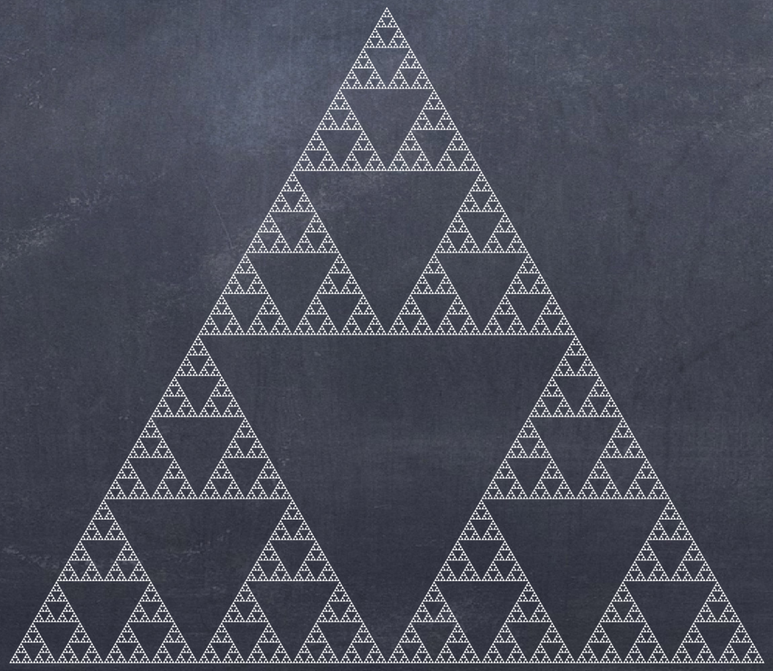
递归



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

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

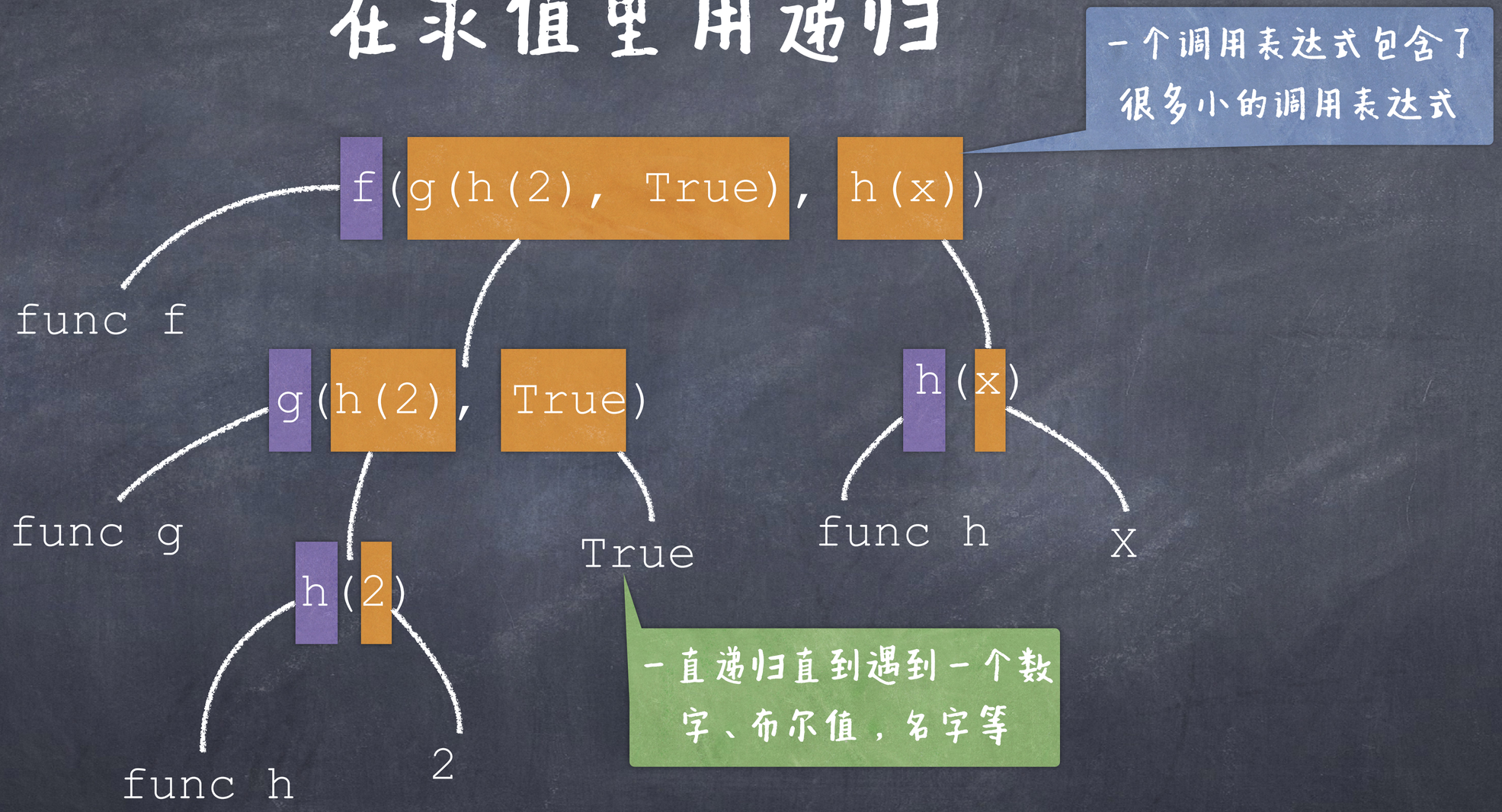
排队位置问题

- 一个递归的做法如下：
  - 1. 如果你在队伍的最前面，那么你就是第一个
  - 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？"
  - 3. 当你得到前面人的答案时，你只需要将其答案加一即可
- 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
f(g(h(2), True), h(x))
func f
g(h(2), True)
h(x)
func g
True
func h
X
h(2)
一直递归直到遇到一个数字、布尔值，名字等
func h
2

递归函数（Recursive Functions）

- 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。
- 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。
- 递归其本质就是一种函数抽象。

避免重复地（甚至是无限地）写同一个逻辑

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？'"（问题输入减少1个size）
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	值
fact(1)	1
fact(3)	6 (3*2*1)
fact(4)	24 (4*3*2*1)
fact(n-1)	(n-1)*(n-2)*…1
fact(n)	~~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

Python Tutor

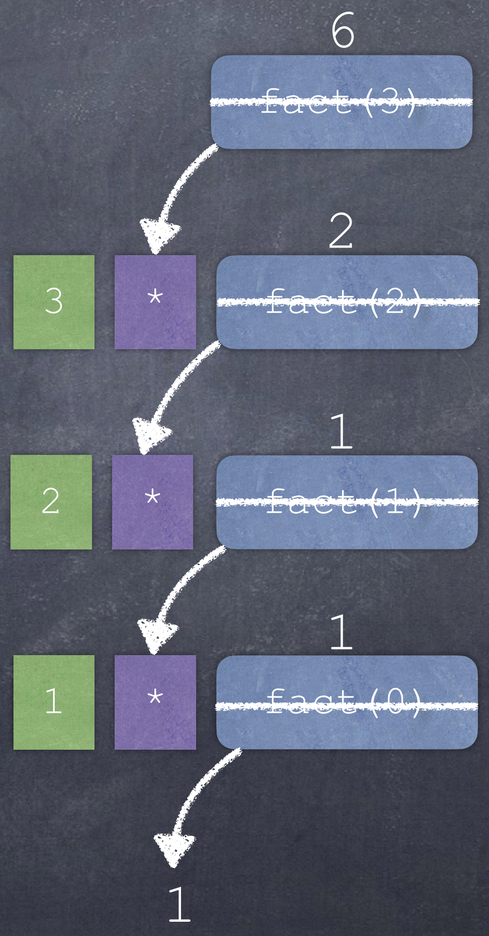
在环境图中的递归

```
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)
```



更加简洁的视角，如果一个"大"问题一次需要多个"小问题"的解的话，该视角会更加清晰

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

Demo

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

Demo

Sum Digits

- 输入：正整数
- 输出：所包含的所有数字之和
- 基本情形：
  - 单个数字
- 更小问题：
  - 除了一位数字（最小的一位），所包含的其他所有数字之和
- 更大问题：
  - 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

Demo

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)

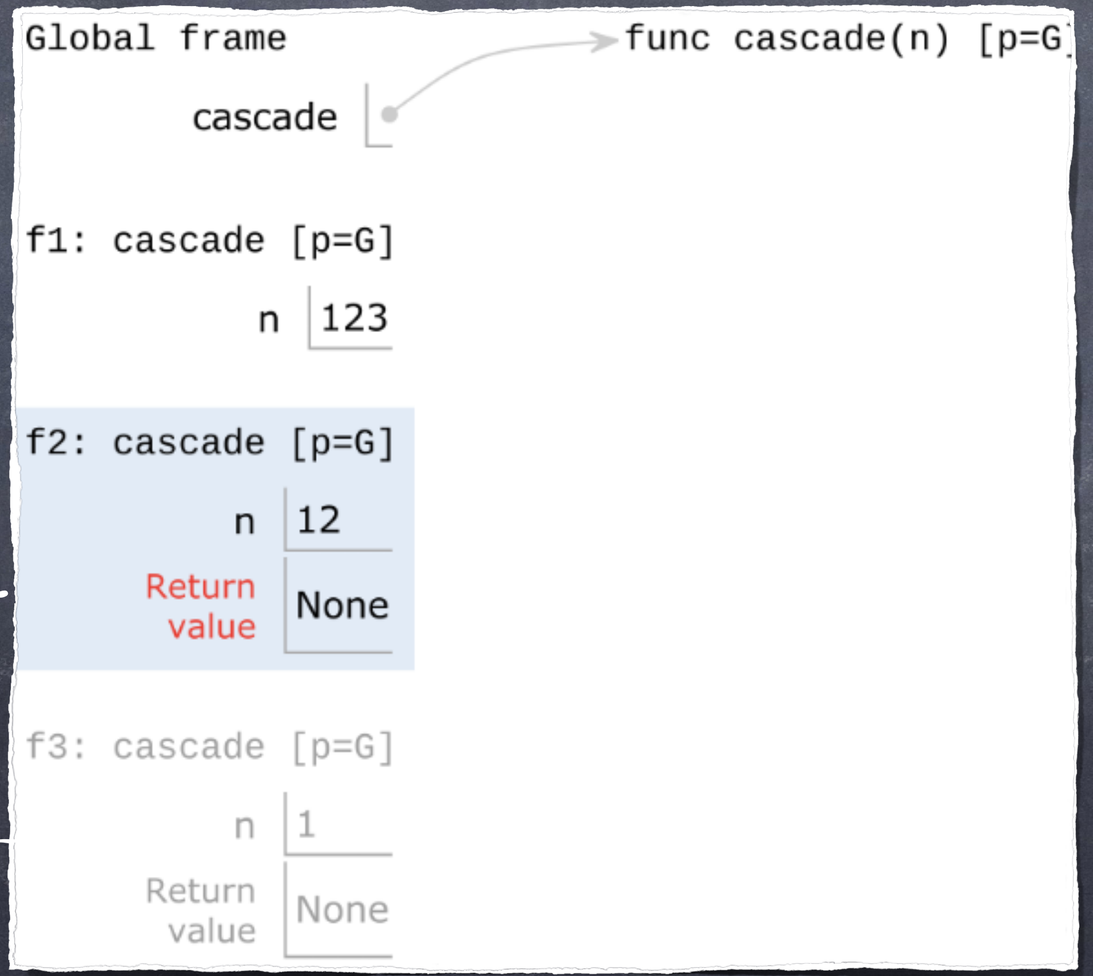
Python tutor

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。

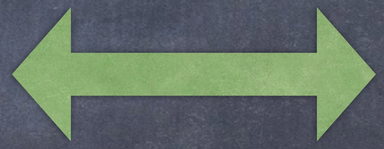
直到return语句出现前，这个调用都没有终止（因此，帧还是存在，里面的names绑定也存在）。

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

更多的底层细节

更高的抽象

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)
```

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

Names: n, fact

将迭代转化为递归

- 关键点：迭代的状态可以作为参数传递（简单）

迭代中记录的状态

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

将递归转化为迭代

- 1. 尽量转化为尾递归（Tail Recursion）
  - 尾递归为一类特殊的递归函数，其函数体的最后一句执行语句为一个递归调用（即递归调用之后什么都没有了）

Non-tail-recursion

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

Tail-recursion

```
def tail_factorial(n, accumulator=1):
  if n == 0:
     return accumulator
  else:
     return tail_factorial(n-1, accumulator * n)
```

将递归转化为迭代

1. 尽量转化为尾递归（Tail Recursion）

```
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
```

partial sum

remaining part to do

关键点：找出什么状态需要被追踪 (tricky)

```
def tail_sum_digits(n, accumulator = 0):
    if n < 10:
        return n + accumulator
    else:
        all_but_last, last = n // 10, n % 10
        return tail_sum_digits(all_but_last, accumulator + last)
```

将递归转化为迭代

- 2、将尾递归转化为迭代（基本方法）：
  - 初始化accumulator（partial result）的值
  - 使用基本case条件的逆命题作为迭代终止条件
  - 将递归的函数体转换为迭代的主体部分
  - 循环结束后，利用基本case的值更新accumulator的值并返回

将递归转化为迭代

2、递归转化为迭代（基本方法）：

```
def tail_sum_digits(n, accumulator = 0):
    if n < 10:
        return n + accumulator
    else:
        all_but_last, last = n // 10, n % 10
        return tail_sum_digits(all_but_last, accumulator + last)
```

```
def iter_tail_sum_digits(n):
    accumulator = 0
    while n >= 10:
        n, last = n // 10, n % 10
        accumulator += last
    accumulator += n
    return accumulator
```

更一般的递归转化为迭代

- 对任意一个递归函数，一个通用的将其转化为迭代的方法：
  - 基于递归函数（以及所有的函数调用）都是通过计算机的栈空间进行push frame（对应call）和pop（对应return），那么一个可行的方法即为用代码模拟栈的操作，即可转化所有递归为迭代

另一种转化

- 回顾不动点（五奇）：
  - $\mathbf{Y} = \lambda f.(\lambda x.f\ (x\ x))(\lambda x.f\ (x\ x))$
  - $\mathbf{Y}\ f = f(\mathbf{Y}\ f)$
- 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)
```

Demo

另一种转化

- python 过早求值函数
- 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 factorial(n - 1)是正确的
    - 验证一下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fact(n-1)的实现，只需认为其可以正确给出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 Recursison）

例子

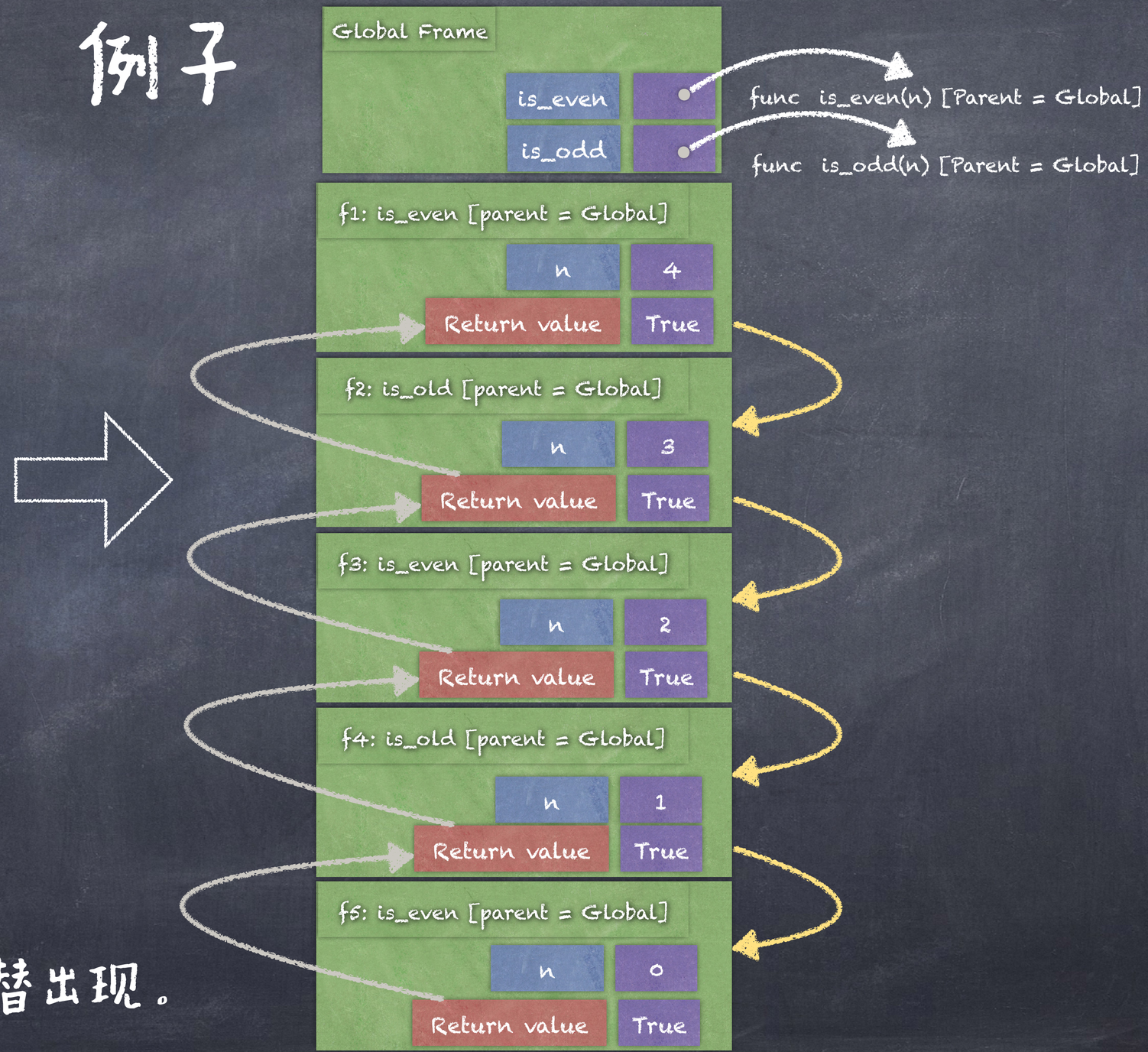
- 判断一个数字是奇数还是偶数
  - 0是偶数
  - 如果n-1是奇数，那么n是偶数
  - 如果n-1是偶数，那么n是奇数

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

- 可以将互递归转化为直接递归
  - 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。

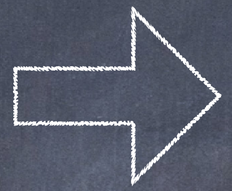
Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. 1993. On the conversion of indirect to direct recursion.

转化为直接递归

内联是重要的代码优化技术！在编译器里广泛使用

```
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

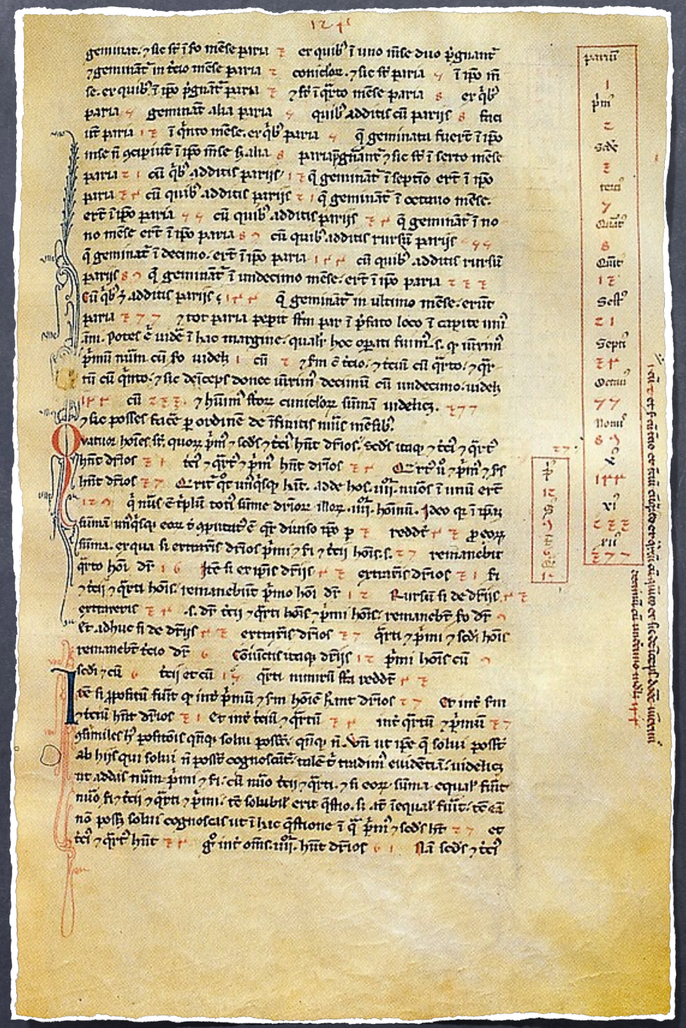
- 这种转化（内联）虽然有效。但是如果互递归里包含了多个（重复）的调用，那么就要替换多次
- 造成的后果可能会造成代码膨胀，从而进一步限制了这种转化的有效性。

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）

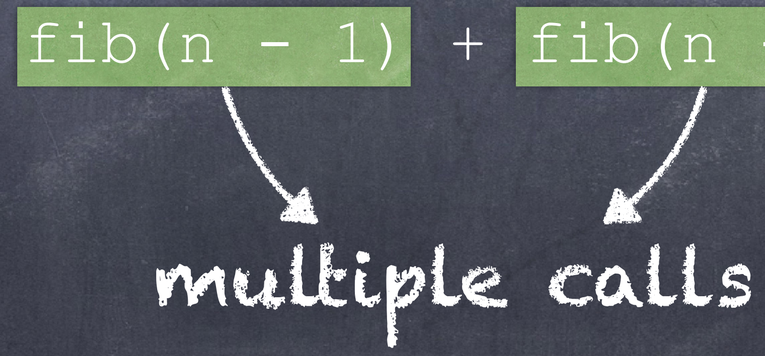
- 最早由印度人Virahanka发现，
  后由Fibonacci独立给出的数列：
  - $F_0 = 0$
  - $F_1 = 1$
  - $F_n = F_{n-1} + F_{n-2}$，where $n > 1$



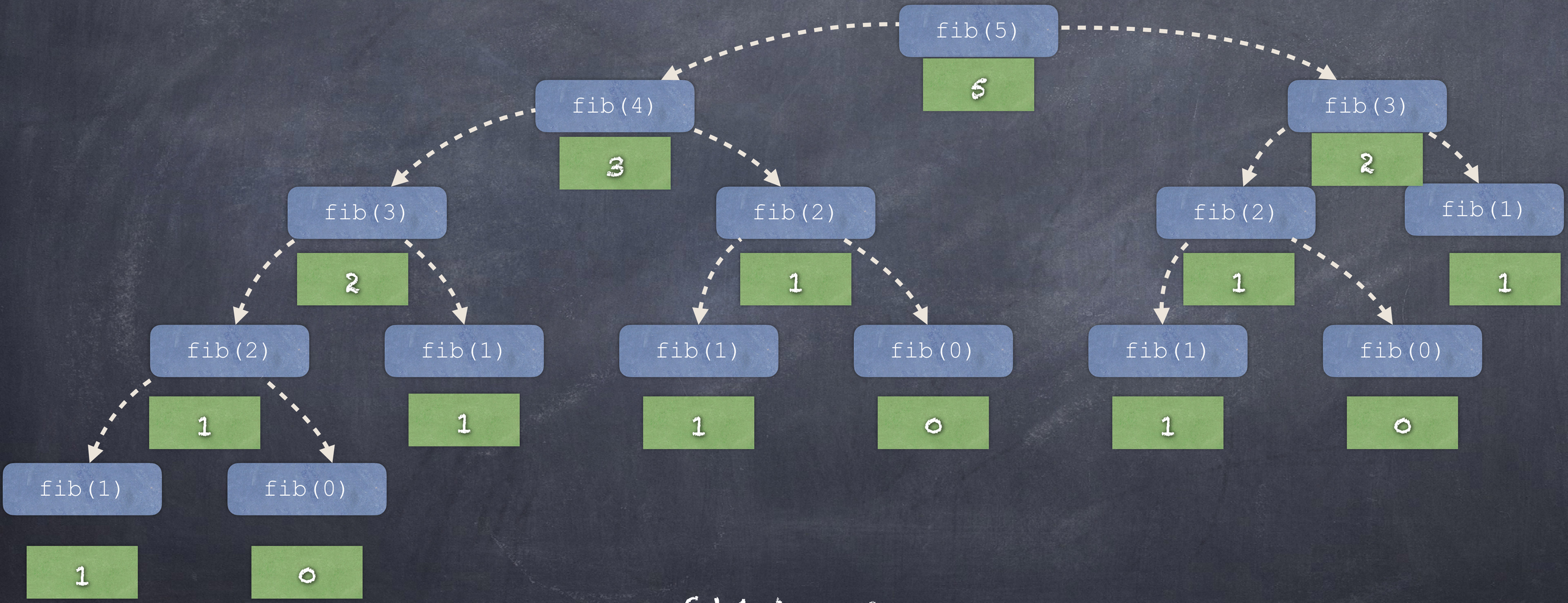
Liber Abaci (The Book of Calculation, 1202) by Fibonacci

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)
```



recursive tree of Fibonacci



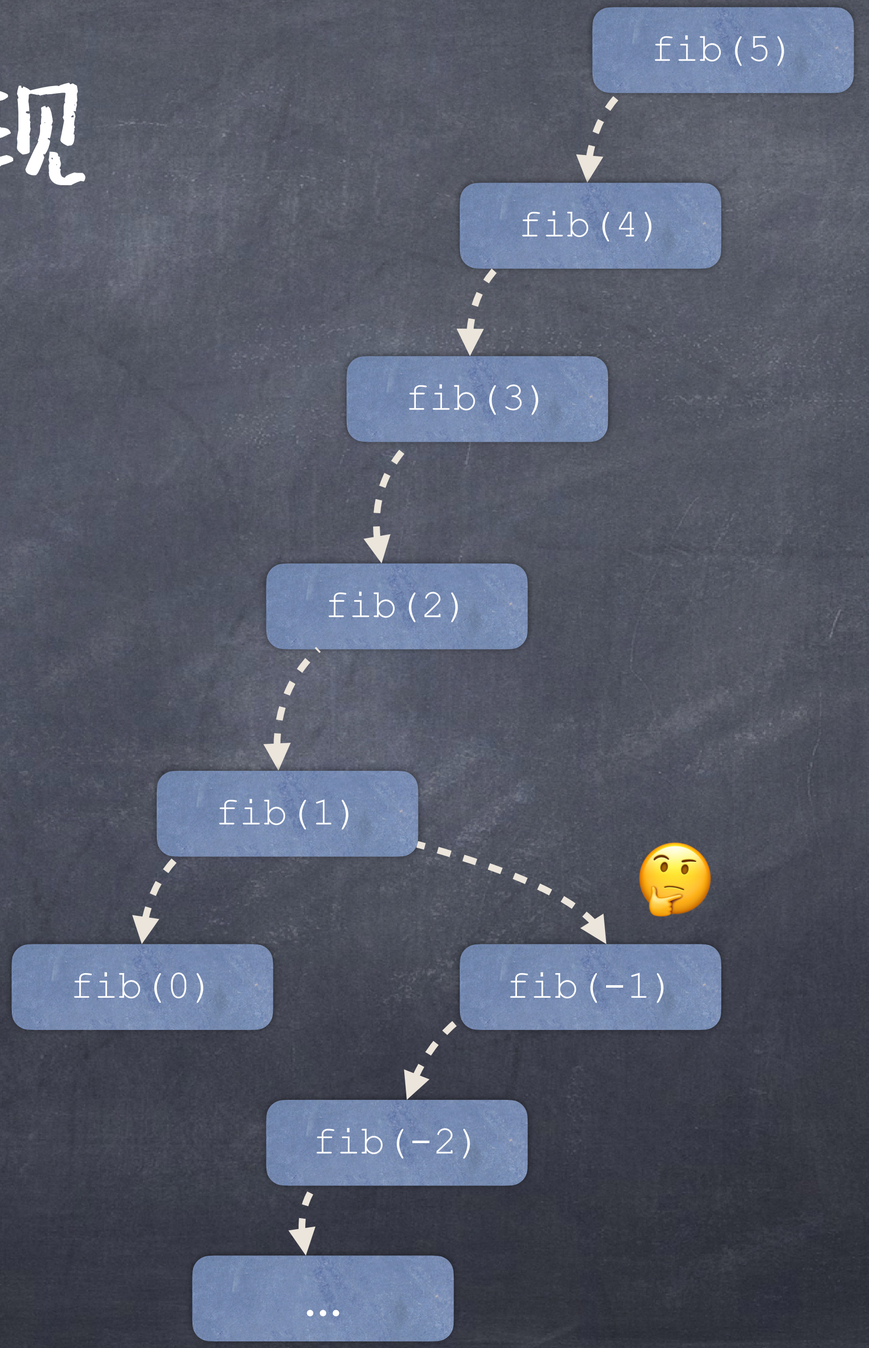
fib(n): a tree-recursive process

错误的实现

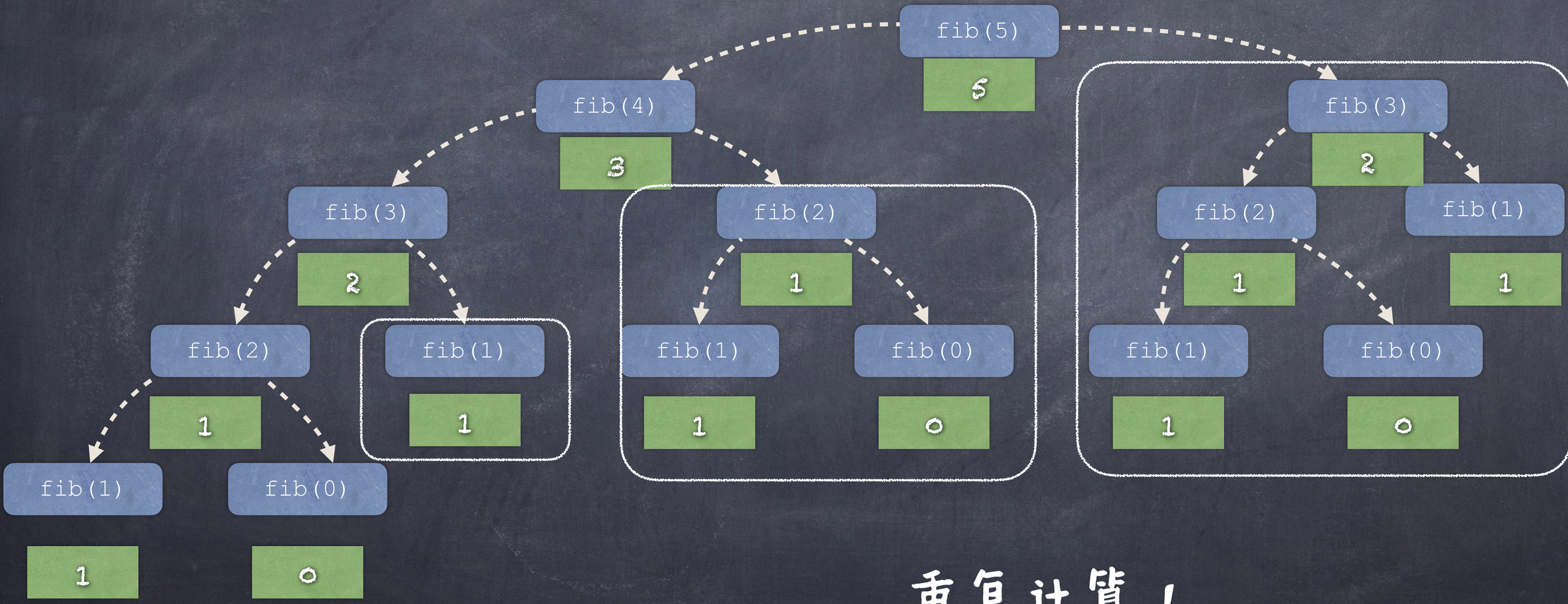
```
def fib(n):
    if n == 0:
        return 0
    #missing base case!
    else:
        return fib(n - 1) + fib(n - 2)
```

```
>>> broken_fib(5)

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



优化递归


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

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)
```

动态规划技术

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

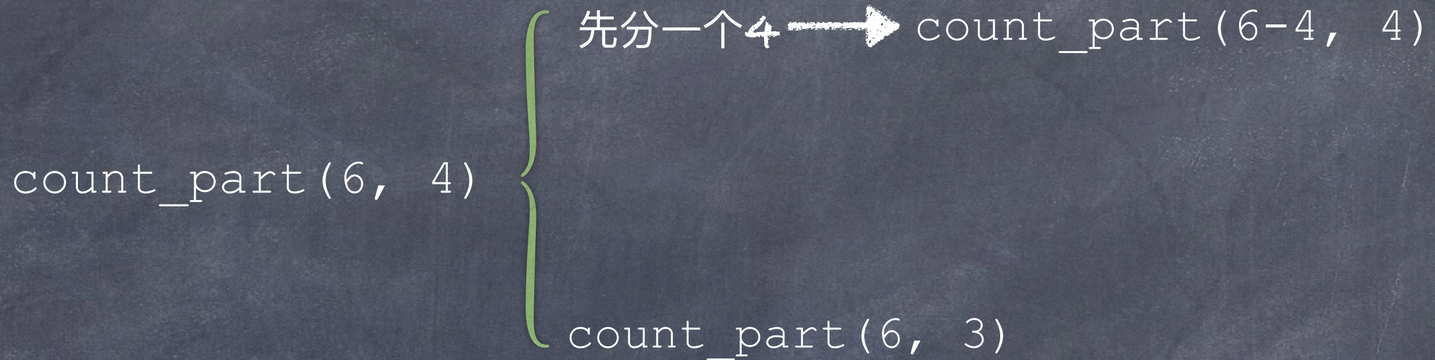
最大的分割数是1

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

Count partitions

- 对于一个大问题count_part(6, 4)而言，怎么把它分割为小问题？

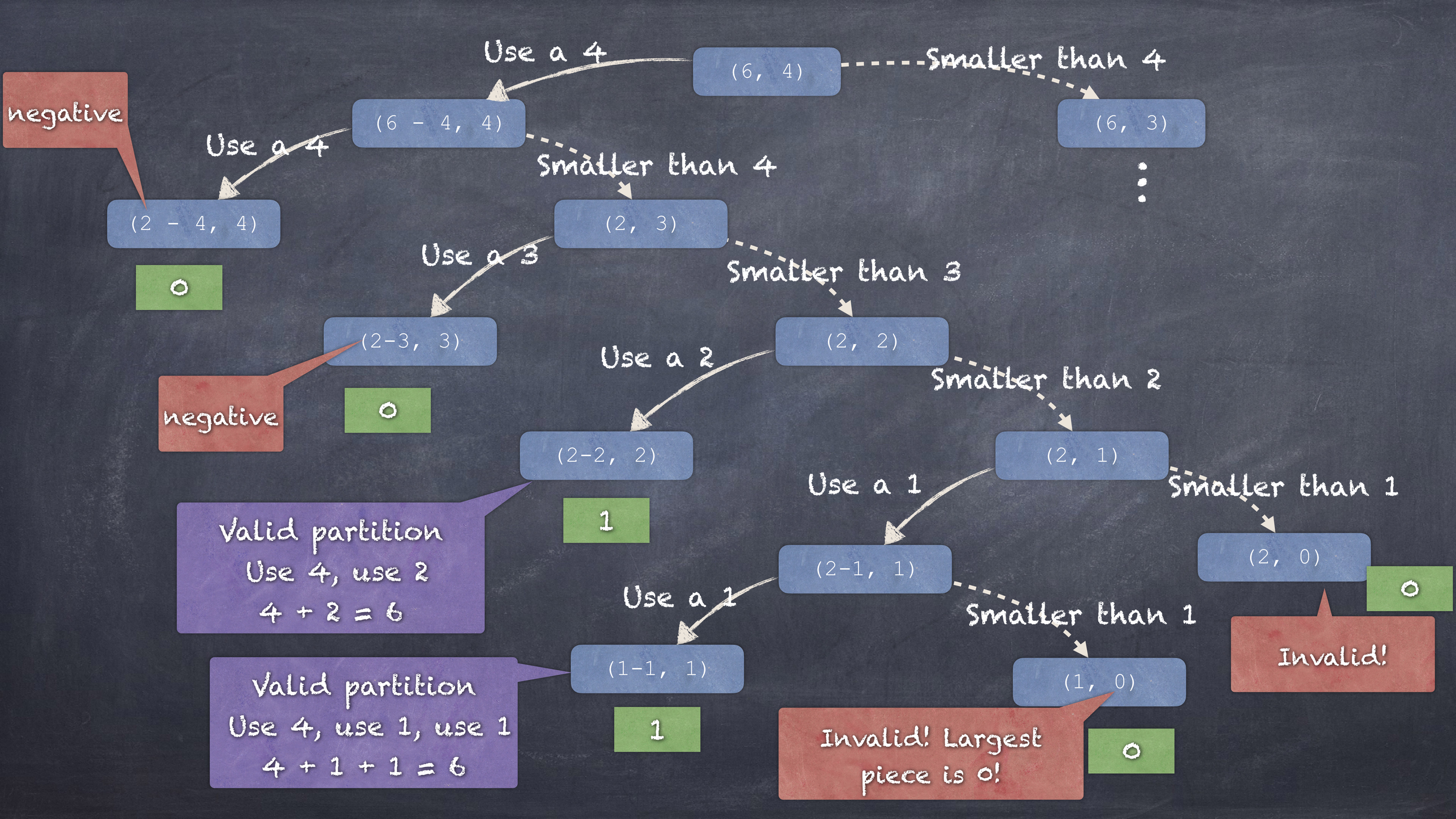
Count partitions




```
def count_part(n, m):
  if
```

The base case?

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


(6, 4)
Use a 4
Smaller than 4
(6 - 4, 4)
(6, 3)
Use a 4
Smaller than 4
negative
(2 - 4, 4)
(2, 3)
0
Use a 3
Smaller than 3
(2-3, 3)
(2, 2)
negative
0
Use a 2
Smaller than 2
(2-2, 2)
(2, 1)
Valid partition
Use 4, use 2
4 + 2 = 6
1
Use a 1
Smaller than 1
(2-1, 1)
(2, 0)
0
Use a 1
Smaller than 1
Invalid!
Valid partition
Use 4, use 1, use 1
4 + 1 + 1 = 6
(1-1, 1)
(1, 0)
1
Invalid! Largest
piece is 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 ?