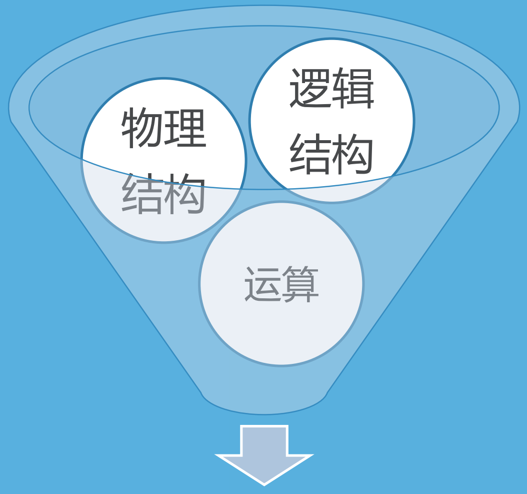


DSA 2025

数据结构与算法（Python）-04/递归

刘云淮 Yunhuai.liu@pku.edu.cn

http://www.yunhuai.net/DSA2025/CoursePage/DSA2025.html

北京大学计算机学院

目录

- 本章目标
- 什么是递归
- 实现递归
- 递归的可视化
- 复杂递归问题
- 动态规划Dynamic Programming

本章目标

- 了解某些难解的问题具有简单的递归解决法；
- 学会如何用递归方式写程序；
- 了解和应用递归的“三定律”；
- 了解递归是迭代iteration的一种形式；
- 实现问题的递归描述；
- 了解递归在计算机系统中是如何实现的。

从哆啦A梦说起

- 哆啦A梦要帮大雄写作业
- 作业太多，哆啦A梦找了2小时的**自己**
- 又分别找来了4小时、6小时以后的自己
- **分别**完成一部分作业

什么是递归Recursion?

- 递归是一种解决问题的方法，其精髓是将问题分解为规模更小的相同问题，持续分解，直到问题规模小到可以用非常简单直接的方式来解决。
- 递归的问题分解方式非常独特，其算法方面的明显特征就是：在算法流程中调用自身。
- 递归为我们提供了一种对复杂问题的优雅解决方案，精妙的递归算法常会出奇简单，令人赞叹。

初识递归：数列求和——从简单问题开始

- 问题：给定一个列表，返回列表中所有数的和

列表中数的个数不定，需要一个循环和一个累加变量来迭代求和

```python
def listsum(numList):
    theSum = 0
    for i in numList:
        theSum = theSum + i
    return theSum

print(listsum([1,3,5,7,9]))
```

- 程序很简单，但假如没有循环语句？

既不能用for，也不能用while

还能对不确定长度的列表求和么？

- 我们认识到求和实际上最终是由一次次的加法实现的，而加法函数恰有两个参数，这个是确定的。

- 看看怎么想办法，将问题规模较大的列表求和，分解为规模较小而且固定的两个数求和（加法）？

同样是求和问题，但规模发生了变化，符合递归解决问题的特征！

初识递归：数列求和

- 我们换一个方式来表达数列求和：全括号表达式

(1+(3+(5+(7+9))))

- 上面这个式子，最内层的括号(7+9)，这是无需循环即可计算的，实际上整个求和的过程是这样：

$$\begin{aligned} total &= (1+(3+(5+(7+9)))) \\ total &= (1+(3+(5+16))) \\ total &= (1+(3+21)) \\ total &= (1+24) \\ total &= 25 \end{aligned}$$

- 观察上述过程中所包含的重复模式，可以把求和问题归纳成这样：

数列的和= “首个数” + “余下数列” 的和

$$\underbrace{listSum(numList)}_{\text{问题}} = first(numList) + \underbrace{listSum(rest(numList))}_{\text{相同问题，规模更小}}$$

分解

初识递归：数列求和

- 上面的递归算法变成程序

最小规模

减小规模

调用自身

```python
def listsum(numList):
    if len(numList) == 1:
        return numList[0]
    else:
        return numList[0] + listsum(numList[1:])

print(listsum([1,3,5,7,9]))
```

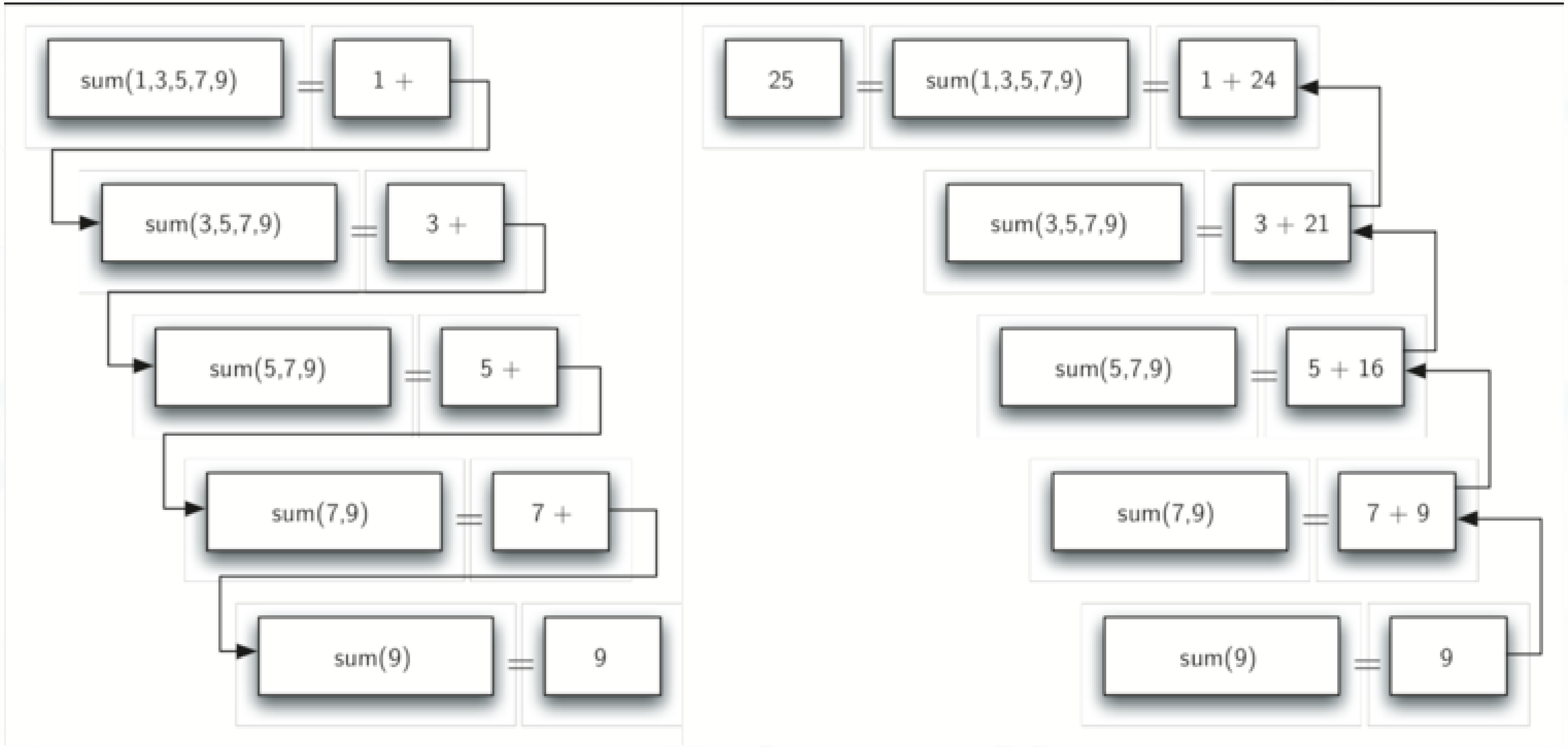
- 上面程序的要点：

1，问题分解为更小规模的相同问题，并表现为“调用自身”

2，对“最小规模”问题的解决：简单直接

递归程序如何被执行?

- 递归函数调用和返回过程的链条



递归“三定律”

- 为了向阿西莫夫的“机器人三定律”致敬，递归算法也总结出“三定律”

1，递归算法必须有一个基本结束条件（最小规模问题的直接解决）

2，递归算法必须能改变状态向基本结束条件演进（减小问题规模）

3，递归算法必须调用自身（解决减小了规模的相同问题）

递归“三定律”：数列求和问题

- 数列求和的递归算法首先具备了基本结束条件：当列表长度为1的时候，直接输出所包含的唯一数，使递归算法具备了最终出口

数列求和的基本结束条件还可以更简单，想一想是什么？

- 数列求和处理的数据对象是一个列表，而基本结束条件是长度为1的列表，那递归算法就要改变列表并向长度为1的状态演进，我们看到其具体做法是将列表长度减少1。
- 调用自身是递归算法中最难理解的部分，实际上我们理解为“问题分解成了规模更小的相同问题”就可以了，在数列求和算法中，就是“更短数列的求和问题”。

整数转换为任意进制

- 这个在数据结构栈里讨论过的算法，又回来了！

递归和栈，一定有关联

- 如果上次你被“入栈”“出栈”搞得挺晕乎的话，这次递归算法一定会让你感到清新

而且这次我们要解决从二进制到十六进制的任意进制转换

整数转换为任意进制

- 我们用最熟悉的十进制分析下这个问题

十进制有十个不同符号：convString = ″0123456789″

比十小的整数，转换成十进制，直接查表就可以了：convString[n]

想办法把比十大的整数，拆成一系列比十小的整数，逐个查表

比如七百六十九，拆成七、六、九，查表得到769就可以了

所以，在递归三定律里，我们找到了“基本结束条件”，就是小于十的整数

拆解整数的过程就是向“基本结束条件”演进的过程

整数转换为任意进制

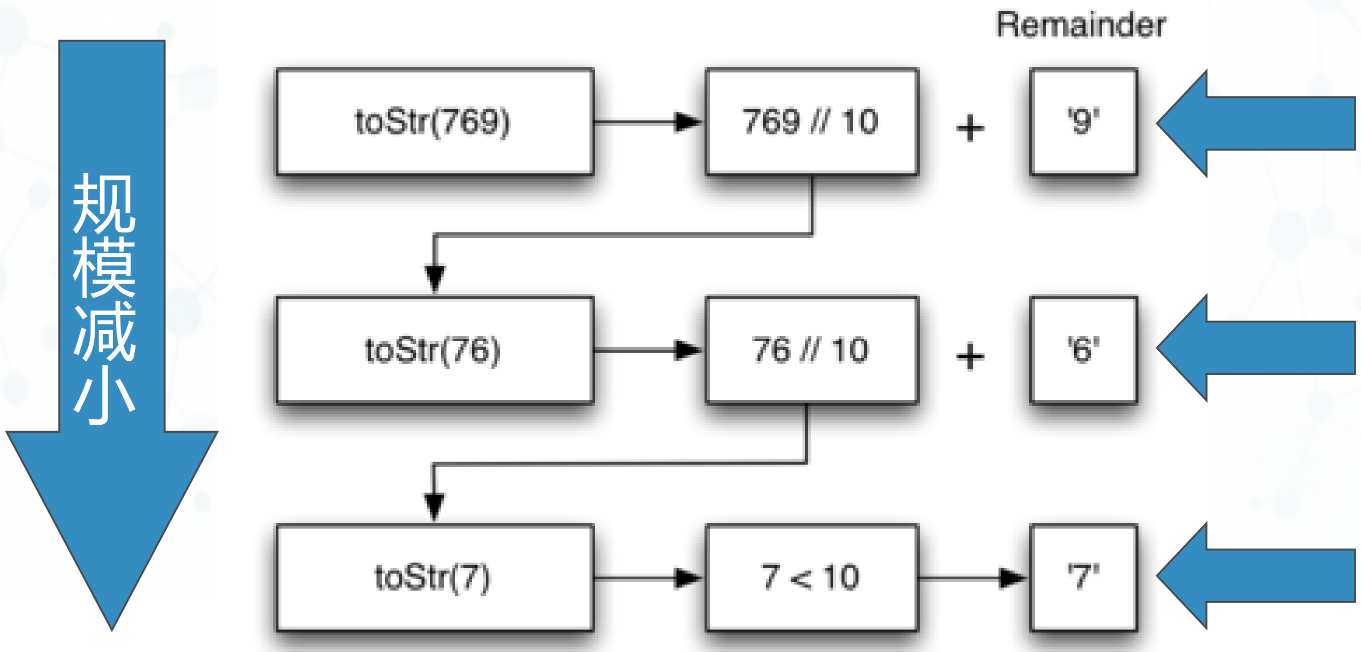
- 我们用整数除，和求余数两个计算来将整数一步步拆开

除以“进制基base”，和对“进制基”求余数

- 问题就分解为：

余数总小于“进制基base”，是“基本结束条件”，可直接进行查表转换

整数商成为“更小规模”问题，通过递归调用自身解决



整数转换为任意进制：代码

- 下面就是递归算法的代码

```python
def toStr(n,base):
    convertString = "0123456789ABCDEF"
    if n < base:
        return convertString[n]
    else:
        return toStr(n//base,base) + convertString[n%base]

print(toStr(1453,16))
```

最小规模

减小规模

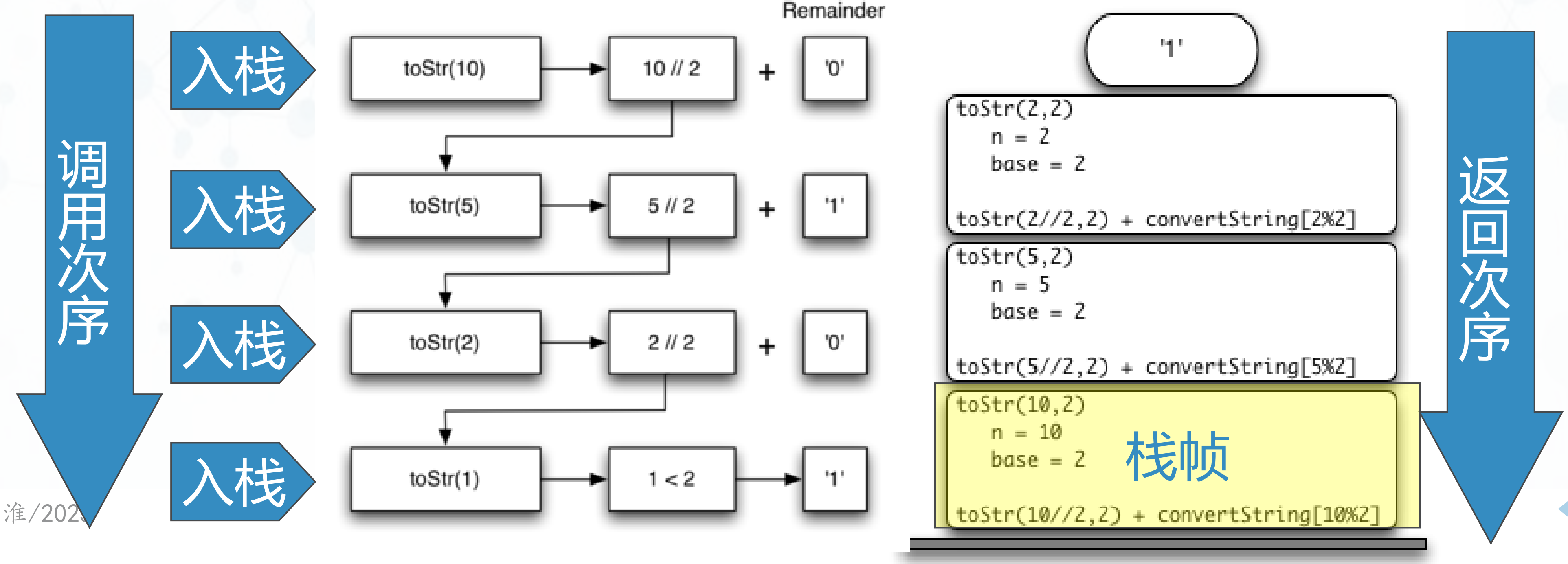
调用自身

递归调用的实现

- 当一个函数被调用的时候，系统会把调用时的现场（包括所有的局部变量，以及返回地址）压入到调用栈Call Stack

每次调用，压入栈的现场数据称为Stack Frame栈帧

- 当函数执行完成，返回时，要从调用栈的栈顶取得返回地址，把返回值放到栈顶，恢复现场，弹出栈帧，按地址返回。



递归的故事

- 一个古老的传说：有始无终的无尽递归

从前有座山，山上有座庙，庙里有个老和尚，他在讲：

从前有座山，山上有座庙，庙里有个老和尚，他在讲：

从前有座山，山上有座庙，庙里有个老和尚，他在讲：

……

- 前目的地. Predestination. 2014

自身产生自身的闭环烧脑递归

- 恐怖游轮. Triangle. 2009

调用栈栈帧大混合，如何才能终结一切，返回主函数？

- 爱奇艺有高清视频

随堂思考

- 编写将字符串反转的递归函数

```
def revstring(s):
```

- 编写回文词判断的递归函数

```
def palcheck(s):
```

递归可视化：图示

- 前面的种种递归算法展现了其简单而强大的一面，但还是难有个直观的概念
- 下面我们通过递归作图来展现递归调用的视觉影像
- Python的海龟作图系统turtle module

Python内置，随时可用，以LOGO语言的创意为基础

其意象为模拟海龟在沙滩上爬行而留下的足迹

爬行：forward(n)；backward(n)

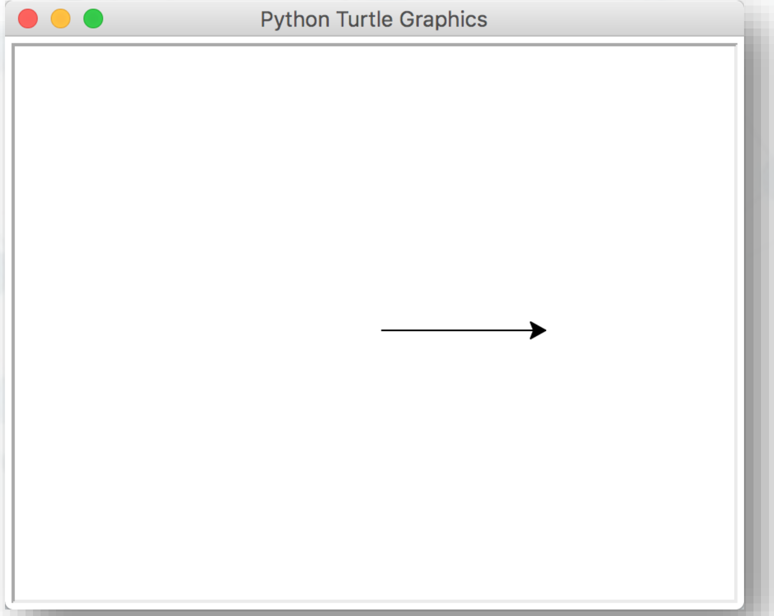
转向：left(a)；right(a)

笔触：penup()；pendown()；pensize()；pencolor(c)

递归可视化：图示

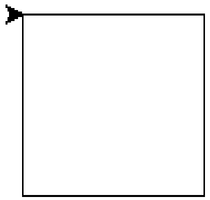
- 使用方法

```
import turtle #导入模块
t= turtle.Turtle() #生成一只海龟
w= turtle.Screen() #获取屏幕对象，用于最后的点击自动关闭窗□
t.forward(100)....... #指挥海龟作图
w.exitonclick() #作图完毕，欣赏结束后可以点击关闭窗□
```

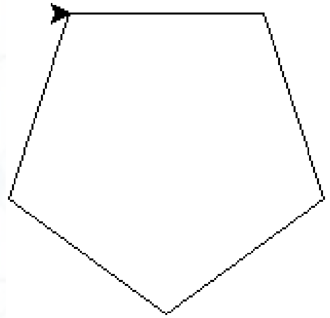


海龟作图

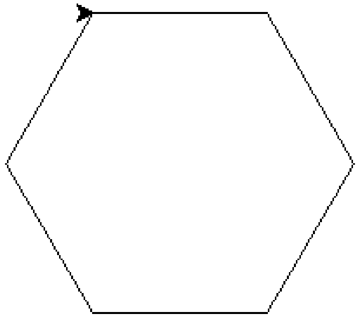
- 画正方形
- 画五边形
- 画六边形
- 画五角星



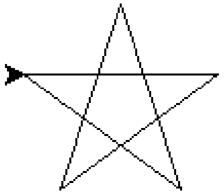
```python
import turtle
t= turtle.Turtle()
w= turtle.Screen()

for i in range(4):
    t.forward(100)
    t.right(90)

w.exitonclick()
```



```python
import turtle
t= turtle.Turtle()
w= turtle.Screen()

for i in range(5):
    t.forward(100)
    t.right(72)

w.exitonclick()
```



```python
import turtle
t= turtle.Turtle()
w= turtle.Screen()

for i in range(6):
    t.forward(100)
    t.right(60)

w.exitonclick()
```



```python
import turtle
t= turtle.Turtle()
w= turtle.Screen()

for i in range(5):
    t.forward(100)
    t.right(144)

w.exitonclick()
```

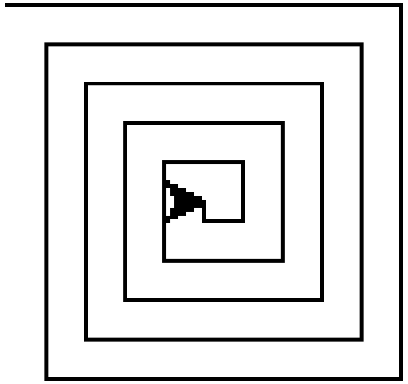
一个递归作图的例子：螺旋


```python
import turtle

myTurtle = turtle.Turtle()
myWin = turtle.Screen()

def drawSpiral(myTurtle, lineLen):
    if lineLen > 0:
        myTurtle.forward(lineLen)
        myTurtle.right(90)
        drawSpiral(myTurtle,lineLen-5)

drawSpiral(myTurtle,100)
myWin.exitonclick()
```

最小规模，0直接退出

减小规模，边长减5

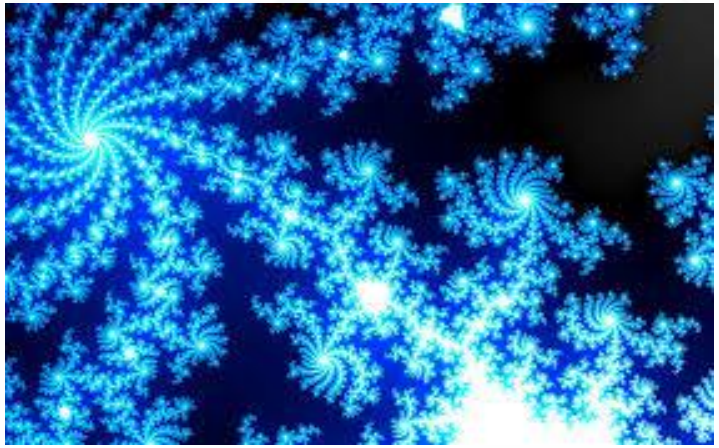
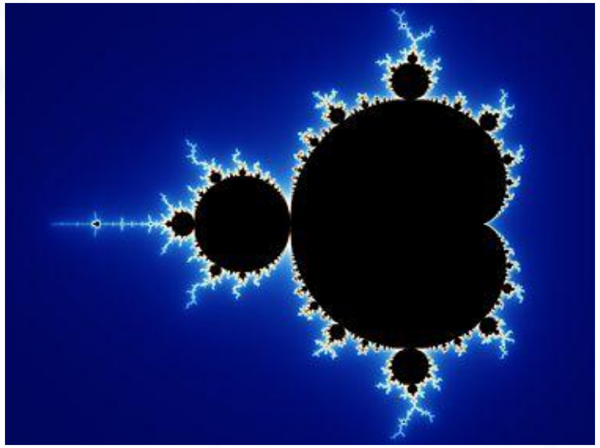
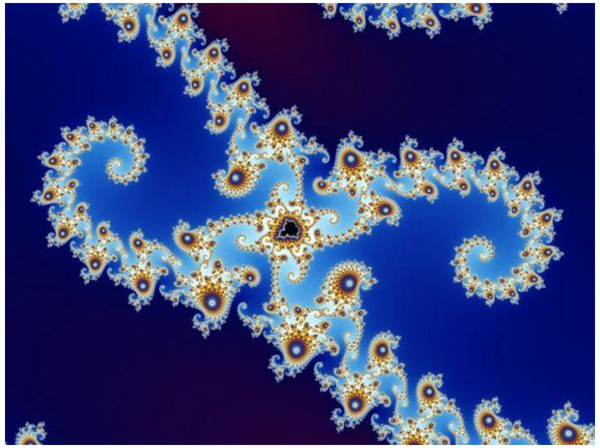
调用自身



分形树：自相似递归图形

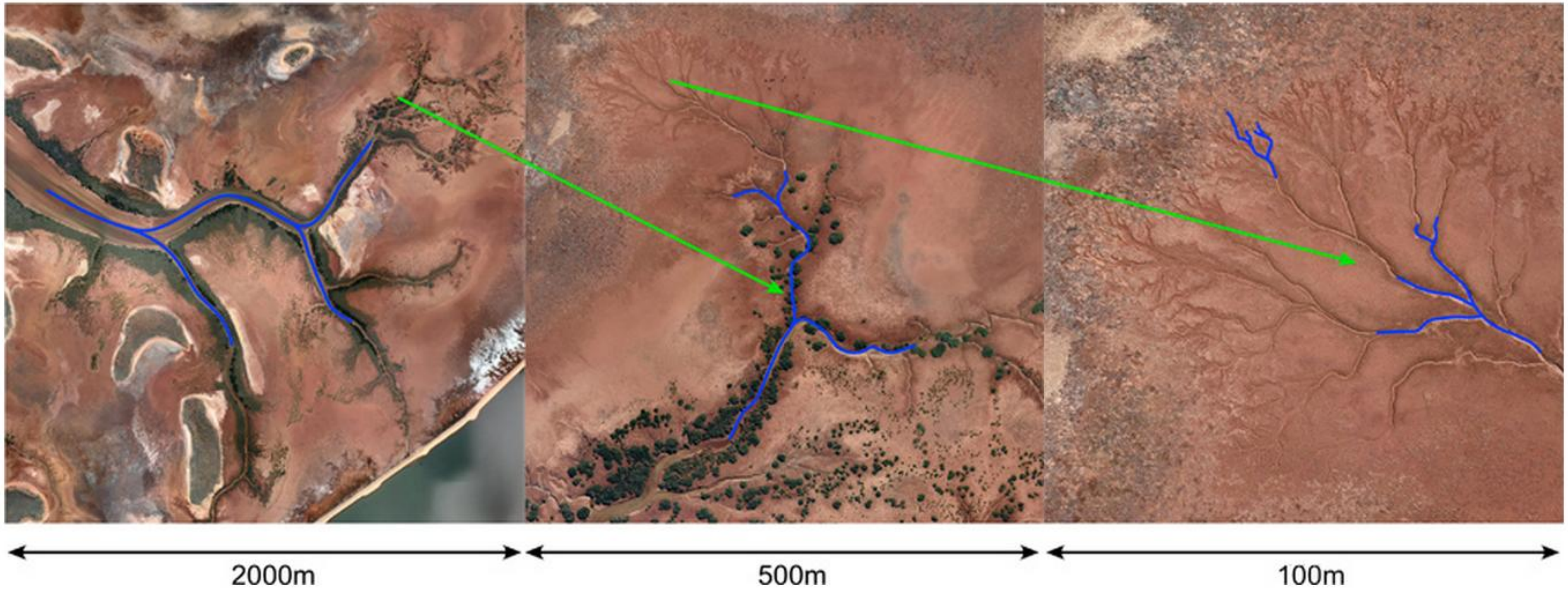
- 分形Fractal，是1975年由Mandelbrot开创的新学科
- 通常被定义为“一个粗糙或零碎的几何形状，可以分成数个部分，且每一部分都（至少近似地）是整体缩小后的形状”，即具有自相似的性质。
- 自然界中能找到众多具有分形性质的物体

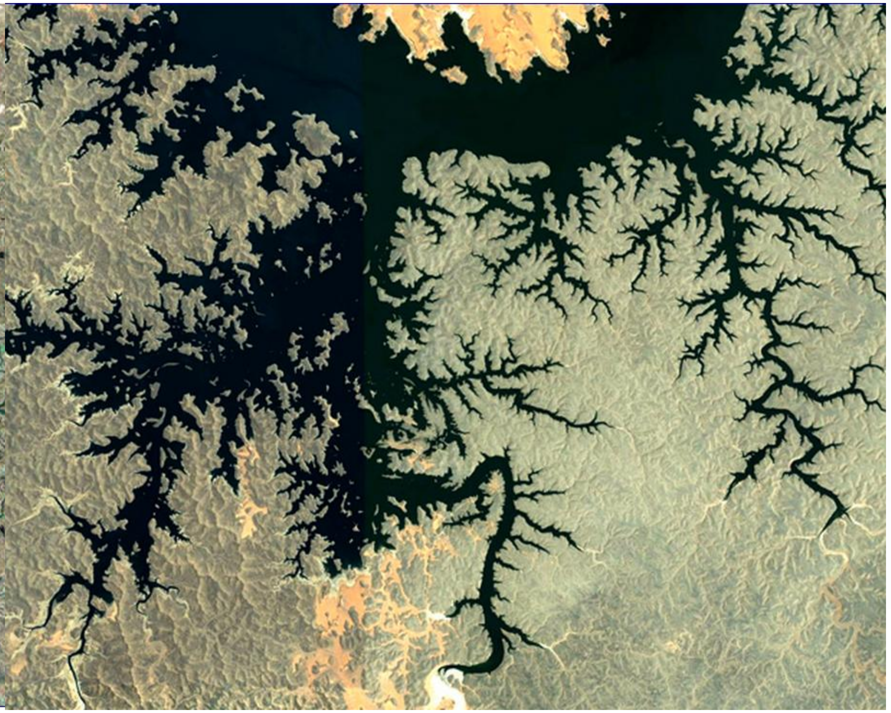
海岸线、山脉、闪电、云朵、雪花、树

分形树：自相似递归图形

- 【【分形】一根线，能折叠出整个宇宙吗？】

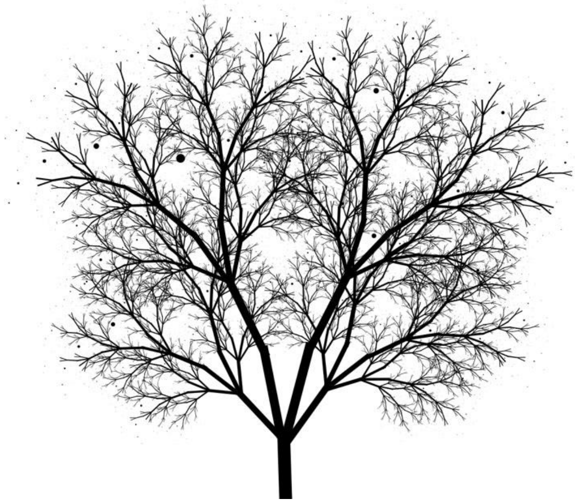

2000m
500m
100m



分形树：自相似递归图形

- 自然现象中所具备的分形特性，使得计算机可以通过分形算法生成非常逼真的自然场景，下面我们以树为例做一个粗糙的近似
- 分形是在不同尺度上都具有相似性的事物，将这种观点放在对树的观察上，我们就能看出，一棵树的每个分叉和每条树枝，实际上都具有整棵树的外形特征（也是逐步分叉的）
- 这样，我们可以把树分解为三个部分：树干、左边的小树、右边的小树

这样的分解，正好符合递归的定义：对自身的调用



分形树：代码

```
import turtle

def tree(branchLen,t):
    if branchLen > 5:
        t.forward(branchLen)
        t.right(20)
        tree(branchLen-15,t)
        t.left(40)
        tree(branchLen-15,t)
        t.right(20)
        t.backward(branchLen)
```

海龟

画树干

最小规模，0直接退出

右倾20度

减小规模，树干减15

回左倾40度，即左20

减小规模，树干减15

回右倾20度，即回正

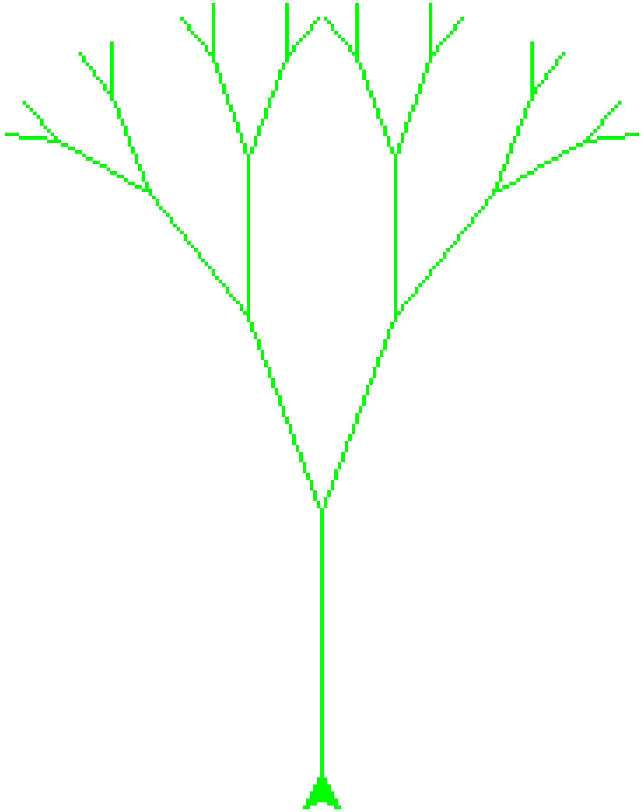
调用自身

海龟回到原位置

分形树：运行

- 注意海龟作图的次序

先画树干，再画右树枝，最后画左树枝：与递归函数里的流程一致



生成海龟

海龟位置调整

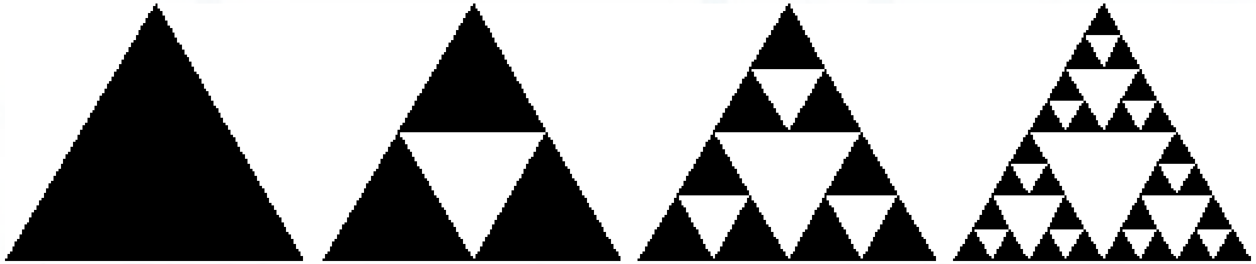
画树，树干长度75

```python
def main():
    t = turtle.Turtle()
    myWin = turtle.Screen()
    t.left(90)
    t.up()
    t.backward(100)
    t.down()
    t.color("green")
    tree(75,t)
    myWin.exitonclick()

main()
```

谢尔宾斯基三角形Sierpinski Triangle

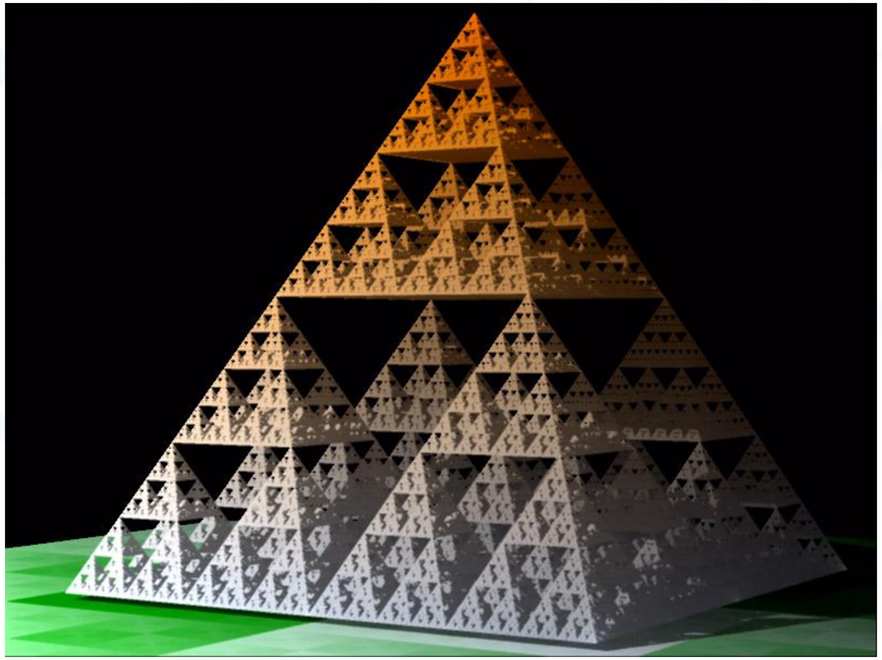
- 分形构造，平面称谢尔宾斯基三角形，立体称谢尔宾斯基金字塔

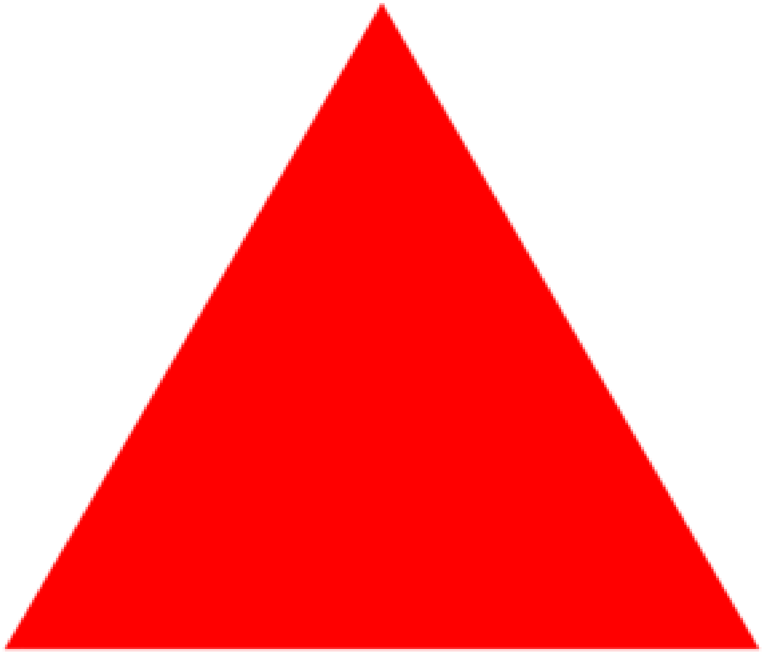
实际上，真正的谢尔宾斯基三角形是完全不可见的，其面积为0，但周长无穷，是介于一维和二维之间的分数维（约1.585维）构造。



degree=3

degree->∞

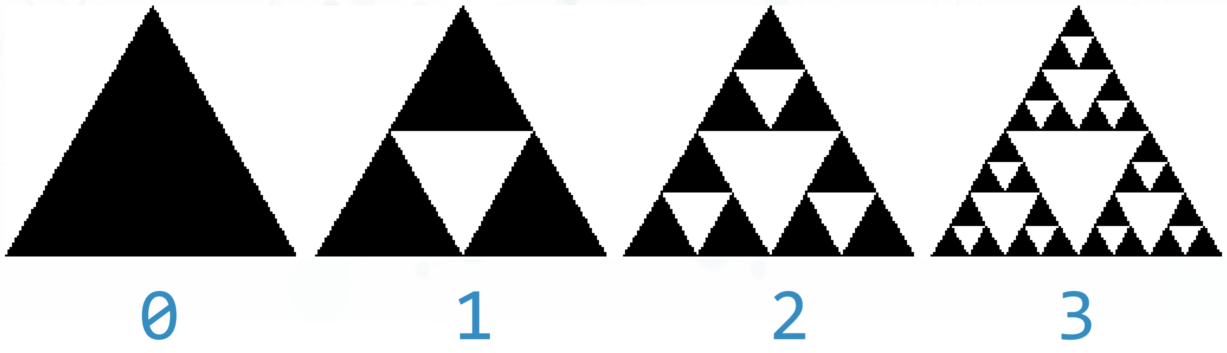




谢尔宾斯基三角形：作图思路

- 首先，根据自相似特性，谢尔宾斯基三角形是由3个相同的谢尔宾斯基三角形按照品字形拼叠而成。
- 由于我们无法真正做出谢尔宾斯基三角形（degree->∞），只能做degree有限的近似图形。
- 在degree有限的情况下，degree=n的三角形，是由3个degree=n-1的三角形按照品字形拼叠而成，同时，这3个degree=n-1的三角形边长均为degree=n的三角形的一半（规模减小）。

当degree=0，则就是一个等边三角形，这是递归基本结束条件



谢尔宾斯基三角形：代码


```python
def sierpinski(points,degree,myTurtle):
    colormap = ['blue','red','green','white','yellow', \
                'violet','orange']
    drawTriangle(points,colormap[degree],myTurtle)
    if degree > 0:
        sierpinski([points[0], \
                        getMid(points[0], points[1]), \
                        getMid(points[0], points[2])], \
                        degree-1, myTurtle)
        sierpinski([getMid(points[0], points[1]), \
                        points[1], \
                        getMid(points[1], points[2])], \
                        degree-1, myTurtle)
        sierpinski([getMid(points[0], points[2]), \
                        getMid(points[2], points[1]), \
                        points[2]], \
                        degree-1, myTurtle)
```

等边三角形（左上右顶点次序）

最小规模，0直接退出

减小规模：
getMid边长减半

调用自身，左上右次序

谢尔宾斯基三角形：代码

```python
import turtle

def drawTriangle(points,color,myTurtle):
    myTurtle.fillcolor(color)
    myTurtle.up()
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.down()
    myTurtle.begin_fill()
    myTurtle.goto(points[1][0],points[1][1])
    myTurtle.goto(points[2][0],points[2][1])
    myTurtle.goto(points[0][0],points[0][1])
    myTurtle.end_fill()

def getMid(p1,p2):
    return ( (p1[0]+p2[0]) / 2, (p1[1] + p2[1]) / 2)

def main():
    myTurtle = turtle.Turtle()
    myWin = turtle.Screen()
    myPoints = [[-200,-100],[0,200],[200,-100]]
    sierpinski(myPoints,3,myTurtle)
    myWin.exitonclick()

main()
```

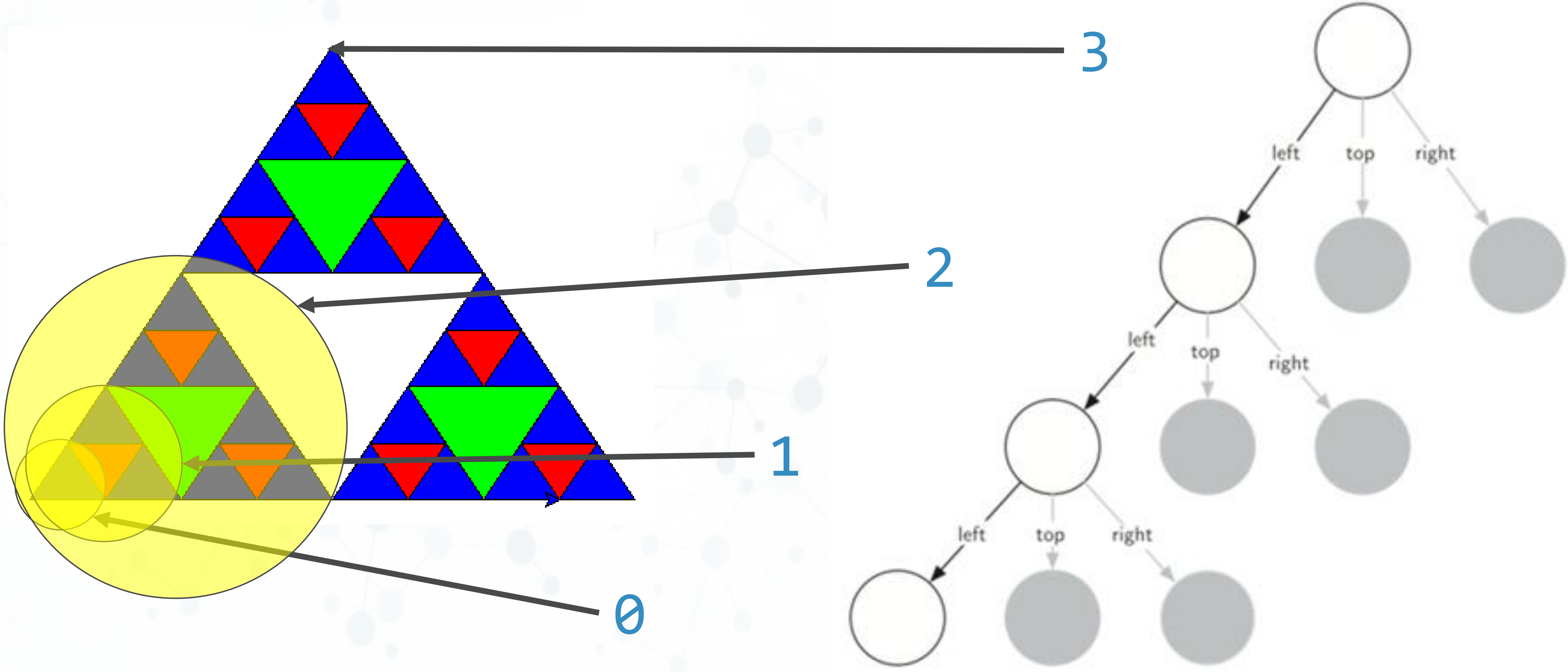
画指定顶点的等边三角形
指定填充颜色
1，抬笔到左顶点
2，落笔开始填充
3，到上顶点
4，到右顶点
5，回到左顶点闭合

取两个点的中点

外轮廓的3个顶点

画degree=3的三角形

谢尔宾斯基三角形：递归调用过程



复杂递归问题：河内塔Tower of Hanoi

- 河内塔问题是法国数学家Edouard Lucas于1883年，根据传说提出来的。
- 传说在一个印度教寺庙里，有3根柱子，其中一根套着64个由小到大的黄金盘片，僧侣们的任务就是要把这一叠黄金盘从一根柱子搬到另一根，但有两个规则：

一次只能搬1个盘子

大盘子不能叠在小盘子上

- 神的旨意是说，一旦这64个盘子完成迁移：

寺庙将会坍塌

世界将会毁灭……

- 神的旨意是千真万确的！

河内塔（汉诺塔）问题：3盘片演示

河内塔（汉诺塔）问题：4盘片演示

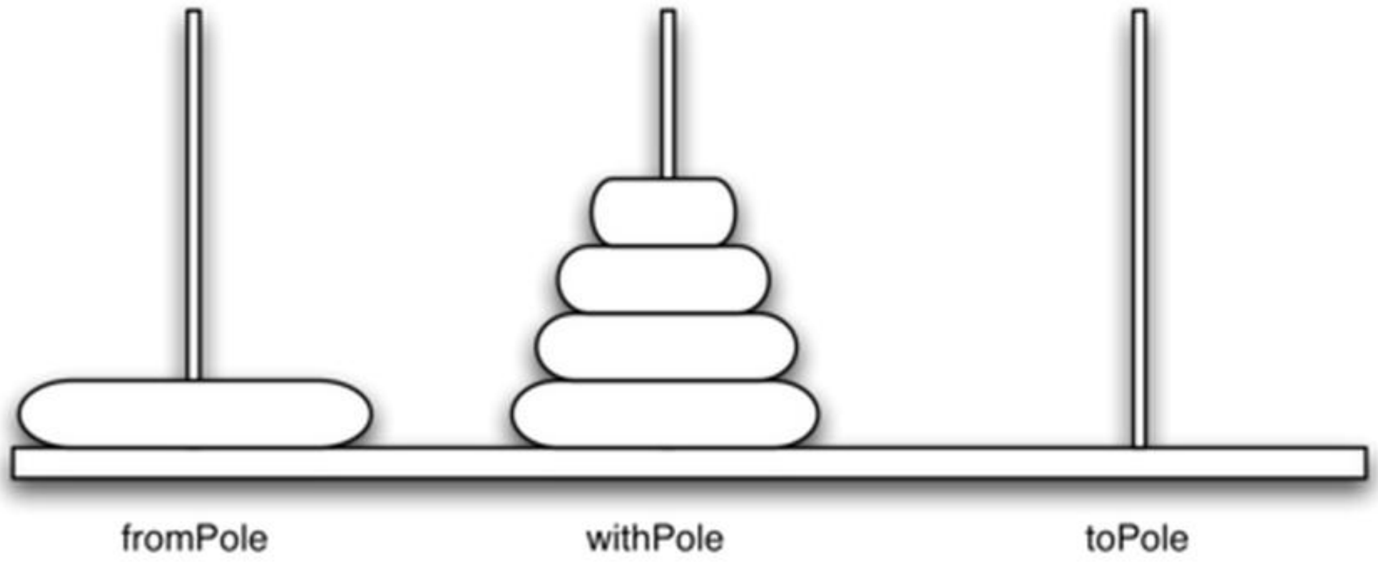
河内塔问题

- 虽然这些黄金盘片跟世界末日有着神秘的联系，但我们却不必太担心，据计算，要搬完这64个盘片：

需要的移动次数为$2^{64}-1$=18,446,744,073,709,551,615次

如果每秒钟搬动一次，则需要584,942,417,355（五千亿）年！

- 问题如何分解为递归形式？



河内塔问题：分析

- 我们还是从递归三定律来分析河内塔问题

基本结束条件（最小规模问题），如何减小规模，调用自身

- 假设我们有5个盘子，穿在1#柱，需要挪到3#柱

如果能把上面的一摞4个盘子统统挪到2#柱，那问题就好解决了：

把剩下的最大号盘子直接从1#柱挪到3#柱，再用同样的办法把2#柱上的那一摞4个盘子挪到3#柱，就完成了整个移动

- 接下来的问题就是4个盘子怎么从1#挪到2#？

此时问题规模已经减小！

同样是想办法把上面的一摞3个盘子挪到3#柱，把剩下最大号盘子从1#挪到2#柱，再想办法把一摞3个盘子从3#挪到2#柱

河内塔问题：分析

- 一摞3个盘子的挪动也照此：分为上面一摞2个，和下面最大号盘子
- 那么2个盘子怎么移动呢？（*啥？这还要想？*）
- （实在想不出，那么）最后是1个盘子的移动……

河内塔问题：递归思路

- 将盘片塔从开始柱，经由中间柱，移动到目标柱：

首先将上层N-1个盘片的盘片塔，从开始柱，经由目标柱，移动到中间柱；

然后将第N个（最大的）盘片，从开始柱，移动到目标柱；

最后将放置在中间柱的N-1个盘片的盘片塔，经由开始柱，移动到目标柱。

- 基本结束条件，也就是最小规模问题是：1个盘片的移动问题
- 上面的思路用Python写出来，几乎跟语言描述一样：

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)
```

河内塔问题：代码

```
def moveTower(height,fromPole, toPole, withPole):
    if height >= 1:
        moveTower(height-1,fromPole,withPole,toPole)
        moveDisk(fromPole,toPole)
        moveTower(height-1,withPole,toPole,fromPole)

def moveDisk(fp,tp):
    print "moving disk from",fp,"to",tp

moveTower(3,"A","B","C")
```

最小规模，0直接退出

减小规模：
盘片数量减1

调用自身，两次腾挪

```
>>>
moving disk from A to B
moving disk from A to C
moving disk from B to C
moving disk from A to B
moving disk from C to A
moving disk from C to B
moving disk from A to B
```

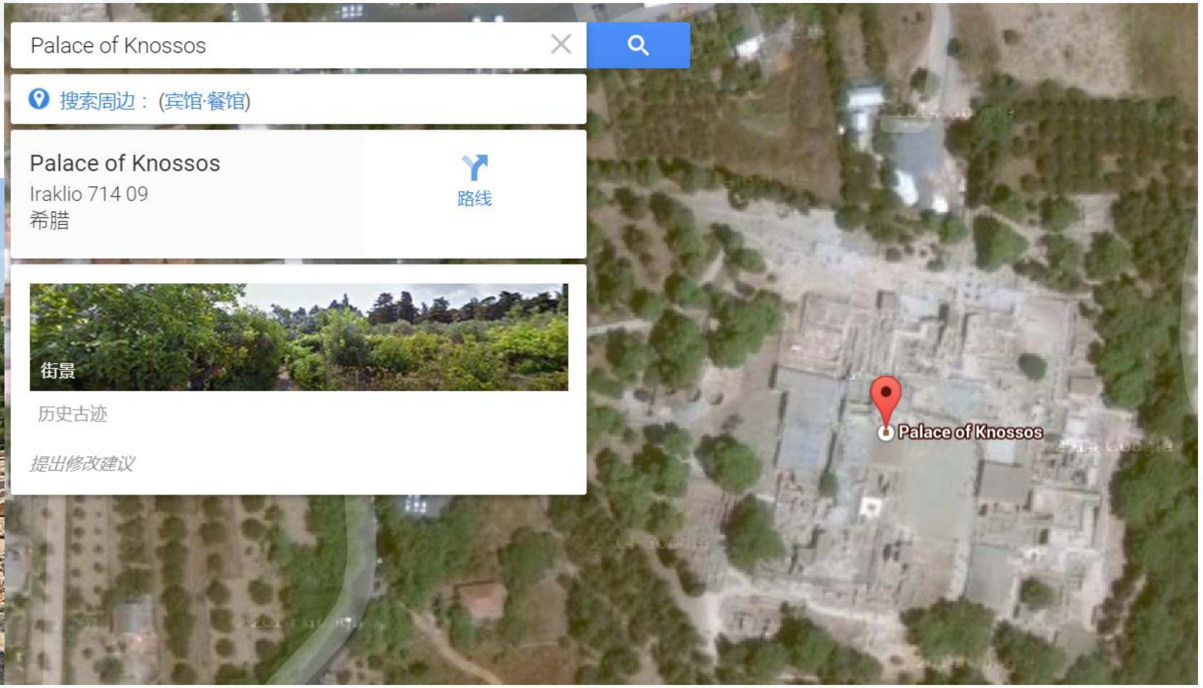
思考：找到本问题的非递归算法，试着看懂它。

探索迷宫：古希腊的迷宫

- 古希腊克里特岛米诺斯王
- 牛头人身怪物米诺陶洛斯

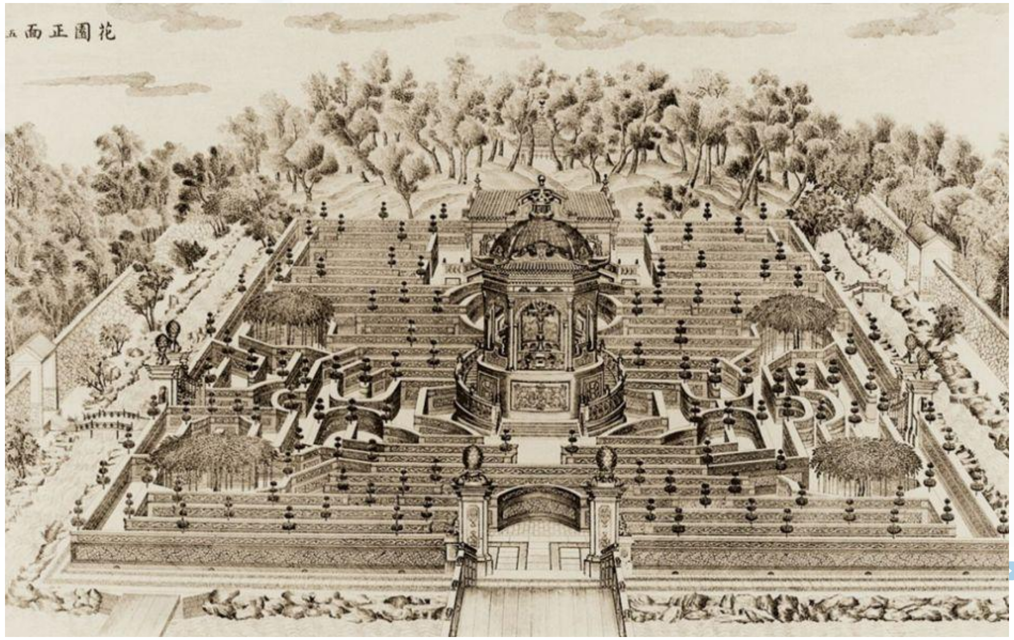
童男童女献祭，雅典王子忒修斯

- 公主，利剑，线团
- 老国王投海
- 爱琴海



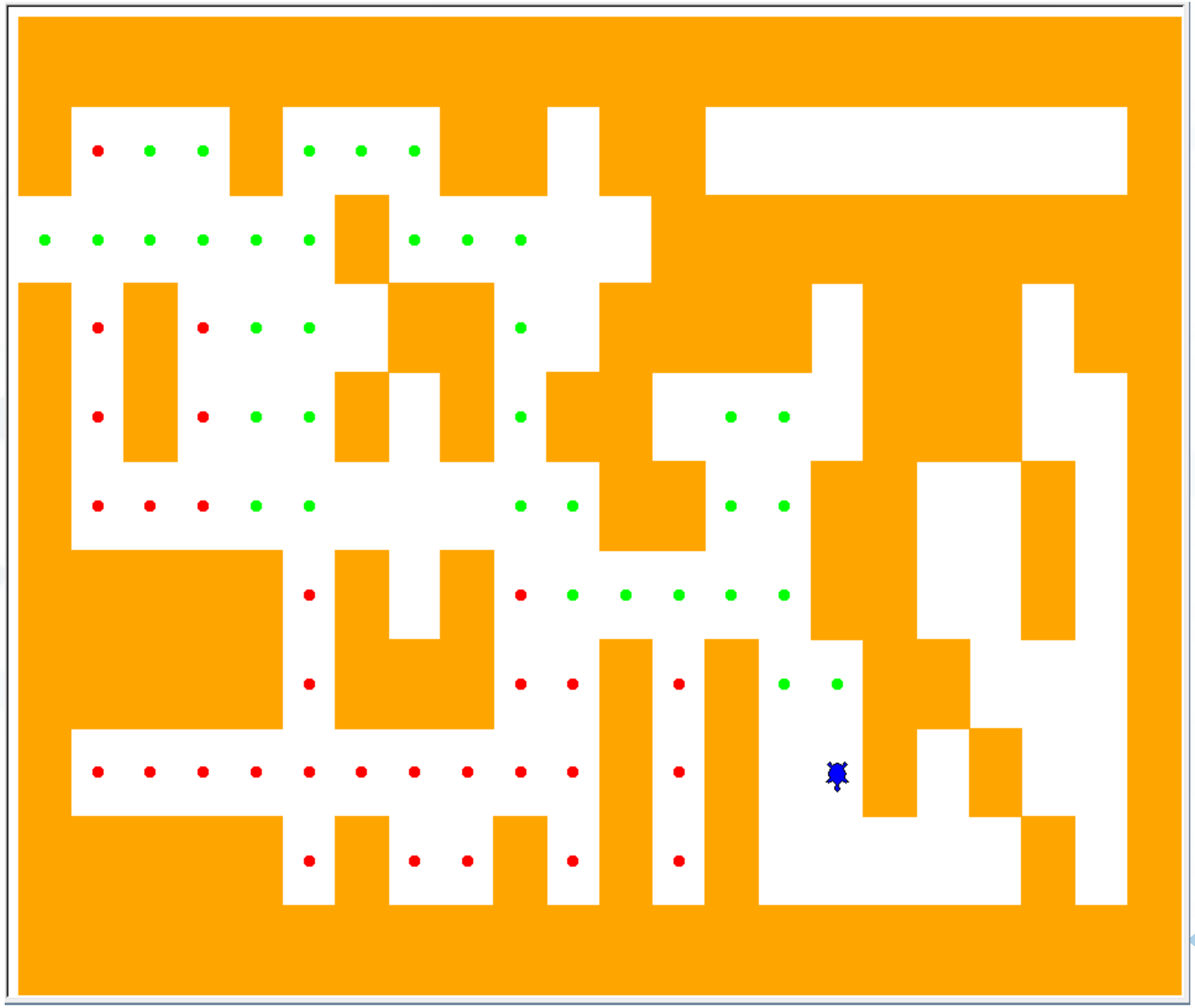
探索迷宫：圆明园的黄花阵

- 位于圆明园西洋楼景区



探索迷宫

- 我们将海龟放在迷宫中间，看它如何能找到出口



迷宫的数据结构

- 首先，我们将整个迷宫的空间（矩形）分为行列整齐的方格，区分出墙壁和通道。

给每个方格具有行列位置，并赋予“墙壁”、“通道”的属性

- 考虑用矩阵方式来实现迷宫数据结构

采用“数据项为字符列表的列表”这种两级列表的方式来保存方格内容

采用不同字符来分别代表“墙壁”“通道”“海龟投放点”

  - +：墙壁
  - 空格：通道
  - S：海龟

从一个文本文件逐行读入迷宫数据

maze2.tx

```
++++++++++++++++++++++
+   +   ++ ++        +
      +     ++++++++++
+ +    ++  ++++ +++ ++
+ +   + + ++    +++  +
+          ++  ++  + +
+++++ + +      ++  + +
+++++ +++  + +  ++   +
+          + + S+ +  +
+++++ +  + + +     + +
++++++++++++++++++++++
```

迷宫的数据结构：Maze Class

- Maze Class的成员

mazelist：保存矩阵

rowsInMaze：矩阵的行数

columnsInMaze：矩阵的列数

- 读入数据文件成功后

mazelist如下图示意

mazelist[row][col]==’+’

```
[ ['+','+','+','+',...,'+','+','+','+','+','+','+'],
  ['+',' ',' ',' ',...,' ',' ',' ','+',' ',' ',' '],
  ['+',' ','+',' ',...,'+','+',' ','+',' ','+','+'],
  ['+',' ','+',' ',...,' ',' ',' ','+',' ','+','+'],
  ['+','+','+',' ',...,'+','+',' ','+',' ',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ',' ',' ','+'],
  ['+','+','+','+',...,'+','+','+','+','+',' ','+'],
  ['+',' ',' ',' ',...,'+','+',' ',' ','+',' ','+'],
  ['+',' ','+','+',...,' ',' ','+',' ',' ',' ','+'],
  ['+',' ',' ',' ',...,' ',' ','+',' ','+','+','+'],
  ['+','+','+','+',...,'+','+','+',' ','+','+','+']]
```

```python
class Maze:
    def __init__(self,mazeFileName):
        rowsInMaze = 0
        columnsInMaze = 0
        self.mazelist = []
        mazeFile = open(mazeFileName,'r')
        rowsInMaze = 0
        for line in mazeFile:
            rowList = []
            col = 0
            for ch in line[:-1]:
                rowList.append(ch)
                if ch == 'S':
                    self.startRow = rowsInMaze
                    self.startCol = col
                col = col + 1
            rowsInMaze = rowsInMaze + 1
            self.mazelist.append(rowList)
            columnsInMaze = len(rowList)
```

探索迷宫：算法思路

- 确定了迷宫数据结构之后，我们知道，对于海龟来说，其身处某个方格之中

它所能移动的方向（东南西北），必须是向着通道的方向

如果某个方向是墙壁方格，就要换一个方向移动

- 这样，探索迷宫并找到出口的递归算法思路如下：

将海龟从原位置向北移动一步，然后以海龟的新位置递归调用探索迷宫寻找出口；

如果上面的步骤找不到出口，那么将海龟从原位置向南移动一步，然后以海龟的新位置递归调用探索迷宫寻找出口；

如果向南还找不到出口，那么将海龟从原位置向西移动一步，然后以海龟的新位置递归调用探索迷宫寻找出口；

如果向西还找不到出口，那么将海龟从原位置向东移动一步，然后以海龟的新位置递归调用探索迷宫寻找出口；

如果上面四个方向都找不到出口，那么这个迷宫没有出口！

探索迷宫：算法思路

- 上面这个思路看起来很完美，但有些细节是至关重要：

如果我们向某个方向（如北）移动了海龟，那么如果新位置的北正好是一堵墙壁，那么在新位置上的递归调用就会让海龟向南尝试

可是新位置的南边一格，正好就是递归调用之前的原位置

这样就陷入了无限递归的死循环之中

- 所以需要有个机制来记录海龟所走过的路径

沿途洒“面包屑”，一旦前进的方向发现“面包屑”，就不能再踩上去，而必须换下一个方向尝试

对于递归调用来说，就是某方向的方格上发现“面包屑”，就立即从递归调用返回上一级。

探索迷宫：算法思路

- 这样，我们对递归调用的“基本结束条件”归纳如下：

海龟碰到了“墙壁”方格，递归调用结束，返回失败；

海龟碰到了“面包屑”方格，表示此方格已访问过，递归调用结束，返回失败；

海龟碰到了“出口”方格，即“位于边缘的通道”方格，递归调用结束，返回成功！

海龟在四个方向上探索都失败，递归调用结束，返回失败

- 为了让海龟在迷宫图里跑起来，我们给迷宫数据结构Maze Class添加一些成员和方法

t：一个作图的海龟，设置其shape为海龟的样子（缺省是一个箭头）

drawMaze()：绘制出迷宫的图形，墙壁用实心方格绘制

updatePosition(row, col, val)：更新海龟的位置，并做标注

isExit(row, col)：判断是否“出口”

探索迷宫：算法代码

```
def searchFrom(maze, startRow, startColumn):
    # try each of four directions from this point until we find a way out.
    # base Case return values:
    #  1. We have run into an obstacle, return false
    maze.updatePosition(startRow, startColumn)
    if maze[startRow][startColumn] == OBSTACLE :
        return False
    #  2. We have found a square that has already been explored
    if maze[startRow][startColumn] == TRIED or maze[startRow][startColumn] == DEAD_END:
        return False
    # 3. We have found an outside edge not occupied by an obstacle
    if maze.isExit(startRow,startColumn):
        maze.updatePosition(startRow, startColumn, PART_OF_PATH)
        return True
    maze.updatePosition(startRow, startColumn, TRIED)
    # Otherwise, use logical short circuiting to try each dire
    # in turn (if needed)
    found = searchFrom(maze, startRow-1, startColumn) or \
            searchFrom(maze, startRow+1, startColumn) or \
            searchFrom(maze, startRow, startColumn-1) or \
            searchFrom(maze, startRow, startColumn+1)
    if found:
        maze.updatePosition(startRow, startColumn, PART_OF_PA
    else:
        maze.updatePosition(startRow, startColumn, DEAD_END)
    return found
```

墙，失败

已过，失败

出口，成功

标注洒“面包屑”

北、南、西、东
依次尝试

成功，标注

失败，标注

由于or的短路算法，只要有一个方向返回成功，则后面的尝试就不需要做了，直接返回成功！

动态规划Dynamic Programming

- 计算机科学中许多程序都是为了找到某些问题的最优解

例如，两个点之间的最短路径

能最好匹配一系列点的直线

满足一定条件的最小集合

- 人们会采用各种策略来解决这些问题，我们这里介绍其中的一种策略：动态规划

- 优化问题的一个经典的案例是为找零兑换最少个数的硬币

假设你为一家自动售货机厂家编程序，自动售货机要每次找给顾客最少数量的硬币；

假设某次顾客投进$1纸币，买了¢37的东西，要找¢63，那么最少数量就是：两个quarter（¢25）、一个dime（¢10）和三个penny（¢1），一共6个

一般我们这么做：从最大面值的硬币开始，用尽量多的数量，有余额的，再到下一最大面值的硬币，还用尽量多的数量，一直到penny（¢1）为止

动态规划：兑换硬币

- 这种方法称为“贪心法Greedy Method”

因为我们每次都试图解决问题的尽量大的一部分

对应到兑换硬币问题，就是每次以最多数量的最大面值硬币来迅速减少找零

- “贪心法”在美元硬币体系（quarter/dime/nikel/penny）下表现尚好
- 但如果你的老板决定把自动售货机出口到Elbonia（系列漫画Dilbert里杜撰的国家），事情就会有点复杂，因为这个古怪的国家除了上面3种面值之外，还有一种【¢21】的硬币！
- 按照“贪心法”，在Elbonia，¢63还是原来的6个硬币
- 但实际上最优解是3个面值¢21的硬币！
- “贪心法”失效了。

Inflation in Elbonia

兑换硬币：递归解法

- 我们来找一种肯定能找到最优解的方法，既然本章是介绍递归，那这个解法肯定是递归的

- 首先是确定基本结束条件，兑换硬币这个问题最简单直接的情况就是，需要兑换的找零，其面值正好等于某种硬币

如找零25分，答案就是1个硬币！

- 其次是减小问题的规模，在美元硬币体系里，我们要尝试4次

找零减去1分(penny)后，求兑换硬币最少数量的解；

找零减去5分(nikel)后，求兑换硬币最少数量的解；

找零减去10分(dime)后，求兑换硬币最少数量的解；

找零减去25分(quarter)后，求兑换硬币最少数量的解；

上述4项中选择最小的一个。

$$numCoins = min \begin{cases} 1 + numCoins(originalamount - 1) \\ 1 + numCoins(originalamount - 5) \\ 1 + numCoins(originalamount - 10) \\ 1 + numCoins(originalamount - 25) \end{cases}$$

兑换硬币：递归解法代码

最小规模，直接返回

减小规模：
每次减去一种硬币面值
挑选最小数量

```python
def recMC(coinValueList,change):
    minCoins = change
    if change in coinValueList:
        return 1
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recMC(coinValueList,change-i)
            if numCoins < minCoins:
                minCoins = numCoins
    return minCoins

print(recMC([1,5,10,25],63))
```

调用自身

兑换硬币：递归解法分析

- 递归解法虽然能解决问题，但其最大的问题是：极！其！低！效！

对63分的兑换硬币问题，需要进行67,716,925次递归调用！

在我这台笔记本电脑上花费了半分钟时间得到解：6个硬币

- 以26分兑换硬币为例，看看递归调用过程（377次递归的一小部分）

节点是剩余的找零数额，边上标记了所取用的硬币面值

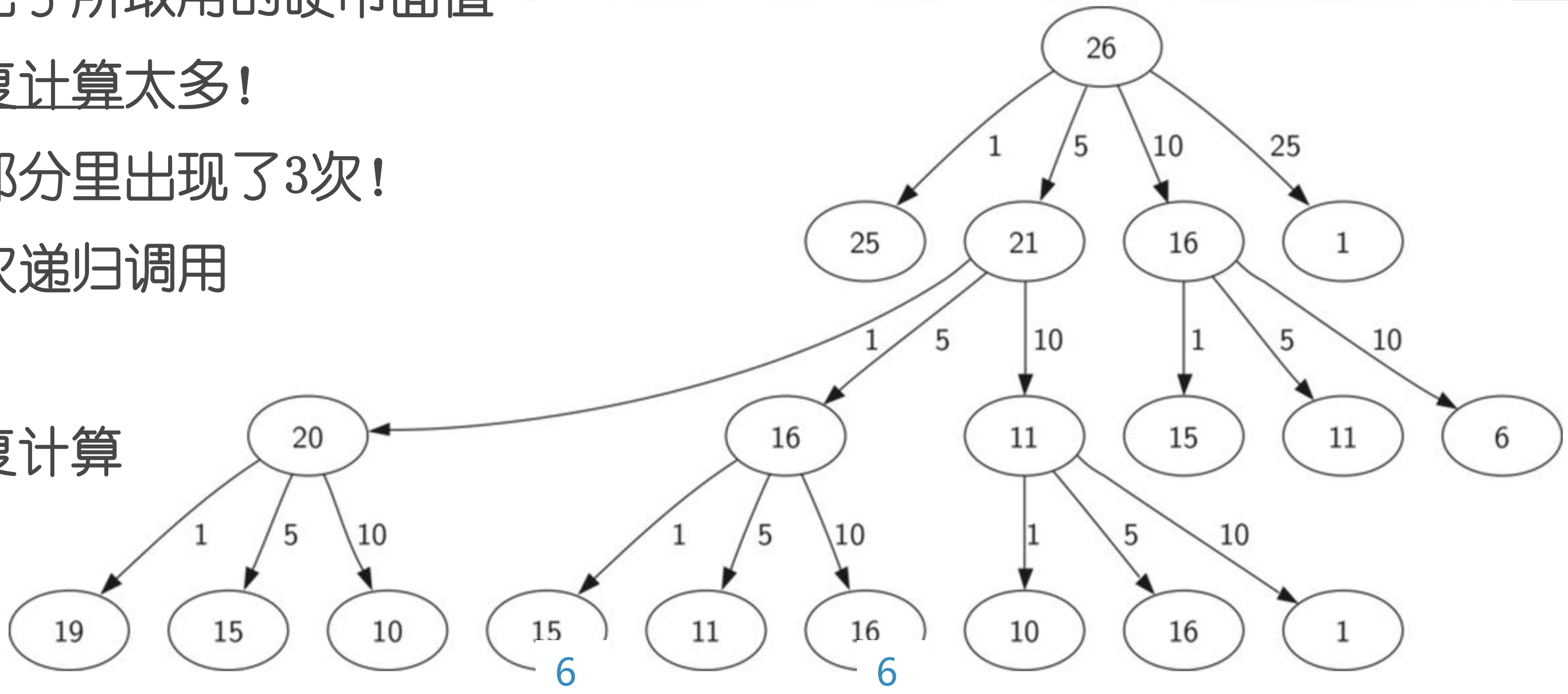
我们发现一个重大秘密，就是重复计算太多！

例如找零数额15分的，在这个小部分里出现了3次！

而找零数额15分最终解决还要52次递归调用

仅这一点就白费104次调用

很明显，这个算法致命缺点是重复计算



兑换硬币：递归解法改进

- 对这个递归解法进行改进的关键就在于消除重复计算

我们可以用一个表将计算过的中间结果保存起来，在计算之前查表看看是否已经计算过

- 这个算法的中间结果就是部分找零的最优解，在递归调用过程中已经得到的最优解被记录下来

在递归调用之前，先查找表中是否已有部分找零的最优解

如果有，直接返回最优解而不进行递归调用

如果没有，才进行递归调用

- 改进后的解法，极大减少了递归调用的次数

对63分的兑换硬币问题，仅仅需要221次递归调用，是改进前的三十万分之一，瞬间返回！

兑换硬币：递归解法改进代码

最小规模直接返回

查表命中直接返回

记录部分找零最优解

最小规模记录结果

```
def recDC(coinValueList,change,knownResults):
    minCoins = change
    if change in coinValueList:
        knownResults[change] = 1
        return 1
    elif knownResults[change] > 0:
        return knownResults[change]
    else:
        for i in [c for c in coinValueList if c <= change]:
            numCoins = 1 + recDC(coinValueList, change-i,\
                                 knownResults)
            if numCoins < minCoins:
                minCoins = numCoins
                knownResults[change] = minCoins
    return minCoins

print(recDC([1,5,10,25],63,[0]*64))
```

兑换硬币：动态规划解法

- 虽然上述的改进可以很好地解决兑换硬币的问题，但记录中间结果的表有不少未定义的“空洞”，实际上，这种方法还不能称为动态规划，而是利用了一种叫做“memoization（函数值缓存）”的技术提高了递归解法的性能，或者一般称为“caching（缓存）”
- 动态规划算法采用了一种更有条理的方式来得到问题的解
- 兑换硬币的动态规划算法从最简单的“1分钱找零”的最优解开始，逐步递加上去，直到我们需要的找零钱数
- 在找零递加的过程中，设法保持每一分钱的递加都是最优解，一直加到求解找零钱数，自然得到最优解
- 递加的过程能保持最优解的关键是，其依赖于更少钱数最优解的简单计算，而更少钱数的最优解已经得到了。

主定理

主定理：设$a\geq 1, b>1$为常数，$f(n)$为函数，$T(n)$为非负整数，且

$$T(n) = aT(n/b) + f(n)$$

则有以下结果：

1. 若$f(n) = O(n^{\log_b a-\varepsilon}), \varepsilon > 0$，那么$T(n) = \Theta(n^{\log_b a})$
2. 若$f(n) = \Theta(n^{\log_b a})$，那么$T(n) = \Theta(n^{\log_b a} \log n)$
3. 若$f(n) = \Omega(n^{\log_b a+\varepsilon}), \varepsilon > 0$，且对于某个常数 $c < 1$和充分大的n有 $a f(n/b) \leq cf(n)$，那么 $T(n)=\Theta(f(n))$

兑换硬币：动态规划算法

- 我们来看看如何采用动态规划来解决11分钱的兑换问题

从1分钱兑换开始

逐步建立一个兑换表

到5分钱兑换有两个选择：

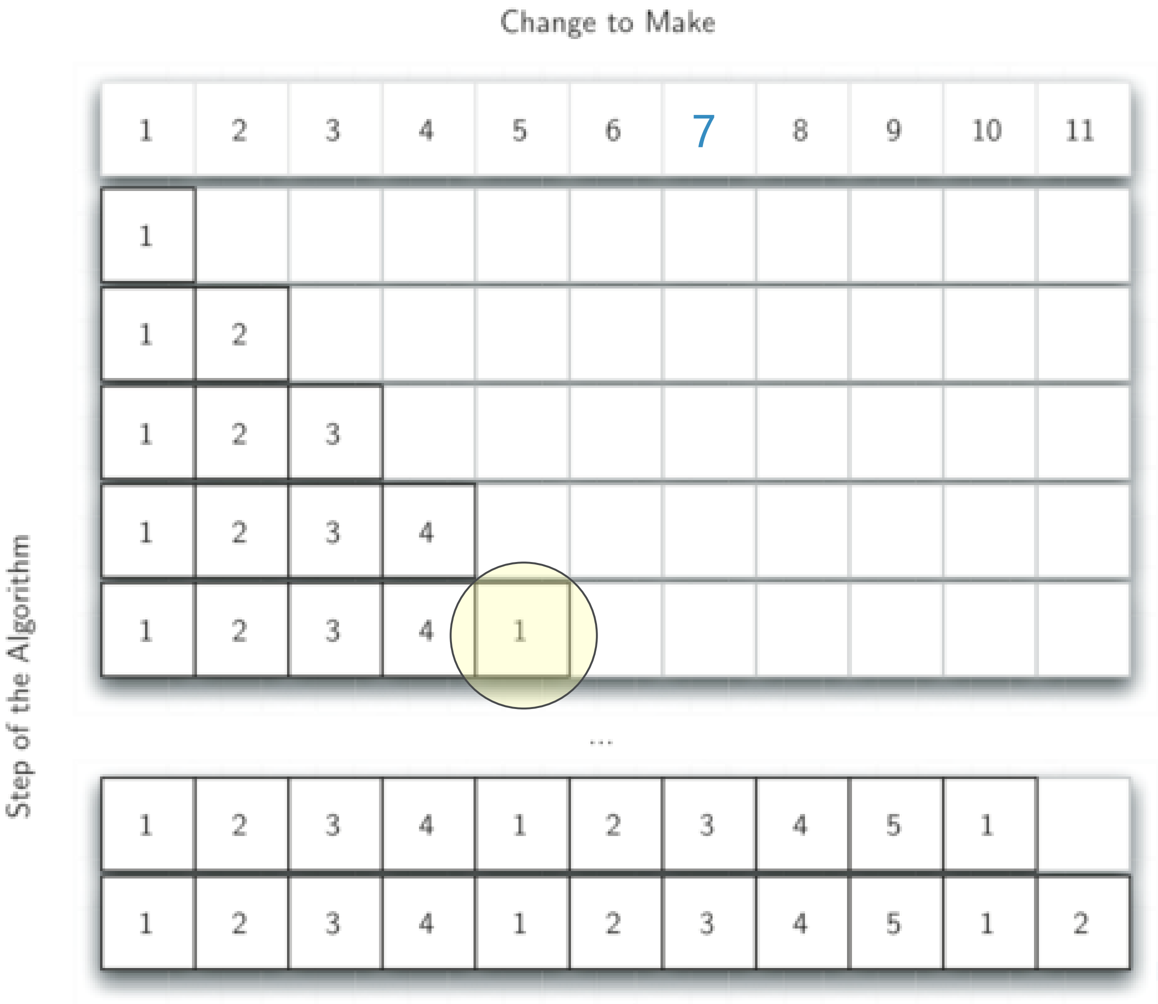
<u>5个1分或者1个5分</u>硬币

显然1个是最优解

所以将1填入表中

那么更多的钱数是如何得到解？

下面我们看11分钱的过程



兑换硬币：动态规划解法

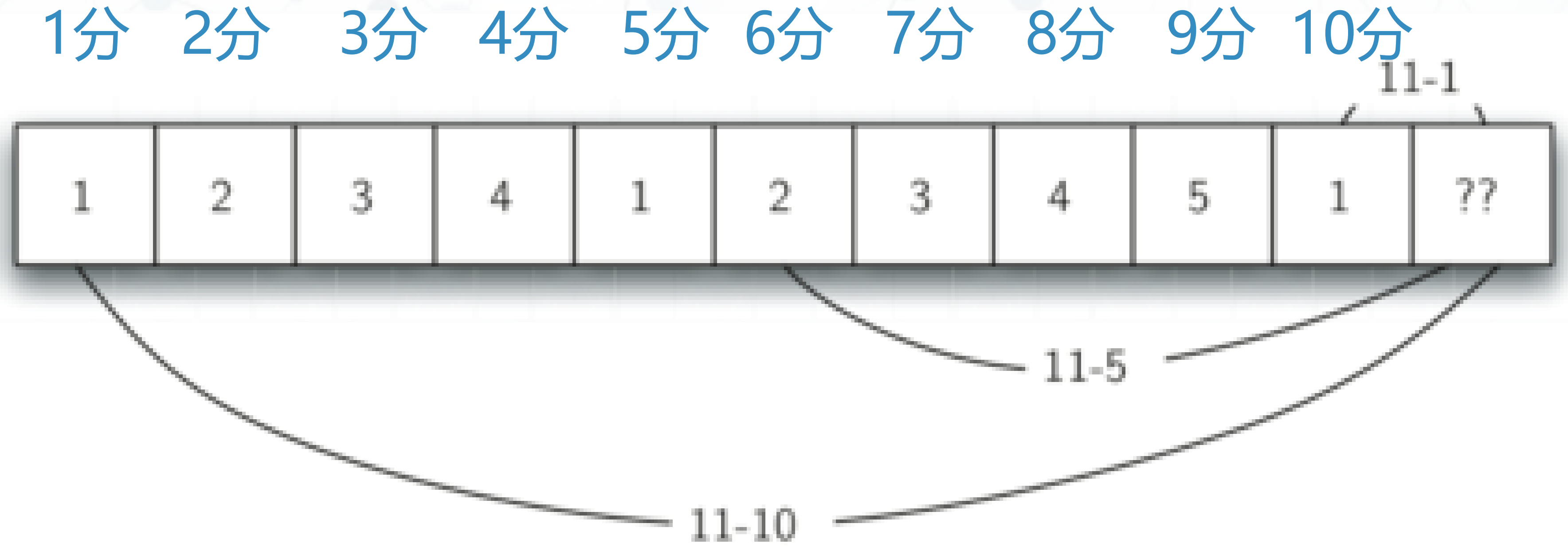
- 计算11分钱的兑换法，我们做如下几步：

首先11分钱减去1个1分钱的币值，剩下10分钱，查表，10分钱的最优解是1

然后11分钱减去1个5分钱的币值，剩下6分钱，查表，6分钱的最优解是2

最后11分钱减去1个10分钱的币值，剩下1分钱，查表，1分钱的最优解是1

- 这样，第1、3步我们都可以得到11分钱的最优解：2个硬币



兑换硬币：动态规划算法代码

从1分到所需找零

减除各种硬币后查表

```
def dpMakeChange(coinValueList,change,minCoins):
  for cents in range(change+1):
      coinCount = cents
      for j in [c for c in coinValueList if c <= cents]:
          if minCoins[cents-j] + 1 < coinCount:
              coinCount = minCoins[cents-j]+1
      minCoins[cents] = coinCount
  return minCoins[change]
```

得到部分找零的最优解

循环结束，得到最优解

兑换硬币：动态规划算法扩展

- 我们注意到动态规划算法的dpMakeChange并不是递归函数，虽然这个问题是从递归算法开始解决，但最终我们得到一个更有条理的高效非递归算法

动态规划中最主要的是从最简单情况开始到达所需找零的循环，其每一步都依靠以前的最优解来得到本步骤的最优解，直到得到答案。

- 前面的算法已经得到了最少硬币的数量，但没有返回硬币如何组合

扩展算法的思路很简单，只需要在生成最优解列表同时跟踪记录所选择的那个硬币币值即可

在得到最后的解之后，减去选择的硬币币值，回溯到表格之前的部分找零，就能逐步得到每一步所选择的硬币币值

兑换硬币：动态规划算法扩展代码

```
def dpMakeChange(coinValueList,change,minCoins,coinsUsed):
    for cents in range(change+1):
        coinCount = cents
        newCoin = 1
        for j in [c for c in coinValueList if c <= cents]:
            if minCoins[cents-j] + 1 < coinCount:
                coinCount = minCoins[cents-j]+1
                newCoin = j
        minCoins[cents] = coinCount
        coinsUsed[cents] = newCoin
    return minCoins[change]

def printCoins(coinsUsed,change):
    coin = change
    while coin > 0:
        thisCoin = coinsUsed[coin]
        print(thisCoin)
        coin = coin - thisCoin
```

兑换硬币：动态规划算法扩展代码

```python
def main():
    amnt = 63
    clist = [1,5,10,21,25]
    coinsUsed = [0]*(amnt+1)
    coinCount = [0]*(amnt+1)

    print("Making change for",amnt,"requires")
    print(dpMakeChange(clist,amnt,coinCount,coinsUsed),"coins")
    print("They are:")
    printCoins(coinsUsed,amnt)
    print("The used list is as follows:")
    print(coinsUsed)

main()
```

```
>>>
('Making change for', 63, 'requires')
(3, 'coins')
They are:
21
21
21
The used list is as follows:
[1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 1, 1, 1, 1, 5, 1, 1, 1, 1, 10, 21, 1, 1, 1, 2
5, 1, 1, 1, 1, 5, 10, 1, 1, 1, 10, 1, 1, 1, 1, 5, 10, 21, 1, 1, 10, 21, 1, 1, 1,
 25, 1, 10, 1, 1, 5, 10, 1, 1, 1, 10, 1, 10, 21]
>>>
```

讨论：博物馆大盗问题

- 大盗潜入博物馆，面前有5件宝物，分别有重量和价值，大盗的背包仅能负重20公斤，请问如何选择宝物，总价值最高？

item	weight	value
0	2	3
1	3	4
2	4	8
3	5	8
4	9	10

讨论：单词最小编辑距离问题

- 任意两个单词之间的变换，长度可以不同。

从源单词每复制一个字母到目标单词，计5分

从源单词每删除一个字母，计20分

在目标单词每插入一个字母，计20分

- 例如从单词“algorithm”变为“alligator”

源	al		g	ori		t	hm		
目标	al	lli	g		a	t		or	
操作	复制	插入	复制	删除	插入	复制	删除	插入	
分数	10	60	5	60	20	5	40	40	240

- 求多种操作方案中，分数最低的一种方案（编辑距离最小）

本章小结

- 在本章我们研究了几种递归算法，表明了递归是解决某些具有自相似性的复杂问题的有效技术
- 递归算法“三定律”

递归算法必须具备基本结束条件

递归算法必须要减小规模，改变状态，向基本结束条件演进

递归算法必须要调用自身

- 某些情况下，递归可以代替迭代循环
- 递归算法通常能够跟问题的表达很自然地契合
- 递归不总是最合适的算法，有时候递归算法会引发巨量的重复计算