程序设计基础
——递归

林大 经管学院 瞿华

递归

一. 递归简介

二. 递归与分治法

三. 递归与分形*

递归

一. **递归简介**

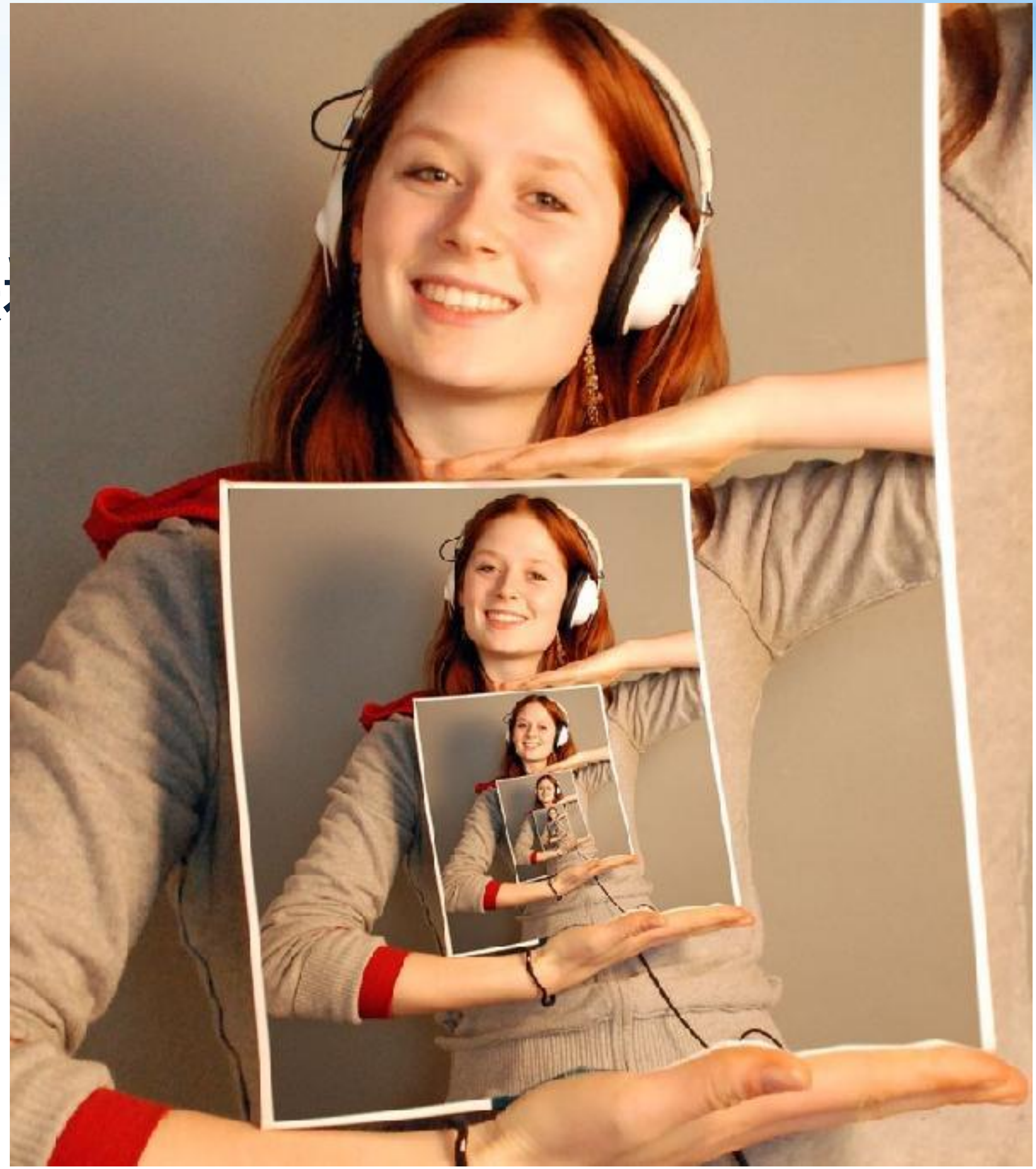
二. 递归与分治法

三. 递归与分形*

1.1 递归简介

- 听过这个故事吗？
  - 从前有座山，山里有座庙，庙里有一个老和尚和一个小和尚，有一天老和尚给小和尚讲故事，讲得什么呢：
    - 从前有座山，山里有座庙，庙里有一个老和尚和一个小和尚，有一天老和尚给小和尚讲故事，讲的什么呢：
      - 从前有座山……

❖ 再来

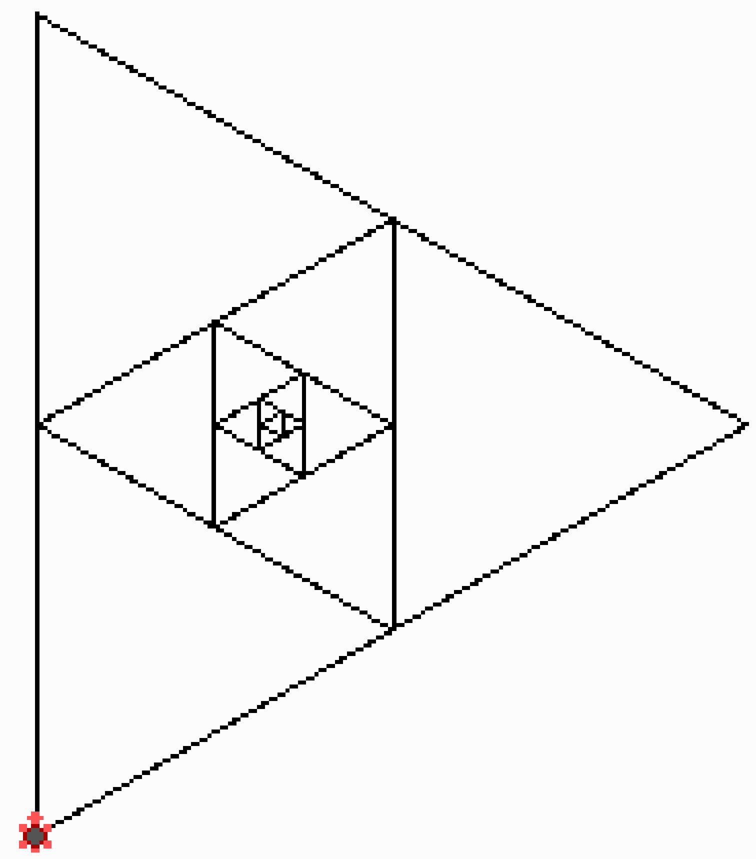


1.1 递归简介

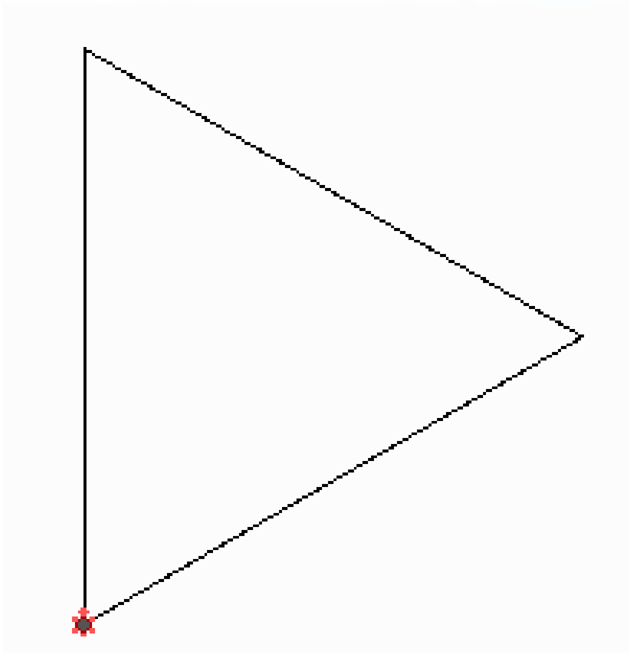
- 几个例子的共同点：
  - 重复地包含自身
- 在函数中，同样可以通过调用自身来实现自我的重复，这种编程技巧就是所谓的**递归（recursion）**。
- python语言中的递归是通过函数来实现的。既然在函数内能调用其他函数，那么当然也可以调用自己。这就形成了递归。

1.2 递归三角形

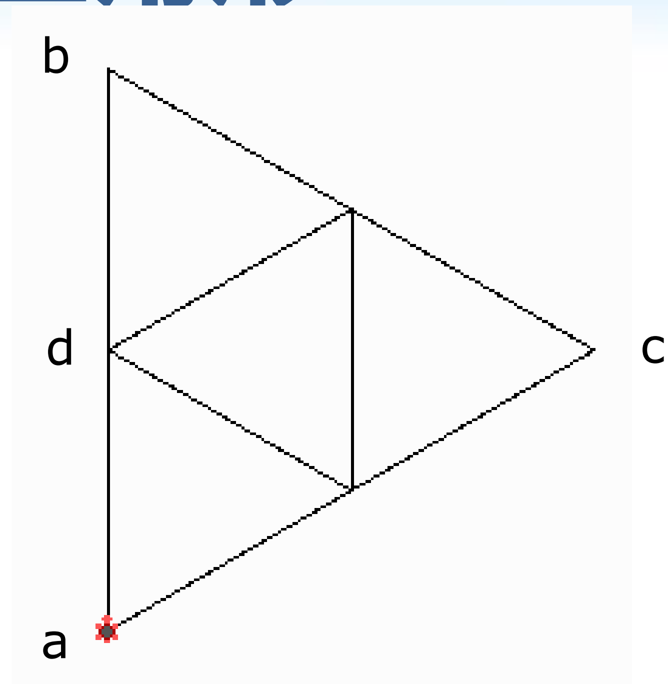
- 例1-1：使用海龟作图，绘制如图所示的内接递归三角形



1.2 递归三角形



level=1



level=2

可以这样来画大三角形：

1. 画线段ad
2. 以d点为起点，画内接三角形。画完后回到d点
3. 画线段db、bc、ca，回到起点

显然在画内接三角形时，可以用同样的方法绘制，从而形成递归

1.2 递归三角形

- 绘制三角形的递归函数如右图所示。
- 注意：
  - 是怎么递归调用的？
  - 会不会一直递归下去？到什么时候结束？

```
def inner_triangle(size,level):
    if level == 0 :
        return
    fd(size/2)  # 绘制线段ad
    rt(60)  # 设定内接三角形起始朝向
    inner_triangle(size/2,level-1) # 绘制内接三角形
    lt(60) # 恢复海龟原有朝向
    fd(size/2)  # 绘制线段db
    rt(120)
    fd(size)  # 绘制线段bc
    rt(120)
    fd(size)  # 绘制线段ca
    rt(120)  # 恢复海龟最初朝向
```

1-1.内接递归三角形.py

1.3 简单递归

- 总结起来，所有的递归都包含下面两个基本的操作：
  - 在函数中**调用自身**，从而实现重复
  - **终止条件判断**，即满足一定条件后就结束重复调用，否则就会形成无限递归（最后会怎样？）

1.3 简单递归

- 例1-2：不使用循环和mapreduce，计算n的阶乘 n!=1*2*3*…*n
- 分析：
  - 输入输出略
  - 处理步骤：我们可以利用公式n!=n*(n-1)!，这样来求解：
    - ✓ n!=n*(n-1)!
    - ✓ (n-1)!=(n-1)*(n-2)!
    - ✓ ……
    - ✓ 0!=1
    - ✓ 由于这个公式使用阶乘(n-1)!来计算阶乘n!，因此它是一个**递归公式**

1.3 简单递归

- 总结起来，该计算阶乘的方法实际上是这样来计算n！的：
  - 当n大于0时，结果为n*(n-1)!
  - 当n等于0时，结果为1（递归结束条件）
- 这类使用递归思想来求解问题的算法就是递归算法。
- 在python语言中，在函数中可以调用自身，从而实现递归算法。

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

1-2.递归求阶乘.py

1.3 简单递归

- 例1-3： “兔子问题”
  - 假定一对大兔子每月能生一对小兔子
  - 每对当月新生的小兔子，经过一个月成长，在下下个月可以长成一对大兔子
  - 如果不发生死亡，且每次均生下一雌一雄，问一年后共有多少对兔子
- 分析：（输入输出略）
  - 第一个月是最初的一对兔子生下一对兔子，共有2对兔子；
  - 第二个月仍是最初的一对兔子生下一对兔子，共有3对兔子；
  - 到第三个月除最初的兔子新生一对兔子外，第一个月生的兔子也开始生兔子，因此共有5对兔子；
  - 从第三个月开始，总兔子数=上个月的兔子数+新生的兔子数(恰好等于上上个月的兔子数)

1.3.1 斐波那契数列

- 令a_n为第n个月的兔子总数，则有：
  - $a_1=2$;
  - $a_2=3$;

  …
  - $a_n=a_{n-1}+a_{n-2}$
- 这样就构成了一个数列2,3,5,8…
- 如果我们取每个月的大兔子数，得到数列1,1,2,3,5,8…，仍满足公式$a_n=a_{n-1}+a_{n-2}$，这就是所谓的斐波那契数列。其中每一项a_n可以用递归公式来计算：
  - $a_n=a_{n-1}+a_{n-2}$ （当n>2时）
  - $a_1=1$、$a_2=1$;（递归结束条件）

1.3.1 斐波那契数列（2）

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

1-3.斐波那契数列.py

1.3.2 递归求最大值

- 循环和递归是实现重复计算的两种不同方式，两者是可以相互转化的。
- 例如，使用递归，同样可以实现列表的迭代访问。
- 例1-4：已知列表lst中的n个元素lst[0]...lst[n-1]，求其最大值。
- 分析：输入输出略
  - 要用递归方式求解，那么就需要找出递归公式或者递归算法
  - 递归的核心：借助用类似子问题的解，来求解
  - 我们的问题是求l最大值。那么比它规模小一点的问题，自然也是求最大值。
  - 原问题是求n个元素的最大值，那么比它规模再小一点，就应该是求n-1个元素的最大值了。

1.3.2 递归求最大值（2）

- 那么，如果已知lst前n-1个元素lst[0]..lst[n-2]中的最大值m，和lst中的第n个元素lst[n-1]，能否求出lst[0]……lst[n-1]中的最大值?
  - 显然，m和lst[n-1]谁大，谁就是lst[0]……lst[n-1]中的最大值
- 找出了核心的递归求解步骤，下一步是找递归停止条件（何时停止递归）：
  - 显然，每递归一次，要找最大值的列表范围元素减一（从n开始、n-1、n-2...）。当范围中只剩一个元素时（n=1时），这个范围中的最大值就是这个元素，即lst[0]。

1.3.2 递归求最大值（3）

```
def find_max(lst,n):
    if n==1:
        return lst[0]
    m=find_max(lst,n-1)
    if m>lst[n-1]:
        return m
    else:
        return lst[n-1]
```

1-4.递归求最大值.py

1.4 递归与递推

- 递归调用的结果到底是怎么计算出来的呢?
- 递归调用实际上就是多层函数嵌套。我们可以近似将其计算过程展开。
- 例：求3的阶乘:

```
f(3) -> {
  return 3*f(2) -> {
    return 2*f(1) -> {
      return 1*f(0) -> {
         return 1
      }
    }
  }
}
```

调用的顺序是:

1. f(3)调用f(2)
2. f(2)调用f(1)
3. f(1)调用f(0)

实际得到计算结果的顺序正好反过来:

1. 先得到f(0)=1
2. 再得到f(1)=1*1=1
3. 再得到f(2)=2*1=2
4. 最后得到f(3)=3*2=6

典型的后进先出!

1.4 递归与递推

- 因此，这一类递归算法的计算过程可以分为两步：
  1. **递归降解**：递归调用直到遇到结束条件为止。这是一个逐步缩小待解决问题规模的过程。
     - ✓ 例：阶乘中的f(5)调用f(4)..一直到f(0)
  2. **递推**：从结束条件开始，逐层反推出最终结果。
     - ✓ 算出f(0)的结果，然后f(1)…一直到f(5)
- **对于很多递归算法，可以取消递归降解的过程，直接进行递推，从而将递归算法转换为非递归算法。**
- 显然，非递归的递推算法比递归算法的执行效率要更高。
  - 但递归算法往往更容易设计和理解。
  - 已经有部分语言的编译器可以自动将简单递归算法转换为递推算法。

1.4.1 递推求阶乘

- 例1-5：用递推法求阶乘
- 分析：直接将递归过程反过来进行递推即可
  - 先求0!=1
  - 再求1!=1*1=1
  - ……
  - 最后求得n!=n*(n-1)!
- 实际上这就是最基本的用循环求阶乘。

```python
def factorial(n):
    result = 1
    for i in range(1,n+1):
        result = result * i
    return result
```

1-5.递推求阶乘.py

1.4.2 递推求斐波那契数列

- 例1-6：递推求斐波那契数列
- 分析：将递归公式倒过来就得到了递推公式：
  - $a_1=1;$
  - $a_2=1;$

  …
  - $a_n=a_{n-1}+a_{n-2}$
- 用循环实现即可
  - 简单起见，引入列表a，以保存之前的结算结果以便获得a_{n-1}和a_{n-2}

1.4.2 递推求斐波那契数列（2）

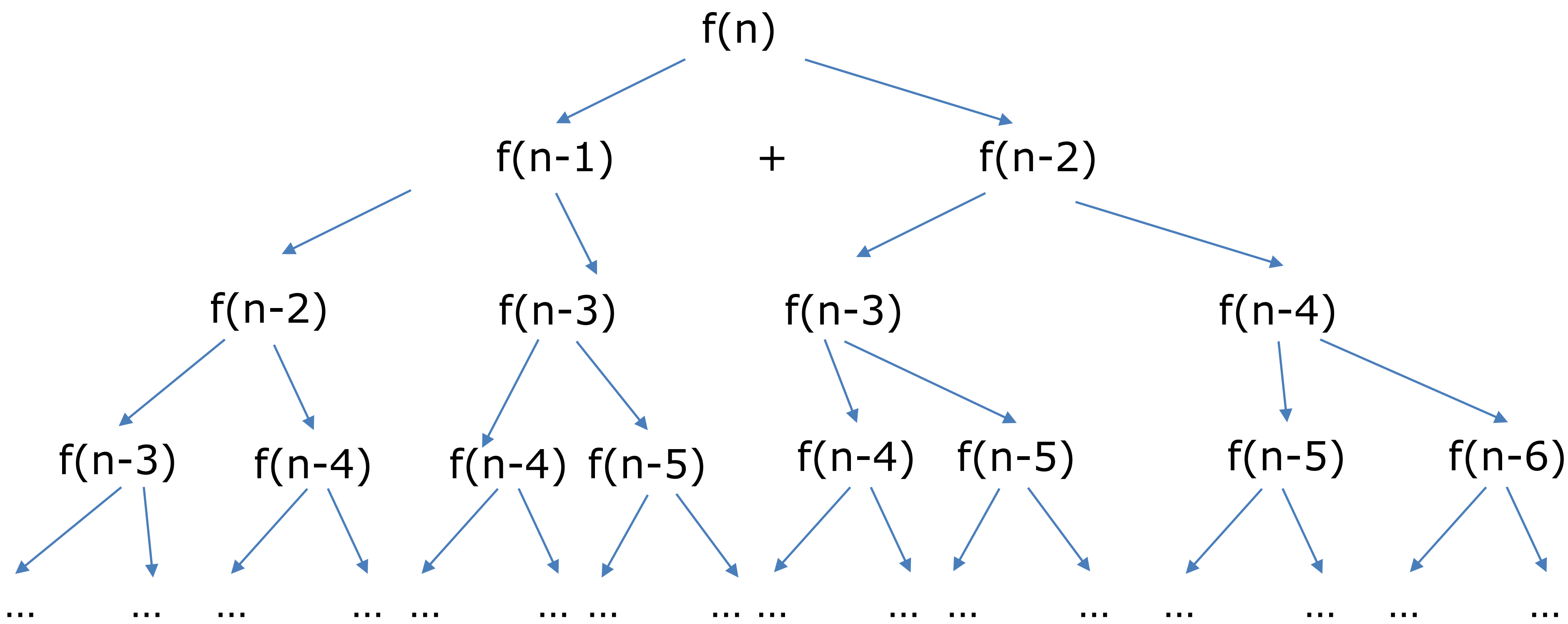
```
def fibonacci(n):
    a=[0]*(n+1)
    lst[1]=1
    lst[2]=1
    for i in range(3,n+1):
        lst[i]=lst[i-1]+lst[i-
2]
    return lst[n]
```

1-6.递推求斐波那契数列.py

1.5 备忘录算法

- 在很多问题中，如果我们能够将中间的计算结果保存下来重复使用，能够有效地提升程序的效率
- 例1-7：求斐波那契数列的第n项a_n。
- 在例1-3中我们已经编写一个用递归公式求a_n的函数，在其基础上计算求阶乘的总运行时间，得到程序1-7-1.timeforfibonacci.py
  - 令n=40，需要较长时间才能计算出结果。为什么会这样?
- 仔细分析一下这个函数f的运行过程，会发现它存在着重复计算的问题：

1.5 备忘录算法（2）



- f(n-2)被重复调用了2次，f(n-3)被重复调用了3次，f(n-4)被重复调用了5次，f(n-5)被重复调用了8次……

1.5 备忘录算法（2）

- 为了解决这个问题，我们可以引入一个全局列表F，用于保存已经计算出来的f(n)的值。
  - 一开始F全部元素都初始化为0。
  - 在用递归公式计算f(n)之前，先检查对应的元素F[n]是否为0。如果不为0，则说明f(n)过去已经计算过了，结果是F[n]。直接返回lst[n]
  - 每当计算出一个f(n)的结果时，将其保存在对应的元素F[n]中再返回
- 这种利用外部数组来保存递归计算中间结果，以减少不必要的递归调用的算法，就是所谓的**备忘录算法（memoization）**。

1.5 备忘录算法(3)

```
F=[0]*1000

def fibonacci(n):
    if (F[n]!=0):
        return F[n]
    if (n<=2) :
        F[n]=1
    else:
        F[n]=fibonacci(n-1)+fibonacci(n-2)
    return F[n]
```

1-7-2.备忘录算法求斐波那契数列.py

1.5.1 动态规划

- 仔细分析备忘录算法求斐波那契数列的过程，我们会发现，虽然递归过程是从n开始，n-1,n-2逐步向前的递归降解的；但是F列表的填写过程，其实和递推一样，是从F[1]、F[2]开始逐步向后进行的。
- 这就告诉我们，其实备忘录算法和其他的递归算法一样，可以通过省略递归降解过程来转化为递推形式的算法。
- 当这种带记忆的递推算法被用于解决运筹学中的最优化问题时，就是所谓的“动态规划”
  - 大家会在大二下学期的运筹学课程中学习动态规划，如果一下子学不明白的话，不妨先尝试用备忘录算法来实现，因为其通常更容易理解。

课后练习

- 1.使用递归计算列表a中所有元素lst[0],lst[1]...lst[n-1]的和
- 2.使用递推计算列表a中所有元素lst[0],lst[1]...lst[n-1]的和
- 3.使用例1-4中的递归函数，求斐波那契数列从1到n的所有项
  - 例：当n=5时，应输出1,1,2,3,5

递归

一. 递归简介

二. 递归与分治法

三. 递归与分形*

二、递归与分治

- 对于上一章我们所接触的递归的例子，实际上用迭代和递推（即循环）来解决，更符合我们的直觉。
- 但是也有很多问题，更适合用递归形式来解决。
- 比如我们接下来要看到的这几个例子。

2.1 汉诺塔问题

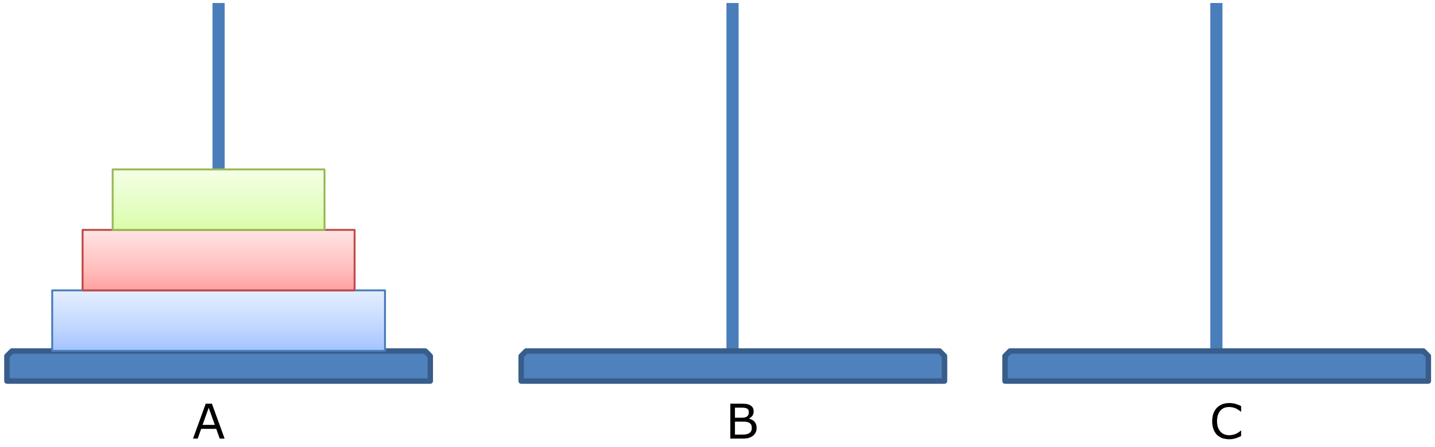
- 1883年法国数学家Edouard Lucas发明了一个益智游戏，称为“汉诺塔” （Tower of Hanoi)
- 还为其包装了一个有趣的故事：世界中心贝拿勒斯（在印度北部）的圣庙里，一块黄铜板上插着三根宝石针。印度教的主神梵天在创造世界的时候，在神庙其中一根针上从下到上地穿好了由大到小的64片金片，这就是所谓的汉诺塔。

2.1 汉诺塔问题(2)

- 不论白天黑夜，总有一个僧侣在按照下面的法则移动这些金片：
  - 一次只移动一片
  - 不管在哪根针上，小片必须在大片上面。
- 当所有的金片都从梵天穿好的那根针上移到另外一根针上时，世界就将在一声霹雳中消灭，而梵塔、庙宇和众生也都将同归于尽。

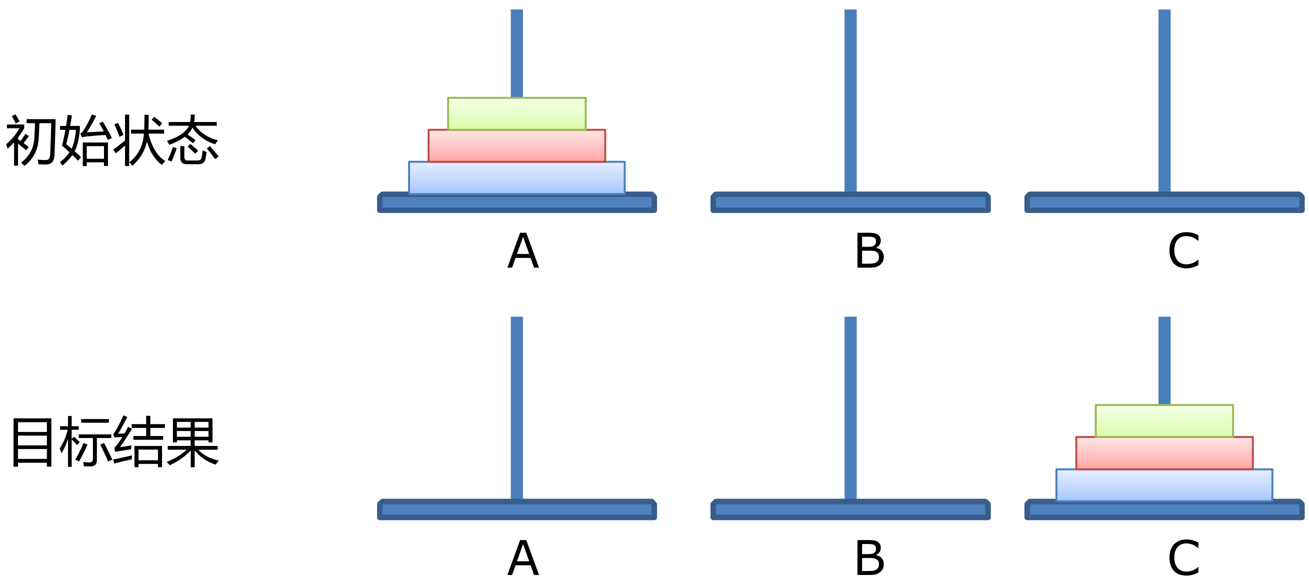
2.1 汉诺塔问题(3)

- 汉诺塔问题的正式表述：Hanoi(A,B,C,n)
  - 柱子(开始柱A、过渡柱B、目标柱C）和n个盘子
  - 开始时，盘子从大到小依次叠放穿过柱子A
  - 每次只能移动某个柱子最顶上的一个盘子移动到另一个柱子上
  - 在任何时刻，大盘子都不能放在小盘子上面
  - 最终将所有盘子都移动到柱子C上



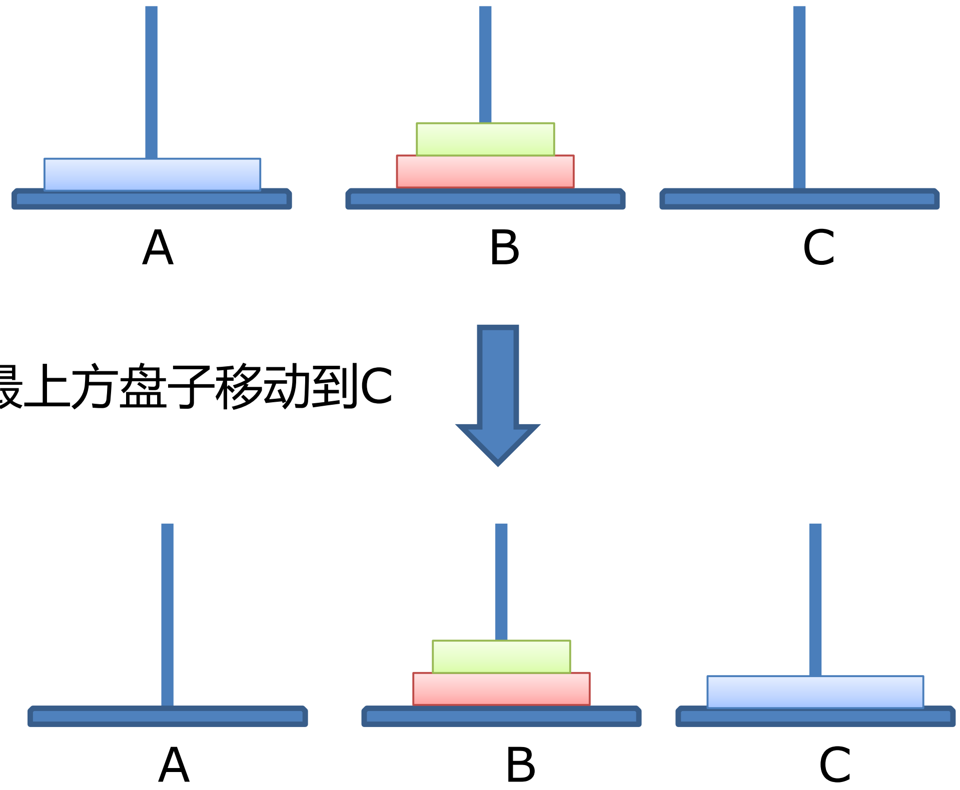
2.1 汉诺塔问题(4)

- 使用递归，可以很自然的解决汉诺塔问题
- 以n=3为例:



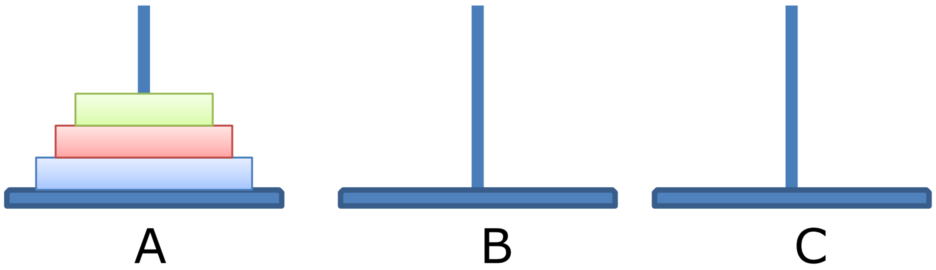
2.1 汉诺塔问题(5)

在移动过程中，要把最大的那个盘子移动到C柱上，必然经过右边这个两个状态和步骤：

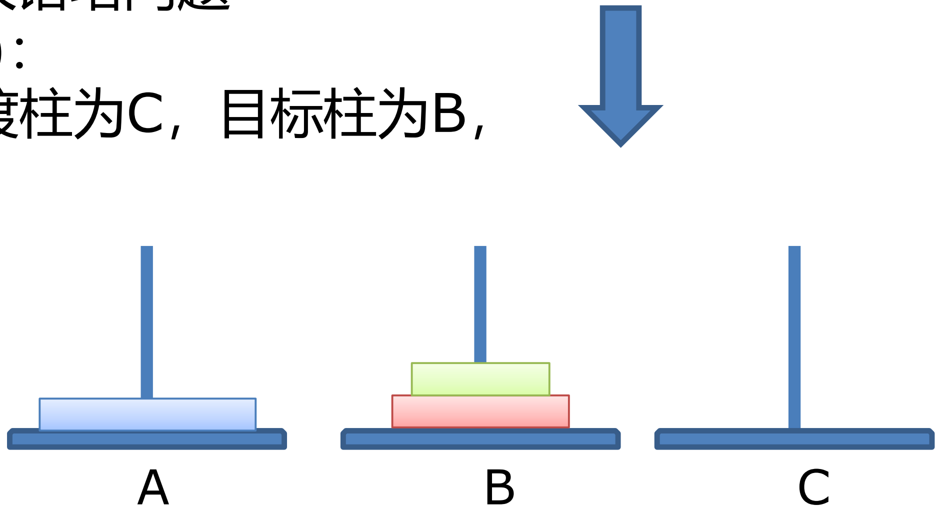


将A柱最上方盘子移动到C

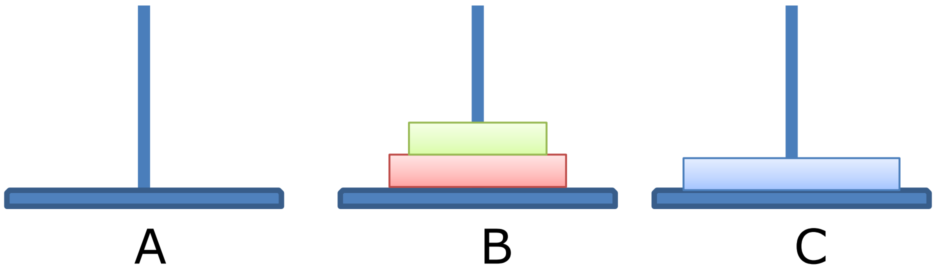
2.1 汉诺塔问题(6)



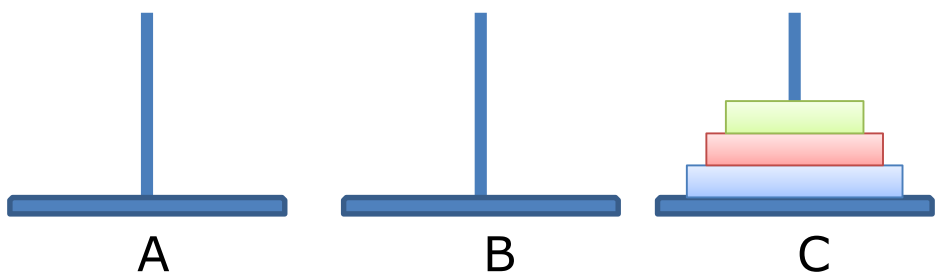
解决2个盘子的汉诺塔问题
hanoi(A,C,B,2):
开始柱为A，过渡柱为C，目标柱为B，
移动两个盘子



2.1 汉诺塔问题(7)



解决2个盘子的汉诺塔问题
hanoi(B,A,C,2):
开始柱为B，过渡柱为A，目标柱为C，
移动两个盘子



2.1 汉诺塔问题(8)

- 因此，我们可以把n个盘子的汉诺塔问题转换为n-1个盘子的汉诺塔问题：

```
def hanoi(n, from_pole, to_pole, aux_pole):
    if n==1: # 递归结束条件
        move_disk(from_pole, to_pole)
    else:
        hanoi(n - 1, from_pole, aux_pole,
to_pole)  # 把from的上面n-1个借助to移动到aux（递归
解决子问题）
        move_disk(from_pole, to_pole);
        hanoi(n - 1, aux_pole, to_pole,
from_pole)  # 把aux上的n-1个借助from移动到to（递归
解决子问题）
```

2.1 汉诺塔问题(9)

- 对于汉诺塔问题，我们利用递归，将一个复杂的问题（n阶汉诺塔）转换为了两个较简单的问题（n-1阶汉诺塔）。
- 通过这样的不断递归转化，最终将一个复杂问题转化为多个可以直接解决的结单问题。
- 这种将复杂问题逐步分解为简单问题的求解方法就是所谓的**分治法（Divide and Conquer)**
  - 递归法是一种典型的分治法

2.2 快速排序

- 利用递归和分治思想，可以实现一种较为高效的列表排序方法。
- 例：对数组lst[]=[45,26,34,90,51,4,80,27,100,72]按照从小到大顺序排列。
- 分析：
  - 从a中任选一个中间元素。为了简单起见，我们选择最后一个元素72
  - 将a中所有元素分成小于等于72和大于72两部分，即[45,26,34,4,51,27],72,[90,100,80]
  - 然后，按上述办法分别对小于等于72的所有元素和大于72的所有元素排序
- 显然，如果中间元素选取的比较好，能将数组均匀的分成元素数量相当的两部分，那么该算法只需要执行$nlog(n)$次即可，比选择排序效率更高。
- 这就是所谓的**快速排序（Quick Sort）**。

2.2 快速排序(2)

- 伪码形式（对lst[p], lst[p+1]...lst[r-1],lst[r]排序）：

```
def qsort(lst, p, r) :
    if (p>=r)      # 递归结束条件（lst中没有或只有一个元素）
        return
    q=partition(lst,p,r)
    qsort(lst,p,q-1)  # 递归解决子问题
    qsort(lst,q+1,r) # 递归解决子问题
```

- 显然，快速排序最大的难点就在于partition()函数，即如何较为简单、高效的将列表lst分为两部分：
  - lst[p], lst[p+1],... lst[q-1]都小于等于lst[q]
  - lst[q+1], lst[q+2],..., lst[r]

2.2 快速排序(3)

- partition(lst,p,r)函数中可以这样处理*：
  1. 选择lst[r]为中间元素，令x=lst[r];
  2. 令变量i为数组两部分的分界点下标，即lst[p]...lst[i]都小于等于x
     - ✓ partition()函数返回时，lst[p]... lst[i]都小于等于x，lst[i+1]... lst[r]都大于x。
  3. 初始化i=p-1;
  4. 令j从p循环到r-1
     - ① 如果lst[j]小于等于x，那么（找到一个小于等于x的元素，就把它放到lst[i]左边)
       - a) i+=1; 交换lst[i]和lst[j];
  5. 将lst[r]放到lst[i]左边（因为lst[r]也小于等于x）：
     - ① i+=1; 交换lst[i]和lst[r];
  6. 返回i

2.2 快速排序(4)

	p									r
	45	26	34	90	51	4	80	27	100	72
i	j									

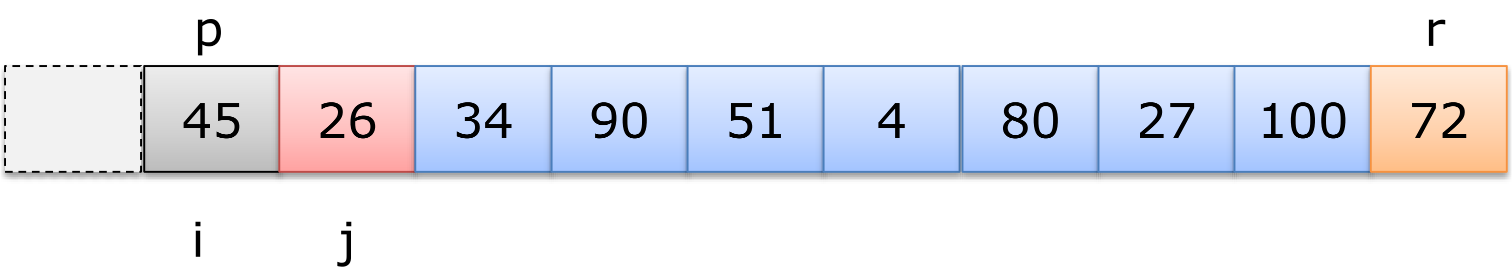
x=72

函数开始，x=lst[r]
i=p-1;
j=p

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]... lst[i]都小于等于 x
lst[i+1]... lst[j-1]都大于x

2.2 快速排序(5)



x=72

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序(6)

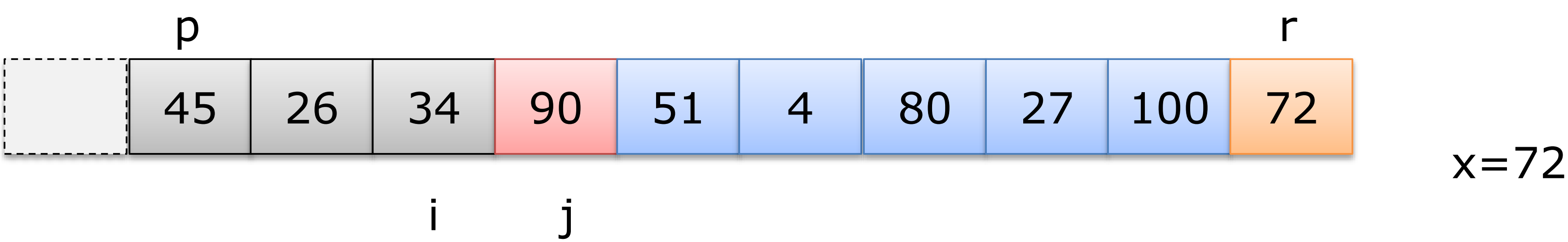
	p									r
	45	26	34	90	51	4	80	27	100	72
		i	j							

x=72

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序(7)



lst[j]>x
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序(8)

	p									r
	45	26	34	90	51	4	80	27	100	72
			i		j					

x=72

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序(9)

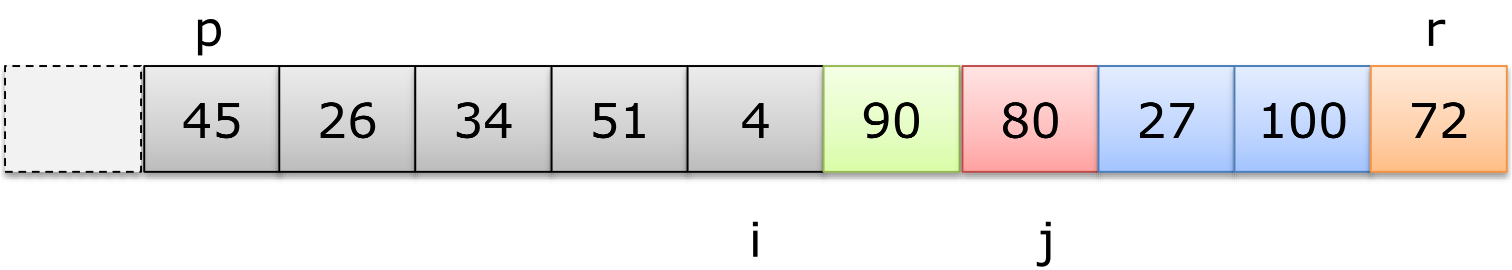
	p									r
	45	26	34	51	90	4	80	27	100	72
				i		j				

x=72

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序(10)



x=72

lst[j]>x
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]... lst[i]都小于等于 x
lst[i+1]... lst[j-1]都大于x

2.2 快速排序(11)

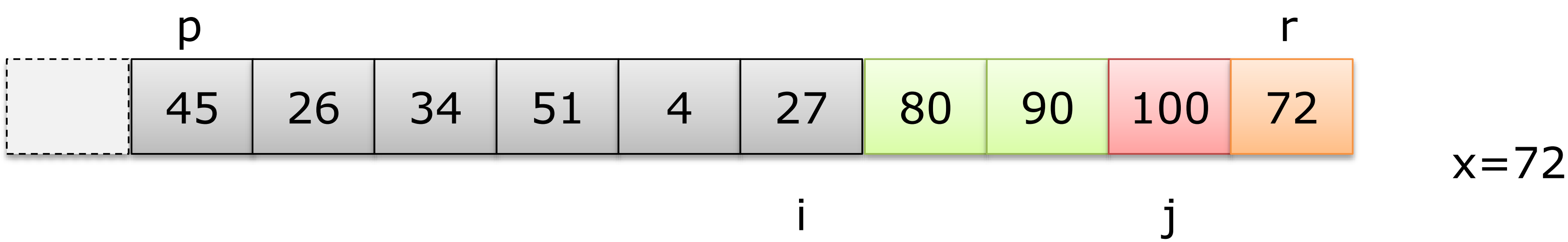
	p									r
	45	26	34	51	4	90	80	27	100	72
					i			j		

x=72

lst[j]<=x
i+=1;
交换lst[i]和lst[j];
j+=1;

x为划分的中间元素，j从p循环到r-1
lst[p]... lst[i]都小于等于 x
lst[i+1]... lst[j-1]都大于x

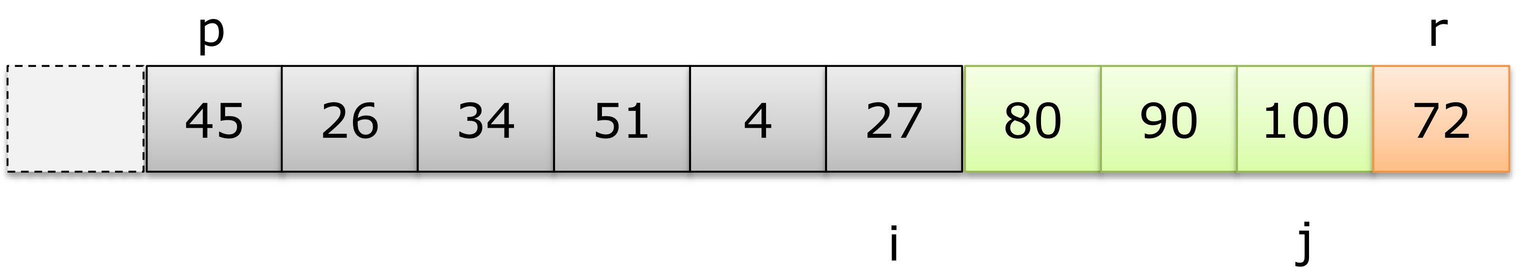
2.2 快速排序



lst[j]>x
j+=1;
循环结束

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

2.2 快速排序

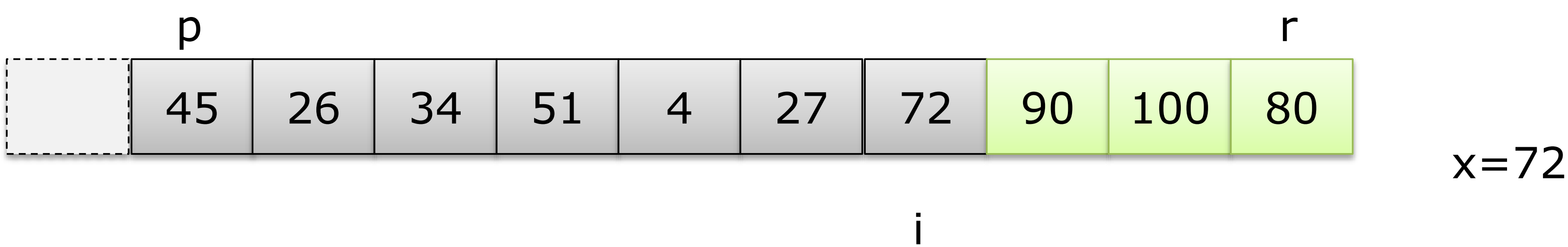


x=72

i+=1;
交换lst[i]和lst[r];

x为划分的中间元素，j从p循环到r-1
lst[p]... lst[i]都小于等于 x
lst[i+1]... lst[j-1]都大于x

2.2 快速排序



处理完毕
返回i

x为划分的中间元素，j从p循环到r-1
lst[p]… lst[i]都小于等于 x
lst[i+1]… lst[j-1]都大于x

程序见2-2qsort.c

2.3 查找文件

- 我们在使用电脑的时候，经常会遇到需要在某个文件夹（或者整个盘）里查找某个文件的情况。
- 如果该文件夹下没有子文件夹，那么只需要用循环迭代逐一访问文件夹里的所有文件即可。
- 如果该文件夹下包含子文件（子文件夹下还包含子子文件夹），该怎么办呢?
- 显然，最自然的解决方式就是用递归来访问子文件夹。
- 例2-3：在指定文件夹中寻找文件名中包含指定内容的文件。
- 分析：
  - 输入：指定文件夹和指定内容。可以用easygraphics.dialog里的函数来处理
  - 输出：直接使用print
  - 处理：综合使用循环和递归访问指定文件夹下所有文件

2.3 查找文件（2）

- 递归函数分析:
  - 递归处理：遇到子文件夹时，则递归访问子文件夹
  - 递归结束条件: 当文件夹中不包含子文件夹时，不会产生递归。因此不需要额外的递归结束条件。

2.3 查找文件（3）

```
from pathlib import Path

def find_in_dir(path,key):
    results = []
    for entry in path.iterdir(): # 遍历目录中的所有条目
        if entry.is_dir(): # 如果是子目录
            # 递归在子目录中查找
            results0=find_in_dir(entry,key)
            results.extend(results0) # 将查找结果并入结果集
        if entry.is_file(): #如果是文件
            if entry.name.find(key)!=-1:
                # 找到一个，加入结果集
                results.append(str(entry.resolve()))
    return results
```

注意：使用python标准库中的pathlib和Path对象来遍历目录中的条目
列表的extend()方法，将另外一个列表（或可迭代对象）中所有元素加入本列表
注意extend()方法和append()方法的区别

2.3 查找文件（4）

```
dir_path = dlg.get_directory_name("请选择文件夹")
s = dlg.get_string("请输入要查找的关键字:")

dir = Path(dir_path)
results = find_in_dir(dir, s)
for file in results:
    print(file)
```

注意我们先建立了Path对象，以在find_in_dir()函数中获取包含文件夹中所有条目的可迭代对象

2.3.1 非递归访问目录*

- 利用队列（queue）这种数据结构，我们可以实现目录的非递归访问
- 队列：和列表类似，元素在其中顺序排列，但是：
  - 新元素只能添加在末尾
  - 不能访问中间和末尾的元素
  - 只能从头部取出元素来访问
  - 取出的元素会立即从队列中删除
- 和我们现实中的排队很像！
- python的collection包中提供了更基本的deque（双向队列）
  - append()方法将元素放入队列尾
  - popleft()方法将元素从队列头部取出

2.3.1 非递归访问目录*（2）

```python
from collections import deque

def find_in_dir(path,key):
    results = []
    queue = deque()
    queue.append(path) #加入队列尾
    while len(queue)>0: #只要队列非空，就继续
        dir = queue.popleft() #从队列头部取出一个目录（并从队列中删除）
        for entry in dir.iterdir(): # 遍历目录中的所有条目
            if entry.is_dir(): # 如果是子目录
                queue.append(entry) # 将子目录加入队列
            if entry.is_file(): #如果是文件
                if entry.name.find(key)!=-1:
                    results.append(str(entry.resolve())) # 找到一个，加入结果集
    return results
```

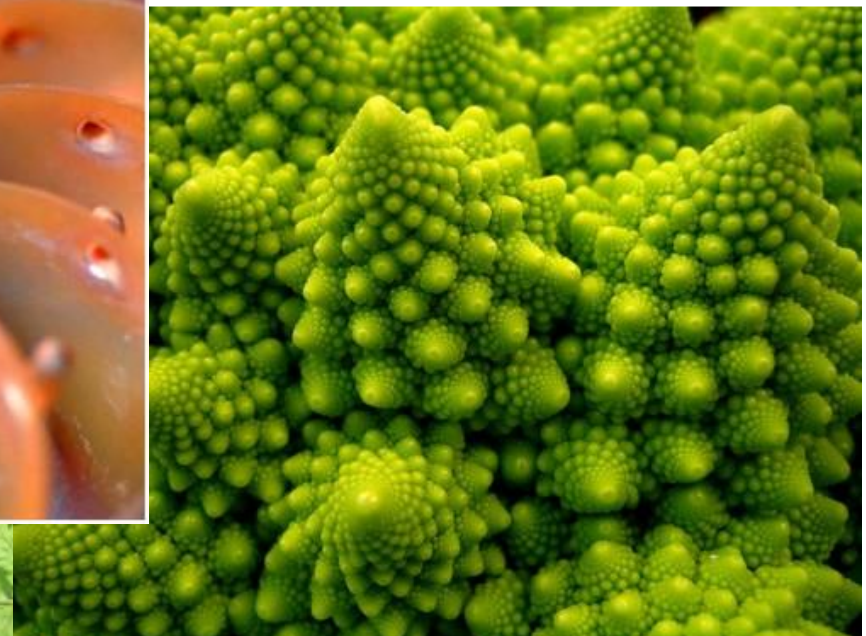
递归

一. 递归简介

二. 递归与分治法

三. 递归与分形*

三、递归与分形



3.1 分形简介

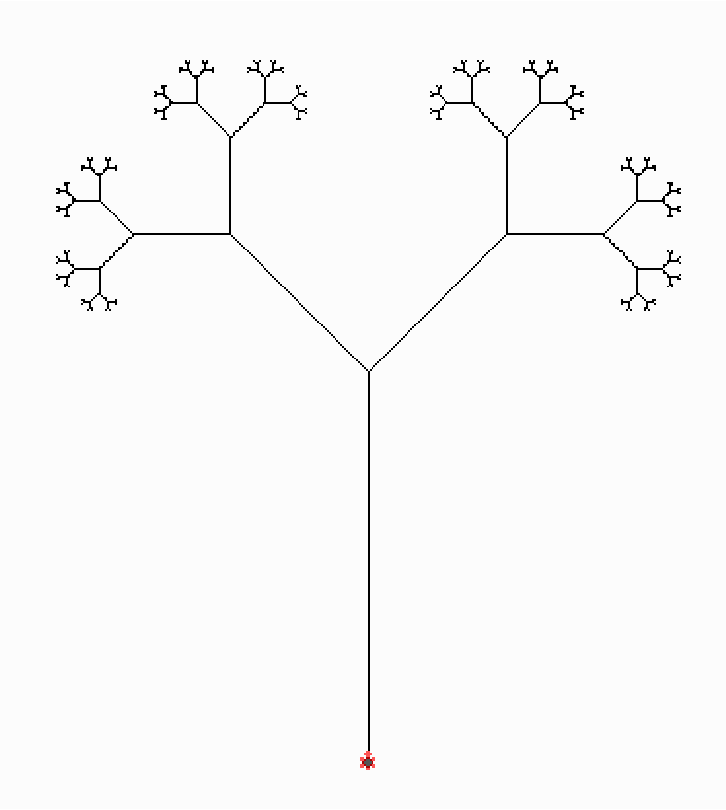
- 分形（Fractal）是计算几何学的重要分支，是非线性科学的重要学科和前沿研究领域。
- 分形几何的特点：
  1. 从整体上看，分形几何图形是处处不规则的。例如，海岸线和山川形状，从远距离观察，其形状是极不规则的。
  2. 在不同尺度上，图形的规则性又是相同的。上述的海岸线和山川形状，从近距离观察，其局部形状又和整体形态相似，它们从整体到局部，都是**自相似的**。

3.1 分形简介

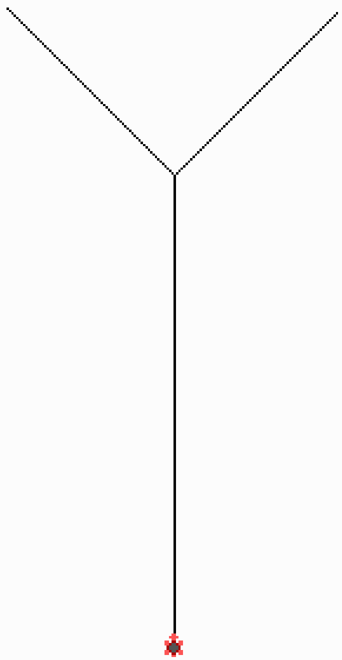
- 很显然，分形的这种自相似性非常适合用递归来表达。
- 利用递归函数，我们可以用计算机绘制出许多漂亮的分形图形。
  - 这也是计算机图形学的一个重要领域。

3.2 分叉树

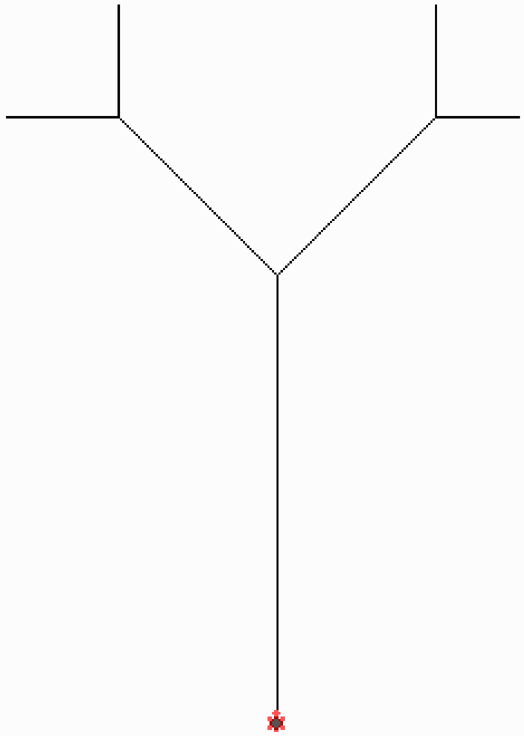
❖ 例3-1： 45度等分树



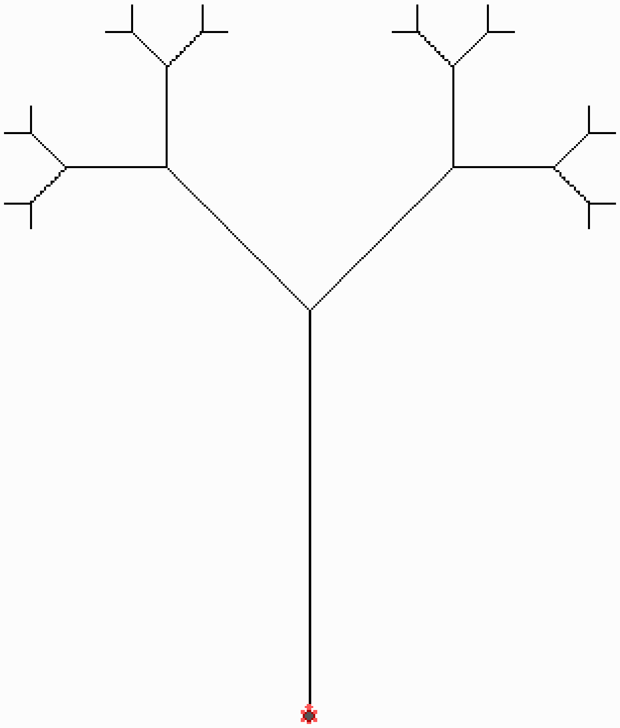
3.2.1 45度等分树



level=2



level=3



level=5

3.2.1 45度等分树

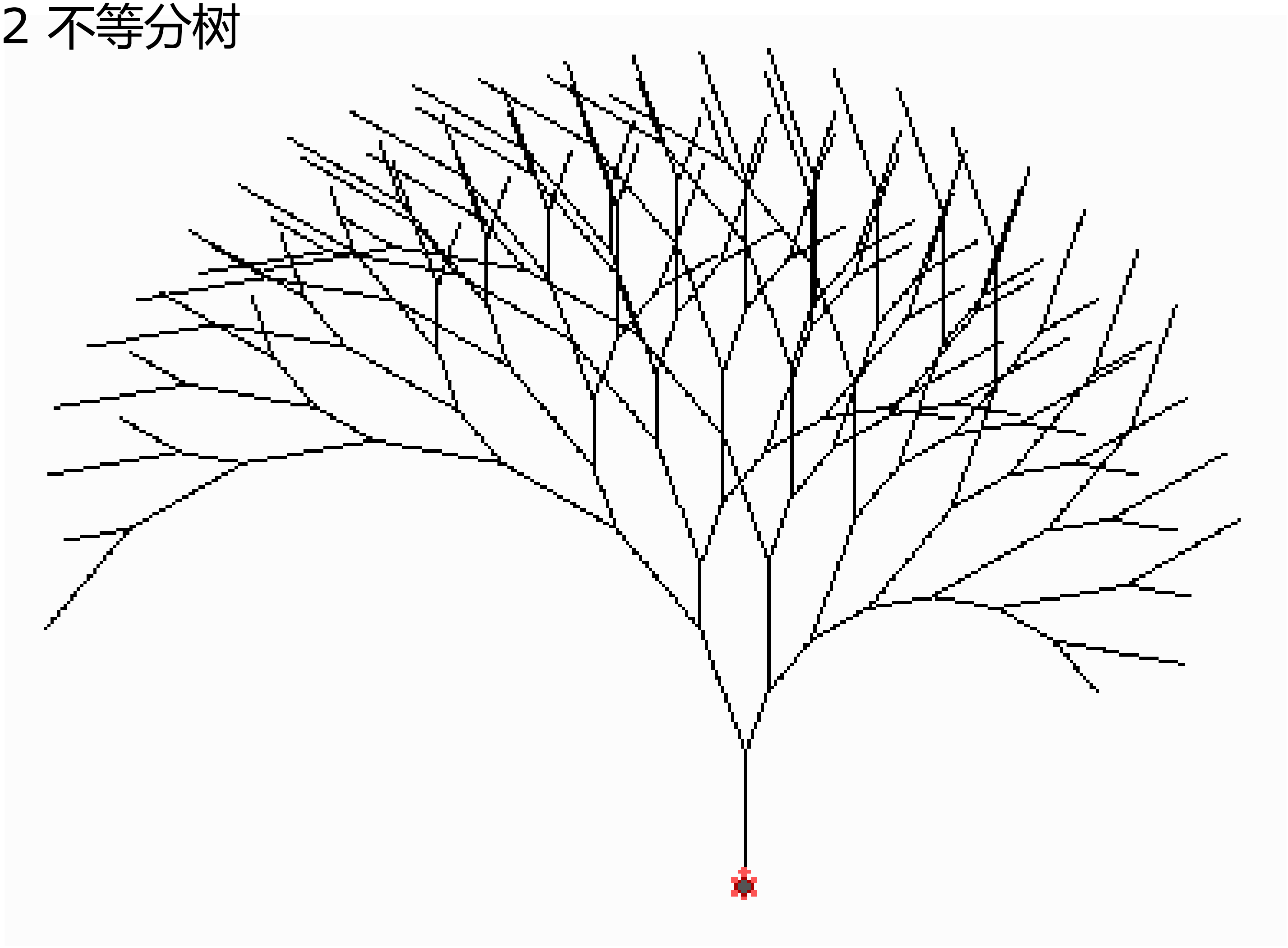
```python
def branch(len, level):
    if level == 0:
        return
    fd(len)
    lt(45)
    branch(len / 2, level - 1)
    rt(90)
    branch(len / 2, level - 1)
    lt(45)
    bk(len)
```

3-1.45tree.py

递归调用何时停止？如果不停的递归下去会怎样？

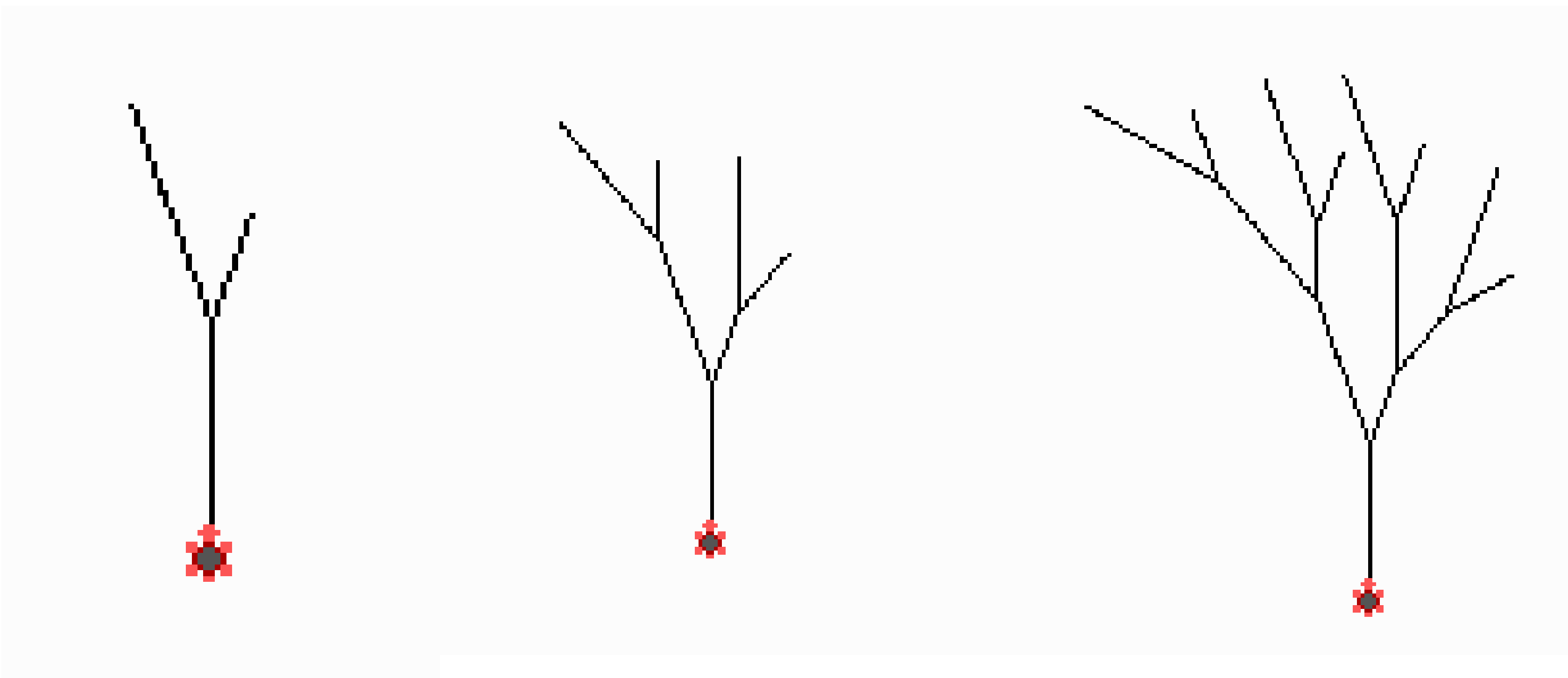
3.2.2 左右不等树

例3-2 不等分树



3.2.2 左右不等树

- 左侧树枝的长度是右侧的两倍，树枝与树干间夹角为20度



level=2　　level=3　　level=4

3.2.2 左右不等树

```python
def lbranch(size, angle, level):
    fd(2 * size)
    node(size, angle, level - 1)
    bk(2 * size)
```

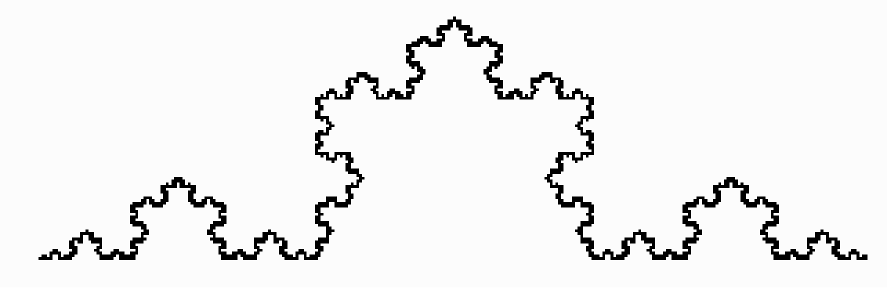
```python
def rbranch(size, angle, level):
    fd(size)
    node(size, angle, level - 1)
    bk(size)
```

```python
def node(size, angle, level):
    if level == 0:
        return
    lt(angle)
    lbranch(size, angle, level)
    rt(2 * angle)
    rbranch(size, angle, level)
    lt(angle)
```

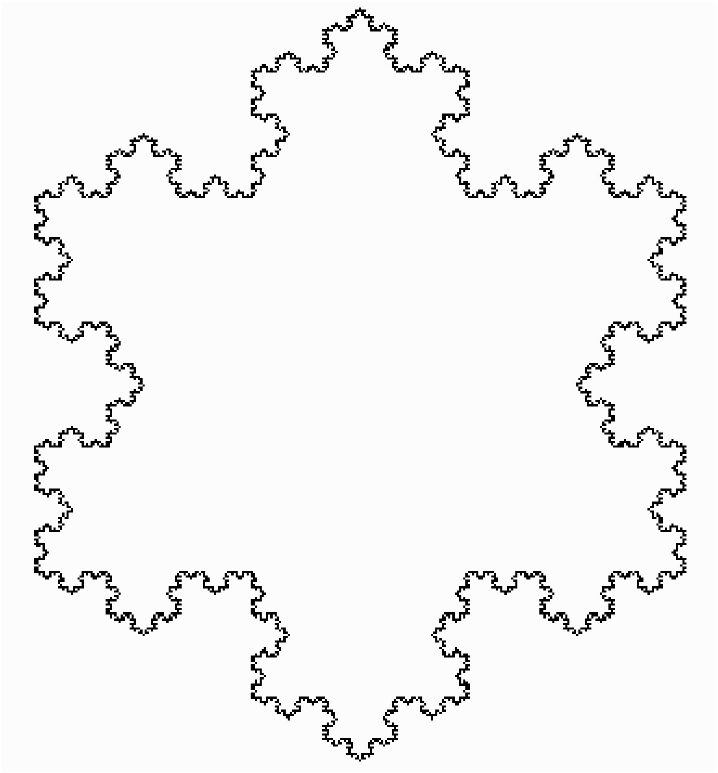
3-2.tree.py

间接递归

3.3 koch曲线与雪花



koch曲线



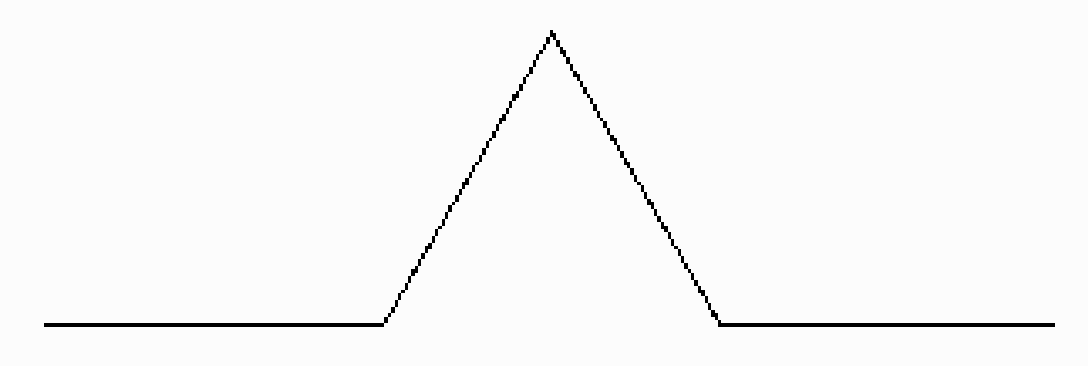
三条koch曲线组成的雪花图案

3-3 koch曲线

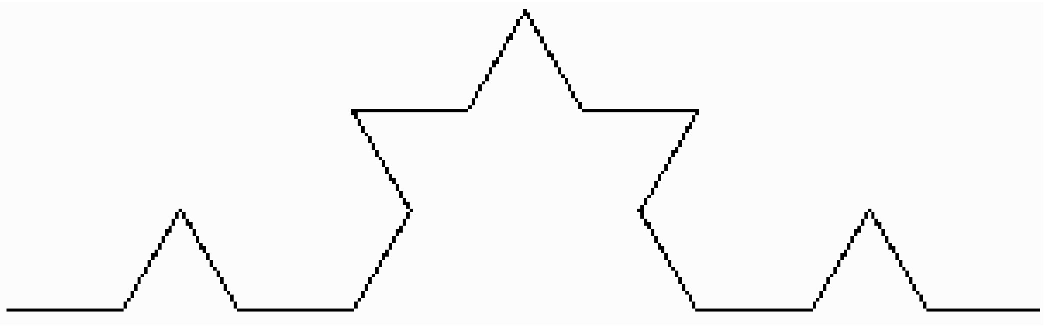
3.3 koch曲线与雪花

level=0

level=1



level=2

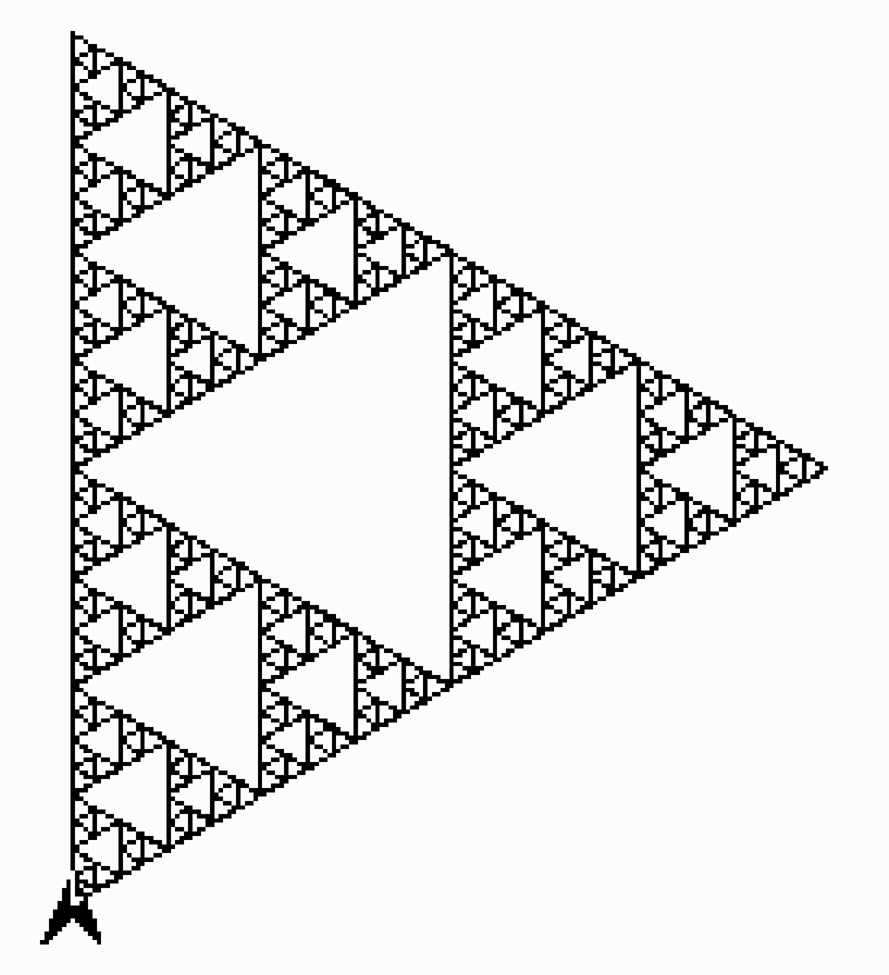


3.3 koch曲线与雪花

```
def koch(size, level):
    if level == 0:
        fd(size)
        return
    koch(size / 3, level - 1)
    lt(60)
    koch(size / 3, level - 1)
    rt(120)
    koch(size / 3, level - 1)
    lt(60)
    koch(size / 3, level - 1)
```

3-3.koch曲线与雪花.py

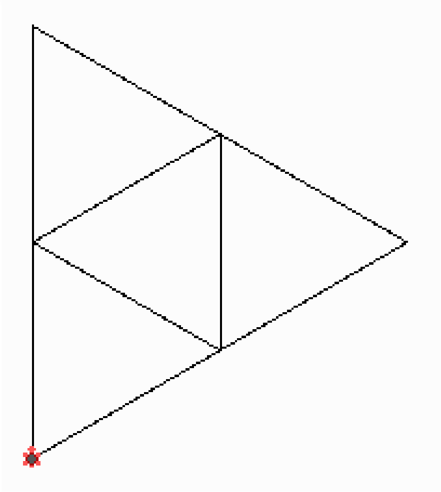
3.4 斯宾斯基三角形



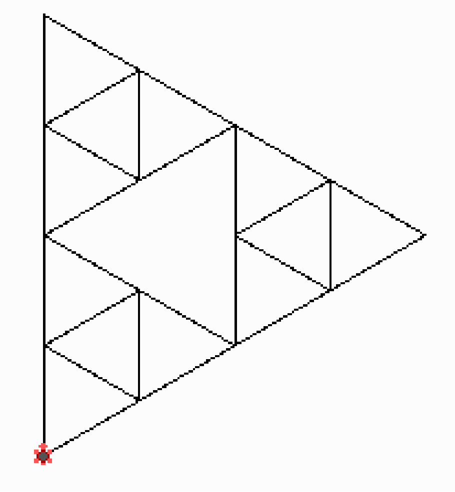
3-4 斯宾斯基三角形

3.4 斯宾斯基三角形

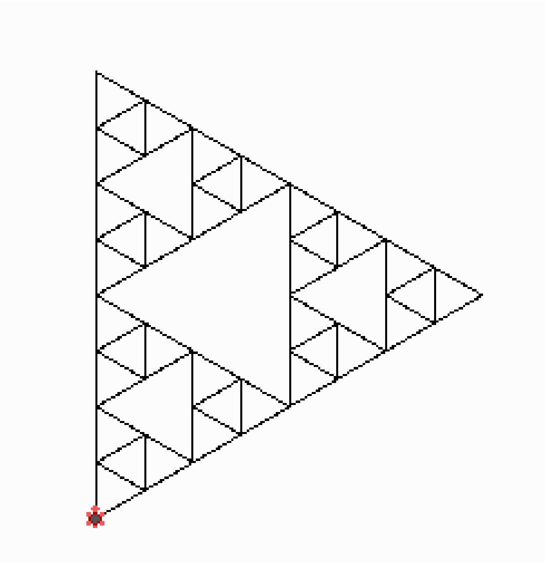
level=2



level=3



level=4



3.4 斯宾斯基三角形

```python
def nest_tri(size, level):
    if level == 0:
        return
    for i in range(3):
        nest_tri(size / 2, level - 1)
        fd(size)
        rt(120)
```

3-4.斯宾斯基三角形.py

编程思路总结

- 递归为我们提供了循环之外的另一种实现重复执行代码的方式（Map、Filter和Reduce只是把循环封装在了函数中）
- 递归和循环可以相互转换。
- 递归也给我们提供了另外一种解决问题的思路。
- 有些问题，使用递归来解决更加简单（如汉诺塔、访问目录中所有文件）。
- 循环比递归的运行效率更高，因此在能直接用循环解决问题的场合，应优先使用循环。
- 递归是继续学习树、图等部分高级数据结构的基础，因此必须掌握。

本章重点

- 递归及其应用

本章练习

- 见练习《第五章 递归》