第4章

递归

递归(recursion)是计算机科学中的重要概念. 我们可以递归地定义事物, 也即利用该事物的本身去定义它, 例如数学函数、算法描述等. 从某种意义上说, 递归是一种描述技巧, 它将较长的逐步描述转化为规则来递归, 其方向与数学归纳法相反, 但本质完全相同.

在算法设计中, *递归算法*(recursive algorithm)占据了重要的地位, 而且很多问题只能通过递归算法才能解决.[1] 我们需要掌握递归算法的设计方法并能证明其正确性, 只有这样才能达到利用递归高效完成任务的目的. 与此同时, 递归算法的分析更为重要, 时空效率决定了是否适宜采用递归算法解决某个问题.

4.1 概述

数学中的递归定义通常表现为一些简单的表达形式, 它通过一个较"小"的自我形态来递归定义较"大"的自我形态, 并利用*基础情形*(base case)终止递归过程. 例如最常见的Fibonacci数列:

$$F_0 = 0, \quad F_1 = 1, \quad F_n = F_{n-1} + F_{n-2} \ (n > 1).$$

可以看到较大的F_n依赖较小的F_{n-1}和F_{n-2}获得定义, 并在F_0和F_1这两个基础情形处终止. 需要注意很多问题的基础情形有多个, 缺一不可.

算法同样有递归形式, 其形式是利用一些较"小"的递归过程实现算法, 并利用基础情形终止算法. 例如可以实现x的自然数次幂:

```
double power_uint(double x, unsigned int n)  // 对于自然数n, 实现x的n次幂.
{
    if (n == 0)          // 基础情形.
        return 1;        // power_uint(x, 0)值恒为1, 但注意x不要取0.
    double p = power_uint(x, n / 2);
    p *= p;
    if (n % 2 == 1)
        p = p * x;
    return p;
}
```

[1] 当然, 递归算法在计算机中的实现却不是递归的, 递归过程是通过栈来完成的.

可以看到该函数在参数n输入时通过较小的参数n / 2获得实现, 并在参数n为0处得到终止.

上述两个递归例子都有非递归版本: Fibonacci数列存在一个显式的数学表达式, 而power_uint函数可由迭代形式更快地实现. 看似递归可有可无, 然而许多复杂的问题却只能由递归给出表述, 其经典例子就是Ackermann函数:

$$A(m,n)=\begin{cases} n+1 & m=0; \\ A(m-1,1) & m>0, n=0; \\ A(m-1,A(m,n-1)) & m>0, n>0. \end{cases}$$

而在很多算法中递归实现更为简洁而且有助于设计优化方案, 这方面的例子举不胜举, 读者可参考任意一本算法教材, 本章余下部分也将举出若干应用. 一言以蔽之, 递归其用大焉.

从递归的本质看, 递归是将问题通过有限个步骤一直减少其规模量, 最后终止于基础情形, 而这条路线的正确性决定了递归是否能运作. 一个常犯的错误是完全没有基础情形而导致无穷递归,[1] 例如:

```
void story()
{
    std::cout << "从前有座山, 山里有座庙. 庙里有一个老和尚给小和尚讲故事: ";
    story();
}
```

另一个常犯的错误是无法将递归推向基础情形, 这属于逻辑性的错误. 当然更重要的是递归本身的正确性, 也即能否由递归获得自身的实现.

4.2 [技巧] 递归设计与归纳证明

我们已经涉猎了一些较为简单的递归算法, 但尚未对复杂的问题设计算法并解决之. 本节将讲解如何设计递归算法, 并着重从数学归纳法的角度证明递归算法的正确性.

例 4.1 河内塔(Tower of Hanoi)问题是一个经典的智力问题, 它涉及三根柱子(记为A, B, C)和n个($n>0$)尺寸不同的碟子(中间开孔类似于光碟). 初始所有碟子全部套在A上且小碟位于大碟之上, 要求将这n个碟子全部移动到C上, 可借助B暂时放置一些碟子, 但必须保证在移动过程中和移动结束后所有柱子上不能出现大碟放在小碟之上的情况. 试设计算法输出具体移动步骤.

解: 移动碟子的方案应该由基本步骤组成, 可假设一次只移动一个碟子, 格式为move x to y, 即移动x柱顶部的碟子套入y柱, 利用这些基本步骤即可按指示操作. 由于问题的参数很明显, 不妨设

[1] 这种无穷递归一般是理论上的, 实际计算机一般采用调用栈实现递归, 会在有限次递归调用之后发生栈溢出而终止.

```
Hanoi(n, source, other, dest);
```

执行后可在屏幕上输出遵循游戏规则前提下将n个碟子从source到dest(可使用other来中转)的所有基本移动步骤. 注意, Hanoi函数所处理的三根柱子上的碟子情况只需"逻辑上"与河内塔问题一致. 例如source上的碟子数超过n, 而other和dest上无碟子, 如果可以移动的只是上面的n个碟子, 显然这个问题和n个碟子情况下的河内塔问题逻辑等价.

利用递归可以很简便地解决该问题. 首先n为1是基础情形, 只需要将source上的碟子直接移动到dest即可. 接下来假设Hanoi函数在较小的n时能完成工作, 那么用它递归即可:

(1) 利用一块布蒙住source上最底部的碟子, 并规定暂时不能接触和移动该碟. 从直观上看, 蒙布之后原问题可转换成$n-1$情况下的河内塔问题. 事实上, 被蒙住的是最大的碟子, 蒙住它不影响其他碟子的移动, 因为任何碟子都可以套在它之上, 因此这和$n-1$情况下的河内塔问题逻辑等价.
(2) 利用Hanoi函数解决$n-1$情况下的河内塔问题, 是从source到other(可用dest中转). 从某种意义上说, 这相当于改变了柱子之间的排放次序.
(3) 取掉蒙在source上的布, 将露出的碟子直接移动到dest上, 再重新将这块布蒙在dest上遮住碟子. 被蒙住的依然是最大的碟子, 蒙布之后当前问题仍可转换成$n-1$情况下的河内塔问题.
(4) 利用Hanoi函数解决$n-1$情况下的河内塔问题, 是从other到dest(可用source中转). 这同样改变了柱子之间的排放次序.
(5) 取掉蒙在dest上的布, 露出的碟子位于最底部, 完全满足游戏规则.

易证如此递归能不断减少河内塔问题的规模量, 最终使其到达基础情形.

◁ 河内塔问题求解: 原始版本 ▷

```
void Hanoi(int n,
        const string& source, const string& other, const string& dest)
{
    // 执行Hanoi函数的前提是n > 0.
    if (n == 1)
        cout << "move " << source << " to " << dest << endl;
    else
    {
        Hanoi(n - 1, source, dest, other);
        cout << "move " << source << " to " << dest << endl;
        Hanoi(n - 1, other, source, dest);
    }
}
```

▶ 河内塔问题求解: 原始版本 ◀

下面考虑该算法的正确性. 首先当n为1时输出结果显然正确. 其次若$n=k$时输出结果正确, 那么$n=k+1$时从算法设计的思路易知能利用n为k时的方案来得到正确的输出结果. 最后我们可知对于任意正整数n, Hanoi函数都可以在屏幕上输出正确的移动步骤. 不过我

们在后文将看到, 实际中Hanoi函数所能处理的参数值n只能在合理范围内, 上述证明仅是理论意义上的.

需要再次强调基础情形的重要性, 它是保证递归算法终止和算法正确性的重要一环. 本例在一个碟子($n = 1$)情况下输出移动步骤虽然简单, 但如果没有它递归不可能终止, 更不可能正确.

事实上, 如果假设计算机能像人一样看到柱子和碟子的情况, 那么算法就是不断调整计算机的思路, 通过改变柱子之间的排放次序等技巧让计算机机械地执行基本步骤. 读者可用纸制作柱子和碟子, 并将柱子涂上红、黄、蓝三种颜色, 按照上述方案移动便可清楚算法的实质.

当然, 对于有一定基础的程序员, 我们可以写出更好的代码实现. 其基础情形是$n = 0$(也可以认为是$n \leqslant 0$)情况, 对于此基础情形无需任何操作(也称空操作).

◁ 河内塔问题求解: 简洁版本 ▷

```
void Hanoi_SimpleVersion(int n,
        const string& source, const string& other, const string& dest)
{
    // 对于n <= 0情况无任何操作.
    if (n > 0)
    {
        Hanoi(n - 1, source, dest, other);
        cout << "move " << source << " to " << dest << endl;
        Hanoi(n - 1, other, source, dest);
    }
}
```

► 河内塔问题求解: 简洁版本 ◄

尽管上述代码的函数调用次数略多, 但更清晰而且容错性更好. ⋆

从河内塔问题中可以发现递归程序的设计模式. 一般设计递归程序都遵循如下思路:

(1) 设计一个函数接口并对它完成的功能给予精确的描述.
(2) 实现函数的基本功能以确保在基础情形下该函数能完成任务.
(3) 利用设计好的函数接口以递归完成该函数的内部构造, 必须充分利用函数的功能描述.
(4) 通过数学归纳法证明所设计算法的正确性.
(5) 特别要注意保证所设计的递归能终止于基础情形, 而且必须找出所有可能的基础情形.

在河内塔问题中. 我们首先假设Hanoi函数能完成在屏幕上输出移动n个碟子的具体步骤. 随后初步完成了基础情形的移动步骤. 然后利用已有的功能描述来完成Hanoi函数的内

部构造, 即通过两个$n-1$情况下的Hanoi函数调用和一个移动步骤来完成n情况下的函数实现. 事实上, 递归程序设计是构造算法框架, 而数学归纳法是证明框架的正确性, 两者互相依存, 缺一不可.

可以看到, 在递归程序的设计中, 提出函数接口及该函数能完成的功能是成功的第一步, 初学者一定要确信自己所描述的功能可以完成任务才能继续开展工作, 否则往往会对问题束手无策. 当然, 接口设计常要根据后面的具体实现情况不断修改, 难以一步到位, 熟练的程序员有着长期的经验积累能减少不必要的反复. 其次, 接口功能描述相当于数学归纳法证明时所设计的归纳假设, 必须加以大量的实践方能熟练运用, 注意有时候解决更强的问题即增强归纳假设会有意想不到的收获. 最后也是最重要的一点是, 递归只是给出了框架描述, 而完成函数的全部实现细节则依赖于对具体问题的分析, 要善于利用问题的特点设计出高效简洁的递归算法.

4.3 递归与进程模型

初看递归似乎是自己调用自己, 但事实并非如此, 其奥妙在于所调用"自己"的参数不同. 如果完全调用自己, 意味着调用一个与自身参数完全相同的函数, 一般肯定会导致无穷递归.

为阐明递归的工作方式, 不妨回到河内塔的问题上来. 设Hanoi函数初始的参数表值为(3, "A", "B", "C"), 那么一共存在7次Hanoi函数的调用, 我们按照函数完成的先后次序写出:

```
Hanoi(1, "A", "B", "C");        // H1函数
Hanoi(1, "C", "A", "B");        // H2函数
Hanoi(2, "A", "C", "B");        // H3函数
Hanoi(1, "B", "C", "A");        // H4函数
Hanoi(1, "A", "B", "C");        // H5函数
Hanoi(2, "B", "A", "C");        // H6函数
Hanoi(3, "A", "B", "C");        // H7函数
```

为方便起见我们对上述每个函数作了编号. 实际上这些函数构成了递归树, 如图4.1所示, 其中箭头表示调用关系.

尽管使用了递归, 但每个函数实体是相互独立的存在, 即便它们参数完全一致也不例外(例如H1和H5). 递归的唯一体现仅在于递归函数共用一段静态的程序代码, 若为每个函数分别保存数据以记录它执行到程序代码的哪一步, 并保证各个函数的局部数据与其他函数无关, 则可实现函数之间的独立性, 而上述只与具体函数执行有关的信息称为其上下文. 执行中的函数是动态的概念, 它类似于操作系统中的进程, 不妨为函数执行建立"函数进程"模型. 函数进程存在*新生*(被其他函数调用而创建), *运行*(执行非函数调用的代码), *等待*(调用

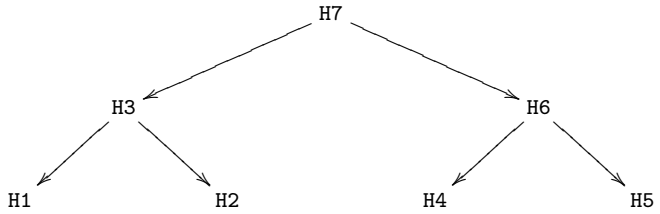

图 4.1 河内塔问题函数调用所形成的递归树示例

其他函数而自身暂停)和终止(运行结束)四种状态,[1] 很容易通过改变上下文达到切换函数进程的目的. 对于递归来说不是真地调用自己, 而是创建一个与自身拥有共同代码但包含独立上下文的函数进程.

依据上述模型可分析递归函数执行的过程, 注意我们考察的是串行计算, 因此任一时刻只能有一个函数进程处于运行状态. 例如H7首先被创建随即开始运行, 随后H7创建H3, 那么H3运行而H7暂停等待, 待H3终止后H7恢复运行, 尔后H7又创建H6, 在H6消亡后H7再次恢复运行, 不过迅即终止. 在这个过程中, 系统会自动管理调用(创建)者和被调用者之间的运行次序, 比如H7创建H3, 那么系统会在H3终止时唤醒H7继续运行. 此外, 函数暂停等待后若再次运行必须由系统完全恢复其上下文. 可以看到, 递归只是存在于人类思维中的概念而已, 真正的执行必须由系统加以控制方得实现. 事实上, 函数调用本身并不是真正递归的, 只是在*调用栈*(call stack)或称*控制栈*(control stack)上执行若干简单步骤而已. 更进一步说, 栈的操作也只是计算机按照简单的规则机械地执行语句, 这意味着递归确实只是一种描述技巧, 解决问题还得依靠若干函数进程实体来完成.

至此递归的实现原理便可明了. 由于已对递归函数所能完成的功能作了精确描述, 而且通过归纳法给出了证明, 那么对于互相独立的这些函数执行过程, 必然能全部按照要求完成任务且互不干扰, 因此最初开始执行的函数(上述例子中是H7)最终也顺利执行完毕.

允许递归的程序设计语言必须实现上述机制, 显然函数执行时的上下文乃至于调用者信息都比较容易保存, 而难点是如何保证调用次序. 由于函数执行完毕后只需追溯到其调用者即可, 这样自然就形成了调用栈, 它巧妙地利用了栈的特性从而高效方便地解决了调用次序问题, 具体实现细节我们不再展开讨论. 读者可自行思考如何在内存中高效组织数据, 其目标是以某种统一的方式实现数据快速切换.

4.4 递归算法性能分析

尽管递归算法表达明晰而且使用方便, 但常有不当使用. 例如有人会使用递归求阶乘, 显然不如直接利用定义$n! = \prod_{i=1}^{n}(i)$迭代效率更高. 由于阶乘的迭代实现去掉了递归所带来的函数调用耗费, 肯定速度更快. 需要指出, 阶乘递归仅在函数末尾才使用, 实为*尾递归*(tail recursion), 一般应消去.[2]

[1] 这里略去了*就绪*状态, 一旦所调用函数终止, 调用者将直接从等待状态转为运行状态.

[2] 消去阶乘的尾递归后则形成了反向迭代计算, 即从n连续乘到1.

对于一般情况下的递归程序首先需要在理论上分析其效率, 特别是时间复杂度, 只有这样才能决定是否应使用递归. 武断地说“递归比非递归慢”或者“递归比非递归快”都是不可取的. 选择了正确的方案, 接下来则可在代码级优化.[1]

需要指出, 尽管可自行利用栈模拟递归并优化, 但由人来做此工作未必能比调用栈和编译器完成得更好, 特别是一些较为复杂的递归算法. 这意味着一旦采用递归方案且算法设计完成后, 就应将重心放在如何写好递归程序上, 例如应尽量减少不必要的参数引入等.

例 4.2 直接利用递归程序生成Fibonacci序列的前m个元素, 并测试$m = 50$情况下的程序运行时间.

解: 很容易写出产生Fibonacci序列的递归程序:

◁ 生成Fibonacci序列 ▷

```
#include "book.h"

unsigned long long int Fibonacci(unsigned int n)
{
    // 返回值类型取unsigned long long int是为了避免溢出.
    if (n <= 1)
        return n;
    else
        return Fibonacci(n - 1) + Fibonacci(n - 2);
    // 为后文分析方便写成上述形式. 更简洁的表述是条件表达式.
}

void Fibonacci_sequence(unsigned int m)
{
    for (unsigned int i = 0; i < m; i++)
        cout << Fibonacci(i) << ' ' << i << endl;
}
```

▶ 生成Fibonacci序列 ◀

事实上, 该程序速度相当慢, 在一台CPU为Intel Core i5-3210且内存为4GB的笔记本上测试大约需要运行169秒. 显然可以写出更快的程序, 其时间不会超过`xTimer`的最小计时单位(即显示为0秒).

Fibonacci序列通常作为讲解递归的例子在众多教材中出现, 不过许多书并未告诫读者切不可直接递归计算它, 因此这里的实验结果对初学者有着很好的警示作用. ⋆

例 4.3 分析河内塔问题中`Hanoi`函数的时间复杂度, 并求$n = 64$情况下的碟子移动次数.

[1] 切勿过分优化, 有些优化工作大可交给编译器, 我们应将主要精力放在算法上. Donald E. Knuth有句名言: “We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil”.

解: 不妨记n个碟子情况下算法所需时间为T_n. 每次Hanoi函数执行时除了递归之外, 其他操作都是一些简单的判断和输出语句, 只需常数时间, 不妨设Hanoi函数执行该类操作的总时间不超过C, 易知当$n > 1$时

$$T_n \leqslant 2T_{n-1} + C,$$

进而可得

$$T_n + C \leqslant 2^{n-1}(T_1 + C),$$

而$T_1 \leqslant C$, 因此$T_n = O(2^n)$.

关于移动次数也可作类似分析, 易知当$n = 64$时需移动$2^{64} - 1$次(留作练习). 如果每秒可以移动两次, 则大约需要2925亿年, 远超过太阳所能供给能量的时间, 这也是为何在传说中宣称64个碟子移动完毕则到达世界末日的原因. ⋆

为更好地分析存在子程序调用情况下算法的时空效率, 可对其作一简单建模. 设算法A的程序代码中调用p个子程序$A_1, A_2, ...A_p$, 执行中各种耗费如表4.1所示.

表 4.1 算法存在子程序调用时的各项时空开销

耗费	说明
$t_i(1 \leqslant i \leqslant p)$	完成A_i所需时间
$c_i(1 \leqslant i \leqslant p)$	系统调用A_i的上下文切换时间(包括参数传递所需时间)
t_{p+1}	A中非子程序部分所需时间
c_0	系统调用A的上下文切换时间(包括参数传递所需时间)
$l_i(1 \leqslant i \leqslant p)$	专为完成A_i所需空间的峰值
l_{p+1}	A中不进行子程序调用时所需空间的峰值
$g_i(1 \leqslant i \leqslant p)$	A_i开始时A占用的空间(包括保存A的上下文所需空间)
g_{p+1}	取值为0

一般而言, 算法A在串行情况下所需总时间T和所需空间峰值S分别为:

$$T = \sum_{i=1}^{p+1} t_i + \sum_{i=0}^{p} c_i, \quad S = \max_{1 \leqslant i \leqslant p+1} \{l_i + g_i\}.$$

请注意, 尽管一般$p = \Theta(1)$, 即所写代码中的所调用函数个数为常数, 但有时p也可能与输入规模量n有关.

例 4.4 分析例4.2中Fibonacci函数的时间复杂度与空间复杂度, 并解释该方法为什么如此低效.

解: 设Fibonacci函数所处理的参数值为n, 我们将该函数的运行时间记为T_n, 而所需空间记为S_n. 由于Fibonacci函数的代码中调用了2个子程序(请注意不是执行过程中总调用次数), 于是$n > 1$时T_n满足

$$T_n = T_{n-1} + T_{n-2} + \Theta(1),$$

注意这种表述更具算法味, 而等式形式更利于理论分析. 事实上, 一般递归算法的时间复杂度都能转换为这样的*递推式*(Recurrence), 只需求解递推式即可得到较为精确的时间性能描述.

初步估计$T_n \leqslant 2T_{n-1} + \Theta(1)$, 于是$T_n = O(2^n)$. 更精确的结果是$T_n = O(\phi^n)$, 其中$\phi = (1+\sqrt{5})/2$, 可通过数学归纳法证明之[CLRS09].

之所以这个算法效率较低的原因是`Fibonacci`函数的重复调用过多, 读者可画出其递归树以验证. 而且更关键的是计算F_n时, 这种递归将原问题最终会分成n类子问题, 分别为计算$F_{n-1}, F_{n-2}, ..., F_0$, 而每类子问题的解决依赖于较小下标子问题的结果, 若直接递归需要重复计算相同子问题. 所以, 应将已计算的结果予以保存再顺次前进, 一旦轮到新的子问题只需直接查出已有结果来完成当前任务, 这种方法一般称为*动态规划*(dynamic programming). 使用动态规划可为计算Fibonacci序列设计一个线性算法. 不过, 还存在一个更快的$O(\log n)$时间算法([CLRS09]中问题31-3).

空间复杂度分析稍微简单一些. 当$n > 1$时S_n满足

$$S_n = \max\{S_{n-1}, S_{n-2}\} + \Theta(1),$$

容易知道$S_{n-1} \geqslant S_{n-2}$, 于是$S_n = \Theta(n)$. 实际上, 递归树的深度是影响递归算法空间复杂度的决定性因素, 读者可验证之. ⋆

一般求解递推式需要使用*生成函数*(generating function), 该方法虽然相当有效但需要较复杂的计算, 这里介绍一类常见模式的简单求解方法——主定理[CLRS09].

定理 4.1 [*主定理*(master theorem)] *设有常数$a > 1$, $b > 1$和定义在$\mathbb{N}$上的函数$f(n)$, 若"算法性能"函数$T(n)$满足递推式*

$$T(n) = aT(n/b) + f(n),$$

其中n/b可由$\lceil n/b \rceil$或$\lfloor n/b \rfloor$取代, 则$T(n)$随$f(n)$情况的不同而存在下列渐近记号:

(1) *若存在实数$\epsilon > 0$使得$f(n) = O(n^{\log_b a - \epsilon})$, 则$T(n) = \Theta(n^{\log_b a})$.*
(2) *若$f(n) = \Theta(n^{\log_b a})$, 则$T(n) = \Theta(n^{\log_b a} \log n)$.*
(3) *若存在实数$\epsilon > 0$使得$f(n) = \Omega(n^{\log_b a + \epsilon})$, 且存在实数$c < 1$能在较大的$n$情况下满足$af(n/b) \leqslant cf(n)$, 则$T(n) = \Theta(f(n))$.*

证明: 主定理的证明较为复杂, 可参见[CLRS09]. ∎

4.5 [实例] 排列生成器*

本节讨论一个稍微复杂的递归实例——排列生成器, 它在实际中特别是解决组合类问题非常有用. 该问题的提法是: 设有集合S, 任务是生成关于S中元素的所有*排列*(permutation). 常见的情形是处理集合$S = \{1, 2, ..., n\}$, 可讨论对该集合生成所有排列的过程. 它不但便于算法实现, 也不失一般性.

容易想到, 对$S = \{1, 2, ..., n\}$生成所有排列的简单方案是:

- 1为首元素, 后面配上$S - \{1\}$的所有排列;
- 2为首元素, 后面配上$S - \{2\}$的所有排列;

  ⋮

- n为首元素, 后面配上$S - \{n\}$的所有排列.

上述任一$S - \{i\}$也仍是一个集合, 因此问题可以递归下去. 递归过程中可以想象S的元素不断减少, 终止条件就是S变为空集, 这时应输出一个排列.

需要注意, 虽然当集合的元素个数为1时也可以终止, 但为了考虑在n个元素取k个排列的一般问题实现, 这里用空集来终止, 于是可以统一方便地过渡到一般情况. 显然, P_n^{n-1}尽管和P_n^n相等, 但实质有别.

从程序设计的角度看, 本问题中的集合S应选用存储方便且取用快捷的实现形式, 例如可存于int型数组A中, 其中前n个元素形成了S. 我们不必强求元素的次序, 例如A为1, 2, 3, 4和A为3, 1, 2, 4都对最终结果毫无影响, 仅仅是排列之间生成顺序不同而已. 由于排列生成器的处理对象是集合, 而集合的相等没有顺序上的要求, 只需要其中的元素一致即可. 因此, 在具体实现的时候不应将A看成数组, 在这里数组只是集合的承载形式而已.

容易想到, 对于数组A而言, 递归算法所完成的功能应表述为: 当A[0], ... , A[k - 1]固定时, 可产生轮换A[k], A[k + 1], ... , A[n - 1]而得到的所有排列, 并将所生成的每个排列拼接上固定前缀A[0], ... , A[k - 1]后在屏幕上输出. 因此permute函数的原型可设计为

```
void permute(int A[], size_t k, size_t n);
```

本节的问题是生成A中元素的所有排列, 只需使用permute(A, 0, n)即可, 因为没有元素固定, 便可得到{A[0], ..., A[n - 1]}的所有排列.

假设在permute(A, k, n)执行前, 数组中A[k], A[k + 1], ... , A[n - 1]的值分别为B[k], B[k + 1], ... , B[n - 1], 这些值形成集合S(k). 可设计permute的框架如下:

```
void permute(int A[], size_t k, size_t n)
{
    if (k == n)
        在屏幕上输出A[0]到A[n - 1];
    else
        for (size_t i = k; i < n; i++)
        {
            令A[k]所存放的值变为B[i], 而A[k + 1]到A[n - 1]存放S(k) - {B[i]}
            permute(A, k + 1, n);
        }
}
```

根据前述permute函数的功能设计要求, 结合上述程序框架知道permute函数的内部实现利用permute(A, k + 1, n)可以产生:

- B[k]为首元素配上S(k) − {B[k]}的所有排列;
- B[k + 1]为首元素配上S(k) − {B[k + 1]}的所有排列;

  ⋮

- B[n - 1]为首元素配上S(k) − {B[n - 1]}的所有排列;

产生上述每个排列后会拼接上固定前缀输出, 于是permute(A, k, n)在递归意义上可以完成预先设计的功能. 而终止条件是k等于n, 由于A[k]已经超越数组的边界, 即S(k)为空集这个基础情形, 我们只需简单输出固定前缀A[0]到A[k - 1](即A[0]到A[n - 1])即可.[1] 因此, 上述递归程序是正确的.

递归程序的框架尽管已经设计完毕, 但具体细节仍需斟酌. permute函数中最关键的一点是赋值的实现. 从设计上看{B[k],B[k + 1],...,B[n - 1]}可以作为一个permute函数执行时不能改变其值的数组B, 而这样会带来额外的存储空间, 还要频繁从B中复制元素到A中以便递归调用, 我们应考虑更好的策略. 另外, 我们还应使用更方便的向量来实现permute函数. 下面讨论如何解决上述难点.

4.5.1 利用vector传值实现

考虑用vector<int>型向量V存储S, 难点在于产生不同的V − {i}. 我们希望每次所产生的V − {i}最好与其他集合互不相关, 那么可以考虑传值的策略, 也即每次复制V的值传入而不改变它, 因此可修改permute函数原型为

```
void permute(vector<int> V, size_t k);
```

由于V作为参数传值相当于在permute函数内部多复制了一个向量, 因此不再需要前文中所提到的数组B, 于是permute的框架细化如下:

```
void permute(vector<int> V, size_t k)
{
    if (k == V.size())
        在屏幕上输出V[0]到V[V.size() - 1];
    else
        for (size_t i = k; i < V.size(); i++)
        {
            交换V[k]与V[i];
            permute(V, k + 1);
        }
}
```

[1] 本章注释部分将严格地论证空集的排列问题.

实际上, 上述函数permute已经基本接近于程序实现, 可自行编写实际代码.

设向量V中元素为1, 2, 3, 下面以此为例具体分析permute(V, 0)的执行过程:

1. 交换V[0]和V[0], V变成1, 2, 3. 再执行permute(V, 1)相当于给出{2,3}即V − {1}, 执行完permute(V, 1)后, V保持1, 2, 3不变.
2. 交换V[0]和V[1], V变成2, 1, 3. 再执行permute(V, 1)相当于给出{1,3}即V − {2}, 执行完permute(V, 1)后, V保持2, 1, 3不变.
3. 交换V[0]和V[2], V变成3, 1, 2. 再执行permute(V, 1)相当于给出{1,2}即V − {3}, 执行完permute(V, 1)后, V保持3, 1, 2不变.

注意这里产生V − {i}的方法, 它完全能满足前面的要求, 因为permute(V, k + 1)传递的是V的值, 因而是正确的(其证明作为练习). 不过, 这种方案效率很低, 因为每次要复制传递长为n的向量V, 还要作为函数进程的上下文信息保存, 多次函数调用之后就会形成极大的负担.

4.5.2 利用vector引用实现

使用引用是一个较好的方法, 但交换的模式需要考虑. 如果仅将函数的参数表改为

```
void permute(vector<int>& V, size_t k);
```

而仍采用传值时的函数框架显然是不够的. 仍设向量V中元素为1, 2, 3, 我们以此为例分析之. 下面的数据由实际程序(可运行permute(V, 0))调试结果中得来, 具体如下:

1. 交换V[0]和V[0], V变成1, 2, 3. 再执行permute(V, 1)相当于给出{2,3}即V − {1}, 执行完permute(V, 1)后, V变为1, 3, 2.
2. 交换V[0]和V[1], V变成3, 1, 2. 再执行permute(V, 1)相当于给出{1,2}即V − {3}, 执行完permute(V, 1)后, V变成3, 2, 1.
3. 交换V[0]和V[2], V变成1, 2, 3. 再执行permute(V, 1)相当于给出{2,3}即V − {1}, 执行完permute(V, 1)后, V变成1, 3, 2.

在程序执行过程中V − {1}会重复出现, 导致结果不正确. 该方法的错误原因在于不能保证按permute函数功能要求向下递归, 上述数据结果亦可以纸笔分析, 更能加深对错误的认识.

从引用的实质看, 直接写引用表示其参数值可能在函数体内发生改变, 也即下述方式实质一样:

```
void permute(vector<int>& V, size_t k);
void permute(int a[], size_t k, size_t n);
```

所以应想其他方案解决该问题. 顺便提及, 如果想使用常量引用

```
void permute(const vector<int>& V, size_t k);
```

来获得正确算法是不可行的, 因为在函数体内有交换V中元素的操作, 此类操作由于const的限制而不允许执行. 尽管这种“语法”上的限制难以成立, 但它提示我们可以从“语义”上保证数据不变.

要解决问题, 需要保证引用情况交换后保持不变, 也即利用“先使用再恢复”的方式进行. 具体策略是在使用后将V[k]和V[i]交换回来, 这样既保持了V的状态, 还不带来额外的负担. 于是新的程序框架如下:

```
void permute(vector<int>& V, size_t k)
{
    if (k == V.size())
        在屏幕上输出V[0]到V[V.size() - 1];
    else
        for (size_t i = k; i < V.size(); i++)
        {
            交换V[k]与V[i];
            permute(V, k + 1);
            交换V[k]与V[i];
        }
}
```

事实上, 这对于只有数组的程序设计语言来说也是很好的方案, 只需稍微转换参数形式即可. 要知道, 不是任何语言都有向量容器, 而排列问题又具有普遍性, 因此能用数组实现该算法是相当必要的.

例 4.5 证明并分析: 当0 ⩽ k ⩽ V.size() = n时, 若使用向量引用方式和交换后恢复策略, 任意permute(V, k)运行后不改变V[k], V[k + 1], ... , V[n - 1]的值. 换言之, 该程序的执行不改变整个V的状态.

证明: 需要使用逆向归纳. 容易知道执行permute(V, V.size())后显然不改变V的状态, 这是归纳基础. 或者考虑permute(V, V.size() - 1)后不改变V的状态亦可.

假设执行permute(V, k + 1)后不改变V[k + 1], V[k + 2], ... , V[n - 1]的值. 注意到在permute(V, k)的运行过程中, 每次调用permute(V, k + 1)函数前后都存在交换V[k]和V[i]的操作, 因此, 执行permute(V, k)后也不会改变V[k], V[k + 1], ... , V[n - 1]的值. 再利用归纳基础, 便可证明本例中的结论.

为了清晰展示程序运行情况, 可给出图4.2以描述每次交换、排列、交换的步骤中数据变换情况. 显然V依然是原来的状态.

利用V不改变的结论再综合上述讨论, 可知每次一定会产生V − {i}的集合, 因此permute函数可以产生指定要求的所有排列. 此外, permute(V, 0)之后, V的状态保持不变. 但请注意, 切不要误认为permute函数是就地算法. ∎

可以看到, 尽管向量V贯穿整个递归过程, 但我们未将其作为全局对象使用, 其原因是面向对象程序设计不鼓励这种用法. 然而频繁传递V在上述递归程序中毫无必要, 读者可思考如何解决此问题. 需要指出, V在使用时具有一定的局部特性, 例如不能对一份V的数据直接并行执行本节中的递归调用, 而我们对其设计了两次交换的精巧方案, 避免V中元

初始数据情况为								
元素	V[k]	V[k + 1]	...	V[i - 1]	V[i]	V[i + 1]	...	V[n - 1]
当前值	b[k]	b[k + 1]	...	b[i - 1]	b[i]	b[i + 1]	...	b[n - 1]
交换V[k]和V[i]后为								
元素	V[k]	V[k + 1]	...	V[i - 1]	V[i]	V[i + 1]	...	V[n - 1]
当前值	b[i]	b[k + 1]	...	b[i - 1]	b[k]	b[i + 1]	...	b[n - 1]
由于执行permute(V, k + 1)后V的状态不变, 执行后仍为								
元素	V[k]	V[k + 1]	...	V[i - 1]	V[i]	V[i + 1]	...	V[n - 1]
当前值	b[i]	b[k + 1]	...	b[i - 1]	b[k]	b[i + 1]	...	b[n - 1]
再次交换V[k]和V[i]后为								
元素	V[k]	V[k + 1]	...	V[i - 1]	V[i]	V[i + 1]	...	V[n - 1]
当前值	b[k]	b[k + 1]	...	b[i - 1]	b[i]	b[i + 1]	...	b[n - 1]

图 4.2 permute函数执行过程中数据变换情况展示

素作为函数进程的局部数据从而节约了大量的空间. 事实上, 涉及此类引用参数(reference parameter)要特别小心, 必须给出证明以保证数据正确性, 而且在没有必要的情况下应尽量避免使用.

4.6 [实例] 乐高铺砖

乐高(LEGO)是风靡全球的积木玩具品牌, 它设计精美并且变幻无穷, 深受消费者喜爱. 本节需要处理一个有趣的问题: 假定我们有一张$2^m \times 2^m (m \in \mathbb{N})$的底板(baseplate), 其中有一个位置已被一块1×1平瓦(flat tile 1×1)覆盖, 任务是利用若干$1 \times 2 \times 2$角型砖(brick corner $1 \times 2 \times 2$)将剩下的位置铺满. 图4.3给出了一个较为具体的示例.

解决乐高铺砖问题的关键是*分治*(divide and conquer), 即尽力将原问题划分成更小的同类型子问题. 由于$m > 1$情况下$2^m \times 2^m$的底板已经有一个被覆盖的位置, 只需在中心放置一块$1 \times 2 \times 2$角型砖, 使得铺下此砖后原底板可分成4块$2^{m-1} \times 2^{m-1}$同类型的较小底板即可, 也就是说这4块$2^{m-1} \times 2^{m-1}$的小底板中每块都有一个位置已被覆盖. 于是可以写出相应的程序实现, 并对图4.3输出解答.

这个问题的基础情形可以取边长为2, 但取边长为1形式更为简洁. 基础情形取边长为1以空操作结束递归, 注意该空操作蕴含在不满足if语句判断的分支里, 不过这将多调用一次函数. 基础情形若取边长为2, 则需在代码中补充单独针对该情形下的编号赋值操作的语句. 实际上, 许多递归算法都存在这样的空操作形式, 有的甚至无法避免. 由于引入空操作可能需要更多的函数调用, 要仔细斟酌考虑权衡代码形式与程序效率之间的关系.

考虑到本问题的具体情况, 我们以类的方式实现, 这样有助于提高内聚性. 由于baseplate和label在贯穿整个求解过程且要保持数据单一性, 因此它们不能作为SolveLEGO函数的参数, 否则频繁将其作为上下文频繁保存会降低效率. 需要注意, 递归调用时处理baseplate是相互独立的, 而label变量在使用时则具有一定的局部特性,

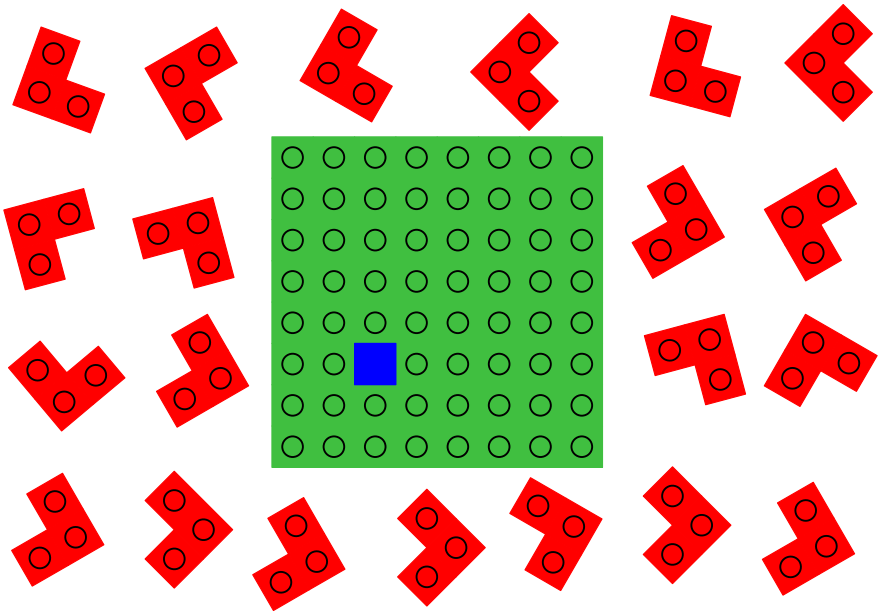

图 4.3 一块8 × 8底板和21块1 × 2 × 2角型砖, 底板上有一孔已装上1 × 1平瓦

证明其数据正确性留作练习. 此外, 若要并行执行上述算法, 则应预先计算出label的取值范围并使用多个变量分别标记.

◁ xLEGO.h ▷

```
#include "book.h"

#ifndef LEGO_CLASS
#define LEGO_CLASS

class LEGO {
public:
    LEGO(int N, int init_spot_x, int init_spot_y);
    // 构造函数.
    void Solve();
    // 求解.
    void Output();
    // 在屏幕上输出解法.
private:
    void SolveLEGO(int N, int start_x, int start_y,
        int spot_x, int spot_y);
    // N为所求解底板当前边长, start_x和start_y为该底板左上角坐标,
    // spot_x和spot_y为当前底板中已铺的位置.
    matrix<int> baseplate;          // 存储底板编号数据的矩阵.
```

```
    int init_spot_x, init_spot_y;   // 初始已铺平瓦的位置坐标.
    int length;                     // 底板长度.
    int label;                      // 编号, 为新铺砖标记编号且不断增加.
};

#endif
```

▶ xLEGO.h ◀

◁ xLEGO.cpp ▷

```
#include "xLEGO.h"
#include "book.h"

LEGO::LEGO(int N, int given_spot_x, int given_spot_y)
    : baseplate(N, N, -1), length(N),
        init_spot_x(given_spot_x), init_spot_y(given_spot_y)
{
    // 初始已铺平瓦位置编号为0.
    baseplate(init_spot_x, init_spot_y) = 0;
}

void LEGO::SolveLEGO(int N, int start_x, int start_y,
    int spot_x, int spot_y)
{
    if (N > 1)
    {
        int half = N / 2;
        int i, j;
        // spot_x_temp和spot_y_temp描述更小底板的已铺位置坐标.
        int spot_x_temp[2][2];
        int spot_y_temp[2][2];
        // 先统一给出中心位置4个点的坐标, 也可用for语句实现(但稍慢).
        spot_x_temp[0][0] = start_x + half - 1;
        spot_y_temp[0][0] = start_y + half - 1;
        spot_x_temp[0][1] = start_x + half - 1;
        spot_y_temp[0][1] = start_y + half;
        spot_x_temp[1][0] = start_x + half;
        spot_y_temp[1][0] = start_y + half - 1;
        spot_x_temp[1][1] = start_x + half;
        spot_y_temp[1][1] = start_y + half;
        // 再将中心位置4个点中不需要铺砖的位置为真正的已铺位置.
```

```
        int p = (spot_x - start_x) / half;
        int q = (spot_y - start_y) / half;
        spot_x_temp[p][q] = spot_x;
        spot_y_temp[p][q] = spot_y;
        // 暂时保存不需要铺砖的位置原有的值.
        int temp = baseplate(spot_x_temp[p][q], spot_y_temp[p][q]);
        // 对中心位置进行一次铺砖操作
        baseplate(spot_x_temp[0][0], spot_y_temp[0][0]) = label;
        baseplate(spot_x_temp[0][1], spot_y_temp[0][1]) = label;
        baseplate(spot_x_temp[1][0], spot_y_temp[1][0]) = label;
        baseplate(spot_x_temp[1][1], spot_y_temp[1][1]) = label;
        // 恢复不需要铺砖的位置原有的值, 此方式避免了判断语句.
        baseplate(spot_x_temp[p][q], spot_y_temp[p][q]) = temp;
        // 所铺砖的编号增1, 为下次铺砖准备.
        label++;
        // 最后平均分割为4块较小尺寸的底板进行递归铺砖.
        SolveLEGO(half, start_x, start_y,
            spot_x_temp[0][0], spot_y_temp[0][0]);
        SolveLEGO(half, start_x, start_y + half,
            spot_x_temp[0][1], spot_y_temp[0][1]);
        SolveLEGO(half, start_x + half, start_y,
            spot_x_temp[1][0], spot_y_temp[1][0]);
        SolveLEGO(half, start_x + half, start_y + half,
            spot_x_temp[1][1], spot_y_temp[1][1]);
    }
}

void LEGO::Solve()
{
    label = 1;
    SolveLEGO(length, 0, 0, init_spot_x, init_spot_y);
}

void LEGO::Output()
{
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
            cout << setw(5) << baseplate(i, j);
```

```
        cout << endl;
    }
}
```
► xLEGO.cpp ◄

我们的程序可在屏幕上输出角型砖编号, 每块角型砖对应的3个位置编号相同, 初始平瓦编号为0. 图4.4给出了图4.3的解答.

```
 3  3  4  4  8  8  9  9
 3  2  2  4  8  7  7  9
 5  2  6  6 10 10  7 11
 5  5  6  1  1 10 11 11
13 13 14 14  1 18 19 19
13 12  0 14 18 18 17 19
15 12 12 16 20 17 17 21
15 15 16 16 20 20 21 21
```

图 4.4 乐高铺砖输出实例

SolveLEGO函数的时间复杂度分析可根据主定理求得. 设问题规模量$n = 2^m \times 2^m$, 且算法时间复杂度为$T(n)$, 于是$T(n) = 4T(n/4) + \Theta(1)(n > 1)$, 因此$T(n) = \Theta(n)$, 也即该算法是线性算法. 此外, 主定理还有助于我们更好地设计算法. 注意到主定理中a是子问题个数, 而b决定了子问题规模量, 由于它们的取值对算法性能起关键作用, 那么在划分问题之初就应考虑a和b是否合适, 否则不要继续下一步的细化工作, 而应尽量去寻找更合适的划分方案.

本节问题的求解过程提示我们, 若要使用分治算法则应尽量均匀地对问题进行分割, 否则难以达到效果. 实际上, 保持*均衡*/*平衡*(balance)是算法和数据结构领域乃至于计算机科学都应遵循的一个基本原则.

章节注释

递归与可计算性

递归是计算机科学的基石之一, 其理论意义重大. Church-Turing论题指出, 任何计算设备都可由Turing机模拟, 也即Turing机决定了是否可以计算. 递归函数已证明与Turing机是等价的模型, 因此这意味着可计算函数(computable functions)[1]就是递归函数, 有兴趣的读者可参阅*可计算性*(computability)或者*递归论*(recursion theory)的相关著作.

尽管递归是算法乃至于计算的本质属性之一, 而且其形式简单优美, 但切勿夸大递归的作用. 事实上, 计算存在多种在理论上互相等价的形式, 应依具体情况而选择合适的模型. 例如用RAM模型对算法进行理论分析较为方便, 而我们一般采用计算机运行程序.

[1] 早期称为能行可计算函数(effectively calculable function), 直观地说就是我们能够去实施并完成计算的函数.

归纳与算法设计

[Man89]从归纳的观点来审视并组织众多算法, 是一本极具特色的教材. 该书将归纳法作为方法论来指导算法设计, 非常值得一读. 不过作者也提到, 归纳法不具备通用性, 也即算法设计还有更多的内容值得研究和学习.

排列

集合S的所有排列是一个由排列组成的集合, 记为$P(S)$. 需要注意的是空集$\varnothing$的所有排列, 它是包含一个特殊元素ϵ的集合, 即$P(\varnothing) = \{\epsilon\}$. 注意$\epsilon$相当于一个没有体积的空符号, 一般也不显示. 一般而言, $|P(S)| = |S|!$, 这里恰好有$|P(\varnothing)| = 0! = 1$.

对于$t > 0$, 不妨定义S中元素x和S上的t元组$\boldsymbol{y} = (y_1, y_2, ..., y_t)$的"拼接"运算$\circ$为

$$x \circ \boldsymbol{y} = (x, y_1, y_2, ..., y_t),$$

其中$y_i \in S, 1 \leqslant i \leqslant t$. 也即$x \circ \boldsymbol{y}$形成了一个$S$上的$(t+1)$元组. 再定义$S$中元素$x$和非空集合$U \subset S$的拼接运算$\circ$为

$$x \circ U = \bigcup_{u \in U} \{x \circ u\},$$

于是非空集合S的所有排列是

$$P(S) = \bigcup_{x \in S} (x \circ P(S - \{x\})),$$

并令任意$x \in S$满足$x \circ \epsilon = x$, 这样排列即获得了递归定义, 其基础情形是$P(\varnothing) = \{\epsilon\}$, 这就解释了本章中`permute`函数基础情形(`n`等于`k`)下的输出.

生成$\{1, 2, ..., n\}$的所有排列这个问题一般称为*排列生成*(permutation generation). 本章所讨论的方案效率不是很高, 其原因是存在许多不必要的元素交换操作. 有兴趣的读者可以参阅[Knu11], 其中详尽地讨论了各种高效的排列生成算法. 此外, STL为我们提供了`next_permutation`和`prev_permutation`函数, 可按一定的规则循次生成所有排列, 建议在仅处理排列生成问题时使用它们.

4.7 习题

1. 有的人爱定计划而从不实施: 从年计划定到月计划, 再从月计划分解到周计划, 周计划还分每天的计划, 结果一事无成. 从递归的角度分析这种行为.
2. 当河内塔问题中所输入碟子个数为负值时, 我们要求在屏幕上输出提示语句. 比较下面两段代码, 分别从效率和处理分支次序两方面分析.

◁ 河内塔问题求解：处理负数情况 ▷

```
void Hanoi(int n,
        const string& source, const string& other, const string& dest)
{
    if (n == 1)
        cout << "move " << source << " to " << dest << endl;
    else if (n > 1)
    {
        Hanoi(n - 1, source, dest, other);
        cout << "move " << source << " to " << dest << endl;
        Hanoi(n - 1, other, source, dest);
    }
    else
        cout << "n is negative!" << endl;
}
```

▶ 河内塔问题求解：处理负数情况 ◀

◁ 河内塔问题求解：判断次序变更 ▷

```
void Hanoi(int n,
        const string& source, const string& other, const string& dest)
{
    if (n > 1)
    {
        Hanoi(n - 1, source, dest, other);
        cout << "move " << source << " to " << dest << endl;
        Hanoi(n - 1, other, source, dest);
    }
    else if (n == 1)
        cout << "move " << source << " to " << dest << endl;
    else
        cout << "n is negative!" << endl;
}
```

▶ 河内塔问题求解：判断次序变更 ◀

如果只能二者选一, 你会选哪种方案? 此外, 你能否提出更好的解决方案? 问题何在?

3. 分析排列生成器算法的时间复杂度和空间复杂度并证明.

4. 为Ackermann函数设计动态规划求解算法, 并比较该算法与直接利用递归定义求解方案的优劣. 需要注意Ackermann函数增长很快, 不妨取函数值模2的余数.

5. 有人认为: 算法的本质是递归函数, 因此我们必须全用递归设计并实现算法. 这种说法合理么?

6. 自行编写程序解决乐高铺砖问题, 并对比本章中的解决方案分析其优劣.

7.(*) 设计算法以$O(\log n)$时间求出Fibonacci序列中的F_n.