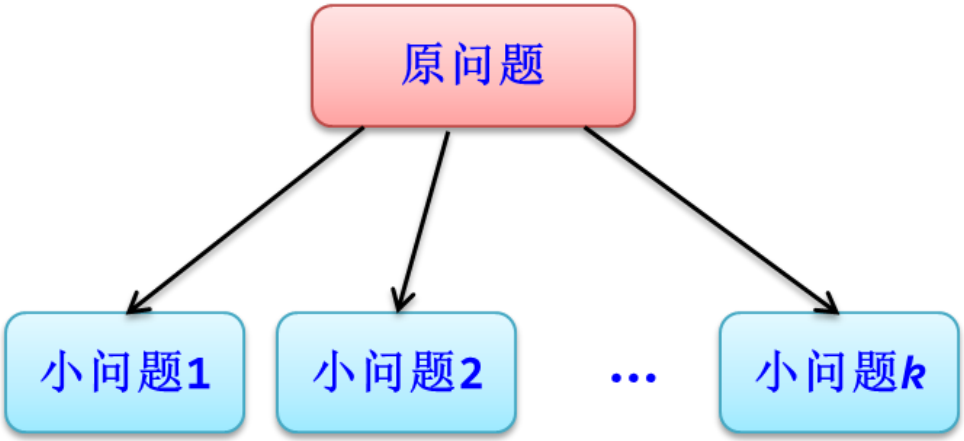
第5章 递归

提纲

CONTENTS

将一个难以直接解决的大问题，分割成一些规模较小的子问题，通过求解子问题来获得原问题的解。

- 纵向分割：分步
- 横向分割：子问题是原问题的较小模式。



5.1 什么是递归

5.1.1 递归的定义

- 在一个函数或过程中调用自己的方法，即称之为递归方法。
- 分治与递归就像一对孪生的兄弟，经常被应用在算法设计中，由此产生很多高效的算法。
- 如果一个递归过程或递归函数中递归调用语句是最后一条执行语句，称为尾递归。

【例5.1】以下是求n!（n为正整数）的递归函数。

```
def fun(n):
  if n==1:                    #语句1
    return 1                  #语句2
  else:                       #语句3
    return fun(n-1)*n         #语句4
```

由于递归调用是最后一条语句，故属于尾递归。

一般来说，能够用递归解决的问题应该满足以下3个条件：

- 需要解决的问题可以转化为一个或多个子问题来求解，而这些子问题的求解方法与原问题完全相同，只是在数量规模上不同。
- 递归调用的次数必须是有限的。
- 必须有结束递归的条件来终止递归。

5.1.2 何时使用递归

1．定义是递归的

有许多数学公式、数列等的定义是递归的。例如，求$n!$和Fibonacci数列等。

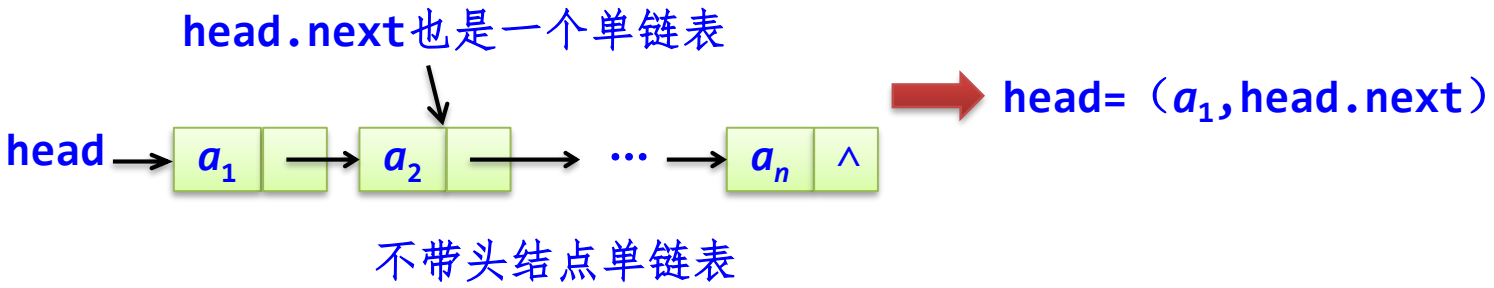
$$F(n)=\begin{cases}1 & n=1\\ 1 & n=2\\ F(n-1)+F(n-2) & n>2\end{cases}$$

```
def Fib(n):                          #求Fibonacci数列的第n项
  if n==1 or n==2:
    return 1
  else:
    return Fib(n-1)+Fib(n-2)
```

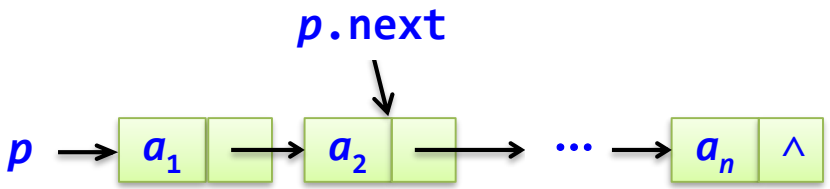
2．数据结构是递归的

有些数据结构是递归的。如单链表就是一种递归数据结构。

```
class LinkNode:                          #单链表结点类
  def __init__(self,data=None):          #构造函数
    self.data=data                       #dat属性
    self.next=None                       #next属性
```



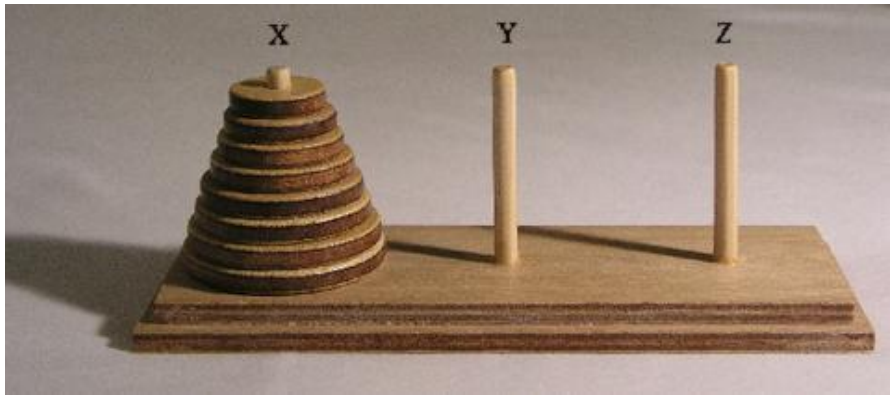
示例 求一个不带头结点单链表p中所有data成员（假设为int型）之和。




```
def Sum(p):                    #求不带头结点单链表p所有结点值之和
  if p==None:
    return 0
  else:
    return p.data+Sum(p.next)
```

3．问题的求解方法是递归的

Hanoi问题

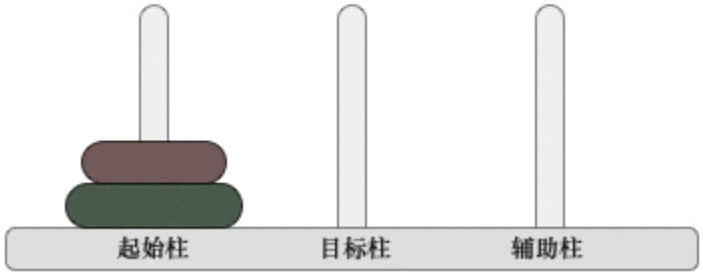


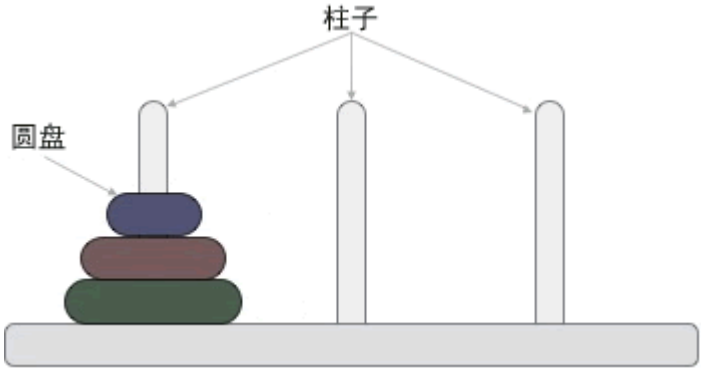
设Hanoi(*n*，*x*，*y*，*z*)表示将*n*个盘片从*x*塔座借助*y*塔座移动到*z*塔座上：

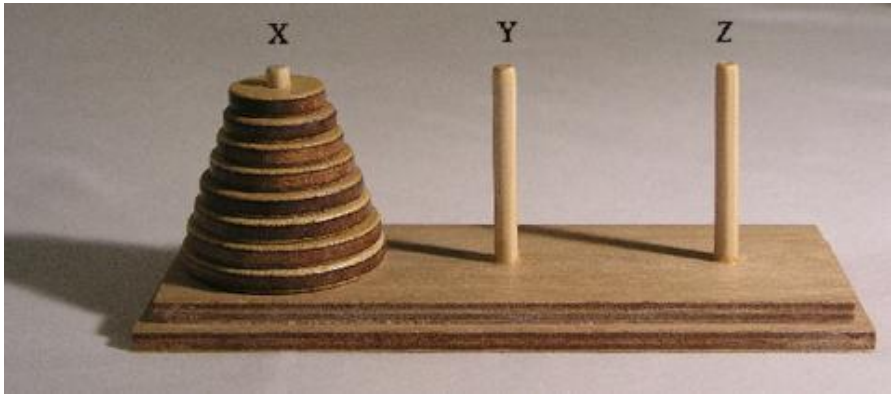
Hanoi(n，x，y，z) ➡

Hanoi(n-1，x，z，y);
move(n，x，z)：将第*n*个圆盘从*x*移到*z*;
Hanoi(n-1，y，x，z)

初始状态








```
def Hanoi(n,X,Y,Z):                          #Hanoi递归算法
  if n==1:                                   #只有一个盘片的情况
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
  else:                                      #有两个或多个盘片的情况
    Hanoi(n-1,X,Z,Y)
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
    Hanoi(n-1,Y,X,Z)
```

4．很多问题都可采用递归方法求解

实际上很多问题都可采用递归算法求解。

例．给定一个列表，求该列表中所有元素的和。

```
def  listsum(numList)
    Lsum=0
    for i in mumList
      Lsum=Lsum+i
    return Lsum
Print(listsum([1, 2, 3, 4]))
```

如果不用迭代求和，有没有别的思路来求上述和呢？

上述求和问题归纳为：

listsum(0）=第一个元素+余下子列表元素和

listsum=前一半元素的子列表元素之和+后一半子列表元素

listsum=前三分之一子列表元素和+中间三分之一子列表元素和+后三分之一子列表元素和

⋮

假设有n个硬币，其中有一个假币，已知假币比真币轻，问如何找出其中的假币？

◆ 实际上往往从小规模入手，逐步归纳出递归模型，进而采用递归方法求解问题。

5.1.3 递归模型

递归模型是递归算法的抽象，它反映一个递归问题的递归结构。

```
def fun(n):
  if n==1:                    #语句1
    return 1                  #语句2
  else:                       #语句3
    return fun(n-1)*n         #语句4
```

递归模型

$$f(n)=1 \qquad n=1$$
$$f(n)=n*f(n-1) \qquad n>1$$

$f(n)=1$ $n=1$ ← 出口条件

$f(n)=n*f(n-1)$ $n>1$ ← 递归体

- 一般地，一个递归模型是由递归出口和递归体两部分组成。
- 出口条件：确定递归到何时结束，即指出明确的递归结束条件。
- 递归体：确定递归求解时的递推关系。

又如，斐波那契数列

$$F(n)=\begin{cases}1 & n=1 \\ 1 & n=2 \\ F(n-1)+F(n-2) & n>2\end{cases}$$

出口条件

递归体

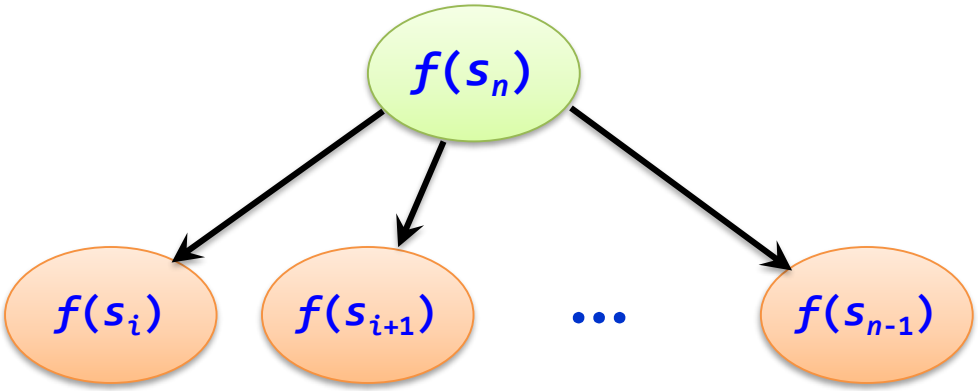
递归出口的一般格式如下：

$$f(s_1)=m_1$$

递归体的一般格式如下：

$$f(s_n)=g(f(s_i),\ f(s_{i+1}),\ \cdots,\ f(s_{n-1}),\ c_j,\ c_{j+1},\ \cdots,\ c_m)$$

非递归函数



大问题求解

转化

若干个相似子问题求解

5.1.4 递归的执行过程

简化的递归模型

$f(s_1)=m_1$

$f(s_n)=g(f(s_{n-1}), c_{n-1})$

求大问题$f(s_n)$：

分解（递推）和求值

求$f(s_n)$的分解过程如下：

$f(s_n)$

↓

$f(s_{n-1})$

↓

…

↓

$f(s_2)$

↓

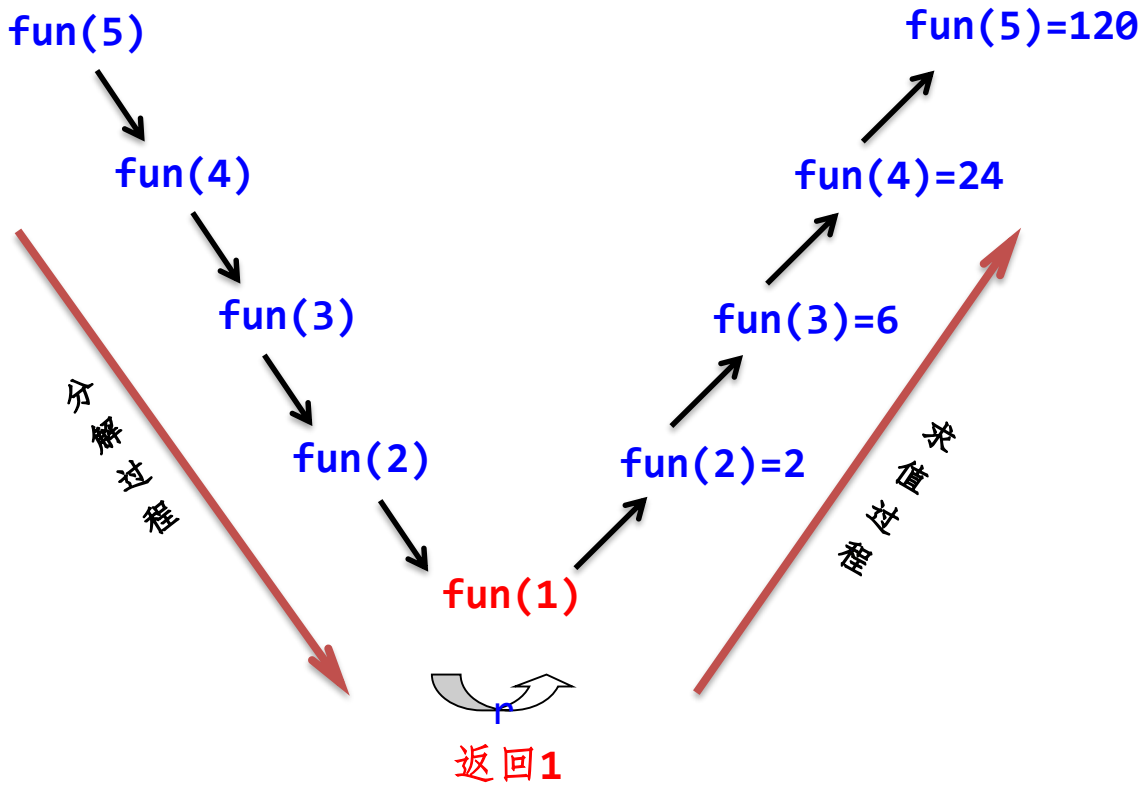
$f(s_1)$

遇到递归出口发生“质变”，原递归问题便转化成可以直接求解的问题。

求值过程：

$$f(s_1)=m_1$$
$$\downarrow$$
$$f(s_2)=g(f(s_1),c_1)$$
$$\downarrow$$
$$f(s_3)=g(f(s_2),c_2)$$
$$\downarrow$$
$$\cdots$$
$$\downarrow$$
$$f(s_n)=g(f(s_{n-1}),c_{n-1})$$

例如求5!。


fun(5)
fun(4)
fun(3)
fun(2)
fun(1)
返回1
fun(2)=2
fun(3)=6
fun(4)=24
fun(5)=120
分解过程
求值过程

系统内部如何执行递归算法

- 一个递归函数的调用过程类似于多个函数的嵌套的调用，只不过调用函数和被调用函数是同一个函数。
- 为了保证递归函数的正确执行，系统需设立一个工作栈。

1）执行开始时，首先为递归调用建立一个工作栈，其结构包括值参、局部变量和返回地址。

2）每次执行递归调用之前，把递归函数的值参和局部变量的当前值以及调用后的返回地址进栈。

3）每次递归调用结束后，将栈顶元素出栈，使相应的值参和局部变量恢复为调用前的值，然后转向返回地址指定的位置继续执行。

例如，有以下程序段：

```
def S(n):
  if n<=0: return 0
  else: return S(n-1)+n

def main():
  print(S(1))
```

程序执行时使用一个栈来保存调用过程的信息，这些信息用main()、S(0)和S(1)表示，那么自栈底到栈顶保存的信息的顺序是怎么样呢？

```
def S(n):
  if n<=0: return 0
  else: return S(n-1)+n

def main():
  print(S(1))
```

执行过程：

① 调用main()
② 调用S(1)
③ 调用S(0)
④ 从S(0)返回
⑤ 从S(1)返回
⑥ 从main()返回

- 用递归算法的形参值表示**状态**，由于递归算法执行中系统栈保存了递归调用的值参、局部变量和返回地址。
- 所以在递归算法中一次递归调用后会自动恢复该次递归调用前的状态。

```
def f(n):                                  #递归函数
  if (n==0):                               #递归出口
    return
  else:                                    #递归体
    print("Pre:  n=%d" %(n))
    print("执行f(%d)" %(n-1))
    f(n-1)
    print("Post: n=%d" %(n))
```

```
def f(n):                                   #递归函数
  if (n==0):                                #递归出口
    return
  else:                                     #递归体
    print("Pre:  n=%d" %(n))
    print("执行f(%d)" %(n-1))
    f(n-1)
    print("Post: n=%d" %(n))
```

执行f(4)的结果

```
Pre: n=4
执行f(3)          //递归调用f(3)
Pre: n=3
执行f(2)          //递归调用f(2)
Pre: n=2
执行f(1)          //递归调用f(1)
Pre: n=1
执行f(0)          //递归调用f(0)
Post:n=1          //恢复f(0)调用前的n值
Post:n=2          //恢复f(1)调用前的n值
Post:n=3          //恢复f(2)调用前的n值
Post:n=4          //恢复f(3)调用前的n值
```

5.1.5 Python中递归函数的参数

- Python递归函数中的参数分为可变类型和不可变类型。
- 不可变类型的参数保存在系统栈中，具有自动回退。
- 可变类型的参数类似全局变量，不具有自动回退的功能。

id()返回对象的唯一身份标识（相当于对象地址）

```
def fun(i,lst):
  print(id(i),id(lst))
  if i>=0:
     print(lst[i],end=' ')
     fun(i-1,lst)

#主程序
L=[1,2,3]
fun(len(L)-1,L)
```

```
1518494912  1721848
3 1518494896  1721848
2 1518494880  1721848
1 1518494864  1721848
```

可以直接用全局变量lst替代形参lst

```
lst=[1,2,3]
def fun(i):
  print(id(i),id(lst))
  if i>=0:
     print(lst[i],end=' ')
     fun(i-1)

#主程序
fun(len(lst)-1)
```

```
1518494912  1721848
3 1518494896  1721848
2 1518494880  1721848
1 1518494864  1721848
```

5.1.6 递归算法的时空分析

- 从前面的递归算法实例可以看出，递归算法的时空分析也不同于非递归算法，其时空复杂度一般可表示为递推方程。

1．递归算法的时间分析

```
def Hanoi(n,X,Y,Z):                    #Hanoi递归算法
  if n==1:                             #只有一个盘片的情况
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
  else:                                #有两个或多个盘片的情况
    Hanoi(n-1,X,Z,Y)
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
    Hanoi(n-1,Y,X,Z)
```

执行Hanoi(*n*,*x*,*y*,*z*)的时间复杂度为O(1)吗？

```
def Hanoi(n,X,Y,Z):                    #Hanoi递归算法
  if n==1:                             #只有一个盘片的情况
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
  else:                                #有两个或多个盘片的情况
    Hanoi(n-1,X,Z,Y)
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
    Hanoi(n-1,Y,X,Z)
```

设大问题Hanoi(*n*，*x*，*y*，*z*)的执行时间为T(*n*)，则小问题Hanoi(*n*-1，*x*，*y*，*z*)的执行时间为T(*n*-1)。递推式：

T(*n*)=1	当*n*=1时
T(*n*)=2T(*n*-1)+1	当*n*>1时

$T(n)=1$ 当$n=1$时

$T(n)=2T(n-1)+1$ 当$n>1$时

$$T(n)=2T(n-1)+1=2(2T(n-2)+1)+1$$

$$=2^2T(n-2)+2+1=2^2(2T(n-3)+1)+2+1$$

$$=2^3T(n-3)+2^2+2+1$$

$$=\cdots$$

$$=2^{n-1}T(1)+2^{n-2}+\cdots+2^2+2+1$$

$$=2^n-1=O(2^n)$$

2．递归算法的空间分析

```
def Hanoi(n,X,Y,Z):                    #Hanoi递归算法
  if n==1:                             #只有一个盘片的情况
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
  else:                                #有两个或多个盘片的情况
    Hanoi(n-1,X,Z,Y)
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
    Hanoi(n-1,Y,X,Z)
```

执行Hanoi(*n*,*x*,*y*,*z*)的空间复杂度为O(1)吗？

```
def Hanoi(n,X,Y,Z):                    #Hanoi递归算法
  if n==1:                             #只有一个盘片的情况
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
  else:                                #有两个或多个盘片的情况
    Hanoi(n-1,X,Z,Y)
    print("将第%d个盘片从%c移动到%c" %(n,X,Z))
    Hanoi(n-1,Y,X,Z)
```

设大问题Hanoi(*n*，*x*，*y*，*z*)的占用空间为S(*n*)，则小问题Hanoi(*n*-1，*x*，*y*，*z*)的占用空间为S(*n*-1)。递推式：

S(*n*)=1	当*n*=1时
S(*n*)=S(*n*-1)+1	当*n*>1时

$S(n)=1$	当$n=1$时
$S(n)=S(n-1)+1$	当$n>1$时

$$
\begin{aligned}
S(n) &= S(n-1)+1 \\
&= S(n-2)+1+1=S(n-2)+2 \\
&= \cdots \\
&= S(1)+(n-1)=1+(n-1) \\
&= n=O(n)
\end{aligned}
$$

- 将一个规模为n的问题分成k个规模为n/m的子问题去解。设分解阈值n_0=1，且adhoc解规模为1的问题耗费1个单位时间。再设将原问题分解为k个子问题以及用merge将k个子问题的解合并为原问题的解需f(n)个单位时间。用T(n)表示解规模为n的问题所需计算时间，则有：

$$T(n)=\begin{cases}O(1) & n=1\\ kT\left(\frac{n}{m}\right)+f(n) & n>1\end{cases}$$

这个问题如何求解呢？大家课下思考，可以课程群里讨论。

5.2 递归算法的设计

5.2.1 递归算法设计的步骤

- 设计求解问题的递归模型。
- 转换成对应的递归算法。

求递归模型的步骤如下：

（1）对原问题$f(s)$进行分析，称为“大问题”，假设出合理的“小问题”$f(s')$；

数学归纳法

（2）假设$f(s')$是可解的，在此基础上确定$f(s)$的解，即给出$f(s)$与$f(s')$之间的关系 ⇨ 递归体。

⟷ 假设$n=k-1$时等式成立；求证$n=k$时等式成立

（3）确定一个特定情况（如$f(1)$或$f(0)$）的解 ⇨ 递归出口。

⟷ 求证$n=1$时等式成立

【例5.2】采用递归算法求整数数组$a[0..n-1]$中的最小值。

解：假设$f(a, i)$求数组元素$a[0..i]$（共$i+1$个元素）中的最小值。

- 当$i=0$时，有$f(a, i)=a[0]$。
- 假设$f(a, i-1)$已求出，显然有$f(a, i)=MIN(f(a, i-1), a[i])$，其中$MIN()$为求两个值较小值函数。

得到递归模型

$$f(a, i)=a[0] \quad \text{当} i=0\text{时}$$
$$f(a, i)=MIN(f(a, i-1), a[i]) \quad \text{其他情况}$$

$f(a, i)=a[0]$　　当$i=0$时

$f(a, i)=MIN(f(a, i-1), a[i])$　　其他情况

```
def Min(a,i):                          #求a[0..i]中的最小值
  if i==0:                             #递归出口
    return a[0]
  else:                                #递归体
    min=Min(a,i-1)
    if (min>a[i]): return a[i]
    else: return min
```

5.2.2 基于递归数据结构的递归算法设计

递归数据结构的数据特别适合递归处理 ⇨递归算法

种瓜得瓜：递归性

数据：D={瓜的集合}

运算：Op={种瓜}

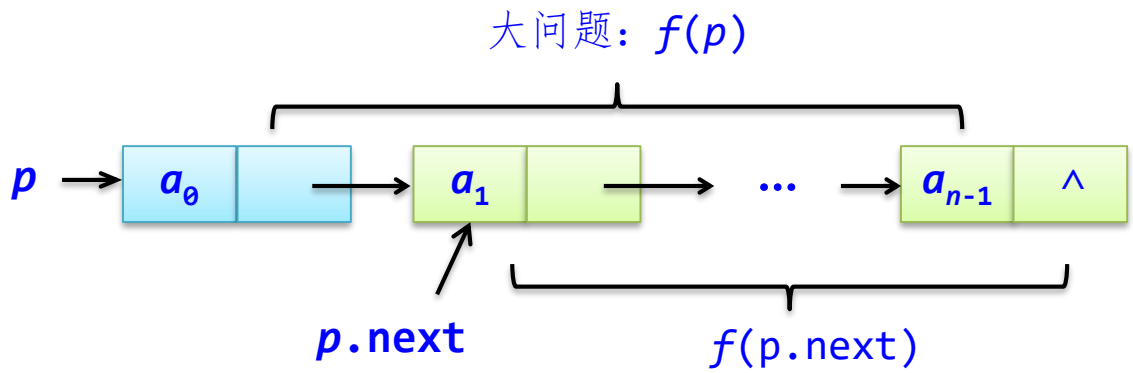
递归性：

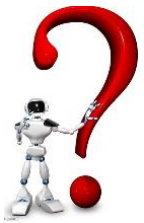
$Op(x \in D) \in D$

【例5.3】假设有一个不带头结点的单链表p，完成以下两个算法设计：

（1）设计一个算法正向输出所有结点值。

（2）设计一个算法反向输出所有结点值。

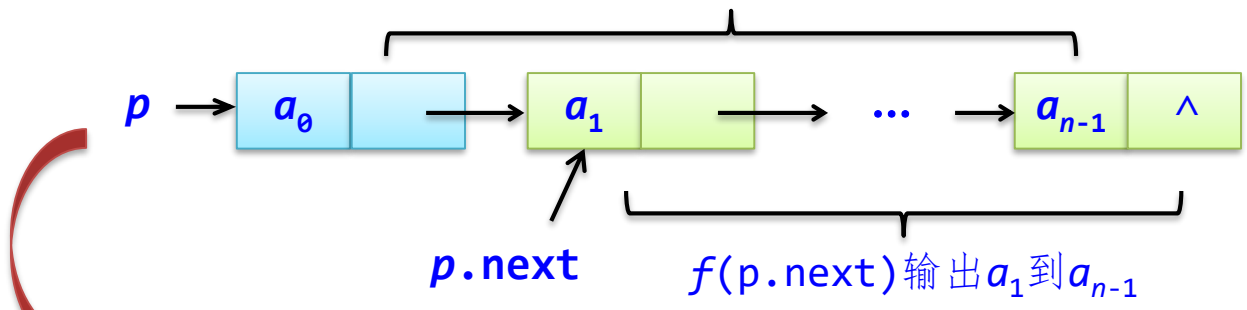




为什么在这里设计单链表的递归算法时不带头结点?

如何将带头结点转换为不带头结点的单链表?

(1)正向输出



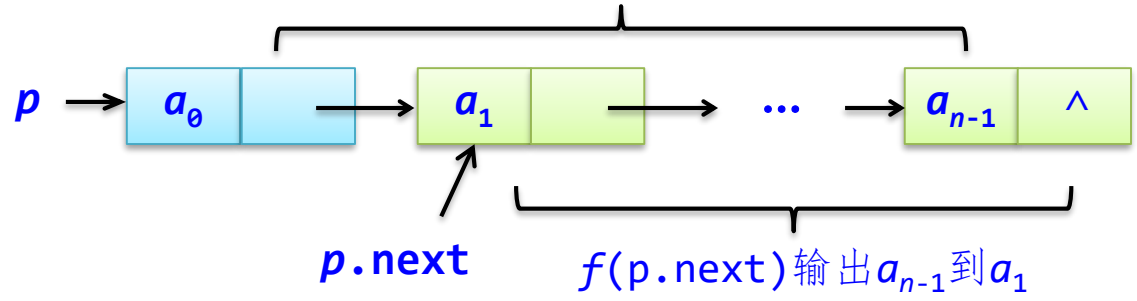
$f(p)$ ≡ 不做任何事件　　当p=None时

$f(p)$ ≡ 输出p结点值; f(p.next)　　其他情况

```
def Positive(p):                    #正向输出所有结点值
  if p==None:
    return
  else:
    print("%d" %(p.data),end=' ')
    Positive(p.next)
```

(2)反向输出

大问题：$f(p)$输出a_{n-1}到a_0



$f(p)$ ≡ 不做任何事件　　当p=None时

$f(p)$ ≡ f(p.next);输出p结点值　　其他情况

```
def Reverse(p):                    #反向输出所有结点值
  if p==None:
    return
  else:
    Reverse(p.next)
    print("%d" %(p.data),end=' ')
```

比较

```
def Positive(p):                    #正向输出所有结点值
  if p==None:
    return
  else:
    print("%d" %(p.data),end=' ')
    Positive(p.next)
```

```
def Reverse(p):                     #反向输出所有结点值
  if p==None:
    return
  else:
    Reverse(p.next)
    print("%d" %(p.data),end=' ')
```

5.2.3 基于归纳方法的递归算法设计

- 通过对求解问题的分析归纳来转换成递归方法求解（如皇后问题等）。
- 关键是对问题本身进行分析，确定大、小问题解之间的关系，构造合理的递归体，而其中最重要的又是假设出“合理”的小问题。

【例5.4】若算法pow(*x*，*n*)用于计算x^n（*n*为大于1的整数）。完成以下任务：

（1）采用递归方法设计pow(*x*，*n*)算法。

（2）问执行pow(*x*，10)发生几次递归调用？求pow(*x*,*n*)对应的算法复杂度是多少？

解： (1) 设$f(x, n)$用于计算x^n，则有以下递归模型：

$$
\begin{aligned}
&f(x, n)=1 && \text{当}n=0\\
&f(x, n)=x*f(x, n/2)*f(x, n/2) && \text{当}n\text{为奇数}\\
&f(x, n)=f(x, n/2)*f(x, n/2) && \text{当}n\text{为偶数}
\end{aligned}
$$

```
def pow(x,n):                          #求x的n次幂
  if n==1:
    return x;
  p=pow(x,n//2)
  if n%2==1:
    return x*p*p                       #n为奇数
  else:
    return p*p                         #n为偶数
```

（2）执行pow(*x*，10)的递归调用顺序是：

pow(*x*，10) → pow(*x*，5) → pow(*x*，2) → pow(*x*，1)

共发生4次递归调用。

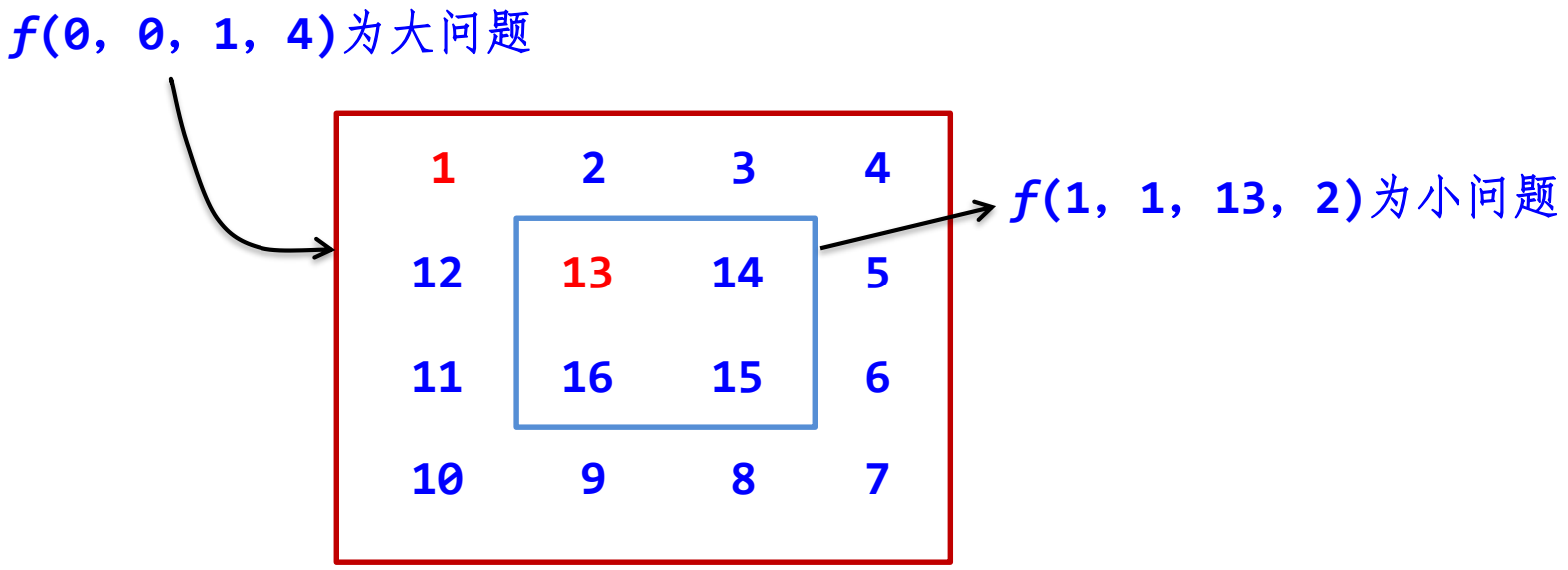
求pow(*x*，*n*)对应的算法复杂度是$O(\log_2 n)$。

【例5.5】创建一个*n*阶螺旋矩阵并输出。例如，*n*=4时的螺旋矩阵如下：

```
1     2     3     4
12    13    14    5
11    16    15    6
10    9     8     7
```

参考答案

- 设*f*(*x*，*y*，start，*n*)用于创建左上角为（*x*，*y*）、起始元素值为start的*n*阶螺旋矩阵，共*n*行*n*列，它是大问题。
- *f*(*x*+1，*y*+1，start，*n*-2)用于创建左上角为（*x*+1，*y*+1）、起始元素值为start的*n*-2阶螺旋矩阵，共*n*-2行*n*-2列，它是小问题。



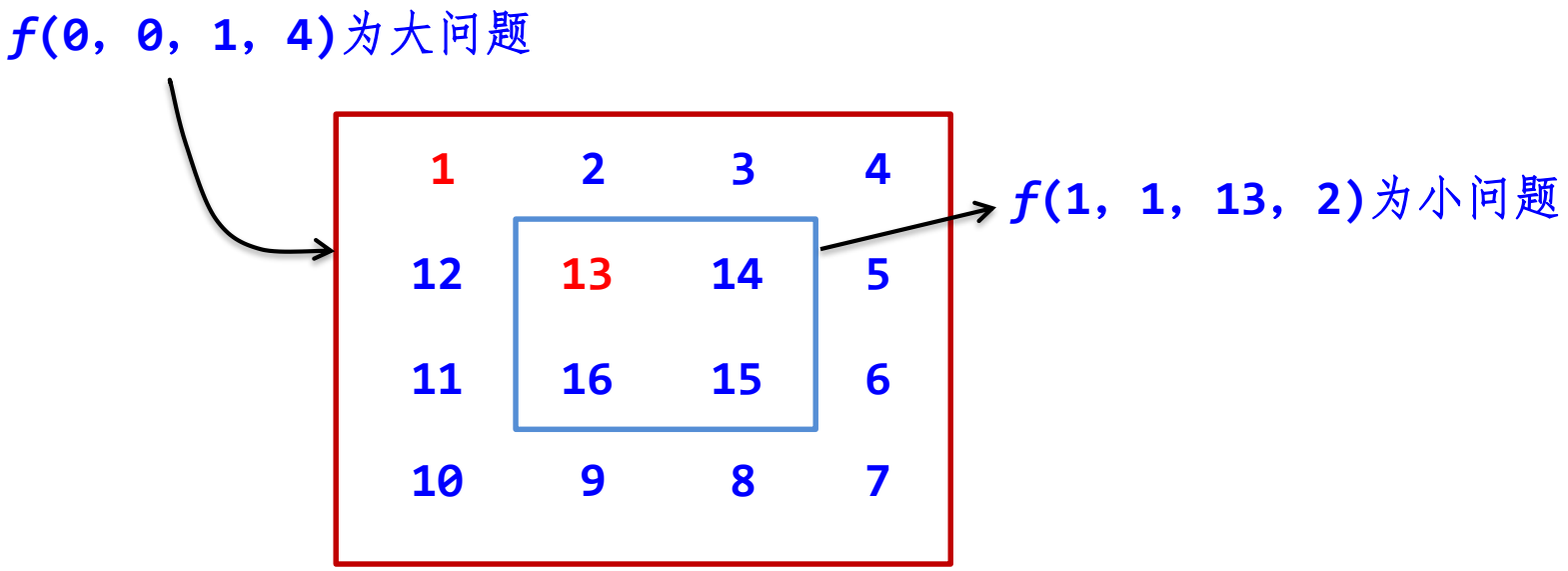
对应的递归模型如下：

$f(x, y, start, n) \equiv$ 不做任何事情　　当$n \le 0$

$f(x, y, start, n) \equiv$ 产生只有一个元素的螺旋矩阵　　当$n = 1$

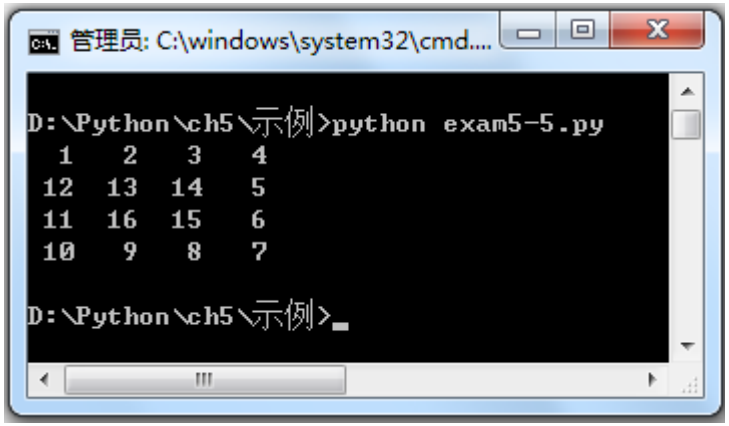
$f(x, y, start, n) \equiv$ 产生（x，y）的那一圈；　　当$n > 1$

$f(x+1, y+1, start, n-2)$



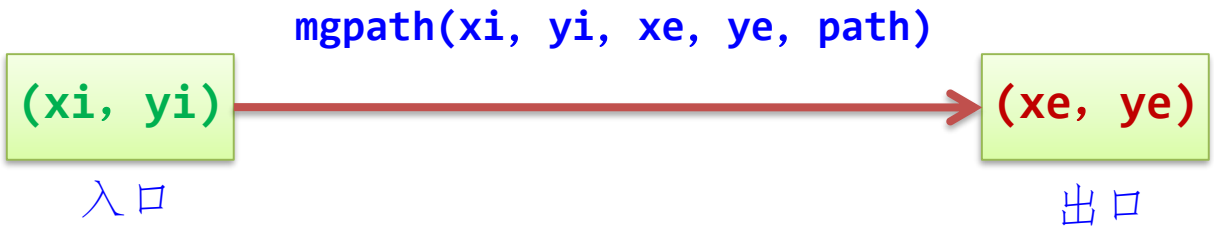
```
def Spiral(x,y,start,n):                    #递归创建螺旋矩阵
  if n<=0: return                           #递归结束条件
  if n==1:                                  #矩阵大小为1时
    a[x][y]=start
    return

  for j in range(x,x+n-1):                  #上一行
    a[y][j]=start
    start+=1
  for i in range(y,y+n-1):                  #右一列
    a[i][x+n-1]=start
    start+=1
  for j in range(x+n-1,x,-1):               #下一行
    a[y+n-1][j]=start
    start+=1
  for i in range(y+n-1,y,-1):               #左一列
    a[i][x]=start
    start+=1

  Spiral(x+1,y+1,start,n-2)                 #递归调用
```

```
#主程序
n=4
Spiral(0,0,1,n)
for i in range(0,n):
  for j in range(0,n):
    print("%3d" %(a[i][j]),end=' ')
  print()
```

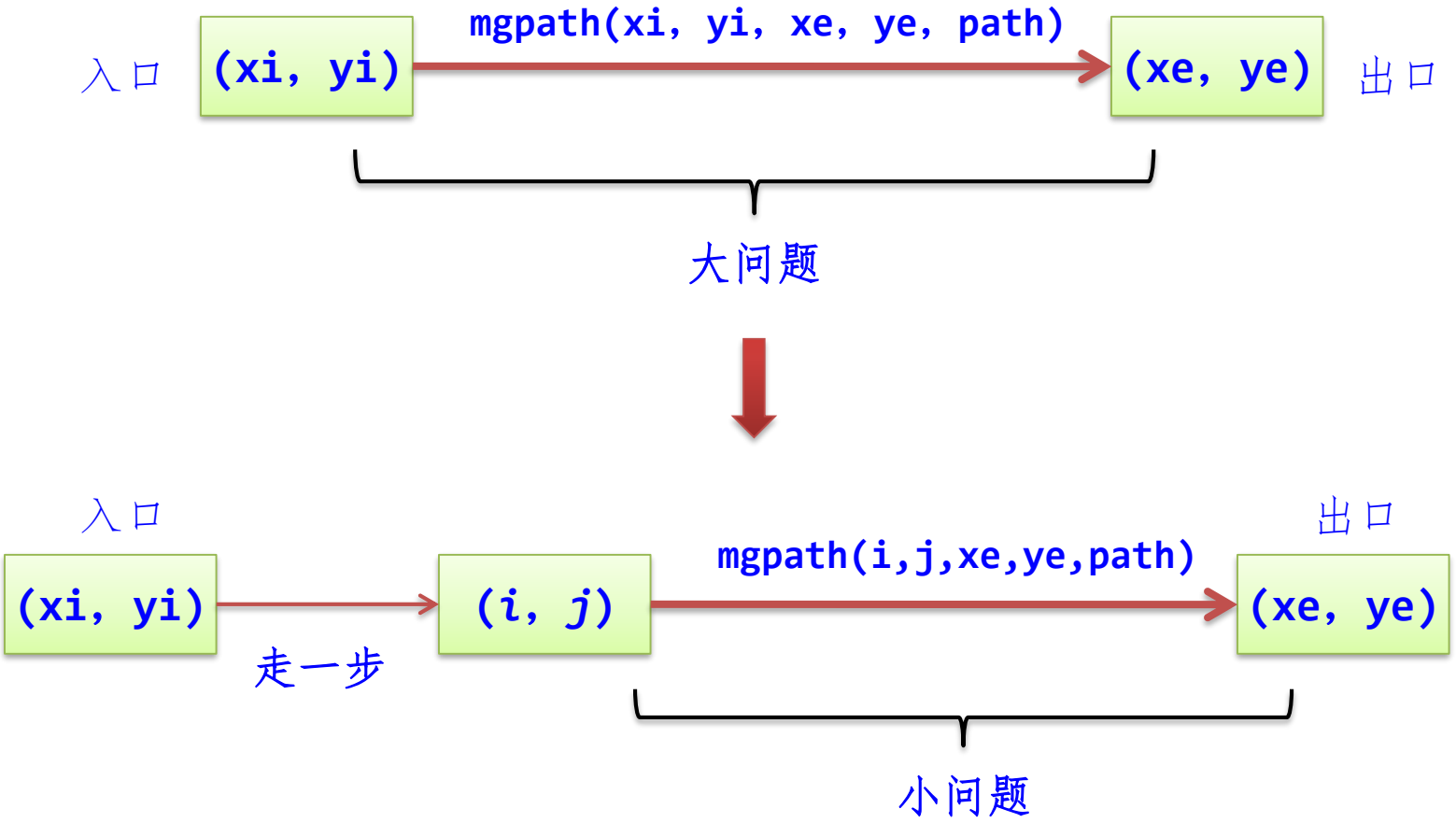


【例5.6】采用递归算法求解迷宫问题，并输出从入口到出口的所有迷宫路径。

求解问题描述：



mgpath(int xi, int yi, int xe, int ye, Box path)：求从(xi, yi)到(xe, ye)的迷宫路径，用path变量保存迷宫路径。


入口
(xi, yi)
mgpath(xi, yi, xe, ye, path)
(xe, ye)
出口
大问题
入口
出口
(xi, yi)
走一步
(i, j)
mgpath(i,j,xe,ye,path)
(xe, ye)
小问题

大问题 ≡ 走一步 + 小问题

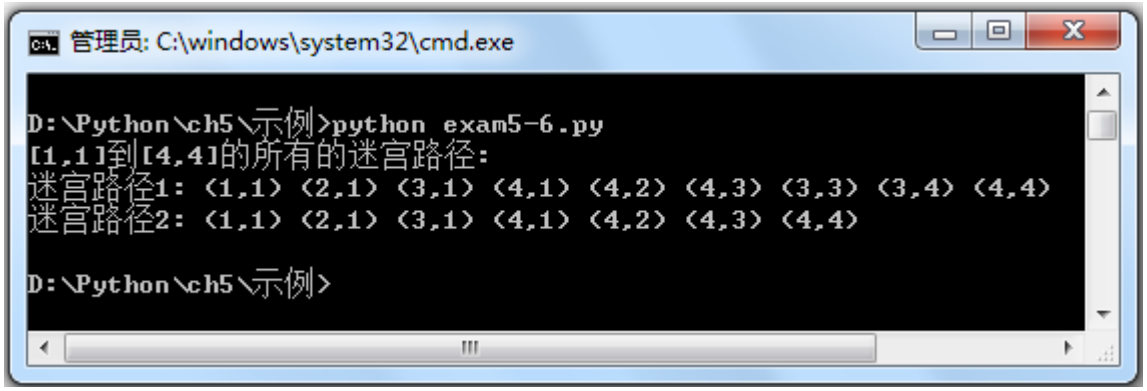
求解迷宫问题的递归模型如下：

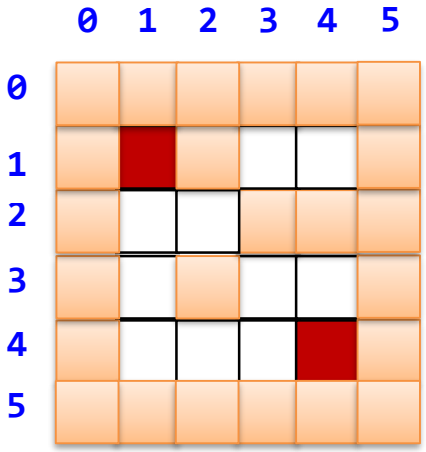
```
mgpath(xi,yi,xe,ye,path,d) ≡ d++;将(xi,yi)添加到path中;
                             置mg[xi][yi]=-1;
                             输出path中的迷宫路径;
                             恢复出口迷宫值为0即置mg[xe][ye]=0
                                          若(xi,yi)=(xe,ye)即找到出口
mgpath(xi,yi,xe,ye,path,d) ≡ d++; 将(xi,yi)添加到path中;
                             置mg[xi][yi]=-1;
                             对于(xi,yi)每个相邻可走方块(i,j),
                                 调用mgpath(i,j,xe,ye,path,d);
                             从(xi,yi)回退一步即置mg[xi][yi]=0;
                                          若(xi,yi)不是出口
```

```
mg=[[1,1,1,1,1,1],[1,0,1,0,0,1],[1,0,0,1,1,1],
    [1,0,1,0,0,1],[1,0,0,0,0,1],[1,1,1,1,1,1]]
dx=[-1,0,1,0]                                    #x方向的偏移量
dy=[0,1,0,-1]                                    #y方向的偏移量
cnt=0                                            #累计迷宫路径数
```

```
def mgpath(xi,yi,xe,ye,path,d):     #求解迷宫路径为:(xi,yi)->(xe,ye)
  global cnt
  d+=1; path[d]=[xi,yi]             #将(xi,yi)方块对象添加的路径中
  mg[xi][yi]=-1                     #mg[xi][yi]=-1

  if xi==xe and yi==ye:             #找到了出口,输出一个迷宫路径
    cnt+=1
    print("迷宫路径%d: " %(cnt),end='')
    for k in range(d+1):            #输出第cnt条迷宫路径
      print("(%d,%d)" %(path[k][0],path[k][1]),end=' ')
    print()
    mg[xi][yi]=0                    #从出口回退，恢复其mg值
    return

  else:                             #(xi,yi)不是出口
    di=0
    while di<4:                     #处理(xi,yi)四周的每个相邻方块(i,j)
      i,j=xi+dx[di],yi+dy[di]       #找(xi,yi)的di方位的相邻方块(i,j)
      if mg[i][j]==0:               #若(i,j)可走时
         mgpath(i,j,xe,ye,path,d)   #从(i,j)出发查找迷宫路径
      di+=1                         #继续处理(xi,yi)的下一个相邻方块
    mg[xi][yi]=0                    #(xi,yi)的所有相邻方块处理完，从其回退
```

```
#主程序
xi,yi=1,1
xe,ye=4,4
print("[%d,%d]到[%d,%d]的所有的迷宫路径:" %(xi,yi,xe,ye))
path=[None]*100
d=-1
mgpath(xi,yi,xe,ye,path,d)
```





迷宫问题的递归求解与用栈和队列求解有什么异同？


未来的你
会 感 谢
今 ❤ 天
努力的你
GO AND BE FREE