第5章 计算思维的核心——算法

由于各类专业都需要利用计算机来解决问题，对于广大非计算机专业的、没有受过较严格计算机科学教育的人们而言，“计算思维(Computational Thinking)”成为他们必须要掌握的知识，也就是如何用计算机来解决问题。而对于计算机科学专业的人来说，几十年来，计算机科学很少强调所谓的“计算思维”这个名词，因为“计算思维”是理所当然的，老早就根深蒂固在计算机科学的血脉里，从一开始这门学科就是研究用计算机解决问题的方法。如何用计算机解决问题就是计算思维的范畴。发展多年来，我们将此称为算法(Algorithms)。计算机专业的人不需要去刻意区分这两个名词。本章当讲到较大的概念时会不免俗套地用“计算思维”这个名词。当谈到具体的实现方法时，本章就用“算法”以代之。

算法是计算机科学之美的体现之一。算法不是用背诵的，而是要理解的。我们要把算法理解透彻，成为我们的习惯思维，或许这就是所谓的计算思维。对算法的深刻理解到计算思维的养成，可以帮助我们在日常生活、行政管理、时间规划、经营理财等各类问题的解决上得到莫大的助益。注意，算法是超乎于程序语言之外的，设计好算法后，用哪个程序语言来编程(例如 Python、C、C++、Java)是个直接而相对简单的事了。

本章会向大家介绍如何用计算机解决问题的思维方式，5.1 节通过简单的例子向大家介绍什么是计算思维，并给出计算思维的定义；5.2 节介绍递归，它是计算机科学解决问题的基本思路与技巧；5.3 节、5.4 节和 5.5 节会分别为大家介绍分治法、贪心算法和动态规划，这些是非常重要的解题方法；5.6 节以老鼠走迷宫为例，向大家展示怎样利用计算思维求解问题；5.7 节通过总结本章所学的内容谈谈计算思维的美。

虽然多年的经验告诉我，动态规划技术是计算机算法里最重要的技术，但是它比较复杂，需要多点时间去熟练它。假如学时数不够，动态规划部分可以先行跳过，等以后有足够的时间再回来学习这部分。本章提供了足够的材料和 Python 例子，由老师自行计划和掌握要教的部分。

沙老师：算法就好像是内功心法，计算思维就好像是修炼好心法后的内功。举手投足，起心动念，皆是算法，而不自知。

5.1 计算思维是什么

大家可能对“计算思维”这个词很陌生。其实你已经接触过计算思维了，只是自己还蒙在鼓里。本书的第 1 章给出的三种计算平方根的方法都是计算思维，让我们再重新回顾

一下。

在第 1 章中，使用了三种不同的计算思维求 y 的平方根。

第一种考虑到可以根据已知平方根的数来确定 y 的平方根的范围，然后在这个范围内寻找答案。例如，如果 $y=10$，根据 3 的平方是 9，而 4 的平方是 16，所以 y 的平方根 g 一定满足：$3<g<4$。那么首先可以让 $g=3$，然后重复给 g 加一个很小的数 h，直到 g^2 足够接近于 y，从而求得 y 的平方根 g。

第二种采用更快的二分法逼近的方法来求解。使 $f(g)=g^2-y$，此时满足 $f(g)=0$ 的那个 g 就是答案。首先确定 y 的平方根 g 最小为 min=0，最大为 max=y，使 $g=(\min+\max)/2$，然后通过判断 $f(g)>0$ 还是 $f(g)<0$，从而去掉一半的可能范围，缩小 g 的取值范围，一步步逼近解。这种逼近方法是有效的，每一次 g 的取值范围会缩小一半，缩小的速度相当快。这种方法叫作“二分法”。

第三种也是一步步逼近解，只是逼近方式更加高效。使 $f(g)=g^2-y$，此时满足 $f(g)=0$ 的那个 g 就是答案。通过每次求 $f(g)$ 切线的斜率，从而一步步逼近解。

借助计算机强大的计算能力，上述三种计算思维都能解决平方根问题，只是效率不同。大家再也不用死记硬背 2 的平方根是 1.414，3 的平方根是 1.732 了。只要用上述的计算思维就能求解任何数的平方根。

小明：计算思维就是要像计算机那样思考吗？

沙老师：你有点傻。计算思维就像我们平时所说的数学思维、抽象思维一样，只是一种用来解决问题的方法和途径，并不是要人像计算机那样思考。

平方根的例子是做数学运算，计算机更多地是做逻辑决策。现在用一个比较简单的找假币问题为例。假设你有 $n(n\geqslant 2)$ 枚硬币，知道其中有一枚假币，而这枚假币的重量比真币要轻，怎样才能找出这枚假币呢？

自己先想想，你可以想出多少种方法呢？

提示：既然知道假币的重量较轻，那么只要比较一下重量就知道哪枚是假币了，根据比较的策略，我们可以分为下面三种方式。

第一种方式：就是一个个比较硬币，直到找到假币为止。假设 $n=10$，需要在 10 枚硬币中找出假币。首先，比较硬币 1 和 2，这样会出现两种情况：

(1) 如果两枚硬币重量不一样，那么重量较轻的就是假币了；

(2) 如果两枚硬币重量一样，就从两枚中随便找出一枚与下面的硬币比较。

像上面所述依次比较硬币 3，4，5，…直到找出假币。在最差的情况下，要比较 9 次才能找出假币，比较过程如图 5-1 所示。而要在 n 枚硬币中找出假币，就要比较 $n-1$ 次。

但是观察上面的比较过程，好像有很多不必要的比较。比如既然知道假币的重量较轻，并且只有一枚假币，那么如果两枚硬币重量一样，这两枚硬币就一定都是真币了，在接下来的比较中也就不用比较这两枚硬币了。因此，可以去掉这些不必要的比较，这样就能得到第二种方式。

第二种方式：将 n 枚硬币中每两枚硬币分为一组，依次比较每组中的两枚硬币，直到找到假币为止，最差情况下只须比较 $n/2$ 次。假设 $n=10$，将 10 枚硬币两两分组，可以分成五

组。首先比较第一组中的硬币 1 和 2,会出现两种情况:

(1) 如果两枚硬币重量不一样,那么重量较轻的就是假币了;

(2) 如果两枚硬币重量一样,就继续比较下一组的两枚硬币。

像上述过程依次比较,直到找到假币为止,最坏情况下要比较 5 次,分组情况如图 5-2 所示,依次对 5 组进行比较,最多比较 5 次就能找出假币了。而要在 n 枚硬币中找出假币,最差情况下要比较 $n/2$ 次。

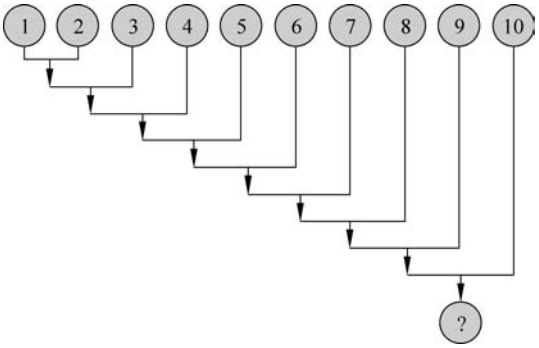


图 5-1 简单比较法

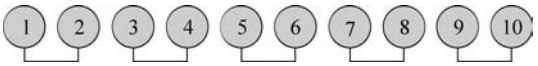


图 5-2 分 $n/2$ 组比较法

但是比较 $n/2$ 次才能找出假币并不是最快的方式。既然所有真币的重量都一样,可以将硬币分成个数相同的两份,有假币的一份会轻一些。而较重的那堆硬币一定都是真币,也就不用做比较了。按照这种方法可以设计出更快的方式,只需要比较 $\log_2 n$(取 $\log_2 n$ 的整数部分)次就能找出假币,即下面所述的方法。

第三种方式:二分法。

(1) 如果 n 是偶数,将 n 枚硬币平均分成两份,比较这两份硬币的重量,假币在重量较轻的那份中,继续对重量较轻的那份硬币使用二分法,直到找出假币;

(2) 如果 n 是奇数,随意取出一枚硬币,然后将剩下的 $n-1$ 枚硬币平均分成两份,比较这两份硬币的重量。如果两份硬币重量相等,那么取出的那枚硬币就是假币;如果两份硬币重量不相等,那么假币在重量较轻的那份中。继续对重量较轻的那份硬币使用二分法,直到找出假币。

假设 $n=10$,将 10 枚硬币{1,2,3,4,5,6,7,8,9,10}平均分成两份:{1,2,3,4,5}和{6,7,8,9,10}。比较这两份的重量,假设第一份较轻,那么假币一定就在硬币 1,2,3,4,5 中,而硬币 6,7,8,9,10 一定都是真币。继续用二分法在{1,2,3,4,5}这 5 枚硬币中找假币。因为 5 是奇数,首先随意取出一枚硬币,假设取出第 5 枚硬币。然后将剩下的 4 枚硬币平均分成两份:{1,2}和{3,4}。比较两份的重量,如果两份硬币重量相等,那么第 5 枚硬币就是假币;如果两份硬币重量不相等,假设第一份较轻,那么假币一定就在硬币 1 和 2 中,而硬币 3,4,5 一定都是真币,此时只要再比较硬币 1 和 2 就能找出假币了。

使用二分法在 10 枚硬币中找出假币最多要比较 3 次,过程如图 5-3 所示。

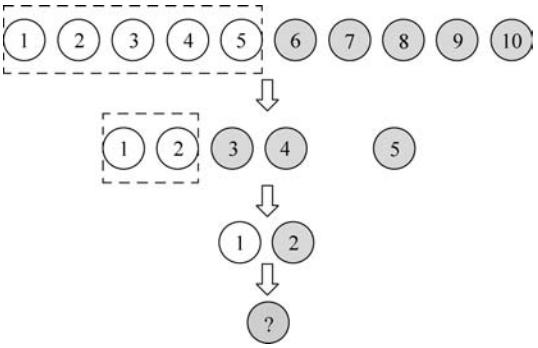


图 5-3　二分法

观察二分法，先将 n 枚硬币平均分成两份做比较，然后将 $n/2$ 枚硬币平均分成两份做比较，继续将 $n/4$ 枚硬币平均分成两份做比较……直到将两枚硬币平均分成两份做比较。在整个过程中，比较的次数就是划分的次数，而做划分的次数其实就是 $\log_2 n$（以 2 为底 n 的对数）。

上面三种找假币的方式都能找出假币，但是有的速度快，有的速度慢。在例子中，$n=10$ 时可能并不明显，但是当 n 非常大的时候，速度的快慢就相差很大了。比如当 $n=10^6$ 时，第一种方式要比较 10^6-1 次；第二种方式要比较 $10^6/2$ 次；而第三种方式只要比较 20 次就可以了（想想为什么？注意 $\log_2 10$ 差不多等于 3.32）。由图 5-4 可以看出，三种方式的比较次数 $F(n)$ 随着 n 的增长而变化的情况。第三种方式的比较次数 $\log_2 n$ 增长速度明显比前两种慢很多，因此第三种方式是最好的找假币的方法。

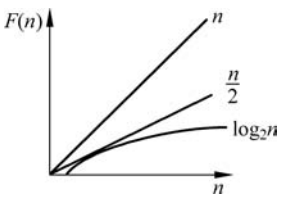


图 5-4　三种方式比较次数的增长情况

上面三种找假币的方式也是三种不同的计算思维。已知假币较轻的情况下，利用第三种方式在 n 枚硬币中找出假币只需要比较 $\log_2 n$ 次。

```
#<程序：找假币的第一种方法> by Edwin Sha
def findcoin_1(L):
    if len(L) <= 1:
        print("Error: coins are too few"); quit()
    i = 0
    while i < len(L):
        if L[i] < L[i+1]: return (i)
        elif L[i] > L[i+1]: return (i+1)
        i = i+1
    print("All coins are the same")
    return(len(L))   # should not reach this point
```

练习题 5.1.1：请用 Python 实现找假币问题的第二种方式。这个程序需要实现的功能有：要求用户输入硬币个数 n；在 2 到 5 中随机选取真币的重量，假币的重量是真币的重量减 1；再从 0 到 $n-1$ 中随机产生一个数，作为假币的位置；产生一个列表 L，依序包含每一

个钱币的重量,例如 L=[3,3,3,3,3,2,3,3,3,3];利用算法,找出假币所在的位置(第 1 枚硬币的位置是 0)。文中已列出实现找假币问题第一种方法的 Python 程序。

```
#主要程序
import random
n = int(input("Enter the number of coins >= 2: "))
w_normal = random.randint(2,5)
index_faked = random.randint(0,n-1) # 0 <= index <= n-1
L = []
for i in range(0,n):
    L.append(w_normal)
L[index_faked] = w_normal - 1
print(L)
print("The index of faked coin:",findcoin_1(L))
```

练习题 5.1.2:用 Python 实现第三种方式,即二分法算法。请不要用递归函数,可以利用 Python 原有的 sum()函数,将一堆钱币的重量加起来。

练习题 5.1.3:用 Python 递归函数的方式实现二分法算法。可以利用 Python 原有的 sum()函数,将一堆钱币的重量加起来。解释<程序:二分法找假钱币>,并且分析:这个程序的参数 a 是做什么用的? return(-1)代表有哪些情况发生?

```
<程序:二分法找假钱币>
def findcoin(a,L):
    x = len(L)
    print(a,L)
    if x == 1: return(a)
    if x % 2 == 1: x = x - 1;y = 1
    else: y = 0
    if (sum(L[:x//2])< sum(L[x//2:x])):
        return(findcoin(a, L[:x//2]))
    elif (sum(L[:x//2])> sum(L[x//2:x])):
        return(findcoin(a + x//2,L[x//2:x]))
    else:
        if y == 0: return( - 1)
        else:
            if(L[x]< L[0]):return(a + x)
            else: return( - 1)
```

练习题 5.1.4:上面我们用了二分法解决找假币问题,那么能不能用三分法(将硬币分成三份来进行比较)呢? 能不能用 $k(3\leqslant k\leqslant n)$分法呢? 请分析 k 分法的优劣。

练习题 5.1.5:如果在 $n(n\geqslant 4)$枚硬币中有两枚较轻的假币,要怎么找出假币?

练习题 5.1.6:如果只知道假币的重量和真币不同,怎么才能在 n 枚硬币中找出这枚假币呢?

练习题 5.1.7:根据练习题 5.1.6 的算法,写出 Python 程序。

小结

本节我们向大家介绍了计算思维的一些内容。计算思维是运用计算机科学的基础概念进行问题求解、系统设计，以及人类行为理解等涵盖计算机科学之广度的一系列思维活动。其实简单来说，计算思维就是用计算机科学解决问题的思维。它是每个人都应该具备的基本技能，而不仅仅属于计算机科学家。对于学计算机科学的人来说，培养计算思维是至关重要的。

5.2 递归的基本概念

用递归(Recurrence)的方法来解决问题是计算机科学里面最美的部分之一，基本概念就是一个问题的解决方案是由其小问题的解决方案构成的。本节讲授其基本概念，接下来的各种算法技巧，如动态规划、分治法、贪心算法都是基于递归概念的方法，所以对递归概念的熟练运用是计算机科学学习的重中之重。

递归函数是自己调用自己的函数，在本质上形成一个循环。稍不小心“循环”就会变成烦恼的缘由。不管是自己循环自己，还是在一个共同工作的团队里，“我等它完成，它也在等我完成”这类的死锁循环，我们不可不慎啊。

先讲一个在语言上的递归循环。

A 对 B 说：“我给你讲个故事吧。”

B：“好啊。”

A：“从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？‘从前有座山，山里有座庙，庙里有个老和尚，正在给小和尚讲故事呢！故事是什么呢？……(没完没了的重复)’’’”

上面这个恶作剧其实就是一种语言上的递归。还有一些语言上的递归，比如“我下句话是对的”和“我上句话是错的”这两句话就是在相互调用，谁也不知道“我”说的话是对的还是错的。再比如“我在说谎”这句话在自己调用自己，如果我说谎，那么“我在说谎”这句话就是一句谎话，也就是说我没有说谎；如果我没有说谎，那么“我在说谎”这句话就是一句真话，也就是我说谎，谁也不知道我说没说谎。所以“我在说谎”这句话在逻辑上是没有对错的。

除了语言上的递归外，还有很多形式的递归，请看图 5-5 和图 5-6 所示的两幅画。《画手》和《瀑布》是错觉图形大师埃舍尔(Maurits Cornelis Escher)的两幅著名作品，这两幅画是一种图形上的递归。

不过，计算机科学中所要学的递归与上面的递归有点儿不同。我们需要用递归思想来解决问题，可不能永远循环。

小明：假如我对小丽说：“我爱你，我在说谎。”我到底有没有说谎啊？真是搞迷糊了，头痛。

图 5-5 《画手》

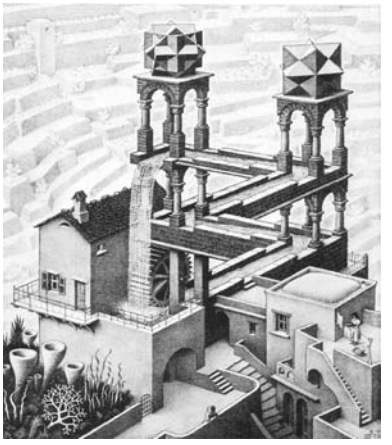


图 5-6 《瀑布》

1. 加法问题

问题描述：有 n 个数 $a_1,a_2,\cdots,a_n$，求这 n 个数的和 $F(n)$。

如果 $a_1=1,a_2=2,\cdots,a_n=n$，则 $F(n)=1+2+3+\cdots+n$。这个问题大家在中学就知道，$F(n)$的封闭型解(Closed Form Solution)是 $n(n+1)/2$，不需要编写递归程序就能得到 $F(n)$。但是如果 $a_1=1^k,a_2=2^k,\cdots,a_n=n^k(k>3)$，可能就很少有人知道准确的封闭型解了，我们只能编程计算 $F(n)$了。这个时候用递归的方式是最简单的方式，不需要用任何 for 循环、while 循环的格式。

递归函数：$F(1)=a_1$；$F(n)=F(n-1)+a_n$。

用 Python 编程是很简单的：第一步写上终止条件，然后调用递归函数。

```
#<程序：递归加法>
def F(a):
    if len(a) == 1: return(a[0])      #终止条件非常重要
    return(F(a[1:]) + a[0])
a = [1,4,9,16]
print(F(a))
```

练习题 5.2.1：前面的 Python 函数的递归调用形式其实是第一个数 a[0]加上剩下的 n−1 个数的和。请改写这个程序，使得程序的递归成为前 n−1 个数的和加上最后一个数 a[n−1]。

为了让大家进一步的了解递归的思想，我们再来看一个用递归求解的例子。

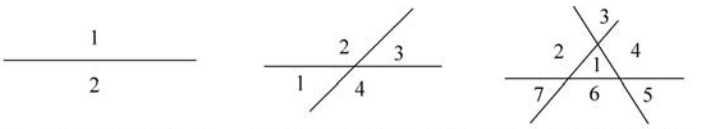
2. 平面划分问题

问题描述：求 $f(n)=n$ 条直线最多可以划分的平面个数。

应用计算思维的解题习惯，首先要将大问题划分成小问题。求 n 条直线最多可以划分多少个平面的问题是一个很复杂的大问题。首先可以知道，1 条直线最多可以划分两个平面，两条直线最多可以划分 4 个平面，3 条直线最多可以划分 7 个平面。

如图 5-7(a)所示，1 条直线最多划分出两个平面，即 $f(1)=2$；求两条直线最多划分的

平面数 $f(2)=4$，可以在 1 条直线划分的情况下，加上第 2 条直线，使其与第 1 条直线相交，如图 5-7(b)所示。这样可以在 1 条直线划分的情况下多划分出两个平面，也就是 $f(2)=f(1)+2$；求 3 条直线最多划分的平面数 $f(3)$时，可以在两条直线划分的情况下，加上第 3 条直线，使其分别与前 2 条直线相交于不同点，如图 5-7(c)所示。这样可以在两条直线划分的情况下多划分出 3 个平面，也就是 $f(3)=f(2)+3$。



(a) 1条直线划分的平面　(b) 2条直线划分的平面　(c) 3条直线划分的平面

图 5-7　当 $n=1,2,3$ 时，最多划分的平面个数

那么 n 条直线最多划分的平面数 $f(n)$能否用 $f(n-1)$构筑而成呢？

根据上面用 $f(1)$构筑 $f(2)$和用 $f(2)$构筑 $f(3)$的情况，同样地，求 n 条直线最多划分的平面数 $f(n)$时，可以在 $n-1$ 条直线划分的情况下，加上第 n 条直线，使其分别与前 $n-1$ 条直线相交于不同点，每有一个交点就多一个平面，最后一个交点之外还会增加一个平面。这样可以在 $n-1$ 条直线划分的情况下多划分出 n 个平面，也就是 $f(n)=f(n-1)+n$。如此，就可以得到递归式(5-1)：

$$f(n)=\begin{cases}2, & n=1\\ f(n-1)+n, & n>1\end{cases} \tag{5-1}$$

有了递归式，这问题基本就解决了，可以编程来算出任何 $f(n)$的值。假如要在数学上求出封闭型解(Closed Form Solution)也不难。根据递归式(5-1)，可以知道 n 条直线最多划分的平面数 $f(n)=f(n-1)+n=\cdots=2+2+3+\cdots+(n-1)+n=n(n+1)/2+1$。大家可以用 $n=1,2,3$ 来做验证。

练习题 5.2.2：请用 Python 编写一个解决平面划分问题的递归程序。输入 n，输出 $f(n)$。

递归是计算机科学解决问题的基本思路与技巧，简单来说，就是通过不断地调用自己来解决问题的一种思路。下面通过最经典的汉诺塔问题向大家介绍递归。

3. 汉诺塔(Hanoi Tower)问题

汉诺塔(又称河内塔)问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘。大梵天命令婆罗门把圆盘按大小顺序重新摆放在另一根柱子上，并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。当所有的黄金圆盘都从大梵天穿好的那根柱子上移到另外一根上时，世界就将在一声霹雳中毁灭。那么移动 64 片黄金圆盘到底需要移动多少次呢？我们后面会分析给大家看，需要移动约 10^{19} 次。但是我们学计算机科学的人，只需要短短的几行代码就能解决了。所以我们只要将这几行代码交给大梵天就完成任务了，我们是怎么做到的呢？

我们先不考虑移动 64 片圆盘的次数，这个问题太大太复杂，想想都会让人头晕。让我们先从比较少的圆盘数开始，这样有助于发现这个问题的规律。设 n 表示圆盘的片数，有 A、B、C 三个柱子，原来那个圆盘在 A 柱子上，要全部移动到 C 柱子上，用 B 柱子做中间柱

子。当 $n=1$ 时很简单,只要移动一次就好了,即移动次数 $f(1)=1$。当 $n=2$ 时也不难知道,移动次数 $f(2)=3$。

接着我们思考要移动 n 个圆盘要怎么做?我们有计算思维的人解决这个问题是很简单的。大问题的解答要由小问题的解答来构建,$f(n)$的求解可以由 $f(n-1)$的解答来完成。我们可以先移动 A 柱上的 $n-1$ 个圆盘到中间 B 柱子上,A 柱子只留下最大的那个圆盘,然后移动这个最大的圆盘到 C 柱子上,这时 A 柱子就空了,可以作为中间柱子,所以问题就又变成了移动 $n-1$ 个圆盘从 B 柱子到 C 柱子。也就是做一次 $f(n-1)$,移动 $n-1$ 个圆盘,加上移动一个圆盘,再加上一次 $f(n-1)$,移动 $n-1$ 个圆盘。所以 $f(n)=2f(n-1)+1$;$f(1)=1$。

下面我们仔细研究 $n=3$ 时的移动次数,如图 5-8 所示。有 A、B 和 C 三个柱子,开始时 3 片黄金圆盘(编号为 1,2 和 3)按上小下大的顺序放在柱子 A 上,如图 5-8(a)所示。第一步,将圆盘 1 从 A 移到 C,如图 5-8(b)所示;第二步,将圆盘 2 从 A 移到 B,如图 5-8(c)所示;第三步,将圆盘 1 从 C 移到 B,放在圆盘 2 上,如图 5-8(d)所示;第四步,将圆盘 3 从 A 移到 C,如图 5-8(e)所示;第五步,将圆盘 1 从 B 移到 A,如图 5-8(f)所示;第六步,将圆盘 2 从 B 移到 C,放在圆盘 3 上,如图 5-8(g)所示;第七步,将圆盘 1 从 A 移到 C,放在圆盘 2 上,至此圆盘就全部移完了,如图 5-8(h)所示。经过上述 7 步,可以完成 3 片黄金圆盘的移动,即 $f(3)=7$。

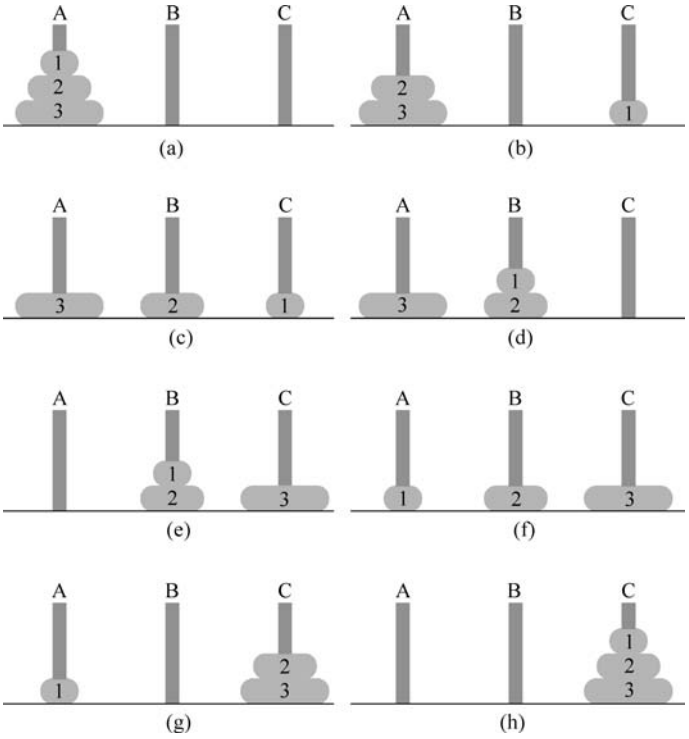


图 5-8　当 $n=3$ 时,将所有圆盘从柱子 A 移动到柱子 C 共要 7 步

总结上面对 3 片圆盘的移动过程。如图 5-8 的(a)～(d)所示,是将上面两片圆盘移到 B,其实就是移动两片圆盘的过程,移动次数是 $f(2)$;而图 5-8 的(d)～(e)是将第 3 片圆盘从

A 移到 C,移动 1 次；图 5-8 的(e)～(h)是将上面两片圆盘移到 C,这也是移动两片圆盘的过程,移动次数是$f(2)$。综上所述,3 片圆盘的移动次数 $f(3)=f(2)+1+f(2)=2f(2)+1$。这样,3 片圆盘的移动次数 $f(3)$可以用两片圆盘的移动次数 $f(2)$来表示。而且,我们也可以知道 $f(2)=2f(1)+1=3$,即两片圆盘的移动次数 $f(2)$可以用 1 片圆盘的移动次数 $f(1)$来表示。

那么 $n(n>3)$片圆盘的移动次数 $f(n)$是不是也可以用 $n-1$ 片圆盘的移动次数 $f(n-1)$来表示呢?

如果想将 $n(n>3)$片圆盘从 A 移到 C,那么必须先将 $n-1$ 片圆盘按上小下大的顺序从 A 移到 B,然后将第 n 片圆盘从 A 移到 C,最后将 $n-1$ 片圆盘从 B 移到 C。因此,$f(n)=2f(n-1)+1$,即 $n(n>3)$片圆盘的移动次数 $f(n)$可以用 $n-1$ 片圆盘的移动次数 $f(n-1)$来表示。这样一来,就将求 $f(n)$的问题变为了求 $f(n-1)$的问题。由此我们可以得到汉诺塔问题的递归式：

$$f(n)=\begin{cases}1, & n=1\\ 2f(n-1)+1, & n>1\end{cases} \tag{5-2}$$

根据递归式(5-2),得到 $f(n)=2f(n-1)+1=2\times(2f(n-2)+1)+1=4f(n-2)+2+1=4\times(2f(n-3)+1)+2+1=8f(n-3)+4+2+1=\cdots=2^n-1$。因此,要将 64 片黄金圆盘从一根柱子移到另一根柱子上,并且始终保持上小下大的顺序,需要移动 $2^{64}-1$(约为 10^{19})次。也许移动 $2^{64}-1$ 次的概念不太直观,那么我们来算一算需要的时间好了。假如移动一次需要一秒,移动完 64 片圆盘需要多久的时间呢？答案是：5845 亿年以上！而地球存在至今不过 45 亿年,宇宙至今也不过 138 亿年左右,即便真的等待 5845 亿年,且不说太阳系和银河系,至少地球上的一切生命,连同梵塔、庙宇等,都早已经灰飞烟灭。

在有了递归式之后,就可以用递归程序得到盘子的移动步骤。我们给出用递归方法解决汉诺塔问题的 Python 代码,大家可以在自己的计算机上试一下。

```
#<程序：汉诺塔_递归>
count = 1
def main():
    n_str = input('请输入盘子个数：')
    n = int(n_str)
    Hanoi(n,'A','C','B')
def Hanoi(n, A, C, B):
    global count
    if n < 1:
        print('False')
    elif n == 1:
        print ("%d:\t%s -> %s" % (count, A, C))
        count += 1
    elif n > 1:
        Hanoi (n - 1, A, B, C)
        Hanoi (1, A, C, B)
        Hanoi (n - 1, B, C, A)
if(__name__ == "__main__"):
    main()
```

如果我们想求将 3 片黄金圆盘从柱子 A 移到柱子 C 的步骤,可以得到下面的结果:

```
>>>
请输入盘子个数: 3
1:A -> C
2:A -> B
3:C -> B
4:A -> C
5:B -> A
6:B -> C
7:A -> C
```

上面的步骤和图 5-8 完全一样。看,就是这么简单。有了递归,计算机科学可以用很简单的几行代码解决汉诺塔问题。那么递归为什么能用很少的代码解决很复杂的问题呢?这就要从递归的定义和本质说起了。

练习题 5.2.3:请用递归求解斐波那契(Fibonacci)数列问题。Fibonacci 为 1200 年代的欧洲数学家,在他的著作中曾经提到:若有一只兔子每个月生一只小兔子,一个月后小兔子也开始生产。起初只有一只兔子,一个月后就有两只兔子,两个月后有三只兔子,三个月后有五只兔子……这就是 Fibonacci 数列,又称黄金分割数列,指的是这样一个数列:1、1、2、3、5、8、13、21、34、55、89、…。问 n 个月后会有多少只兔子?

练习题 5.2.4:请用 Python 编写一个递归程序,求解两个正整数 x 和 y 的最大公约数。

一般来说,递归是一个过程或函数在它的定义或说明中又直接或间接调用它自己的一种方法,例如在解决汉诺塔问题时,函数 Hanoi 调用了它本身。

递归本质是把一个复杂的大问题层层转化为一个与原问题相似的小问题,利用小问题的解来构筑大问题的解。学习用递归解决问题的关键就是找到问题的递归式,有了递归式就可以知道大问题与小问题之间的关系,从而解决问题。例如在解决汉诺塔问题时,通过分析可以将求解 n 个圆盘的移动次数 $f(n)$转化成求解 $n-1$ 个圆盘的移动次数 $f(n-1)$,求解 $n-1$ 个圆盘的移动次数 $f(n-1)$转化成求解 $n-2$ 个圆盘的移动次数 $f(n-2)$……直到转化为求解 1 个圆盘的移动次数 $f(1)$,从而可以得到如公式(5-2)所示的递归式。

递归只需少量的程序就可描述出解题过程所需要的多次重复计算,大大地减少了程序的代码量。它的能力在于用有限的语句来定义无限集合。正是如此,递归才能用很少的代码解决很复杂的问题。然而在使用递归解决问题时要特别注意,**一定要有一个明确的递归结束条件**,否则就会陷入无限循环中。例如在解决汉诺塔问题时,递归结束条件就是 $n=1$。只要判断 $n=1$,就停止调用 Hanoi,开始返回。

> **小明**:无限循环就像我们学过的无限循环小数一样吗?
>
> **阿珍**:差不多,不过影响可不一样了。可以有无限循环小数,但是不能有无限循环的程序。想一下,如果你在解题的时候用了无限循环的程序,那就永远也得不到答案啦,切记!

在本节的开始,我们卖了个关子,让大家找出语言上的递归和图形上的递归与我们计算

机科学中递归的不同。到这里你找到答案了吗？其实就是上面提到的，在计算机科学中使用递归解决问题时，一定会有一个明确的递归结束条件。而语言上的那个递归是没有结束条件的，它可以讲到海枯石烂地老天荒。

接下来是个练习，大家要习惯用递归来解决问题。习惯递归的思想后，可以在很短的时间里写出正确的程序。写递归程序的诀窍就是：①怎么分，怎么合；②怎么终止。

Python 练习：编写 merge(L1,L2)函数：输入参数是两个从小到大排好序的整数列表 L1 和 L2，返回合成后的从小到大排好序的大列表。例如 merge([1,4,5],[2,7])会返回[1,2,4,5,7]，merge([],[2,3,4])会返回[2,3,4]。要求：

(1) 一定要用递归方式；

(2) 只能用列表的 append()和 len()函数。

首先是“怎么分，怎么合”。L1[0]是 L1 列表中最小的，L2[0]是 L2 列表中最小的。比较这两个数，小的数从列表中拿出来，将这个少一个数的列表和另一个列表作为递归调用的参数；得到返回的已经排好序的列表后，将前面拿出来的那个较小的数放在这个返回的列表的最前面；然后再返回这个排好序的列表。

接着是“决定终止条件”，终止条件就是其中一个列表是空的。如果判断其中一个列表是空的，就返回另一个列表。

```
#<程序：merge 函数> by Edwin Sha
def merge(L1,L2):
    if len(L1) == 0:
        return(L2)
    if len(L2) == 0:
        return(L1)
    if L1[0] < L2[0]:
        return([L1[0]] + merge(L1[1:len(L1)],L2))
    else:
        return([L2[0]] + merge(L1,L2[1:len(L2)]))

X = merge([1,4,9],[10])
print(X)
```

小结

递归是计算思维最重要的一种基本思想，是大家在中学没有学习过的。递归是一个过程或函数在它的定义或说明中又直接或间接调用自己的一种思想。它的本质是把一个复杂的大问题层层转化为一个与原问题相似的小问题，利用小问题的解来构筑大问题的解。利用递归思想求解问题时，只需少量的程序就可描述出解题过程所需要的多次重复计算，从而大大地减少程序的代码量。它的能力在于用有限的语句来定义无限集合。正因如此，递归能用很少的代码解决很复杂的问题。学习用递归解决问题的关键就是找到问题的递归式，也就是用小问题的解构造大问题解的关系式。通过递归式可以知道大问题解与小问题解之间的关系，从而解决问题。

5.3 分治法

其实在 5.1 节的找假币的例子中,我们就用到了分治法(Divide-and-Conquer Algorithm)。第三种方式的二分法就是分治法。分治法是我们计算机科学解决问题的一种基本方法,从字面上来理解就是“分而治之”。它的基本思想是把一个复杂的问题分成两个或更多的相同或相似的互相独立的子问题,再把子问题分成更小的子问题,直到最后的子问题可以简单地直接求解,然后将这些子问题的解合并,从而构造出原问题的解。而用分治法求解问题的时候,通常会用到递归的思想来求解子问题。

在具体的介绍分治法之前,先来看一个求最小值的例子。我们会分别用一般的循环比较法、递归比较法和分治比较法求解最小值问题。

求最小值:假设有 n 个数,分别为 $a_1,a_2,a_3,\cdots,a_n$,求 n 个数中的最小值。

想要找到最小值,就需要将 n 个数作比较,但是怎么比较就是关键了。因为不同的比较策略,找出最小值所花费的时间也不同。首先我们来看一个最常用、也是最容易想到的方法。

1. 循环(Loop)比较法

在 n 个数中找出最小值,可以从第一个数 a_1 开始依次做比较。首先比较 a_1 和 a_2,将较小的一个与 a_3 做比较;然后再将较小的一个与 a_4 做比较……直到与 a_n 做比较,找到所有 n 个数中最小的值。

我们可以用循环程序实现上面的策略,用 Python 表示如下。用循环的方法求得最小值共要比较 $n-1$ 次。

```
#<程序:最小值_循环>
def M(a):
    m = a[0]
    for i in range(1,len(a)):
        if a[i]< m:
        m = a[i]
    return m
a = [4,1,3,5]
print(M(a))
```

2. 递归(Recurrence)比较法

除了用循环程序实现上面的策略之外,我们学习计算机科学的人更喜欢用递归的方式实现,因为它简单。这个方法的主要思想是:要求 n 个数中的最小值 $M(n)$,就需要知道 $n-1$ 个数中的最小值 $M(n-1)$,然后比较 a_n 和 $M(n-1)$,较小的就是 $M(n)$;要求 $n-1$ 个数中的最小值 $M(n-1)$,就需要知道 $n-2$ 个数中的最小值 $M(n-2)$,然后比较 a_{n-1} 和 $M(n-2)$,较小的就是 $M(n-1)$……要求两个数中的最小值 $M(2)$,就需要知道 1 个数中的最小值 $M(1)$,然后比较 a_2 和 $M(1)$,较小的就是 $M(2)$,而 1 个数中的最小值 $M(1)$就是它本身 a_1。有了 $M(1)$就可以得到 $M(2)$,有了 $M(2)$就可以得到 $M(3)$,……,有了 $M(n-1)$

就可以得到 $M(n)$，从而得到 n 个数中的最小值。用公式可以表示为：

$$M(n)=\begin{cases}a_1, & n=1\\ \min(a_n,M(n-1)), & n>1\end{cases} \tag{5-3}$$

这种递归比较的方法可以用函数调用来实现。请注意终止条件一定要在函数里面首先设定。在此当数组 a 的长度为 1，返回 a[0]。用 Python 实现如下：

```
#<程序：最小值_递归> a是个数组
def M(a):
    print(a)
    if len(a) == 1: return a[0]
    return (min(a[len(a) - 1], M(a[0:len(a) - 1])))

L = [4,1,3,5]
print(M(L))
```

递归比较和循环比较一样共要比较 $n-1$ 次。

3. 分治(Divide-and-Conquer)比较法

其实我们在做比较的时候，不一定要按照 $a_1,a_2,a_3,\cdots,a_n$ 的顺序来比较，而是可以从任何一个数开始，所得到的结果都是一样的。那么我们可以将这 n 个数分组做比较吗？让 $M(i,j)$表示 $a_i,\cdots,a_j$ 这 $j-i+1$ 个数的最小值，其中 $0\leqslant i\leqslant j\leqslant n-1$。比如将 $a_1,a_2,a_3,\cdots,a_n$ 分成两组：$a_1,\cdots,a_{n/2}$ 和 $a_{n/2-1},\cdots,a_n$，先分别找出它们各自的最小值 $M(1,n/2)$和 $M(n/2-1,n)$，然后比较 $M(1,n/2)$和 $M(n/2-1,n)$，从而得到 n 个数的最小值 $M(1,n)$。这里的除法都是整数除法。显然这种方法也能找到最小值，我们称这种方法为分治比较法。

分治比较法的基本思想是：要求 $M(1,n)$，可以先求得 $M(1,n/2)$和 $M(n/2+1,n)$，$M(1,n/2)$和 $M(n/2+1,n)$中较小的就是 $M(1,n)$；而要求 $M(1,n/2)$，可以先求得 $M(1,n/4)$和 $M(n/4+1,n/2)$，其中较小的就是 $M(1,n/2)$……直到要求 $M(1,1),M(2,2),\cdots,M(n,n)$。而根据 $M(i,j)$的定义可知：$M(1,1)=a_1,M(2,2)=a_2,\cdots,M(n,n)=a_n$。既然知道了 $M(1,1),M(2,2),\cdots,M(n,n)$，通过比较也就可以得到 $M(1,2),M(3,4),\cdots,M(n-1,n)$，…通过比较 $M(1,n/2)$和 $M(n/2+1,n)$，从而得到 $M(1,n)$。按照上述基本思想，可以求得 n 个数中的最小值 $M(1,n)$，用公式可以表示为：

$$M(1,n)=\min(M(1,n/2),M(n/2+1,n)) \tag{5-4}$$

这种分治比较也可以用函数调用来实现。递归函数编程的诀窍如下。

(1) 决定终止条件。以此为例，就是数组只有一个值时，要返回此值。你也可以再加上额外的终止条件使得程序能稍微快点，例如，当数组 a 的长度为 2 时，返回较小的那个值。

```
if  len(a) == 1: return(a[0]);
elif  len(a) == 2: return(min(a[0], a[1]))
```

其实检查数组 a 的长度是否为 2 是没有必要的。

(2) 设定调用的递归函数的参数，也就是大问题要如何分成小问题。(Divide 部分)

(3) 所调用的函数完成后，也就是子问题解决后，如何构建大问题的解答(Conquer 部分)，然后返回此解答。

用 Python 实现如下：

```
#<程序：最小值_分治>
def M(a):
    print(a)     #可以列出程序执行的顺序
    if len(a) == 1: return a[0]
    return (min(M(a[0:len(a)//2]),M(a[len(a)//2:len(L)])))
L = [4,1,3,5]
print(M(L))
```

用这种方法，同样需要比较 $n-1$ 次。但是这种方法可能在多核的情况下要比前两种方法的效率高，大家可以思考下是为什么呢？其实秘诀就在分治比较法的基本思想中。在求 $M(1,n/2)$ 和 $M(n/2+1,n)$ 时所要进行的比较是互不影响的，因此这些比较可以同时进行，进而推广到求 $M(1,2),\cdots,M(n-1,n)$ 时，比较都是可以同时进行的。目前我们所用的计算机都是多核的体系结构，完全可以并行地执行比较操作，这样一来就可以大大地节约时间，因此第三种方法的效率会高于前两种方法。

练习题 5.3.1：上面的分治法的思路是否可以求 $a_0,a_1,a_2,\cdots,a_{n-1}$ 的总和、乘积或者最大数？那减法行吗？运算要符合什么样的运算律才能用分治法？

练习题 5.3.2：请用 Python 分别用递归和分治法求 n 个数 $a_0,a_1,a_2,\cdots,a_{n-1}$ 中的最大值。

通过上面求最小值的例子，相信大家已经对分治法有了一些了解，下面我们会以求最小值和最大值问题为例，详细地为大家讲解分治法。

最小值和最大值(Minimum and Maximum)问题

求最小值和最大值是比较常见的问题，如果是单独地求最小值或者最大值，可以用上面方法。以统计成绩为例，往往需要得到最小值和最大值以确定成绩的分布区间。这时就需要设计某种方法找到 n 个数中的最小值和最大值。

如果用前面的方法分别找最小值和最大值。找最小值要比较 $n-1$ 次，找最大值也要比较 $n-1$ 次，要找到最小值和最大值共需要比较 $2n-2$ 次。但是还可以设计更好的方法，它最多只需要比较 $1.5n-2$ 次就可以同时找到最小值和最大值。这种方法就是用分治法实现的。

问题描述：求 n 个数 $a_1,a_2,a_3,\cdots,a_n$ 中的最小值和最大值。

我们先假设 $n=4$，看看在 a_1,a_2,a_3,a_4 中找最小值和最大值是什么情况。

如果用 5.3 节中的方法分别找最小值和最大值，可以先找最小值，然后再找最大值。首先，将 4 个数分成两份，即 a_1,a_2 和 a_3,a_4；然后，分别比较 a_1 和 a_2，a_3 和 a_4，找到各自的最小值 $M(1,2)$ 和 $M(3,4)$；最后，比较 $M(1,2)$ 和 $M(3,4)$ 找到 4 个数中的最小值 $M(1,4)$。同理找到最大值。

用上述方法，找最小值要比较 3 次，找最大值要比较 3 次，共需比较 6 次。观察上述过程，在找最小值和最大值时存在很多重复的比较。例如，求最小值时要比较 a_1 和 a_2，求最大值时还要比较一次 a_1 和 a_2。如果用上面的方式，每个数都要分别与最小值和最大值做比较，而实际上并不需要这样。

让 Min(i,j)表示 a_i,…,a_j 这 $j-i+1$ 个数的最小值,Max(i,j)表示 a_i,…,a_j 这 $j-i+1$ 个数的最大值,其中 $1\leqslant i\leqslant j\leqslant n$。

同时求最小值和最大值:

在 4 个数中同时找最小值和最大值。首先,将 4 个数分成 2 份,即 a_1,a_2 和 a_3,a_4;然后,比较 a_1 和 a_2,得到最小值 Min(1,2)和最大值 Max(1,2);同理比较 a_3 和 a_4,得到最小值 Min(3,4)和最大值 Max(3,4);最后,分别比较两个最小值和两个最大值:即 Min(1,2)和 Min(3,4)比,Max(1,2)和 Max(3,4)比,从而找个 4 个数中的最小值 Min(1,4)和最大值 Max(1,4)。用上述方法,只需比较 4 次就可以同时找到最小值和最大值。

用分治法同时求 n 个数 a_1,a_2,a_3,…,a_n 中的最小值 Min(1,n)和最大值 Max(1,n)的基本思想是:

(1) 要求 Min(1,n)和 Max(1,n),可以先求得 Min(1,$n/2$)和 Min($n/2+1$,n)以及 Max(1,$n/2$)和 Max($n/2+1$,n),Min(1,$n/2$)和 Min($n/2+1$,n)中较小的就是 Min(1,n),Max(1,$n/2$)和 Max($n/2+1$,n)中较大的就是 Max(1,n)……直到要求 Min(1,1)和 Max(1,1),…,Min(n,n)和 Max(n,n)。

(2) 根据 Min(i,j)和 Max(i,j)的定义可知:Min(1,1)=Max(1,1)=a_1,…,Min(n,n)=Max(n,n)=a_n。

(3) 知道了 Min(1,1)和 Max(1,1),…,Min(n,n)和 Max(n,n),通过分别比较 Min(1,1)和 Min(2,2),Max(1,1)和 Max(2,2)可以得到 Min(1,2)和 Max(1,2),……,直至得到 Min($n-1$,n)和 Max($n-1$,n);同理通过分别比较 Min(1,2)和 Min(3,4)、Max(1,2)和 Max(3,4),可以得到 Min(1,4)和 Max(1,4),……,直至得到 Min($n-3$,n)和 Max($n-3$,n),最终得到 Min(1,n)和 Max(1,n)。

用分治法同时求最小值和最大值所需要的比较次数为 $3n/2-2$。比较次数如图 5-9 所示。从下往上,求第 0 层的最小值和最大值,需要的比较次数为 0;求第 1 层中每组的最小值和最大值要比较 1 次,而要求 $n/2$ 组的最小值和最大值要比较 $n/2$ 次;求第 2 层中每组最小值和最大值要比较 2 次,而要求 $n/4$ 组的最小值和最大值要比较 $2\times n/4=n/2$ 次;求第 3 层中每组最小值和最大值要比较 2 次,而要求 $n/8$ 组的最小值和最大值要比较 $2\times n/8=$

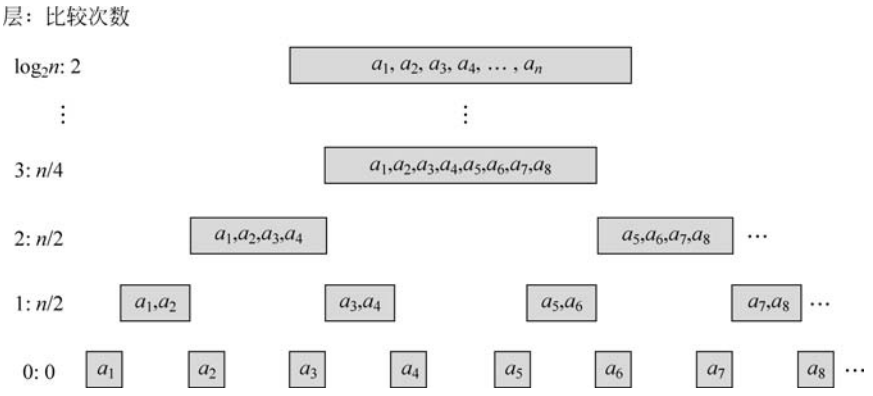


图 5-9 分治法同时求最小值和最大值

$n/4$ 次……求第 $\log_2 n$ 层中每组最小值和最大值要比较两次,而第 $\log_2 n$ 层只有 1 组,因此需要比较两次。所有层的比较次数之和为:$2+4+\cdots+n/4+n/2+n/2=3n/2-2$ 次。

实现同时求最小值和最大值的方法可以 Python 实现,代码如下:

```
#<程序:最小值和最大值_分治>
A = [3,8,9,4,10,5,1,17]
def Smin_max(a):
    if len(a) == 1:
        return(a[0],a[0])
    elif len(a) == 2:
        return(min(a),max(a))
    m = len(a)//2
    lmin,lmax = Smin_max(a[:m])
    rmin,rmax = Smin_max(a[m:])
    return min(lmin,rmin),max(lmax,rmax)

print("Minimum and Maximum: %d, %d" %(Smin_max(A)))
```

运行可以得到:

```
>>>
Minimum and Maximum:1,17
```

在计算机科学中,分治法是非常重要的算法。分治法字面上的解释是“分而治之”,就是把一个复杂的问题分成两个或更多的相同或相似的互相独立的子问题,再把子问题分成更小的子问题,直到最后子问题可以简单地直接求解,原问题的解是子问题解的合并。以分治法求最小值和最大值为例,其基本思想如下。

分:将 n 个数分成两部分,每部分包含 $n/2$ 个数;

治:如果 $n/2=1$,那么可以直接得到最小值和最大值;如果 $n/2=2$,可以直接比较两个数从而得到最小值和最大值;否则,用分治法求 $n/2$ 个数的最小值和最大值;

合并:分别比较两部分的最小值和最大值,从而找到 n 个数的最小值和最大值。

上述分治法求最小值和最大值包括三个步骤:分→治→合并。在“治”中可以看到递归的身影,即如果 $n/2>2$,那么就递归的调用它本身来求最小值和最大值。我们再回过头去看用 Python 的代码。与上述的基本思想相应的,在判断 $n/2$ 不为 1 或 2 之后,调用 Smin_max(array[:m])和 Smin_max(array[m:])。

其实在我们用分治法解题时,往往会用到递归的思想。分治法产生的子问题往往是原问题的较小模式,反复应用分治法,可以使子问题与原问题的类型保持一致,而其规模却不断缩小,最终使子问题缩小到很容易直接求出其解,这就为使用递归提供了方便。在分治法中用递归的思想求解问题是计算机科学解决问题时常用的一种手段,由此也产生了很多高效的算法。

对于求 n 个数的最小值和最大值问题,可能有人认为可以先将这 n 个数排好序,这样最小值和最大值不就一目了然了吗!这种做法是舍近求远,排序的复杂度要比找最大值和最小值复杂得多。我们不需要排序就可以找到最大和最小值了。尤其是分治法常会给出在

多核上可以并行的算法。

用分治法求 n 个数最小值和最大值的时候，将 n 个数分成两部分，然后分别对这两部分求最小值和最大值，即 Min(1,n/2)和 Max(1,n/2)，Min(n/2+1,n)和 Max(n/2+1,n)。而这两个过程是可以并行运算的，因为它们彼此没有依赖关系。同理，求 Min(1,2)和 Max(1,2)的过程，求 Min(3,4)和 Max(3,4)的过程……求 Min(n−1,n)和 Max(n−1,n)的过程都可以相互并行执行。目前计算机已经能够集成越来越多的核，设计并行执行的程序能够有效利用资源，提高对资源的利用率。因此，用分治法求解问题也变得越来越重要。

小明：理解递归函数执行的细节，好像很复杂。我们用递归方式写程序，也是这么复杂吗？

沙老师：递归函数美的地方就是它写起来简单，它将复杂的执行细节都隐藏在执行过程的背后，就是函数调用时栈上的管理。我们设计程序的人不需要在细节上去一步步追踪。我们写递归程序时就是要从上往下，Top-Down 的方式，加上一个终止条件。实在是行云流水，漂亮极了。例如我们写如下的排序程序，几分钟就写出来了。

程序练习 5.3.1：在 5.2 节中，我们展示了 merge(L1,L2)函数。现在利用函数来写一个排序程序。这个算法叫作归并排序(Merge Sort)。给一个列表 L，写 Python 程序 msort(L)，返回一个排好序的列表。这个算法是分治法的典型例子。将 L 分成两半 L1 和 L2，然后调用 msort(L1)、msort(L2)，得出两个排好序的列表 X1 和 X2，最后返回 merge(X1,X2)，是排好序的列表。

```
#<程序：归并排序 Merge Sort>
def msort(L):
    k = len(L)
    if k == 0: return(L)
    if k == 1: return(L)
    X1 = L[0:k//2]; X2 = L[k//2:k] #X1,X2 是局部变量
    print("X1 = ",X1," X2 = ",X2) #看看输出是什么，可以知道递归是如何执行的
    X1 = msort(X1); X2 = msort(X2)
    return(merge(X1,X2))
```

程序练习 5.3.2：在第 2 章中，我们曾经给出了一个用 Python 实现的二进制全加器，结合本节所学的分治法，请研究以下的 Python 程序，分治法实现二进制全加器。

其中代码“c2,s2=add_divide(x[0:len(x)//2],y[0:len(y)//2],c1)”中的 c1，使得这个函数调用一定要在前半部的 add_divide()完成后才能执行，所以不能够并行来执行这两个 add_divide()，要如何改动代码才能并行执行呢？在此假设有足够多的核来做运算。所以可以同时运行 c1=1 和 c1=0 的两种情形，然后再来选择。

```
#<程序：全加器>
def  FA(a,b,c): # Full adder
    carry = (a and b) or (b and c) or (a and c)
    sum = (a and b and c) or (a and (not b) and (not c)) \
        or ((not a) and b and (not c)) or ((not a) and (not b) and c)
    return carry, sum

#<程序：二进制加法-二分法算法> by Edwin Sha
def add_divide(x,y,c = False):
        # x, y are lists of True or False, c is True or False
                # return carry and a list of x + y
    while len(x) < len(y): x = [False] + x
    while len(y) < len(x): y = [False] + y
    if len(x) == 1:
        ctemp, stemp = FA(x[0],y[0],c)
        return (ctemp, [stemp])
    if len(x) == 0: return c, []
    c1,s1 = add_divide(x[len(x)//2:len(x)],y[len(y)//2:len(y)],c)
    c2,s2 = add_divide(x[0:len(x)//2],y[0:len(y)//2],c1) #依赖关系!
    return(c2,s2 + s1)
```

小结

分治法是计算机科学中非常重要的算法，字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的互相独立的子问题，再把子问题分成更小的子问题，直到最后子问题可以简单地直接求解，原问题的解是子问题解的合并。在用分治法求解问题时一般分为三个步骤：分→治→合并。在“治”中往往会用到递归的思想。用分治法求解问题的优势是可以并行地解决相互独立的子问题。目前计算机已经能够集成越来越多的核，设计并行执行的程序能够更有效地利用资源，提高对资源的利用率。因此，用分治法求解问题也变得越来越重要。

5.4 贪心算法

贪心算法(Greedy Algorithm)，又称为贪婪算法，应该算是我们最熟悉、最常用的方法。贪心算法是用来求解最优化问题的一种方法。一般来说，求解最优化问题的过程就是做一系列决定从而实现最优值的过程。最优解就是实现最优值的这些决定。贪心算法考虑局部最优，每次都做当前看起来最优的决定，得到的解不一定是全局最优解。举例而言，例如我们从 A 处到 B 处，要经过许多道路，有不同的路径方案可以选择，想求出最快的路径，假如用贪心算法，选了局部最优的路，但是可能下一条路会很拥堵，这样用贪心算法就无法保证所走的路是最快的路径了。虽然贪心算法不一定能得到最优解，但是有些问题能够用贪心算法求得最优解，例如下面的找零钱问题。

找零钱问题

问题描述：假设有 4 种硬币，它们的面值分别为 2 角 5 分、1 角、5 分和 1 分。现在要找给某顾客 6 角 3 分钱。问怎样找零钱才能使给顾客的硬币个数最少？

一般来说，我们会拿出两个 2 角 5 分的硬币、1 个 1 角的硬币和 3 个 1 分的硬币交给顾客，共找给顾客 6 枚硬币。

这种找零钱的基本思想是：每次都选择面值不超过需要找给顾客的钱的最大面值的硬币。以上面找零钱问题来说：选出一个面值不超过 6 角 3 分的最大面值硬币 2 角 5 分找给顾客，然后还要找 3 角 8 分；选出一个面值不超过 3 角 8 分的最大面值硬币 2 角 5 分找给顾客，然后还要找 1 角 3 分；选出一个面值不超过 1 角 3 分的最大面值硬币 1 角找给顾客，然后还要找 3 分；选出一个面值不超过 3 分的最大面值硬币 1 分找给顾客，然后还要找 2 分；选出一个面值不超过 2 分的最大面值硬币 1 分找给顾客，然后还要找 1 分；最后选出一个面值不超过 1 分的最大面值硬币 1 分找给顾客。这种找硬币的方法实际上就是贪心算法。

用 Python 实现找零钱问题的贪心算法，代码如下：

```
#<程序：找零钱_贪心>
v = [25,10,5,1]
n = [0,0,0,0]
def change():
    T_str = input('要找给顾客的零钱,单位：分：')
    T = int(T_str)
    greedy(T)
    for i in range(len(v)):print('要找给顾客',v[i],'分的硬币：',n[i])
    s = 0
    for i in n:s = s + i
    print('找给顾客的硬币数最少为：',s)

def greedy(T):
    if T == 0:return
    elif T >= v[0]:
        T = T - v[0]; n[0] = n[0] + 1
        greedy(T)
    elif v[0]> T >= v[1]:
        T = T - v[1]; n[1] = n[1] + 1
        greedy(T)
    elif v[1]> T >= v[2]:
        T = T - v[2]; n[2] = n[2] + 1
        greedy(T)
    else:
        T = T - v[3]; n[3] = n[3] + 1
        greedy(T)

if(__name__ == "__main__"):
    change()
```

例如需要找给顾客 63 分(6 角 3 分),可以得到如下结果:

```
>>>
要找给顾客的零钱,单位: 分: 63
要找给顾客 25 分的硬币: 2
要找给顾客 10 分的硬币: 1
要找给顾客 5 分的硬币: 0
要找给顾客 1 分的硬币: 3
找给顾客的硬币数最少为: 6
```

找给顾客两个 2 角 5 分的硬币、1 个 1 角的硬币和 3 个 1 分的硬币,共 6 枚硬币。通过找出所有是 6 角 3 分的硬币组合可以知道,上面贪心算法得到的解是最优解。

对于一些问题,贪心算法能够得到最优解。但是大多数情况下,贪心算法不能得到最优解。例如,我们将上述找零钱问题的硬币面值改为 2 角 5 分、2 角、5 分和 1 分。如果要找给顾客 4 角,利用上述贪心算法会找给顾客 1 枚 2 角 5 分、3 枚 5 分,共 4 枚硬币。可是如果找给顾客两枚 2 角,只要两枚硬币就可以了。

贪心算法虽然不能保证得到最优解,但是它是一种高效的方法。在某些情况下,即使贪心算法不能得到整体最优解,但其最终结果也不会太差,甚至非常近似于最优解。在计算机科学中,有时候可能找不到问题的最佳解决方法,这时可以尝试用贪心算法来求解。虽然可能不是最优解,但也是很有意义的。

我们再来看一个有趣的问题——最大公约数问题。这个问题的求解思路也是用了贪心算法。

最大公约数问题(Greatest Common Divisor,GCD)

问题描述:请写一个程序,求两个正整数 x 和 y 的最大公约数。

最大公约数是指两个或多个整数共有约数中最大的一个。首先,我们要介绍一下最大公约数的一个重要性质:如果 a 是 x 和 y 的最大公约数并且 $x>y$,那么 a 也是 $x-y$ 和 y 的最大公约数。

例如,15 和 10 的最大公约数是 5。15−10=5,而 5 和 10 的最大公约数也是 5。

练习题 5.4.1:请证明 GCD(x,y)=GCD(x,$x-y$),当 $x>y$。

证明:假设 GCD(x,y)=k,那么 $x=ak$, $y=bk$。$x-y=(a-b)k$,所以 GCD(ak,$(a-b)k$)=k。

有了上述性质,利用贪心的思想就可以写出求两个正整数最大公约数的程序了。用贪心的思想解 GCD 的基本思想:用较大值尽可能多地减去较小值,使最后的差是小于较小值的非负整数。应用上述思想,可以得到下述解 GCD 的步骤:

(1) 如果 $x>y$,计算 $x-y$;

(2) 如果 $x-y>y$,令 $x=x-y$,转步骤(1);

(3) 如果 $0<x-y<y$,令 $x=x-y$,交换 x 和 y 的值,转步骤(1);

(4) 如果 $x=y$,y 就是所要求的最大公约数。

其实步骤(1)和(2)的循环计算就是算出 x 除以 y 的余数,也就是 $x\%y$。

练习题 5.4.2:请证明 GCD(x,y)=GCD(x,$x\%y$),当 $x>y$。

用 Python 实现上述贪心算法求解 GCD 的方法,代码如下:

```
#<程序：GCD_贪心>
def main():
    x_str = input('请输入正整数 x 的值：')
    x = int(x_str)
    y_str = input('请输入正整数 y 的值：')
    y = int(y_str)
    print(x,'和',y,'的最大公约数是：', GCD(x,y))

def GCD(x,y):
    if x > y: a = x;b = y
    else:    a = y;b = x
    if a % b == 0: return(b)
    return(GCD(a % b,b))

if(__name__ == "__main__"):
    main()
```

当输入 x＝625，y＝75 时，会得到如下结果：

```
>>>
请输入正整数 x 的值：625
请输入正整数 y 的值：75
625 和 75 的最大公约数是：25
```

小明：我们中学学过怎么求最大公约数，是用因数分解的方式。用取余数的方式求最大公约数有什么好处呢？

沙老师：当 x 和 y 都非常大的时候，用中学学过的方法求最大公约数会变得非常复杂，要花很长很长的时间。而用取余数的方式会简单得多，相关的讨论会在 7.5.1 节给出。在练习题 5.4.3 中，你可以证明每一次 $x\%y$ 的结果都小于 $x/2$，即每一次运算都会使 x 的值减小一半以上，所以此算法收敛的速度非常快。即使 x 和 y 是 100 位的整数，也可以很快地求出它们的最大公约数。

练习题 5.4.3：假设 $x>y$，请证明 $x\%y<x/2$（提示：分别在 $y>x/2$ 和 $y\leqslant x/2$ 两种情况下进行验证）。

小结

贪心算法，又称为贪婪算法，也是用来求解最优化问题的一种方法。一般来说，求解最优化问题的过程就是做一系列决定从而实现最优值的过程。最优解就是实现最优值的这些决定。动态规划考虑全局最优，得到的解一定是最优解。贪心算法是一种在每一步选择中都采取当前状态下最好或最优（即最有利）的选择，从而希望导致结果是最好或最优的算法。贪心算法考虑局部最优，每次都做当前看起来最优的决定，得到的解不一定是全局最优解。但是在有最优子结构的问题中，贪心算法能够得到最优解。最优子结构的局部最优解能决定全局最优解。简单地说，问题能够分解成子问题来解决，子问题的最优解能递推到最终问

题的最优解。虽然对于很多问题贪心算法不一定能得到最优解,但是它的效率高,所求得的答案比较接近最优结果。因此,贪心算法可以用作辅助算法或者直接解决一些对结果的精确度要求不高的问题。

5.5 动态规划

在 5.3 节用分治法求解问题时,待解问题要能够被分成相互独立的子问题。也就是说,这些子问题的解是相互没有关系的,例如最小值和最大值的问题,求解 Min(1,$n/2$)和 Max(1,$n/2$)与求解 Min($n/2+1$,n)和 Max($n/2+1$,n)互不影响。

在这一节会学习一种新的解题方法——动态规划(Dynamic Programming)。动态规划与分治法类似,其基本思想也是将待求解问题分解成若干个子问题,先求解子问题,然后从这些子问题的解得到原问题的解。与分治法不同的是,适合于用动态规划求解的问题,经分解得到子问题往往不是互相独立的,即子问题之间具有重叠的部分。在这种情况下,如果用分治法求解就会重复地求解这些重叠的部分。而动态规划只会对这些重叠的部分求解一次并用表格保存这些解,如此一来就可以避免大量的重复计算。为了清楚的说明问题,请看下面的例子。

最长递增子序列问题

问题描述:已知有 n 个数的序列 L,求它的最长递增子序列的长度。假设序列 L 的一个递增子序列为$[a_1,a_2,\cdots,a_k]$,这些数必须满足 $a_1<\cdots<a_i<\cdots<a_j<\cdots<a_k$($1\leqslant i<j\leqslant k$),而最长递增子序列就是所有递增子序列中长度最大的那个。例如序列 L=[5,2,4,7,6,3,8,9]的最长递增子序列是[2,4,7,8,9],其长度是 5。

根据计算思维求解问题的基本思路:首先将原问题分解成小问题,再用小问题的解构筑原问题的解。因此,我们需要考虑的是"怎么分,怎么合"的问题。最长递增子序列问题的"怎么分"就是考虑怎么将 n 个数的最长递增子序列问题划分成 $n-1$ 个数的最长递增子序列问题;"怎么合"就是考虑怎么用 $n-1$ 个数的最长递增子序列问题的解构筑 n 个数的最长递增子序列问题的解。

我们可以尝试不同的分法,然后验证这种分法是否能够正确地构筑出原问题的解。

第一种方法(这是错误的方法):

最直观地,以待求解的问题进行分解,定义 Asc(i)是 i 个数的序列$[a_1,a_2,\cdots,a_i]$的最长递增子序列的长度。

例如对于序列 L=[5,2,4,7,6,3,8,9],Asc(1)是序列$[a_1]$的最长递增子序列的长度,而 Asc(1)=1。

用 $x(i)$表示这个最长递增子序列中最大值的索引。例如$[a_1]$的最长递增子序列就是它自己,因此 $x(1)=1$,也就是 a_1 是这个最长递增序列的最大值。假设已知 Asc($n-1$),考虑能否用 Asc($n-1$)构造出 Asc(n)。

验证:

如果 $a_n>a_{x(n-1)}$,可以将 a_n 放入 $n-1$ 个数的最长递增子序列的最后,这样就可以形成一个递增序列,而这个递增序列就是 n 个数的最长递增子序列,因此 Asc(n)=Asc($n-1$)+1;

如果 $a_n=a_{x(n-1)}$,那么情况就比较复杂了。例如,如果序列 L=[1,3,5,5],已知[1,3,

5]的最长递增子序列是[1,3,5],其长度 Asc(3)=3,最大值的索引 $x(3)=3$,此时 $a_4=a_3$,L 的最长递增子序列就是[1,3,5],即 Asc(4)=Asc(3);如果序列 L=[1,3,5,2,3,5],已知[1,3,5,2,3]的最长递增子序列是[1,3,5]或[1,2,3],假如我们记录的是[1,3,5],其长度 Asc(5)=3,最大值的索引 $x(5)=3$,此时 $a_6=a_3$,但是 L 的最长递增子序列是[1,2,3,5],并不是由[1,3,5,2,3]的最长递增子序列构筑而成的。

因此,第一种方法不能正确构筑出原问题的解,我们需要重新考虑划分方式。

受第一种方法的启示,最长递增子序列中的最大值是非常重要的信息。而序列 L=$[a_1,a_2,\cdots,a_n]$的最长递增子序列可能以 $a_i(1\leqslant i\leqslant N)$为最大值。

第二种方法:

用以 a_i 为最大值的最长递增子序列这个启示,我们定义 Asc(i)是以 $a_i(1\leqslant i\leqslant n)$为最大值的最长递增序列的长度。

例如对于序列 L=[5,2,4,7,6,3,8,9],Asc(1)就是以 a_1 为最大值的最长递增序列的长度,这个序列是[5],因此 Asc(1)=1;Asc(2)就是以 a_2 为最大值的最长递增序列的长度,这个序列是[2],因此 Asc(2)=1;Asc(3)就是以 a_3 为最大值的最长递增序列的长度,这个序列是[2,4],因此 Asc(3)=2;Asc(4)就是以 a_4 为最大值的最长递增序列的长度,这个序列是[2,4,7],因此 Asc(4)=3;Asc(5)就是以 a_5 为最大值的最长递增序列的长度,这个序列是[2,4,6],因此 Asc(5)=3。接下来要怎么求出 Asc(6)呢?已知 $a_6=3$,首先我们记录序列 L 中所有比 3 小的值的索引,并存储在集合 X。这里 $X=\{2\}$,即 a_2 比 3 小。对于集合 X 里的所有 x,Asc(6)等于最大的 Asc(x)加 1。所以 Asc(6)=Asc(2)+1=2。同理,Asc(7)=4,Asc(8)=5。

根据上面的“分”的方式,求最长递增子序列的问题可以转化为求最大的 Asc(i)的问题,即:

$$\text{最长子序列的长度}=\text{Max}(\text{Asc}(i))(1\leqslant i\leqslant n) \tag{5-5}$$

假设已知 Asc($k-1$),…,Asc(1),考虑是否能用 Asc($k-1$),…,Asc(1)构造 Asc(k)。

验证:

假设已知 Asc(i),如果 $a_k>a_i(1\leqslant i\leqslant k-1)$,那么将 a_k 加到以 a_i 为最大值的最长递增序列的后面,就可以构造一个递增序列,而这个递增序列的长度就是 Asc(i)+1。所以对所有的 $a_i<a_k(1\leqslant i\leqslant k-1)$,找出 max(Asc($i$)),然后加 1 就是了。例如对于序列 L=[5,2,4,7,6,3,8,9],已知 Asc(2)=1,且以 a_2 为最大值的最长递增序列是[2]。由于 $a_3>a_2$,可以将 4 放入序列[2]的最后,形成递增序列[2,4],这个序列的长度是 Asc(2)+1=2。注意,可能有多于一个 i 赋予了相同的 max(Asc(i)),我们取任意一个都可以。在我们的程序里,我们记录的是最小的 index i 赋予 max(Asc(i))。

注意,Python 程序的序列下标索引是从 0 开始的,即对于序列$[a_0,a_1,\cdots,a_n]$的递归式。下标以 0 开始是我们计算机科学中默认的,前面的下标以 1 开始是为了简化说明。以 0 开始或者以 1 开始对上面的性质是没有影响的。

用数学式表达即:

$$\text{Asc}(k)=\begin{cases}1, & k=0\text{ 时}\\ \max(\text{Asc}(i))+1, & \forall i\ (1\leqslant i\leqslant k-1)\text{ 且 }a_k>a_i\end{cases} \tag{5-6}$$

根据公式(5-6),用 Asc($k-1$),…,Asc(0)构造 Asc(k),计算完所有的 Asc(k)后,取最大值就是最长递增子序列的长度了。

在解决最长递增子序列问题的时候,我们可以用前面已经解决的 Asc(0),Asc(1),…,Asc($n-1$)构造后面的 Asc(n)的解。因此,可以将 Asc(0),Asc(1),…,Asc($n-1$)用一个表格保存起来,这样在解决后面问题的时候就不用重复计算了,从而提高解题的速度。根据公式(5-6),已知 Asc(0),Asc(1),…,Asc($n-1$),最多比较 n 次就能得到 Asc(n)。所以求解所有的 Asc(0),Asc(1),…,Asc(n),最多只要比较 $n(n+1)/2$ 次。因此,用这种方法解决最长递增子序列问题是很有效率的。

同时,为了得到这个最长递增子序列,我们用 Tra(i)($0 \leqslant i \leqslant n$)记录 Asc($i$)的生成过程。例如对于序列 L=[5,2,4,7,6,3,8,9],Asc(2)是通过 Asc(1)+1 得到的,因此 Tra(2)=1。

应用上述方法,求解序列 L=[5,2,4,7,6,3,8,9]的最长递增子序列。根据公式(5-6),可以得到如表 5-1 所示的 Asc 和 Tra。

表 5-1　Asc 和 Tra

i	0	1	2	3	4	5	6	7
a_i	5	2	4	7	6	3	8	9
Asc(i)	1	1	2	3	3	2	4	5
Tra(i)	−1	−1	1	2	2	1	3	6

如表 5-1 所示,Asc(7)是 Asc 中的最大值,因此序列 L 的最长递增子序列的长度为 Asc(7)=5。

但是到这里还没有结束,我们还要回溯 Asc 的生成过程,得到这个最长递增子序列。如表 5-1 所示的 Tra 就是用来记录 Asc 生成过程的列表。

由 Asc(7)=5,根据 Asc 的定义可知,这个最长递增子序列的最后一个元素是 a_7;根据 Tra(7)=6 可知,Asc(7)是由 Asc(6)+1 得到的,因此 a_7 前面的元素是 a_6;根据 Tra(6)=3 可知,Asc(6)是由 Asc(3)+1 得到的,因此 a_6 前面的元素是 a_3;根据 Tra(3)=2 可知,Asc(3)是由 Asc(2)+1 得到的,因此 a_3 前面的元素是 a_2;根据 Tra(2)=1 可知,Asc(2)是由 Asc(1)+1 得到的,因此 a_2 前面的元素是 a_1;Tra(1)=−1 代表 a_1 是这个递增子序列的第一个元素。因此这个最长递增子序列为:[a_1,a_2,a_3,a_6,a_7],即[2, 4, 7, 8, 9]。

实现最长递增子序列问题的 Python 程序如下:

```
#<程序: 最长递增子序列_动态规划>
def LIS(L): #LIS (L): Longest Increasing Sub-list of List L
    Asc = [1] * len(L);Tra = [-1] * len(L) #设定起始值
        #Asc[i] 存放从 L[0]到 L[i]以 L[i]为最大值的最长递增子序列的长度,
        #        这个最长数列肯定以 L[i]结尾
        #Tra[i] 存放此最长数列的前一个索引,以便最后连起整个递增序列
    for i in range(1,len(L)):
        X = []
        for j in range(0,i):
            if L[i] > L[j]: X.append(j) #所有比 L[i]小 L[j]的索引存放在 X
        for k in X:   #Asc[i] = max Asc[k] + 1, for each k in X
```

```
                if Asc[i] < Asc[k] + 1: Asc[i] = Asc[k] + 1; Tra[i] = k
        print("Asc:",Asc)
        print("Tra:",Tra)
        max = 0      #找到 Asc 中的最大值
        for i in range(1,len(Asc)):
            if Asc[i]> Asc[max]: max = i
        print("最长递增子序列的长度是",Asc[max])

        #将最长递增数列存到 X
        X = [L[max]]; i = max;
        while (Tra[i] >= 0):
            X = [L[Tra[i]]] + X
            i = Tra[i]
        print("最长递增子序列 = ",X)

    L = [5,2,4,7,6,3,8,9]
    LIS(L)
```

运行上述程序，可以得到如下结果：

```
>>>
Asc: [1, 1, 2, 3, 3, 2, 4, 5]
Tra: [ - 1,  - 1, 1, 2, 2, 1, 3, 6]
最长递增子序列的长度是 5
最长递增子序列 = [2, 4, 7, 8, 9]
```

解决最长递增子序列问题的基本思路：首先，通过对原问题的分析将最长递增子序列问题转化成为求 Asc(i)($0 \leqslant i \leqslant n$)的小问题；其次，找出小问题之间的关系并建立如公式(5-6)所示的递归式；然后，根据公式(5-6)和已知条件，计算 Asc(i)并保存，同时保存 Asc(i)的生成过程 Tra(i)；最后，比较所有的 Asc(i)，找出最大值，并通过 Tra(i)找出最长递增子序列。

上述解决最长递增子序列问题的方法叫作动态规划。动态规划是求解最优化问题的一种方法。例如最长递增子序列问题中，我们可以找到很多的递增子序列，每一个递增子序列都对应一个长度，最长递增子序列问题是要找到长度最大的那个递增子序列。动态规划的方法是找到递归关系，全局解是用局部解来完成的。然后在计算一个个局部解后，用表格来存放，这样就不会重复计算这些局部解了。

在此总结一下：计算 Asc(k)的时候，要用到 Asc(i)($0 \leqslant i \leqslant k$)，这就是递归的关系！我们可以把 Asc($k$)当作一个函数来直接编程：def Asc($k$)。这样编写的程序是正确的，但是执行起来非常慢，因为在计算 Asc(2)时需要计算 Asc(1)，而在计算 Asc(3)时，又要重复计算 Asc(2)和 Asc(1)函数，重复计算的次数是很惊人的。动态规划是用“表格”从小到大来记录已经算过的 Asc(i)，这样就不会重复计算已经算过的 Asc(i)了。大家可以试试运行以下直接用递归方式编程的 Python 程序，是不是很慢？

```
#<程序：直接用递归函数计算 Asc(k)>
def Asc(k):
    if k == 0: return(1)
    X = []
    for i in range(0,k):
        if L[k] > L[i]: X.append(Asc(i))      #记录所有比 L[k]小的 Asc()
    if len(X) > 0: return (max(X) + 1)
    else: return(1)

def LIS_R(L):
    X = []
    for k in range(0, len(L)):
        X.append(Asc(k))
    print(X)
    print(max(X))

L = [5,2,4,7,6,3,8,9]
LIS_R(L)
L = list(range(1,31)) #L = [1,2,3,4,…,29,30]
LIS_R(L)
```

当序列已经是递增序列时，对这个程序而言是最坏的情况，也就是它在计算 Asc(n)时要计算 Asc($n-1$)，Asc($n-2$)，…，Asc(1)，Asc(0)。假设 $T(n)$是计算 Asc(n)调用所有 Asc(i)函数的次数，则 $T(n)=T(n-1)+T(n-2)+\cdots+T(1)+T(0)$且 $T(0)=1$，你们知道这个 $T(n)$的增长速度有多快吗？你可以用数学来证明 $T(n)=2^{n-1}$。这就是为什么我们前面的递归程序运行非常慢，当 $n=30$ 的时候，执行时要调用 Asc 函数 2^{29} 次，所以用时如此之长啊。

使用动态规划求解问题，是用递归函数来定义问题，但是编程时不直接用递归函数，而是用表格从小到大地记录递归函数的结果。

一般可以分为以下几个步骤。

(1) 定义递归函数(其实是用表格实现)。

(2) 递归函数要如何算出来(如何用前面的表格单元来计算表格第 i 个单元)？

(3) 用第(2)步中计算过程的信息构造最优解。

(4) 整个问题最优解的值如何从表格求出。

以最长递增子序列问题为例：

(1) 定义递归的结构 Asc(i)，即 Asc(i)是以第 i 个数 a_i($1\leqslant i\leqslant n$)为最大值的最长递增子序列的长度。

(2) 计算 Asc(k)，即 Asc(k)=max(Asc(i))+1，$\forall a_k>a_i$($0\leqslant i\leqslant n-1$)。

(3) 按照 Asc(0)，Asc(1)，…，Asc(n)这种自底向上的方式计算 Asc。根据 Asc 的生成过程信息 Tra 构造最优解。

(4) 求出所有的 Asc(i)后，整个问题的答案是 max(Asc(i)，$0\leqslant i\leqslant n-1$)。

练习题 5.5.1：证明：当 $T(n)=T(n-1)+T(n-2)+\cdots+T(1)+T(0)$且 $T(0)=1$ 时，$T(n)=2^{n-1}$。

练习题 5.5.2：在我们的例子中，[2，4，7，8，9]和[2，4，6，8，9]都是最优解，如何修

改我们的程序，使得输出结果是[2，4，6，8，9]？

练习题 5.5.3：(讨论题)可不可能利用第一种方法，加以改进，设计出正确的算法？

通过上面的例子，你是否对动态规划求解问题有了一些了解呢？我们再给出一个能够用动态规划求解的例子。

三角数塔问题

问题描述：图 5-10 是一个由数字组成的三角形，请找出一条从上到下的路径，使这条路径上的数值之和最大。

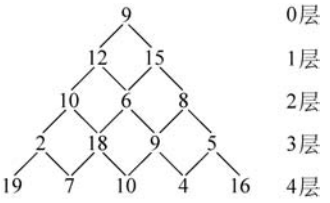


图 5-10　三角数塔

首先定义递归结构，将问题转化成较小问题，也就是考虑“怎么分，怎么合”的问题。

如果在找该路径时，从上到下走到了第 3 层第 0 个数(也就是 2)，那么接下来必然选择走 19。如果从上到下走到了第 3 层第 1 个数(也就是 18)，那么接下来必然选择走 10。如果从上到下走到了第 3 层第 2 个数(也就是 9)，那么接下来必然选择走 10。同理，如果从上到下走到了第 3 层第 3 个数(也就是 5)，那么接下来必然选择走 16。根据这个思路可以更新第 3 层的数，即把 2 更新成 21(2＋19)，把 18 更新成 28(18＋10)，把 9 更新成 19(9＋10)，把 5 更新成 21(5＋16)。更新后的三角数塔如图 5-11 所示。

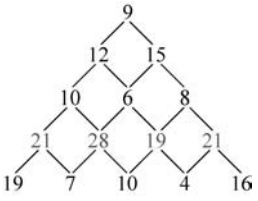
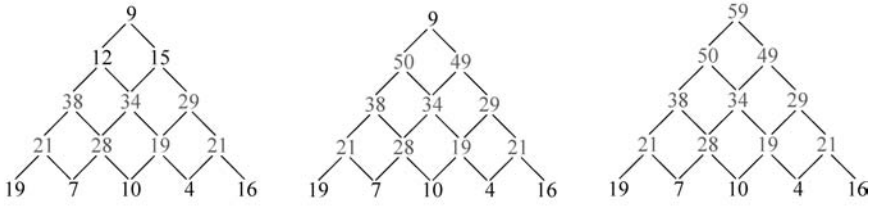


图 5-11　更新第 3 层后的数塔

根据上面的思路，可以相继将第 2 层、第 1 层和第 0 层更新，如图 5-12 所示。



(a) 更新第2层后的数塔　(b) 更新第1层后的数塔　(c) 更新第0层后的数塔

图 5-12　相继更新第 2 层、第 1 层和第 0 层后的数塔

定义 $a(i,j)$ 为从第 i 层第 j 个数到最底层的所有路径中最大的数值之和。

在本例中，第 4 层是最底层。用 5×5 二维数组 T 存储数塔的初始值，根据上面的思

路,$a(3,0)$等于 $a(4,0)$和 $a(4,1)$中较大的数值加上 T[3,0]; $a(3,1)$等于 $a(4,1)$和 $a(4,2)$中较大的数值加上 T[3,1]; $a(3,2)$等于 $a(4,2)$和 $a(4,3)$中较大的数值加上 T[3,2]; $a(3,3)$等于 $a(4,3)$和 $a(4,4)$中较大的数值加上 T[3,3]。由此,可以得到如下递归式:

$$a(i,j)=\begin{cases}\mathrm{T}[i,j], & i=4\text{ 时}\\ \max(a(i+1,j),a(i+1,j+1))+\mathrm{T}[i,j], & \forall i(0\leqslant i<4)\text{ 且 }j\leqslant i\end{cases}\tag{5-7}$$

假设数塔有 n 层,那么最底层就是第 $n-1$ 层,即 $i=n-1$ 时,$a(i,j)=\mathrm{T}[i,j]$。有了如公式(5-7)的递归式,就知道如何计算 $a(i,j)$了,也就解决了如何计算递归函数的问题。根据自底向上的方式,先计算最底层的 $a(n-1,0)$,$a(n-1,1)\cdots a(n-1,n-1)$,然后计算 $a(n-2,0)$,$a(n-2,1)\cdots a(n-2,n-2)$,…直到计算最顶层的 $a(0,0)$。根据公式(5-7)可以生成本例的动态规划表,如表 5-2 所示,其中 $a(0,0)$就是最大的数值之和 59。

表 5-2　三角数塔的动态规划表

j	i				
	0	1	2	3	4
4	19	7	10	4	16
3	21	28	19	21	0
2	38	34	29	0	0
1	50	49	0	0	0
0	59	0	0	0	0

接下来用回溯的方法找出最大数值之和的路径。首先从 $a(0,0)=59$ 开始,$a(0,0)-\mathrm{T}[0,0]=59-9=50$,即 $a(0,0)$是通过 T[0,0]加上 $a(1,0)$得到的;回溯到 $a(1,0)=50$,$a(1,0)-\mathrm{T}[1,0]=50-12=38$,即 $a(1,0)$是通过 T[1,0]加上 $a(2,0)$得到的;回溯到 $a(2,0)=38$,$a(2,0)-T(2,0)=38-10=28$,即 $a(2,0)$是通过 T[2,0]加上 $a(3,1)$得到的;回溯到 $a(3,1)=28$,$a(3,1)-\mathrm{T}[3,1]=28-18=10$,即 $a(3,1)$是通过 T[3,1]加上 $a(4,2)$得到的。从而得到路径为:(0,0),(1,0),(2,0),(3,1),(4,2)。回溯的路线如表 5-2 中的箭头所示。

上述过程就是用动态规划求解三角数塔问题的步骤。用 Python 实现三角数塔问题的动态规划算法,代码如下:

```
#<程序:三角数塔问题_动态规划>
def TriNumPagoda():
    N = int(input('请输入数塔的层数:'))
    P = [[]for i in range(N)]
    for i in range(N):
        L = []
        S = input('请输入' + str(i + 1) + '个数:')
        L = S.split(sep = ',')
        for a in L:
            P[i].append(int(a))
        if len(L)> i + 1:
```

```
            print('输入数值的个数不正确!')
            return
    D = [[]for i in range(N)]              #初始化动态规划表 D
    for i in range(N):
        for j in range(i + 1):
            D[i].append(P[i][j])
    for i in range(N - 2, - 1, - 1):       #生成动态规划表
        for j in range(i + 1):
            if D[i + 1][j] + P[i][j]> = D[i + 1][j + 1] + P[i][j]:
                D[i][j] = D[i + 1][j] + P[i][j]
            else:
                D[i][j] = D[i + 1][j + 1] + P[i][j]
    print('最大数值之和为:',D[0][0])
    path = []                              #记录路径
    path.append('(0,0)')
    m = 0
    Max = D[0][0]
    for i in range(1,N):                   #回溯动态规划表,找到路径
        if (Max - P[i - 1][m]) == D[i][m]:
            Max = D[i][m]
        else:
            Max = D[i][m + 1]
            m = m + 1
        path.append('(' + str(i) + ',' + str(m) + ')')
    print('路径为:',path)
if(__name__ == "__main__"):
    TriNumPagoda()
```

运行上面的程序,可以得到如下结果:

```
>>>
请输入数塔的层数:5
请输入 1 个数:9
请输入 2 个数:12,15
请输入 3 个数:10,6,8
请输入 4 个数:2,18,9,5
请输入 5 个数:19,7,10,4,16
最大数值之和为: 59
路径为:['(0,0)', '(1,0)', '(2,0)', '(3,1)', '(4,2)']
```

接下来我们再给一个例子,这个例子的递归关系是二维的,也就是需要构建二维的表格,这是著名的背包问题(Knapsack Problem)。我们思考要如何用动态规划的方法来解这个问题。首先是问题的定义。

背包问题

背包问题是在 1978 年由 Merkel 和 Hellman 提出的一种组合优化问题。我们以小偷

偷东西为例：一个小偷撬开了一个保险箱，保险箱中有 5 种物品，每种物品的重量不同、价值也不同，物品的重量与价值如表 5-3 所示，小偷的背包只能负重 8kg，要怎样才能使偷走的物品总价值最大？

表 5-3　物品的重量和价值

物品	编号 i	重量 $w(i)$/kg	价值 $v(i)$/万元
A	1	4	45
B	2	5	57
C	3	2	22
D	4	1	11
E	5	6	67

如果你是小偷，你会偷走哪些物品，从而使总价值最大？

有一种最简单的方法，就是找出所有能够放入背包使得总重量不超过 $W=8$kg 的物品组合。这样的组合可以找到{A}、{B}、{C}、{D}、{E}、{A,C}、{A,D}、{B,C}、{B,D}、{C,E}、{D,E}、{A,C,D}和{B,C,D}，其中能得到最大总价的是最后一种组合，它的总价 $V=90$ 万元。上述方式虽然可以找到最大总价的物品组合，可是如果有 n 个物品，就会有 2^n-1 种组合，要在 2^n-1 种组合中找到价值最大的组合，显然是非常耗时的。

根据计算思维的解题思路，我们需要考虑的是“怎么分，怎么合”的问题。也就是设计递归关系，看这个问题是否可以转换成比较小的问题。最直观的是，n 个物品的背包问题能否转换成 $n-1$ 个物品的背包问题，我们首先将 n 个物品用索引从 1 到 n 来指定。

定义 $a(i,j)$为：考虑前 i $(1\leqslant i\leqslant n)$个物品，能够装入容量为 j 的背包的物品组合所形成的最大价值。

假设已经知道前 $n-1$ 个物品的最优解 $a(n-1,j)$，要求 n 个物品的最优解 $a(n,j)$，会出现以下几种情况。

(1) 当背包的容量 j 大于或等于第 n 个物品的重量 $w(n)$，即 $j-w(n)\geqslant 0$。我们可以考虑放进第 n 个物品，在这种情况下，背包的总价值为 $a(n-1,j-w(n))+v(n)$。

(2) 当背包的容量 j 大于或等于第 n 个物品的重量 $w(n)$，我们可以考虑不放进第 n 个物品到背包。在这种情况下，背包的总价值为 $a(n-1,j)$。

(3) 背包的容量 j 小于第 n 个物品的重量 $w(n)$，第 n 个物品不能放入背包。在这种情况下，背包的总价值为 $a(n-1,j)$。

当 $j\geqslant w(n)(1\leqslant i\leqslant n)$，背包的最大价值应该为 $a(n-1,j-w(n))+v(n)$和 $a(n-1,j)$中较大的值。当 $j<w(n)$，背包的最大价值应该为 $a(n-1,j)$。

通过上述分析，可以得到背包问题的递归式如下：

$$a(i,j)=\begin{cases}0, & i=0 \text{ 或 } j=0\\ a(i-1,j), & j<w(i)\\ \max(a(i-1,j),a(i-1,j-w(i))+v(i)), & j\geqslant w(i)\end{cases} \tag{5-8}$$

假设有 n 个物品，背包可承受 m 千克的重量，那么整个问题的最佳解就是 $a(n,m)$的值了。

有了如公式(5-8)的递归式之后，就可以用递归的方法解决背包问题了。用递归求解背

包问题的 Python 代码如下：

```
#<程序：背包问题_递归>
w = [0,4,5,2,1,6]          #w[i]是物品的重量
v = [0,45,57,22,11,67]     #v[i]是物品的价值
n = len(w) - 1
j = 8                      #背包的容量
x = [False for raw in range(n + 1)]#x[i]为 True,表示物品被放入背包
def knap_r(n,j):
    if (n == 0)or(j == 0):
        x[n] = False
        return 0
    elif (j >= w[n])and(knap_r(n - 1,j - w[n]) + v[n]> knap_r(n - 1,j)):
        x[n] = True
        return knap_r(n - 1,j - w[n]) + v[n]
    else:
        x[n] = False
        returnknap_r(n - 1,j)
print("最大价值为：",knap_r(n,j))
print("物品的装入情况为：",x[1:])
```

但是用递归来解决这个问题是很耗时的，为什么呢？在用递归实现的代码中，knap_r(n,j)函数中有 3 次调用自身。在前两次函数调用中，要计算 knap_r(n－1,j－w[n])＋v[n]和 knap_r(n－1,j)，而在第 3 次调用时还要重新计算 knap_r(n－1,j－w[n])＋p[n]或者 knap_r(n－1,j)。显然这里存在很多重复计算。

为了避免这些重复计算，下面我们用动态规划来解决背包问题。根据动态规划的基本思想，有了递归式，接下来就是建立动态规划表了。根据公式(5-8)可以生成本例的动态规划表，如表 5-4 所示，其中 $a(5,8)$就是所能得到的最大价值 90。

表 5-4　装物品的动态规划表

i	j								
	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	45	45	45	45	45
2	0	0	0	0	45	57	57	57	57
3	0	0	22	22	45	57	67	79	79
4	0	11	22	33	45	57	68	79	90
5	0	11	22	33	45	57	68	79	90

接下来我们用回溯的方法找出背包里装入的物品。首先从最大价 $a(5,8)=90$ 开始，在判断物品 E 时，由于 $8\geqslant w(5)$，而 $a(4,8)$较大，则 E 没有放入背包；回溯到 $a(4,8)=90$，在判断物品 D 时，由于 $8\geqslant w(4)$，而 $a(3,7)+11$ 较大，则 D 被放入背包；回溯到 $a(3,7)=79$，在判断物品 C 时，由于 $7\geqslant w(3)$，而 $a(2,5)+22$ 较大，则 C 被放入背包；回溯到 $a(2,5)=57$，在判断物品 B 时，由于 $5=w(2)$，而 $a(1,0)+57$ 较大，则 B 被放入背包；回溯到 $a(1,0)=0$，则物品 A 没有放入背包。从而得到被放入背包的物品是：B，C 和 D。回溯的路线如

表 5-4 中的箭头所示。

用 Python 实现背包问题的动态规划算法,代码如下:


```
#<程序:背包问题_动态规划>
w = [0,4,5,2,1,6]              #w[i]是物品的重量
v = [0,45,57,22,11,67]         #v[i]是物品的价值
n = len(w) - 1
m = 8                          #背包的容量
x = [False for raw in range(n + 1)]#x[i]为 True,表示物品被放入背包
#a[i][j]是 i 个物品中能够装入容量为 j 的背包的物品所能形成的最大价值
a = [[0 for col in range(m + 1)] for raw in range(n + 1)]
def knap_DP(n,m):
    #创建动态规划表
    for i in range(1,n + 1):
        for j in range(1,m + 1):
            a[i][j] = a[i - 1][j]
            if (j >= w[i]) and(a[i - 1][j - w[i]] + v[i]> a[i - 1][j]):
                a[i][j] = a[i - 1][j - w[i]] + v[i]
    #回溯 a[i][j]的生成过程,找到装入背包的物品
    j = m
    for i in range(n,0, - 1):
        if a[i][j]> a[i - 1][j]:
            x[i] = True
            j = j - w[i]
    Mv = a[n][m]
    return Mv
print("最大价值为: ",knap_DP(n,m))
print("物品的装入情况为: ",x[1:])
```

运行程序,可以得到:

```
>>>
最大价值为: 90
物品的装入情况为: [False, True, True, True, False]
```

比较上面两个程序,其实它们之间的不同就在于是否建立动态规划表。用动态规划实现的代码中,首先建立了动态规划表,这样对于已经计算过的 a(i,j)就不需要进行重复计算了,从而减少程序的运行时间。其实动态规划算法是一种以空间换取时间的算法,它将可能会重复用到的数据保存起来,后面一旦要使用这些数据,只要去表中查找即可。

动态规划通常用于求解具有某种最优性质的问题。它也是将待求解问题分解成若干个子问题,先求解子问题,然后从这些子问题的解得到原问题的解。动态规划分解得到的子问题并不是相互独立的,因此在求解子问题时,可以利用已经求解的子问题的解来构造待求解的子问题的解。如此一来,通过将已经求解的子问题的解保存起来,在求解后面的子问题时就能省掉很多重复计算。

练习题 5.5.4:在 5.4 节我们用贪心算法求解找零钱问题,这个问题也可以用动态规划求解。请写出用动态规划求解找零钱问题的基本思想。

动态规划求解找零钱问题的基本思想：已知有 4 种硬币，我们以它们的面值从小到大排序，如表 5-5 所示。求找给顾客 63 分的最少硬币个数。

根据动态规划求解问题的基本思想，我们首先将原问题划分成小问题。原问题是在 4 种面值的硬币中，找给顾客 63 分钱的最少硬币数。根据原问题，我们定义小问题 $a(i,j)$：$a(i,j)$表示在 i 种面值的硬币中找给顾客 j 分钱的最少硬币数。

表 5-5　硬币种类和面值

硬币种类 i	面值 $v(i)$/分
1	1
2	5
3	10
4	25

程序练习题 5.5.1：请编写用动态规划求解找零钱问题的 Python 程序。

小结

动态规划是求解最优化问题的一种方法。通常这种问题有很多解，每个解都对应一个值，最优化问题是希望找到一个对应最优值（最大值或最小值）的解。动态规划与分治法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的，即子问题之间具有重叠的部分。在这种情况下，如果用分治法求解就会重复地求解这些重叠的部分，而动态规划只会对这些重叠的部分求解一次并用表格保存这些解，如此一来就可以避免大量的重复计算。动态规划最特别的地方是自底向上地求解子问题并将这些子问题的解保存起来。这种用空间换取时间的方式大大提高了求解问题的效率。动态规划求解问题一般可以分为 4 个步骤：

(1) 定义最优解的结构。

(2) 递归地定义最优解的值。

(3) 以自底向上的方式计算最优解的值。

(4) 用第(3)步中计算过程的信息构造最优解。

5.6　以老鼠走迷宫为例

通过前面对计算思维的基本思想和解题思路的学习，大家是不是已经跃跃欲试了？本节就以老鼠走迷宫为例向大家介绍，在遇到具体问题时，我们学计算机科学的人是怎么思考并解决问题的。

老鼠走迷宫问题

问题描述：一只老鼠在一个 $n\times n$ 迷宫的入口处，它想要吃迷宫出口处放着奶酪，问这只老鼠能否吃到奶酪？如果可以吃到，请给出一条从入口到奶酪的路径。

沙老师：这个老鼠走迷宫问题是我在读大一时的程序作业题，那个时候对递归没有学习，只好用栈来实现，现在你们学了递归，这个程序就可以用递归来实现了。

思考：解决问题之前，我们首先要做的就是仔细研究问题，找出问题的已知条件和要得到的是什么，和解数学问题、物理问题一样要先弄懂问题。那么，老鼠走迷宫问题的已知条

件有什么呢?

(1) 用数学模型重新定义问题。

已知条件包括:$n \times n$ 迷宫,迷宫的入口,迷宫的出口。图 5-13 是一个 10×10 的迷宫,绿色部分是墙,白色部分是可以走的路,10×10 表示这个迷宫的长和宽分别是 10。如图 5-13 所示,迷宫的入口在上面,迷宫的出口在右面。

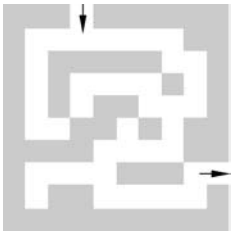
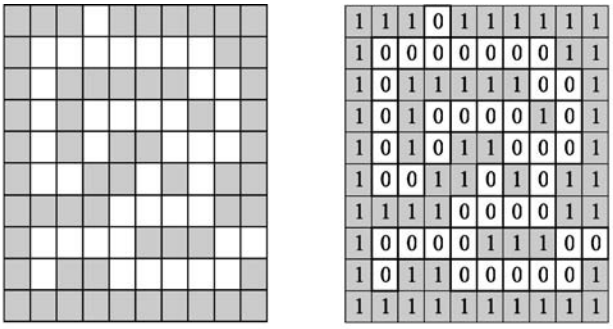


图 5-13　一个 10×10 的迷宫

问题:问老鼠能否吃到奶酪就是问能否找到一条从迷宫入口到出口的路径。如果不能找到,那么老鼠就吃不到奶酪;如果能够找到,那么就给出这条路径。

分析了已知条件和问题之后,有时还不能直接对问题进行求解。一般来说,很多问题都是用语言描述的,而计算机科学求解问题的方式是计算。因为语言上的描述是不能进行计算的,所以需要将这些问题用数学的形式重新描述。也就是说,要用数学模型重新定义问题。用数学模型重新定义问题是计算机科学求解问题时至关重要的环节。它的成功与否直接决定着能否解决问题。

观察如图 5-13 所示 10×10 的迷宫。这个迷宫其实是由 10×10=100 个格子组成的,其中绿色格子代表墙,白色格子代表路,如图 5-14(a)所示。“绿色格子代表墙,白色格子代表路”是用语言形式描述的,需要转换成数学的形式。用 1 和 0 分别定义绿色格子和白色格子,可以得到如图 5-14(b)的迷宫。



1	1	1	0	1	1	1	1	1	1
1	0	0	0	0	0	0	0	1	1
1	0	1	1	1	1	1	0	0	1
1	0	1	0	0	0	0	1	0	1
1	0	1	0	1	1	0	0	0	1
1	0	0	1	1	0	1	0	1	1
1	1	1	1	0	0	0	0	1	1
1	0	0	0	0	1	1	1	0	0
1	0	1	1	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1

(a) 将10×10的迷宫划分成100个格子　(b) 用1和0定义绿色格子和白色格子

图 5-14　用数学形式重新定义 10×10 的迷宫

观察图 5-14,这个迷宫是不是看起来很像一个二维数组?将上面 10×10 的迷宫定义为二维数组,即

```
m[10][10] = [1,1,1,0,1,1,1,1,1,1,
             1,0,0,0,0,0,0,0,1,1,
             1,0,1,1,1,1,1,0,0,1,
             1,0,1,0,0,0,0,1,0,1,
             1,0,1,0,1,1,0,0,0,1,
             1,0,0,1,1,0,1,0,1,1,
             1,1,1,1,0,0,0,0,1,1,
             1,0,0,0,0,1,1,1,0,0,
             1,0,1,1,0,0,0,0,0,1,
             1,1,1,1,1,1,1,1,1,1]
```

有了对迷宫的数学定义，就可以很简单地定义迷宫的入口和出口了。如图 5-13 所示的迷宫，入口是 m[0][3]，出口是 m[7][9]。

老鼠走迷宫问题就是要找一条从入口到出口的路径，如果存在就返回这条路径；如果不存在，就返回不存在这种路径。观察图 5-14(b)，能够走的路是用 0 标示的白色格子，因此如果存在，这种路径一定由 0 标示的白色格子组成。也就是说，要在二维数组 m 中找一条从 m[0][3]到 m[7][9]的全部为 0 的路径。到目前为止，我们已经对老鼠走迷宫问题进行了数学形式的定义。

(2) 将原问题分解成小问题。

按照计算机科学解决问题的基本思路，我们也将老鼠走迷宫问题分解成小问题。

走迷宫时，只知道下一步可以走的路。在每一个白格子上，老鼠可以选择向上、下、左、右这 4 个相邻的格子走。但是只有当相邻的格子是白色的时候才能走。如图 5-15 所示，如果老鼠在中间的白色格子，它可以选择向上、下、左、右这 4 个相邻的格子走。但是因为右边和下边相邻的格子是墙，所以它只能向上或者向左走。

1	0	1
0	0	1
0	1	1

图 5-15 判断每个格子的行走方向

在每一个格子上的行走情况可以用数组的形式表示为：假设老鼠在 m[i][j](0<i<9,0<j<9)，与 m[i][j]上、下、左、右相邻的元素分别是 m[i−1][j]、m[i+1][j]、m[i][j−1]、m[i][j+1]。只有这些相邻元素中至少一个为 0 时，老鼠才能走过去。

因此，可以通过决定下一步走的方向，将当前位置到出口的路径问题转化成 m[i][j]到出口的路径问题，其中 m[i][j]是当前位置的相邻位置，并且 m[i][j]=0。这样就能将原问题分解成小问题。

(3) 求解小问题，用小问题的解构造大问题的解。

走迷宫的时候，如果走到了死胡同就返回到前面，去尝试没有走过的路。最终会出现两种结果：第一种，找到出口；第二种，走了所有能走的路都走不到出口。

转化路径问题的时候，如果尝试了所有相邻位置 m[i][j]，都不能得到入口到出口的路径问题，就相当于走到了死胡同。此时，就要返回到最近的、没有走过的位置，继续转化路径问题。最后会出现两种结果：第一种，最终能将原问题转化成入口到出口的路径问题，从而得到一条从入口到出口的路径；第二种，找遍所有转化方式都不能得到入口到出口的路径问题，从而可以知道没有从入口到出口的路径。

求解问题：假设老鼠在如图 5-13 所示的 10×10 迷宫的入口处，它想要吃迷宫出口处放的奶酪，问这只老鼠能否吃到奶酪？如果可以吃到，请给出一条从入口到奶酪的路径。根据前面思考中的解题思路，用 Python 实现老鼠走迷宫问题的代码如下：

```
#<程序:老鼠走迷宫_递归>
m=[[1,1,1,0,1,1,1,1,1,1],
   [1,0,0,0,0,0,0,0,1,1],
   [1,0,1,1,1,1,1,0,0,1],
   [1,0,1,0,0,0,0,1,0,1],
   [1,0,1,0,1,1,0,0,0,1],
   [1,0,0,1,1,0,1,0,1,1],
```

```
    [1,1,1,1,0,0,0,0,1,1],
    [1,0,0,0,0,1,1,1,0,0],
    [1,0,1,1,0,0,0,0,0,1],
    [1,1,1,1,1,1,1,1,1,1]]
sta1 = 0;sta2 = 3;fsh1 = 7;fsh2 = 9;success = 0
def legal(x,y):
    if 0 <= x < len(m) and 0 <= y < len(m[0]) and m[x][y] == 0: return True
    else: return False
def visit(i,j):
    m[i][j] = 2
    global success
    if(i == fsh1)and(j == fsh2): success = 1
    if(success!= 1)and legal(i - 1,j): visit(i - 1,j)
    if(success!= 1)and legal(i + 1,j): visit(i + 1,j)
    if(success!= 1)and legal(i,j - 1): visit(i,j - 1)
    if(success!= 1)and legal(i,j + 1): visit(i,j + 1)
    if success!= 1: m[i][j] = 3
    return success
def LabyrinthRat():
    print('显示迷宫:')
    for i in range(len(m)): print(m[i])
    print('入口:m[ %d][ %d]:出口:m[ %d][ %d]' %(sta1,sta2,fsh1,fsh2))
    if (visit(sta1,sta2)) == 0:print('没有找到出口')
    else:
        print('显示路径:')
    for i in range(10):print(m[i])
LabyrinthRat()
```

运行程序,可以得到下面的结果:

```
>>>
显示迷宫:
[1, 1, 1, 0, 1, 1, 1, 1, 1, 1]
[1, 0, 0, 0, 0, 0, 0, 0, 1, 1]
[1, 0, 1, 1, 1, 1, 1, 0, 0, 1]
[1, 0, 1, 0, 0, 0, 0, 1, 0, 1]
[1, 0, 1, 0, 1, 1, 0, 0, 0, 1]
[1, 0, 0, 1, 1, 0, 1, 0, 1, 1]
[1, 1, 1, 1, 0, 0, 0, 0, 1, 1]
[1, 0, 0, 0, 0, 1, 1, 1, 0, 0]
[1, 0, 1, 1, 0, 0, 0, 0, 0, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
入口: m[ 0 ][ 3 ]: 出口: m[ 7 ][ 9 ]
显示路径:
[1, 1, 1, 2, 1, 1, 1, 1, 1, 1]
[1, 3, 3, 2, 2, 2, 2, 2, 1, 1]
[1, 3, 1, 1, 1, 1, 1, 2, 2, 1]
[1, 3, 1, 0, 0, 0, 0, 1, 2, 1]
```

```
[1, 3, 1, 0, 1, 1, 0, 2, 2, 1]
[1, 3, 3, 1, 1, 3, 1, 2, 1, 1]
[1, 1, 1, 1, 2, 2, 2, 2, 1, 1]
[1, 0, 0, 0, 2, 1, 1, 1, 2, 2]
[1, 0, 1, 1, 2, 2, 2, 2, 2, 1]
[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

结果中的路径用 2 标示，走过的失败的路用 3 标示，将结果所得的二维数组转化成迷宫可以得到图 5-16，其中黄色（浅色）标示了从入口到出口的路径，红色（深色）标示了走过的失败的路径。

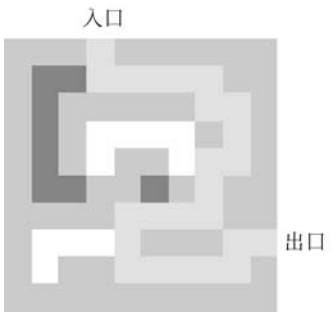


图 5-16　从入口到出口的路径图

练习题 5.6.1：将 m 改为

```
m = [[1,1,1,0,1,1,1,1,1,1], [1,0,0,0,1,0,1,0,1,1],
     [1,0,1,0,0,0,0,0,0,1], [1,0,1,0,1,0,0,1,0,1],
     [1,0,1,0,1,1,0,0,0,1], [1,0,0,0,1,0,1,0,1,1],
     [1,1,1,1,0,0,0,0,1,1], [1,0,0,0,0,1,1,1,0,0],
     [1,0,1,1,0,0,0,0,0,1], [1,1,1,1,1,1,1,1,1,1]]
```

输出的结果为何？将起点和终点互换后，结果又是如何？

练习题 5.6.2：在第 4 章我们提供了小乌龟画迷宫的程序，而本章展示了小老鼠走迷宫的程序，请利用小乌龟画迷宫的方式，画出小老鼠从起点到终点的路线。请在方格中画小圆圈来代表路径走过的方格。

练习题 5.6.3：上述解决老鼠走迷宫问题的程序使用了递归。请编写不用递归解决老鼠走迷宫问题的 Python 程序（提示：借助栈来实现）。

小结

通过前面对计算思维的基本思想和解题思路的学习，我们以老鼠走迷宫为例向大家介绍遇到具体问题时应该怎样思考并解决问题。首先，需要仔细分析问题并给出问题的数学描述；然后，尝试将原问题分解成小问题，并找出原问题和小问题或小问题和小问题之间的关系；最后，选取适当的方法求解问题。我们利用递归的思想求解老鼠走迷宫问题，并用 Python 实现了这种解法。

5.7　谈计算思维的美

本章我们向大家介绍了计算思维，也就是算法的一些内容。计算思维是运用计算机科学的基础概念进行问题求解、系统设计，以及人类行为理解等涵盖计算机科学之广度的一系列思维活动。简单来说，计算思维就是用计算机科学解决问题的思维。它是每个人都应该具备的基本技能，而不仅仅属于计算机科学家。对于学计算机科学的人来说，培养计算思维是至关重要的，而计算思维最重要的思想就是递归。

递归是计算思维最重要的一种基本思想，是大家在中学没有学习过的。递归是一个过程或函数在它的定义或说明中直接或间接调用自己的一种思想。它的本质是把一个复杂的大问题层层转化为一个与原问题相似的小问题，利用小问题的解来构筑大问题的解。利用

递归思想求解问题时,只需少量的程序就可描述出解题过程所需要的多次重复计算,从而大大地减少程序的代码量。它的能力在于用有限的语句来定义无限集合。正因如此,递归能用很少的代码解决很复杂的问题。学习用递归解决问题的关键就是找到问题的递归式,也就是用小问题的解构造大问题解的关系式。通过递归式可以知道大问题解与小问题解之间的关系,从而解决问题。

在介绍了递归这种基本思想之后,我们还介绍了分治法、动态规划和贪心算法这三种解决问题的基本方法。这三种方法在解题过程中都直接或间接地使用了递归这种基本思想。

分治法是计算机科学中非常重要的算法,字面上的解释是"分而治之",就是把一个复杂的问题分成两个或更多的相同或相似的互相独立的子问题,再把子问题分成更小的子问题,直到最后子问题可以简单地直接求解,原问题的解是子问题解的合并。在用分治法求解问题时一般分为三个步骤:分→治→合并。在"治"中往往会用到递归的思想。用分治法求解问题的优势是可以并行地解决相互独立的子问题。目前计算机已经能够集成越来越多的核,设计并行执行的程序能够有效利用资源,提高对资源的利用率。因此,用分治法求解问题也变得越来越重要。

动态规划是求解最优化问题的一种方法。通常这种问题有很多解,每个解都对应一个值,最优化问题是希望找到一个对应最优值(最大值或最小值)的解。动态规划与分治法类似,其基本思想也是将待求解问题分解成若干个子问题,先求解子问题,然后从这些子问题的解得到原问题的解。与分治法不同的是,适合于用动态规划求解的问题,经分解得到的子问题往往不是互相独立的,即子问题之间具有重叠的部分。在这种情况下,如果用分治法求解就会重复地求解这些重叠的部分,而动态规划只会对这些重叠的部分求解一次并用表格保存这些解,如此一来就可以避免大量的重复计算。动态规划最特别的地方是自底向上地求解子问题,并将这些子问题的解保存起来。这种用空间换取时间的方式大大提高了求解问题的效率。

贪心算法,又称为贪婪算法,也是用来求解最优化问题的一种方法。一般来说,求解最优化问题的过程就是做一系列决定从而实现最优值的过程,最优解就是实现最优值的这些决定。动态规划考虑全局最优,目标是得到全局最优解。贪心算法是一种在每一步选择中都简单地采取当前状态下最好(即最有利)的选择。贪心算法考虑局部最优,每次都做当前看起来最优的决定,得到的解不一定是全局最优解。但是在有最优子结构的问题中,贪心算法能够得到最优解。最优子结构就是局部最优解能决定全局最优解,简单地说,问题能够分解成子问题来解决,子问题的最优解能递推到最终问题的最优解。虽然对于很多问题贪心算法不一定能得到最优解,但是它的效率高,所求得的答案比较接近最优结果。因此,贪心算法可以用作辅助算法或者直接解决一些要求结果不特别精确的问题。

通过前面对计算思维的基本思想和解题思路的学习,我们以老鼠走迷宫为例向大家介绍遇到具体问题时应该怎样思考并解决问题。首先,需要仔细分析问题并给出问题的数学描述;其次,尝试将原问题分解成小问题,并找出原问题和小问题或小问题和小问题之间的关系;最后,选取适当的方法求解问题。我们利用递归的思想求解老鼠走迷宫问题,并用 Python 实现了这种解法。

我们在第 3 章最后用猜数字为例,说明计算机的智能是计算出来的,向大家展示了计算机的"思路"。这种"思路"其实是计算机程序的编写者赋予的。计算机应用人类赋予的计算

思维和其强大的计算能力，可以又快又准确地解决很多问题。

通过以上的学习，相信大家对计算思维已经有了一定的了解。那么现在，让我们来谈谈计算思维的美。

5.7.1 递归思想的美

学计算机科学的人在解决任何问题的时候，都喜欢先将一个很大很复杂的问题分解成小问题，然后对小问题进行求解，得到小问题的解后，用小问题的解来构筑大问题的解。这就是前面学习的递归，它是计算思维中非常美的一种思想，是以前没有学习和训练过的。

1. 递归思想美在能够用简单的描述解决复杂的问题

递归能够用简单的描述解决复杂问题的关键在于找到问题与较小规模问题间的递归关系，具体的表现形式是递归式。例如在求解汉诺塔问题时，最重要的是找到了 n 片圆盘和 $n-1$ 片圆盘移动次数的递归式：

$$f(n)=\begin{cases}1, & n=1\\ 2f(n-1)+1, & n>1\end{cases} \tag{5-9}$$

这种递归关系非常强大，它能描述所有相似问题间的关系。例如公式(5-9)描述了 $f(n)$与 $f(n-1)$、$f(n-1)$与 $f(n-2)$、…、$f(2)$与 $f(1)$之间的关系。只要实现一次递归关系就能解决所有满足这种关系的问题。因此，递归能够用简单的描述解决复杂问题。

2. 递归思想美在利用直接或间接地调用自己来减小问题规模

在日常生活中就存在直接或间接调用自己的这种方式，对此要特别小心。递归很容易产生逻辑上的悖论，无法说其真，也无法说其假。比如“我说谎”这句话直接调用自己，而“我下句话是对的”和“我上句话是错的”这两句话在间接调用自己。又如理发师悖论，某个村落，理发师挂出一块招牌：“我只给村里所有那些不给自己理发的人理发。”有人问他：“你给不给自己理发?”理发师无言以对。各位想想为什么？因为他假如给自己理发，那么他就不应该属于那些不给自己理发的集合，而假如他不给自己理发，他是属于那个不给自己理发的集合，但是照他的招牌，他又必须给自己理发，所以自相矛盾。

除此之外，前面介绍过的《画手》和《瀑布》两幅画也是直接或间接调用自己的形式。有兴趣的同学可以去看一下 *Godel*，*Escher*，*Bach*：*An Eternal Golden Braid*(中文版：(美)侯世达.哥德尔、艾舍尔、巴赫——集异璧之大成.严勇，等译.北京：商务印书馆，1997)这本书。哥德尔是著名的数学家，他证明了任何形式系统都包含了一个命题，它是无法证明真或假的，也就是哥德尔不完全性定理。艾舍尔是个画家，他喜欢画那些递归的图，例如前面的《画手》。而巴赫是伟大的音乐家，他的卡农(Canon)乐曲表现出递归的结构。数学、绘画、音乐因为递归而关联。这是一本杰出的科学普及著作，获得了普利策奖，以精心设计的巧妙笔法深入浅出地介绍了数理逻辑、可计算理论、人工智能、哲学等学科领域中的许多艰深理论。

而计算机科学中的递归是要解决问题的，可不能产生悖论。通过直接或间接地调用自己来减小问题规模，每次调用自己时参数的大小会减小，最后减小到 0 或可能的最小数，所以不会产生无限循环。例如用递归求解 n 片圆盘的汉诺塔问题时，函数 Hanoi 通过直接调用自己将求解 Hanoi(n，A，C，B)转化为求解 Hanoi($n-1$，A，B，C)、Hanoi(1，A，C，B)和

Hanoi($n-1$, B, C, A)的问题,从而将求解 n 片圆盘的汉诺塔问题减小为求解 $n-1$ 片圆盘的汉诺塔问题。递归会到 Hanoi(1,X,Y,Z)时停止而返回,其中,X、Y、Z 可为 A、B、C。

5.7.2 计算思维求解问题的基本方式的美

计算思维求解问题的方式与数学上求解问题的方式不同,以 5.2 节中的平面划分问题为例。通过对问题的分析,我们找到了递归式:

$$f(n)=\begin{cases}2, & n=1\\ f(n-1)+n, & n>1\end{cases} \tag{5-10}$$

在计算机科学中,只要有了上面的递归式,就可以编写程序解这个问题了。但是从数学上来说,有递归式还不够,需要通过推导得到一种称为闭合式(Close Form)的式子。由 $f(n)=f(n-1)+n$,同理可知 $f(n-1)=f(n-2)+n-1$,$f(n-2)=f(n-3)+n-2\cdots$,将 $f(n-1)$、$f(n-2)\cdots$依次代入前面的等式,得到:$f(n)=f(n-1)+n=f(n-2)+n-1+n=f(n-3)+n-2+n-1+n=\cdots=2+2+3+\cdots+n-1+n$。通过对 $f(n)=2+2+3+\cdots+n-1+n$ 进行整理,可以得到如下闭合式:

$$f(n)=\frac{n(n+1)}{2}+1 \tag{5-11}$$

有了上面的闭合式,才可以从数学上求解平面划分问题了。从数学上求解问题需要推导出闭合式,但是在很多情况下,是很难推导出这种闭合式的。大家学微积分时,是不是很苦恼去推导积分的闭合式?比如下式:

$$\int_a^b \frac{\sqrt{x}\sin x}{e^x}\ln x\,dx \tag{5-12}$$

如此一来,就不能在数学上求解了。但是计算机科学可以用趋近的方式找出数值来解这个问题。根据定积分的定义,其实就是求这个函数的图像在区间$[a,b]$的面积。例如求下式:

$$\int_a^b f(x)dx \tag{5-13}$$

根据定积分的定义,可以把直角坐标系上的函数图像用平行于 y 轴的直线分割成无数个矩形,如图 5-17 所示,然后把区间$[a,b]$上的矩形的面积累加起来,就得到了这个问题的解。

虽然这种方式需要很大的计算量,但是强大的计算能力正是计算机的优势。类似于上面这种数学上解不出的问题,可以在计算机科学中求解。当然,如果能够推导出闭合式,利用闭合式求解是再好不过的了。有了这种闭合式,一方面可以通过简单的计算得到结果;另一方面有助于推导出很多重要的理论。但是前提是要能推导出闭合式来。

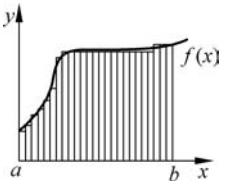


图 5-17 计算机对定积分求解

计算机科学虽然能求解平面划分这种数学问题,但更多的是做逻辑决策。利用计算思维中的基本解题方法,如分治法、动态规划和贪心算法等,求解如汉诺塔问题、最小值和最大值问题、背包问题、老鼠走迷宫问题等逻辑决策问题。

5.7.3 问题复杂度的研究之美

计算机科学不仅研究怎么解问题，更重要的是研究问题本身的复杂度。对问题复杂度的研究，一方面是想知道问题本身是不是能够求解，如果问题本身就是无解的，那么不管花费多少的人力、物力和财力都是求不出结果的；另一方面可以知道，如果问题能够求解，要求解这个问题需要花费多少代价。知道了求解问题的代价，我们就可以决定要不要解这个问题了。复杂度分析不仅可以分析问题本身的复杂度，还可以作为衡量一个解决方案好坏的工具。通过对同一个问题的不同解决方案进行复杂度分析，就可以知道哪种解决方案比较好。

下面给大家几个问题，请大家猜一下哪个问题最难。

(1) 找假币的问题：已知 n 枚硬币中有一枚是假币，并且知道这枚假币的重量要比真币轻，请找出这枚假币。

(2) 排序问题：对 n 个数进行非递减排序。

(3) 因数分解问题：A 和 B 是两个 200 位的质数，而 $C=A\times B$，已知 C，求 A 和 B。

(4) 停机问题(Halting Problem)：给一段程序代码，判断这段程序是否会停止。

前三个问题，计算机是可以解的，只不过因为问题的复杂度不同，求解问题的速度也不同。

第一个找假币的问题，我们在 5.1 节中已经给出了三种解决方式，其中最快的复杂度是 $\lg n$。

第二个排序问题，可以通过分治法解决，其时间复杂度是 $n\lg n$。比第一个问题的复杂度要高，也就代表排序问题比找假币问题要难一些。

第三个因数分解问题也是能够用计算机求解，只是速度很慢。我们知道，已知 A 和 B，要求 C 是很容易的，即使 A 和 B 都是 200 位的数，计算机也可以在瞬间(1 秒内)完成。但是已知 C，求 A 和 B，虽然能够用计算机求解，但到目前为止至少要花几个世纪的计算机时间才能解出来，而是否存在快速的求解方式还是个 21 世纪未知的科学问题。也正是因为很难找到解，信息安全才可以用这个方式在 RSA 加密算法中对密码进行保护。

举例来说：

```
>>>
C = 567171897715199615457525959834524213288953446523321038263482368178083258345843631380424449087280089617643567833490599233573928362318478052151664090732771626110066369590900271217260875902539234603387187704803877844320008746721453189170252325692202660485972529377473613639605269426749535980294483304533829648254862105493349041486310628762293895409821389282192327585094378958594369134310323407516309624498724654 9
```

这个 C 是两个 200 位左右的质数相乘而得到的。你能写个程序去算出是哪两个质数相乘而得到的吗？

产生 200 位的质数是不难的。首先可用 Python 写个简单又迅速的程序来检查 n 是否为质数(需要学习一种有趣的算法——随机算法)，然后随机产生 200 位的数，用前面的程序去检查是否为质数，尝试几次后就可以找到 200 位的质数。所以找到 200 位的质数不是难事，但是给一个两个质数的乘积，要你去分解，这就非常难了！这个“难”不是你写程序难，而是要在短时间得到解答难。看看下面的程序，这个程序很简单，可以找到所有 x 的因数。但是这个程序会花很久的时间也找不到前面 C 的因数，因为 C 的因数值太大了。读者可以试试看。

```
#<程序:Find all the factors of x and put them in list L>
import math
def factors(x,L):
    y = int(math.sqrt(x))          #x 的平方根
    for i in range(2,y + 1):       #一个个找
        if (x % i == 0):           #找到一个因数 i
            print(i)
            L.append(i)
            factors(x//i,L)        #递归找因数
            break                  #跳出 for 循环
    else: #找不到因数,x 是质数
        L.append(int(x))
        print(int(x))
```

前面的 C 是下面两个 200 位左右的质数相乘而得的：

```
A = 25955873305610796270399132756589243636705310864984605396274236420487096659713870289
18768664238621751027551621673246386392774936796078839001128518099837918049768238662502
94099493876066299839240396154824193 9
B = 21851389511621476464931020167297460701268598699989220759515741893917550762575408551
97119521601546991425667728902309075175868353364983425512289703511655544066381695780990
30098671969231506700261983613361599 1
```

其实有很多问题属于第三类问题，现在人类还没有找到任何一个较快速的解决方法，它们被称为 NP 完全的问题，比如旅行商问题(Travelling Salesman Problem，TSP)和布尔可满足性问题(Boolean Satisfiabilty problem，SAT)。

(1) 旅行商问题。

问题描述：有一位旅行商人要拜访 n($n>100$)个城市，求一条路径使得旅行商人每个城市只拜访一次，并且最终回到出发的城市。

要拜访 n 个城市共存在 $n!$ 条路径，要在这么多条路径中选择一条路径，因此 TSP 的时间复杂度是 $n!$。$n!$ 到底有多大呢，大家不妨想象一下 100! 有多大。在数学上，有一个斯特林公式(Stirling's Approximation)是用来取 $n!$ 的近似值的，即：

$$n!=\sqrt{2\pi n}\left(\frac{n}{e}\right)^{n} \tag{5-14}$$

光看右边的 n^n 就知道 $n!$ 是非常大的数了。

(2) 布尔可满足性问题。

问题描述：对于一个有 n 个变量的布尔方程式，是否存在一种输入使得输出是 True?

有 n 个要确定真假的变量，那么就要在 2^n 种输入中寻找解，因此 SAT 的复杂度是 2^n。在 n 很大的情况下，2^n 就是一个很大的数。比如 $n=10^9$，那么就要在 2^{10^9} 种输入中寻找解。

再回过头来看第四个问题。第四个停机问题是计算机解不出的问题，是不能保证有解的。不管你写什么样的程序，用几千万核的计算机都不能保证有解。实际上，与程序相关的问题(比如判断某行代码是否会执行等)都是计算机解不出的问题。在哲学层面来说，有些问题确实是计算机解不出来的，比如在介绍递归时说的语言上的递归“我下一句是错的”和“我上一句是对的”。

复杂度分析是计算机科学中最美的理论，它的出现和发展主要是因为第三类问题。很多在工业界的实际问题，如芯片设计、物流调度、管理优化等数千种问题，都需要快速解决方法，不希望花几个世纪来找到最优解，但是大家很多年来都找不出能快速解决的算法。大家就产生了疑问，这些问题到底是不是存在快速算法呢？还是我们人类的智商不够，找不到快速解法呢？至今，这个问题仍然是21世纪最大的科学谜团。假如有人找到了数千个NP完全问题中任意一个问题的快速解法，这个人将会改变文明，国际金融将要崩溃，因为这意味着信息安全所依赖的复杂问题能快速地被破解。人类还是有点自负，经过了几十年的研究至今仍然未找到快速解法，所以大部分人就认为这些NP完全问题是不存在快速解法的，但是至今还没有人能证明NP完全问题确实不存在快速解法！

计算机科学除了研究解决问题的算法外，也研究问题本身的复杂度。它可解吗？还是不可解？假如可以解，那么解决它最快要花多少时间？这就是问题的复杂度了。

习题5

习题5.1：请写出求 $n!$ 的递归表达式。

习题5.2：请写出用递归求 13^n 的Python程序，其中 n 作为输入。

习题5.3：已知数列 $\{a_n\}$ 的前几项为：

$$1,\quad \frac{1}{1+1},\quad \frac{1}{1+\dfrac{1}{1+1}},\quad \frac{1}{1+\dfrac{1}{1+\dfrac{1}{1+1}}},\quad \cdots$$

已知 $a_0=1$，请写出 a_n 的递归表达式。

习题5.4：根据 a_n 的递归表达式，请编程求 a_n 的精确分数解。输入为 n，输出为 a_n 的精确分数解。例如输入3，则输出3/5。

习题5.5：阿明写了 n 封信和 n 个信封，需要将信装入正确的信封才能邮寄出去。求所有的信都装错信封共有多少种情况？设 n 封信都装错信封有 $D(n)$ 种情况，请写出 $D(n)$ 的递归表达式。

习题5.6：请写出求解上面装错信封问题的递归程序。

习题5.7：写出Python程序，一定要利用递归函数，尽量不用for循环、while循环。输入整数 B 代表进制数，再输入两个 B 进制的数，用列表 x、y 表示，输出 $x+y$ 的 B 进制数，也可以用列表来表示。例如，$B=16$，输入[10,9,9]和[9,9]，它们加起来后的十六进制是[11,3,2]；假如 $B=11$，$x+y$ 就等于[1,0,8,7]。

习题5.8：如同前一个Python程序完成加法，请用递归函数来完成两个 B 进制数的乘法。输出乘积后的结果，以列表方式输出就可以了。例如 $B=10$，$x=[2,0,1]$，$y=[1,0]$，$x*y=[2,0,1,0]$，可以利用前面完成的加法函数。

习题5.9：利用递归求一个整数的各位数字。例如输入：2351554，则应输出：2 3 5 1 5 5 4。

习题5.10：利用递归程序生成 n 个数的全部可能的排列，例如 $n=3$ 时，应该输出：

```
123 132 231 232 321 312
Total = 6
```

习题 5.11：编写递归程序实现将一个较大的整数分解为若干个素数因子的乘积，并打印分解的结果。例如输入 24，则应输出：24＝2×2×2×3。

习题 5.12：拿牌游戏。假设面前有三堆扑克牌，其中每堆各有 10 张牌。两个人交替从某一堆中拿牌，谁拿到最后一张牌谁就输，请找出所有必赢的拿牌方式。

习题 5.13：用 Python 实现第 3 章最后所示的猜数字游戏。在这个作业里不考虑有重复数字的 3 位数，例如 335 等。两个人都各自选定一个秘密的三位数，然后相互猜对方的数字。用“几个 A”来表示对方猜得三位数中有几个数是完全正确的。用“几个 B”来表示有几个数正确但是位置不对，看是计算机还是你先猜到对方的数字。

习题 5.14：有 n 级台阶，一次可跨 1 级、2 级或 3 级，这样走完 n 级台阶的方法有很多种。例如 $n=4$ 时，可得：4＝1＋1＋1＋1＝1＋1＋2＝1＋2＋1＝2＋1＋1＝1＋3＝3＋1＝2＋2，共 7 种走法。请编写递归程序，求解 n 级台阶共有多少种走法，其中输入为台阶数 n，输出为走法的总数。

习题 5.15：假设有 n 个数 $a_0, a_1, a_2, \cdots, a_{n-1}$，请写出用分治法求 n 个数和的基本思想。

习题 5.16：请用递归思想实现快速排序(Quick Sort)算法。Quick Sort 是使用了分治法的一种排序方法，假设要对列表 L 进行非递减的排序，它的主要思想是：

(1) 在 L 中随机选择一个数 k，将 L 中所有小于或等于 k 的数都放到 k 的左边，将所有大于 k 的数都放到 k 的右边；

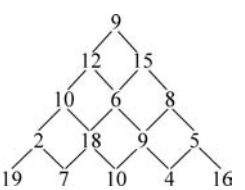
(2) 分别用 Quick Sort 算法对 k 的左边和右边进行非递减的排序。

请用 Python 实现 Quick Sort 算法，其中输入是整数列表 L，输出是一个非递减的有序列表 L′。

习题 5.17：请用 Python 实现 Partition(L, k)函数，其中 L 是一个整数列表，k 是列表 L 的一个下标。Partition 函数返回两个列表：L0 和 L1，其中 L0 中的所有数都小于或等于 L[k]，L1 中的所有数都大于 L[k]，L[k]不在 L0 和 L1 中，也就是说 len(L0)＋len(L1)＝len(L)－1。

习题 5.18：合唱队形问题。N 位同学站成一排，音乐老师要请其中的($N-K$)位同学出列，使剩下的 K 位同学排成合唱队形。合唱队形是指这样的一种队形：设 K 位同学从左到右依次编号为 1，2，…，K，他们的身高分别为 $T_1, T_2, \cdots, T_K$，则他们的身高满足 $T_1 < T_2 < \cdots < T_{i-1} < T_i > T_{i+1} > \cdots > T_K$，其中 $1 \leqslant i \leqslant K$。已知所有 N 位同学的身高，计算最少需要几位同学出列，可以使剩下的同学排成合唱队形(提示：用动态规划找最长上升子序列和最长下降子序列)？

习题 5.19：有如下图所示的数塔，在每一个结点都可以选择向左下或者右下走。请找出一条从顶部到底层的数值之和最大的路径。请用 Python 实现解决上述问题的方法。(提示：用动态规划求解)



习题 5.20：在第 4 章我们提供了小乌龟画迷宫的程序，而本章展示了小老鼠走迷宫的程序，请利用小乌龟画迷宫的方式，画出小老鼠从起点到终点的路线来。请在方格中画小圆圈来代表路径走过的方格。

习题 5.21：连接成最大整数问题。设有 n 个正整数，将它们连接成一排，组成一个最大的多位整数。例如，$n=3$ 时，3 个整数分别为 13，312 和 343，连成的最大整数为：34331213。又如，$n=4$ 时，4 个整数 7，13，4 和 246 连接成的最大整数为 7424613（提示：用贪心算法求解。策略：先把整数化成字符串，然后比较 $a+b$ 和 $b+a$，如果 $a+b>b+a$，就把 a 排在 b 的前面，反之则把 a 排在 b 的后面）。

习题 5.22：有 n 堆纸牌，分别编号为 1，2，…，n。每堆有若干张纸牌，但纸牌总数是 n 的倍数。可以在任意一堆取若干张牌，然后将这些牌移动到其他堆上，移动纸牌的规则为：

（1）在编号为 1 的堆上取的纸牌只能移动到编号为 2 的堆上；

（2）在编号为 n 的堆上取的纸牌只能移动到编号为 1 的堆上；

（3）在其他堆上取的纸牌，可以移动到相邻的左边或者右边的堆上。

求使每堆纸牌一样多的最少移动次数。

例如 $n=4$ 时，4 堆的纸牌数分别为：9，8，17，6。使每堆纸牌一样多的最少移动次数为 3 次，即按照下面的方式移动：

（1）从编号为 3 的堆上取 4 张牌放到编号为 4 的堆上，此时 4 堆的纸牌数分别为：9，8，13，10；

（2）从编号为 3 的堆上取 3 张牌放到编号为 2 的堆上，此时 4 堆的纸牌数分别为：9，11，10，10；

（3）从编号为 2 的堆上取 1 张牌放到编号为 1 的堆上，此时 4 堆的纸牌数分别为：10，10，10，10。

请用 Python 实现解决上述问题的方法。输入：n 堆纸牌（$1\leqslant n\leqslant 100$）；每堆的纸牌数 a[0]，a[1]，…，a[$n-1$]（$1\leqslant$a[$i$]$\leqslant 10\,000$）。输出：最少的移动次数（提示：用贪心算法，按照从左到右的顺序移动纸牌）。

习题 5.23：有一条由 N 颗珠子组成的项链（$3\leqslant N\leqslant 200$），珠子有两种颜色：白色和黑色，这些珠子随意串起。假如要在一点截断项链，将它展开成一条直线，从一端收集同色的珠子，再从另一端收集同色的珠子（颜色可能与前面收集的不同）。确定应该在哪里截断项链从而能够收集到最大数目的珠子。假设有一条由 29 颗珠子串成的项链，如下图所示，其中第 1 颗和第 2 颗珠子做了标记。

用 b 代表黑色珠子，w 代表白色珠子，上图的项链可以用字符串表示为：bwbwwwbbbwwwwwbwwbbwbbbbwwwwb。上图所示的项链最大可以收集到 8 颗珠子，截断的地方是：第 9 和第 10 颗珠子间或者第 24 和第 25 颗珠子间。请用 Python 实现解决上述问题的方法。输入：珠子总数和用字符串表示的项链；输出：最大可以收集到的珠子

数和截断的地方。

习题 5.24：由于乳制品产业利润很低，所以降低原材料(牛奶)价格就变得十分重要。帮助 Marry 乳业找到最优的牛奶采购方案。Marry 乳业从一些奶农手中采购牛奶，并且每一位奶农为乳制品加工企业提供的价格是不同的。此外，每头奶牛每天只能挤出固定数量的奶，每位奶农每天能提供的牛奶数量是一定的。每天 Marry 乳业可以从奶农手中采购到小于或者等于奶农最大产量的整数数量的牛奶。给出 Marry 乳业每天对牛奶的需求量，还有每位奶农提供的牛奶单价和产量。计算采购足够数量的牛奶所需的最小花费。

注：每天所有奶农的总产量大于 Marry 乳业的需求量。请用 Python 编写程序解决上述问题。输入：需要的牛奶总量 $N(1\leqslant N\leqslant 2\,000\,000)$，奶农数量 $M(1\leqslant M\leqslant 5000)$，以及每位奶农提过牛奶的单价 $P_i(1\leqslant P_i\leqslant 1000)$和产量 $A_i(1\leqslant A_i\leqslant 2\,000\,000)$；输出：采购到所需牛奶的最小花费。