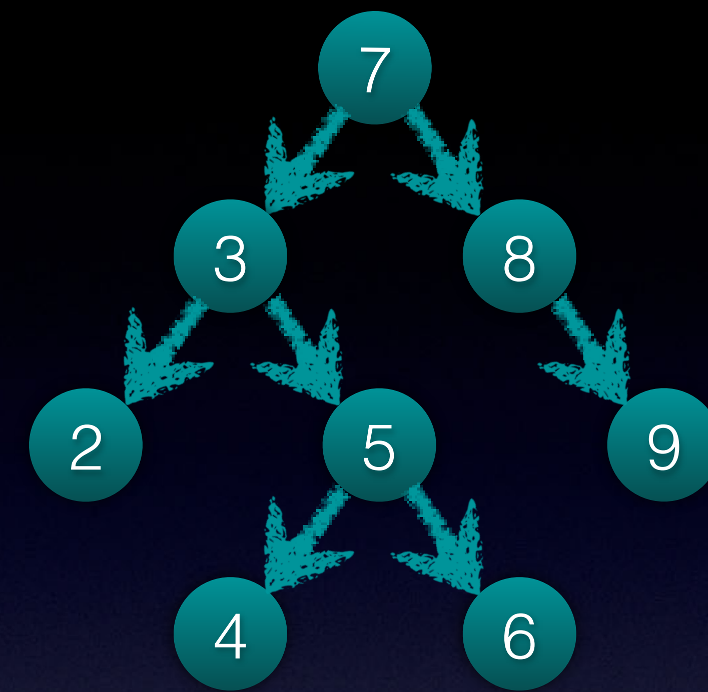


Executing this function on the example tree:

```
2 3 4 5 6 7 8 9
```

Recursion

There are two other ways to walk through a binary tree:




```
def walk_preorder(node):
    if node is not None:
        print(node.value)
        walk_inorder(node.left)
        walk_inorder(node.right)

def walk_postorder(node):
    if node is not None:
        walk_inorder(node.left)
        walk_inorder(node.right)
        print(node.value)
```

Executing these functions on the example tree:

```
 7  3  2  5  4  6  8  9     # preorder
 2  4  6  5  3  9  8  7     # postorder
```

Recursion

What are the costs of a recursive solution?

- every nested function call costs memory
- every function call costs time

When the costs of using recursion are too high we might use an iterative algorithm. There are algorithms to iteratively walk through a binary tree (eg, Shorr Waite), but they are not as simple as the recursive approach.

It is possible to combine recursion with various techniques to achieve an efficient solution.

Recursion

The Fibonacci function:

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

Recursion

The Fibonacci function:

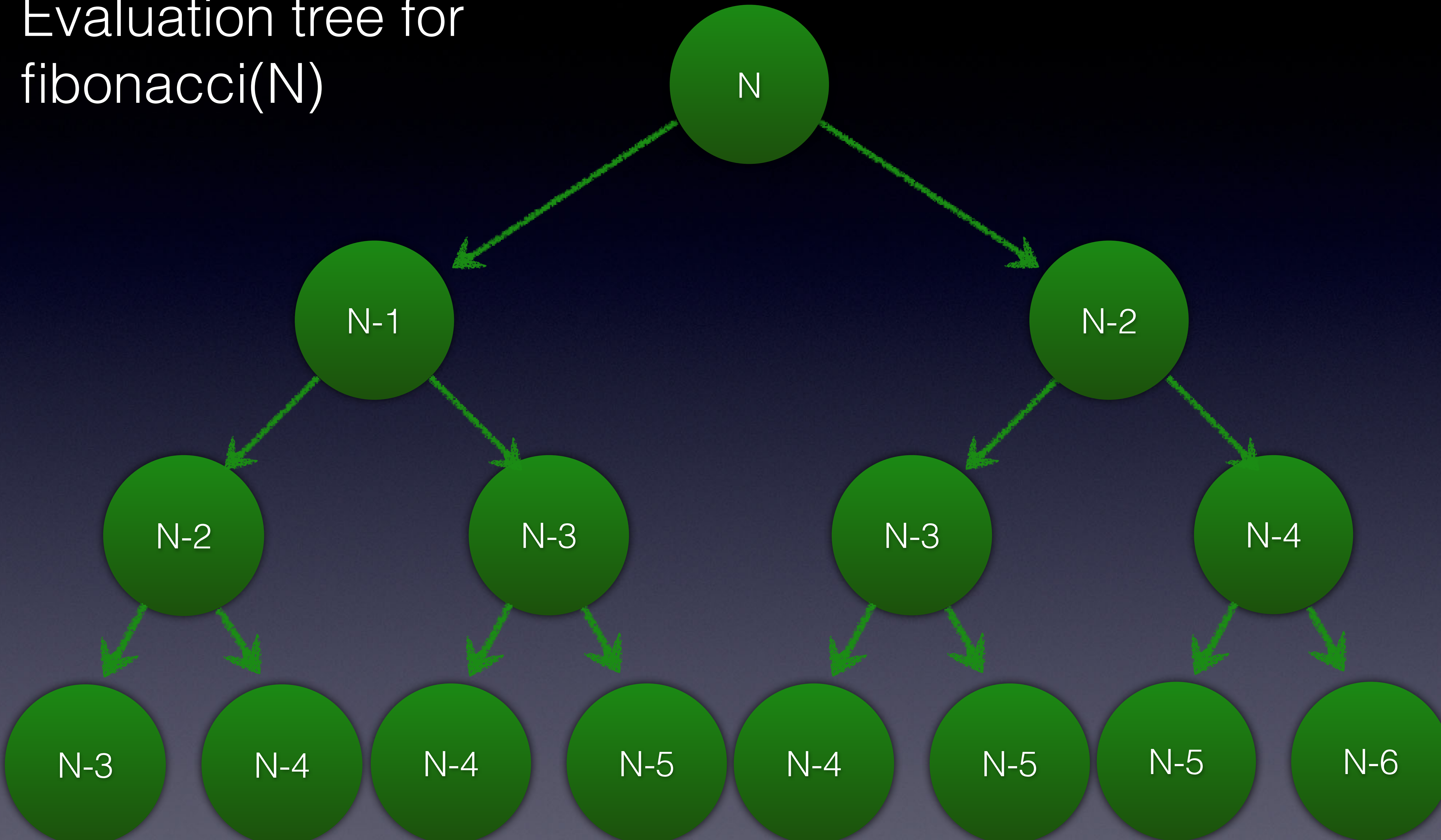
```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

This naive Fibonacci function is very inefficient. Try evaluating `fibonacci(40)`. It's really slow!

```
     fibonacci(40)=102334155  took 57.7889390s
```

Recursion

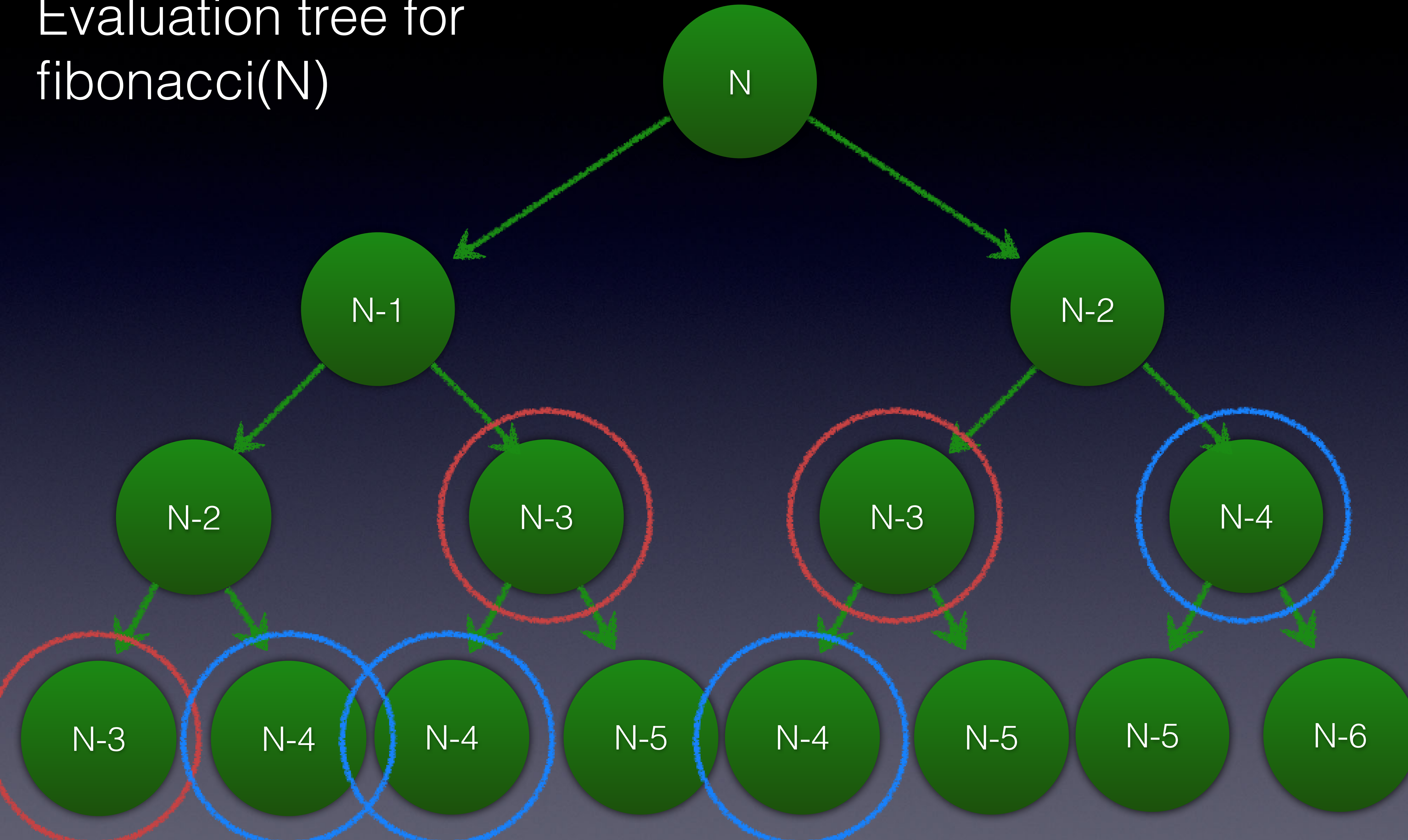
Evaluation tree for fibonacci(N)



code.google.com/p/rzzzwilson/talks/recursion

Recursion

Evaluation tree for fibonacci(N)



Recursion

If we could evaluate `fibonacci(m)` once and use the remembered result, we would save time.

```
fib_memo = {0: 0, 1: 1}   # base cases

def fibonacci_memo(n):
    global fib_memo  # so we can update fib_memo
    if n not in fib_memo:
        fib_memo[n] = (fibonacci_memo(n-1) +
                       fibonacci_memo(n-2))
    return fib_memo[n]
```

Recursion

If we evaluate `fibonacci(m)` once and use the remembered result, we save time.

```
fib_memo = {0: 0, 1: 1}    # base cases

def fibonacci_memo(n):
    global fib_memo  # so we can update fib_memo
    if n not in fib_memo:
        fib_memo[n] = (fibonacci_memo(n-1) +
                       fibonacci_memo(n-2))
    return fib_memo[n]
```

```
     fibonacci(40)=102334155  took 57.788939s
fibonacci_memo(40)=102334155  took  0.000025s
```

A useful speedup - more than 2,000,000 times faster!

Recursion

This approach is called memoisation. It's part of what is called dynamic programming.

There are other ways to memoise using decorators, but that's another talk.

Recursion

?