第5章 递 归

在算法设计中经常需要用递归方法求解，特别是后面的树和二叉树、图、查找及排序等章节中会大量地遇到递归算法。递归是计算机科学中的一个重要工具，很多程序设计语言（如 Python）都支持递归程序设计。本章介绍递归的定义和递归算法设计方法等，为后面的学习打下基础。

本章主要学习要点如下。

（1）递归的定义和递归模型。

（2）递归算法设计的一般方法。

（3）灵活运用递归算法解决一些较复杂的应用问题。

5.1 什么是递归

5.1.1 递归的定义

扫一扫 视频讲解

在定义一个过程或函数时出现调用本过程或本函数的成分，称为递归。若调用自身，称为直接递归。若过程或函数 A 调用过程或函数 B，而 B 又调用 A，称为间接递归。在算法设计中，任何间接递归算法都可以转换为直接递归算法来实现，所以后面主要讨论直接递归。

递归不仅是数学中的一个重要概念，也是计算技术中重要的概念之一。在计算技术中，与递归有关的概念有递归数列、递归过程、递归算法、递归程序和递归方法等。

【例 5.1】 以下是求 $n!$（n 为正整数）的递归函数，它属于什么类型的递归？

```
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
```

解 在函数 fun(n)的求解过程中调用 fun($n-1$)（语句 4），它是一个直接递归函数。

递归算法通常把一个大的复杂问题层层转化为一个或多个与原问题相似的规模较小的问题来求解，递归策略只需少量的代码就可以描述出解题过程所需要的多次重复计算，大大减少了算法的代码量。

一般来说，能够用递归解决的问题应该满足以下 3 个条件。

① 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。

② 递归调用的次数必须是有限的。

③ 必须有结束递归的条件来终止递归。

递归算法的优点是结构简单、清晰，易于阅读，方便其正确性证明；缺点是算法执行中占用的内存空间较多，执行效率低，不容易优化。

5.1.2 何时使用递归

扫一扫 视频讲解

在以下 3 种情况下经常要用到递归方法。

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（斐波那契）数列等。这些问题的求解可以将其递归定义直接转化为对应的递归算法，例如求 $n!$ 可以转化为例 5.1 的递归算法。求 Fibonacci 数列的递归算法如下：

```
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。例如，第 2 章中介绍过的单链表就是一种递归数据结构，其结点类定义如下：

```
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
```

其中，next 属性是一种指向自身类型结点的指针，所以它是一种递归数据结构。

对于递归数据结构，采用递归的方法编写算法既方便又有效。例如，求一个不带头结点单链表 p 中所有 data 成员(假设为 int 型)之和的递归算法如下：

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

说明：对于第 2 章讨论的单链表对象 L，L.head 为头结点，将 L.head.next 看成不带头结点的单链表。

3. 问题的求解方法是递归的

有些问题的解法是递归的，典型的有 Hanoi 问题。该问题的描述是设有 3 个分别命名为 X、Y 和 Z 的塔座，在 X 塔座上套有 n 个直径各不相同的盘片，从小到大依次编号为 1，2，…，n。现要求将 X 塔座上的这 n 个盘片移到 Z 塔座上并仍按同样的顺序叠放，盘片移动时必须遵守以下规则：每次只能移动一个盘片；盘片只能套在 X、Y 和 Z 中的任一塔座；任何时候都不能将一个较大的盘片放在较小的盘片上。图 5.1 所示为 $n=4$ 时的 Hanoi 问题。设计求解该问题的算法。

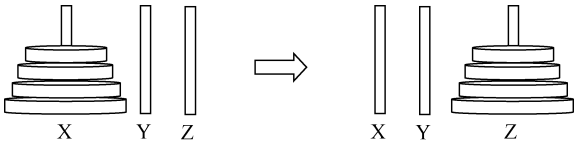
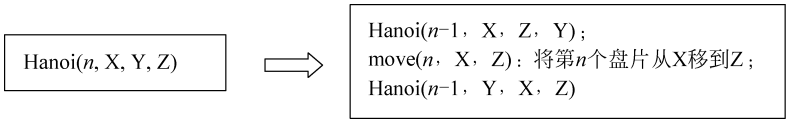


图 5.1 Hanoi 问题($n=4$)

Hanoi 问题特别适合采用递归方法来求解。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，递归分解的过程如下：



其含义是首先将 X 塔座上的 $n-1$ 个盘片借助 Z 塔座移动到 Y 塔座上；此时 X 塔座上只有一个编号为 n 的盘片，将其直接移动到 Z 塔座上；再将 Y 塔座上的 $n-1$ 个盘片借助 X

塔座移动到 Z 塔座上。由此得到 Hanoi 递归算法如下：

```
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)
```

调用 Hanoi(3,'X','Y','Z')的输出结果如下：

```
将第 1 个盘片从 X 移动到 Z
将第 2 个盘片从 X 移动到 Y
将第 1 个盘片从 Z 移动到 Y
将第 3 个盘片从 X 移动到 Z
将第 1 个盘片从 Y 移动到 X
将第 2 个盘片从 Y 移动到 Z
将第 1 个盘片从 X 移动到 Z
```

扫一扫

视频讲解

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。例如，例 5.1 的递归算法对应的递归模型如下：

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

其中，第一个式子给出了递归的终止条件，第二个式子给出了 $f(n)$的值与 $f(n-1)$的值之间的关系，把第一个式子称为递归出口，把第二个式子称为递归体。

一般地，一个递归模型由递归出口和递归体两部分组成。**递归出口**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。

递归出口的一般格式如下：

$$f(s_1)=m_1$$

这里的 s_1 与 m_1 均为常量，有些递归问题可能有几个递归出口。递归体的一般格式如下：

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

其中，n、i、j、m 均为正整数。这里的 s_n 是一个递归“大问题”，s_i、s_{i+1}、……、s_{n-1} 为递归“小问题”，c_j、c_{j+1}、……、c_m 是若干可以直接（用非递归方法）解决的问题，g 是一个非递归函数，可以直接求值。

实际上，递归思路是把一个不能或不好直接求解的“大问题”转化成一个或几个“小问题”来解决，如图 5.2 所示，再把这些“小问题”进一步分解成更小的“小问题”来解决，如此分

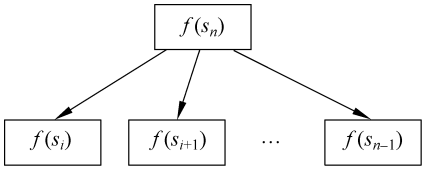


图 5.2 把大问题 $f(s_n)$转化成几个小问题来解决

解，直到每个“小问题”都可以直接解决（此时分解到递归出口）。但递归分解不是随意分解，递归分解要保证“大问题”与“小问题”相似，即求解过程与环境都相似。

5.1.4 递归与数学归纳法

数学归纳法是一种数学证明方法，通常被用于证明某个给定命题在整个（或者局部）自然数范围内成立。先看一个简单示例，采用数学归纳法证明下式：

$$1+2+\cdots+n=n(n+1)/2$$

其证明过程如下：

① 当 $n=1$ 时，左式=1，右式=(1×2)/2=1，左、右两式相等，等式成立。

② 假设当 $n=k-1$ 时等式成立，有 $1+2+\cdots+(k-1)=k(k-1)/2$。

③ 当 $n=k$ 时，左式=$1+2+\cdots+k=1+2+\cdots+(k-1)+k=k(k-1)/2+k=k(k+1)/2=$右式。即证。

数学归纳法证明问题的过程分为两个步骤，先考虑特殊情况，然后假设 $n=k-1$ 成立（第二数学归纳法是假设 $n\leqslant k-1$ 均成立），再证明 $n=k$ 时成立，即假设“小问题”成立，再推导出“大问题”成立。

而递归模型中的递归体就是表示“大问题”和“小问题”解之间的关系，如果已知 s_i，s_{i+1}，…，s_{n-1} 这些“小问题”的解，就可以计算出 s_n“大问题”的解。从数学归纳法的角度来看，这相当于数学归纳法的归纳步骤。只不过数学归纳法是一种论证方法，而递归是算法和程序设计的一种实现技术，数学归纳法是递归求解问题的理论基础。

扫一扫

视频讲解

5.1.5 递归的执行过程

为了讨论方便，将前面一般化的递归模型简化如下（即将一个“大问题”分解为一个“小问题”）：

$$f(s_1)=m_1$$

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

在求 $f(s_n)$时的分解过程是 $f(s_n)\rightarrow f(s_{n-1})\rightarrow\cdots\rightarrow f(s_2)\rightarrow f(s_1)$。

一旦遇到递归出口，分解过程结束，开始求值过程，所以分解过程是“量变”过程，即原来的“大问题”在慢慢变小，但尚未解决，遇到递归出口后便发生了“质变”，即原递归问题转化成直接可解问题。

其求值过程是 $f(s_1)=m_1\rightarrow f(s_2)=g(f(s_1),c_1)\rightarrow f(s_3)=g(f(s_2),c_2)\rightarrow\cdots\rightarrow f(s_n)=g(f(s_{n-1}),c_{n-1})$。

这样 $f(s_n)$便计算出来了。因此递归的执行过程由分解和求值两部分构成，分解部分就是用递归体将“大问题”分解成“小问题”，直到递归出口为止，然后进行求值过程，即已知“小问题”，计算“大问题”。前面的 fun(5)求解过程如图 5.3 所示。

在递归算法的执行中，最长的递归调用的链长称为该算法的**递归调用深度**。例如，求 $n!$对应的递归算法在求 fun(5)时递归调用深度是 5。

对于复杂的递归算法，在其执行过程中可能需要循环反复地分解和求值才能获得最终解。例如，对于前面求 Fibonacci 数列的 Fib 算法，求 Fib(6)的过程构成的递归树如图 5.4

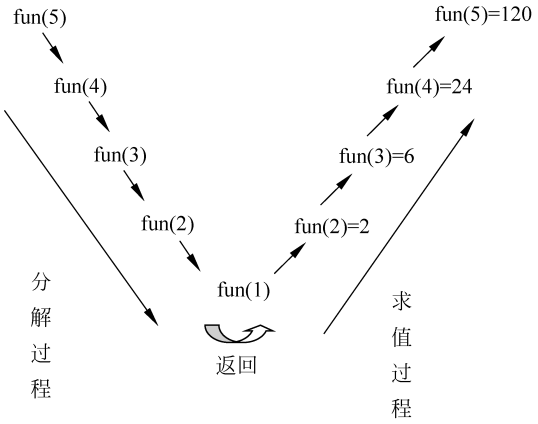


图 5.3 fun(5)求解过程

所示，向下的实箭头表示分解，向上的虚箭头表示求值，每个方框旁边的数字是该方框的求值结果，最后求得 Fib(6)为 8。该递归树的高度为 5，所以递归调用深度也是 5。

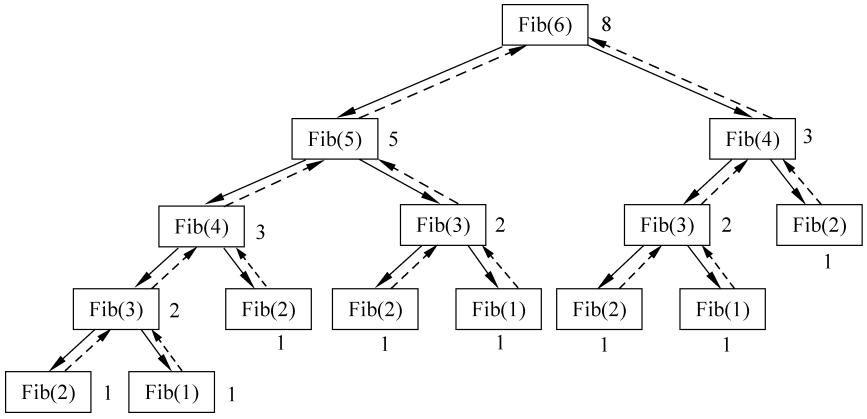


图 5.4 求 Fib(6)对应的递归树

那么在系统内部如何执行递归算法呢？实际上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。具体地说，递归调用的内部执行过程如下。

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

② 每次执行递归调用之前，把递归函数的参数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，使相应的参数值和局部变量值恢复为调用前的值，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))
```

在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的呢？

首先从 main()开始执行程序，将 main()信息进栈，遇到调用 S(1)，将 S(1)信息进栈，在执行递归函数 S(1)时又遇到调用 S(0)，再将 S(0)信息进栈，如图 5.5 所示。所以，自栈底到栈顶保存的信息的顺序是 main()→S(1)→S(0)。

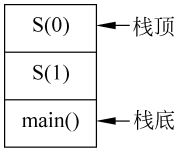


图 5.5 系统栈的状态

用递归算法的参数值表示状态，由于递归算法执行中系统栈保存了递归调用的参数值、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。例如有以下递归算法，其状态为参数 n 的值：

```
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)的结果如下：

```
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
```

从中看出，参数 n 的值在每次递归调用后都自动恢复了，有时说递归算法参数可以自动回退(回溯)就是这个意思。但全局变量并不能自动恢复，因为在系统栈中并不保存全局变量值。

扫一扫

视频讲解

5.1.6 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，实际上只有不可变类型的参数才保存在系统栈中，具有自动回退的功能，而可变类型的参数类似全局变量，不具有自动回退的功能。例如有以下 fun()函数，其中形参 i 为整数(不可变类型)，lst 为列表(可变类型)，每次调用时输出它们的地址：

```
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)
```

上述程序的执行结果如下：

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

从中看到，每次递归调用 fun()时形参 i 的地址均不同，而 lst 的地址始终是一样的，所以完全可以直接用全局变量 lst 替代形参 lst，对应的程序如下：

```
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)
```

扫一扫

视频讲解

5.1.7 递归算法的时空分析

从前面递归算法的执行过程看到，递归算法的执行过程不同于非递归算法，所以其时空分析也不同于非递归算法。如果非递归算法分析是定长时空分析，递归算法分析就是变长时空分析。

1. 递归算法的时间分析

在递归算法的时间分析中，首先给出执行时间对应的递推式，然后求解递推式得出算法的执行时间 $T(n)$，再由 $T(n)$得到时间复杂度。

例如，对于前面求解 Hanoi 问题的递归算法，求问题规模为 n 时的时间复杂度。求解方法是设大问题 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 $T(n-1)$。当 $n>1$ 时，大问题分解为两个小问题和一步移动操作，大问题的执行时间为两个小问题的执行时间+1，对应的递推式如下：

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

求解递推式的过程如下：

$$\begin{aligned} T(n)&=2T(n-1)+1=2(2T(n-2)+1)+1\\ &=2^2T(n-2)+2+1=2^2(2T(n-3)+1)+2+1 \end{aligned}$$

$$
\begin{aligned}
&= 2^3 T(n-3) + 2^2 + 2 + 1 \\
&= \cdots \\
&= 2^{n-1} T(1) + 2^{n-2} + \cdots + 2^2 + 2 + 1 \\
&= 2^n - 1 = O(2^n)
\end{aligned}
$$

所以问题规模为 n 时的时间复杂度是 $O(2^n)$。

2. 递归算法的空间分析

对于递归算法，为了实现递归过程用到一个递归工作栈，所以需要根据递归深度得到执行算法的空间。

例如，对于前面求解 Hanoi 问题的递归算法，求问题规模为 n 时的空间。求解方法是设大问题 Hanoi(n,X,Y,Z)的临时空间为 $S(n)$，则小问题 Hanoi($n-1$,X,Y,Z)的临时空间为 $S(n-1)$。当 $n>1$ 时，大问题分解为两个小问题和一步移动操作，但第一个小问题执行后会释放其空间，释放的空间可以被第二个小问题重复使用，所以大问题的临时空间为一个小问题的临时空间+1，对应的递推式如下：

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

求解递推式的过程如下：

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

所以问题规模为 n 时的空间复杂度是 $O(n)$。

5.2 递归算法的设计

扫一扫

视频讲解

5.2.1 递归算法设计的步骤

递归算法设计的基本步骤是先确定求解问题的递归模型，再转换成对应的 Python 语言方法。由于递归模型反映递归问题的“本质”，所以前一步是关键，也是讨论的重点。

递归算法的求解过程是先将原问题划分为若干子问题，通过分别求解子问题，最后获得原问题的解。这是一种分而治之的思路，通常由原问题划分的若干子问题的求解是独立的，所以求解过程对应一棵递归树。如果在设计算法时就考虑递归树中每一个分解/求值部分，会使问题复杂化。不妨只考虑递归树中第 1 层和第 2 层之间的关系，即“大问题”和“小问题”的关系，其他关系与之相似。

由此得出构造求解问题的递归模型(简化递归模型)的步骤如下。

① 对原问题 $f(s_n)$进行分析，假设出合理的“小问题”$f(s_{n-1})$。

② 假设小问题 $f(s_{n-1})$是可解的，在此基础上确定大问题 $f(s_n)$的解，即给出 $f(s_n)$与 $f(s_{n-1})$之间的关系，也就是确定递归体(与数学归纳法中假设 $i=n-1$ 时等式成立，再求

证 $i=n$ 时等式成立的过程相似）。

③ 确定一个特定情况（例如 $f(1)$ 或 $f(0)$）的解，由此作为递归出口（与数学归纳法中求证 $i=1$ 或 $i=0$ 时等式成立相似）。

【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() 为求两个值中较小值的函数。因此得到如下递归模型：

$$
\begin{aligned}
&f(a,i)=a[0] && i=0\\
&f(a,i)=\min(f(a,i-1),a[i]) && \text{其他}
\end{aligned}
$$

由此得到如下递归求解算法：

```
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
```

例如，若一个 $a=[3,2,1,5,4]$，调用 min(a,4) 返回最小元素 1。

扫一扫

视频讲解

5.2.2 基于递归数据结构的递归算法设计

具有递归特性的数据结构称为**递归数据结构**。递归数据结构通常是采用递归方式定义的。在一个递归数据结构中总会包含一个或者多个递归运算。

例如，正整数的定义为 1 是正整数，若 n 是正整数（$n\geqslant 1$），则 $n+1$ 也是正整数。从中看出，正整数就是一种递归数据结构。显然，若 n 是正整数（$n>1$），$m=n-1$ 也是正整数。也就是说，对于大于 1 的正整数 n，$n-1$ 是一种递归运算。

所以在求 $n!$ 的算法中，递归体 $f(n)=nf(n-1)$ 是可行的，因为对于大于 1 的 n，n 和 $n-1$ 都是正整数。

一般地，对于递归数据结构：

$$\mathrm{RD}=(D,\mathrm{Op})$$

其中，$D=\{d_i\}(1\leqslant i\leqslant n$，共 n 个元素）为构成该数据结构的所有元素的集合，Op 是递归运算的集合，$\mathrm{Op}=\{\mathrm{op}_j\}(1\leqslant j\leqslant m$，共 m 个递归运算），对于 $\forall d_i\in D$，不妨设 op_j 为一元运算符，则有 $\mathrm{op}_j(d_i)\in D$，也就是说递归运算具有封闭性。

在上述正整数的定义中，D 是正整数的集合，$\mathrm{Op}=\{\mathrm{op}_1,\mathrm{op}_2\}$ 由两个基本递归运算符构成，op_1 的定义为 $\mathrm{op}_1(n)=n-1(n>1)$；$\mathrm{op}_2$ 的定义为 $\mathrm{op}_2(n)=n+1(n\geqslant 1)$。

对于不带头结点的单链表 head（head 结点为首结点），其结点类型为 LinkNode，每个结点的 next 属性为 LinkNode 类型的指针。这样的单链表通过首结点指针来标识。采用递归数据结构的定义如下：

$$\mathrm{SL}=(D,\mathrm{Op})$$

其中，D 是由部分或全部结点构成的单链表的集合（含空单链表），Op＝{op}，op 的定义如下：

```
op(head)＝head.next                    # head 为含一个或一个以上结点的单链表
```

显然这个递归运算符是一元运算符，且具有封闭性。也就是说，若 head 为不带头结点的非空单链表，则 head.next 也是一个不带头结点的单链表。

实际上，构造递归模型中的第 2 步是用于确定递归体。在假设原问题 $f(s_n)$ 合理的“小问题”$f(s_{n-1})$ 时，需要考虑递归数据结构的递归运算。例如，在设计不带头结点的单链表的递归算法时，通常设 s_n 为以 head 为首结点的整个单链表，s_{n-1} 为除首结点外余下结点构成的单链表（由 head.next 标识，而其中的“.next”运算为递归运算）。

【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。

解 （1）设 $f(p)$ 的功能是正向输出单链表 p 的所有结点值，即输出 $a_0 \sim a_{n-1}$，为大问题。小问题 f(p.next) 的功能是输出 $a_1 \sim a_{n-1}$，如图 5.6 所示。对应的递归模型如下：

```
f(p)≡不做任何事件                    p＝None
f(p)≡输出 p 结点值；f(p.next)        其他
```

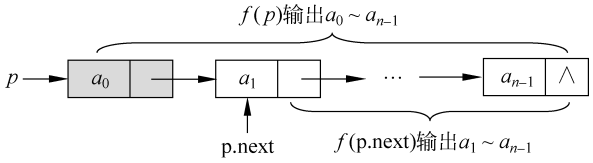


图 5.6 正向输出 head 的所有结点值

其中，“≡”表示功能等价关系。对应的递归算法如下：

```
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）设 $f(p)$ 的功能是反向输出 p 的所有结点值，即输出 $a_{n-1} \sim a_0$，为大问题。小问题 f(p.next) 的功能是输出 $a_{n-1} \sim a_1$，如图 5.7 所示。对应的递归模型如下：

```
f(p)≡不做任何事件                    p＝None
f(p)≡f(p.next)；输出 p 结点值        其他
```

对应的递归算法如下：

```
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='')
```

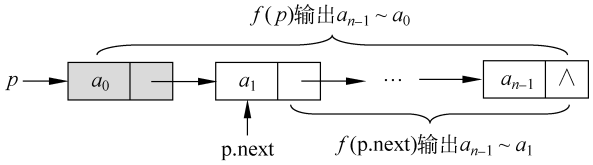


图 5.7　一个不带头结点的单链表

从中看出，两个算法的功能完全相反，但算法在设计上仅两行语句的顺序不同，而且两个算法的时间复杂度和空间复杂度完全相同。如果采用第 2 章的遍历方法，这两个算法在设计上有较大的差异。

说明：在设计单链表的递归算法时，通常采用不带头结点的单链表，这是因为小问题处理的单链表不可能带头结点，大、小问题处理的单链表需要在结构上相同，所以整个单链表也不应该带头结点。实际上，若单链表对象 L 是带头结点的，则 L.head.next 就看成是一个不带头结点的单链表。

所以在对采用递归数据结构存储的数据设计递归算法时，通常先对该数据结构及其递归运算进行分析，从而设计出正确的递归体，再假设一种特殊情况，得到对应的递归出口。

扫一扫

视频讲解

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转换成递归方法求解（如 n 皇后问题等）。

例如有一个位数为 n 的十进制正整数 x，求所有数位的和。例如 $x=123$，结果为 $1+2+3=6$。

不妨将 x 表示为 $x=x_{n-1}x_{n-2}\cdots x_1x_0$，设大问题 $f(x)=x_{n-1}+x_{n-2}+\cdots+x_1+x_0$ $(n\geqslant 1)$，由于 $y=x/10=x_{n-1}x_{n-2}\cdots x_1$，$x\%10=x_0$，所以对应的小问题为 $f(y)=x_{n-1}+x_{n-2}+\cdots+x_1$。假设小问题是可求的，则 $f(x)=f(x/10)+x\%10$。特殊情况是 x 的位数为 1，此时结果就是 x。对应的递归模型如下：

$$
\begin{aligned}
&f(x)=x && x\text{ 为一位整数}\\
&f(x)=f(x/10)+x\%10 && \text{其他}
\end{aligned}
$$

对应的递归算法如下：

```
def Sum(x):                                          # 求整数 x 的所有数位的和
    if x>=0 and x<=9:
        return x
    else:
        return Sum(x//10)+x%10
```

从中看出，在采用递归方法求解时，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。对于上一个问题，如果假设小问题为 $f(y)=x_{n-2}+x_{n-2}+\cdots+x_0$，就不如假设小问题为 $f(y)=x_{n-1}+x_{n-2}+\cdots+x_1$ 简单。

【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)对应的算法时间复杂度是

多少？

解 (1) 设 $f(x,n)$用于计算 x^n，则有以下递归模型。

$$
\begin{aligned}
&f(x,n)=x && n=1\\
&f(x,n)=x*f(x,n/2)*f(x,n/2) && n\text{ 为大于 1 的奇数}\\
&f(x,n)=f(x,n/2)*f(x,n/2) && n\text{ 为大于 1 的偶数}
\end{aligned}
$$

对应的递归算法如下：

```
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)对应的算法时间复杂度是 $O(\log_2 n)$。

【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
```

解 设 $f(x,y,\text{start},n)$用于创建左上角为(x，y)、起始元素值为 start 的 n 阶螺旋矩阵，如图 5.8 所示，共 n 行 n 列，它是大问题。$f(x+1,y+1,\text{start},n-2)$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。例如，如果 4 阶螺旋矩阵为大问题，那么 2 阶螺旋矩阵就是小问题，如图 5.9 所示。

对应的递归模型如下(大问题的 start 从 1 开始，在产生一圈元素时 start 依次递增)：

$$
\begin{aligned}
&f(x,y,\text{start},n)\equiv\text{不做任何事情} && n\leqslant 0\\
&f(x,y,\text{start},n)\equiv\text{产生只有一个元素的螺旋矩阵} && n=1\\
&f(x,y,\text{start},n)\equiv\text{产生}(x,y)\text{的那一圈；}f(x+1,y+1,\text{start},n-2) && n>1
\end{aligned}
$$

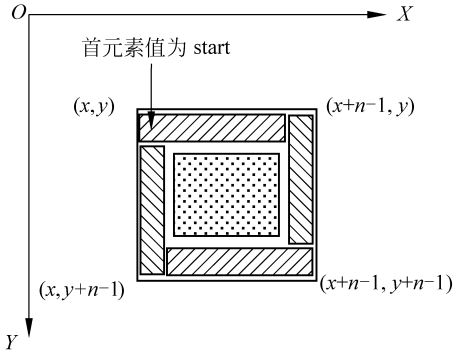


图 5.8 n 阶螺旋矩阵

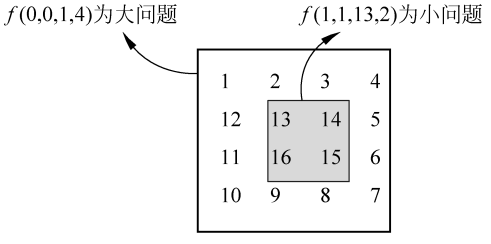


图 5.9 $n=4$ 时的大问题和小问题

对应的完整程序如下：

```
N=15
a=[[0]*N for i in range(N)]                    #存放螺旋矩阵
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()
```

扫一扫

视频讲解

【例 5.6】 采用递归算法求解迷宫问题，输出从入口到出口的所有迷宫路径。

解 迷宫问题在第3章介绍过，这里用 path[0..d]数组存放迷宫路径（d 为整数，表示路径中最后一个方块的索引），其中的元素[i,j]为迷宫路径上的方块。

设 mgpath(xi,yi,xe,ye,path,d)是求从入口(xi,yi)到出口(xe,ye)的所有迷宫路径，是“大问题”，当从(xi,yi)方块找到一个相邻可走方块(i,j)后，mgpath(i,j,xe,ye,path,d)表示求从(i,j)到出口(xe,ye)的所有迷宫路径，是“小问题”，则有大问题＝走一步＋小问题，如图5.10所示。

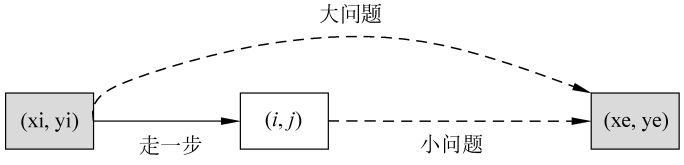


图 5.10 大、小问题的关系

求解上述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;      若(xi,yi)=(xe,ye),即找到出口
                           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
```

```
mgpath(xi,yi,xe,ye,path,d)≡d++；将(xi,yi)添加到 path 中；        若(xi,yi)不是出口
                              置 mg[xi][yi]＝－1；
                              对于(xi,yi)的每个相邻可走方块(i,j)，调用 mgpath(i,j,xe,ye,path,d)；
                              从(xi,yi)回退一步，即置 mg[xi][yi]＝0；
```

以第 3 章图 3.17 所示的迷宫为例，求入口(1,1)到出口(4,4)所有迷宫路径的完整程序如下：

```
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)方块对象添加到路径中
    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＝'')        #输出第 cnt 条迷宫路径
        for k in range(d＋1):
            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
                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，回退并恢复 mg 值

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

上述程序的执行结果如下：

```
[1,1]到[4,4]的所有迷宫路径:
迷宫路径 1: (1,1) (2,1) (3,1) (4,1) (4,2) (4,3) (3,3) (3,4) (4,4)
迷宫路径 2: (1,1) (2,1) (3,1) (4,1) (4,2) (4,3) (4,4)
```

本例算法求出所有的迷宫路径，可以通过比较路径长度求出最短迷宫路径(可能存在多条最短迷宫路径)。

5.3 练 习 题

扫一扫

自测题

1. 求两个正整数的最大公约数(gcd)的欧几里得定理是，对于两个正整数 a 和 b，当 $a>b$ 并且 $a\%b=0$ 时，最大公约数为 b，否则最大公约数等于其中较小的那个数和两数相除余数的最大公约数。请给出对应的递归模型。

2. 有以下递归函数：

```
def fun(n):
    if n==1:
        print("a:%d" %(n))
    else:
        print("b:%d" %(n))
        fun(n-1)
        print("c:%d" %(n))
```

分析调用 fun(5)的输出结果。

3. 有以下递归函数 fact(n)，求问题规模为 n 时的时间复杂度和空间复杂度。

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

4. 对于含 n 个整数的数组 $a[0..n-1]$，可以这样求最大元素值：

① 若 $n=1$，则返回 $a[0]$。

② 否则，取中间位置 mid，求出前半部分中的最大元素值 max1，求出后半部分中的最大元素值 max2，返回 max(max1,max2)。

给出实现上述过程的递归算法。

5. 设有一个不带表头结点的整数单链表 p，设计一个递归算法 getno(p,x)查找第一个值为 x 的结点的序号(假设首结点的序号为 0)，没有找到时返回 -1。

6. 设有一个不带表头结点的整数单链表 p(所有结点值均位于 1～100)，设计两个递归算法，maxnode(p)返回单链表 p 中的最大结点值，minnode(p)返回单链表 p 中的最小结点值。

7. 设有一个不带表头结点的整数单链表 p，设计一个递归算法 delx(p,x)删除单链表 p 中第一个值为 x 的结点。

8. 设有一个不带表头结点的整数单链表 p，设计一个递归算法 delxall(p,x)删除单链表 p 中所有值为 x 的结点。

5.4 上机实验题

5.4.1 基础实验题

1. 在求 $n!$的递归算法中增加若干输出语句，以显示求 $n!$时的分解和求值过程，并输出

求 5!的过程。

2. 采用递归算法求 Fibonacci 数列的第 n 项时存在重复计算，设计对应的非递归算法避免这些重复计算，并且输出递归和非递归算法的前 10 项结果。

5.4.2 应用实验题

1. 求楼梯走法数问题。一个楼梯有 n 个台阶，上楼可以一步上一个台阶，也可以一步上两个台阶。编写一个实验程序，求上楼梯共有多少种不同的走法，并用相关数据进行测试。

2. 假设 L 是一个带头结点的非空单链表，设计以下递归算法：

(1) 逆置单链表 L。

(2) 求结点值为 x 的结点个数。

并用相关数据进行测试。

3. 输入一个正整数 $n(n>5)$，随机产生 n 个 1～99 的整数，采用递归算法求其中的最大整数和次大整数。

5.5 LeetCode 在线编程题

扫一扫 视频讲解

1. LeetCode7——整数反转

问题描述：给出一个 32 位的有符号整数，你需要将这个整数中每位上的数字进行反转。

示例 1：输入 123，输出 321。

示例 2：输入－123，输出－321。

示例 3：输入 120，输出 21。

假设我们的环境只能存储得下 32 位的有符号整数，则其数值范围为$[-2^{31}, 2^{31}-1]$。根据这个假设，如果反转后整数溢出，就返回 0。

要求设计满足题目条件的如下方法：

```
def reverse(self, x: int) -> int:
```

扫一扫 视频讲解

2. LeetCode24——两两交换链表中的结点

问题描述：给定一个链表，两两交换其中相邻的结点，并返回交换后的链表。注意不能只是单纯地改变结点内部的值，而是需要实际地进行结点交换。例如，给定链表为 1-> 2-> 3-> 4，返回结果为 2-> 1-> 4-> 3。要求设计满足题目条件的如下方法：

```
def swapPairs(self,head:ListNode)-> ListNode:
```

扫一扫 视频讲解

3. LeetCode59——螺旋矩阵 Ⅱ

问题描述：给定一个正整数 n，生成一个包含 $1～n^2$ 的所有元素，且元素按顺时针顺序螺旋排列的正方形矩阵。例如输入 3，输出结果如下：

```
[
 [1,2,3],
 [8,9,4],
 [7,6,5]
]
```

要求设计满足题目条件的如下方法：

```
def generateMatrix(self,n:int)->List[List[int]]:
```

扫一扫

视频讲解

4. LeetCode52——n 皇后 Ⅱ

问题描述：n 皇后问题研究的是如何将 n 个皇后放置在 $n \times n$ 的棋盘上，并且使皇后彼此之间不能相互攻击。给定一个整数 n，返回 n 皇后不同的解决方案的数量。要求设计满足题目条件的如下方法：

```
def totalNQueens(self,n:int)->int:
```