第 3 章　常用的算法思想

对于计算机科学而言，算法（Algorithm）是一个非常重要的概念。它是程序设计的灵魂，它是将实际问题同解决该问题的计算机程序建立起联系的桥梁。可以这样讲，在编写任何一个计算机程序时（无论使用什么编程语言），都不可避免地要进行算法的设计。本章将重点介绍算法的基本概念，以及一些常用的算法思想。

3.1　什么是算法

一个程序往往要包含两个方面的描述：一是对数据组织的描述；二是对程序操作流程的描述。对数据组织的描述主要是指定数据的类型和数据的组织形式（例如数组、结构等），称做数据结构（Data Structure）。对程序操作流程的描述就是程序的执行步骤，也就是本节所要介绍的算法（algorithm）。正如 Nikiklaus Wirth 提出的公式：

数据结构+算法=程序

一样，算法是程序中不可缺少的一部分。如果把一个可运行的程序比喻成一个具有生命的人，那么数据结构就是这个人的躯体和骨架，而算法则是这个人的灵魂或者精神。

所谓算法，广义地讲就是解决问题的方法和过程。例如洗衣服的过程就可描述为以下几步：

（1）用盆盛足量的清水。

（2）将要洗的衣物浸入水中。

（3）放入洗衣粉进行清洗。

（4）用清水漂洗。

（5）拧干衣物进行晾晒。

那么上述 5 步就可以叫做完成洗衣服这项工作的算法。

在计算机领域，算法有其更为严格的定义：若干条指令组成的有穷序列，如果它满足以下几条性质：

（1）输入：有 0 个或多个由外部提供的值作为算法的输入。

（2）输出：产生至少一个量作为输出。

（3）确定性：组成算法的每条指令确定无二义。

（4）有限性：算法中每条指令执行的次数是有限的，每条指令的执行时间也是有限的。

以下讨论的算法都是指这种计算机领域所描述算法。

3.2　算法的分类表示及测评

“算法”只是一个笼统的叫法。在计算机科学领域，将解决不同性质问题的算法划分成为不同的类型。另外，算法本身也存在优劣之分，因此必须有一套完整的评价体系去测评某一个算法的性能，从而给出该算法一个客观的评价。本节将对算法的分类、表示以及算法的测评进行简要介绍。

3.2.1　算法的分类

计算机算法可分为两大类，一类叫做数值算法，它主要是解决一些工程上的数值计算问题，例如数值积分、数值求解微分方程等，有一门叫做《数值分析》或《计算方法》的课程是专门讨论这类数值算法的，有兴趣的读者可以参考学习。还有一类叫做非数值算法，它主要用于解决那些非数值的计算机问题，一般的程序设计中使用的算法多为这类非数值算法。

3.2.2　算法的表示

宽泛地讲，不管用哪种形式描述一个解题过程，只要它逻辑清晰，结果正确，哪怕就是在脑子里构思的算法也是好的算法。但是在解决实际问题时，问题往往比较复杂，并不是可以直接用大脑就想得清楚的，因此就需要借助一些简单、清晰的“描述语言”来辅助大脑构建解题过程。这也就是算法的表示形式。

算法的表示形式很多。有以下几种表示形式可供参考。

1．用自然语言描述

如同上面描述的洗衣服的过程那样就是一个用自然语言描述的算法。除了这种很简单的问题，一般不用自然语言描述计算机算法，因为自然语言存在二义性，表达起来并不十分清晰和精准。

2．用流程图表示

这是一种使用最为普遍的算法表示方法，其主要优点是简单直观，便于理解。如图 3-1 所示，给出了流程图的图元表示方法。

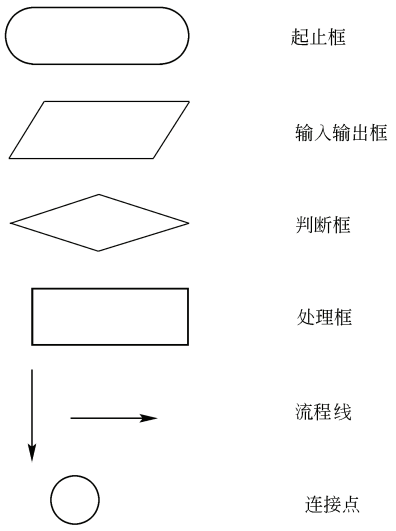


图 3-1　流程图的图元表示方法

下面通过一个例子来理解用流程图表示算法。

【实例 3-1】用流程图描述判断一个数 *i* 是否为素数的算法。

如图 3-2 所示为“判断一个数 *i* 是否为素数”算法的流程图。

3．用NS流程图表示

1973 年美国学者提出了一种新型流程图：N-S 流程图。N-S 流程图表示方法如图 3-3 表示。

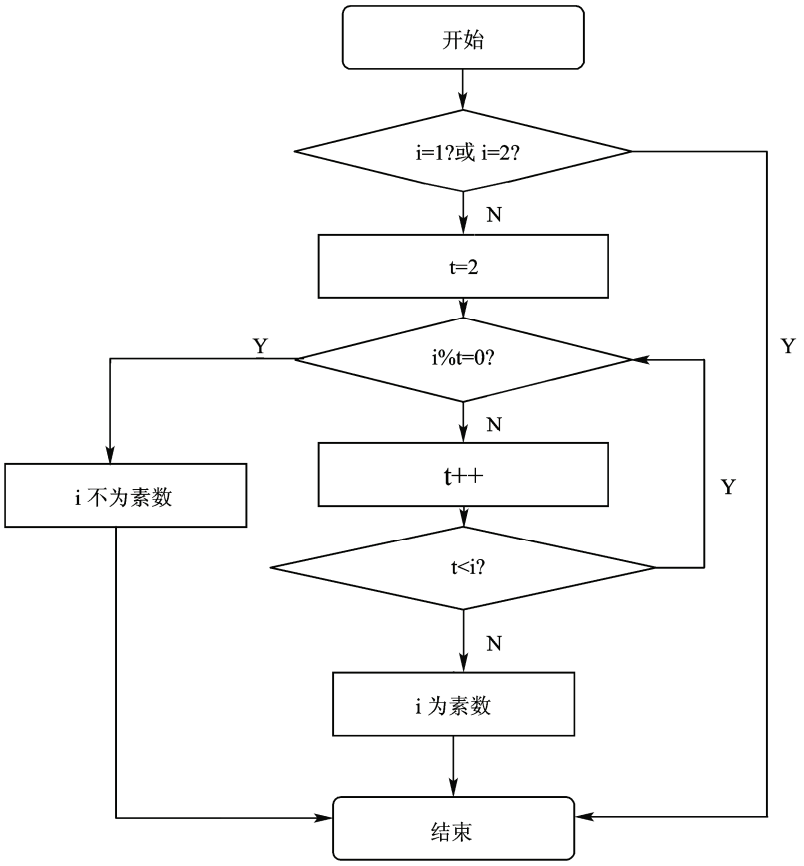
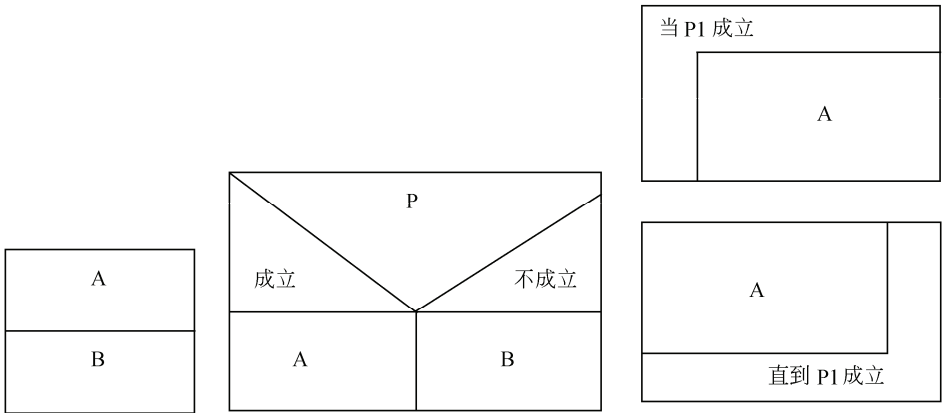


图 3-2　流程图的算法描述



（1）顺序结构　　（2）分支结构　　（3）循环结构

图 3-3　NS 流程图的表示方法

4．用伪代码表示算法

所谓伪码（pseudocode）是指帮助程序员制定算法的智能化信息语言。伪代码使用介于自然语言和计算机语言之间的文字和符号来描述算法。伪代码程序并不能在计算机上运行，它只作为程序员设计程序之前的一种辅助工具。因此伪代码并没有固定的语法和格式，常根据程序员的习惯而定，随意性很大。

【实例 3-2】用伪代码描述判断一个数 i 是否为素数的算法。

```
If  i=1 or i=2
Then    i 为素数，程序结束
Else
t←2
Repeat:
        If  i mod t =0
            Then  i 不为素数，程序结束
        Else  t←t+1
Until  t>=i
i 为素数，程序结束
```

以上这段描述就是判断一个数 *i* 是否为素数的伪码算法描述。可以看出，伪码描述较流程图算法描述更加接近程序代码形式，因此也更加容易转换为实际的程序。

3.2.3　算法性能的测评

既然算法是解决实际问题的方法，那么算法就必然存在着好坏优劣之分。一个算法的好坏是有指标能够加以测评的。这个指标通常称为算法的复杂度。

算法的复杂度体现在运行该算法时所需要的系统资源开销上。如果计算机执行一个算法时所需要的系统资源开销很大，就说这个算法复杂度很高。相反，如果计算机执行一个算法时所需要的系统资源开销很小，那么就说这个算法复杂度很低。在设计算法时自然是希望算法的复杂度越低越好。

由于计算机最重要的资源就是时间资源和系统的空间资源，因此，算法的复杂度分为时间复杂度和空间复杂度。要想更深一步地探讨算法的复杂度问题，可以参看《算法分析》等书籍。

3.3　穷举法思想

穷举法（Exhaustive Attack method），又称为强力法（Brute-force method），它是一种最为直接、实现最为简单，同时又最为耗时的一种解决实际问题的算法思想。本节将详细介绍穷举法的算法思想。

3.3.1　基本概念

穷举法算法的基本思想是：在可能的解空间中穷举出每一种可能的解，并对每一个可能解进行判断，从中筛选出问题的答案。

使用穷举法思想解决实际问题，最关键的步骤是划定问题的解空间，并在该解空间中一一枚举每一个可能的解。这里有两点需要注意。一是解空间的划定必须保证覆盖问题的全部解。如果解空间集合用 H 表示，问题的解集用 h 表示，那么只有当 $h \subset H$ 时，才能使用穷举法求解。二是解空间集合及问题的解集一定是离散的集合，也就是说集合中的元素是可列的、有限的。

穷举法用时间上的牺牲换来了解的全面性保证，因此穷举法的优势在于确保得到问题的全部解，而瓶颈在于运算效率十分低下。但是穷举法算法思想简单，易于实现，在解决一些规模不是很大的问题上，使用穷举法不失为一种很好的选择。另外，随着计算机硬件性能的不断改善，CPU 运算速度的不断提高，以及多处理器并行计算技术的发展，穷举法的形象已经不再是最低等和最原始的无奈之举，它将越来越为人们所重视。

下面通过具体的实例来理解穷举法思想。

3.3.2　寻找给定区间的素数

【实例 3-3】寻找[1，100]之间的素数。

【分析】

解决这个问题最简便的方法就是使用穷举法。在[1，100]中对每一个整数进行判断，看它是不是素数。在这里，问题的解空间自然就是[1，100]中的全部整数，因为不会有任何一个解超出这个范围，同时该解空间构成的集合元素是可列有限的。算法描述如下：

```
for(i=1;i<=100;i++)
    if (i 是素数) 输出 i ;
```

判断一个数是否是素数的算法描述在 3.2 节中已详细介绍，这里不再赘述。

程序清单 3-1

```
/*------------------------------ 3-1.c  ----------------------------*/
#include "stdio.h"
int isPrime(int n)
{               /*判断 n 是否是素数，如是则返回 1，如不是则返回 0*/
    int i;
    for(i=2;i<n;i++)
    {
        if(n%i==0)return 0;  /*n 可以被 i 整除，因此 n 不是素数，返回 0*/
    }
    return 1;               /*n 是素数，返回 1*/
}

getPrime(int low,int high)
{               /*寻找[low,high]之间的素数*/
    int i;
    for(i=low;i<=high;i++) {
        if(isPrime(i)) {
            printf("%d ",i);      /*如果 i 是素数，则打印出来*/
        }
     }
}
```

```
main()
{
    int low,high;
    printf("Please input the domain for searching prime\n");
    printf("low limitation:");
    scanf("%d",&low);
    printf("high limitation:");
    scanf("%d",&high);
    printf("The whole primes in this domain are\n");
    getPrime(low,high);
    getche();
}
```

本程序的运行结果如图 3-4 所示。

```
Please input the domain for searching prime
low limitation:1
high limitation:100
The whole primes in this domain are
1 2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

图 3-4　例 3-3 的运行结果

3.3.3　TOM 的借书方案

【实例 3-4】TOM 共有 5 本新书，要借给 A、B、C 3 位同学，每人只能借 1 本书，则 TOM 可以有多少种不同的借书方法。

【分析】

这个问题仍然可以用穷举法轻松地解决。假设 TOM 的 5 本书编号为{1，2，3，4，5}，每个同学可能借到的书的范围就限定在{1，2，3，4，5}之中。因此 TOM 借书给 3 位同学的组合方案不可能超过 5^3=125 种。由这 125 种借书方案构成的解空间可描述为$\{(x_1,x_2,x_3)|1\leqslant x_i\leqslant 5$，且 $x_i\in R\}$，该解空间是由 3 个各包含 5 个元素（图书编号）的集合排列组合而成。可以应用穷举法在该解空间中搜索答案。

解空间中哪些元素是该问题的解呢？当然问题的解不可能就是解空间中这 125 种借书方案，因为在这 125 种借书方案中还包含着不符合实际要求的排列组合。由于 1 本书一次只能借给 1 位同学，因此在每一种借书方案中，元素有重复的排列组合一定不会是问题的答案。于是可以得出 TOM 借书方案的解集为：$\{(x_1,x_2,x_3)|1\leqslant x_i\leqslant 5$，且 $x_i\in R$，$x_1\neq x_2\neq x_3\}$。这个解集可通过穷举解空间集合中的每一个元素，并添加判断条件 $x_1\neq x_2\neq x_3$ 获得。

综合上述，该问题算法描述如下：

```
for(i=1;i<=5;i++)
    for(j=1;j<=5;j++)
        for(k=1;k<=5;k++)
            if(i!=j && j!=k && i!=k) {得到一种借书方案;}
```

该算法中，变量 i、j、k 表示图书的编号。

程序清单 3-2

```
/*-------------------------------- 3-2.c  ----------------------------*/
#include "stdio.h"
```

```
main()
{
    int i,j,k;
    printf("There are different methods for TOM to distribute his books to
    A,B,C\n");
    for(i=1;i<=5;i++)
        for(j=1;j<=5;j++)
            for(k=1;k<=5;k++)
                if(i!=j && j!=k && i!=k){
                    printf("(%d,%d,%d) ",i,j,k);  /*输出一种借书方案*/
                    }
    getche();
}
```

本程序的运行结果如图 3-5 所示。

```
There are different methods for TOM to distribute his books to  A,B,C
(1,2,3) (1,2,4) (1,2,5) (1,3,2) (1,3,4) (1,3,5) (1,4,2) (1,4,3) (1,4,5) (1,5,2)
(1,5,3) (1,5,4) (2,1,3) (2,1,4) (2,1,5) (2,3,1) (2,3,4) (2,3,5) (2,4,1) (2,4,3)
(2,4,5) (2,5,1) (2,5,3) (2,5,4) (3,1,2) (3,1,4) (3,1,5) (3,2,1) (3,2,4) (3,2,5)
(3,4,1) (3,4,2) (3,4,5) (3,5,1) (3,5,2) (3,5,4) (4,1,2) (4,1,3) (4,1,5) (4,2,1)
(4,2,3) (4,2,5) (4,3,1) (4,3,2) (4,3,5) (4,5,1) (4,5,2) (4,5,3) (5,1,2) (5,1,3)
(5,1,4) (5,2,1) (5,2,3) (5,2,4) (5,3,1) (5,3,2) (5,3,4) (5,4,1) (5,4,2) (5,4,3)
```

图 3-5　例 3-4 的运行结果

3.4　递归与分治思想

递归与分治的算法思想往往是相伴而生的，它们在各类算法中使用非常频繁，应用递归和分治的算法思想有时可以设计出代码简洁且比较高效的算法来。本章将详细介绍递归与分治的算法思想。

3.4.1　基本概念

在解决一些比较复杂的问题，特别是解决一些规模较大的问题时，常常将问题进行分解。具体来说，就是将一个规模较大的问题分割成规模较小的同类问题，然后将这些小的子问题逐个加以解决，最终也就将整个大的问题解决了。这种分而治之的思想称做分治的思想。在解决一些问题比较复杂、计算量庞大的问题时经常被用到。

一个最为经典的使用分治思想设计的算法就是第 2 章中介绍过的“折半查找算法”。折半查找算法利用了元素之间的顺序关系（有序序列），采用分而治之的策略，不断缩小问题的规模，每次都将问题的规模减小至上一次的 1/2。采用顺序查找的方法对关键字进行搜索的时间复杂度为 $O(n)$，而采用折半查找方法的时间复杂度仅为 $O(\log_2 n)$。

递归的思想也是一种常见的算法设计思想。所谓递归算法，就是一种直接或间接地调用原算法本身的一种算法。可以通过一个具体的例子来理解递归的算法思想。

【实例 3-5】计算 n 的阶乘 $n!$。

这是一个再简单不过的问题了。只要学过程序设计语言的人都可以很容易地编程解决此题。一般采用循环累乘的算法求解，算法如下：

```
int factorial(n)
{
    int i, res = 1;
    for(i=1;i<=n;i++)
        res = res * i;
    return res;
}
```

其实阶乘的数学定义也可以用递归函数来简单地描述：

$$n!=\begin{cases}1 & n=0\\ n(n-1)! & n>0\end{cases}$$

这样的函数称为递归函数，因为该函数内部直接或间接地调用了该函数本身。基于阶乘的递归函数的描述，就不难设计出计算 n 的阶乘 $n!$的递归算法。

```
int factorial(n)
{
    if(n == 0) return 1;
    else return n* factorial(n-1);
}
```

可以看出，使用递归算法解决阶乘问题形式上更加简洁，更易于人们理解。

在设计递归算法时要注意以下几点。

（1）每个递归函数都必须有一个非递归定义的初始值，作为递归结束标志，或递归结束的出口。就像实例 3-5 所描述的递归算法中的 if(n==0) return 1;。如果一个递归算法中没有这个非递归定义的初始值，那么该递归调用是无法计算出具体的值的（或无法得到结果），同时该递归调用也无法结束。

（2）在设计递归算法时，要解决的问题需具有递归性。例如要计算 n 的阶乘 $n!$，$n!$的定义本身具有递归性。这种所谓的递归性实际上就是一种反复调用自身过程的特性。

（3）虽然采用递归算法解决问题，特别是一些复杂问题，更加方便且容易实现，但是递归算法的运行较低，时间和空间复杂度都比较高，因此对于一些对时间复杂度和空间复杂度要求较高的程序，建议使用非递归算法设计。

在实际的算法设计中，递归与分治如同一对兄弟，经常结合在一起使用。这是因为，由分治的方法产生的子问题往往都是原问题的更小规模。反复使用分治的手段，可使子问题与原问题类型一致，但规模不断缩小，最终使子问题比较容易求解。既然子问题与原问题的类型一致，这就具有了所谓的递归性，因此可以使用递归的方法用解决原问题的算法去解决同类型的子问题。在第 2 章中介绍的折半查找算法只是单纯地使用了分治的策略，在下面的实例分析中，将使用递归与分治思想相结合的方法进行折半查找算法的设计。

3.4.2　计算整数的划分数

【实例 3-6】将一个正整数 n 表示成一系列的正整数之和：

$$n=n_1+n_2+\cdots+n_k \qquad (n_1\geqslant n_2\geqslant\cdots\geqslant n_k\geqslant 1,\ \ k\geqslant 1)$$

被称作正整数 n 的一个划分。一个正整数 n 可能存在着不同的划分，例如正整数 6 的全部的划分为：

6=6

6=5+1

6=4+2　6=4+1+1

6=3+3　6=3+2+1　6=3+1+1+1

6=2+2+2　6=2+2+1+1　6=2+1+1+1+1

6=1+1+1+1+1+1

正整数 n 的不同的划分的个数称为该正整数 n 的划分数。例如正整数 6 的划分数为 11。编写一个程序，计算输入的正整数 n 的划分数。

【分析】

根据正整数划分的定义，可以总结出以下规律：

设标记 P(n,m)表示正整数 n 的所有不同划分中，最大加数不大于 m 的划分个数。例如 P(6,2)=4，因为在正整数 6 的全部划分中，最大加数不大于 2 的只有：

6=2+2+2　6=2+2+1+1　6=2+1+1+1+1

6=1+1+1+1+1+1

这 4 个划分。

可以建立起如下递归关系：

（1）P(n,1)=1，$n\geqslant 1$；

这是因为任何正整数 n 的划分中，加数不大于 1 的划分有且仅有 1 种，即 $n=\overbrace{1+1+\cdots+1}^{n\text{ 个}}$。

（2）P(n,m)=P(n,n)，$m\geqslant n$；

这是因为任何正整数 n 的划分中，只存在一种划分即 $n=n$，其最大加数等于 n。不存在最大加数大于 n 的情况，（根据划分的定义，任何加数必须大于 1）。

（3）P(n,n)=P(n,n-1)+1；

因为最大加数为 n 的划分只有 1 种，即 $n=n$。因此最大加数不大于 n 的划分数就是：P(n,n-1)+1。

这里要注意，“n 的最大加数为 n 的划分数”不同于“n 的最大加数不大于 n 的划分数”，即 P(n,n)，如图 3-6 所示。

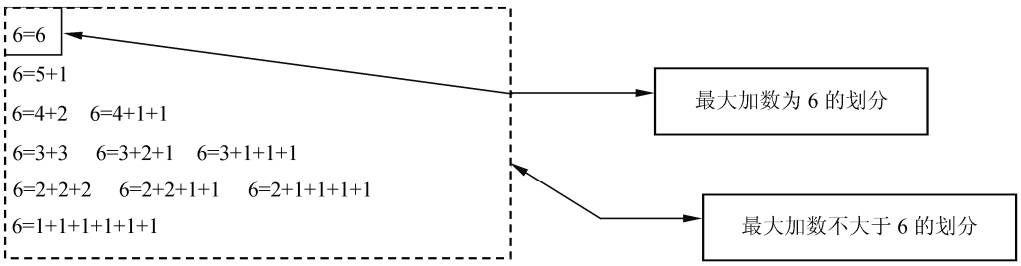


图 3-6　最大加数为 6 的划分以及最大加数不大于 6 的划分

因此最大加数不大于 n 的划分数 P(n,n)等于最大加数不大于 n–1 的划分数 P(n,n–1)，与最大加数为 n 的划分数 1 之和。

（4）P(n,m)=P(n,m–1)+P(n–m,m)，$n>m>1$；

“正整数 n 的最大加数不大于 m 的划分数（$n>m>1$）”等于“n 的最大加数不大于 m–1 的划分数 P(n,m–1)”与“n 的最大加数为 m 的划分数”之和，如图 3-7 所示。

如图 3-7 所示，“正整数 6 的最大加数不大于 4 的划分数 P(6,4)”等于“6 的最大加数不大于 3 的划分数 P(6,3)”与“6 的最大加数为 4 的划分数”之和。而“6 的最大加数为 4

的划分数”为 2，其实它等于“6–4=2 的最大加数不大于 4 的划分数”，即等于 P(6–4,4)=P(2,4)=2。

有些读者可能会提出问题来，可不可以说 *n* 的最大加数为 *m* 的划分数等于 P(*n*–*m*,*n*–*m*)呢？因为 P(6–4,4)=P(2,4)=P(2,2)。为什么这里要写成 P(*n*–*m*,*m*)，而不能写成 P(*n*–*m*,*n*–*m*)呢？答案是否定的。因为 P(6–4,4)=P(2,4)=P(2,2)只是一个特例。如果求 6 的最大加数为 2 的划分数，那么它等于 P(6–2,2)=P(4,2)=3 而不等于 P(6–2,6–2)=P(4,4)=5，这一点应当注意。因此可以得出结论：P(*n*,*m*)=P(*n*,*m*–1)+P(*n*–*m*,*m*)，*n*>*m*>1。

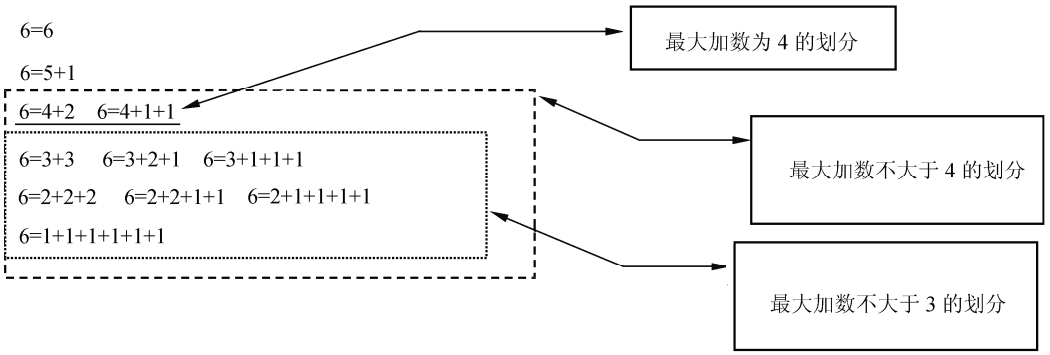


图 3-7　6 的最大加数不大于 4 的划分组成

根据以上归纳出的递归关系，可以得出计算 P(*n*,*m*)的递归函数式：

$$\mathrm{P}(n,m)=\begin{cases}1 & m=1\\ \mathrm{P}(n,n) & n<m\\ 1+\mathrm{P}(n,n-1) & n=m\\ \mathrm{P}(n,m-1)+\mathrm{P}(n-m,m) & n>m>1\end{cases}$$

很显然，这是递归函数。下面可以由此递归函数设计出求解正整数 *n* 的划分数的递归算法。其实正整数 *n* 的划分数即为 P(*n*,*n*)。

```
int P(int n,int m)
{
    if(m=1) return 1;
    if(m>n) return P(n,n);
    if(m==n) return 1+P(n,m-1);
    return P(n,m-1)+P(n-m,m);
}
```

下面给出完整的程序清单。

程序清单 3-3

```
/*------------------------------ 3-3.c ------------------------------*/
#include "stdio.h"
int P(int n,int m)
{
    if(m==1 || n == 1) return 1;
    if(m>n) return P(n,n);
    if(m==n) return 1+P(n,m-1);
    return P(n,m-1)+P(n-m,m);
}

main()
```

```
{
    int n,s;
    printf("Please input a integer for getting the number of division\n");
    scanf("%d",&n);                    /*输入正整数 n*/
    s = P(n,n);                        /*求出正整数 n 的划分数*/
    printf("The number of division of %d is %d\n",n,s);
    getche();
}
```

本程序的运行结果如图 3-8 所示。

```
Please input a integer for getting the number of division
6
The number of division of 6 is 11
```

图 3-8　例 3-6 的运行结果

3.4.3　递归的折半查找算法

【实例 3-7】有一个数组 A[10]，里面存放了 10 个整数，顺序递增。

A[10] = {2,3,5,7,8,10,12,15,19,21}

任意输入一个用数字 *n*，用折半查找法找到 *n* 位于数组中的位置。如果 *n* 不属于数组 A，显示错误提示。要求用递归的方法实现折半查找。

【分析】

在第 2 章中曾详细地介绍过折半查找的算法，它是一种针对有序序列（或文件记录）的高效的查找算法。折半查找的基本思想是：减小查找序列的长度。它的查找过程是：先确定待查找记录所在的范围，然后逐渐缩小查找的范围，直至找到该记录为止（也可能查找失败）。因此它也是一种基于分治的算法思想设计出来的查找算法。折半查找的算法如下：

```
int bin_search(keytype key[],int n,keytype k)
{
    int  low = 0, high = n-1, mid;
    while(low<=high){
    mid = (low+high)/2;
    if(key[mid] == k)
        return mid;          /*查找成功，返回 mid*/
    if(k > key[mid])
        low = mid + 1;       /*在后半序列中查找*/
    else
        high = mid - 1;      /*在前半序列中查找*/
}
return -1;                   /*查找失败，返回-1*/
}
```

从上述的算法中不难看出，折半查找的过程实际上也是一个递归的过程。因为在折半查找的过程中，每次都将问题的规模减小至原问题的一半，而缩小后的子问题与原问题类型保持一致，也就是说子序列的查找步骤与原序列的查找步骤大同小异，只是查找序列的范围（*low*，*high*）不一样。因此，解决缩小后的子问题的方法与解决原问题的方法是一

样的。这就说明折半查找的思想具有递归的性质。因而比较容易写出折半查找的递归算法。

```
int bin_search(keytype key[],int low, int high,keytype k)
{
    int mid;
    if(low>high)
        return -1;
    else{
        mid = (low+high) / 2;
        if(key[mid]==k)
            return mid;
        if(k>key[mid])
            return bin_search(key,mid+1,high,k);   /*在序列的后半部分查找*/
        else
            return bin_search(key,low,mid-1,k);    /*在序列的前半部分查找*/
        }
}
```

这里要注意一点，只有原序列中的元素是以递增不减的顺序排列时，才能使用上述的算法进行元素的查找。使用该算法进行折半查找，若查找成功，返回元素记录在原序列中的位置；若查找不成功，返回–1。

下面给出完整的程序清单。

程序清单 3-4

```
/*--------------------------------- 3-4.c  ----------------------------*/
#include "stdio.h"
int bin_search(int key[],int low, int high,int k)
{
   int mid;
   if(low>high)
      return -1;
   else{
      mid = (low+high) / 2;
      if(key[mid]==k)
         return mid;
      if(k>key[mid])
         return bin_search(key,mid+1,high,k);     /*在序列的后半部分查找*/
      else
         return bin_search(key,low,mid-1,k);      /*在序列的前半部分查找*/
   }
}

main()
{
   int n , i , addr;
   int A[10] = {2,3,5,7,8,10,12,15,19,21};         /*初始化数组序列 A*/
   printf("The contents of the Array A[10] are\n");
   for(i=0;i<10;i++)
   printf("%d ",A[i]);                             /*显示数组 A 中的内容*/
   printf("\nPlease input a interger for search\n");
   scanf("%d",&n);                                 /*输入待查找的元素*/
   addr = bin_search(A,0,9,n);
   if(-1 != addr)                                  /*查找成功*/
```

```
    printf("%d is at the %dth unit is array A\n ",n,addr);
    else printf("There is no %d in array A\n",n);  /*查找失败*/
    getche();
}
```

本程序的运行结果如图 3-9 所示。

注意：在这里数组的下标是从 0 开始的，因此 12 位于数组 A 的第 6 单元。

```
The contents of the Array A[10] are
2 3 5 7 8 10 12 15 19 21
Please input a interger for search
12
12 is at the 6th unit is array A
```

图 3-9　例 3-7 的运行结果

3.5　贪心算法思想

贪心算法又称为贪婪算法，是指在对问题求解时，总是做出在当前看来是最好的选择。它不从整体最优上加以考虑，所做出的仅是在某种意义上的局部最优解。而局部的最优解叠加到一起便是该问题整体的最优解，或者近似最优解。贪心算法的思想非常简单而且算法效率很高，在一些问题的解决上有着明显的优势。本章将详细介绍贪心算法的基本思想。

3.5.1　基本概念

先来看一个找零钱的问题。假设有 3 种硬币，面值分别为 1 元、5 角、1 角。这 3 种硬币各自的数量不限，现在要找给顾客 2 元 7 角钱，请问怎样找钱才能使得找给顾客的硬币数量最少呢？你也许会不假思索地说出答案：找给顾客 2 枚 1 元硬币，1 枚 5 角硬币，2 枚 1 角硬币。其实找给顾客 5 枚 5 角硬币，2 枚 1 角硬币也能凑够 2 元 7 角，但是这种找零钱的方法硬币数量较多，不符合题目的要求。在这里，我们下意识地应用了所谓贪心算法解决找零钱的问题。

所谓贪心算法，就是总是做出在当前看来是最好的选择的一种方法。以上述的找零钱为例，为了找给顾客的硬币数量最少，在选择硬币的面值时，当然是尽可能地选择面值大的硬币。因此要找给顾客 2 元 7 角钱，下意识地遵循了以下方案。

（1）首先找出一个面值不超过 2 元 7 角的最大硬币，即 1 元硬币。

（2）然后从 2 元 7 角中减去 1 元，得到 1 元 7 角，再找出一个面值不超过 1 元 7 角的最大硬币，即 1 元硬币。

（3）然后从 1 元 7 角减去 1 元，得到 7 角，再找出一个面值不超过 7 角的最大硬币，即 5 角硬币。

（4）然后从 7 角中减去 5 角，得到 2 角，再找出一个面值不超过 2 角的最大硬币，即 1 角硬币。

（5）然后从 2 角中减去 1 角，得到 1 角，再找出一个面值不超过 1 角的最大硬币，即 1 角硬币。

（6）找零钱过程结束。

这个找钱过程实际应用的就是一种典型的贪心算法思想。

因此不难看出，应用贪心算法求解问题，并不从问题的整体最优上加以考虑，它所做出的每一步选择只是在某种意义上的局部最优选择。因此严格意义上讲，要使用贪心算法求解问题，该问题应当具备以下性质。

（1）贪心选择性质

所谓贪心选择性质就是指所求解的问题的整体最优解可以通过一系列的局部最优解得到。所谓局部最优解，就是指在当前的状态下做出的最好选择。例如上述的找零钱问题，当前状态下的最好选择就是使找过硬币后剩余的（应当找给顾客的）零钱数最接近于 0。因此每次找钱都要找面值尽可能大的硬币。

（2）最优子结构性质

当一个问题的最优解包含着它的子问题的最优解时，就称此问题具有最优子结构性质。找零钱问题本身就具有最优子结构性质。

但是，要判断一个问题是否具备以上两种性质，也就是说判断一个问题是否可以通过贪心算法得到最优解，是一件比较困难的事情。这里需要比较复杂而严格的数学证明。因此虽然贪心算法简单易实现，并且效率很高，但是使用贪心算法之前必须对问题本身进行深入而透彻地分析和证明，以保证使用贪心算法得到最优解。

其实，虽然贪心算法不是对所有问题都能得到整体的最优解，但是实际应用中的许多问题都可以使用贪心算法得到最优解。与此同时，即使使用贪心算法不能产生出问题的最优解，但最终结果也是最优解的很好的近似解。因此在解决一般性问题时，我们大可不必过分要求一定要得到问题的最优解，也没有必要进行严格的推理证明，使用贪心算法是一种不错的选择。

我们经常使用的哈夫曼（Huffman Tree）编码算法，求解最小生成树的克鲁斯卡尔（Kruskal）算法和普里姆（Prim）算法，求解图的单源最短路径的迪克斯特拉（Dijkstra）算法等都是基于贪心算法的思想设计出来的。

3.5.2　最优装船问题

【实例 3-8】有一批集装箱要装入一个载重量为 C 的货船中，每个集装箱的质量由用户自己输入指定，在货船的装载体积不限的前提下，如何装载集装箱才能尽可能多地将集装箱装入货船中？

【分析】

这个问题可以用贪心算法求得最优解。只要每次装船时，采取“质量最轻的集装箱先装船”的策略，就可以得到最优装船问题的一个最优解。

在设计算法时，可用一个数组 w[]存放每个集装箱的质量，例如 w[2]=5，就表示第 2 个集装箱的质量为 5。再设置一个向量数组 x[]存放集装箱的取舍状态。其中，x[i]=1 表示将第 i 个集装箱放入船中，x[i]=0 表示第 i 个集装箱不装入货船。最终输出的结果应当是向量数组 x[]的下标，如果输出结果为{0，1，4，5}表示将第 0，1，4，5 个集装箱装入货船中。也就是输出数组 x[]中所有 x[i]=1 的下标 i。

应用贪心算法求解，为了装载尽可能多的集装箱，每次都要选出数组 w[]中当前质量最轻的（即 w[i]值最小的）集装箱，并将 x[i]置 1，同时重置 c=c-w[i]，即变量 c 中存放货船的剩余载重量，c 的初始值为货船总载重量 C。循环执行上述操作，直到 w[i]>c，表明

此时货船装货已达到最大载重量;或者集装箱全部装完为止。然后输出数组 x[]中所有 x[i]=1 的下标 i。可用下面这段代码描述这个算法过程。

```
Loading(int x[],int w[],int c,int n)
{
    int i , s = 0 ;
   /*动态开辟一个临时数组，存放 w[]的下标，如果 t[i],t[j],i<j，则 w[t[i]]≤w[t[j]] */
    int *t = (int *)malloc(sizeof(int) * n);
    sort(w,t,n);                                /*排序，用数组 t[]存放 w[]的下标*/
    for(i = 0;i<n;i++)
          x[i] = 0;                             /*初始化数组 x[]*/
    for(i=0;i<n && w[t[i]]<=c; i++){            /*使用贪心策略求解*/
        x[t[i]] = 1;                            /*将第 t[i]个集装箱装入货船中*/
        c = c - w[t[i]];                        /*变量 c 中存放货船的剩余载重量*/
        }
}
```

上述算法中,用一个临时数组 t 来存放数组 w[]的下标,使得如果 i<j,则 w[t[i]]≤w[t[j]]。例如 w[3]={5,3,2}，那么 t[3]={2,1,0}，它表明集装箱 w[t[0]]即 w[2]的质量小于集装箱 w[t[1]]，即 w[1]的质量小于集装箱 w[t[2]]即 w[0]的质量。生成数组 t[]的工作由函数 sort()来完成。函数 sort()的算法描述如下：

```
sort(int w[], int t[], int n)
{
    int i,j;
    /*动态开辟一个临时数组，存放 w[]中的内容，用于排序*/
    int w_tmp= (int *)malloc(sizeof(int) * n);
    for(i=0;i<n;i++)
        t[i] = i;           /*初始化数组 t*/
    for(i=0;i<n;i++)
        w_tmp[i] = w[i];    /*复制数组 w[]到 w_tmp[]*/

/*冒泡排序实现*/
/*将 w_tmp[]与 t[]共同排序，当 w_tmp[]实现从小到大排列时，t[]中存放的就是 w_tmp[]中的元素在 w[]中对应的下标*/
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++)
            if(w_tmp[j]>w_tmp[j+1])
            {
                tmp = w_tmp[j];
                w_tmp[j] = w_tmp[j+1];
                w_tmp[j+1] = tmp;
                tmp = t[j];
                t[j] = t[j+1];
                t[j+1] = tmp;
            }
}
```

下面给出完整的程序清单。

程序清单 3-5

```
/*------------------------------- 3-5.c  ----------------------------*/
#include "stdio.h"

void sort(int w[], int t[], int n)
```

```
{
    int i,j,tmp;
    /*动态开辟一个临时数组，存放 w[]中的内容，用于排序*/
    int *w_tmp= (int *)malloc(sizeof(int) * n);
    for(i=0;i<n;i++)
        t[i] = i;             /*初始化数组 t*/
    for(i=0;i<n;i++)
        w_tmp[i] = w[i];
    for(i=0;i<n-1;i++)
        for(j=0;j<n-i-1;j++) /*冒泡排序实现*/
            if(w_tmp[j]>w_tmp[j+1])
            {
                tmp = w_tmp[j];
                w_tmp[j] = w_tmp[j+1];
                w_tmp[j+1] = tmp;
                tmp = t[j];
                t[j] = t[j+1];
                t[j+1] = tmp;
            }
}

void Loading(int x[],int w[],int c,int n)
{
    int i , s = 0 ;
    /*动态开辟一个临时数组，存放 w[]的下标，如果 t[i],t[j],i<j，则 w[t[i]]≤w[t[j]]*/
    int *t = (int *)malloc(sizeof(int) * n);
    sort(w,t,n);                          /*排序，用数组 t[]存放 w[]的下标*/
    for(i = 0;i<n;i++)
    x[i] = 0;                             /*初始化数组 x[]*/
    for(i=0;i<n && w[t[i]]<=c; i++){
        x[t[i]] = 1;                      /*将第 t[i]个集装箱装入货船中*/
        c = c - w[t[i]];                  /*变量 c 中存放货船的剩余载重量*/
        }
}

main()
{
    int x[5],w[5],c,i;
    printf("Please input the maximum loading of the sheep\n");
    scanf("%d",&c);                     /*输入货船的最大载重量*/
    printf("Please input the weight of FIVE box\n");
    for(i=0;i<5;i++)                    /*输入每个集装箱的质量*/
        scanf("%d",&w[i]);
    Loading(x,w,c,5);                   /*进行最优装载*/
    printf("The following boxes will be loaded\n");
    for(i=0;i<5;i++)                    /*输出结果*/
    {
        if(x[i] == 1)     printf("BOX:%d ",i) ;
    }
    getche();
}
```

本程序的运行结果如图 3-10 所示。

注意：本次运行程序，设定船的最大载重量为 13，5 个集装箱分别重{5，7，6，3，2}，得到的最佳装载方案是：装载集装箱 0，质量为 5；装载集装箱 3，质量为 3；装载集装箱 4，质量为 2。总的装载质量为 10，小于最大载重量 13，装载的集装箱数为 3 个。

```
Please input the maximum loading of the ship
13
Please input the weight of FIVE box
5 7 6 3 2
The following boxes will be loaded
BOX:0 BOX:3 BOX:4
```

图 3-10　例 3-8 的运行结果

3.6　回　溯　法

回溯法是一种非常有效，适用范围相当广泛的算法思想。许多复杂的问题，规模较大的问题都可以使用回溯法求解。因此回溯法又有“通用解题方法”的美称。本章将详细介绍回溯法的算法思想。

3.6.1　基本概念

回溯法的基本思想是：在包含问题的所有解的解空间树中，按照深度优先搜索的策略，从根结点出发深度探索解空间树。当探索到某一结点时，要先判断该结点是否包含问题的解，如果包含，就从该结点出发继续探索下去；如果该结点不包含问题的解，那就说明以该结点为根结点的子树一定不包含问题的最终解，因此要跳过对以该结点为根的子树的系统探索，逐层向其祖先结点回溯。这个过程叫做解空间树的“剪枝”操作。

如果应用回溯法求解问题的所有解，要回溯到解空间树的树根，这样才能保证根结点的所有子树都被探索到才结束。如果只要求得到问题的一个解，那么在探索解空间树时，只要搜索到问题的一个解就可以结束了。

许多复杂的问题都可以用回溯法的思想来求解，例如经典的 N 皇后问题。

N 皇后问题的描述为：求解如何在一个 N×N 的棋盘上无冲突地摆放 N 个皇后棋子。在国际象棋里，皇后的移动方式为横竖交叉的，因此在任意一个皇后所在位置的水平、竖直，以及 45° 斜线上都不能出现其他皇后的棋子，也就是不能有冲突。如图 3-11 所示，就是八皇后问题一种解。

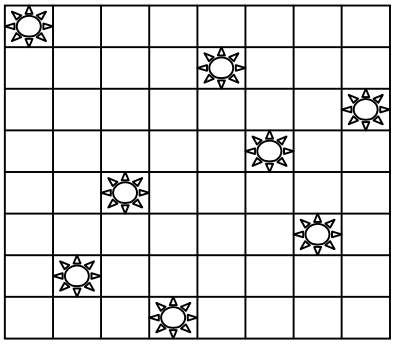


图 3-11　八皇后问题的一种解

N 皇后问题的解法很多，这里我们使用回溯法解决 N 皇后问题。以四皇后问题为例，可以构建出一棵解空间树，通过探索这棵解空间树，可以得到四皇后问题的一种或几种解。这样的解空间树共有 4 棵。如图 3-12 所示为四皇后问题的一种解空间树。

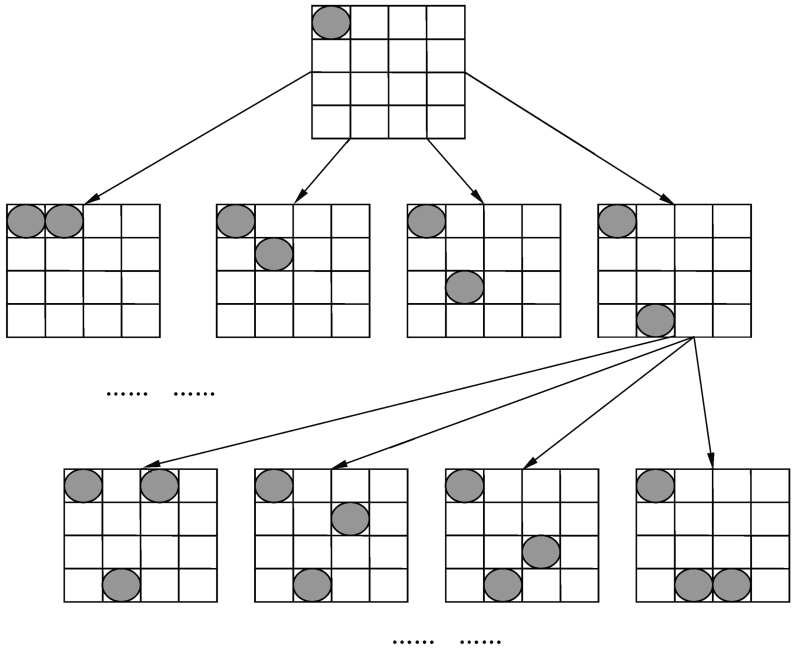


图 3-12　四皇后问题的一种解空间树

由于版面的限制，该解空间树并不完整。该解空间树的根结点为第一个皇后的一种摆法（这里规定第 i 个皇后摆在棋盘的第 i 列上），显然它还有另外 3 种摆法，因此一共可以构造出 4 棵解空间树。通过探索上述这棵解空间树，可以搜索出由根结点为这种棋面所衍生出来的所有四皇后问题的解（如果有解的话）。

如图 3-12 所示，根结点派生出 4 个子结点。每个结点都示意出前两个皇后可能摆放的位置。每个子结点又可派生出 4 个子结点，每个结点都示意出前 3 个皇后可能摆放的位置……整个解空间树为一棵四叉的满树，共包含 85 个结点。

应用回溯法求解四皇后问题，从根结点出发，深度优先搜索整个解空间树。当访问到根结点的第一个孩子时，发现该结点不包含问题的解。也就是说，该结点所示的皇后的摆放方法不符合四皇后问题的要求（因为两个皇后产生了冲突），那么由该结点作为根结点派生出来的子树中也肯定不会包含四皇后问题的解，于是停止向下探索，回溯到根结点，继续探索根结点的下一个孩子结点。这就是所谓的剪枝操作，这样可以减少搜索的步数，以尽快找到问题的答案。实践证明，如图 3-12 所示的解空间树中不包含四皇后问题的解。于是需要探索第二棵解空间树，如图 3-13 所示。

按照上述的探索过程深度优先搜索如图 3-13 所示的解空间树，最终可以搜索出四皇后问题的一个解，它的搜索路径在图中用粗体表示，其叶结点即为四皇后问题的一种解。图中用虚线描绘的结点之间的连线表示在此执行剪枝操作。如果按照上述步骤继续探索另外 2 棵解空间树，最终可找出四皇后问题的全部解。

下面通过实例加深对回溯法的理解。

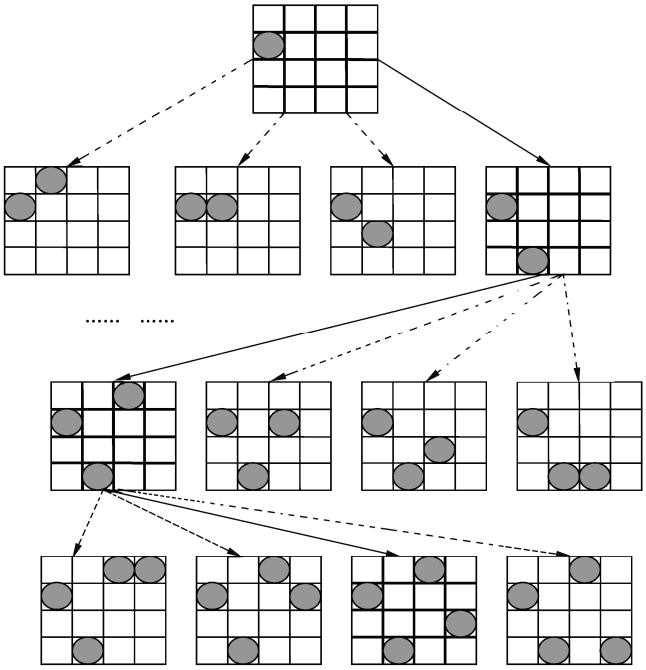


图 3-13　四皇后问题的另一种解空间树

3.6.2　四皇后问题求解

【实例 3-9】应用回溯法的思想求解四皇后问题。

【分析】

3.6.1 节中已经详细介绍了回溯法解决四皇后问题的基本过程。在这里将给出具体的算法描述和程序清单。

其实在解决四皇后问题时，并不一定要真的构建出这样一棵解空间树，它完全可以通过一个递归回溯来模拟。所谓解空间树只是一个逻辑上的抽象。当然也可以用树结构真实地创建出一棵解空间树，不过那样会比较浪费空间资源。

解决四皇后问题的算法描述如下：

```
void Queen(int j,int (*Q)[4]){
   int i , k;
   if(j==4)
   {
        for(i=0;i<4;i++)
        {                                  /*得到了一个解，在屏幕上输出结果*/
            for(k=0;k<4;k++)
                printf("%d ",Q[i][k]);
            printf("\n");
        }
          printf("\n");
        getche();
         return;
   }
```

```
   for(i=0;i<4;i++)                    /*i 为行号*/
   {
       if(isCorrect(i,j,Q))            /*如果 Q[i][j]可以放置皇后*/
       {
           Q[i][j] = 1;                /*放置皇后*/
           Queen(j+1,Q) ;              /*递归深度优先搜索解空间树*/
           /*Q[i][j]置 0，以便循环 试探 Q[i+1][j]是否可以摆放皇后棋子*/
           Q[i][j] = 0;
       }
   }
}
```

在该算法中，用一个二维数组 Q[4][4]存放棋盘布局。Q[i][j]=0 表示该位置不放置皇后，Q[i][j]=1 则表示该位置放置皇后。在这里采用了递归回溯的方法深度优先搜索解空间树，可以将四皇后问题的全部解找到并输出。

该算法中，核心的代码是：

```
   for(i=0;i<4;i++)                    /*i 为行号*/
   {
       if(isCorrect(i,j,Q))            /*如果 Q[i][j]可以放置皇后*/
       {
           Q[i][j] = 1;                /*放置皇后*/
           Queen(j+1,Q) ;              /*递归深度优先搜索解空间树*/
           /*Q[i][j]置 0，以便循环 试探 Q[i+1][j]是否可以摆放皇后棋子*/
           Q[i][j] = 0;
       }
   }
```

函数 Queen()包括两个参数，参数 j 为二维数组 Q 的列号（0~3），最开始调用 Queen()时，j 的初始值为 0，表明第一个皇后摆放在棋盘的第一列上。变量 i 为二维数组 Q 的行号（0~3），通过对变量 i 的 4 次循环，可以分别探索四皇后问题的 4 棵解空间树（以 Q[0][0]，Q[1][0]，Q[2][0]，Q[3][0]为解空间树的根结点）。

当 j 的值等于 4 时，表明数组 Q 中第 0~3 列都已放置好皇后棋子，说明此时得到了一个四皇后的解，因此程序返回上一层。参数(*Q)[4]为指向二维数组每一行的指针。

在该算法中调用了子函数 isCorrect(i,j,Q)，它的功能是判断棋盘中 Q[i][j]的位置是否可以放置一个皇后。它的算法描述如下：

```
int isCorrect(int i,int j,int (*Q)[4])
{
   int s,t;
   for(s=i,t=0;t<4;t++)
   if(Q[s][t]==1 && t!=j)return 0;      /*判断行*/

   for(t=j,s=0;s<4;s++)
   if(Q[s][t]==1 && s!=i)return 0;      /*判断列*/

   for(s=i-1,t=j-1;s>=0&&t>=0;s--,t--)
   if(Q[s][t] == 1)return 0;            /*判断左上方*/

   for(s=i+1,t=j+1;s<4&&t<4;s++,t++)
   if(Q[s][t] == 1) return 0;           /*判断右下方*/
```

```
   for(s=i-1,t=j+1;s>=0&&t<4;s--,t++)
   if(Q[s][t] == 1) return 0;           /*判断右上方*/

   for(s=i+1,t=j-1;s<4&&t>=0;s++,t--)
   if(Q[s][t] == 1) return 0;           /*判断左下方*/

   return 1;                            /*否则返回 1*/
}
```

该算法的设计思想是，以 Q[i][j]为中心，分别判断二维数组 Q 的行、列、左上方、右下方、右上方、左下方的状态。如果存在 1（有皇后棋子），则表明 Q[i][j]的位置不能放置皇后，返回 0；否则可以放置皇后，返回 1，如图 3-14 所示。

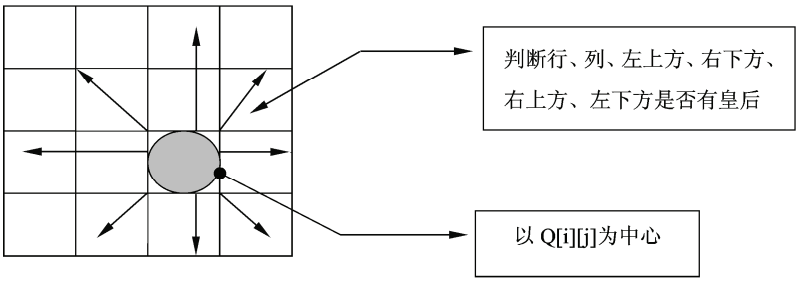


图 3-14　判断 Q[i][j]是否可以放置皇后

下面给出四皇后问题完整的程序清单。

程序清单 3-6

```
/*------------------------------ 3-6.c  --------------------------*/
#include "stdio.h"
int count=0;                            /*记录四皇后问题解的个数*/
int isCorrect(int i,int j,int (*Q)[4])
{
   int s,t;
   for(s=i,t=0;t<4;t++)
   if(Q[s][t]==1 && t!=j)return 0;      /*判断行*/

   for(t=j,s=0;s<4;s++)
   if(Q[s][t]==1 && s!=i)return 0;      /*判断列*/

   for(s=i-1,t=j-1;s>=0&&t>=0;s--,t--)
   if(Q[s][t] == 1)return 0;            /*判断左上方*/

   for(s=i+1,t=j+1;s<4&&t<4;s++,t++)
   if(Q[s][t] == 1) return 0;           /*判断右下方*/

   for(s=i-1,t=j+1;s>=0&&t<4;s--,t++)
   if(Q[s][t] == 1) return 0;           /*判断右上方*/

   for(s=i+1,t=j-1;s<4&&t>=0;s++,t--)
   if(Q[s][t] == 1) return 0;           /*判断左下方*/

   return 1;                            /*否则返回 1*/
}
```

```
void Queen(int j,int (*Q)[4]){
   int i , k;
   if(j==4)
   {                                          /*得到了一个解*/
        for(i=0;i<4;i++)
        {
            for(k=0;k<4;k++)
                printf("%d ",Q[i][k]);
            printf("\n");
        }
          printf("\n");
        getche();
         count++;
         return;

   }
   for(i=0;i<4;i++)
   {
       if(isCorrect(i,j,Q))                   /*如果 Q[i][j]可以放置皇后*/
        {

            Q[i][j] = 1;                      /*放置皇后*/
            Queen(j+1,Q) ;                    /*深度优先搜索解空间树*/
              Q[i][j] = 0;
        }
   }
}

main()
{
    int Q[4][4];
     int i,j;
     for( i=0;i<4;i++)
       for( j=0;j<4;j++)
            Q[i][j] = 0;                      /*初始化数组 Q*/
    Queen(0,Q);                               /*执行四皇后求解*/
     printf("The number of the answers of FOUR_QUEEN are %d",count);
    getche();
}
```

本程序的运行结果如图 3-15 所示。

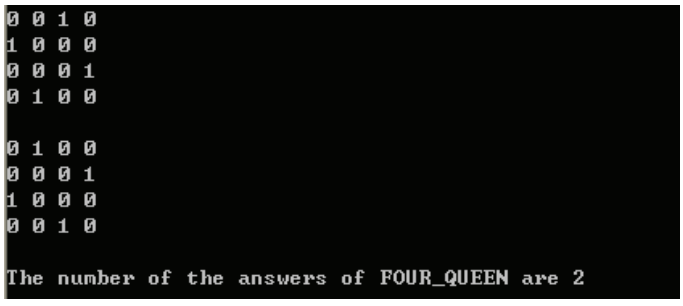


图 3-15　例 3-9 的运行结果

由四皇后问题很容易扩展成为八皇后问题或 N 皇后问题，在此不再详述。

3.7　数值概率算法

在解决实际问题时，有时会用到所谓的概率算法。概率算法允许在执行过程中随机地选择下一步的计算步骤，因此使用概率算法有时会大大降低复杂度，提高算法的效率，但有时也可能得不到问题的全部答案。本节将简要介绍一下常用的数值概率算法。

3.7.1　基本概念

概率算法大致分为 4 类：数值概率算法、蒙特卡洛（Monte Carlo）算法、拉斯维加斯（Las Vegas）算法和舍伍德（Sherwood）算法。这里只介绍最为基础的数值概率算法。

数值概率算法常应用于解决数值计算的问题。应用数值概率算法往往只能得到问题的近似解，并且该近似解的精度一般随着计算时间的增加而不断提高。因为在一些数值问题中，不可能也没有必要计算出问题的精确解（例如计算无理数 π 的取值等）。因此，在解决一些数值计算的问题时，数值概率算法常能派上用场。下面通过实例来认识数值概率算法。

3.7.2　计算定积分

【实例 3-10】设 $f(x)=1-x^2$，计算定积分：$I=\int_0^1 f(x)\mathrm{d}x$ 的值。

分析

函数 $f(x)=1-x^2$ 的在[0，1]上的图像如图 3-16 所示。

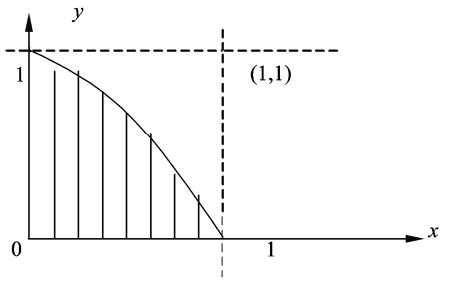


图 3-16　函数 $f(x)=1-x^2$ 的在[0，1]上的图像

要计算的定积分值的几何含义就是图中阴影部分的面积。可以试想，如果随机地向图中虚线与 x，y 坐标轴所围成的正方形中投点，那么根据几何概率的知识可知，随机点落入阴影区域的概率即为阴影部分的面积与虚线与 x，y 坐标轴所围成的正方形的面积之比。而由定积分的意义又可知：

$$P_{\text{投点}}=\frac{S_{\text{阴影}}}{S_{\text{正方形}}}=\frac{\int_0^1(1-x^2)\mathrm{d}x}{\int_0^1 1\mathrm{d}x}=\int_0^1(1-x^2)\mathrm{d}x$$

因此，只要求出随机点落入阴影区域的概率就求出了定积分 $I=\int_0^1 f(x)\mathrm{d}x$ 的近似值。

假设向单位正方形中随机投入 n 个点(x_i,y_i)，i=1,2,3…n。随机点(x_i,y_i)是否落入阴影区域内，可由 $y_i \leqslant f(x_i)=1-x_i^2$ 来判断。如果有 m 个点落入阴影区域内，则概率 P≈m/n。

该算法描述如下：

```
double Darts(int n)
{
    double x,y;
    time_t t;
```

```
    int i,count = 0;
    srand((unsigned)time(&t));
    for(i=0;i<n;i++)          /*n 为试验投点的个数，决定了计算概率的精度*/
    {
        x = rand()%100/100.0;
        y = rand()%100/100.0;
        if(y<=1 - pow(x,2))
            count++;          /*如果随机点落入阴影区域，则用 count++记录个数*/
    }
    return (double)count/(double)n;
}
```

本算法中应用函数 srand()和 rand()产生随机数，用系统时间作为产生随机数的种子，这样每次产生的随机数都不一样。因为函数 rand()产生的随机数返回值为整数，因此在这里先产生 100 以内的随机数（rand()%100），再将它除以 100.0，可得到双精度的[0，1]内的随机数点。试验投点的个数由函数 Darts 的参数 n 决定，n 值越大，投点的个数越多，相应的计算概率的精度也就越高。最终，用落入阴影区域的点数 count 除以总的投点数 n，得到的就是落入阴影区域的随机点的概率 P，也就是定积分 $I=\int_0^1 f(x)\mathrm{d}x$ 的近似值。

下面给出完整的程序清单。

程序清单 3-7

```
/*--------------------------- 3-7.c ---------------------------*/
#include "stdio.h"
#include "math.h"
#include "stdlib.h"
#include "time.h"

double Darts(int n)
{
    double x,y;
    time_t t;
    int i,count = 0;
    srand((unsigned)time(&t));
    for(i=0;i<n;i++)
    {
        x = rand()%100/100.0;
        y = rand()%100/100.0;
        if(y<=1 - pow(x,2))
            count++;
    }
    return (double)count/(double)n; /*返回落入阴影区域的点数与总点数 n 的比值*/
}

main()
{
    int n;
    printf("Please input the accuracy\n") ;     /*输入精度，即投点数*/
    scanf("%d",&n);
    printf("The result is about\n");            /*输出计算结果*/
    printf("%f\n",Darts(n));
    getche();
}
```

本程序的运行结果如图 3-17 所示。

```
Please input the accuracy
10000
The result is about
0.672900
```

图 3-17　例 3-10 的运行结果

注意：$I=\int_0^1(1-x^2)\mathrm{d}x$ 的计算结果应为 2/3 ≈ 0.667。

3.8　章 后 习 题

习题 3-1：上楼梯问题

已知楼梯有 20 阶台阶，上楼可以一步上 1 阶，也可以一步上 2 阶。请编写一个程序计算共有多少种不同的上楼梯方法。

提示：

我们可以采用递归回溯的方法求解此题。首先需要明确这个题目解的范围。我们可以用一棵二叉解空间树进行描述，如图 3-18 所示。

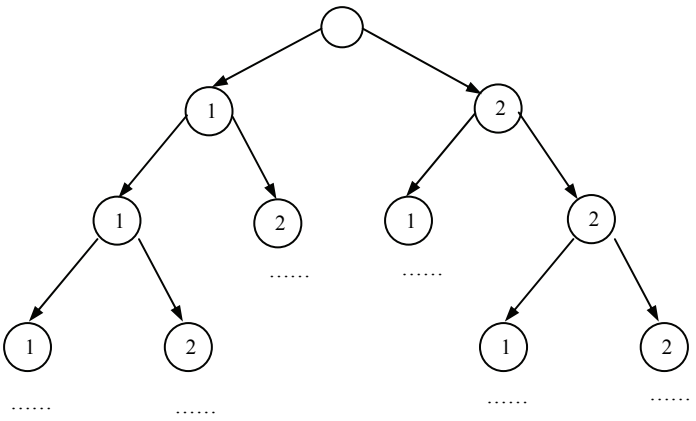


图 3-18　上楼梯问题的解空间树

图 3-18 为该问题的解空间树，所有的上楼梯的方案都包含在这棵二叉解空间树中。从根结点出发，每次都有 2 种选择：（1）上一阶台阶（对应解空间树中的“1”结点）；（2）上两阶台阶（对应解空间树中的“2”结点）。

从解空间树的根结点出发向下探索，每经过一个结点就表示这一步所登上的台阶数（1 或 2）。每次都将访问结点中的数字相加，就可以记录当前已登上的台阶数，当这个数字等于 20 时，就表示寻找出了一种上楼梯的方案，那么该结点以下的结点也就不必再访问，而是向其父结点回溯并继续探索下一分支。

在实做中，我们也可以不去真的构建这样一棵解空间树，而是通过递归回溯的方法模拟进行解空间树的搜索。具体的代码实现请参看下面的答案。

答案：

```
/*--------------------------- XT3-1.c  ---------------------------*/
#include "stdio.h"
```

```
#define  MAX_STEPS  20       /*定义 20 个台阶的楼梯*/

int Steps[MAX_STEPS] = {0};/*Steps[i]等于 1 或者 2，记录第 i 步登上的台阶数*/
int cnt = 0;                /*记录上楼梯方案的数目*/

void Upstairs(int footStep, int count, int steps) {
    /*参数 footStep 为当前要登的台阶数，count 是已走过的阶数，steps 为走过的步数*/
    int i;
    if (count + footStep == MAX_STEPS) {
       /*已走过的台阶数+当前要登的台阶数=20，得到一种上楼梯的方案*/
        Steps[steps] = footStep;    /*记录下这一步登上的台阶数*/
        for (i=0; i<=steps; i++) {  /*输出这种上楼梯的方案*/
            printf("%d ", Steps[i]);
        }
        printf("\n");
        cnt++;;                     /*累计上楼梯的方案数目*/
        return;
    }
    if (count + footStep > MAX_STEPS) {
    /*超过了楼梯的阶数，后续的解空间树不再探索*/
        return;
    }

    Steps[steps] = footStep;        /*记录当前上楼梯的阶数*/
    count = count + footStep;       /*记录目前已走过的台阶数*/
    steps++;                        /*步数加 1*/

    for (i=1; i<=2; i++) {          /*递归探索后续的分支*/
        Upstairs(i, count, steps);
    }
}

void Upstairs_All() {
    Upstairs(1,0,0);    /*从第一步上 1 个台阶开始探索解空间树*/
    Upstairs(2,0,0);    /*从第一步上 2 个台阶开始探索解空间树*/
}

main() {
    Upstairs_All();
    printf("共有%d 种走法\n",cnt);    /*输出上楼梯方案的总数*/
    getche();
}
```

程序的执行结果如图 3-19 所示。

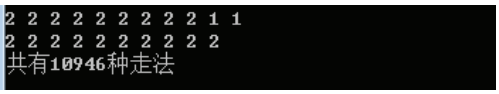


图 3-19　程序 XT3-1.c 的运行结果

注意：本程序将输出每一种上楼梯的方案，例如截图中的“2 2 2 2 2 2 2 2 2 2”就表示每步走 2 阶台阶，共走 10 步。总体计算下来共 10946 中走法，因此截图中不可能完全展现每一种上楼梯的方案。