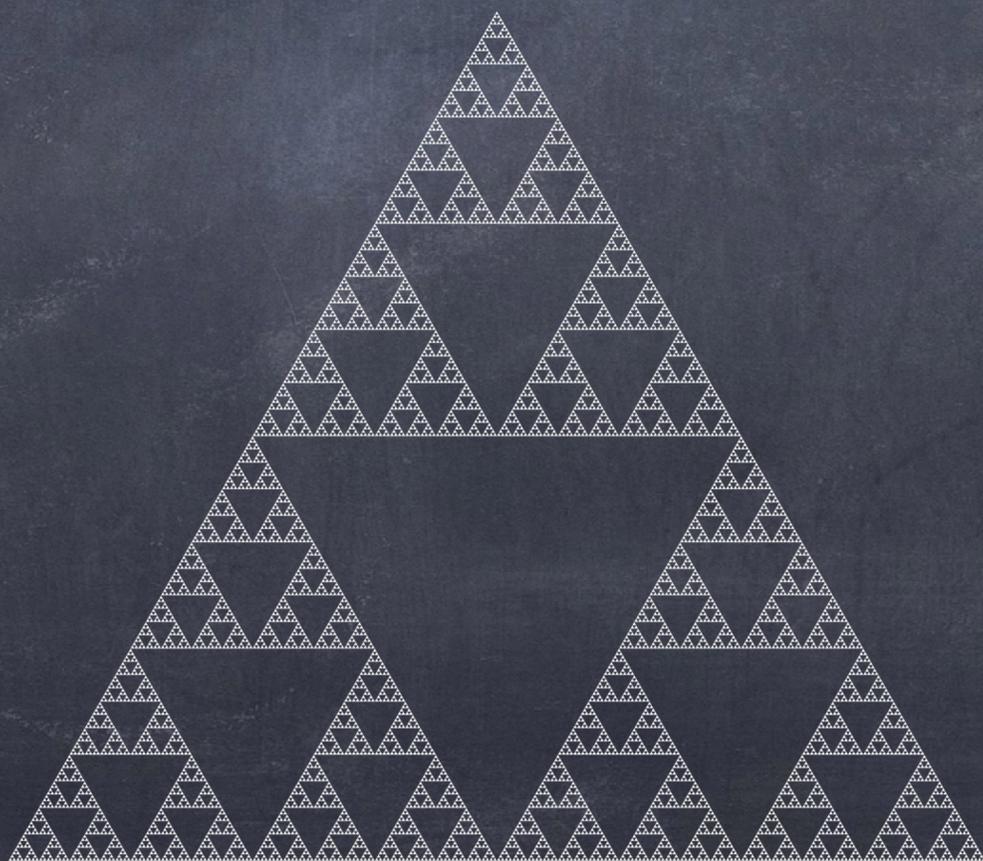


递归



排队位置问题

- 假如你在排一个长队买票，你想知道你前面有多少人，因为只剩下100张票了。
- 你不能离开队伍，然后进行统计（因为回来时你得重排），该怎么办？



排队位置问题

- ① 一种迭代的做法可以这么做：
 - ① 1. 找一个不在队伍的人走到队伍前面
 - ② 2. 然后挨个数
 - ③ 3. 直到数到自己停下，告诉数的答案即可

排队位置问题

① 一个递归的做法如下：

② 1. 如果你在队伍的最前面，那么你就是第一个

③ 2. 否则，你需要问一下那个排在你前面一位的那个人，“你排在第几位？”

④ 3. 当你得到前面人的答案时，你只需要将其答案加一即可

⑤ 前面人怎么得到答案呢？做同样上述三个操作即可

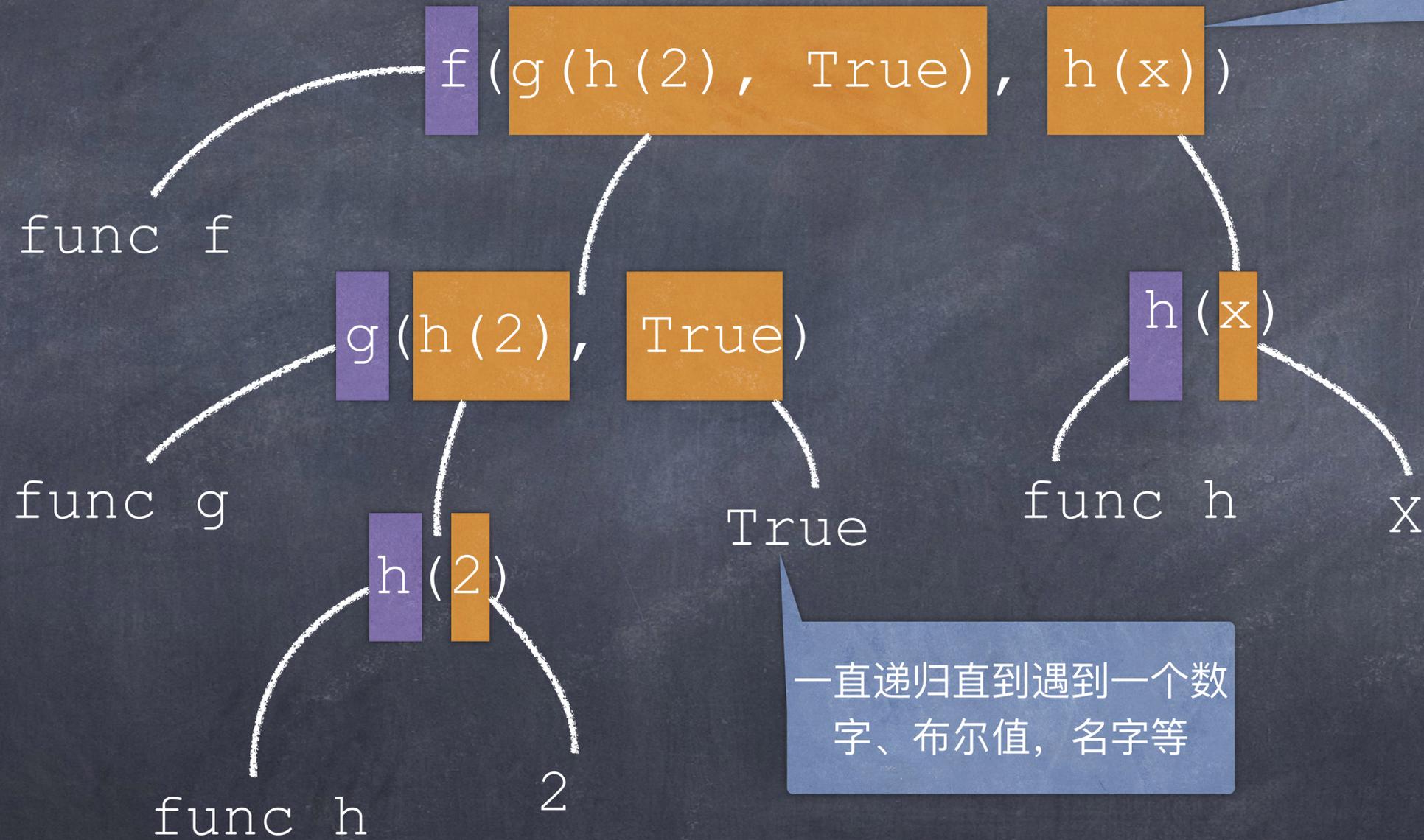
递归 (Recursion)

● 递归用来解决具有自我重复结构的问题

● 关键就是找到这样的重复结构，将问题分解为更小的问题，并且定义出最小的问题。

在求值里用递归

一个调用表达式包含了很多小的调用表达式



一直递归直到遇到一个数字、布尔值、名字等

递归函数 (Recursive Functions)

- ① 一个函数被称为递归函数如果在自己的函数体内直接或者间接调用自己。
- ② 这意味着执行一个递归函数的函数体往往会作用该函数多次。
- ③ 递归其本质就是一种函数抽象。

递归函数的结构

1

一个或多个基本情形（通常是在最小输入下）

◆ 比如：“如果你在最前面，那么你就是第一个”

2

一个或多个约减问题的方式，然后利用递归解决更小的问题

◆ 比如：“问你排在前面一位的人，‘你是第几个？’”

3

一个或多个使用更小问题的解来解决更大问题

◆ 比如：“当前面的人得到其结果，只要加一就是你的排名”

函数抽象 & 递归

表达式

fact (1)

fact (3)

fact (4)

fact (n-1)

fact (n)

值

1

6 (3*2*1)

24 (4*3*2*1)

(n-1) * (n-2) * ... 1

~~n * (n-1) * (n-2) * ... 1~~



n * fact (n-1)

函数抽象 & 递归

```
1 def fact(n):  
2     """Calculates n!"""  
3     if n == 0:  
4         return 1  
5     else:  
6         return n * fact(n-1)
```

base case

smaller problems

solve larger problems

“看”递归

在环境图中的递归

```
1 def fact(n):  
2     if n == 0:  
3         return 1  
4     else:  
5         return n * fact(n-1)  
6  
7 fact(3)
```

函数 `fact` 被多次调用，每次调用都是解决一个更加简单的问题

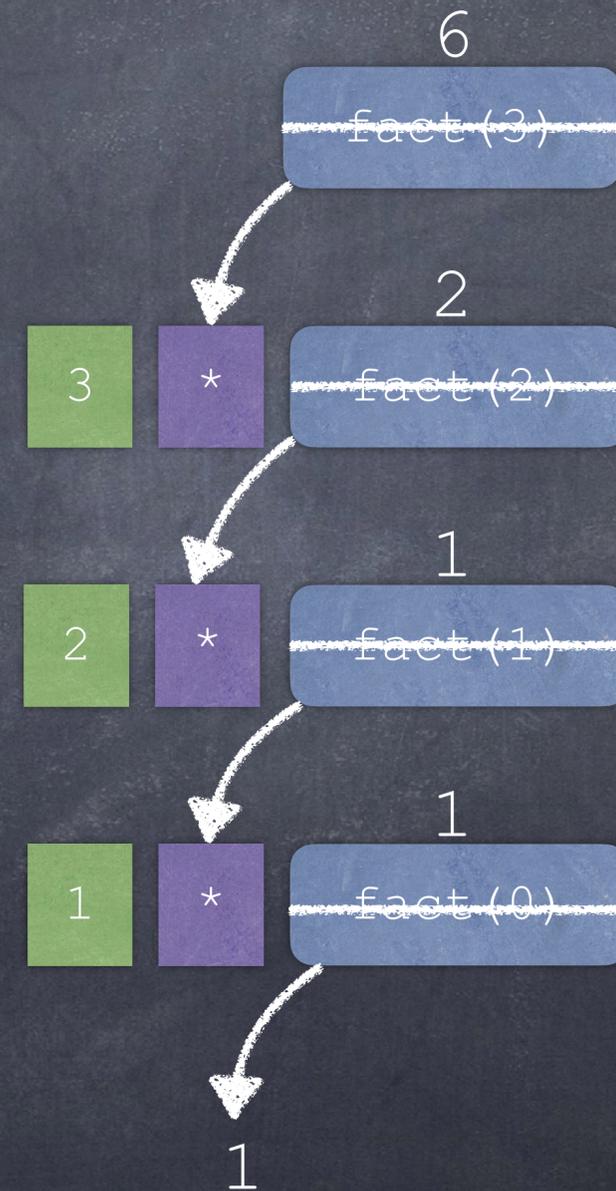
每个帧的创建都是一样，只有实参不同

`n` 的求值取决于当前的环境



另一种可视角度：递归树

```
1 def fact(n):  
2     """Calculates n!"""  
3     if n == 0:  
4         return 1  
5     else:  
6         return n * fact(n-1)
```



例子

Count up

实现一个递归函数来打印1到n。这里假设n是正数

```
def count_up(n):  
    """Prints the numbers from 1 to n.
```

```
>>> count_up(1)
```

```
1
```

```
>>> count_up(2)
```

```
1
```

```
2
```

```
>>> count_up(4)
```

```
1
```

```
2
```

```
3
```

```
4
```

```
"""
```

```
*** YOUR CODE HERE ***"
```

◆ 一个或多个基本情形

◆ 一个或多个使用更简单的实参进行递归调用

◆ 利用一个或多个递归的返回值来解决更大的问题

Count up

基本情况：

- 最小的值（也就是在该值上我们不再需要做更多过程）是多少？

- n 是正数，因此最小的正整数是 1。当数为 1 时，我们只需要打印即可，无需更多工作

递归更小实参：

- 处理更大的数字这个任务需要处理比其更小的数字

利用递归的解来解决更大问题：

- 一旦我们打印到 $n - 1$ ，还有什么需要做？ 打印自身

Sum Digits

实现一个递归函数可以把一个正整数 n (10进制) 的所包含的所有数字加起来。

```
def sum_digits(n):  
    """Calculates the sum of the digits `n`.  
>>> sum_digits(9)  
9  
>>> sum_digits(19)  
10  
>>> sum_digits(2023)  
7  
"""  
""" YOUR CODE HERE """
```

◆ 一个或多个基本情形

◆ 一个或多个使用更简单的实参进行递归调用

◆ 利用一个或多个递归的返回值来解决更大的问题

Sum Digits

① 输入：正整数

② 输出：所包含的所有数字之和

③ 基本情形：

④ 单个数字

⑤ 更小问题：

⑥ 除了一位数字（最小的一位），所包含的其他所有数字之和

⑦ 更大问题：

⑧ 之前更小问题的解加上剩下的这一位数字。

Cascade

实现一个可以打印一个正整数 n 的级联树

```
def cascade(n):  
    """Calculates the cascade tree of `n`.  
>>> cascade(486)  
486  
48  
4  
48  
486  
>>> sum_digits(48)  
48  
4  
48  
>>> sum_digits(4)  
4  
"""  
""" YOUR CODE HERE """
```

◆ 一个或多个基本情形

◆ 一个或多个使用更简单的实参进行递归调用

◆ 利用一个或多个递归的返回值来解决更大的问题

Cascade

- ◎ 基本情况：
 - ◎ 如果 n 是一个单个数字，直接打印出来
- ◎ 递归更小实参：
 - ◎ 处理 n 这个任务需要处理 $n//10$
- ◎ 利用递归的解来解决更大问题：
 - ◎ 将 $n//10$ 的结果放在中间，周围 `print(n)`

Cascade

```
1 def cascade(n):  
2     if n < 10:  
3         print(n)  
4     else:  
5         print(n)  
6         cascade(n // 10)  
7         print(n)  
8  
9 cascade(123)
```

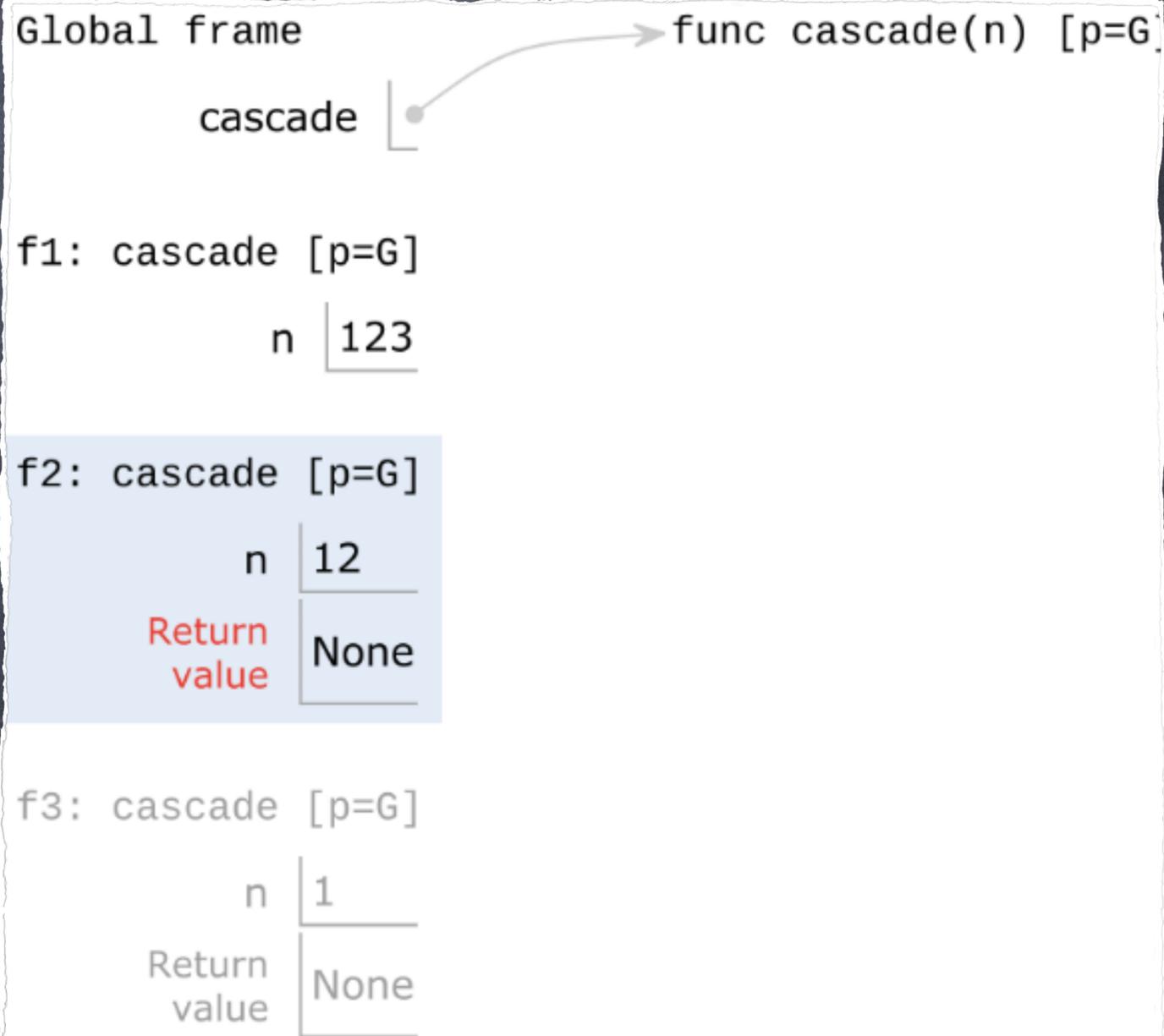
Output

123

12

1

12



每一个cascade帧都是由一个不同cascade函数调用产生的。

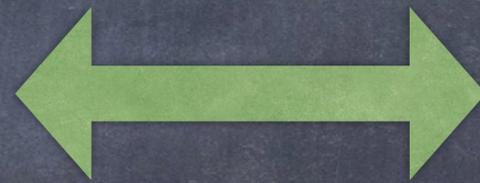
直到return语句出现前，这个调用都没有终止（因此，帧还是存在，里面的names绑定也存在）。

在递归的函数调用语句前和后都可以出现其它语句。

Cascade

两种实现

```
1 def cascade(n):  
2     if n < 10:  
3         print(n)  
4     else:  
5         print(n)  
6         cascade(n // 10)  
7         print(n)
```



```
1 def cascade(n):  
2     print(n)  
3     if n >= 10:  
4         cascade(n // 10)  
5         print(n)
```

- 如果两种实现是等价的，通常越短的实现越好 **why?**
- 在这个例子中，长一点的更加有助于理解
- 对于初学递归者而言，先把base case写出来是一种比较好的策略

迭代和递归

迭代和递归是紧密相关的

迭代

```
def fact_iter(n):  
    total, k = 1, 1  
    while k <= n:  
        total, k = total*k, k+1  
    return total
```

$$n! = \prod_{k=1}^n k$$

Names: n, total, k, fact_iter

递归

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

$$n! = \begin{cases} 1 & \text{if } n = 0 \\ n * (n-1)! & \text{otherwise} \end{cases}$$

Names: n, fact

将递归转化为迭代

👁️ 关键点：找出什么状态需要被追踪

```
def sum_digits(n):  
    """return the sum of the digits of `n`."""  
    if n < 10:  
        return n  
    else:  
        all_but_last, last = n // 10, n % 10  
        Return sum_digits(all_but_last) + last
```

What's left to sum

A partial sum

将迭代转化为递归

👁️ 关键点：迭代的状态可以作为参数传递

```
def sum_digits_iter(n):  
    """return the sum of the digits of `n`."""  
    digit_sum = 0  
    while n > 0:  
        n, last = n // 10, n % 10  
        digit_sum = digit_sum + last  
    return digit_sum  
  
def sum_digits_rec(n, digit_sum):  
    """return the sum of the digits of `n`."""  
    if n == 0:  
        return digit_sum  
    else:  
        n, last = n // 10, n % 10  
        return sum_digits_rec(n, digit_sum + last)
```

另一种转化

回顾不动点 (丘奇) :

$$Y = \lambda f. (\lambda x. f (x x)) (\lambda x. f (x x))$$

$$Y f = f(Y f)$$

如果用python写的话 :

```
Y = lambda f: (lambda x: f(x(x))) (lambda x: f(x(x)))
```

另一种转化

以 factorial 函数为例

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

用 lambda 函数来表示 :

```
F = lambda f: lambda n : 1 if n == 0 else n * f(n - 1)
```

```
fact = F(fact)
```

另一种转化

```
F = lambda f: lambda n : 1 if n == 0 else n * f(n - 1)
fact = F(fact)
```

因此，fact 是 F 的不动点

由于

```
Y = lambda f: (lambda x: f(x(x))) (lambda x: f(x(x)))
```

那么

```
fact = Y(F)
```

另一种转化

① python 过早求值函数

② 换一个组合子 : Z combinator

```
Z = (lambda f: (lambda x: f(lambda v: x(x)(v))))(lambda x: f(lambda v: x(x)(v)))
```

```
fact = Z(F)
```

验证递归的实现

这个阶乘实现的对吗？

1. 确认一下最基本的情形

他们正确吗？

他们足够吗？

2. 现在，使用函数抽象的性质

假设 $\text{factorial}(n-1)$ 是正确的

验证一下 $\text{factorial}(n)$ 是正确的

```
def fact(n):  
    if n == 0:  
        return 1  
    else:  
        return n * fact(n-1)
```

函数抽象：此处无需关心 $\text{fact}(n-1)$ 的实现，只需认为其可以正确给出 $\text{fact}(n-1)$ 的值。

互递归 (Mutual Recursion)

- 如果两个函数A和B，A调用了（间接或直接）B，B调用了（间接或直接）A，那么就称A和B是互递归的 (Mutual Recursive)。
- 与直接递归 (direct recursion) 直接调用自身不同，互递归是通过调用其他函数来间接的递归调用自己，因此也称为间接递归 (indirect recursion)

例子

① 判断一个数字是奇数还是偶数

① 0 是偶数

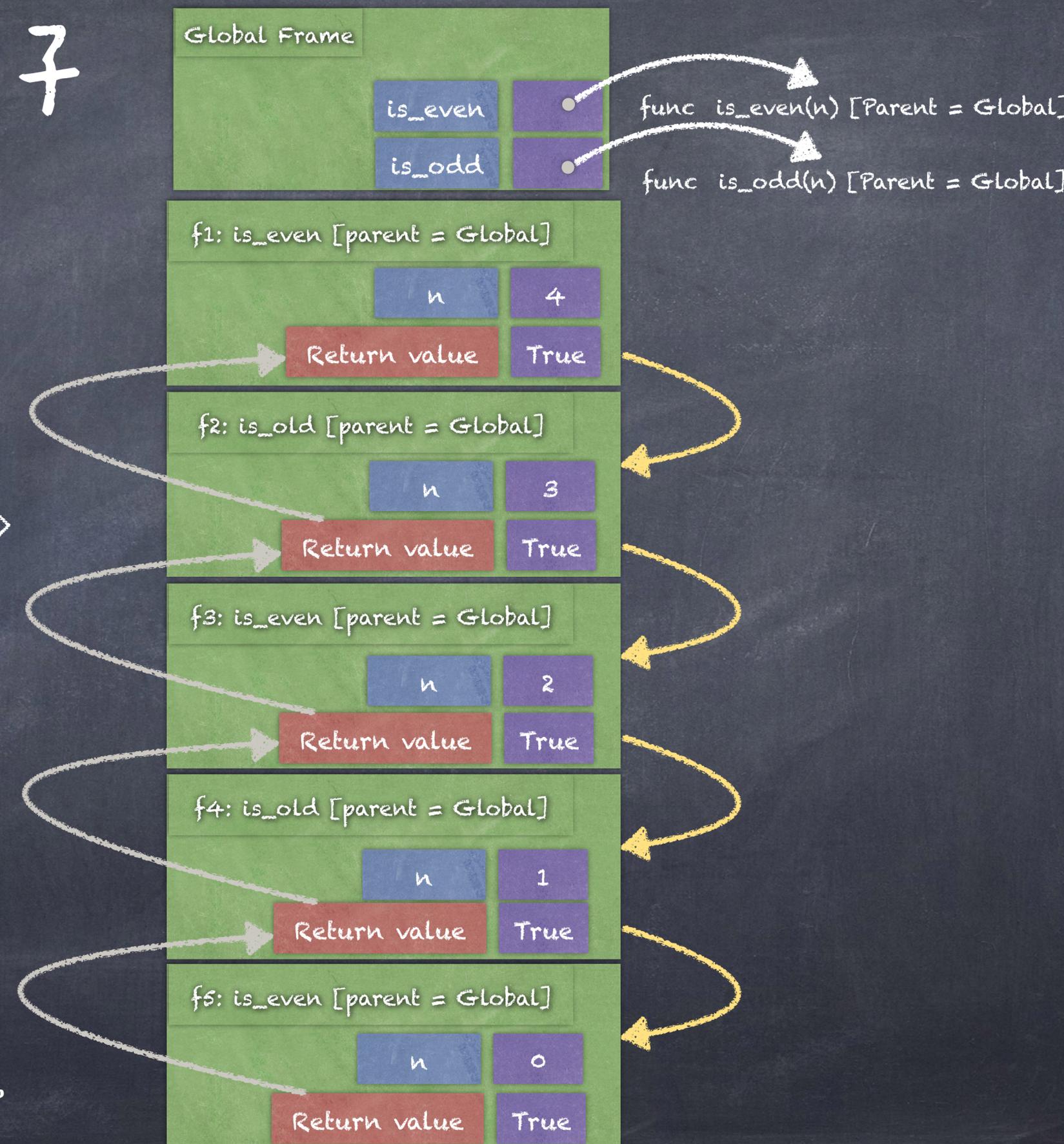
① 如果 $n-1$ 是奇数，那么 n 是偶数

① 如果 $n-1$ 是偶数，那么 n 是奇数

例子

```
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)
```

```
result = is_even(4)
```



可以看到互递归的两个函数的帧交替出现。

转化为直接递归

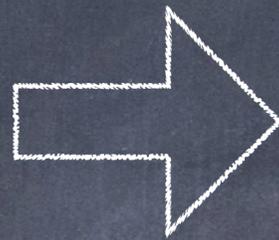
Why?

- ① 互递归的效率一般比直接递归要低
- ② 可以将互递归转化为直接递归
 - ③ 互递归可以通过将一个函数的代码内联进另一个函数中从而变为单个的递归函数。

Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. 1993. On the conversion of indirect to direct recursion.

转化为直接递归

```
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)
```



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

在本例中，将 `is_even(n)` 中 `else` 语句块里面的“`is_odd(n-1)`”使用 `is_odd` 函数具体实现进行替换（内联）。

转化为直接递归

- ① 这种转化（内联）虽然有效。但是如果互递归里包含了多个（重复）的调用，那么就要替换多次
- ② 造成的后果可能会造成代码膨胀，从而进一步限制了这种转化的有效性。

树形递归 (Tree Recursion)

- 一个递归函数如果在其函数体内多次调用自身，那么其就可以称其为树形递归函数

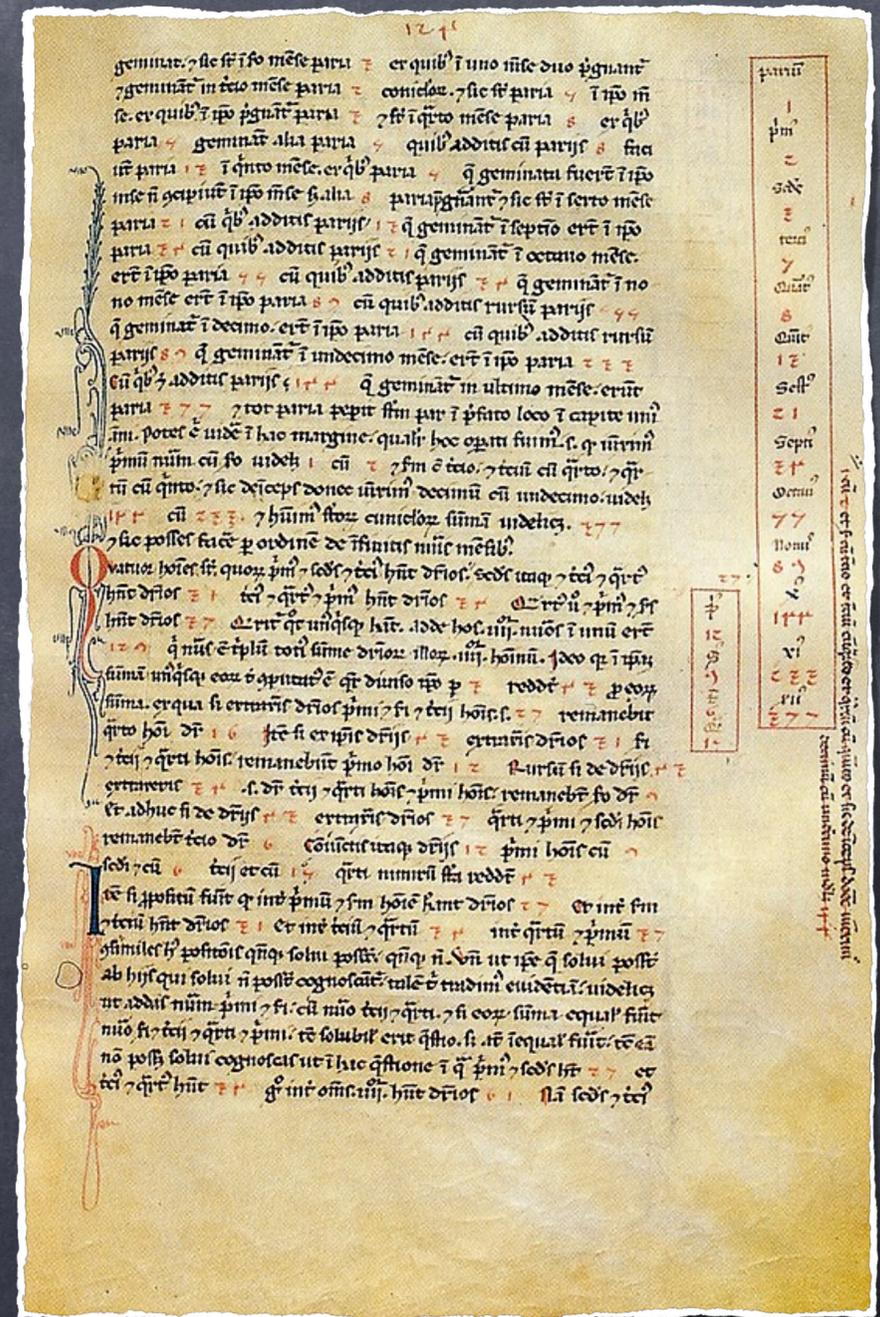
例子：斐波那契数列 (Fibonacci)

最早由印度人Virahanka发现，后由Fibonacci独立给出的数列：

• $F_0 = 0$

• $F_1 = 1$

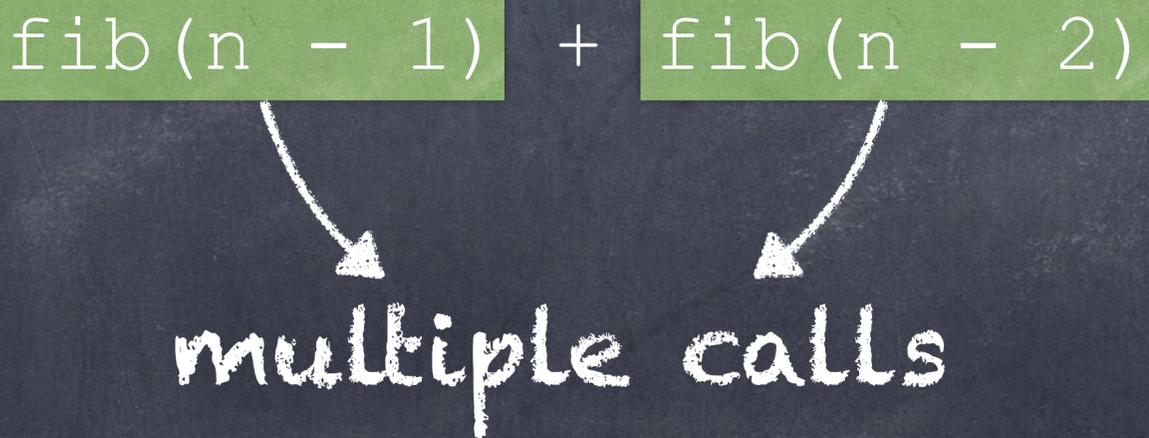
• $F_n = F_{n-1} + F_{n-2}$, where $n > 1$



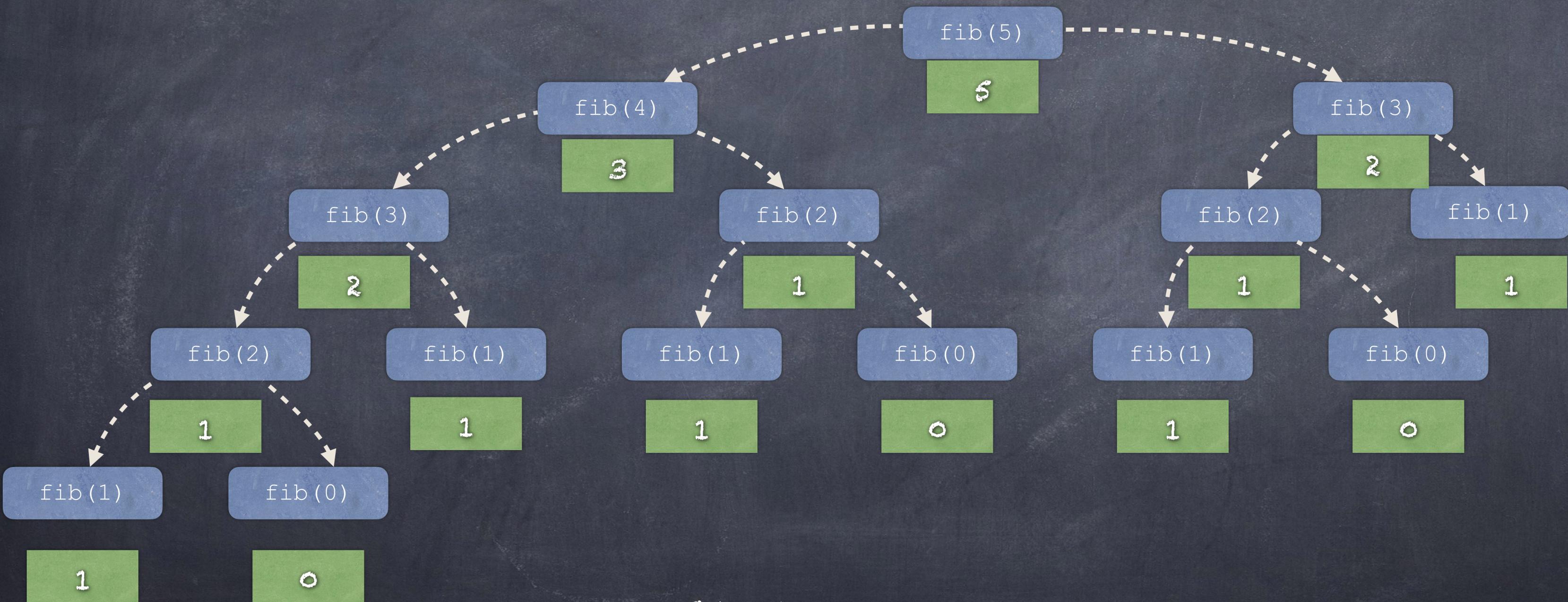
例子：斐波那契数列 (Fibonacci)

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

multiple calls



recursive tree of Fibonacci



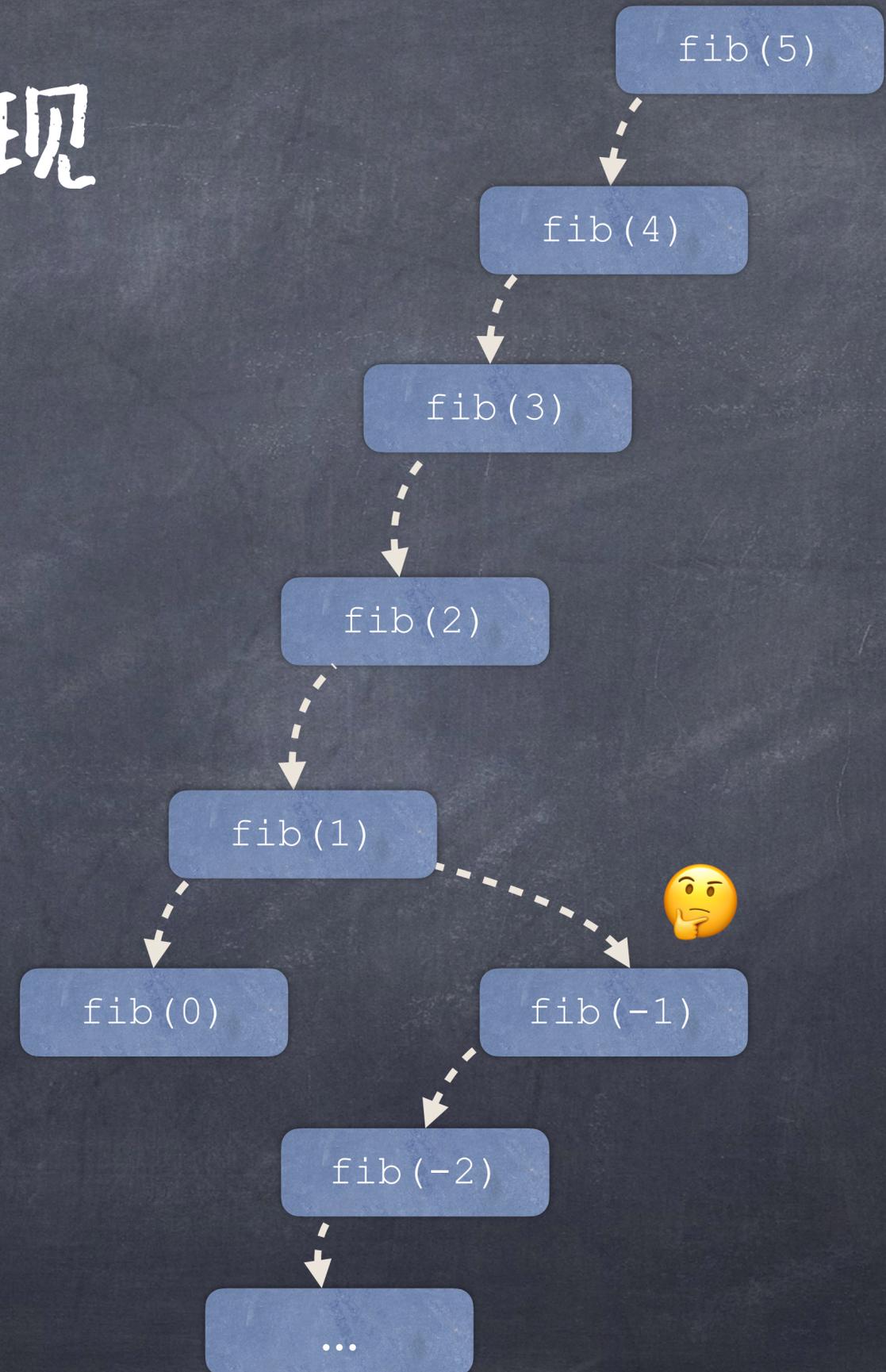
fib(n): a tree-recursive process

错误的实现

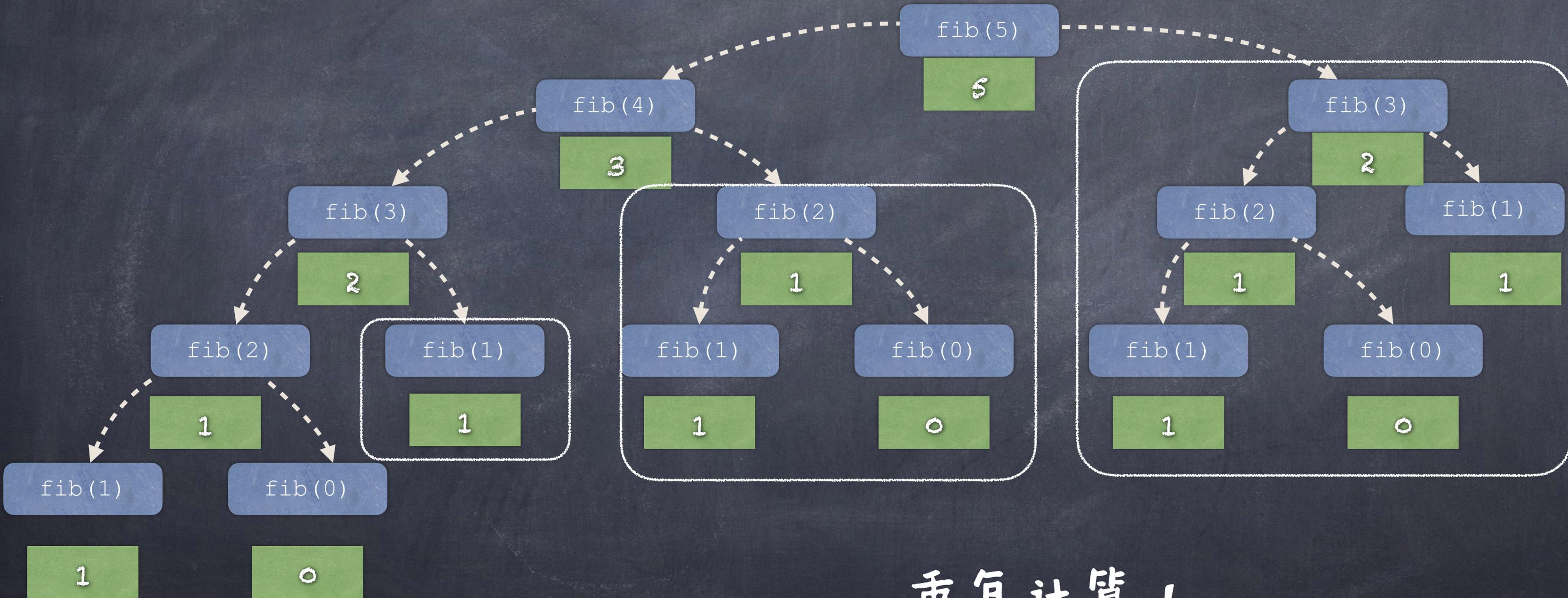
```
def fib(n):  
    if n == 0:  
        return 0  
    #missing base case!  
    else:  
        return fib(n - 1) + fib(n - 2)
```

```
>>> broken_fib(5)
```

```
Traceback (most recent call last):  
...  
RecursionError: maximum recursion  
depth exceeded in comparison
```



优化递归



重复计算!

优化递归

```
1 def better_fib(n):  
2     if n == 0:  
3         return 0  
4     elif n == 1:  
5         return 1  
6     elif already called better_fib(n):  
7         return stored value  
8     else:  
9         store & return better_fib(n - 2) + better_fib(n - 1)
```

复杂一点的例子：Count partitions

- 目标：对一个正整数 n 进行分割成多个数字，使得分割下来的数字加起来等于 n ，此外这些数字不能大于 m 。问：有多少种分割方式？

Count partitions

● 比如：现在要分割6，最大分割下来的数字不超过4，那么

```
>>> count_part(6, 4)
```

9

最大的分割数是4

$$4+2 = 6$$
$$4+1+1 = 6$$

最大的分割数是3

$$3+3 = 6$$
$$3+2+1 = 6$$
$$3+1+1+1 = 6$$

最大的分割数是2

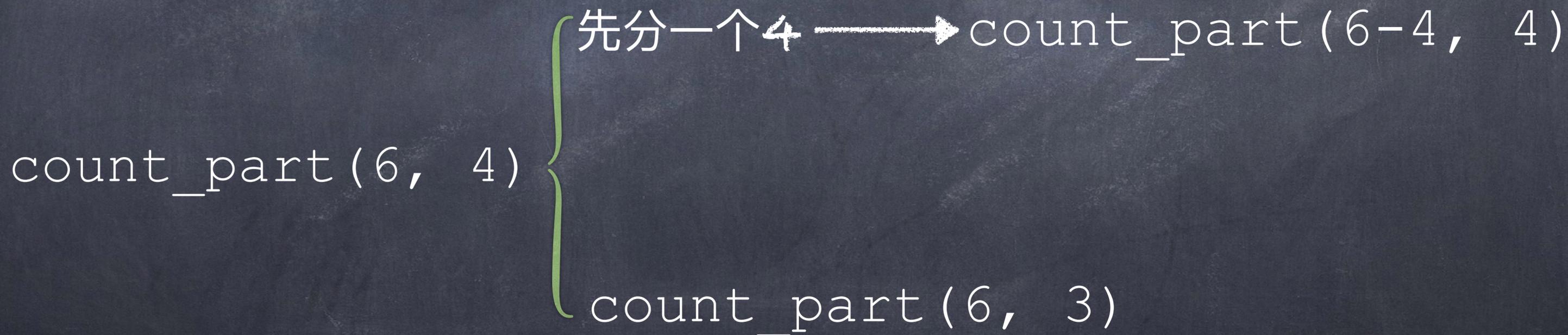
$$2+2+2 = 6$$
$$2+2+1+1 = 6$$
$$2+1+1+1+1 = 6$$

最大的分割数是1

$$1+1+1+1+1+1 = 6$$

Count partitions

• 对于一个大问题 $\text{count_part}(6, 4)$ 而言, 怎么把它分割为小问题?



Count partitions

`count_part(6, 4)` {
 先分一个4 → `count_part(6-4, 4)`
 `count_part(6, 3)`

The base case?

```
def count_part(n, m):  
    if  
  
    else:  
        with_m = count_part(n-m, m)  
        smaller_m = count_part(n, m - 1)  
        return with_m + smaller_m
```

negative

Use a 4

(6, 4)

Smaller than 4

(6, 3)

Use a 4

(6 - 4, 4)

Smaller than 4

(2, 3)

Use a 3

Smaller than 3

(2, 2)

Use a 2

Smaller than 2

(2, 1)

Use a 1

Smaller than 1

(2, 0)

negative

0

(2-3, 3)

(2-2, 2)

1

(2-1, 1)

Use a 1

Smaller than 1

(1, 0)

(1-1, 1)

1

Invalid! Largest piece is 0!

0

Invalid!

Valid partition

Use 4, use 2

$4 + 2 = 6$

Valid partition

Use 4, use 1, use 1

$4 + 1 + 1 = 6$

(2 - 4, 4)

0

0

Count partitions

```
def count_part(n, m):  
    if n == 0:  
        return 1 #valid partition  
    elif n < 0:  
        return 0 #cannot partition negative  
    elif m == 0:  
        return 0 #invalid partition if the largest piece is 0  
    else:  
        with_m = count_part(n-m, m)  
        smaller_m = count_part(n, m - 1)  
        return with_m + smaller_m
```

Any questions ?