第5章 熟练递归编程

引 言

第4章中讲解了递归函数以及递归思维，也带大家练习了很多递归的小程序，熟练递归编程是编程教育的重点！所以本章将讲解多个较为复杂和完整的例子，使读者能进一步熟悉递归算法的思维。在本章的前面，首先讲解一个在解决问题中十分常用也非常重要的思想：二分法思想。通过二分查找、求解算术平方根等小例子让大家熟悉二分法思想。然后再通过一些完整的实例来带大家熟练递归编程，如：求两个数的最大公因数、中国余数定理、解线性方程组、排列组合问题等。其中排列问题和组合问题是本章的重点，我们希望通过讲解各种可用的求解方法来培养大家思考问题的方式并拓宽思路。相信从本章中读者也会渐渐体会到计算机思维和它的美丽。

5.1 二分法求解问题

二分法思想在平时解决问题时十分重要，它可大大降低问题的复杂度，故本节首先讲解什么是二分法思想，然后通过几个小例子带领大家熟练掌握这一思想，并体会二分法思想的妙处。

5.1.1 什么是二分法

何为“二分法”？首先来看一个小游戏：以前电视台上有个很受大家欢迎的栏目《看商品，猜价格》，游戏规则是给出一件商品让你猜出它的准确价格，主持人给的提示只有“高了”或“低了”，如果在规定时间里猜中商品价格，这件商品就是你的了。例如，主持人给出一个微波炉的价格介于 200～1000 元，它的实际价格是 860 元。

其中一种猜价格的方法为：参赛者按照价格依次递增的顺序进行猜测，比如依次猜 300、400、500、600、700、800，主持人都会给出“低了”的提示，接下来猜 900，主持人则给出“高了”的提示，这时我们知道价格介于 800～900 元，然后又从 800 开始递增地去猜，重复上述操作，直到猜中为止。

上述方法显然很慢，那么有没有一种快速而简单的方法呢？我们可以这样猜，第一次猜 200～1000 元中间的价格 600 元，这时主持人会给出“低了”的提示，我们立马知道价格在 600～1000 元了，第二次猜 600 到 1000 的中间价格 800 元，这时主持人给出“低了”的提示，我们便知道价格在 800～1000 元，第三次再取中间价格 900 元，主持人给出“高了”的提示，而此时我们只用了 3 次就把区间锁定在 800～900 了，利用这种方法再去猜 800～900 的数，直到猜中为止。

综上所述，第二种方法每次将猜测的区间缩小到原来的一半，明显好于第一种方法。这种每次将解空间缩小为原来空间的一半，逐步逼近正确的解的方法就是二分法，图 5-1 所示为二分法的思路图。有的同学可能会说本书前面的排序例子，比如归并排序、快速排序也是用的二分法，这样的看法不够全面，我们在排序中是先将问题分成两部分，然后分别对每部分进行处理，再将处理结果合并成最终的答案，是一种“先分后合，分而治之”的思想，严格说这是分治法，就如同在 4.1.2 节中讲解用分治法的思想求数列的和、求数列的最小值和最大值一样；而二分法主要强调的是“分”，其基本思想是，每次将搜索的区间减少一半，因此可以快速缩小搜索范围，区别是二分法“分而未合”。

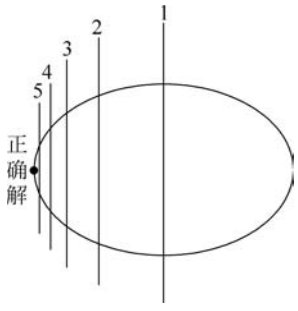


图 5-1 二分法思路图

5.1.2 在有序序列中使用二分法查找元素位置

本节主要研究在给定的一个有序序列中如何快速找出某个给定的元素位置，简称为二

分查找，5.1.1 节的猜价格游戏其实就是二分查找的例子。那么如何进行二分查找呢？我们可以这样做：每次取当前所剩序列的中间元素作为比较对象，若给定的值和中间元素相等，则查找成功；若给定值小于中间元素，则在中间元素的左半区继续查找；若给定值大于中间元素，则在中间元素的右半区继续查找。不断重复上述过程直到查找成功，或所查找的区域没有元素，查找失败。

上述二分查找方法的过程如图 5-2 所示，k 为要查找的值，查找区间是 $r_0 \sim r_n$（有序区间）。我们选择区间中间的值 r_{mid} 作为比较对象，若 $k=r_{mid}$ 则查找成功；若 $k>r_{mid}$，则在右半区进行查找；若 $k<r_{mid}$ 则在左半区进行查找。在左半区或右半区中进行查找时，继续使用取中间值比较的方式。

图 5-2　二分查找的基本思想图解

本节将通过讲解在有序序列中查找和插入元素这两个小例子来带领大家理解二分法思想。

1. 在有序序列中查找元素位置（二分查找）

【问题描述】 对于给定的有序序列 L，在该序列中用二分法的思想查找元素 k 是否存在于 L 中。若存在，则返回索引值；否则返回 −1 或者 False。

【解题思路】 假定一个有序序列为 L=[7,14,18,21,23,29,31,35,38,42,46,49,52]，利用二分查找的方法在该序列中查找值为 14 的元素。我们可以利用图 5-3 清晰地表示查找过程。最开始用变量 low 和 high 来标识当前查找的区间范围即为整个 L，之后每次计算出该区间的中点 mid=(low+high)//2，则将 L[mid]的值与 14 做对比，若 L[mid]>14，则将 mid−1 赋值给 high；若 L[mid]<14，则将 mid+1 赋值给 low，如此循环下去，直到找到该元素为止，即 L[min]值为 14。

问题：请各位同学想一想，为什么是将 mid−1 而不是 mid 赋值给 high？同样，为什么是将 mid+1 而不是 mid 赋值给 low？

了解了二分法查找的过程，就可以方便地用程序实现了，如<程序：非递归实现二分查找>所示，BinSearch_ non_recursive 函数有两个参数：L 和 k，该函数可实现在列表 L 中找元素 k 的位置的功能。在函数的开始，我们需要一些辅助变量：low 表示待查找区间的起始位置，high 表示结束位置。每次都取中间元素和 k 进行比较，若 k 比中间位置的元素小，则说明待查找的元素在左半区间，则令 high=mid−1，继续执行循环；若 k 比中间位置的元素大，则说明待查找的元素在右半区间，则令 low=mid+1，继续执行循环；若 k 和中间位置元素相等，则查找成功，可以直接返回中间位置 mid，无须继续循环。当然，while 循环要有终止条件：当不满足 low≤high 时，即若出现 low>high 的情况，则表明没有找到 k，查找失败，返回 −1。

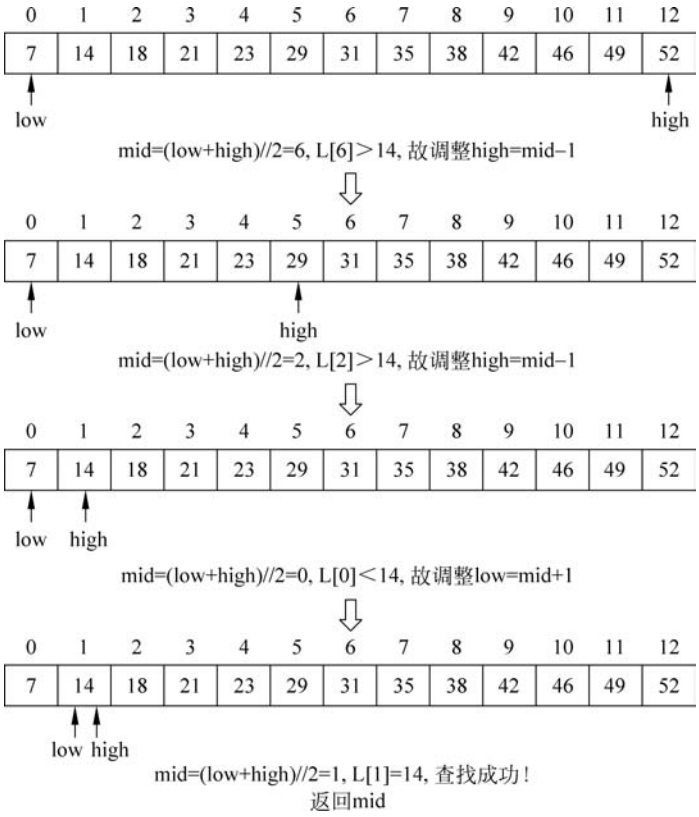


图 5-3　二分查找过程

```
#<程序：非递归实现二分查找>
def BinSearch_non_recursive(L,k):
    low = 0;high = len(L) - 1
    while(low <= high):
        mid = (low + high)//2              #注意整除符号
        if k < L[mid]: high = mid - 1
        elif k > L[mid]: low = mid + 1
        else: return mid
    return -1
```

当然，二分法查找的过程也可以用递归实现，递归的思路是要查找的元素 k 每次和列表中的中间元素比较，若不相等，则将查找区间分为左半区和右半区，若 k 小于中间元素，则在左半区继续查找；若 k 大于中间元素，则在右半区继续查找。递归终止的条件是查找区间的中间元素为要查找元素 k，即查找成功，或者查找区间为空，即查找失败。注意，在编写递归函数时需要返回两个值：第一个返回值是布尔值，True 表示找到，False 表示没找到；第二个返回值表示所找到的元素在序列中的索引。分析到这里，有的同学可能会问：当前函数所返回的索引是折半后新序列的索引，并不是原序列的索引啊！没错，所以如果查找的是右半部分，还需要在返回索引时加上另一半序列的长度。程序实现如<程序：递归实现二分查找>所示。

```
#<程序：递归实现二分查找>
def BinSearch(L,k):
    if L==[]:return False,-1                    #没找到
    if len(L)==1:
        if k==L[0]:return True,0
        return False,-1
    if k==L[len(L)//2]:return True,len(L)//2
    if k<L[len(L)//2]:return BinSearch(L[0:len(L)//2],k)
    else:
        flag,index=BinSearch(L[len(L)//2:],k)    #可否 len(L)//2+1?
        return flag,len(L)//2+index
```

函数 BinSearch(L,k)表示在列表 L 中查找元素 k，若 L 为空，则返回 False、－1，表示查找失败，递归终止。若 L 中只剩一个元素，则判断该元素是否和 k 相等，若相等则返回 True 以及 k 在当前 L 中的位置(即 0)，查找成功，递归终止；若不相等则返回 False、－1，递归终止。若 L 中的元素个数大于 1，则 k 和 L 的中间元素比较，若 k 等于 L[len(L)//2]，则查找成功，返回 True 以及 k 在当前 L 中的位置 len(L)//2，递归终止；若 k 小于 L[len(L)//2]，则继续调用 BinSearch(L[0:len(L)//2],k)，在 L 的左半区域继续查找，若 k 大于 L[len(L)//2]，继续调用 BinSearch(L[len(L)//2:],k)，在 L 的右半区域继续查找，并将调用函数返回的元素位置加 len(L)//2。

问题(见习题 5.1) 要如何改写程序，可使得若 k 大于 L[len(L)//2]，调用 BinSearch(L[len(L)//2+1:],k)。注意，要改动 return 的索引 len(L)//2+index。

兰 兰：为什么一定要用两个返回值呢？既然没有找到，则索引值可以返回－1，也就是说，可以用－1 表示没有找到，那么第一个布尔类型的返回值岂不是多余了？

沙老师：两个返回值是必需的！请仔细想一想，虽然在没有找到时会返回－1，但是这个－1 会返回给上一层函数，而在计算右半部分的索引值时需要加上序列长度值的一半，所以一旦－1 加上这个数之后就是一个正数，如果没有 False 标志，就无法判别是否真的找到了 k 这个值。所以再次提醒大家，递归函数的终止条件，包括终止的时候需要返回什么，很重要！

练习题 5.1.1 如何改写程序使得上述递归函数的返回值只有一个。

【解题思路】 上述递归函数之所以需要返回两个值，是因为在传递参数的过程中列表发生了改变，则其索引也相应地发生了变化，而我们要求的是原列表的索引，所以在对列表索引进行返回时会相应进行一定的加法运算而造成结果出错。所以，为了避免这种情况的发生，可以将新的参数列表在原列表中的起始位置作为参数进行传递。但是这就造成递归函数的参数与之前定义的不一致，所以可以通过嵌套函数的方式使得递归函数与外界的界面保持接口一致，且接口相对简洁。代码如<程序：嵌套递归实现二分查找 1>所示。

```
#<程序：嵌套递归实现二分查找 1>
def binary_r0_search(L,a):
    def r0_search(L,index_min):
```

```
        if len(L) == 0: return -1              # -1 代表没有找到
        mid = len(L)//2
        if L[mid]> a: x = r0_search(L[0:mid],index_min)
        elif L[mid]< a:x = r0_search(L[mid + 1:len(L)],index_min + mid + 1)
        else: x = index_min + mid
        return x
    return r0_search(L,0)
```

练习题 5.1.2 由于分片需要复制成为新的子列表,较为耗时,如何改写程序使得递归函数传递的列表参数不使用分片?

【解题思路】 将列表的索引作为递归函数的参数来传递,由参数中传递的索引来确定递归执行的子列表的起始和终止位置,这样就可以避免通过分片的方式来确定子列表。代码如<程序:嵌套递归实现二分查找 2>所示。

```
#<程序:嵌套递归实现二分查找 2>
def Binary_r1_search(L,a):
    def r1_search(index_min,index_max):
        if index_min > index_max: return -1
        mid = (index_min + index_max)//2
        if L[mid]> a: x = r1_search(index_min,mid - 1)
        elif L[mid]< a: x = r1_search(mid + 1,index_max)
        else: x = mid
        return x
    return r1_search(0,len(L) - 1)
```

2. 二分查找待插入元素位置

【问题描述】 编写程序,对于给定的有序序列 L,在该序列中用二分查找的思想找到待插入元素 k 应该插入 L 的哪个位置,使得插入后的序列仍然有序。

【解题思路】 其实这个问题与我们前面介绍插入排序的过程类似。插入排序时,就是每次将无序区的元素插入有序区,那么我们需要为该元素找一个正确的插入位置,该位置也就是有序区第一个比它大的元素的位置或者排在所有元素之后的位置,例如,有序列表 L=[1,3,5,8],我们需要将 k=7 插入 L 中,有序表中第一个比它大的元素是 8,它在 L 中的位置是 3,所以通过分片这样做:L[0:3]+[k]+L[3:],从而得到插入后的新的列表。在前面介绍插入排序的时候使用的是顺序查找的方法找到需要插入的位置,在了解了二分法的思想之后,便可以使用二分法找到该位置再将元素插入。所以可以编写一个函数 BinInsert(L,k),该函数可以实现对于给定的元素 k,利用二分查找在序列 L 中找到准确的插入位置,并将该位置的索引返回。代码如<程序:二分法查找插入位置>所示。

```
#<程序:二分法查找插入位置>
def BinInsert(L,k):
    low = 0;high = len(L) - 1
    while(low < = high):
        mid = (low + high)//2
```

```
            if k < L[mid]:high = mid - 1
            elif k > L[mid]:low = mid + 1
            else:return mid + 1
        return low
```

当在 L 中找到 k 时，返回的 mid 是该元素的位置，所以插入位置应为 mid 的后一个位置，即 mid+1；当没有找到该元素时，low 记录的位置就是要插入的位置。

二分法查找元素插入位置的过程也可以用递归实现，递归的思路是：首先将待插入的元素 k 和列表的第一个元素以及最后一个元素比较，如果小于第一个元素，则插入列表的起始位置，返回索引 index_min；如果大于列表的最后一个元素，则插入列表最后一个元素的下一位，返回索引 index_min+len(L)。如果以上两种情况都不成立，就将 k 和列表的中间元素 L[mid]比较，将比较区间分为左半区和右半区，若 k 小于中间元素，则在左半区继续比较，递归执行 r0_Insert(L[0:mid],index_min)；若 k 大于中间元素，则在右半区继续比较，递归执行 r0_Insert(L[mid+1:],index_min+mid+1)；若 k 等于中间元素，则插入位置就是中间元素的后面一位，返回索引 index_min+mid+1。递归终止的条件分为 3 种情况：一是列表为空时返回起始位置索引；二是待插入元素小于列表的第一个元素或大于列表的最后一个元素时，返回列表的第一个或者最后一个位置之后的索引；三是列表的中间元素等于要插入的元素 k 时，返回中间元素后一个位置的索引。代码如<程序：二分法递归查找插入位置>所示。

问题：可否去掉“if k<L[0]：return index_min”？

```
#<程序：二分法递归查找插入位置>
def binary_r0_Insert(L,k):
    def r0_Insert(L,index_min):
        if len(L) == 0: return index_min
        if k < L[0]: return index_min
        if k > L[len(L) - 1]: return index_min + len(L)
        mid = len(L)//2
        if L[mid]> k:x = r0_Insert(L[0:mid],index_min)
        elif L[mid]< k:x = r0_Insert(L[mid + 1:],index_min + mid + 1)
        else: x = index_min + mid + 1
        return x
    return r0_Insert(L,0)
```

5.1.3 求解算术平方根

二分法不止有二分查找这一处应用，还有很多问题可以利用二分法的思想来解决。再来看一个求解算术平方根的例子。在中学的时候我们就学过算术平方根这一概念，一个正数有正负两个平方根，其中，正的叫作算术平方根。本节将讲解如何利用二分法的思想求解算术平方根。

【问题描述】 在给定精度的前提下，求解一个实数 c 的算术平方根。

【解题思路】 比如 4 的算术平方根是 2，25 的算术平方根是 5，那么如果 c=10，就很难

求得 10 的精准的算术平方根的值，若想求精度为 0.01 的 c 的算术平方根的解，又该怎么算呢？

首先解释一下精度为 0.01 是什么意思：假设 c 的算术平方根的真实解为 x，而计算所求得的解为 x′，那么只要 $|x'-x|\leqslant 0.01$，就认为 x′为精度是 0.01 的 c 的算术平方根的解。

现在再来想想怎样计算 c 的算术平方根，可以想到这样一种方法：根据已知算术平方根的数确定 c 的算术平方根的范围，然后在这个范围内寻找答案。例如，如果 c=10，根据 3 的平方是 9，4 的平方是 16，所以 c 的算术平方根 g 一定满足：3<g<4。那么可以让 g=3，然后重复给 g 加一个很小的数 h，直到 g^2 足够接近 c，从而求得 c 的算术平方根 g。但当输出结果精度增加时，算法不得不减小步长 h，以避免 g+h 跳过可接受的解范围。随着精度的提高，h 的值减小，算法所需要进行查找的次数将大大增加。由此我们会想，如何能够减少逼近最终解的步骤，加快逼近过程，从而更快速地求得解。此时就可以利用二分法的思想，每次将求解值域的区间减少一半，从而快速缩小搜索的范围。

当 c≥1 时，解的范围是 0<x<c。不妨先假设 min=0，max=c。则 x 的值肯定介于 min 到 max，然后取中间值(min+max)/2，令该值为 g。比较 $|g^2-c|$ 与 0.01 的大小，如果 $|g^2-c|$ 在求解精度范围内(<0.01)，该值即为所求解；如果 $g^2-c>0.01$，表示 g 的值偏大，因此从 g 到 max 的区间不可能包含要找的最终解。于是，可以在算法中将新的 max 设定为当前 g 的值，并继续搜索。同样，如果 $g^2-c<0.01$，表示 g 的值偏小，此时可以将新的 min 设定为当前 g 的值。你发现了吗？每次循环都将求解范围缩小了一半。这就是二分法的求解过程。它大大加快了问题求解的速度。

下面以 max=10，min=0，中点值 g=5 为例，详细讲解二分法计算 10 的算术平方根的过程。首先，测试 g=5，发现 5×5=25>10，这表示正确的平方根值 x 不在 5～10 的这一区域内，所以可以不再考虑 5～10 的区域。于是可以将 max 设定为 5。这样，求解空间立刻变为了原来的一半。接下来在 0～5 的区间中，以同样的方式用二分法缩小解的空间。对于新的中间值 g=2.5，比较 2.5^2 与 10 的大小。因为 2.5^2 比 10 小，那么从 0～2.5 的区间就可以不考虑了，得到新的求解空间为[2.5, 5]。以此类推，经过 n 次循环后，所得到的范围就减到 $10/2^n$ 数量级。例如，当 n=40 时，$10/2^n$ 就已经到小数点后 11 位了。

设定精度为 0.000 000 000 01，改进后求算术平方根的具体算法描述如下：

输入：一个任意实数 c。

输出：c 的算术平方根 g。

(1) 令 min=0，max=c；

(2) 令 g′=(min+max)/2；

(3) 如果 g'^2-c 足够接近于 0，g′即为所求解 g；

(4) 否则，如果 $g'^2<c$，min=g′，否则 max=g′；

(5) 重复步骤(2)，直到满足条件，输出 g′，终止程序。

该算法实现代码如<程序：算术平方根运算——二分法>所示。

```
#<程序：算术平方根运算——二分法>
def square_root_2(c):
    i = 0 ; m_max = c ; m_min = 0
```

```
    g = (m_min + m_max)/2
    while (abs(g * g - c) > 0.00000000001):          # while 循环开始
        if (g * g < c): m_min = g
        else: m_max = g
        g = (m_min + m_max)/2
        i = i + 1
        print ("%d: %.13f" % (i,g))                  # while 循环结束
# 函数之外执行
square_root_2 (10)
```

如上程序用短短几行代码实现了求解算术平方根的功能。程序包括一个 while 循环部分以及一个 if 语句。循环部分判断 g^2 与 c 的大小，然后针对不同的情况，改变相应 m_min 或 m_max 的值，快速缩小求解空间。运行该程序的输出如下：

```
1:2.5000000000000
2:3.7500000000000
3:3.1250000000000
…
38:3.1622776601762
39:3.1622776601671
```

分析结果可知，该算法仅仅用了 39 次循环迭代便实现了算术平方根的计算，并且精度达到了 0.000 000 000 01。

5.1.4　二分答案问题——木料加工

二分法能够在每次的“选择”中去掉一半的可能解，这个选择是通过一个函数来确定的，这样的函数称为决策(Decision)函数。举例来说，当我们想要用小数来近似$\sqrt{10}$的时候，首先要确定解可能的范围为(3,4)，接下来在解范围内取中间值 g，要看真正的解在比 g 大还是比 g 小的范围内，我们用 g×g−10＞ε(ε 在前面小节中取 0.000 000 000 01，以表示真正解的精度)代表真正的解应该在比 g 小的范围内，否则真正的解在比 g 大的范围内。所以，g×g−10 这个函数就是一个二分法的决策函数。二分法的精髓就是这样的决策函数，我们记为 decision()。通过给 decision()函数传入每个解范围的中间值 mid，即 decision(mid)，根据该函数可以决定接下来选择 mid 的左边范围还是右边范围来继续判断。

在二分法的一些实例中，决策函数的计算可能会比较复杂，我们用木材加工这个例子来做说明。木材厂有一些原木，现在想把这些原木切割成 k 个长度相同的小段木料，每小段木料长度为 x(原木可以有剩余)，k 的大小是事先给定的，在切割时希望得到的小段木料长度 x 越大越好。这类问题我们可以称为二分答案问题，即对具有单调性的解空间进行二分查找，大多数情况下用于求解满足某种条件下的最大(小)值。所谓单调性，就是一个函数有可能单调递增或者单调递减。对于函数 f(x)，随着自变量 x 的增加，f(x)的值不会减小，这就是单调递增。单调递减也是用同样道理定义的。

【问题描述】　木材厂有 n 根原木，长度分别为 L[i]。如果需要从它们中切割出 k 根长度相同的小段木料，那么，对于这 k 根小段木料，每根的长度 x 最长为多少？(注意：原木的

长度都是正整数,要求切割得到的小段木料的长度 x 也是正整数,原木可以有剩余)。

例如,给定 3 根原木,长度记录在列表 L 中,L=[1,3,5]。当 k=1 时,切割得到的 1 根小段木料长度为 5(长度最长的那根原木即为所求);当 k=2 时,切割得到的 2 根小段木料长度均为 3(由长度为 3 和长度为 5 的原木各切得一根);当 k=3 时,切割得到的 3 根小段木料长度均为 2(由长度为 3 的原木切割得到一根,由长度为 5 的原木切割得到两根)。

【解题思路】 给定 n 根原木的长度,要求切出 k 根等长的小段木料,求每根小段木料长度 x 的最大值。解决这个问题可以采用二分法在解空间中查找 x 满足约束条件的最大值。约束条件就是当前长度 x 是否可以切出 k 根小段木料,容易得到所能切得的最大根数 $num=\sum_{i=0}^{len(L)-1} L[i]//x$。如果 num≥k,满足约束条件,即可以切出 k 根木料,但是这个 x 是不是最大值呢?如果 x 满足约束条件,同时 x+1 不满足约束条件,那么这个边界值 x 即为所求切得的最大小段木料长度。

接下来,我们就可以用二分搜索法来找这个最大长度 x。对于二分搜索,可以将过程分解为以下几步(说明:后续内容使用到的 mid 对应于上述内容的 x)。

- 首先,确定初始搜索的解空间 [low,high]。在该问题中,原木长度均为正整数,且要求切割得到的小段木料长度也是正整数,因此小段木料的最短长度为 1,所以将 low 初始化为 1;high 取最长原木的长度 max(L),因为切割得到的小段木料不可能比最长原木还长。例如,L=[1,3,5],初始解空间就是[1,5]。
- 其次,确定 Decision 函数。该问题的 Decision 函数是 $num=\sum_{i=0}^{len(L)-1} L[i]//x$。首先在解空间中使用二分法得到 mid=(low+high)//2,然后通过 Decision 函数判断当前 mid 能否满足切出 k 根小段木料的要求。如果当前 mid 值可以切割出所需段数(即 num≥k),说明当前 mid 值满足约束条件,但并不能确定其是否已经是最大值,可能 mid+1 也满足条件,因此我们需要选择 mid 值的右边继续尝试,令 low=mid+1,high 不变,这时 mid 值左边的一半解空间就被丢弃了;根据新的 low 和 high 计算当前 mid 值,如果当前 mid 值不满足约束条件,即在当前 mid 值下切割不出所需段数(即 num<k),显然 mid 偏大了,不能够切割出 k 个小段木料,那么就选择 mid 值左边继续尝试,令 high=mid−1,low 不变,也就是将 mid 值右边的一半解空间丢弃掉了。然后通过不断更新 low 与 high 以不断接近答案,直到 low>high 时结束,也就是当解空间变成[high,low]这样一个无意义的搜索范围时结束查找,结束时所得的 high 值就是可以加工出的小段木料的最大长度 x。

我们想想为什么输出是 high 值呢?首先说明,在每次缩小解范围至[low,high]时,最终的解一定小于或等于 high 值,因为当 high 值被收缩至 mid−1 时,mid 已经被验证不满足约束条件,也就是 high 值右边的解一定是不可行的(提醒:本题的解一定是正整数)。那为什么解不一定是大于或等于 low 值的呢?因为当 mid 值满足约束条件时,low 被赋值 mid+1,但若这时 mid 满足约束条件,但 mid+1 不满足约束条件,那么 mid 其实就是正确的解,但是执行 low=mid+1 就正好跳过了正确的解,显然 low 不可用于最终解的确定。

那为什么最终的 high 值就是最终解呢?思考最后一轮迭代时的情况,也就是当 low 和 high 相等时,此时 low=high=mid。若当前 mid 使得 num≥k,也就是当前 mid 能够满足

约束条件，执行 low＝mid＋1＞high，这时 mid 与 high 即为可满足约束的最大木料长度。但如果当前 mid 使得 num＜k，也就是说在目前[low，high]范围内不存在正确的解，这说明在之前解范围的缩小中，low 正好跳过了正确解，而且 low－1 作为之前某一轮的 mid 一定是满足约束条件的，此时执行 high＝mid－1 即为最大可满足约束的木料长度。因此，无论在什么情况下，我们输出 high 作为最终答案。

具体算法描述如下(实现代码见<程序：木料加工-二分法(方法一)>)。

输入：输入一个非负整数列表 L 和一个正整数 k；

输出：每小段木料的最大长度。

(1) 令 low＝1，high＝max(L)；

(2) 令 mid＝(low＋high)// 2；

(3) 计算长度为 mid 的情况下 num 的值，$num=\sum_{i=0}^{len(L)-1} L[i]//mid$；

(4) 如果 num 的值大于或等于 k，令 low＝mid＋1；

(5) 反之，如果 num 的值小于 k，令 high＝mid－1；

(6) 重复步骤(2)，直到不再满足 low≤high 的循环条件，即当 low＞high 时，则输出 high，终止程序。

```
#<程序：木料加工-二分法(方法一)>
def Maxlength(L,k):
    low = 1; high = max(L); count = 0
    while low <= high:
        count += 1
        num = 0
        mid = (low + high)//2
        for i in range(len(L)):
            num += L[i] // mid;
        if num >= k:
            low = mid + 1
        else:
            high = mid - 1
        print('{}:{} {}'.format(count,mid,num))
    return high
#函数之外执行
k = 8
L = [124,224,319]
Maxlength(L,k)
```

运行该程序的输出如下：

```
1:160 2
2:80 6
3:40 15
4:60 10
5:70 8
6:75 7
```

```
7:72 8
8:73 8
9:74 8
```

分析结果可知,算法用了 9 次循环迭代找到了长度为 124cm、224cm 和 319cm 三根原木,切成 8 段小段木料的每段最大长度是 74cm。

练习题 5.1.3 请讨论<程序:木料加工-二分法(方法一)>的代码中,为什么返回的是 high?

【解题思路】 请思考为什么 low 不可能是解?

练习题 5.1.4 可否将<程序:木料加工-二分法(方法一)>代码中的"if num >=k"改为"num > k","else"表示"num <=k"?这样修改后程序还对吗?

【算法再讨论】 上述方法虽然正确,但是让人疑惑终止条件和返回值的设定——为什么返回值是 high 值?需要经过大量的思考和讨论各种情况才能确定返回值,这不是良好的算法,一个优秀的算法不要晦涩难懂,要如何才能避免"烦人"的加 1 减 1 的误差?我们有没有什么系统性的思考方式以避免错误?

我们再思考前一个程序的瑕疵。我们描述如下的情况来说明解竟然不是一直在[low, high]区间内。在 while 循环中,假设 mid 已经是设为最终的解值,这时的 mid 一定是满足约束条件的,执行 low=mid+1 就直接跳过了最终解值,解竟然不是一直在[low,high]区间内。这时候要想得到最终解值,只能等待最后一轮,当 low=high=mid 时,decision(mid)不满足约束条件,执行 high=mid-1 得到最终的解。此时 high 比 low 还小!这不是有些怪异吗?因此我们就需要花费很多时间来检查 while 循环条件、high 值和 low 值的变化过程、二分法的搜索过程,以及最终解到底需要输出什么,来判断程序是否有错。遇到这种需要加 1 减 1 的问题的确让人很头疼,归根结底,还是因为解范围没有被限制在[low,high]区间内,让人没有办法从道理上明白到底程序是否正确。

原来的程序中在缩小解范围时有 low=mid+1,high=mid-1。我们考虑新的算法,当设定新的 low 与 high 时去掉+1、-1。希望保证最后的正确解一定在 while 循环中的[low,high)区间内。当 low 等于 high-1 时,就知道已经得到了最后的解了。这样的算法清楚明了。

我们想要保证解在任何时刻都大于或等于 low,小于 high,首先初始搜索解空间确定为[1,max(L)+1),也就是初始 low=1,high=max(L)+1,切割出长度为 max(L)+1 一定是无法达到的状态。同样在解空间内使用二分法得到 mid=(low+high)//2,然后通过 Decision 函数判断当前 mid 是否满足切出 k 根小段木料的要求,这里的 Decision 函数与上一种方法相同,都是 num=$\sum_{i=0}^{len(L)-1}$ L[i]//mid。当 num≥k 时,说明当前的 mid 值能够满足约束条件,这里我们执行 low=mid,那么一定可以保证的是,解一定是大于或等于 low 的;当 num<k 时,说明当前的 mid 值不能够满足约束条件,我们执行 high=mid,那么可以保证解一定在小于 high 的范围内。因此,当我们以上述步骤改变 low 和 high 值时,任意时刻我们都能够保证解在[low,high)范围内。并且,当解范围没有办法再继续收缩时,也就是 low=high-1 时,最终的解就被确定,因为解是正整数,因此输出 low 即可。

我们再重新思考一下。在该方法中,首先,初始解空间保证解一定在[low,high)范围

内；其次，在每次的解范围收缩过程中，执行 low＝mid 保证了 low 位置一定满足约束条件，执行 high＝mid 保证 high 位置一定不满足约束条件，因此保证解一定在[low，high)范围内；再者，当解范围没有办法再收缩时，只有 low 等于 high－1 这一种情况，因为如果继续收缩解范围，此时 mid＝low，Decision(mid)一定满足约束条件，low 依然等于 mid，位置不会再改变；最后，low 等于 high－1 时，因为解大于或等于 low，小于 high，且解为正整数，因此此时就是 low 满足约束条件、low＋1 不满足约束条件这样一个临界情况，因此 low 就是最终可切割出 k 根小段木料的最大长度。

因此，我们对<程序：木料加工-二分法(方法一)>做如下修改(方法二)：

(1) 初始 high 值由 high＝max(L)修改为

```
high = max(L) + 1
```

(2) low 和 high 值的更新，由 low＝mid＋1，high＝mid－1 修改为

```
low = mid
high = mid
```

(3) 终止条件由 while low<＝high 修改为

```
while low != high - 1
```

(4) 最终解返回值由 return high 修改为

```
return low
```

练习题 5.1.5 请按照方法二自行改写<程序：木料加工-二分法(方法一)>，请思考为什么 low 是最终的解，该方法和方法一的区别在哪里。

5.2 求两个数的最大公因数

相信大家都遇到过求最大公因数(Greatest Common Divisor，GCD)问题。比如求 12 和 8 的最大公因数，求得为 4，一般这样表示：gcd(12，8)＝4。如何编写程序求解最大公因数呢？本节介绍两种方法：方法一是通过分解因数的方式求解最大公因数；方法二是利用欧几里得(Euclid)算法。

5.2.1 因数分解法求最大公因数

对于求 GCD 问题，大家在中学时所学的方法就是先对两个数做因数分解，然后找出所有公共的因子，把它们相乘就得到了最大公因数。比如求 gcd(12，8)，那么首先对 12 进行因数分解，得到 12 的因子序列 A＝[2，2，3]，再对 8 做因数分解得到 8 的因子序列 B＝

[2,2,2]。那么 A 和 B 所有公共的因子为[2,2],所以 gcd(12,8)=2×2=4。根据上述算法,实现代码共需要做的就是两步:第一步,分别对两个数进行因数分解求得对应的因子序列;第二步,找到两个因子序列中的公共元素,把它们相乘从而得到最大公因数。

兰　兰:求最大公因数时,使用先因数分解再找所有公共因子的方式难道不好吗?

沙老师:这种方式虽然能找到最大公因数,但是这是极度没有效率的算法。我们还是先看看怎么实现这个算法,然后再与好的算法比较,你就会发觉两者有天壤之别。

对于第一步因数分解,曾经在 4.2.3 节中讲解过递归方式求解因数分解问题,为方便起见,再次贴出这段代码(<程序:因数分解 1>)。

```
#<程序:因数分解 1>
import math
def factors_1(L,x):                    #L 为最终返回的因子列表,x 为待分解的数
    for i in range(2,int(math.sqrt(x)) + 1):
        if x % i == 0:
            L.append(i)
            factors_1(L,x//i)
            break
    else: L.append(x)
```

同时给出另一种写法,见<程序:因数分解 2>,原理与<程序:因数分解 1>是一样的,此处不再赘述,只不过参数不一样。<程序:因数分解 1>中是通过将因子列表 L 作为参数在递归中进行层层传递的;而<程序:因数分解 2>中只有一个待分解数 x 作为参数,该代码是将因子列表 R 以 return 的方式在递归中进行层层传递的。我们在 4.2.3 节讲解递归的时候曾经提到过,递归编程传递结果比较常用的方式就是传参和 return 的方式,不推荐使用 global 的方式。这里给出了传参和 return 两种不同写法供参考。

```
#<程序:因数分解 2>
import math
def factors_2(x):                      #x 为待分解的数
    y = int(math.sqrt(x))
    R = []                             #R 为最终返回的因子列表
    for i in range(2,y + 1):
        if (x % i == 0):
            R.append(i)
            R = R + factors_2(x//i)
            break
    else: R = [x]                      #找不到因子,故为素数
    return R
```

有了第一步因数分解,再来看看第二步求公共因子该如何编写程序。思路很简单,仍假设求 gcd(12,8),在求得 12 的因子序列 A=[2,2,3]和 8 的因子序列 B=[2,2,2]之后,遍历 A 中的元素,每次检查该元素是否在 B 中,如果存在,则将该元素存放进列表 C 中,并在 B 中将其删除;若不存在,则继续查找 A 中的下一个元素,以此类推,直到循环结束,就可以得

到公共的因子 C=[2,2]。最后将 C 中的所有元素相乘，就得到了 12 和 8 这两个数的最大公因数 4。因数分解方法求解 GCD 问题代码如<程序：因数分解方法求解 GCD 问题>所示。

```
#<程序：因数分解方法求解 GCD 问题>
import math
def product(L):
    y = 1
    for i in L:y = y*i
    return(y)
def GCD_factors(x,y):                    #求解 x 和 y 的最大公因数
    Lx = factors_2(x)                    #见<程序：因数分解 2>
    Ly = factors_2(y)
    def search_and_delete(a, L):         #若元素 a 在列表 L 中,则删除 L 中的一个 a
        for i in range(len(L)):
            if L[i] == a:
                return True, L[0:i] + L[i+1:len(L)]
        return False, L
    R = []
    for e in Lx:
        found, Ly = search_and_delete(e,Ly)
        if found: R.append(e)
    return product(R)
```

其中 GCD_factors()函数为主体部分，用于求解 x 和 y 两个数的最大公因数。先利用 factors_2()函数分别求得两个数的因子序列，然后利用 search_and_delete()函数求得公共因子序列，最后用 product()函数将公共因子序列中的元素相乘，得到最大公因数。

虽然用因数分解法求最大公因数在中学时就学习过，但它的效率很低。后面将把它和更好的算法进行对比，就会发现有在时间开销方面远远优于因数分解法的更好的算法。

5.2.2 欧几里得算法求最大公因数

欧几里得算法可以通过比较灵活的计算方式求得两个数的最大公因数。假设两个数分别为 A 和 B，且 A>B。假设 gcd(A,B)=g，那么 A=a*g，B=b*g，则 A−B=(a−b)*g。g 仍然是 A−B 的因数。所以 gcd(A,B)=gcd(B,A−B)。同理，g 是 A−B，A−2B，甚至是 A−kB 的因数，只要 kB<A 即可。也就是 g 是 A 对 B 取余的因数。所以 gcd(A,B) = gcd(B,A % B)，以此类推，参数越变越小，直到 A % B=0 为止，此时 B 的值即为最大公因数。比如 A=120，B=82。则求解过程如下：

$$gcd(120,82)=gcd(82,38)=gcd(38,6)=gcd(6,2)$$

在上述过程中，最终计算得到 6 % 2 为 0，最大公因数就是 2。再举一个例子，A=12，B=7，求 gcd(12,7)。求解过程如下：

$$gcd(12,7)=gcd(7,5)=gcd(5,2)=gcd(2,1)$$

由于 2%1 为 0，所以 gcd(12,7)=1。可以通过一张图来更形象地解释求 gcd(12,7)的过程，如图 5-4 所示。大家也可以自己用画图的方法去求解一下 gcd(84,35)和 gcd(120,82)，求解示意图分别如图 5-5 和图 5-6 所示。

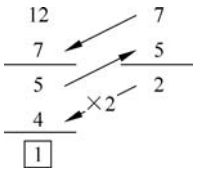


图 5-4 求解 gcd(12,7)

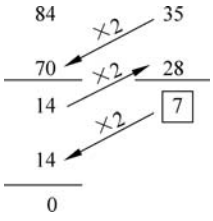


图 5-5 求解 gcd(84,35)

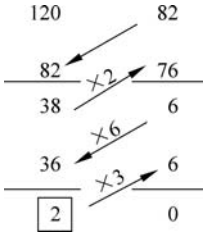


图 5-6 求解 gcd(120,82)

欧几里得算法通常也被大家叫作辗转相除法。相信大家在自己求解的过程中就会发现,欧几里得算法仍然是将大问题分解为小问题,例如,当我们要求解 gcd(12,7)时,可以将其缩小为求解 gcd(7,5),然后还可以继续将问题缩小为求解 gcd(5,2)……直到最终问题缩小到求解 gcd(2,1),已经不能再缩小了,从而返回所得到的解。

其原理是在每次取余化简的时候,这个因子 g 一直都在,并没有因为一系列的加减操作而被去掉,那么只要以这种方式一直化简,最终一定会化简到只剩下 g,也就得到了我们想要的答案。

有了该算法之后,就不难写出其实现代码,见<程序:欧几里得算法求最大公因数>。只需短短几行递归代码,就解决了这一问题,同时该算法也比 5.2.1 节中的解法更快捷(将在 5.2.3 节中详细比较)。

```
#<程序:欧几里得算法求最大公因数>
def GCD_Euclid(x,y):                    #计算 x 与 y 的最大公因数
    if x<y: x,y=y,x                     #确保 x>=y
    if x%y ==0: return(y)
    return GCD_Euclid (x%y,y)
```

兰 兰:其实用 Euclid 算法求最大公因数也是二分法的思想!

可以看到,每次求 gcd(A,B) = gcd(B,A mod B)的过程中,每次都必定有 A mod B≤(A/2)。比如求解 gcd(120,82)时有 38≤(120/2)、6≤(82/2);求 gcd(12,5)时有 5≤(12/2)……也就是每次问题的解空间都缩减了至少一半。我们可以来证明一下:每次 A 除以 B 所得的余数必然小于或等于 A 的一半(即 A % B<(A/2))。

定理:若 A>B,则 A % B 必定小于(A/2)。

证明:这个问题可分成两种情况,依次证明。

(1) 情况一:当 B≤(A/2)时。由于 A % B<B 一定成立(余数一定比除数小),所以 A % B<B≤(A/2)成立。

(2) 情况二:当 B>(A/2)时。A % B = A−B<(A/2)。

综上,得证 A % B<(A/2)。即每次计算都会将原问题缩小到原来的一半,衰减趋势与二分法是一样的。所以,并不是只有每次将问题分成左右两部分的才是二分法,类似欧几里得算法这种每次将问题缩减成一半的都归作二分法思想。

5.2.3 讨论因数分解法与欧几里得算法的优劣

1. 时间开销上的巨大差异

到目前为止，我们已经讲解了两种方法来求最大公因数：第一种是因数分解的方法求GCD，第二种是二分法的思想求解GCD，那么这两种方法在执行的时候有哪些差别呢？同学们或许还没有感受到二分法解决问题的优势到底在哪里，那么现在来试验一下，见<程序：算法比较1>。

```
#<程序：算法比较 1>
p = 1200; q = 248
print("p = %d, q = %d, GCD(p,q) = %d" % (p,q, GCD_factors (p,q)))
#输出结果：p = 1200, q = 248, GCD(p,q) = 8
print("p = %d, q = %d, GCD(p,q) = %d" % (p,q, GCD_Euclid (p,q)))
#输出结果：p = 1200, q = 248, GCD(p,q) = 8
```

可以看到，求得的结果是一样的，从时间上来看，二者都是1s内就完成了计算。那么如果换成更大一些的数呢？我们来试试，见<程序：算法比较2>。

```
#<程序：算法比较 2>
p1 = 128543041447753; q = 123456789 * 99
print("因数分解的方式求 GCD( %d, %d) = %d" % (p1,q,GCD_factors(p1,q)))
#输出结果：因数分解的方式求 GCD(128543041447753,12222222111) = 1
print("Euclid 方式求 GCD( %d, %d) = %d" % (p1,q, GCD_Euclid (p1, q)))
#输出结果：Euclid 方式求 GCD(128543041447753,12222222111) = 1
```

这两种方法仍然都可以在1s内就得到结果。如果需要求解更大的数呢？再来试试，见<程序：算法比较3>。

```
#<程序：算法比较 3>
p2 = 10625738533631454878458 51
p3 = 928176872663158109675503737928857 67 * 99
q = 123456789 * 99
print("Euclid 方式求 GCD( %d, %d) = %d" % (p2,q, GCD_Euclid (p2, q)))
#输出结果：Euclid 方式求 GCD(10625738533631454878458 51,12222222111) = 1
print("Euclid 方式求 GCD( %d, %d) = %d" % (p3,q, GCD_Euclid (p3, q)))
#输出结果：Euclid 方式求 GCD(91889510393652652857874870054956909 33, 12222222111) = 99
print("因数分解的方式求 GCD( %d, %d) = %d" % (p2,q,GCD_factors(p2,q)))
#很久得不到结果……
```

此时就看出了差别！q的值没有变，只是把p的值变得更大，利用欧几里得算法求gcd(p2,q)和gcd(p3,q)时，仍然可以在几秒内得到结果；而如果利用因数分解的方式求gcd(p2,q)，需要好几个小时，要是求gcd(p3,q)，至少需要好几年！

我们可以通过一个大致的计算来验证一下使用因数分解的方式求解GCD是否真的这么慢。已知p1、p2都是质数，所以分解因数的时候一定需要从1遍历到这个质数的算术平

方根，假设利用因数分解法求解 gcd(p1,q)需要 1s 的时间。那么 p2 比 p1 多了 10 位，在用因数分解方法的过程中，由于只是计算到 sqrt(x)，所以计算 p2 时会比 p1 多计算 5 位，已知 p1、p2 都是质数，也就是说对 p2 做因数分解时的遍历次数将是对 p1 做因数分解时的 10 万倍！已经假设求解 gcd(p1,q)是 1s，那么求解 gcd(p2,q)将是 10 万秒！

10 万秒是什么概念？先看看一天是多长时间：86 400s(1×24×60×60=86 400)，还没有 10 万秒多，所以保守估计，用因数分解的方法求 gcd(p2,q)需要一天多的时间！求 gcd(p2,q)已经这么慢了，那么求解 gcd(p3,q)呢？p3 在 p2 的基础上又多加了 10 位数，所以求解 gcd(p3,q)的时间将又是 gcd(p2,q)的 10 万倍！那绝对是需要很多年才能计算出来！相反，利用欧几里得算法无论是求解 gcd(p2,q)还是 gcd(p3,q)，都只需几秒就计算出来了。

2. 100 位数大不大

兰　兰：上面的例子中最大的数 p3 也不过是 30 多位数，100 位的数大不大？欧几里得算法还能很快地求解出最大公因数吗？

沙老师：100 位的数很大，但是欧几里得算法仍然可以很快求出最大公因数！我们可以从理论上对时间进行估算。

前面比较了因数分解法和欧几里得算法的执行时间，那么这里再来具体分析一下这两种算法的执行时间如何用数学方程式表示。对于求 GCD 的问题，数据规模就是待求解最大公因数的两个数 x 和 y(x>y)的数值大小，计算机需要比较的次数也就是需要遍历或者计算的次数。

对于因数分解法，在分解因数的时候，最坏的情况就是这个数为质数，我们要从 1 遍历到$\sqrt{x}$，那么在因数分解的过程中，执行时间就为$\sqrt{x}$，后面再求解公共因数的时间取决于因数分解所得到的因数的个数，我们姑且认为分解因数法求最大公因数的执行时间为$\sqrt{x}$。虽然说只需要分解 x 和 y 这两个数，但是这两个数可能会非常大，比如 x 有 100 位，那么这个数字将比宇宙中微尘的数量还巨大，根据前面的分析，因数分解法求解的时间绝对不只是好几年。

而对于欧几里得算法，我们前面已经证明过，该算法运用了二分法的思想，在求解的过程中，每次问题都缩减为原来的一半，然后在这一半中继续二分求解，直到最终找到答案。所以如果原来的问题规模为 x，则第一次二分后，问题规模下降为 x/2；再下一次还会继续二分，问题规模下降为(x/2)/2=x/4；再继续二分，下降为(x/4)/2=x/8；…；第 n 次二分会将问题规模下降为 $x/2^n$，直到最终求得解。可以看到，二分法可以让问题的规模以 log 以 2 为底的速度下降，即欧几里得算法的执行时间可以表示为 $\log_2 x$。所以即使 x 是 100 位、1000 位的数字，经过对数运算之后也会变得非常小！

如果在因数分解的时候需要从 1 遍历到一个 100 位的数字，那么程序将会耗时多久呢？首先假设遍历 100 个数耗时 10^{-6}s。

那么平均遍历一个数耗时为 10^{-8}s，则 1s 可以遍历 10^8 个数；1 年有 $365\times24\times60\times60=3.15\times10^7$s。所以 1 年可以遍历 $3.15\times10^7\times10^8=3.15\times10^{15}$ 个数。

现在有一个 100 位的数，则从 1 遍历到这个数需要 $10^{100}/(3.15\times10^{15})>10^{84}$ 年！

10^{84} 年是什么概念？据天文学家估算宇宙至今的寿命才仅仅 13.8×10^{9} 年，可见 10^{84} 年是多么巨大的时间长度。

然而对于欧几里得算法来说，因为它采用了二分法的思想，每次都是将问题缩减到一半，问题的复杂度是以 log 以 2 为底的趋势在递减的，所以用欧几里得算法求解 100 位数的最大公因数只需要计算 $\log_2 10^{100}=330$ 次，前面已经求出每次遍历耗时 10^{-8}s，所以总耗时为 $330\times10^{-8}\text{s}=3.3\times10^{-6}\text{s}$！这两种算法的时间差距真是天壤之别！至此，相信现在大家已经看到了二分思想的优势所在，它可以将问题规模以 log 以 2 为底的速度进行缩小，使得计算量很大的问题在短时间内就可以求解，所以，二分法的思想在解决问题中是相当重要的！

兰　兰：那如果用欧几里得算法计算 200 位的数字应该同样会很快吧？

沙老师：那是自然。想想看，其实计算公式只需要修改一下 10 的指数，变为 $\log_2 10^{200}\times10^{-8}=2\times3.3\times10^{-6}\text{s}=6.6\times10^{-6}\text{s}$，所以仍然是很快的。

练习题 5.2.1　给出两个 100 位的质数 x 和 y，请同学们自己用欧几里得算法求解一下 gcd(x,y)是否为 1？计算速度是否真的很快？

x＝39951925432136523804805053328611444026364045891242002698880914967138745160227366669788411817526658 31；

y＝52079632186466584739745193826566411113206191030243247242876565034343098711496731358470573711791361 83。

3. 不同算法的执行时间复杂度

兰　兰：没想到不同的解法竟然会带来这么大的时间上的差异！但是应该怎样衡量不同解法的好坏我还不是很清楚，您能再详细讲解一下吗？

沙老师：这个问题问得很好！衡量算法的好坏标准有很多，最常用的就是以执行时间来作为衡量标准，下面给大家详细讲解一下。

在程序的世界中，仅仅能写出代码只是初级的，还要仔细思考是否有更好的解法来解决这个问题，一个好的解法会把执行时间从很多年都计算不出结果优化到几秒钟就可以解决该问题。当然，有的同学会说执行时间只有真正让计算机执行了才知道是多久啊！没错，所以可以把计算机比较的次数当作执行时间，当写好一个算法后，可以分析该算法，然后用数学方程式来表示该算法随着数据规模 n 的增长，其执行时间(比较次数)将会以怎样的趋势来增长。

这里可以将执行时间的增长趋势大致分为两类：

(1) 指数型增长，比如 2^n；

(2) 多项式型增长，如 n^2、n、$n\log_2 n$、$\log_2 n$ 等。画出其中的几种函数，给大家一个更加直观的感受，如图 5-7 所示。

可以看到 $\log_2 n$ 的增长趋势是最小的，而 2^n 增长得最快，也就是说，随着数据规模 n 的增加，2^n 需要的执行时间越长，而 $\log_2 n$ 执行时间是 4 个函数中最短的。

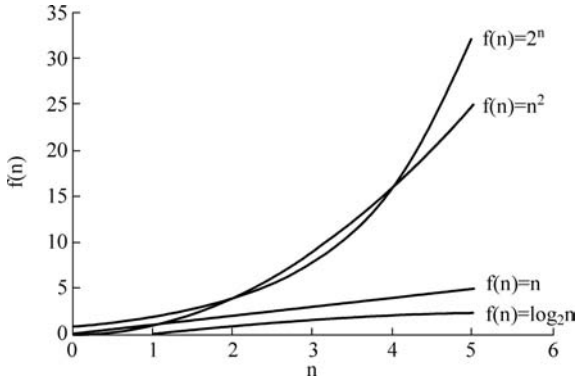


图 5-7　函数增长趋势图

在比较算法的执行时间时基本只看时间趋势，不计较它的系数，在计算机科学中，我们经常用 O 来表示它的时间复杂度。例如，一个算法的趋势是 $\log_2 n$，那么就说它的时间复杂度是 $O(\log_2 n)$，又如一个算法的趋势是 n^2，那么就说它的时间复杂度是 $O(n^2)$。

下面通过几个例子来看看不同算法对应的不同时间复杂度，例如：

(1) 要找出 n 个数中的最小数，需要遍历 n 个数，它的时间复杂度为 O(n)；或者把 n 个数加起来时间复杂度也是 O(n)。

(2) 在有序序列中用二分法查找元素位置的时间复杂度是 $O(\log_2 n)$。

(3) 如果将 n 个数用选择排序的方法从大到小排序，由于每次要从剩余的待排序数中选出最小数，所以执行次数是 $1+2+3+\cdots+n=(n+1)n/2$，我们只关注最高次数的项为 n^2，不重视它的系数，所以选择排序算法的时间复杂度为 $O(n^2)$。

(4) 代码如<程序：输出所有 k 位二进制数>所示：输入一个正整数 k，输出所有 k 位二进制数。这里使用递归的方法，假设 k 位二进制数的后 k－1 位已经确定，则 k 位二进制数有两种可能，第一位为 0 或 1，递归执行 F(k－1)，将第一位与后 k－1 位的递归执行结果相连接，程序的时间复杂度为 $O(2^n)$。

```
#<程序：输出所有 k 位二进制数>
def F(k):
    if k == 1: return ['0','1']
    R = F(k - 1);L = []
    for e in R: L.append('0' + e)
    for e in R: L.append('1' + e)
    return L
```

兰　兰：如果一个算法的时间复杂度是 O(n)或 $O(\log_2 n)$，这个算法是不是就是快速的算法？

沙老师：假如 n 是要处理的个数或位数，那么 O(n)的算法是快速的算法，假如 n 是输入的值，那么 O(n)是非常慢的，所以对于 n 代表的含义不要混淆。例如，n 代表 100 位数，那么 n 并不算大，而如果 n 代表 100 位数的值，那么 n 就非常大了。上文中因数分解的 n 就是值，所以时间复杂度为 O(n)的算法是非常慢的，而时间复杂度为 $O(\log_2 n)$的算法无论 n 是位数还是值，经过二分递减后，速度依然非常快。

经过上面的详细分析，相信大家已然体会到不同算法之间复杂度的差别，所以希望大家在编程的时候能够勤思考，在写出完美的函数的同时也能设计出优美的算法，从而高效地解决问题。

5.3 中国余数定理问题

我们曾经在第 2 章玩过一个游戏：找到一个 0～34 的数，这个数除以 7 余 3，除以 5 会余 2，请问这个数是什么数？答案是 17。在 2.1.4 节中我们讲解了用循环遍历的方式来求解该问题，但是同学们是否想过，如果这个数相当大，这两种方法还会适用吗？显然不能，可能要算上好几万年！这个问题被称为**中国余数定理**问题。本节将讲解如何更快速地解决中国余数定理这一问题。对这个问题解法的讨论，将会对我们的编程思维有较深的启示。

5.3.1 相关的基础知识

1. 什么是中国余数定理

通俗地讲，若有一个数 x 满足如下方程组 $x\%m_1=a_1$，$x\%m_2=a_2$，$x\%m_3=a_3$，…，$x\%m_n=a_n$，且 m_1，m_2，…，m_n 两两互质，则在 0～(m_1 m_2… m_n)－1 必定有解 x 并且是唯一解 x。这就是中国余数定理。本节假设只考虑相对于 m_1、m_2 的两个余数来求解。对应第 2 章中的例子，就是 x 需要满足两个方程组，其中 $m_1=7$、$m_2=5$、$a_1=3$、$a_2=2$，即方程组为：$x\%7=3$、$x\%5=2$，最后求得 x＝17。

以上述例子为例，再来讲解一个数学知识。从中国余数定理的“唯一解”可以知道每个自然数(0 到 m_1 m_2－1)都可以用一对数字(a，b)来唯一表示，当然 m_1 与 m_2 互质，其中 a 为该自然数除以 m_1 的余数、b 为该自然数除以 m_2 的余数。假设 $m_1=7$，$m_2=5$，那么所有 0～34 的数就有了如表 5-1 所示的表示方法。

表 5-1　(a，b)形式表示自然数

0	1	2	3	4	5	6	7	8	9	10	…
(0,0)	(1,1)	(2,2)	(3,3)	(4,4)	(5,0)	(6,1)	(0,2)	(1,3)	(2,4)	(3,0)	…

这是除了二进制、十进制等进制方式外的另外一种数字表示方式。可以看到，每个数字对只会出现一次，所以(0，2)就代表数字是 7。同时，数字对之间还可以做正常的加、减、乘等运算，比如 7＋3＝10，对应的数字对中(0，2)＋(3，3)＝(3，5)＝(3，0)，从表 5-1 可以看到数字对(3，0)表示的正好是自然数 10。

中国余数定理已经在实际中应用，比如在信息安全方面，最为广泛应用的 RSA 加密算法就应用了中国余数定理来节省运算开销。这种加密算法使用的数字长度至少有 200 位，这个 200 位数的域是由两个 100 位的质数 p 和 q 相乘而得的。为了减少在长度为 200 位的数上进行运算所产生的开销，就会将这个域上的数字，用两个 100 位的数 x % p 和

x % q来表示，然后使用这一对数进行运算，这样运算开销就会比较少。有兴趣的读者可以自行查阅资料进行学习。

2. 取余的基本性质

在中国余数定理中，我们看到了取余运算，在之前的章节仅仅只学习了何为取余运算，下面将深入学习取余运算的基本性质。

我们在数学上学习过取余运算：x mod p=r或在Python写成x %p = r，其定义为p能整除x−r，也可以将其写为x=r (mod p)。读作：x等于r相对于对p取余。

但是，为什么可以写作x“=”r (mod p)呢？在数学中的“=”(等于)关系的充要条件是要符合相等律、交换律和传递律。大家可以验证x=r (mod p)确实是等于的关系。相等律：x=x (mod p)；交换律：若x=r (mod p)，则r=x (mod p)；传递律：若a=b (mod p)，b=c (mod p)，则a=c (mod p)，都是可以从定义中轻易证明的。

当x=r (mod p)时，就是p能整除x−r(或x−r是p的倍数)。举个例子，我们写作10=3(mod 7)，是因为10−3是7的倍数，即7能整除10−3。那为什么10= −4 (mod 7)？同样，因为7能整除10−(−4)。−11= −4 (mod 7)也是因为7能整除−11−(−4)。所以，如果对7取余，可以将无限多个整数分为7个集合，并且这7个集合之间没有交集。将0～6称为对7取余的域。大家想想，这是很有意思的，一个无限大的集合，被分成有限个数的子集。而在一个子集中的所有数在取余的定义中是相等的，所以这些数可以在运算中随便地被更换，如图5-8所示。

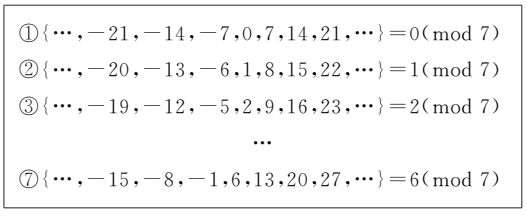


图5-8　对7取余的所有不相交集合

下面来看取余的3个性质。大家可以用定义来证明。

性质一：如果a=b(mod p)，则a+c=b+c(mod p)。例如，10=3(mod 7)，则10+2=3+2(mod 7)。

性质二：如果a=b(mod p)，c=d(mod p)，则a+c=b+d(mod p)。例如，10=3(mod 7)，2=−5(mod 7)，则12=−2(mod 7)。

我们可以来验证一下性质二，若10=3(mod 7)，2=−5(mod p)，则10+2=3−5(mod 7)是否正确？10−3是7的倍数，2−(−5)是7的倍数，并且12−(−2)也是7的倍数，所以结果是正确的。但是我们为什么不用中学时学习的方法来验证呢？比如计算10除以7的余数是否为3。因为通过判断是否可以被整除来定义取余运算时，只需要检查相减的结果是否为7的倍数，而不用计算除以7的余数到底是多少。

性质三：如果a=b(mod p)，则ak=bk(mod p)；同理，a=b(mod p)，c=d(mod p)，则ac=bd(mod p)。

其实，通过上面3个性质，可以得出只要等式相等，结合任意等式做加、减、乘运算的结

果还是相等的。

利用这 3 个性质，可以将复杂的式子简化。举一个较复杂的例子，(51×48+73−15)×8(mod 7)应该等于多少呢？可以看出该式在计算中其值会变得很庞大，但因为在取余之后可以把无限多的数划分为 mod 7 域的集合，然后就可以转换为 0～6(或 0～−6)的加减乘运算。所以，如同 51(mod 7)=2 一样，也可以将其他的数进行同样的取余运算，这些数都会落在 0～6 的集合中，利用取余的 3 个性质，则(51×48+73−15)×8(mod 7)=((2×−1)+3−1))×1(mod 7)，最后结果为 0。

再比如：$8^{100}=1(\text{mod } 7)$也一定是正确的。假如你真的计算 8 的 100 次方，那就太傻了。为什么呢？因为 8=1(mod 7)，则 $8^{100}=1^{100}(\text{mod } 7)$，而 1^{100} 仍然为 1，所以 $8^{100}=1(\text{mod } 7)$。

那么，2^{32} 应该等于多少呢？$2^1=2(\text{mod } 7)$，$2^2=4(\text{mod } 7)$…当计算 2^4 时，可以将 2^4 分解为 $2^2\times2^2$，所以可以将等式写作 $2^2\times2^2=4\times4\ (\text{mod } 7)=2(\text{mod } 7)$，则 $2^4=2(\text{mod } 7)$。同理，可以将 2^8 分解为 $2^4\times2^4$，$2^4\times2^4=2\times2(\text{mod } 7)$，则得到 $2^8=4(\text{mod } 7)$。所以，最后可求得 $2^{32}=4(\text{mod } 7)$。

练习题 5.3.1 请求解 $3^{32}=x(\text{mod } 7)$，$4^{32}=x(\text{mod } 7)$，这两个等式中的 x 分别为多少。

【答案】 因为 $3^2=2(\text{mod } 7)$，则 $3^4=2\times2(\text{mod } 7)=4(\text{mod } 7)$，$3^8=4\times4(\text{mod } 7)=2(\text{mod } 7)$，$3^{16}=2\times2(\text{mod } 7)=4(\text{mod } 7)$，最后，可得 $3^{32}=2(\text{mod } 7)$。所以，该式中 x 为 2。

同理，因为 $4^2=2(\text{mod } 7)$，则 $4^4=2\times2(\text{mod } 7)=4(\text{mod } 7)$，$4^8=4\times4(\text{mod } 7)=2(\text{mod } 7)$，$4^{16}=2\times2(\text{mod } 7)=4(\text{mod } 7)$，最后，可得 $4^{32}=2(\text{mod } 7)$。同样，该式中 x 为 2。

练习题 5.3.2 写 Python 程序算出 a^xmod b 的值。函数为 mod (a, x, b)，返回 a^xmod b 的值。假设 a、b 是最多为 10 位的整数，而 x 可以是最多为 200 位的整数。请用递归的思维来编写此程序。

3. 什么是 1/x（mod p）

在求解中国余数定理的过程中，要通过求解倒数来解决问题。求倒数是解决中国余数定理的重要步骤，下面我们来深入学习倒数。

在小学课本中倒数是这样定义的：如果 a 为 x 的倒数，写为 a=1/x→ax=1。按照这样的定义，倒数可能为分数。然而这里学习的倒数是 x 对 p 取余的倒数，其范围是 x 对 p 取余的域，所以它是一个 0～p−1 的整数。例如，求 3 对 7 取余的倒数，1/3(mod 7)应该是个整数。那么 x 对 p 取余的倒数的定义是什么呢？其实这个定义很简单，就是 1/x (mod p)=a，就是 ax=1 (mod p)。

要特别注意的是，只有在 x 与 p 互质的时候，x 对 p 取余有倒数。例如，p 为 4 时，2 对 4 取余就不会有倒数，2a=1(mod 4)这个等式是不成立的，因为 2a 必然为偶数，对 4 取余不可能为 1。但是 3a=1(mod 4)是可以的，显而易见的，当 p 为质数时，小于 p 的每个数对 p 取余都存在倒数。

接下来如何找到 a 呢？以对 7 取余为例，可以用一个个试的方法，求 2 对 7 取余的倒数 2a=1(mod 7)，当 a 为 4 时，可得 4×2=1(mod 7)，所以 2 对 7 取余的倒数为 4。同理，3、4、5 对 7 取余的倒数也可以通过 3a=1(mod 7)、4a=1(mod 7)和 5a=1(mod 7)分别得出，如表 5-2 所示。

表 5-2　对 7 取余的倒数

1/2(mod 7)	1/3(mod 7)	1/4(mod 7)	1/5(mod 7)
4	5	2	3

兰　兰：这个倒数不是一个分数哦，它一定是 0～p－1 的整数。

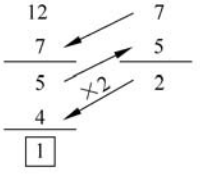
4. 如何快速求 1/x（mod p）

当 p 非常大，例如，p 有上百位这么大时，用一个个试的方法是不可取的。如同前面使用因数分解求最大公因数，使用几万年的时间也得不出结果，那么如何在 1 秒内算出它的倒数呢？如同使用欧几里得算法求最大公因数只需 1 秒钟的时间，要用到 log 函数的概念了，这个问题可以用欧几里得算法加以扩展，用它快速算出对应的倒数。

当要求解 x 对 p 取余的倒数时，第一步要确定 x 与 p 必须互质。第二步，要求解 ax＋bp＝1，x 对 p 取余的倒数就是 a。ax＋bp＝1 的含义为给定 x 与 p，要求出对应的 a 与 b，使得 ax＋bp＝1。为什么可以从 ax＋bp＝1 中求得 x 对 p 取余的倒数？因为 ax－1 是 p 的倍数，所以 ax＋bp＝1→ax＝1（mod p）。也就是 a 与 x 互为倒数。

在数学中，若 x 和 p 已知，则必定存在整数 a、b，使得 ax ＋ bp ＝ gcd(x,p)。当 x 与 p 互质时，gcd(x,p)＝1，即 ax ＋ bp ＝ gcd(x,p)＝1 中 a 为 x 对 p 取余的倒数。那么如何求解 a 和 b？

假设求 7 对 12 取余的倒数，可以以 7a＋12b＝gcd（7,12)为例，因为 gcd（7,12)是以 7 和 12 为系数进行加、减、乘运算的结果，所以，可以在欧几里得算法上加以改变，利用图 5-9(a)求解 gcd（7,12)，逆向推理求得结果如图 5-9(b)所示。请注意，这种逆向推理是为了理解原理，不是我们将来的程序所使用的方式。



1=5–4
1=5–(2×2)
1=5–2×(7–5)
1=5–2×(7–(12–7))
1=(12–7)–2×(7–(12–7))
1=3×12+(–5)×7

(a) 欧几里得算法求解gcd(7, 12)　　(b) 逆向推理gcd(7, 12)

图 5-9　求解 gcd(7,12)

通过推理，可以得出 a＝－5，b＝3。因为 7 对 12 取余的倒数应该介于 0～11，所以如果求得负数，我们需要一直对 a 加 12，当结果为正时停止。此时可以得到－5＋12＝7，7 就是 7 对 12 取余的倒数。而求得的 b＝3 刚好是 12 对 7 取余的倒数，所以可以通过欧几里得算法同时求得 7 对 12 取余的倒数和 12 对 7 取余的倒数。

事实上，我们不需要在求得 gcd(7,12)之后，再从后往前推来求解 a 和 b。在求解 gcd（7,12)的过程中其实都是对 7 和 12 的加、减、乘的运算，可以用向量(m,n)的形式来表

示求 GCD 过程中的每个数，该数字为 7m＋12n。那么，7 可以表示为(1,0)，即 7＝1×7＋0×12；12 可以表示为(0,1)，即 12＝0×7＋1×12。然后利用(m,n)来进行加、减、乘的计算。这样 7 和 12 的加、减、乘运算就可以用简单的向量的加、减、乘运算来表示，相对应位置的值做运算，例如，12－7＝5 的运算就是(1,0)－(0,1)等于(1,－1)，而(1,－1)就是 5；(7×2＋12×3)×k 可以用相应的位置(2,3)乘以 k 表示，即(2k,3k)。

最后，求解 gcd(7,12)的问题就变成了求解 gcd((1,0),(0,1))的问题。最终向量(3,－5)表示 3×12－(－5)×7＝1，其中－5＋12＝7 即为 7 对 12 取余的倒数，3 为 12 对 7 取余的倒数。具体计算过程如图 5-10 所示。

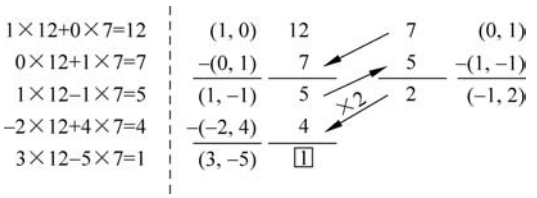


图 5-10　求解倒数示意图

有了这个思路，就可以在原来欧几里得算法求解 GCD 的代码上做一些修改。在求解的过程中记录下这些数字对，所传的参数应该有 4 个：x、y、Vx、Vy。其中 x 和 y 就是待求解最大公因数的两个数，变量 Vx 以[m,n]的形式记录 x 的向量表示形式，变量 Vy 以[m,n]的形式记录 y 的向量表示形式。所以，若求 12 和 7 的最大公因数，则调用函数的时候传递的参数应为 Extended_Euclid (12,7,[1,0],[0,1])，具体的代码如<程序：欧几里得算法求最大公因数扩展>所示。

```
#<程序：欧几里得算法求最大公因数扩展>
def Extended_Euclid(x,y,Vx,Vy):
    r = x % y; z = x//y
    if r == 0: return(y,Vy)                    #y 即为 GCD,并且 Vy[0] * x  + Vy[1] * y = GCD
    Vx[0] = Vx[0] - z * Vy[0]
    Vx[1] = Vx[1] - z * Vy[1]
    return Extended_Euclid(y, r, Vy, Vx)
```

要求解 7 对 12 取余的倒数，因为欧几里得算法要保证 x≥y，此时在调用 Extended_Euclid()函数时应该是 Extended_Euclid(12,7,[1,0],[0,1])，得到的返回结果是(y,Vy)＝(1,[3,－5])，其中 y＝1 是 12 和 7 的最大公因数，Vy ＝[3,－5]是最大公因数的另一种表示形式，现在需要的是 y 对 x 取余的倒数 Vy[1]，因为一定要保证倒数在 0～p－1 的范围内，所以应返回 Vy[1]%12＝7。同理，求解 12 对 7 取余的倒数，此时调用 Extended_Euclid()函数 Extended_Euclid(12,7,[1,0],[0,1])，得到的返回结果是(y,Vy)＝(1,[3,－5])，则 12 对 7 取余的倒数为 Vy[0]%7＝3。有了这一思路，就可以写出<程序：求倒数>。

```
#<程序：求倒数>
def Mod_inverse(e, n):
    Vx = [1,0]; Vy = [0,1]
    if e > n:                    #因为要 Euclid 算法要确保 e≥n
        G,X = Extended_Euclid(e,n,Vx,Vy)
        d = X[0] % n
    else:
        G,X = Extended_Euclid(n,e,Vx,Vy)
        d = X[1] % n
    return d
```

可以按照<程序：求倒数检验>的方式来检验一下，可得求解是正确的。

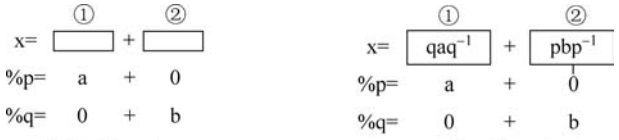
```
#<程序：求倒数检验>
p = 12; q = 7
print("%d's inverse (mod %d) = %d"%(p,q,Mod_inverse(p,q)))
#输出结果：12's inverse (mod 7) = 3
print("%d's inverse (mod %d) = %d"%(q,p,Mod_inverse(q,p)))
#输出结果：7's inverse (mod 12) = 7
```

5.3.2 中国余数定理问题的求解

有了基本概念之后，再来学习如何快速求解中国余数定理问题。这里只以求解满足两个方程组的 x 为例。设一个未知数 x 满足 x%p=a、x%q=b，其中 p 与 q 互质，则如何找到 0～pq−1 范围内的一个解 x。

举一个具体的例子：x%12=2、x%7=1，求解 0～83 的 x 的解，则该例中 p=12、q=7、a=2、b=1。

【分析过程】 如图 5-11(a)所示，令 x 等于①、②两部分数相加，若想使得 x%p=a，x%q=b，则①中应该有 a 且②中应该有 b。而①中必须有 q 相乘才能使得 x%q 的时候①的部分为 0；同理，②中必须有 p 相乘才能使得 x%p 的时候②的部分为 0，此时①中有 qa，②中有 pb，但仅有这些还不够，qa%p 的结果并不一定会是 a；同理，pb%q 的结果并不一定会是 b。因此，还需要在①的部分添加 q 的倒数 q^{-1}，满足 $qq^{-1}=1$，那么①部分变为 qaq^{-1}，此时①部分既能满足除以 p 余 a，又能满足①部分刚好可以整除 q；同理，在②的部分也应添加 p 的倒数 p^{-1}，满足 $pp^{-1}=1$，那么②部分变为 pbp^{-1}，此时②部分既能满足除以 q 余 b，又能满足②部分刚好可以整除 p。如此，就凑成了如图 5-11(b)的形式。



(a) 求解x的示意图1　　(b) 求解x的示意图2

图 5-11　求解 x 的示意图

现在唯一要求解的是 q^{-1} 和 p^{-1}。这里 q^{-1} 和 p^{-1} 就是我们在相关基础知识中介绍的取余运算的倒数：对于式子 $qq^{-1} \bmod p = 1$，且 p 与 q 互质，则在 1～p−1 范围内必定存在一个解 q^{-1} 使得该式子成立，那么 q^{-1} 就是 q 对 p 取余的倒数。同理，对于 $pp^{-1} \bmod q = 1$，p^{-1} 就是 p 对 q 取余的倒数。至此，便了解了中国余数定理的所需求解变量。

前面学习了利用欧几里得算法来求解倒数的方法，通过这种方法，可以求得图 5-11(b)中为了计算 x 所需的所有变量的值，①的部分：$qaq^{-1}=7\times2\times(-5)=-70$，而 −70 刚好除以 12 余 2，且 −70 能整除 7，如果将(−5)%p 使其变为 0～p−1 的整数，则有 −5+12=7，7 是 7 相对 12 而言的倒数；②的部分：$pbp^{-1}=12\times1\times3=36$，而 36 刚好除以 7 余 1，且 36 能整除 12。那么再将①、②部分组合，求得 x：$x=qaq^{-1}+pbp^{-1}=-70+36=$

50。所以，在0～83这个范围内，可以除以12余2、除以7余1的数为50。我们来验证一下这个解：50%12=2，且50%7=1，答案正确！至此，便求解了两个方程的中国余数定理问题。

【编程思路】　由图5-11(b)可以看到，当给定一个有两个方程的中国余数定理问题时，就已经得知了变量p、q、a、b的值，只需要再计算出p^{-1}和q^{-1}就可以求得x。所以现在求解中国余数定理的问题变成了如何求解倒数的问题。

由于前面通过对欧几里得算法进行扩展，已经实现了求倒数的程序<程序：求倒数>。现在就可以写下求解中国余数定理问题的完整代码。首先说一下整体代码的思路：对于求解两个方程的中国余数定理问题，主体函数需要4个参数：p、q、a、b，即Chinese_remainder(p, q, a, b)，返回值就是0～pq－1的一个解x，那么该函数内部具体都需要做些什么？

(1) 首先，检查p和q是否互质，如果否，则不符合中国余数定理的要求，直接返回错误。

(2) 计算p相对q而言的倒数p^{-1}；计算q相对p而言的倒数q^{-1}。

(3) 根据图5-11(b)中的公式计算$x=(qaq^{-1}+pbp^{-1})\%(pq)$。

具体代码如<程序：中国余数定理问题>所示。

```
#<程序：中国余数定理问题>
def Chinese_remainder(p, q, a, b):
    if GCD(p,q)!= 1: return -1
    inv_p = Mod_inverse(p,q)
    inv_q = Mod_inverse(q,p)
    return (q * inv_q * a + p * inv_p * b) % (p * q)      #注：求得的解需要在 0～pq－1
#以下为主函数
p = 12; q = 7
print("中国余数定理问题:找到 x,即 x mod( % d, % d) = ( % d, % d). x =  % d" % (p,q,2,1,Chinese_
remainder(p,q,2,1)))
#输出结果为 中国余数定理问题:找到 x,即 x mod(12,7) = (2,1). x = 50
```

5.4　关于递归函数开销的讨论

递归函数是一个很优美的东西，但直接将递归关系式编写成递归函数的形式将会有一些额外的开销，主要有3种：

(1) 函数调用产生的开销；

(2) 参数传递时产生的开销；

(3) 可能重复计算产生的开销。

此外，递归函数还有深度限制。在Python的默认执行环境下递归只允许最多1000层的深度。所以在编写递归的时候，要小心诸如f(n)=f(n－1)＋kn等关系式，因为这类关系式在n大于1000的时候，递归的深度就超过1000了，这在Python的执行环境中就会出现问题。但是假如递归关系式是f(n)=2f(n/2)＋n这种二分的形式，那么n的值就可以达到2^{1000}，这个值对大部分的应用都是足够大的。所以在设计递归关系式时要尽量用二分法的方式。

5.4.1 函数调用的开销

想要了解函数调用产生的开销,首先要了解调用函数的执行过程。

程序中调用函数时,将调用其他函数的函数称为主调函数(Caller);而被主函数调用的其他函数称为被调函数(Callee)。一个函数很可能既调用别的函数,又被另外的函数调用,但我们只关注某一次调用过程中谁为主调函数,谁为被调函数。在图 5-12 所示的函数调用过程示意图中,第一次为 Fun0 函数调用 Fun1 函数,其中 Fun0 函数为主调函数,Fun1 函数为被调函数。第二次为 Fun1 函数调用 Fun2 函数,此时 Fun1 函数为主调函数,Fun2 函数为被调函数。至于递归函数的调用,由于是自己调用自己,所以递归函数既是主调函数又是被调函数。

发生函数调用时,程序会从主调函数跳转到被调函数的第一条语句,然后按顺序依次执行被调函数中的语句。被调函数执行完后会返回到主调函数中,并且是返回主调函数语句的后一条语句,也就是"从哪里离开,就回到哪里"。图 5-12 中(1)~(6)就与执行过程中函数调用与返回的过程一一对应。

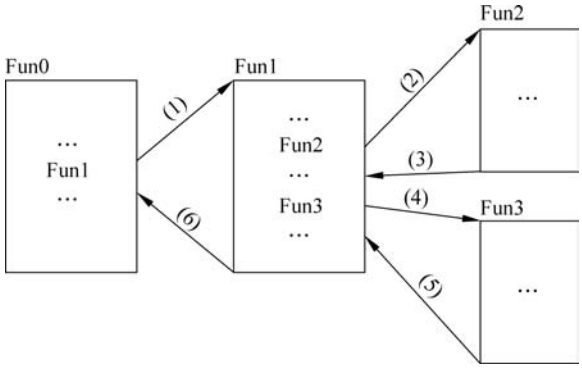


图 5-12 函数调用过程示意图

当被调函数执行完成后返回时,计算机是怎么知道"从哪里离开,回到哪里"呢?为了解决这个问题,主调函数事实上会在离开前先将现在的程序地址保存为"返回地址"。在一个系统的数据结构上,当被调函数完成并返回时,会先从该数据结构中调取"返回地址",然后再返回到指定地址。由图 5-12 可知函数调用的特点是:越晚被调用的函数,越早返回,即"后进先出",因此我们采用"栈"这个数据结构来保存返回地址。

所谓"栈",就是一个允许在同一端进行插入和删除操作的特殊线性表,在 Python 中可以用列表实现。允许进行插入和删除操作的一端称为栈顶(top),另一端为栈底(bottom)。插入数据称为进栈(Push),删除则称为出栈(Pop)。图 5-13 所示为函数 Fun1 调用函数 Fun2 以及调用 Fun3 时栈中返回地址的变化过程。

函数调用过程不仅仅是保存返回地址这么简单,事实上,函数的局部变量也是与返回地址绑定在一起用栈来管理的。因为局部变量只有在所在函数内才有意义,一旦所在函数被调用并跳到另一个函数中,如果不将局部变量保存下来,该局部变量就会失效,等被调函数执行结束并返回时,即使有返回地址也不能还原主调函数的信息,所以局部变量如同返回地

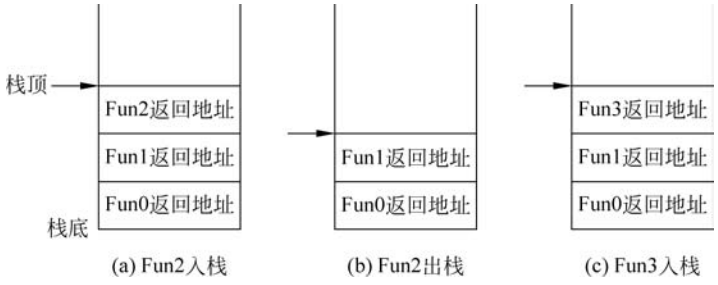


图 5-13　返回地址的存储

址一般也是需要保存在栈中的，如图 5-14 所示。

图 5-14　栈内存储信息内容

综合前面所讲到的知识，可以总结出一个函数调用过程分为两部分：

（1）在函数被调用时，要为该函数开辟栈空间，将数据（包括参数、返回值、局部变量和函数执行过程中产生的临时变量等）以及控制信息（返回地址等）压入栈中；

（2）在函数返回时，需要将这些信息从栈中弹出，并释放这些空间。

这些工作都是由栈来完成的，而创建栈空间、进栈和出栈都是需要一定的开销的，这些开销就是函数调用的开销。整体而言，这个开销是可以忍受的，大家不要为了节省这个开销，就不写函数了，因为函数可以让程序清楚、易懂、容易找错、容易维护、容易修改。

5.4.2　参数传递过程中的开销

调用函数时一般会有参数的传递。主调函数在调用被调函数时会将某一变量（或几个变量）以参数的形式传递过去，但是请注意，在 Python 中，参数传递时并不是将变量的具体值传递过去，而是将变量的地址传递到被调函数中，无论该变量是可变量（如列表等），还是不可变量（如字符串、整数等）。这个特性和其他编程语言是不同的。

在 Python 中要注意，由于可变参数和不可变参数具有的不同性质，导致参数在被调函数中改变时会有不同的效果。如果在被调函数中不可变参数被改变，会自动创建新的存储空间保存新的值，那么原来地址中保存的参数值不会被改变。如果在被调函数中可变参数被改变，并不会创建新的空间，而是在原地址上对参数值进行修改。为了避免可变参数在被调函数中改变，并导致主调函数中的变量发生变化，我们在被调函数中一般先将可变参数的值整体复制到一个新的局部变量，再进行改变。

上述参数传递过程中的地址传递和空间创建并不会造成程序过多的开销，但是当传递的参数为列表的形式，传递时又对列表进行**分片操作**，就会在传参之前先创建空间，复制及保存列表分片之后的结果，然后再将新的地址传递给被调函数。假如分片的长度长，则分片的开销是很明显的，这在前面的章节已经有详细的介绍。对于递归函数，如果调用时采用的

是对列表进行分片之后再进行传参，就要想想是否可以避免这个开销，因为递归函数是一个自己调用自己的过程，也就是说，调用了几次递归函数就要分片几次，分片开销积累下来对程序的效率会造成一定的影响。所以，在对函数进行传参时要少用列表的分片形式，尽可能地用传递索引的形式来减少开销。

下面以 5.1.2 节介绍的在有序序列中使用二分查找元素的位置为例，比较在递归中分别采用列表分片形式和传递索引形式的时间差异。由于这两种传参形式已经在前面编程实现过，其中<程序：递归实现二分查找>采用的是列表分片的方式进行参数传递，<程序：嵌套递归实现二分查找 2 >采用的是传递索引的方式，在接下来对这两种方式进行比较时就直接调用程序，而不再次对程序进行编写。时间开销对比程序如<程序：递归实现二分查找时间开销对比>所示。

```
#<程序：递归实现二分查找时间开销对比>
Import time
def test_time(k):
    print("##########列表长度%d##########"%(k))
    L=[i for i in range(k)]
    start=time.clock()
    print(BinSearch(L,k-1))                    #调用传递列表分片参数的函数
    elapsed=time.clock()-start
    print("使用传递分片形式花时间：",elapsed)
    start=time.clock()
    print(binary_r1_search(L,k-1))             #调用传递索引参数的函数
    elapsed=time.clock()-start
    print("使用传递索引形式花时间：",elapsed)
test_time(10000000)                            #主函数
test_time(20000000)
```

在运行上述程序之后，会发现使用列表分片的开销较大，导致传递参数过程中开销变大。所以需尽量少在递归调用中使用列表分片进行参数传递，而是用起始索引和终止索引来代替列表分片。上述程序的执行结果如下：

```
##########列表长度 10000000##########
(True, 9999999)
使用传递分片形式花时间：0.13213618599999988
9999999
使用传递索引形式花时间：0.008603355000000201
##########列表长度 20000000##########
(True, 19999999)
使用传递分片形式花时间：0.2511941200000001
19999999
使用传递索引形式花时间：0.009255024999999861
```

5.4.3 重复计算的开销

除了以上两种开销，递归调用过程中还可能会有许多重复计算，这种重复计算是可以用 8.4 节介绍的动态规划方式来避免的。

举例而言，递归关系式如斐波那契关系 f(n)＝f(n－1)＋f(n－2)这种形式，假如直接写程序就是直接写成递归函数。当计算 f(n)时会计算 f(n－1)和 f(n－2)，而计算 f(n－1)时又会计算 f(n－2)和 f(n－3)，计算 f(n－2)时也会计算 f(n－3)和 f(n－4)，以此类推，会产生大量重复计算。所以，在这种递归关系式的情形下，不应该直接用递归函数来编程，正确的编程方式是从已知的 f(0)与 f(1)推出 f(2)，f(3)……一直推到 f(n)。这种基于递归关系式从小到大推导的方式叫作动态规划，在 8.4 节会有详细的介绍。

动态规划的重点在于递归关系式的推导以及终止条件的确定，这与递归思维一致，只是在编程的时候不直接写成递归函数，而是采用循环方式从小到大求得结果，其中主要用列表存储中间已算出的值。

综上所述，为了减少递归函数造成的开销以及减少递归的深度，可以通过以下两种方式来解决：

(1) 尽可能使用二分法思想，并且参数传递时用传递索引的方式代替列表分片。

(2) 如果直接将递归关系式编写成递归函数会产生大量重复计算，那么请利用动态规划算法从小往大推导求解。感兴趣的同学可以提前自学 8.4 节。

5.5　用递归思维解决线性方程组问题

第 1 章曾经提到鸡兔同笼问题，该问题其实就是两个变量的线性方程组求解问题。当然，还有 3 个变量、4 个变量的线性方程求解问题。那么对于这种线性方程求解问题有没有一个通用的求解方法呢？这个例子可以让我们体会到递归思维的简洁和优美。下面分析一下这个问题。

先来看一个有 3 个变量的方程组 S：

$$S:\begin{cases}2x_1+3x_2+x_3=4 & ①\\ 4x_1+2x_2+3x_3=17 & ②\\ 7x_1+x_2-x_3=1 & ③\end{cases}$$

【分析过程】　对于给定的方程组，可以用矩阵相乘的形式表示：系数为一个矩阵 A，变量为一个矩阵 X，等号右侧的值为一个矩阵 b。实际上就是 AX＝b，对于方程组 S 可以表示为：A＝[[2,3,1],[4,2,3],[7,1,－1]]，X＝[x_1,x_2,x_3](实际编程中并不需要 X，因为从 A 或者 b 中都可以得知有多少个未知数)，b＝[4,17,1]，如图 5-15 所示。

$$\underset{A}{\begin{bmatrix}2 & 3 & 1\\ 4 & 2 & 3\\ 7 & 1 & -1\end{bmatrix}}\cdot\underset{X}{\begin{bmatrix}x_1\\ x_2\\ x_3\end{bmatrix}}=\underset{b}{\begin{bmatrix}4\\ 17\\ 1\end{bmatrix}}$$

图 5-15　方程组 S 的表示方式

以上面的方程组 S 为例，我们来看以下具体的执行过程。

(1) 对于输入的 A 和 b，将三维的式子降为二维(通过消除 x_1)：首先将②和①中 x_1 的系数统一后两个式子再相减，②×2－①×4(注意：在代码中只需要利用矩阵 A 和 b 就可以进行计算，为什么以互相乘以对方的系数这种方式进行计算请同学们先思考一下，我们将在后面进行讲解)，得到一个新的式子④：$-8x_2+2x_3=18$。然后将③和①中 x_1 的系数统一后两个式子再相减，③×2－①×7，得到一个新的式子⑤：$-19x_2-9x_3=-26$。这样就成功地将三维的方程式 S 降为二维的方程式 S1，如图 5-16 所示。

$$\underset{A}{\begin{bmatrix}2&3&1\\4&2&3\\7&1&-1\end{bmatrix}}\cdot\underset{X}{\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}}=\underset{b}{\begin{bmatrix}4\\17\\1\end{bmatrix}}\begin{matrix}①\\②\\③\end{matrix}\xrightarrow{\text{降维}}\begin{matrix}\\②\times2-①\times4\\③\times2-①\times7\end{matrix}\quad\underset{A}{\begin{bmatrix}-8&2\\-19&-9\end{bmatrix}}\cdot\underset{X}{\begin{bmatrix}x_2\\x_3\end{bmatrix}}=\underset{b}{\begin{bmatrix}18\\-26\end{bmatrix}}\begin{matrix}④\\⑤\end{matrix}$$

方程式S　　方程式S1

图 5-16　三维降到二维示意图

(2) 同样输入新的 A 和 b，递归处理该问题，将二维的式子降为一维(通过消除 x_2)，具体过程如图 5-17 所示，得到一维的式子 S2。

$$\underset{A}{\begin{bmatrix}-8&2\\-19&-9\end{bmatrix}}\cdot\underset{X}{\begin{bmatrix}x_2\\x_3\end{bmatrix}}=\underset{b}{\begin{bmatrix}18\\-26\end{bmatrix}}\begin{matrix}④\\⑤\end{matrix}\xrightarrow{\text{降维}}\begin{matrix}\\5\times(-8)-4\times(-19)\end{matrix}\quad\underset{A}{\begin{bmatrix}110\end{bmatrix}}\cdot\underset{X}{\begin{bmatrix}x_3\end{bmatrix}}=\underset{b}{\begin{bmatrix}550\end{bmatrix}}⑥$$

方程式S1　　方程式S2

图 5-17　二维降到一维示意图

(3) 输入新的 A 和 b，递归处理该问题。由于现在 b 中只有一个元素了，也就意味着只有一个未知数，可以求解了。所以根据式子⑥$110x_3=550$ 可以求得 $x_3=5$。将其添加至结果集中，则结果集为[5]，同时将该结果返回给上一层的方程式 S1。

(4) 利用方程式 S1 中的式子④，将 $x_3=5$ 代入，就可以求得 x_2。$x_2=-1$，并将其添加至结果集中，则结果集为[－1,5]，同时将该结果返回给上一层方程式 S。

(5) 利用方程式 S 中的式子①，将 $x_2=-1$、$x_3=5$ 代入，可以求得 x_1。$x_1=1$，并将其添加至结果集中，结果集为[1,－1,5]。至此，便求得了方程组 S 的解为[1,－1,5]。

【解题思路】 经过上面的描述，相信大家已经看出来，这同样是一个递归的思想：对于一个有 n 个变量的方程组，假设已经求出了后 n－1 个变量，可以很容易地求出第一个变量，只不过在每次求解变量时要检查一下该变量的系数是否为 0。终止条件为：直到递归到求第一个变量，如果该变量系数不为零，则返回该变量解；否则，返回错误信息。代码如<程序：线性方程组求解>所示。

```
#<程序：线性方程组求解>
#解决 Ax = b.A 是个 n·n 的矩阵,b 是个 n 维的向量,x 是 n 个未知数所组成的向量
def solve_linear(A0,b0):
    A = A0[:]; b = b0[:]            #将参数先复制下来,以避免修改原有数据,这是一种好习惯!
    if len(b) == 1:
        if A[0][0] != 0: return True, [b[0]/A[0][0]]
        else:
```

```
            print("Error: Division by ZERO. Your input is wrong. NO solution.")
            return False, [9999999999]
    for i in range(len(b)):                    #判断 A[0][0]是否为 0
        if A[i][0] != 0: base = i;break
    else: print("ERROR: All zeros"); return False, []
    if base != 0:                              #确认 A[0][0],第一个等式的第一项系数不是 0
        A[0],A[base] = A[base],A[0]
        b[0],b[base] = b[base],b[0]
    c1 = A[0][0]
    Anew = []; bnew = []                       #会减少一个维度
    for i in range(1,len(b)):
        if A[i][0]!= 0:
            a = []; c2 = A[i][0]
            for j in range(1,len(b)):
                t = A[i][j] * c1 - c2 * A[0][j]          #尽量少用除法!
                a.append(t)
            Anew.append(a)
            bnew.append(b[i] * c1 - c2 * b[0])
        else:                                  #如果 A[i][0] == 0,则不需要做计算,直接就可以降维
            Anew.append(A[i][1:len(b)])
            bnew.append(b[i])
    flag, Ans1 = solve_linear(Anew,bnew)                 #递归
    if flag == False: return False, Ans1
    t = 0
    for i in range(len(Ans1)):                 #将 A[0]等式代入 Ans1,求出 x 值
        t = t + A[0][i + 1] * Ans1[i]
    x = (b[0] - t)/A[0][0]                     #最后用了除法
    Answer = [x] + Ans1
    return flag, Answer
#以下为主函数
A = [[10,1,5],[0, - 1,9],[4,1,1]]
b = [7, - 11,5]
print("A = ",A, "\nb = ",b)
flag, X = solve_linear(A,b)
if flag:
    print("The solution is",X)
print("\n","###   ##" * 10)
```

通过使用递归实现求解线性方程组的程序，有 3 点需注意的地方：

(1) 在编程中尽量少用除法。由于计算机中有精准度的问题，所以在降维的时候应尽量少用除法。比如式子①$10x_1+x_2=9$ 和式子②$3x_1+5x_2=8$。降维可以用很多种计算方式，可以采用②×(10/3)－①，但是由于计算机在计算除法的时候会将结果存为实数，那么 10/3＝3.33333…这样后面的计算就会显得很麻烦。因此，我们聪明地采取了另一种方式，也就是通过②×10－①×3 的方式来降维，方便后面的计算。

(2) 可以看到，solve_linear()函数中有很多地方在处理第一个方程式中的第一项(即 A[0][0])是不是 0 这一问题。因为在降维的时候每次都是用其他方程式的第一个系数与第一个方程式的第一个系数来做化简，所以需要保证 A[0][0]!＝0。那么如果 A[0][0]＝0，

就可以将当前的第一个方程式与第一个系数不为 0 的方程式做交换，从而保证交换后整个方程式中 A[0][0]＝0，如图 5-18 所示。

$$\begin{cases} -x_2+9x_3=-11 & ① \\ 10x_1+x_2+5x_3=7 & ② \\ 4x_1+x_2+x_3=5 & ③ \end{cases} \xrightarrow{\text{交换方程}} \begin{cases} 10x_1+x_2+5x_3=7 & ① \\ -x_2+9x_3=-11 & ② \\ 4x_1+x_2+x_3=5 & ③ \end{cases}$$

图 5-18　交换方程示意图

当化简到只有一维的时候，仍然要检查 A[0][0]是否为 0，因为此时 A[0][0]要作为被除数，若为 0，则说明该方程无解。

(3) 有些方程是无解的，比如式子①$2x_1+3x_2=4$ 和式子②$4x_1+6x_2=0$。这明显是两条平行线，在降维的时候②×2－①×4 得到结果为 0＝－16，即 $x_2=(-16)/0$，很显然，无解。

5.6 用各种编程方式解决排列问题

接下来要展现如何用多种编程思维方式来解决一个常用的问题——排列组合问题。希望同学们能深入探讨和理解这些不同的编程思维，同时也希望各位能挑战自己，试着设计不同的程序来解此问题。学好此节后，编程的境界就能上一个新台阶。

我们平时在电视中经常会看到“开彩票”节目，开彩票就是在一个箱子里放有 n 个球，每个球上都标有数字，先后从中抽取出 k 个球，所得到的这个数字序列就是开彩票的结果，若买家买的彩票序列与该结果完全一样，买家才算中了大奖。比如现在箱子里有[12,3,6]3 个球，要从中抽取 2 个球，假设先抽取出了 3 号球，然后又抽取出了 6 号球，则得到的开彩票结果为[3,6]，如果某个买家买的彩票为[3,6]，就要恭喜他中了大奖。请注意，若有人买的彩票为[6,3]，由于彩票的顺序不对，我们要很遗憾地告诉他，他没有中奖。

像这种从 n 个数中按顺序抽取 k 个数，从而得到新的序列的问题称为排列问题。那么在刚才的例子中，我们只列举出了从 3 个球中按顺序抽取两个球可能得到的两种结果，分别是[3,6]和[6,3]，实际上，还有很多种可能的结果，如[12，3]、[12，6]、[3，12]…那么，像这种对于一个有 n 个元素的序列，选取其中的 k 个元素进行排列，并将所有可能得到的结果以序列的形式输出，到底有多少种可能呢？数学上将该排列问题表示为求解 A_n^k，其中 n 为所有元素的个数，k 为进行排序的元素个数。例如，L＝[12,3,6]，则 A_3^2 的所有可能结果为[[12，3]，[12，6]，[3，12]，[3，6]，[6，12]，[6，3]]共有 6 种。当然也有 n＝k 的情况，这种情况称为全排列，在这个例子中，A_3^3 的所有可能结果为[[12，3，6]，[12，6，3]，[3，12，6]，[3，6，12]，[6，12，3]，[6，3，12]]。如何编写程序来求解出排列问题？本节将分成两大部分来介绍，首先介绍全排列问题的解法(即 k＝n)，然后介绍通用的排列问题的解法(即 k≤n)。

5.6.1 全排列问题

对于全排列问题，可以表示为 A_n^n。例如，L=[12，3，5，3]，则全排列的结果为[[12，3，5，3]，[12，5，3，3]，[3，12，5，3]，[3，5，12，3]，[5，3，12，3]，[5，12，3，3]]。再如，S = [d,d,a]，则全排列的结果为[[d,d,a]，[d,a,d]，[a,d,d]]。我们知道，对 n 个数全排列的个数为 n 的阶乘。设 f(n)为对 n 个不同数的排列个数，则递归关系为 f(n)=nf(n−1)，f(1)=1。

本节将讲解两种方法来解决全排列问题。这两种都是递归的思维。

1. 全排列问题解法一

怎么样实现一个序列的全排列呢？先定义 P(L)表示对长度为 n 的序列 L 所对应的全排列的所有解，P(L)是个双层列表，每个元素代表一种排列。由前面的递归知识可以想到，对于一个长度为 n 的序列 L，可以假设已经求得后面 n−1 位数的全排列 P(L[1:n])，然后再把第一位元素 L[0]插入不同的位置上。先插在所有元素的最开始，再依次往后插在两项元素中间，共有 n 个位置可以插入。终止条件为：当 n=1 时，返回这个值所组成的列表。注意，程序最终返回的全排列结果应该是一个双层的列表。先来看具体的实现，代码如<程序：全排列解法一>所示。

```
#<程序：全排列解法一>
def permutation_all_1(L):
    if len(L) <= 1: return [L]
    T = permutation_all_1(L[1:len(L)])
    R = []
    for i in range(0, len(L)):
        for t in T:
            x = t[0:i] + [L[0]] + t[i:len(L)]
            if x not in R:          #去掉因 list 中有重复的值而产生的一样的排列方式
                                    #假如 L 中没有重复元素，这个 if 可以去掉
              R.append(x)
    return(R)
```

上述代码的思想是这样的：利用递归的方式，每次将序列切分成两部分，第 0 项为一部分，剩下的为另一部分。每次处理剩下的那一部分，把 L[0]插入剩下的这部分序列的不同位置中去，最终得到结果。大家可以验证全排列的个数确实是遵循 f(n)=nf(n−1)的关系式的。

我们通过图解的方式具体解释一下是怎样一步步得到最终结果的。假设 L = [8，3，5]，则具体的执行流程如图 5-19 所示。

当然，这里有一点需要注意，对于像 S=['d'，'d'，'a']这种列表中有重复值的情况，需要做一些处理，不然会有很多重复的结果。去重的处理方式有很多种，在上述代码中，选择使用“if x not in R:”这一方式来去掉重复的解。如果没有这句代码，会得到如下的结果：[['d'，'d'，'a']，['d'，'a'，'d']，['d'，'d'，'a']，['a'，'d'，'d']，['d'，'a'，'d']，['a'，

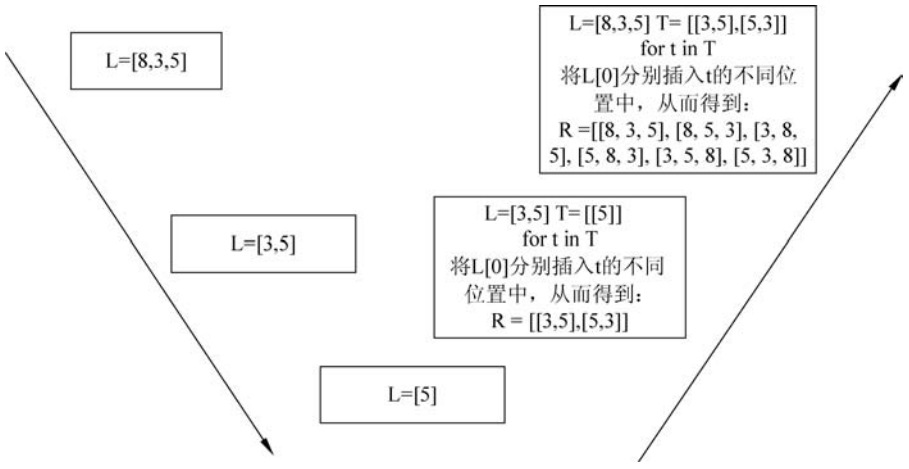


图 5-19　全排列递归示意图

'd', 'd']],可以看到有很多重复的解。而有了这句代码,就可以得到正确的结果:[['d','d','a'], ['d', 'a', 'd'], ['a', 'd', 'd']]。

2. 全排列问题解法二

全排列解法一是将递归放在 for 循环之外,有了解法一的思路,我们还可以做进一步的思考。如果输入是一个从小到大有序的序列,比如 L=[0,1,2],是否可以再设计一个解法,使得输出全排列的结果也是一个从小到大排列好的列表?

比如对上述的列表 L 进行全排序,得到的从小到大排好序的列表结果 R 为:R=[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]。即结果 R 中,先按照每个元素的第 0 项排序(即 L[0][0]、L[1][0]、L[2][0]、L[3][0]、L[4][0]、L[5][0]是有序的),若第 0 项相同,则再按照每个元素的第 1 项排序,若第 1 项相同,再按照每个元素的第 2 项,以此类推,将 R 中的每个元素按照从小到大的顺序排列好。这种有序的结果在某些情况下是有用的,比如在程序大赛中,要求返回的结果是有序的,如果利用全排列解法一,还需要再对结果进行排序,而如果参赛者可以设计出新的解法,直接得到有序的结果,那么必将节省很多时间。

相比全排列解法一,如何能够保证新设计的解法得到的解是有序的?我们设计出一种将 for 循环放在递归之内的算法,其实大致思路是类似的:每次把 L[i]抽取出来,剩余的元素算作另一部分,假设剩余元素的全排列经过递归已经求得,然后在 L[i]后面添加这部分的解形成完整的一个排列解。终止条件为:当列表中的元素个数小于或等于 1 时,返回原列表。

注意,与解法一不同的是,这里一定要将 L[i]放在最前面,而不是将其插入另一部分得到的解中,只有这样才能够保证解的顺序。比如对于有序的序列 L=[0,1,2,3]来说,就可以这样做:0+递归(1,2,3);1+递归(0,2,3);2+递归(0,1,3);3+递归(0,1,2)。由于每次抽取的 L[i]都是从小到大的,从而保证了整体全排列的解是按照①0…;②1…;③2…;④3…这样 4 部分有序排列的,而对于提取出 L[i]后剩下的部分继续做递归时由于规则是一样的,从而保证了①、②、③、④每一部分的内部同样是有序的,所以最终的解必定是有序

的。有了上述的思路，就可以写出如下代码：

```
#<程序：全排列解法二>
def permutation_all_2(L):
    if len(L) <= 1: return [L]
    R = []
    for i in range(len(L)):
        L1 = L[0:i] + L[i+1:len(L)]          #把 L[i]抽出来，其他顺序不变
        L1_new = permutation_all_2(L1)       #对剩下的部分递归做全排列
        for e in L1_new:
            comb = [L[i]] + e                #注意，是 L[i]在前
            if not comb in R:                #用于去除重复的解，如 L=[0,1,1,1]会有重复解
                R.append([L[i]] + e)
    return(R)
```

<程序：全排列解法二>中利用 for 循环每次提取出 L[i]，把剩下的元素重新组合成一个新的待处理列表 L1，对 L1 以同样的方式递归地进行全排列得到所有的排序结果 L1_new，那么最后需要做的就是将 L[i]分别与 L1_new 中的每个元素做组合，得到最终的解并加入结果集中。当然，编程的时候同样要注意 L 中有重复元素的情况，记得去除重复的解。

至此，代码就编写结束了。我们可以来对比验证一下，若 L=[0,1,2]，则利用<程序：全排列解法二>得到的结果为 R=[[0, 1, 2], [0, 2, 1], [1, 0, 2], [1, 2, 0], [2, 0, 1], [2, 1, 0]]，而利用<程序：全排列解法一>得到的结果为 R=[[0, 1, 2], [0, 2, 1], [1, 0, 2], [2, 0, 1], [1, 2, 0], [2, 1, 0]]。可以看到，对于输入是有序的序列，全排列之后，解法二可以得到一个从小到大的有序解。

5.6.2 通用排列问题

5.6.1 节的代码只能够求解 k=n 的情况，那么当 k<n 的时候应该怎么做？此时就需要重新思考该问题，设计出通用的算法，使得 k<n 或者 k=n 的时候都可以使用。这里将给出两种方法解决 A_n^k 问题：其中解法一为非递归的求解方式，利用 for 循环，通过逐层递增的方式最终求得 k 个数的所有排列；解法二为通过每次将问题分解成两部分，最后再组合在一起的方式求得所有的解。

1. 通用排列问题解法一（非递归方式求解）

如果不用递归的方式，是否可以求解这一问题呢？这里给出一种利用 for 循环逐层递增的方法来求解排列问题。整体思路为：若想求 L 中 k 个数排列的集合，for 循环则循环 k 次，从 1 个数开始一直扩充到 k 个数截止。具体如下：先给第一个位置取数，得到选取一个数的所有可能结果；然后在第一个数已经取好的基础上，再添加第二个数，注意第二个数和前面已有的数不能重复；再在第一、二个数都取好的基础上，添加第三个数，第三个数和前面已有的数不能重复；一直到添加完所有 k 个数，就得到了 n 个数中取 k 个数的排列的全部结果。下面以 L=[1,2,3,4]，k=3 为例，看一下这个方法的具体流程。

(1) 第一次循环：初始化结果集为 R=[[]]。先取第一个数，用 L=[1,2,3,4]对 R 进

行第一次扩充操作，也就是用 L 中的数给结果集中每个子列表添加一个数，得到新的子列表，即得到新的结果集 R=[[1][2][3][4]]。也就是说，第一次循环得到了第一位数字的所有可能的集合，如图 5-20 所示。

(2) 第二次循环：再为上一步产生的 R 中每个元素添加第二个数，对 R 中的每个元素来说，添加第二个数时就要去除自身已有的数。所以在这一步中，去除第一步已经选的 1 个数，可以取的数就只有 n－1 个。例如，对 R 中的[1]来说，可以再取的数字只剩 2、3、4；对[2]来说，可以再取的数字只剩 1、3、4……那么，[1]在扩充第二个数之后，生成了 3 种子结果：[1,2],[1,3],[1,4]；[2]在添加第二个数之后，生成了 3 种子结果：[2,1][2,3][2,4]；同理[3]、[4]也要进行同样的操作。经过这一步，便得到了所有两个数的排列可能，R=[[1, 2], [1, 3], [1, 4], [2, 1], [2, 3], [2, 4], [3, 1], [3, 2], [3, 4], [4, 1], [4, 2], [4, 3]]，如图 5-21 所示。

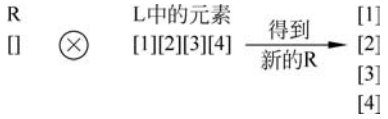


图 5-20　第一次循环示意图

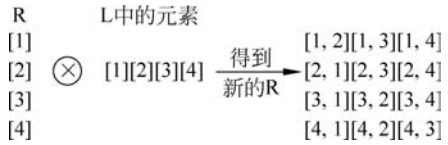


图 5-21　第二次循环示意图

(3) 第三次循环：取第三个数，对上一步的结果 R 中的每个元素来说，可以取的数只有 n－2 个了，因为每个元素中已经有了两个数了。例如，对 R 中的[1,2]来说，可以取的数字只剩 3、4；对[1,3]来说，可以取的数字只剩 2、4……那么，[1,2]在取第三个数之后，产生了两个子结果：[1,2,3] [1,2,4]；[1,3]在取第三个数之后，产生了两个子结果：[1,3,2] [1,3,4]；同理，R 中的所有元素都执行上述操作，最终每个元素都扩充到 3 个数字，即得到了所有 3 个数的排列可能，R= [[1, 2, 3], [1, 2, 4], [1, 3, 2], [1, 3, 4], [1, 4, 2], [1, 4, 3], [2, 1, 3], [2, 1, 4], [2, 3, 1], [2, 3, 4], [2, 4, 1], [2, 4, 3], [3, 1, 2], [3, 1, 4], [3, 2, 1], [3, 2, 4], [3, 4, 1], [3, 4, 2], [4, 1, 2], [4, 1, 3], [4, 2, 1], [4, 2, 3], [4, 3, 1], [4, 3, 2]]。此时便求得了 A_4^3 的解，循环结束，如图 5-22 所示。

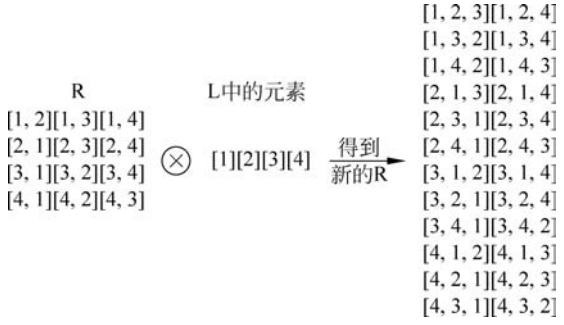


图 5-22　第三次循环示意图

代码如<程序：permutation_1 >所示。这个程序并不是很有效率，因为即使 L 中没有重复元素，程序中的检查“if f not in e:”也会产生许多的开销。这个程序其实是人们习惯使用的方式，不是很好，也较难扩展到当 L 有重复元素的情况。

```
#<程序：permutation_1>
def permutation_1(L,k):
    def product2(R,L):
                                    #辅助函数 product2,给当前结果集 R 中的每个元素扩充一位
        R1 = []
        if R == [[]]:               #如果 R == [[]],则需要给 R 扩充到第一位
            for e in L: R1.append([e])
        else:                       #如果 R 中已经有了元素,则直接进行扩充
            for e in R:
                for f in L:
                    if f not in e: R1.append(e + [f])
        return R1
    if k > len(L) or k < 0:
        print('错误的输入,k 应当小于 L 中的元素个数,且 k 为非负数!')
        return []                   #当 k < 0 或 k > len(L)时,排列数为 0,返回的是空列表[]
    if k == 0: return [[]]          #注意,当 k = 0 这一特殊情况时应返回[[]]
    R = [[]]
    for i in range (k): R = product2(R,L)
    return R
```

练习题 5.6.1 请用递归搜索的方式解决通用排列问题。这是后面将要介绍的深度递归搜索的思想(6.2 节的菜鸡狼过河,8.2 节的图深度优先搜索)。

此问题的递归搜索的思维如下：初始化一个长度为 k 的列表 T,先尝试填写 T[0],再填写 T[1],再填写 T[2],……,T 填写满了之后将其添加到结果集中,再回退到上一状态,尝试填写其他元素,直到搜索完所有的可能。

【解题思路】 先通过一个具体的例子来理解递归搜索的过程：L=[5,3,9],k=3,则初始化 T 使得 T 有 k 个元素(位置)。设 T=[1,-1,-1](或任意值),返回结果用 R 表示。

(1) T[0]=5,此时 T=[5,-1,-1]。T[1]在剩下可选的元素中选择一个,即 T[1]=3,此时 T=[5,3,-1]；则 T[2]还可以在剩下的元素中选 9,即 T[2]=9,此时 T=[5,3,9]且 T 满了,添加到 R 中。R=[[5,3,9]]。

(2) 返回到 T[0]=5,T[1]=3,即 T=[5,3,-1]的状态。T[2]还没有填写,但是发现 9 已经填写过,再也没有可以填写的元素了,所以需要继续回退。

(3) 回退到再上一层,即 T[0]=5,T=[5,-1,-1]的状态。步骤(1)中 T[1]已经填写过 3,所以这次 T[1]=9,那么 T=[5,9,-1]。还剩下 T[2],则 T[2]从剩下的数中选取一个,T[2]=3,此时 T=[5,9,3]且满了,添加到 R 中。R=[[5,3,9],[5,9,3]]。

(4) 再返回到 T[0]=5,T[1]=9,T=[5,9,-1]的状态。此时再次尝试填写 T[2],发现已经没有可以再尝试的数字了,所以继续回退。

(5) 回退到 T[0]=5,T=[5,-1,-1]的状态。此时发现确定了 T[0]=5 之后,已经尝试搜索了 T[1]和 T[2]的所有可能。那么就需要在 T[0]填写其他的数再进行尝试。这次 T[0]=3,T=[3,-1,-1]。再按照上面的思想,去搜索 T[1]的所有可能,然后搜索 T[2]的所有可能。此时得到 R=[[5,3,9],[5,9,3],[3,5,9],[3,9,5]]。

……

最终得到结果 R=[[5,3,9],[5,9,3],[3,5,9],[3,9,5],[9,5,3],[9,3,5]]。

【答案】 代码如<程序：permutation_dfs>所示。其中 dfs 代表深度优先搜索(Depth First Search)的意思。

```
#<程序：permutation_dfs>
def permutation_dfs(L,k):
    R = []
    T = [-1 for i in range(k)]          #这里用-1来进行初始化
    def P(L,num_pick,m):
        if m > num_pick-1: R.append(T[:])
        else:
            for i in L:
                if i not in T[0:m]:     #判断当前T中是否已经有了i这一元素
                    T[m] = i;P(L,num_pick,m+1)
    P(L,k,0)
    return R
```

<程序：permutation_1>和<程序：permutation_dfs>都只适用于 L 中没有重复元素的情况。如果遇到 L 中有重复元素的特殊情况，将不再适用。比如 L=[12,3,3,3]或 L=[6,6,4,7,7,6]都无法得到正确的解。如果想在这两个代码的基础上加以修改来处理 L 中有重复元素的情况会比较复杂，此处不做过多讲解(同学们可以自己仔细思考一下为什么这两种解法不好做修改，并动手尝试一下)。我们将给出排列问题的其他解法(见通用排列问题解法二)，同时也能够处理 L 中有重复元素的特殊情况。

2. 通用排列问题解法二(特殊二分方式求解)

我们可以换一种递归的思路。这种思维方式非常有用，就是将解集合分成没有交集的两部分。列表 L 中选择 k 个元素做排列的时候可以将整个解 T 分成两部分：T1 部分表示一定选取 L[0]的那些排列；T2 部分表示一定不选 L[0]的那些排列，如图 5-23 所示。那么 T1 和 T2 这两部分又分别有多少种排列呢？我们来分析一下。对于 T1 部分，既然选定 L[0]，则还需在剩下的元素中选取 k−1 个进行排列，求得这部分的解 $S=A_{n-1}^{k-1}$(先假定 A_{n-1}^{k-1} 的解已经求出)，那么对于 S 中的每个解 S_i，再把 L[0]分别插入 S_i 的不同位置，最终得到 T1 部分的所有解。对于 T2 部分，既然不选定 L[0]，那么还需在剩下的部分中选取 k 个元素，即 A_{n-1}^{k}。最后把这两部分的解加在一起就是全部的解了，即 $A_n^k=A_{n-1}^{k-1}+A_{n-1}^{k}$。其实上述的思想就是递归的一个思想，把一个大问题分解成小问题，最后再进行组合。

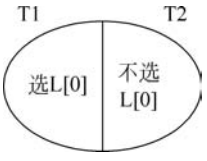


图 5-23 解法二思路示意图

下面通过一个具体的例子来对上述思想进行讲解。假设 L=[1,2,3]，k=2，求所有的解。要求 A_3^2 的解(用 T 表示)，先分成两部分 T1 和 T2，则最后的解为 T=T1+T2。

对于 T1 部分，已经选定 L[0]，还要在 L1=[2,3]中再选一个，求 A_2^1。假设剩余部分元素的全排列结果 A_2^1 已经求得，则需要将 L[0]与每个结果进行排列，进而得到 T1。而 A_2^1 的解(用 S 表示)是以同样的递归方式求出的，即将 S 分成 S1 和 S2 两部分，求得解为 S=S1+S2=[[2],[3]]。即将 L[0]=1 插入[2]中不同位置得到[[2,1],[1,2]]；L[0]=1 插入[3]中不同位置得到[[3,1], [1,3]]。可以得到最终 T1 的解：T1 = [[2,1],[1,2],

[3,1],[1,3]]。至此,得到了所有包含 L[0]的排列的解。

对于 T2 部分,不选定 L[0],则还要在 L1=[2,3]中再选两个,求 A_2^2。使用上述类似的递归方法,将求 T2 的问题又转化为必须选 L1[0]和不选 L1[0]。最终得到 T2 部分的解为 T2=[[2,3],[3,2]]。

通过上面的执行过程,得到了 T1 和 T2 部分的解,则最终的解 T = T1+T2 =[[1,2],[1,3],[2,1],[3,1],[2,3],[3,2]]。具体的代码见<程序:permutation_2>。

```
#<程序: permutation_2>
def permutation_2(L,k):
    if k == 0: return [[]]
    if len(L)< k: return [[]]
    T1 = [];T2 = []
    if len(L) - 1 > = k:
        T2 = permutation_2(L[1:len(L)], k)          #不选第一个元素
    if len(L)> = k:
        T1 = permutation_2(L[1:len(L)],k - 1)       #必须选第一个元素
    R = []
    for i in range(0, k):                           #将 L[0]插入 T1 中
        for t in T1:
            x = t[0:i] + [L[0]] + t[i:k]
            if x not in R: R.append(x)              #去除会有重复的解
      #下面这部分即 R = T1 + T2,但需要去除有重复的元素
        for e in T2:
            if e not in R: R.append(e)
    return(R)
```

前面提到解法一没有解决 L 中有重复元素的情况;在解法二中,分别在 T1 和 T2 部分做了处理,将重复的解去除,最终得到没有重复的解。

5.7 用各种编程方式解决组合问题

组合问题其实与排列问题类似,但是有一个重要的差别在于组合问题的解与顺序无关。生活中也有很多组合的例子,比如一个班级中有 50 人,现需要挑选 3 个人去参加学校组织的活动,那么老师先选择兰兰,再选择阿珍,最后选择小红(即[兰兰,阿珍,小红])和先选择小红,再选择阿珍,最后选择兰兰(即[小红,阿珍,兰兰])的结果其实是一样的。可以看出组合问题与顺序无关,重点在选择哪些元素上。

我们给出组合问题的定义:对于一个有 n 个元素的序列,选取其中的 k 个数进行组合,并将所有可能的组合结果以列表的形式输出(即求 C_n^k,其中 n 为所有元素的个数,k 为进行组合的元素个数)。例如:L = [1,2,3,4],k = 3(即 C_4^3)所有可能的组合为[1,2,3]、[1,2,4]、[1,3,4]和[2,3,4],则结果用双层列表表示为[[1,2,3],[1,2,4],[1,3,4],[2,3,4]]。同样,我们还需要注意 L 中有重复元素的问题,如遇到 L=[3,3,2,3]的时候应该怎么处理。

本节将讲解4种方法解决组合问题。其中解法一是一种比较笨的方式；解法二延续了排列问题逐层递增的思想；解法三延续了排列问题中每次将问题分成两部分的递归思想；而解法四给出了循环递归解决该问题的方法。

5.7.1 在排列问题的解法上解决组合问题(解法一)

有了前面排列的基础，我们首先会想到一个很笨的方法，那就是利用5.6节中求排列问题的函数先求得解T，然后将T中的每个元素进行排序，再去除那些重复的，就得到所有组合的解(因为组合对元素的顺序没有要求)。代码如<程序：combination_1>所示。

```
#<程序：combination_1>
def combination_1(L,k):
    T = permutation_2(L,k)              #5.6节排列问题中的函数
    T1 = []
    for e in T: T1.append(qsort(e))
    return removeDUPLICATE(T1)
def qsort(L):                           #对一个list做快速排序
    if len(L)<=1: return L
    a = L[0]; L0 = []; L1 = []
    for e in L[1:]:
        if (e<=a): L0.append(e)
        else: L1.append(e)
    return qsort(L0) + [a] + qsort(L1)
def removeDUPLICATE(A):                 #去除重复的元素
    T = qsort(A)
    L = []
    if len(T)>=1: L.append(T[0])
    for i in range(1,len(T)):
        if T[i] != T[i-1]: L.append(T[i])
    return(L)
```

上述代码中combination_1()函数就是主体部分，首先利用5.6节中的permutation_2()函数求得排列的解。qsort()函数为快速排序函数，关于排序在前面的部分已经讲解过，此处不再赘述，只不过这里选择的基准值始终是当前L中的第0号元素。removeDUPLICATE()函数的功能就是去重。下面以一个例子来解释：L=[1,2,3]，k=2，则求解排列问题可以得到所有排列的解T=[[1, 2], [1, 3], [2,1], [3, 1], [2, 3], [3, 2]]；对T中的每个元素进行排序得到T1=[[1, 2], [1, 3], [1,2], [1, 3], [2, 3], [2, 3]]；然后调用removeDUPLICATE()函数，首先对T1进行排序得到[[1, 2], [1, 2], [1, 3], [1, 3], [2, 3], [2, 3]]，然后通过比较来去重，得到最终的结果[[1, 2], [1, 3], [2, 3]]。由于我们调用的是permutation_2()函数，并且后面还有去重操作，所以<程序：combination_1>完全可以应对L中有重复元素的情况。

这种方式的执行效率肯定是不高的，但是由于充分利用了已经编写好的函数，所以可以快速地编写出一个正确的程序来完成任务。

5.7.2　非递归方式解决组合问题(解法二)

在排列问题中,我们运用逐层递增的思路解决了排列问题,那么是否能用同样的思路解决组合问题呢?

可以先试验一下看看,以 L=[1,2,3,4]为例:首先取第一个数,得到临时结果 R=[[1],[2],[3],[4]],然后分别对 R 中的每个元素扩充第二个数,[1]扩充后的结果是[[1,2][1,3][1,4]],而[2]扩充后的结果为[[2,1][2,3][2,4]],到这里我们发现[1,2]和[2,1]这两项虽然对于排列问题来说是不同的两项,但是由于组合问题对顺序没有要求,所以这两项对于组合来说其实是重复项。因此不能直接将<程序:permutation_1>中的代码挪用过来,而要在对元素扩充时多加一个限制条件,使最终的组合结果中没有重复项。

综上所述,可以在原有排列算法的基础上这样改变:仍然层层扩充地添加数,初始化临时结果集为[[]],取第一个数,用全部 n 个数对结果集[[]]进行第一次扩充操作,得到新的列表 R。然后,再为 R 中的每个元素扩充第二个数,在扩充的同时要满足下面这两个条件:一是注意要排除已有的数字;二是新添加的数一定要比待扩充列表中的最后一个数大。循环执行扩充步骤,直到扩充到 k 个数时,终止循环。经过 k 次这样的扩充,就得到了 n 个数取 k 个数的全部组合,且没有重复。

下面举一个具体的例子来详细说明一下算法的流程,以 L=[1,2,3,4]为例,k=3,结果为 R。

(1) 扩充第一个数:先使用 L 中的每个元素对空的 R 做元素扩充,得到新的 R=[[1],[2],[3],[4]],如图 5-24 所示。

(2) 扩充第二个数:对于 R 中的每个列表 e,获取 L 中的每个元素 f,只要 f 不在 e 中,且 f 比 e 中的最后一个元素大,就将 e 和[f]连接,将结果 e+[f]更新到列表 R 中,最终 R=[[1,2],[1,3],[1,4],[2,3],[2,4],[3,4]],如图 5-25 所示。

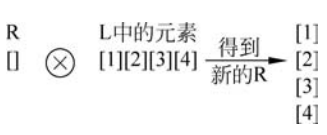


图 5-24　扩充第一个数的示意图

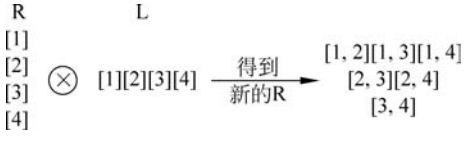


图 5-25　扩充第二个数的示意图

(3) 扩充第三个数:对于 R 中的每个列表 e,获取 L 中的每个元素 f,只要 f 不在 e 中,且 f 比 e 中的最后一个元素大,就将 e 和[f]连接,将结果 e+[f]更新到列表 R 中,R=[[1,2,3],[1,2,4],[1,3,4],[2,3,4]]。至此,便得到了最终的结果。可以看出,在每一步的过程中,每次扩充的时候添加的数必须要满足前面所提到的两个条件,所以结果集中没有重复的元素,如图 5-26 所示。

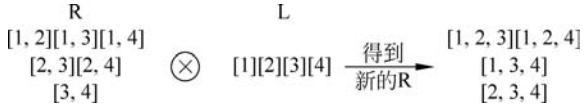


图 5-26　扩充第三个数的示意图

代码实现如<程序：combination_2>所示。需要注意的是，与<程序：permutation_1>类似，<程序：combination_2>目前也不能正确地处理 L 中有重复元素的情况，在该代码的基础上做修改会比较复杂，这里不做过多讲解。

```
#<程序：combination_2>
def combination_2(L,k):
    def product(R,L):          #辅助函数 product,给当前结果集 R 中每个元素扩充一位
        R1 = []                                  #初始化结果为一个空列表 R1
        if R == [[]]:                            #如果 R == [[]],则扩充到第一个数
            for e in L: R1.append([e])
        else:                                    #如果 R!= [[]],则根据两个规则扩充元素
            for e in R:
                for f in L:
                    if not f in e and f > e[len(e) - 1]:   #确保排除已有的数字且新添加的数
                                                           #一定要比待扩充列表中的最后一个数大
                        R1.append(e + [f])
        return R1
    if k > len(L) or k < 0:
        print('错误的输入,k 应当小于 L 中的元素个数,且 k 为非负数!')
        return []
    if k == 0: return [[]]                       #注意!当 k = 0 时,应当返回[[]]而不是[]
    R = [[]]
    for i in range (k): R = product(R,L)
    return R
```

5.7.3 特殊二分方式解决组合问题(解法三)

与排列问题中解法二的思想类似，同样可以用这种思路递归地解决组合问题，而且比排列更简单的是，我们不需要关心数字之间的顺序问题。如图 5-27 所示，对于一个列表，在解决 C_n^k 这个问题的时候可以将该问题分成两部分：T1 部分表示一定选取 L[0]的那些组合；T2 部分表示一定不选 L[0]的那些组合。那么 T1 和 T2 这两部分又分别有多少种组合呢？

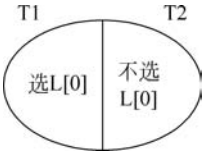


图 5-27 解法二的思路示意图

我们来分析一下：对于 T1 部分，既然选定 L[0]，则还需在剩下的元素中选取 k－1 个元素，所以 T1 部分解的个数也就是 C_{n-1}^{k-1} 的解的个数(假定 C_{n-1}^{k-1} 的解已经求出)，那么只需把 L[0]分别与 C_{n-1}^{k-1} 问题的所有解进行组合就得到了 T1 部分的所有解。对于 T2 部分，既然不选定 L[0]，那么还需在剩下的部分中选取 k 个元素，即 C_{n-1}^{k}。最后把这两部分的解加在一起就是要求的全部解，即 $C_n^k = C_{n-1}^{k-1} + C_{n-1}^{k}$。

具体的代码见<程序：combination_3>。这个程序是不是简洁又美丽？

```
#<程序：combination_3>
def combination_3(L, k):
    if len(L)<=k: return [L]
    if k == 0: return [[]]     #当要选择的元素个数是 0 个时,返回结果为空
```

```
    T1 = combination_3(L[0:len(L) - 1],k - 1)      #一定选择最后一个元素的结果 T1
    T2 = combination_3(L[0:len(L) - 1],k)          #不选最后一个元素,得到的结果 T2
    T = []
    for e in T1:
        e.append(L[len(L) - 1])
        T.append(e)
    return(T2 + T)
```

练习题 5.7.1 请将<程序：combination_3>的代码改成可以处理 L 有重复元素的情况。

【解题思路】 分别处理 T1 和 T2 部分的时候可以对其中的每个子列表进行排序，然后才添加到最终要合并的结果中，这样便于比较是否有重复的元素。

练习题 5.7.2 请参考排列问题中的练习题 5.6.1，同样以递归搜索方式解决组合问题。

【解题思路】 在实际编写代码的过程中，需要考虑到排列与组合的不同点：组合的解与顺序无关。比如，L=[5,3,9,8]，k=2，思考<程序：permutation_dfs>，T[0]=5、T[0]=3、T[0]=9、T[0]=8 都是可以的(T[0]=5 时 T[1]=3 或 T[1]=9 或 T[1]=8；T[0]=8 时 T[1]=5 或 T[1]=3 或 T[1]=9……)，因为排列与顺序有关。对于组合来说，如果还按照排列的方式来选，就会出现重复的解，比如 T[0]=5、T[1]=8 和 T[0]=8、T[1]=5 实际是同一个解。所以在编写组合问题的代码时需要注意，应该如何解决这一问题呢？可以设定规则：若确定 T[0]=5，则 T[1]中的元素就只能从[3,9,8]中选；若确定 T[0]=3，则 T[1]中的元素就只能从[9,8]中选；若确定 T[0]=9，则 T[1]中的元素就只能从[8]中选。这样就不会有重复的解出现了。

具体实现如<程序：combination_dfs>所示。如果 L 中有重复元素，那么通过每次对 T_new 进行排序，从而方便去除重复的解，故该程序可以解决 L 中有重复元素的问题。注意：如果 L 中没有重复元素，那么 sort 和后面的 if 检查是可以去除的。

```
#<程序：combination_dfs>
def combination_dfs(L,k):
    R = []; T = [''] * k
    def F(L,num_pick,m):
        if m > num_pick - 1:
            T_new = T[:]; T_new.sort()                    #若 L 中有重复,才需要做 sort 和检查
            if not T_new in R: R.append(T_new)
        else:
            for i in range(len(L) - (num_pick - m) + 1):  #注意 i 的选择范围
                T[m] = L[i]
                F(L[i + 1:len(L)],num_pick,m + 1)
    F(L,k,0)
    return R
```

5.7.4 循环递归方式解决组合问题(解法四)

除了使用特殊二分法求解外，还可以换一种递归的方式来解决组合问题，也就是多分法

的方式，递归调用是在 for 循环内的。函数的参数仍然是 L 和 k，其中，L 表示当前要处理的列表，k 表示当前要选取的元素个数。对于求从 L 中取 k 个数组合时，可以分为以下情况：

(1) 一定选择倒数第 1 个数，然后从剩下的 n−1 个数中选 k−1 个数的组合。

(2) 一定不选倒数第 1 个数，一定选倒数第 2 个数，再从剩下的 n−2 个数中选 k−1 个数的组合。

(3) 一定不选倒数第 1、2 个数，一定选倒数第 3 个数，从剩下的 n−3 个数中选 k−1 个数的组合。

……

以此类推，考虑剩下的数越来越少，一直到剩下的数为 k−1 的极限。也就是到 n−k+1，直到一定不选倒数第 1,2,…,n−k 个数，一定选倒数第 n−k+1 个数，从剩下的 k−1 个数中选 k−1 个数的组合。

从上面的分析中可总结得出当从 L 中取 k 个数时，可以分为最多 n−k+1(n≥k+1)种需要递归的情况。这里以一个简单的例子来解释这种思路。假设 L=[1,2,3,4]，现在求从 L 中取 2 个数的解，即 C_4^2，可以分为 3 种情况。

情况一：一定选取最后一个数 L[3](即[4])，那么需要求从剩下的前 3 个数([1,2,3])里取一个数(即还需要选 k−1 个数，C_3^1)的解，记为 A1，再与 L[3]进行组合。怎么求解 A1 呢？可以利用递归的方式继续去求得 C_3^1 这一问题的解，则又可以分为如下 3 种情况：

(1) 此时函数应传入的参数为 L=[1,2,3]和 k=1。首先，一定选取 L[2](即[3])，则还需在剩下的前两个数([1,2])中选零个数，即 C_2^0。再将 L[2]与 C_2^0 的所有解组合。故最终返回解 B1=[3]。

(2) 一定选取 L[1](即[2])，还需在剩下的前一个数([1])中选零个数，即 C_1^0。再将 L[1]与 C_1^0 的所有解组合。最终返回解 B2=[2]。

(3) 一定选取 L[0](即[1])，此时 L[0]前面已经没有可选的元素了，即需要将 L[0]与 C_0^0 的所有解组合。最终返回解 B3=[1]。

得到的所有解组合在一起即为 A1 的解。A1=[[3],[2],[1]]。当求得 A1 之后，只需要将 L[3]分别插入 A1 的所有解中，即可得到所有带数字 4 的组合 S1=[[3, 4], [2, 4], [1, 4]]。求得该情况下的结果后，进入下一种情况。

情况二：一定选取 L[2](即[3])，由于包含数字 4 的所有组合已经被找出来了，所以不会再考虑 4 了。那么还需要在剩下的前两个数中([1,2])中选一个数，即求 C_2^1 的解，记为 A2。A2 的求解也是利用递归的方式，参数变为 L=[1,2]、k=1，此处不再赘述，最终求得 A2=[[2],[1]]。再将 A2 的所有元素分别与 L[2]组合，得到 S2，即所有包含数字 3 而没有数字 4 的组合 S2=[[2,3],[1,3]]。求得该情况下的结果后，进入下一种情况。

情况三：一定选取 L[1](即[2])，那么还需在 L[1]前面所有剩下的元素中选一个，即求 C_1^1 的解，记为 A3。A3 的求解也是利用递归的方式，参数变为 L=[1]、k=1，最终求得 A3=[[1]]。再将 A3 的所有元素分别与 L[1]组合，得到 S3，即所有包含 2，但不含 3 和 4 的组合，S3=[[1,2]]。

以上 3 种情况已经囊括了所有组合，因为若从 L[0]开始继续选取元素，即便确定选取 L[0]，还需要在 L[0]之前剩下的元素中选取一个，但显然已经没有元素可以选了。故以上 3 种情况已经求得所有组合，循环终止，将得到的解合并起来就是最终的答案 S=S1+S2+S3=[[3,4], [2,4], [1,4], [2,3],[1,3], [1,2]]。

总结上述解法，得到这样一个公式：$C_4^2=C_3^1+C_2^1+C_1^1$。而 $C_3^1=C_2^0+C_1^0+C_0^0$，这其实就是一个递归思想，把大问题拆分成小问题，最后再把小问题的解组合起来。那么可以总结出这样一个递推关系式：$C_n^k=C_{n-1}^{k-1}+C_{n-2}^{k-1}+\cdots+C_{k-1}^{k-1}$。

最后确定递归的终止条件：当 k=0 或者 k>len(L)时返回空列表[[]]。同时也会出现 L 中所剩下的元素个数等于 k 的情况，此时就应该返回整个当前 L。

有了前面的递推公式和终止条件，现在就可以写出相应的递归函数。见<程序：combination_4>。

```
#<程序：combination_4>
def combination_4(L,k):
    if k == 0 or len(L)< k:return [[]]
    if len(L) == k: return [L]
    n = len(L); T = [L[n - 1]]; R = []
    for i in range(n - 1,k - 2, - 1):        #注意循环的范围
        A = combination_4(L[0:i],k - 1)
        for e in A:
            e_new = e + T
            e_new.sort()                     #排序，方便去重，若 L 没有重复，则 sort 和 if 可去掉
            if not e_new in R: R.append(e_new)
        T = [L[i - 1]]                       #记得在每次外面的 for 循环中更新 T 的值，T 总是表
                                             #示要选取当前要处理的 list 中的最后一个元素
    return R
```

该递归函数的输入仍然是 L(列表)和 k(从列表中选取的个数)，首先需要两个变量：T 用来记录每次 L 中的最后一个数，表示必定要选取它；R 变量用作返回值。

接下来就通过 for 循环的方式来模拟 $C_{n-1}^{k-1}+C_{n-2}^{k-1}+\cdots+C_{k-1}^{k-1}$ 这一部分，即变量 i 是从 n－1 依次递减到 k－1，保证剩下可选的元素个数大于或等于 k。由于 Python 中 range()函数是左闭右开的区间，故应写为 range(n－1,k－2,－1)。可以举例来验证这个 range()函数是否正确，$C_8^5=C_7^4+C_6^4+C_5^4+C_4^4$，即 n=8、k=5，那么变量 i 就应该从 7 依次递减到 4，可以看出 range()函数是正确的。变量 A 则为确定选取当前 L 中最后一个元素 T 之后 C_{n-1}^{k-1} 的解，然后将 T 与 A 中的每个元素组合就得到了包含 T 的所有组合的解，需要注意更新 T 的值。最后通过确认递归的终止条件来返回最终解。<程序：combination_4>同样通过对每个待添加进结果集中的元素进行排序，从而达到去重的目的。注意，若 L 本身就没有重复的元素，那么 sort 和下面的 if 检查都可以去掉。

这个程序很有趣，它的递归调用是在 for 循环内。论其简洁的程度，是没有前面特殊二分法那么简单明了的。

习题

习题 5.1　改写<程序：递归实现二分查找>，使得若 k 大于 L[len(L)//2]，调用 BinSearch(L[len(L)//2+1:],k)。注意，return 的索引 len(L)//2+index 要改动。另外，

可否去掉 if len(L)==1 的检查。

习题 5.2 用非递归实现二分法求解问题中的<程序：二分法递归查找插入位置>。

习题 5.3 <程序：二分法递归查找插入位置>中，假如去掉“if k<L[0]：return index_min”，程序是否还是正确的？

习题 5.4 用递归实现求解算术平方根问题中的<程序：算术平方根运算——二分法>。

习题 5.5 修改求解算术平方根问题中的<程序：算术平方根运算——二分法>的代码，使其可以求解一个实数 c 的 k 次方根(函数有 c 和 k 两个参数)。

习题 5.6 编写代码，求解 $\log_{10} x = a$，精度为 0.000 000 000 01。函数的形式为“def log(a)：… return x”。

习题 5.7 假如有 n 个钱币，其中有一个钱币是假的，已知假的钱币比较轻，你只有一个天平，如何用非递归和递归思维来找到这个假币？请写出 Python 程序。请先写出一个起始函数来设定一个列表，列表有 100 个 1，代表 100 个钱币，再随机设定一个索引值，将其改为小于 1 的任意数，代表是个假钱币。你的程序要输出这个假钱币的索引。

习题 5.8 如果在 n(n≥4)个硬币中有两个较轻的假币，要怎么找出假币？请编程实现。

习题 5.9 n(n≥3)枚硬币中有一枚是假币，但如果只知道假币的重量和真币不同，怎么才能在这 n 枚硬币中找出这枚假币？请编程实现。

习题 5.10 调用 time 库，测试用分解因数和欧几里得两种 gcd(p,q)程序对下列 4 个 p 值的执行时间，$p_1 = 53\ 722\ 280\ 714\ 561$，$p_2 = 658\ 381\ 238\ 967\ 811$，$p_3 = 4\ 890\ 443\ 419\ 341\ 343$，$p_4 = 51\ 610\ 544\ 808\ 296\ 093$，而 q=1 234 567。你能得出什么结论？

习题 5.11 编程实现求 k(k≥3)个数的最大公因数。

习题 5.12 编程实现求 k(k≥3)个数的最小公倍数。

习题 5.13 请不用递归的方式再次实现欧几里得算法求最大公因数中的<程序：欧几里得算法求最大公因数>。

习题 5.14 请根据本章讲解的欧几里得方法，自己在草稿纸上计算 gcd(388,128)，看看是否仅需几步就求解出答案。

习题 5.15 编程实现随机产生 1 个 20 位的数，使得该数与 111 这个数互质。

习题 5.16 请根据 5.3 节中的知识，自己动手写出自然数 1～10 的数字对(a,b)表示方法，其中 a 为该自然数除以 3 的余数，b 为该自然数除以 8 的余数。并检验一下，做加法运算时是否正确。

习题 5.17 请用手算 $3^{64} \bmod 7$。

习题 5.18 请用手算 $(3^{97} \times 2)^{33} \bmod 7$。

习题 5.19 写 Python 程序算出 $a^x \bmod b$ 的值。函数 mod(a，x，b)，返回 $a^x \bmod b$ 的值。假设 a 和 b 是最多 10 位的整数，而 x 可以是最多 200 位的整数。请用递归的思维来编写此程序。

习题 5.20 如同上题，但是 x 可能是 800 位的整数。必须快速地算出答案来。

习题 5.21 利用中国余数定理中的<程序：求倒数>求解 p 对 q 取余的倒数和 q 对 p 取余的倒数。

(1) p=25，q=34，先动手计算一下，然后运行程序，看结果是否正确。

(2) P 为习题 5.15 中得到的 20 位的数，q 为 111，运行<程序：求倒数>求得 p'和 q'。

习题 5.22　改写 Extended_Euclid()函数，不用递归的方式。

习题 5.23　编程实现 3 个方程的中国余数定理求解问题。提示：3 个方程的中国余数定理仍旧可以用“凑”的思想凑出解。

习题 5.24　为了深入理解“凑”的思想，请同学们思考求解 $y=ax^2+bx+c$ 问题。已知该曲线经过的 3 个点分别为(x0,y0)、(x1,y1)、(x2,y2)。如何利用“凑”的思想求得系数 a、b、c?

提示：凑的方法如下所示。

	1		2		3
	$y0\times\frac{(\ldots)(\ldots)}{(\ldots)(\ldots)}$	+	$y1\times\frac{(\ldots)(\ldots)}{(\ldots)(\ldots)}$	+	$y2\times\frac{(\ldots)(\ldots)}{(\ldots)(\ldots)}$
代入x0	y0	+	0	+	0
代入x0	0	+	y1	+	0
代入x0	0	+	0	+	y2

习题 5.25　世界上常用的一种安全编码方式为 RSA，其中产生公钥和私钥的过程中会用到本章介绍的倒数的概念，实现方式为：给定两个质数 p、q，随机产生一个奇数 e，满足 e<(p−1)(q−1)，且与(p−1)(q−1)互质，即 gcd(e,(p−1)(q−1))=1，在 e 的基础上产生 e 的倒数 d，即 ed=1(mod (p−1)(q−1))。以上过程中产生的 e 即为公钥，d 为私钥。

请编程实现求解私钥：对于给定的两个质数 p=128 543 041 447 753 和 q=1 062 573 853 363 145 487 845 851，先随机产生 e<(p−1)(q−1)并且满足 gcd(e,(p−1)(q−1))=1，然后求出 d 并打印出来。

习题 5.26　在上一题的基础上，假设 n=pq，给定任意一个整数 m，m<n，请试验当 m^e mod n，结果为 m'，那么对(m')^d mod n 会产生原来的 m。如果上述结论成立，则在信息传递中，发送者只需要传递 m'，接收者会用私钥 d 就可以得到原来真实的信息 m。请用中国余数定理来完成 m^e mod n 的计算，其原理就是首先计算 m^e mod p，再计算 m^e mod q，最后利用中国余数定理即可得到 m^e mod n 的结果。所以请编写能够实现快速求得 m^e mod p 的程序，注意 e 和 p 都可能是很大的数。

习题 5.27　不用递归方式实现线性方程组求解问题。

习题 5.28　请将<程序：combination_3>的代码改成可以处理 L 有重复元素的情况。

习题 5.29　请认真学习本节中排列问题和组合问题的思想以及程序，在代码中添加一些 print 语句作为辅助信息，分别用有重复元素的列表和没有重复元素的列表作为待处理的参数，执行程序，体会每个程序的执行过程。

习题 5.30　请分析本章所展现的各个组合程序的优劣。

习题 5.31　改写<程序：combination_4>。使得从 L 的第一个元素开始考虑，而不是从最后一个开始考虑。也就是首先考虑一定选择第一个数 L[0]，然后从剩下的 n−1 个数中选 k−1 个数的组合；然后，一定不选 L[0]，一定选第 2 个数 L[1]，再从剩下的 n−2 个数中选 k−1 个数的组合；以此类推。

习题 5.32　你能自己设计出一种本章没有展现的排列或组合的程序吗？请尝试。