第 5 章 递归和分治

一个函数调用了它自己，称为递归。递归和循环可以互相替代。一种程序设计语言，支持递归就可以不需要支持循环，支持循环就可以不需要支持递归。例如早期的 Lisp 语言，就不支持循环，只支持递归。当然，为了方便使用，程序设计语言一般都既支持递归，也支持循环。

下面这个循环调用了 n 次 doSomething 函数：

```
for i in range(n):
    doSomething()
```

可以用对递归函数的调用来替代：

```
def loop(n,f):
    if n == 0:
        return
    f()
    loop(n-1,f)
loop(8,doSomething)                          #调用 8 次 doSomething 函数
```

从替代循环的角度看，递归和循环一样是一种手段，可以用来解决任何问题。但是递归更多地是一种解决问题的思想——从这个角度看，也可以称递归是一种算法。

求列表 a 的最大值，可以如下实现：

```
def maxValue(a):
    ans = a[0]
    for x in a:
        if ans < x:
            ans = x
    return ans
```

也可以用递归的思想实现。思路是：如果 a 的长度是 1，则最大值就是 $a[0]$；否则，最大值就是 $a[1:]$中的最大值以及 $a[0]$这两者中较大的那个。递归程序如下：

```
def maxValue(a):
    def maxV(start):                         #求 a[start:]中的最大值
        if start == len(a)-1:                #a[start:]中只有一个元素
```

```
            return a[start]
        b = maxV(start+1)                    #递归求 a[start+1:]的最大值
        if a[start] < b:
            return b
        else:
            return a[start]
    return maxV(0)
```

该递归函数与循环的写法相比没有什么优势，但其体现的问题分解的思路是值得掌握的，即把规模大的问题不断分解为规模小的问题，最终加以解决。要求 a 的最大值，可以先解决一个形式相同但规模更小的子问题，即 $a[1:]$中的最大值(因 $a[1:]$可以看作一个比 a 更短的列表)。这个子问题解决了，将其答案和 $a[0]$做个比较，即可得到原问题的答案。

本章主要通过具体例题，分以下三种情况讲述递归的用途。

(1) 替代多重循环来进行枚举。

(2) 解决用递归形式定义的问题。

(3) 将问题分解为规模更小的子问题进行求解。

上述三种情况其实没有也不需要有严格的区分界限。有的问题，归类为上述不止一种情况都说得过去。例如，5.2 节的例题“绘制雪花曲线”，归类为第(2)或第(3)种情况都可以。

第(3)种情况，如果分解出来的子问题不止一个且相互无重叠，一般来说就可以算是“分治”。

还有一种递归可以称为“间接递归”，例如函数 A 调用了函数 B，函数 B 调用了函数 C，函数 C 又调用了函数 A，就可以说函数 A、B、C 都间接递归调用了自身。本章不涉及这类递归。

5.1 用递归进行枚举

有一类问题的本质，可以归结为：有 n 个变量，每个变量可以有 m 种取值。这 n 个变量取值的某些组合，能够满足某个条件，欲求出所有满足条件的组合。本章习题中的“奥数问题”“棋盘问题”，接下来要讲的案例“N 皇后问题”和“全排列”，本质都是如此。

这类问题最简单的解法就是暴力枚举所有组合，对每个组合验证其是否满足条件。这样一共要枚举 m^n 种组合，可以用 n 重循环来解决。当 n 比较大的时候，多重循环不好写，因此可以用递归的写法进行枚举。

5.1.1 案例：N 皇后问题(P0230)

将 N 个皇后摆放在一个 N 行 N 列的国际象棋棋盘上，要求任何两个皇后不能互相攻击(两个皇后在同一行，或同一列，或某个正方形的对角线上，就会互相攻击，称为冲突)。输入皇后数 $N(1\leqslant N\leqslant 9)$，输出所有的摆法。无解输出“NO ANSWER”。行列号都从 0 开始算。

输入：一个整数 N，表示要把 N 个皇后摆放在一个 N 行 N 列的国际象棋棋盘上。

输出：所有的摆放放案。每个方案一行，依次是第 0 行皇后位置，第 1 行皇后位置，…，第 $N-1$ 行皇后位置。多种方案输出顺序如下：优先输出第 0 行皇后列号小的方案。如果两个方案第 0 行皇后列号一致，那么优先输出第 1 行皇后列号小的方案，以此类推。

样例输入

```
4
```

样例输出

```
1 3 0 2
2 0 3 1
```

解题思路：在皇后数目不确定的情况下，用循环解决比较麻烦，因此可以采用递归的方式进行枚举。程序如下：

```
#prg0350.py
1.  result = [0] * 12          #本程序最多能解决 12 皇后问题
2.  #result[i]表示第 i 行的皇后已经放在了 result[i]列
3.  def isOk(n,pos):           #判断第 n 行的皇后放在第 pos 列是否和已经摆好的皇后冲突
4.      for i in range(n):     #检查位置 pos 是否会和前 0~n-1 行已经摆好的皇后冲突
5.          if result[i] == pos or abs(i-n) == abs(result[i] - pos):
6.              return False
7.      return True
8.  def queen(N,i):      #解决 N 皇后问题,现在第 0 行到第 i-1 行的 i 个皇后已经摆放好了
9.                             #要摆放第 i 行的皇后,
10.     if i == N:             #已经摆好了 N 个皇后,说明问题已经解决,输出结果即可
11.         for k in range(N):
12.             print(result[k], end=" ")
13.         print("")
14.         return True
15.     succeed = False
16.     for k in range(N):     #枚举所有位置
17.         if isOk(i,k):      #看可否将第 i 行皇后摆在第 k 列
18.             result[i] = k  #可以摆在第 k 列,就摆上
19.             succeed = queen(N,i+1) or succeed     #接着去摆放第 i+1 行的皇后
20.     return succeed
21. N = int(input())
22. if not queen(N,0):
23.     print("NO ANSWER")
```

queen 函数返回值为 True 或者 False，queen(N,i)的返回值表示：当前，前 i 个皇后已经摆好且不冲突，它们的摆法放在 result[0:i]中，在不改变这前 i 个皇后的摆法的前提下，继续往下摆放，最终能否找到至少一种成功的 N 皇后摆法。在第 16 行，函数试图为第 i 行的皇后找到所有和前 i 个皇后不冲突的位置（所谓“枚举”，就体现在本行），每找到一个，就摆放之，然后递归调用一次 queen(N,i+1)继续后面皇后的摆放；如果找不到，则返回 False，表示在当前情况下，最终无法摆放成功。

第 14 行：由于函数是递归调用的，所以有几个解，本行就会被执行几次。例如，在第 0 行皇后摆在第 0 列的情况下，若执行 queen(8,1)最终会成功，本行会得到执行；在第 0 行皇后摆放在第 1 列的情况下，若执行 queen(8,1)最终也会成功，本行又会得到执行。实际上，如果有 k 个解的第 0 行皇后都是摆放在第 0 列的，则会有 k 次执行到本行时，第 0 行皇后是

摆放在第 0 列的。

第 15 行：succeed 表示在当前情况下，再往下摆能否至少找到一个解，先假定为 False。

第 18 行：假设找到的第一个合法摆放位置是 k_1，第 i 行皇后摆放在第 k_1 列的情况下，会继续执行 queen(N,i+1)，从此处一直递归下去，有可能会找到多种 N 皇后的最终合法摆放方案，即多次走到第 11 行，输出多个解。这些解的第 0 行到第 i 行的摆放方案都是相同的，例如，第 i 行都是摆放在 k_1 位置。queen(N,i+1)执行过程中经历层层多分支递归，终究会返回，返回后回到第 16 行的 for 循环，继续寻找下一个可行的第 i 行摆放位置 k_2，然后再继续递归下去。

因此上面的程序会输出所有的解。

第 19 行：在第 16 行开始的循环中，只要有一次执行本行时 queen(N,i+1)返回 True，succeed 的值就会是 True，意味着在当前情况下最终能够摆放成功。

本程序中，result 列表中的一个元素，本质上就相当于多重循环解法里的一个循环控制变量，它们都是表示一行皇后的位置的。

请注意：如果要将程序改写成找到一个解就结束，则只需要将第 19 行替换为下面两行：

```
#prg0351.py
            if queen(N,i+1): #接着去摆放第 i+1 行的皇后
                return True
```

这样的话，摆放第 i 行皇后的时候，一旦发现一个能导致最终摆放成功的摆法，就不会去尝试第 i 行的下一个摆法，因此对每行的皇后，都只找到一个能导致最终成功的摆法，于是程序的第 11 行只会执行 1 次，程序只输出一个解。

N 皇后问题的本质，是有 N 个变量，这 N 个变量取值的某些组合，能够满足某个条件，要求出这些满足条件的组合。下面全排列问题，习题中的棋盘问题本质都是如此，都可以用类似 N 皇后问题的办法解决。

5.1.2 案例：全排列(P0240)

给定一个由不同的小写字母组成的字符串，输出这个字符串的所有排列。假设对于小写字母有‘a’＜‘b’＜…＜‘y’＜‘z’，而且给定的字符串中的字母已经按照从小到大的顺序排列。

输入：输入只有一行，是一个由不同的小写字母组成的字符串，已知字符串的长度在 1 到 6 之间。

输出：输出这个字符串的所有排列方式，每行一个排列。要求字母序比较小的排列在前面。字母序如下定义。

已知 $S = s_1 s_2 \cdots s_k$，$T = t_1 t_2 \cdots t_k$，则 $S < T$ 等价于：存在 p ($1 \leqslant p \leqslant k$)，使得 $s_1 = t_1$，$s_2 = t_2$，…，$s_{p-1} = t_{p-1}$，$s_p < t_p$ 成立。

样例输入

```
abc
```

样例输出

```
abc
acb
bac
bca
cab
cba
```

本题的本质就是在 n 个位置(编号 0～$n-1$)摆 n 个不同字母,要求给出满足“每个字母只出现一次”这个条件的所有摆法。解题程序如下:

```
#prg0360.py
1.  lst = list(input())
2.  lst.sort()                  #因为输入就排好序,所以本行去掉也可以
3.  n = len(lst)
4.  result = [0] * n            #result[i]表示在位置 i 摆放的字母
5.  used = [False] * n          #used[i]表示 lst 中的第 i 个字母是否已经用过
6.  def permutation(i):         #从第 i 个位置起摆放字母
7.      if i == n:              #条件满足则说明 n 个位置都摆上字母了,即发现了一个排列
8.          for x in result:
9.              print(x,end="")
10.         print("")
11.         return
12.     for k in range(n):
13.         if not used[k]:                 #第 k 个字母还没用过
14.             result[i] = lst[k]          #在第 i 个位置摆上第 k 个字母
15.             used[k] = True
16.             permutation(i+1)            #从第 i+1 个位置起继续往下摆
17.             used[k] = False
18. permutation(0)                          #从第 0 个位置开始摆放字母
```

函数 permutation(i)表示,在 0 到 $i-1$ 这 i 个位置已经摆好字母的情况下,从第 i 个位置开始继续摆放字母。位置 k 摆放的字母,记录在 result[k]中($k=0,1,\cdots,n-1$)。

第 12 行:枚举所有在第 i 个位置可能摆放的字母,k 是字母在 lst 中的下标。由于在每个位置枚举字母的时候都是按照字母从小到大的顺序进行,所以最终会按从小到大的顺序得到一个个排列,类似于 N 皇后问题得到解的顺序。

第 13 行:一个排列中,每个字母只能用一次,因此用列表元素 used[k]记录字母 lst[k]是否已经被用过。lst[k]还没用过,就可以摆在位置 i。摆上后就要将 used[k]设置为 True,表示 lst[k]已经被使用,如第 15 行所示。

第 17 行:下次循环就要在位置 i 尝试摆放别的字母。要在位置 i 摆别的字母,就应该将刚才在第 14 行摆放在位置 i 的字母 lst[k]拿走,那么字母 lst[k]就应该恢复成没用过,这样在后续位置还可以摆放它。因此要让 used[k] = False。

本程序在位置 i 摆放字母前,先判断能不能摆,能摆了才摆,而不是不管能不能摆,把 n 个位置都摆上字母然后再判断整个完整的摆法是否符合要求,这样可以大大提高效率,因为不会出现摆出“aaa”然后再判断出“aaa”不符合要求这种浪费时间的情况。

5.2 解决用递归形式定义的问题

有一些问题或者概念，本身的定义就是递归形式的。例如“自然数 n 的阶乘”这个概念，可以定义成 $1\times2\times3\times\cdots\times(n-1)\times n$，也可以用以下两句话来定义。

(1) 1 的阶乘是 1。

(2) $n>1$ 时，n 的阶乘是 n 乘以 $(n-1)$ 的阶乘。

第(2)句话，定义“阶乘”这个概念的时候，用到了“阶乘”这个词，看上去像是循环定义，让人无法理解。但是，由于有第(1)句话的存在，上面这两句“自然数 n 的阶乘”的定义，就是严密和可以理解的。可以说第(2)句话的形式是递归的，第(1)句话就是递归的终止条件。

按照上面的定义方式，求自然数 n 的阶乘的函数可以写成：

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)
```

这是最简单的用递归函数解决递归形式的问题的例子。

在上面的程序中，判断“n == 1”是否为真是基本操作，即进行次数最多的操作。设求 n 阶乘的基本操作次数为 $T(n)$，则有：

$T(n)=1+T(n-1)=1+1+T(n-2)=1+1+1+T(n-3)=\cdots=1+1+\cdots+T(1)=1+1+\cdots+1$（$n$ 个 1）

所以函数的复杂度是 $O(n)$。

5.2.1 案例：波兰表达式(P0250)

波兰表达式是一种把运算符前置的算术表达式。例如，一般形式的表达式 2 + 3 的波兰表示法为+ 2 3。波兰表达式的优点是计算时不需要考虑运算符的优先级，因此不必用括号改变运算次序，例如，(2 + 3) * 4 的波兰表示式为 * + 2 3 4。本题求解波兰表达式的值，其中运算符包括+、-、*、/四个。

输入：输入为一行，其中运算符和运算数之间都用空格分隔，运算数是浮点数。

输出：输出为一行，表达式的值。

样例输入

```
* + 11.0 12.0 + 24.0 35.0
```

样例输出

```
1357.000000
```

虽然什么是“波兰表达式”不难理解，但是题目其实并没有给出“波兰表达式”的准确定义。波兰表达式可以用递归的形式准确定义如下。

(1) 一个数是一个波兰表达式，其值就是该数本身。

(2) 若一个波兰表达式不是一个数，则其形式为："运算符 波兰表达式 1 波兰表达式 2"，其值是以"波兰表达式 1"的值作为第一操作数，以"波兰表达式 2"的值作为第二操作数，进行"运算符"所代表的运算后的值。"运算符"有加减乘除 4 种，分别表示为"+""-""*""/"。

根据上述定义，可以写出解题程序如下。

```
#prg0370.py
1.  def polishExp(exp):
2.      N = 0
3.      def polish():
4.          nonlocal  N
5.          if exp[N] == '+':
6.              N = N + 1
7.              return polish() + polish()
8.          elif exp[N] == '-':
9.              N = N + 1
10.             return polish() - polish()
11.         elif exp[N] == '*':
12.             N = N + 1
13.             return polish() * polish()
14.         elif exp[N] == '/':
15.             N = N + 1
16.             return polish() / polish()
17.         else:
18.             result = float(exp[N])
19.             N = N + 1
20.             return result
21.     return polish()
22. exp = input().split()
23. print('%.6f' % polishExp(exp))
```

第 2 行：N 相当于一个指针，表示 polish 函数被调用时，应该从字符串列表 exp 中下标为 N 的元素开始，取出若干个元素(假设 x 个)构成一个波兰表达式，计算出其值并返回。而且 polish 函数还必须让 N 变为 $N+x$，以便下一次调用 polish 函数时可以从正确的位置继续取元素。请注意，这里所说的"若干个元素"，其实元素个数是确定的，并不存在多种选择。例如，N==0 时调用 polish()，一定是将 exp 中的所有元素都取出作为一个波兰表达式。

第 5 行到第 7 行：按照波兰表达式定义，如果 exp[N]是"+"，则其后面一定跟着两个波兰表达式。"+"取走后，N 的值需要加 1。然后调用一次 polish()取出第一个波兰表达式并算出值，且让 N 推进到合适位置，再调用一次 polish()取出第二个波兰表达式，算出值，和第一个波兰表达式的值相加后返回。当然，第二次调用 polish()的时候也会推进 N 到合适位置。

第 17 行：按照波兰表达式定义，如果 exp[N]不是运算符，则其一定是一个数。那么取出的波兰表达式就是 exp[N]，其值为 float(exp[N])。

由于只需要从头到尾扫描整个表达式一遍，所以复杂度是 $O(n)$。

5.2.2 案例：绘制雪花曲线

篇幅所限，请扫二维码(绘制雪花曲线)查看本小节剩余内容。


绘制雪花曲线

5.3 用递归进行问题分解

递归解决问题的一种特别常用的基本思路是：**要解决某一问题，可以先做一步，做完一步以后，剩下的问题也许就会变成和原问题形式相同但规模更小的子问题，那就可以递归求解了。**第一步如果有多种选择，则要枚举第一步的所有选择。当然，还需要归纳出不需要递归就能直接解决的边界条件。

5.3.1 案例：上台阶(P0260)

有 n 级台阶($0<n<20$)，从最下面开始走，最终要走到所有台阶上面，每步可以走一级或两级，问：有多少种不同的走法？

输入：整数 n。

输出：走法总数。

样例输入

```
4
```

样例输出

```
5
```

解题思路：先走出第一步。第一步有两种走法，走一级台阶，或者走二级台阶。于是所有的走法就被分成两类，即第一步走一级台阶的和第一步走两级台阶的。问第一步走一级台阶共有多少种走法，等于就是问走 $n-1$ 级台阶一共有多少种走法。同样，第一步走两级台阶的走法数，等于 $n-2$ 级台阶的总走法数。于是我们发现，走出一步以后，剩下的两个子问题，和原问题形式相同，但是规模变小了(台阶数由 n 变成了 $n-1$ 和 $n-2$)。如果用 ways(i)表示 i 级台阶的走法数，那么：

```
ways(n) = ways(n-1) + ways(n-2)
```

这就是这个问题的递归公式，或者说递推公式。

应当选取一些边界条件，使得每条递归路径都会终止于边界条件，不会没完没了地递归下去。本程序的两条递归路径 ways($n-1$)和 ways($n-2$)，一条每次将 n 减 1，另一条每次将 n 减 2，那么将 n ==1 及 n == 0 设置为边界条件，就可以阻止无穷递归。程序如下：

```
#prg0390.py
def ways(n):                                    #n 级台阶的走法总数
   if n == 1 or n == 0:                         #0 级台阶有 1 种走法，就是不走
     return 1
```

```
    return ways(n - 1) + ways(n - 2)          #第一步走一级的走法+第一步走两级的走法
print(ways(int(input())))
```

换个边界条件，ways 函数程序改为下面这样也可以：

```
def ways(n):
    if n == 1:
        return 1
    elif n == 2:
        return 2
    return ways(n - 1) + ways(n - 2)
```

可以看出，此题的本质和求斐波那契数列的第 n 项一样。但是上面程序的复杂度，是指数级别的——因为算 ways($n-1$)和 ways($n-2$)的时候都要递归到底，造成大量重复计算。由于本题中的 n 比较小，所以不会超时。

本题用循环写成递推，复杂度只是 $O(n)$。虽然本题写成递归的形式是低效不可取的，但是用递归的思想来考虑这个问题，是合适的。

5.3.2 案例：算 24(P0270)

给出 4 个小于 10 的正整数，可以使用加减乘除 4 种运算以及括号把这 4 个数连接起来得到一个表达式。现在的问题是，是否存在一种方式使得得到的表达式的结果等于 24。

输入：若干行，每行一组数据，4 个整数。4 个 0 表示输入数据结束。

输出：对每组数据，如果能算出 24，输出“YES”，否则输出“NO”。

样例输入

```
5 5 5 1
1 1 4 2
0 0 0 0
```

样例输出

```
YES
NO
```

解题思路：此问题问用 4 个数算 24 能否成功。先做一步，即从 4 个数中取出两个数，做一下运算，得到一个新数。这一步做完后，剩下的问题就变成用 3 个数算 24 能否成功——形式和原问题一样，但是规模由 4 个数变成了 3 个数。边界条件是问用一个数算 24 能否成功，此时答案一目了然，就是看该数是否等于 24。第一步有多种选择，选出的两个数可以不同，选出的两个数做的运算也可以不同。枚举第一步的所有不同做法，只要有 1 种做法导致剩下的子问题答案是“YES”，那么整个原问题的答案就是“YES”。程序如下：

```
#prg0400.py
1.  import math
2.  EPS = 1e-6
3.  def equal( x,y ):                          #判断两个小数 x,y 是否相等
```

```
    return math.fabs(x-y) <= EPS
def count24(a):                              #用列表 a 中的数算 24,看能否成功
    n = len(a)
    if n == 1:
        return equal(24,a[0])
    for i in range(n-1):
        for j in range(i+1,n):
            x,y = a[i],a[j]                  #选两个数进行运算
            t = [x+y,x-y,y-x,x*y]
            if not equal(y,0):
                t.append( x / y)
            if not equal(x,0):
                t.append( y / x)
            for v in t:
                b = [v]                      #v 是 a 中取两个数进行运算的结果
                for k in range(n):
                    if k != i and k != j:
                        b.append(a[k])
                if count24(b):
                    return True
    return False

while True:
    a = list(map(int,input().split()))
    if a[0] == 0:
        break
    if count24(a):
        print( "YES" )
    else:
        print( "NO" )
```

第 18～21 行：b 由 a 中两个数 x、y 的运算结果，以及 a 中 x、y 以外的数构成。

因除法运算可能会得到小数，判断两个数是否相等时不宜直接用==符号，而是用专门编写的 equal 函数。

有的情况下，需要改变问题的描述形式，才能在“做了一步”后，使得剩下的问题和原问题形式一致。

5.3.3 案例：7 的倍数取法有多少种(P0290)

在 n 个($1 \leqslant n \leqslant 16$)不同的正整数里，任意取若干个，不能重复取，要求它们的和是 7 的倍数，问有几种取法。

此题也是本章习题，请读者写出程序。这里只讲思路。

解法 1：

每个整数只有取和不取两种状态，对 n 个整数来说，所有的取法共 2^n 种。对每种取法都算一下和是不是 7 的倍数即可。如果把每个整数的状态(取或不取)用一个二进制位表示，1 代表取，0 代表不取，那么可以发现，一种取法正好对应于一个 n 位的二进制数，而且不同取法对应的二进制数也不同——即 n 位二进制数和 n 个整数的 2^n 种取法正好是一一