第3章 递归算法

在计算机程序设计中，若一个函数在其函数体内又直接或间接调用了自身，则称之为递归函数。递归的例子在生活中比比皆是，如“从前有座山，山里有个庙，庙里有个老和尚，老和尚给小和尚讲故事……”，新闻联播中主播身后背景中的电视画面又出现了主播，在两面镜子中间看自己的像等。

要理解递归，仅了解递归的概念远远不够，需要从本质上掌握递归的特性，递归的使用场景以及递归存在的局限，才能够看懂、写出递归程序。因为递归是在函数内部调用其自身，所以递归要解决的必定是可重复性问题。计算机程序的实现是有穷的，需要在一定时间内执行完毕，故而递归程序不可能一直执行，必须在某个时刻到达出口，这就决定了问题的规模是逐渐递减的。从上述分析可以得出递归的本质特性如下。

(1) 问题及其子问题有相同的结构。

(2) 子问题的规模逐渐递减。

(3) 有终止条件作为出口。

用递归算法解决问题时，程序简洁清晰，易于理解，但递归程序调试复杂，真正理解相当不易。需要注意的问题是，由于函数调用的开销和子问题的重复求解，递归程序效率不高。因此，处理复杂问题时，递归程序往往需要与动态规划等算法结合使用。

3.1 汉诺塔问题

汉诺塔问题是心理学实验研究常用的任务之一，在医学临床上也常将汉诺塔任务用于检查脑损伤者的执行功能。汉诺塔问题主要涉及3根高度相同的柱子和大小互不相同的一套64个圆盘，3根柱子分别用起始柱A、辅助柱B及目标柱C代表。汉诺塔问题的目标是通过辅助柱B，将64个圆盘从起始柱A移动到目标柱C上。移动过程中，每次只能移动一个盘子，所有柱子上的盘子必须小盘在上、大盘在下。

3.1.1 汉诺塔问题解题思路分析

假设初始时盘子依序放在A柱上，盘子由1开始从上向下编号，先对问题进行必要分析获得问题的通解。

(1) $n=1$ 时，只有一个盘子，问题可以直接解决。

序　　号	盘　　子	移 动 过 程
第 1 次	1 号	A→C
合计	1 次	

(2) $n=2$ 时,2 个盘子需要借助其他柱子才能完成。

序　　号	盘　　子	移 动 过 程
第 1 次	1 号	A→B
第 2 次	2 号	A→C
第 3 次	1 号	B→C
合计	3 次	

(3) $n=3$ 时,移动次数明显增多。

序　　号	盘　　子	移 动 过 程
第 1 次	1 号	A→C
第 2 次	2 号	A→B
第 3 次	1 号	C→B
第 4 次	3 号	A→C
第 5 次	1 号	B→A
第 6 次	2 号	B→C
第 7 次	1 号	A→C
合计	7 次	

(4) $n=4$ 时,移动次数为 15 次。

以此类推,得到各项对应的形式为:$F_1=1$,$F_2=3$,$F_3=7$,…,$F_n=2F_{n-1}+1=2^n-1$,当 $n=64$ 时,$F_{64}=2^{64}-1=18446744073709551615$,即使移动一次用一秒,移动完成也要五千多亿年。

要想理解递归算法解决汉诺塔问题,关键在于得"麻痹"自己,相信自己像神笔马良一样会使用"神笔",可以将问题分为 3 个大的步骤解决,如图 3-1 所示。

(1) 将 A 柱顶端的 $n-1$ 个盘子通过施展"神笔"借助 C 柱移动到 B 柱上,此时 A 柱只剩最大一个盘子,B 柱有 $n-1$ 个盘子,C 柱为空。

(2) 第二步非常简单,直接将最大的盘子从 A 柱移动到 C 柱。

(3) 将 B 柱上的 $n-1$ 个盘子再次施展"神笔",借助 A 柱移动到 C 柱上,至此大功告成!

在此过程中,始终让人耿耿于怀的莫过于"神笔"功能是如何施展的。这个"神笔"的施展正是递归的精华所在。两次施展"神笔"处理的盘子数目都是 $n-1$ 个,规模小于原始

的问题的规模，求解方法却与原始问题相同。所以，“神笔”的施展与 n 个盘子时的处理方法相同，只需将图 3-1 中的 n 变为 $n-1$ 即可。处理 $n-1$ 个盘子时，可令 $m=n-1$，问题(1)就变成了将 m 个盘子从 A 柱借助 C 柱移动到 B 柱的过程，问题(3)则是将 m 个盘子从 B 柱借助 A 柱移动到 C 柱的过程。对于 m 个盘子的处理，与 n 个盘子处理的流程完全一致，这样一直递推下去，直至一个盘子时结束。

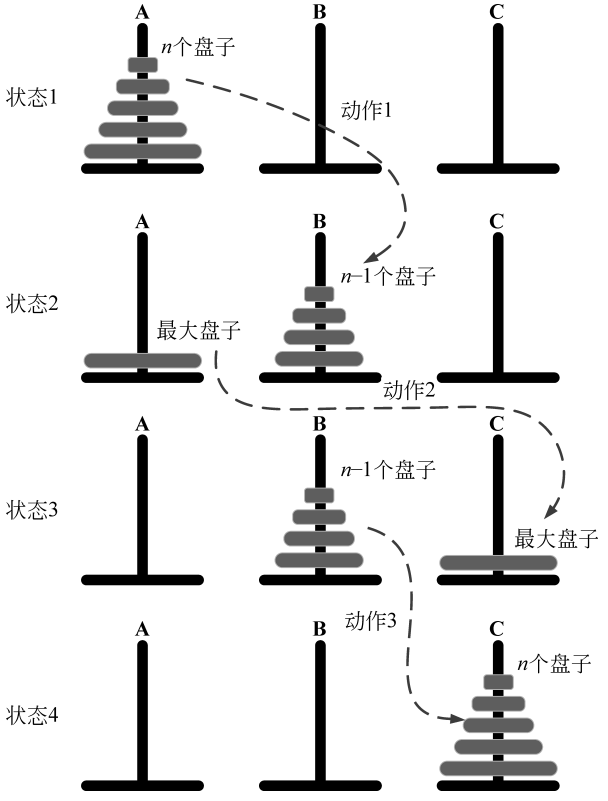

图 3-1　汉诺塔解题 3 步示意图

3.1.2　汉诺塔问题代码分析

求解汉诺塔问题主体功能的是 Hano()函数，main()函数用于测试，MoveDishes()函数用于输出移动圆盘的提示信息。

MoveDishes()函数的主要功能是输出实际移动圆盘时的提示信息，同时统计移动圆盘的总次数。因为函数执行后需要将总移动次数返回到 Hano()函数，供下次计数使用，所以代表移动次数的 count 变量必须为指针变量。

Hano()函数包括 5 个参数，n 为圆盘的个数，A 为起始柱对应的标识字母，B 为辅助柱对应的标识字母，C 为目标柱对应的标识字母，指针 count 用于统计移动次数。当只有一个圆盘时，直接将圆盘从 A 柱移动到 C 柱；当圆盘数多于一个时，需要施展“神笔”进行“三步走”，先将 $n-1$ 个圆盘从 A 柱借助 C 柱移动到 B 柱，然后将最大圆盘从 A 柱移动到 C 柱，最后将 $n-1$ 个圆盘从 B 柱借助 A 柱移动到 C 柱。实现代码如下。

```
#include<iostream>
using namespace std;
void MoveDishes(int disks, char source, char dest, int * count)
{
    (*count)++;//移动次数加 1
    cout << "第" << (*count) << "次移动: ";
    cout<< disks << "号圆盘 " << source << " -> " << dest << endl;
}
void Hano(int n, char A, char B, char C, int * count)
{
    if (n == 1)
        MoveDishes(n, A, C, count);
    else
    {
        Hano(n - 1, A, C, B, count);
        MoveDishes(n, A, C, count);
        Hano(n - 1, B, A, C, count);
    }
}
int main()
{
    int count = 0;
    Hano(3, 'A', 'B', 'C', &count);
    return 0;
}
```

当圆盘个数为 3 时,运行结果如下。

```
第1次移动: 1号圆盘 A -> C
第2次移动: 2号圆盘 A -> B
第3次移动: 1号圆盘 C -> B
第4次移动: 3号圆盘 A -> C
第5次移动: 1号圆盘 B -> A
第6次移动: 2号圆盘 B -> C
第7次移动: 1号圆盘 A -> C
```

3.2 全排列问题

排列就是将元素(或符号)按照确定的顺序进行重排,重排后的每一个顺序称为一个排列。例如,对于元素集 $S=\{1,2,3\}$而言,可获得 6 个排列 $p_1=(1,2,3)$,$p_2=(1,3,2)$,$p_3=(2,1,3)$,$p_4=(2,3,1)$,$p_5=(3,1,2)$,$p_6=(3,2,1)$。从 n 个不重复元素中任取 m 个元素,这 m 个元素构成的所有不同排列的个数称为排列数,记为 A_n^m 或 $A(n,m)$,计算公式如式(3.1)所示。

$$A_n^m=n(n-1)(n-2)\cdots(n-m+1)=\frac{n!}{(n-m)!} \tag{3.1}$$

其中，! 表示阶乘。以 A_6^3 为例，共有 3 个位置可以放置元素，第 1 个位置所有 6 个元素都可以放置，第 2 个位置只有 5 个元素可以放置，第 3 个位置只有 4 个元素可以放置，可构成的排列数为 $A_6^3=6\times5\times4=\frac{6!}{(6-3)!}=\frac{720}{6}=120$。

当 $m=n$ 时，构成的排列称为全排列。可以通过递归来枚举生成排列树的方式求解全排列问题。假定待处理的元素集合为 $S=\{s_1,s_2,\cdots,s_n\}$，令 $S_i=S-s_i$，用 $p(S)$ 表示 S 的全排列，$(s_i)p(S_i)$ 表示由 S_i 的每一个排列加上前缀 s_i 构成的排列，则全排列可用式(3.2)进行描述。

$$p(S)=\begin{cases}(s), & n=1\\ (s_1)p(S_1),(s_2)p(S_2),\cdots,(s_n)p(S_n), & n>1\end{cases} \tag{3.2}$$

图 3-2 给出了集合为{1,2,3,4}时的排列枚举树。

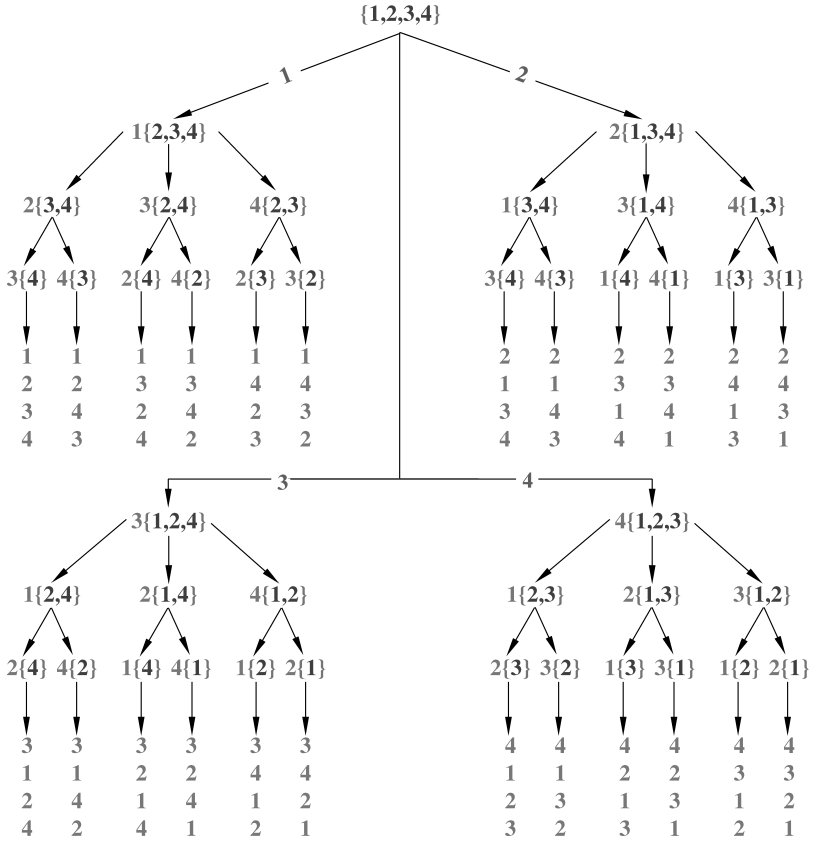

图 3-2　集合为{1,2,3,4}时的全排列枚举树

3.2.1　无重复元素的全排列

对一个无重复元素的有限集或其子集进行全排列时，可以按照最朴素的方法以递归方式进行求解。

1）解题思路

求有 n 个不重复元素的全排列的基本思路：先取出集合中的第 1 个元素 s_1 作为排列的第 1 位，将其余的 $n-1$ 个元素进行全排列；对 $n-1$ 个元素进行全排列时，同样先确定第 1 位，然后其余 $n-2$ 个元素进行全排列，以此类推，直至集合中只余一个元素时结束。从分析过程中可以看出，问题和子问题的求解方法是相同的，与原问题相比，子问题的规模不断缩小；当 $n=1$ 时，处理结束。这是典型的递归求解问题。

2）辅助函数

为了便于处理，假定需要进行排列的元素均为字符型。建立 Swap()函数，该函数用于交换集合中的两元素。函数有 3 个参数，list[]为构成排列的字符元素集合，m 和 n 为待交换值的两个元素对应的下标。

```
void Swap(char list[], int m, int n)
{
    char temp = list[m];
    list[m] = list[n];
    list[n] = temp;
}
```

3）朴素全排列函数

实现朴素全排列功能的函数为 NativeAllPermutation()。NativeAllPermutation()函数为递归函数，实现从 list[]集合中 start 至 last 范围内元素的全排列，使用 current 表示本次递归过程中的当前位置。函数有 4 个参数，list[]为元素集合，start 为起始位置下标(输出时使用)，current 为当前位置下标(递归时使用)，last 为终止位置下标。

当前递归位置 current 到达 last 时，说明已经实现了某个排列，到达出口，可以输出该排列。未到达出口时，从 current 至 last，依次交换 i 与 current 位置的元素，然后再进行全排列。实现代码如下。

```
void NativeAllPermutation(char list[], int start, int current, int last)
{
    int i;
    if (current == last)                    //当前位置到达 last,实现了一个排列,输出该排列
    {
        for (i = start; i <= last; i++)
            cout << list[i];
        cout << endl;
    }
    else
    {
        for (i = current; i <= last; i++)
        {
            Swap(list, current, i);
            //交换位置后,输出 list[j]不变、后面的字母改变的所有排列
            NativeAllPermutation(list, start, current + 1, last);
```

```
            //处理交换后的全排列之后,还需要还原准备下一次交换
            Swap(list, current, i);  //还原非常重要
        }
    }
}
```

至此,已经实现了 n 个不重复元素的全排列。但元素集合中有重复元素的情况尚未解决,例如元素集合为{a,b,b}时,交换后会出现两个值相同的重复排列。因此,需要解决全排列的去重问题,需要对当前算法进一步改进。

3.2.2 有重复元素的全排列

若元素集合中存在重复元素,生成全排列时需要解决重复问题,例如 abb 形式只应输出一次。生成消除重复元素全排列的主体函数是 AllPermutation()函数,参数与朴素全排列函数 NativeAllPermutation()相同,处理流程基本一致,唯一变化的是在进行进一步递归生成排列前调用了 NeedSwap()函数。

1) 辅助函数

NeedSwap()函数用于判定当前元素与之前的元素间是否存在重复,是实现消除重复全排列的关键。该函数有 3 个参数,list[]、current 和 i,作用和意义与朴素全排列函数 NativeAllPermutation()相同。去重全排列的根本在于当前元素只需要与它前面出现的非重复字符交换。因此,NeedSwap()函数增加了判断条件,判定两个待交换元素的值是否相同,若相等则表示元素值重复无须交换,不相同则应进行交换。

```
bool NeedSwap(char list[], int current, int i)
{
    int j;
    //若[current, i) 里存在与 a[i]相同的元素,说明该排列已经存在,不进行交换
    for (j = current; j < i; j++)
        if (list[j] == list[i])           //list[i]是等待被交换的元素
            return false;
    return true;
}
```

2) 整合后全排列代码

本部分代码将朴素全排列函数与消除重复情况的全排列函数合并到同一段代码中,在主函数中进行测试。

```
#include <iostream>
#include <cstring>
using namespace std;
void Swap(char list[], int m, int n)       //交换数组 A 中下标为 m 和 n 的元素
{
    char temp = list[m];
    list[m] = list[n];
```

```
    list[n] = temp;
}
void NativeAllPermutation(char list[], int start, int current, int last)
{
    int i;
    if (current == last)
    {
        for (i = start; i <= last; i++)
            cout << list[i];
        cout << endl;
    }
    else
    {
        for (i = current; i <= last; i++)
        {
            Swap(list, current, i);
            //交换位置后,输出以 list[j]不变,后面的字母改变的所有排列
            NativeAllPermutation(list, start, current + 1, last);
            Swap(list, current, i);       //还原
        }
    }
}
bool NeedSwap(char list[], int current, int i)//list[i]是等待被交换的元素
{
    int j;
    //若[current, i)内存在和 list[i]相同的元素,说明该排列已经存在
    for (j = current; j < i; j++)
        if (list[j] == list[i])
            return false;
    return true;                          //不存在,则 j == i,需要交换
}
//消除重复情况的全排列函数
void AllPermutation(char list[], int start, int current, int last)
{
    int i;
    if (current == last)                  //到达序列结尾,递归终止
    {
        for (i = start; i <= last; i++)
            cout << list[i];
        cout << endl;
    }
    for (i = current; i <= last; i++)     //从 current 至尾依次交换元素,生成全排列
    {
        if (!NeedSwap(list, current, i)) //重复则无须交换
```

```
            continue;
        Swap(list, current, i);
        AllPermutation(list, start, current + 1, last);
        //恢复原状以便进行下一轮交换:确保仍为交换之前的序列
        Swap(list, current, i);
    }
}
int main()
{
    const int MAX_CHARS = 101;
    char str[MAX_CHARS];
    int len, start, k, current;
    cin >> str >> start >> k;
    len = strlen(str);
    len = (len <= MAX_CHARS - 1) ? len : (MAX_CHARS - 1);
    current = start;
    //初始时 current 与 start 值相同,current 在递归过程中不断变化
    NativeAllPermutation(str, start, current, start + k - 1);
    //AllPermutation(str, start, current, start + k - 1);
    return 0;
}
```

3) 测试数据及运行结果

(1) 无重复情况。

当输入数据为 abcdef 1 3 时,运行结果如下(无重复情况)。

```
abcdef 1 3
bcd
bdc
cbd
cdb
dcb
dbc
```

(2) 有重复情况。

当输入数据为 abccd 1 3 时,运行结果如下(有重复)。

```
abccd 1 3
bcc
cbc
ccb
```

3.3 因数分解问题

正整数 S 可表示为若干个正整数的乘积,即 $S=s_1 \cdot s_2 \cdot \cdots \cdot s_n$,其中各个因子构成一个递增序列,即 $s_1 \leqslant s_2 \leqslant s_3 \leqslant \cdots \leqslant s_n$,称为 S 的因数分解。每一种 $s_1 \cdot s_2 \cdot \cdots \cdot s_n$ 称为 S 的一个分解,需要注意的是,$s=S$ 也是一种分解。例如,4,8,12 和 24 的分解数分别为 2,3,4 和 7,分解过程如下。

$$
\begin{aligned}
4 &= 1 \times 4 \\
&= 2 \times 2 \\
8 &= 1 \times 8 \\
&= 2 \times 4 \\
&= 2 \times 2 \times 2 \\
12 &= 1 \times 12 \\
&= 2 \times 6 \\
&= 2 \times 2 \times 3 \\
&= 3 \times 4 \\
24 &= 1 \times 24 \\
&= 2 \times 12 \\
&= 2 \times 2 \times 6 \\
&= 2 \times 2 \times 2 \times 3 \\
&= 2 \times 3 \times 4 \\
&= 3 \times 8 \\
&= 4 \times 6
\end{aligned}
$$

3.3.1　因子递增方式递归求解

令 $S_i = S/(s_1 \cdot s_2 \cdot \cdots \cdot s_i)$，对 S 进行分解时，其分解过程如下。

(1) S 自身算一种分解。

(2) 剩余分解可以通过 $2 \sim \sqrt{S}$ 范围内的每一个整数 k 进行试除。

① 若 $s_i = k$ 为 S 的因子，则寻找到一个新的分解 $s_1 \cdot s_2 \cdot \cdots \cdot s_i \times S_i$。

② 同时还应该注意到，S_i 仍有可能含最小因子不小于 $s_i = k$ 分解，需要对 S_i 从 $s_i = k$ 开始继续分解。

图 3-3 给出了正整数 4,8,12,24 从 2 开始进行因数分解的示意图。

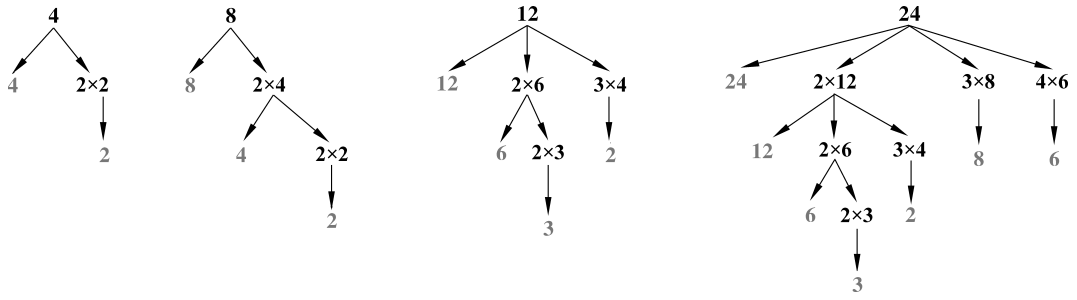

图 3-3　正整数 4,8,12,24 从 2 开始进行因数分解的示意图

以 24 为例，其因数分解过程如下。

(1) 24 自身是一种分解，因此获得第 1 种分解 24＝24。

取 $k = 2,3,4$(因为 $4 < \sqrt{24} < 5$)分别试除进行分解，递增试除保证了各个因子间是递

增的，而且不会出现重复分解。

(2) 当 $s_1=k=2$ 时，4 种分解如下。

① $S_1=S/s_1=24/2=12$，获得第 2 种分解 $S=s_1\cdot S_1$，即 $24=2\times12$。

② S_1 可进一步进行分解，取 $s_2=k=2$，$S_2=S/(s_1\cdot s_2)=24/(2\times2)=6$，获得第 3 种分解 $S=s_1\cdot s_2\cdot S_2$，即 $24=2\times2\times6$；S_2 可进一步进行分解，取 $s_3=k=2$，$S_3=S/(s_1\cdot s_2\cdot s_3)=3$，获得第 4 种分解 $S=s_1\cdot s_2\cdot s_3\cdot S_3$，即 $24=2\times2\times2\times3$。

③ S_1 可进一步进行分解，取 $s_2=k+1=3$，$S_2=S/(s_1\cdot s_2)=4$，获得第 5 种分解 $S=s_1\cdot s_2\cdot S_2$，即 $24=2\times3\times4$。

(3) 当 $s_1=k=3$ 时，$S_1=S/s_1=24/3=8$，获得第 6 种分解 $S=s_1\cdot S_1$，即 $24=3\times8$。

(4) 当 $s_1=k=4$ 时，$S_1=S/s_1=24/4=6$，获得第 7 种分解 $S=s_1\cdot S_1$，即 $24=4\times6$。

3.3.2 子问题分解方式递归求解

第 2 种分解方法主要是通过对问题的分析，获得子问题的表述，根据递归表达式来进行求解。3.3.1 节通过试除法，利用 $2\sim\sqrt{S}$ 的因子获取某个分解。也可以从 S 出发，以因子递减方式获得 S 的各个分解。S 的分解因数问题 $N(S,m)$ 的递归表达式如式(3.3)所示。

$$N(S,m)=\begin{cases}1, & S=1\\ 0, & m=1\\ N(S,m-1)+N(S/m,m), & m\mid S\\ N(S,m-1), & m\nmid S\end{cases}\tag{3.3}$$

其中，$m\mid S$ 表示 S 能被 m 整除，$m\nmid S$ 表示 S 不能被 m 整除。

3.3.3 分解因数问题代码分析

GetFactorNumbers1()函数使用因子递增方式进行分解因数。函数有两个参数，n 是待分解的数值，m 是分解的起始值(一般从 2 开始，即 m 的初值为 2)。函数在进行因数分解时，①将因数为自身的情况计入统计过程；②当重复分解至 1 时，分解过程结束；③未分解至 1 时，用循环变量 i 从 2 到$\sqrt{n}$逐个遍历进行试除，若试除成功则将 n/i 从 i 开始进一步分解。实现代码如下。

```
#include<iostream>
using namespace std;
int GetFactorNumbers1(int n, int m)        //从 2 开始遍历所有可能的分解
{
    int ans = 1;                           //自身算一种，为 1 时不算
    if (n == 1)
        return 0;
    for (int i = m; i * i <= n; i++)
    {
        if (n % i == 0)
```

```
            ans += GetFactorNumbers1(n / i, i);
        }
        return ans;
    }
```

GetFactorNumbers2()函数使用子问题分解方式进行分解因数。函数有两个参数，n是待分解的数值，m是分解的起始值(代表要分解的(疑似)最大因子，一般从n开始，逐渐递减至1)。函数在进行因数分解时，若被分解的数为1时，只有一种分解方式；若最大因子为1时，则返回0；若m为n的因子，则进行递归分解并统计解的总和；若m不为n的因子，则从$m-1$开始继续尝试新的分解过程。实现代码如下。

```
int GetFactorNumbers2(int n, int m)
{
    if (n == 1)
        return 1;                          //若分解的数字为 1
    if (m == 1)
        return 0;                          //若最大因子为 1
    //如果 m 确实是 n 的因子,进行递归调用,并将分解种数相加
    if (n % m == 0)
    {
        int c1 = GetFactorNumbers2(n, m - 1);
        int c2 = GetFactorNumbers2(n / m, m);
        return c1 + c2;
    }
    //m 不为 n 的因子,尝试将 m-1 作为最大因子
    return GetFactorNumbers2(n, m - 1);
}
int main()
{
    int n, m1, m2;
    int count;
    cin >> count;
    while (count--)
    {
        cin >> n;
        m1 = GetFactorNumbers1(n, 2);
        m2 = GetFactorNumbers2(n, n);
        cout << m1 << " " << m2 << endl;
    }
    return 0;
}
```

测试数据及运行结果如下。

当输入两个测试数据：12 和 24 时，12 有 4 种分解方式，24 有 7 种分解方式，运行结果如下。

```
2
12
4 4
24
7 7
```

3.4 分形图形

分形算法是现代数学的一个新分支，与动力系统的混沌理论交叉结合，相辅相成。在一定条件下，事物在形态、结构、信息、功能、时间或能量等在某方面表现出与整体的相似性。分形是指事物的多重自相似性，可以是以自然形态存在的，也可以是人为设计的。

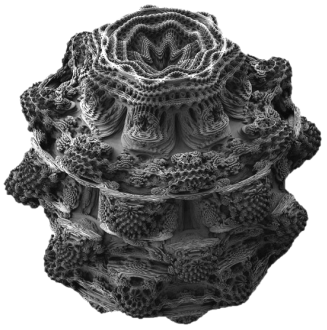

图 3-4 芒德球

美籍法国数学家 B.B.Mandelbrot 在 1975 年首先提出了分形几何的概念。计算机的应用和普及，使人们真正了解了分形。2009 年，英国分形爱好者 Daniel White 创造出 3D 分形影像，并将之命名为芒德球，如图 3-4 所示。尽管芒德球外形极其烦琐，但却是由基于复数的基础算法得出的。

3.4.1 盒分形思路分析

本节主要研究盒分形。盒分形的规模用度表述，盒分形定义如下。

(1) 度为 1 的盒子分形如下。

```
X
```

(2) 度为 2 的盒子分形如下。

```
X   X
  X
X   X
```

(3) 度为 n 的盒分形如下。

用 $B(n-1)$表示度为 $n-1$ 的盒分形，则度为 n 的盒分形可用式(3.4)表示。

$$\begin{matrix} B(n-1) & & B(n-1) \\ & B(n-1) & \\ B(n-1) & & B(n-1) \end{matrix} \tag{3.4}$$

假设输出符号为 X，度为 3 的盒分形输出如下。

```
X  X      X  X
  X         X
X  X      X  X
     X  X
        X
     X  X
X  X      X  X
  X         X
X  X      X  X
```

度为 n 的盒分形的可用边长为 3^{n-1} 的正方形表示其坐标，如图 3-5 所示。以屏幕左上角作为坐标原点，x 轴水平向右为正，y 轴垂直向下为正，递归函数 printBox(k,x,y)生成以坐标(x,y)为左上角的 k 度盒分形。递归生成符号为 X 的 n 度盒分形可表述如下。

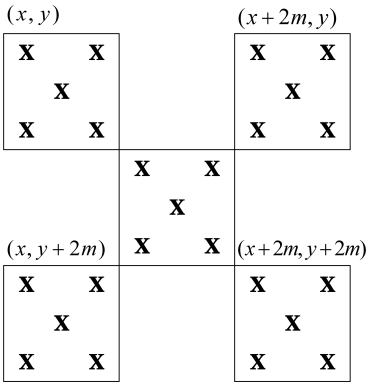

图 3-5 n 度盒分形生成示意图

(1) $n=1$ 时，到达递归边界，在(x,y)输出符号 X。

(2) $n>1$ 时，需要分别在左上、右上、中间、左下和右下生成 5 个 $n-1$ 盒分形，$n-1$ 度盒分形对应正方形的边长为 $m=3^{n-2}$。

① 左上方 $n-1$ 度盒分形，其左上角的坐标为(x,y)，调用 printBox(n-1,x,y)函数生成。

② 右上方 $n-1$ 度盒分形，其左上角的坐标为($x+2m$,y)，调用 printBox(n-1,x+2m,y)函数生成。

③ 居中的 $n-1$ 度盒分形，其左上角的坐标为($x+m$,$y+m$)，调用 printBox(n-1,x+m,y+m)函数生成。

④ 左下方 $n-1$ 度盒分形，其左上角的坐标为(x,$y+2m$)，调用 printBox(n-1,x,y+2m)函数生成。

⑤ 右下方 $n-1$ 度盒分形，坐标为($x+2m$,$y+2m$)，调用 printBox(n-1,x+2m,y+2m)函数生成。

3.4.2 盒分形代码分析

本节的盒分形算法代码将分形信息保存于二维数组 map[][]中。当某个位置存在分形标志时，map[i][j]的值用符号'X'表示，否则用空格表示。分形算法主要通过递归函数 FractalBox()实现。FractalBox()函数有 4 个参数，maps[][MAX_DEGREE]用于保存分形信息对应的标志，degree 代表当前调用对应的分形度数，x 和 y 代表分形对应的坐标位置。

为了简化输出盒分形的复杂度，将盒分形信息保存于二维字符数组 map[][]中，处理结束后以字符串的形式输出。主函数执行时，①计算 n 度分形对应的边长 3^{n-1}；②初始化保存分形信息的数组 map[]各元素为单个空格，将行尾置为'\0'；③调用 FractalBox()

函数以递归方式求解分形信息;④以字符串方式输出分形信息。实现代码如下。

```
#include <iostream>
#include <cmath>
using namespace std;
const int MAX_DEGREE = 730;              //最大规模为 7 度盒分形:n=3^(7-1)=3^6=729
void FractalBox(char maps[][MAX_DEGREE], int degree, int x, int y)
{
    //度为 1 时到达递归边界
    if (degree == 1)
        maps[x][y] = 'X';
    else
    {
        //n-1 度盒分形的规模 m=3^(n-2)
        int m = pow(3.0, degree - 2);
        //x 为行坐标,y 为列坐标
        //左上方的 n-1 度盒分形
        FractalBox(maps, degree - 1, x, y);
        //右上方的 n-1 度盒分形:行未变,列增大
        FractalBox(maps, degree - 1, x, y + 2 * m);
        //中间的 n-1 度盒分形:行列均增大
        FractalBox(maps, degree - 1, x + m, y + m);
        //左下方的 n-1 度盒分形:行增大,列不变
        FractalBox(maps, degree - 1, x + 2 * m, y);
        //右下方的 n-1 度盒分形
        FractalBox(maps, degree - 1, x + 2 * m, y + 2 * m);
    }
}
int main()
{
    char maps[MAX_DEGREE][MAX_DEGREE];
    int n;
    cin >> n;
    int degree = pow(3.0, n - 1);
    //初始化
    for (int i = 0; i < degree; i++) {
        for (int j = 0; j < degree; j++) {
            maps[i][j] = ' ';
            maps[i][degree] = '\0';
        }
    }
    FractalBox(maps, n, 0, 0);
    //输出
    for (int i = 0; i < degree; i++)
        cout << maps[i] << endl;
```

```
    return 0;
}
```

测试数据及运行结果如下。

当输入度数为 3 时,盒分形运行结果如下。

```
3
X X   X X
 X     X
X X   X X
   X X
    X
   X X
X X   X X
 X     X
X X   X X
```