第 24 章：常用算法设计　　作者：希赛教育软考学院　来源：希赛网　2014年03月13日

常用算法设计算法设计概述

第24章 常用算法设计

根据考试大纲，算法设计是每年必考的知识点，一般以C/C++语言的形式进行描述。本章主要讨论经常考的几种算法。

24.1 算法设计概述

算法是在有限步骤内求解某一问题所使用的一组定义明确的规则。通俗 就是计算机解题的过程。在这个过程中，无论是形成解题思路还是编写程序，都是在实施 。前者是推理实现的算法，后者是操作实现的算法。一个算法应该具有以下5个重要的特征。

（1）有穷性：一个算法必须总是（对任何合法的输入值）在执行有穷步之后结束，且每一步都可在有穷时间内完成。

（2）确定性：算法中每一条指令必须有确切的含义，读者理解时不会产生二义性。在任何条件下，算法只有唯一的一条执行路径，即对于相同的输入只能得出相同的输出。

（3）输入：一个算法有0个或多个的输入，以刻画运算对象的初始情况。所谓0个输入是指算法本身定出了初始条件。这些输入取自于某个特定的对象的集合。

（4）输出：一个算法有一个或多个输出，以反映对输入数据加工后的结果。没有输出的算法是毫无意义的。

（5）可行性：一个算法是可行的，即算法中描述的操作都是可以通过已经实现的基本运算执行有限次来实现的。

算法设计要求正确性、可读性、健壮性、效率与低存储量。

效率指的是算法执行时间。对于解决同一问题的多个算法，执行时间短的算法效率高。存储量需求指算法执行过程中所需要的最大存储空间。两者都与问题的规模有关。

算法的复杂性是算法效率的度量，是算法运行所需要的计算机资源的量，是评价算法优劣的重要依据。可以从一个算法的时间复杂度与空间复杂度来评价算法的优劣。当我们将一个算法转换成程序并在计算机上执行时，其运行所需要的时间取决于下列因素：

（1）硬件的速度。例如使用486机还是使用586机。

（2）书写程序的语言。实现语言的级别越高，其执行效率就越低。

（3）编译程序所生成目标代码的质量。对于代码优化较好的编译程序其所生成的程序质量较高。

（4）问题的规模。例如，求100以内的素数与求1 000以内的素数其执行时间必然是不同的。

显然，在各种因素都不能确定的情况下，很难比较出算法的执行时间。也就是说，使用执行算法的绝对时间来衡量算法的效率是不合适的。为此，可以将上述各种与计算机相关的软、硬件因素都确定下来，这样一个特定算法的运行工作量的大小就只依赖于问题的规模（通常用正整数n表示），或者说它是问题规模的函数。

1.时间复杂度

一个程序的时间复杂度是指程序运行从开始到结束所需要的时间。

一个算法是由控制结构和原操作构成的，其执行时间取决于两者的综合效果。为了便于比较同一问题的不同的算法，通常的做法是：从算法中选取一种对于所研究的问题来说是基本运算的原操作，以该操作重复执行的次数作为算法的时间度量。一般情况下，算法中原操作重复执行的次数是规模n的某个函数T(n)。

许多时候要精确地计算T(n)是困难的，我们引入渐近时间复杂度在数量上估计一个算法的执行时间，也能够达到分析算法的目的。

定义（大O记号）：如果存在两个正常数c和n0,对于所有的n,当n≥n0时有：

$$f(n)\leq cg(n)$$

则有：

$$f(n)=O(g(n))$$

也就是说，随着n的增大，f(n)渐进地不大于g(n)。例如，一个程序的实际执行时间为$T(n)=2n^3+3n^2+5$,则$T(n)=O(n^3)$。T(n)和n^3的值随n的增大渐近地靠拢。

使用大O记号表示的算法的时间复杂度，称为算法的渐近时间复杂度。

通常用O(1)表示常数计算时间。常见的渐近时间复杂度有：

$$O(1)<O(\log_2 n)<O(n)<O(n\log_2 n)<O(n^2)<O(n^3)<O(2^n)$$

2.空间复杂度

一个程序的空间复杂度是指程序运行从开始到结束所需的存储量。

程序运行所需的存储空间包括以下两部分。

固定部分：这部分空间与所处理数据的大小和个数无关。主要包括程序代码、常量、简单变量、定长成分的结构变量所占的空间。

可变部分：这部分空间大小与算法在某次执行中处理的特定数据的大小和规模有关。例如100个数据元素的排序算法与1 000个数据元素的排序算法所需的存储空间显然是不同的。

算法由数据结构来体现，所以看一个程序首先要搞懂程序实现所使用的数据结构，如解决装箱问题就使用链表这种数据结构。数据结构是算法的基础，数据结构支持算法，如果数据结构是递归的，算法也可以用递归来实现，如二叉树的遍历。经常采用的算法有迭代法、递推法、递归法、穷举法、贪婪法、分治法和回溯法等，下面我们对这些常用算法设计技术进行探讨，并尽量对解决各个问题所使用的算法所采用的数据结构进行比较详细的分析。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

迭代法

24.2 迭代法

迭代法适用于方程（或方程组）的求解，是使用间接方法求方程近似根的一种常用算法。设要求解的方程为f(x)=0，迭代法将方程表示为其等价的形式：x=g(x)，如果f(x)很复杂，还可以将f(x)拆成两个函数$f_1(x)$、$f_2(x)$，即$f(x)=f_1(x)-f_2(x)=0$，故有$f_1(x)=f_2(x)$。其中$f_1(x)$是这样一个函数，对于任

意数c，容易求出$f_1(x)=c$的精确度很高的实根。也就是说，$f_1(x)$很简单，前面提到的$x=g(x)$只不过是$f_1(x)=x$时的特例。其一般过程如下：

①选一个x的近似根x_0，从x_0出发，代入右边函数，并解方程$f_1(x)=f_2(x_0)$得到下一个近似根x_1。

②将上次近似根x_1代入右边函数，并解方程$f_1(x)=f_2(x_1)$，得到又一个近似根x_2。

③重复步骤②的计算，得到一系列的近似根x_0、x_1、x_2、…、x_n。

如果方程有根，则上述数列收敛于方程的根。若满足$|x_n-x_{n-1}|<\varepsilon$，则认为$x_n$是方程的近似根。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

第 24 章：常用算法设计　作者：希赛教育软考学院　来源：希赛网　2014年01月27日

迭代求解方程

24.2.1 迭代求解方程

问题描述：给出一个方程$f(x)=\sqrt{1+2x^2}-1nx-1n\ (1+\sqrt{2+x^2})+3$，求解f(x)=0的根。

按上述求解方程的步骤，先将f(x)分解为$f_1(x)$和$f_2(x)$，同时考虑到$f_1(x)$要比较简单，故我们对f(x)进行以下变换：

令 $f_1(x)=lnx$

$f_2(x)=\sqrt{1+2x^2}-1n\ (1+\sqrt{2+x^2})+3$

求解f(x)=0的根，则变换为求解$lnx=\sqrt{1+2x^2}-1n\ (1+\sqrt{2+x^2})+3$的根了。迭代法求解，其算法程序流程图如图24-1所示。

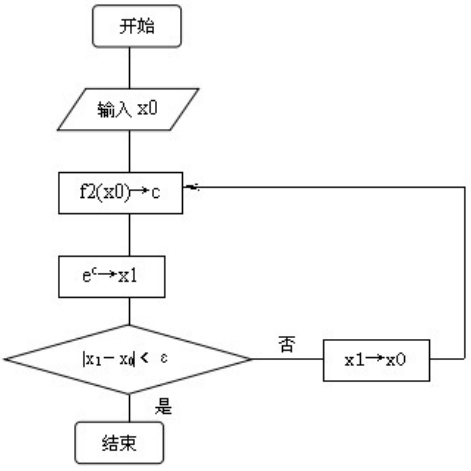


图24-1 迭代法方程求解程序流程图

该问题算法程序实现见程序24-1。

【程序24-1】

```
#include<stdio.h>
#include<math.h>
#define epsilon 1e-10
void main ( )
{ float x0,x1,c;
```

```
printf（"please input x0 :\n"）；
scanf（"%f",&x0）；        /*输入初始近似解*/
x1=x0;
do{                        /*循环迭代*/
x0=x1;
c=sqrt（1+2*x0*x0）-log（1+sqrt（2+x0*x0））+3;
x1=exp（c）；
}while（fabs（x1-x0）>epsilon）；
printf（"方程的近似根是 %f\n",x1）；
}
```

用迭代法求方程组的根和求方程的根类似，可以用数组存放各个方程的近似解。使用迭代法时应注意以下两点：

如果方程无解，数列必不收敛，因而迭代的重复为"死循环",所以使用迭代算法应考查方程是否有解。另外，应对重复次数给予一定的控制，以防死循环。

当方程有解时，若迭代公式选择不当或初始近似根选择不当，也会导致迭代失败，例如，迭代公式中出现除数为0、ln0或tan（π/2）等。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

迭代求解方程组的解

24.2.2 迭代求解方程组的解

问题描述：应用迭代法求解下面给出的一个方程组：

$x_1-x_2-x_3=2$

$2x_1-x_2-3x_3=1$

$3x_1+2x_2-5x_3=0$

迭代法也常用于求方程组的根，解此类问题的通用方法如下：

令$X=（x_0, x_1, \ldots, x_{n-1}）$

同时可设方程组为：$x_i=g_i（X）$（i=0,1,…，n-1）

对于上述方程组：

令

$X=(x[1],x[2],x[3])$

则

$x[1]=x[2]+x[3]+2$

$x[2]=2*x[1]-3*x[3]-1$

$x[3]=(3*x[1]+2*x[2])/5$

迭代求解过程如图24-2所示。

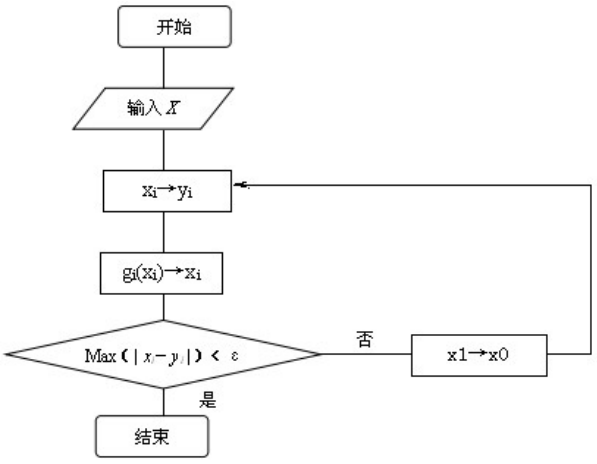


图24-2 迭代法方程组求解程序流程图

该问题算法程序实现见程序24-2.

【程序24-2】

```
#include<stdio.h>
#include<math.h>
#define  MAX  10
#define epsilon 1e-10
void main ( )
{  float x[MAX],y[MAX],delta;
int i,n=3;
for ( i=1;i<=n;i++ ) {                    /*输入初始近似解*/
printf ( "please input x%d :\n",i ) ;
scanf ( "%f",&x[i] ) ;
}
do{                                 /*循环迭代*/
for  ( i=1;i<=n;i++ )
y[i]=x[i];
x[1]=x[2]+x[3]+2;
x[2]=2*x[1]-3*x[3]-1;
x[3]= ( 3*x[1]+2*x[2] ) /5;
for ( delta=0.0,i=1;i<=n;i++ )
if ( fabs ( y[i]-x[i] ) >delta ) delta=fabs ( y[i]-x[i] ) ;
}while ( delta>epsilon ) ;
for ( i=1;i<=n;i++ )
printf ( "x[%d]=%f",i,x[i] ) ;
printf ( "\n" ) ;
}
```

版权方授权希赛网发布，侵权必究

穷举法

24.3 穷举法

穷举搜索法是穷举所有可能的情形，并从中找出符合要求的解，即对可能是解的众多候选解按某种顺序逐一枚举和检验，并从中找出那些符合要求的解作为问题的解。对没有有效解法的离散型问题，规模不大时，穷举法不失是一种可考虑的方法。但规模较大时，穷举显然就不行了。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

组合问题

24.3.1 组合问题

问题描述：找出n个自然数（1,2,3,…，n）中r个数的组合，这里假设r=3.例如，n=5,则所有可能组合为：

5　4　3

5　4　2

5　4　1

5　3　2

5　3　1

5　2　1

4　3　2

4　3　1

4　2　1

3　2　1

对这个问题，我们可以穷举所有可能情形，同时判断每种情形是否是合理的组合。对于n个数中r个数的组合，每r个数中，数不能相同，另外，任何两组组合的数，其包含的数也不应相同。例如，5,4,3和3,4,5应该视为相同组合的数。因此，我们约定前一个数应大于后一个数。这很容易用r重循环来穷举所有可能的组合，同时输出合理的组合。当r=3时，则用三重循环进行搜索。

该问题算法程序实现见程序24-3.

【程序24-3】

```
#include<stdio.h>
void main（ ）{
int i,j,k,n;
printf（ "please input n:" ）；
scanf（ "%d",&n ）；
```

```
for ( i=n;i>=1;i-- )
for ( j=n;j>=1;j-- )
for ( k=n;k>=1;k-- )
if ( ( i!=j ) && ( i!=k ) && ( j!=k ) && ( i>j ) && ( j>k )
printf ( "%3d,%3d,%3d\n",i,j,k ) ) ;
}
```

该问题还可由程序24-4实现。

【程序24-4】

```
#include<stdio.h>
void main ( ) {
int i,j,k,n;
printf ( "please input n:" ) ;
scanf ( "%d",&n ) ;
for ( i=n;i>=3;i-- )                /*在这里r=3,循环重数随r的变化而变化*/
for ( j=i-1;j>=2;j-- )
for ( k=j-1;k>=1;k-- )
if ( ( i!=j ) && ( i!=k ) && ( j!=k ) && ( i>j ) && ( j>k ) )
printf ( "%3d%3d%3d\n",i,j,k ) ;
}
```

比较这两个程序，第一个程序穷举了组合的所有可能情形，并从中选出符合条件的解；而后者比较简洁，显然其比前者效率高。但是这两个程序都有一个明显的问题，当r变化时，循环重数也要改变，而程序不能相应地自动修改循环的重数，这就使这一问题的解没有一般性。但是很多情况下穷举搜索法还是常用的。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

背包问题

24.3.2 背包问题

问题描述：有不同价值、不同重量的物品n件，求从这n件物品中选取一部分物品的选择方案，使选中物品的总重量不超过指定的限制重量，但选中物品的价值之和最大。

解决背包问题的较高效率的方法一般用递归和贪婪法，而背包问题规模不是很大时，也可采用穷举法。此时令n个物品的重量和价值分别存储于数组w[ ]和v[ ]中，限制重量为tw.程序用一个n元组（x_0,x_1,…，x_{n-1}）表示物品选择的解，其中x_i=0 表示第i个物品没有选取，而x_i=1则表示第i个物品被选取。显然这是一个0-1背包问题（0-1背包和部分背包问题的区别见贪婪法部分）。用搜索法解决背包问题，需要搜索所有的选取方案，而根据上述方法，我们只要搜索所有的n元组，就可以得到

问题的解。

仔细分析不难发现，每个分量取值为0或1的n元组的个数共为2^n个，而每个n元组其实对应了一个长度为n的二进制数，且这些二进制数的取值范围为0～2^n-1。因此，如果把0～2^n-1分别转化为相应的二进制数，则可以得到我们所需要的2^n个n元组。

该问题算法程序实现见程序24-5.

【程序24-5】

```
#include<stdio.h>
#include<math.h>
#define  MAX  100                    /*最多物品数*/
conversion(int n,int b[MAX] ) {      /*将n化为二进制形式，结果存放到数组b中*/
int i;
for(i=0;i<MAX;i++ )
{
b[i]=n%2;
n=n/2;
if(n= =0 )  break;
}
}
void main ( ) {
int i,j,n,b[MAX],temp[MAX];
float tw ,maxv, w[MAX],v[MAX], temp_w, temp_v;
printf("please input n:\n");
scanf("%d",&n);
printf("please input tw:\n");
scanf("%f",&tw);
printf("please input the values of w[]:\n");
for(i=0;i<n;i++ ) scanf("%f",&w[i]);
printf("\n");
printf("please input the values of v[]:\n");
for(i=0;i<n;i++) scanf("%f",&v[i]);
maxv=0;
for(j=0;j<n;j++) {b[j]=0;temp[j]=0;}
for(i=0;i<pow(2,n);i++ )       /*穷举2n种可能的选择，找出物品的最优选择*/
{
conversion(i,b);
temp_w=0;
temp_v=0;
for(j=0;j<n;j++)              /*试探当前选择是否是最优选择*/
if(b[j]==1)
```

```
{  temp_w=temp_w+w[j];
   temp_v=temp_v+v[j];
}
if((temp_w<=tw)&&(temp_v>maxv))
{  maxv=temp_v;
   for(j=0;j<n;j++) temp[j]=b[j];
}
}
printf("the max values is %f:\n",maxv);     /*输出最优选择*/
printf("the selection is:\n");
for(j=0;j<n;j++) printf("%d",temp[j]);
}
```

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

变量和相等问题

24.3.3 变量和相等问题

问题描述：将a、b、c、d、e、f这6个变量排成如图24-3（a）所示的三角形，这6个变量分别取1~6的整数，且均不相同。求使三角形三条边上的变量之和相等的全部解。如图24-3（b）所示为一个解。

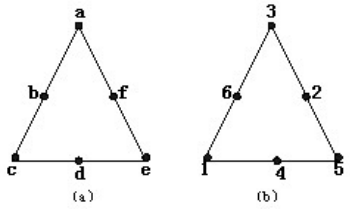


图24-3 6个变量组成三角形的排列形式及其解

程序引入变量a、b、c、d、e、f,并让它们分别顺序取1~6的证书，在它们互不相同的条件下，测试由它们排成的如图所示的三角形三条边上的变量之和是否相等，如相等即为一种满足要求的排列，把它们输出。当这些变量取尽所有的组合后，程序就可得到全部可能的解。

该问题算法程序实现见程序24-6.

【程序24-6】

```
#include <stdio.h>
void main ( )
{int a,b,c,d,e,f;
for ( a=1;a<=6;a++ )
for ( b=1;b<=6;b++ ) {
if ( b= =a ) continue;
```

```
for ( c=1;c<=6;c++ ) {
if ( ( c= =a ) || ( c= =b ) ) continue;
for ( d=1;d<=6;d++ ) {
if ( ( d= =a ) || ( d= =b ) || ( d= =c ) ) continue;
for ( e=1;e<=6;e++ ) {
if ( ( e= =a ) || ( e= =b ) || ( e= =c ) || ( e= =d ) ) continue;
f=21-( a+b+c+d+e );
if ( ( a+b+c= =c+d+e ) && ( a+b+c= =a+e+f ) ) {
printf( "%6d\n",a );
printf( "%4d%4d\n",b,f );
printf( "%2d%4d%4d\n",c,d,e );
scanf( "%*c" );
}
}
}
}
}
}
```

版权方授权希赛网发布，侵权必究

上一节 本书简介 下一节

作者：希赛教育软考学院 来源：希赛网 2014年01月27日

递推法

24.4 递推法

递推法实际上是需要抽象为一种递推关系，然后按递推关系求解。递推法通常表现为两种方式：一是从简单推到一般；二是将一个复杂问题逐步推到一个已知解的简单问题。这两种方式反映了两种不同的递推方向，前者往往用于计算级数，后者与"回归"配合成为一种特殊的算法--递归法。

由简单推到一般的方法，一般是从前面已知的各项（组）的值，采用层层递推，最后得到后面的某个要求的某项（组）数值。如当求解问题的规模为N,且当N=1时解已知或很容易得到解。若要求规模为n时的解，则可以先从规模N=1时求解，再根据N=1时的解求规模N=2时的解，这样依次递推，可求得N=n-1时的解，再根据规模N=n-1时的解即可求得规模n时的解。递推法通常用于计算级数第n项的值。

版权方授权希赛网发布，侵权必究

上一节 本书简介 下一节

最小数生成问题

24.4.1 最小数生成问题

问题描述：按递增次序生成集合M的最小的n个数。M定义如下：

l1∈M.

l若x∈M,则2x+1∈M,3x+1∈M.

l无别的数属于M.

要生成数组M中的最小的n个数，首先1为这n个数中的第一个数，再由1递推出余下的n-1个数。设n个数在数组M中，2x+1与3x+1均作为一个队列，从两队列中选一排头（数值较小者）送入数组M中，所谓"排头"就是队列中尚未选入M的第一个小的数。这里用p2表示2x+1这一列的排头，用p3表示3x+1这一列的排头。

该问题算法程序实现见程序24-7.

【程序24-7】

```
#include<stdio.h>
#define s 100
main（ ）
{ int m[s];
int n,p2,p3,i;
m[1]=p2=p3=1;
scanf（"%d",&n）；
for（i=2;i<=n;i++）
if（2*m[p2]= =3*m[p3]）
{m[i]=2*m[p2]+1;p2++;p3++;} /*如果2x+1=3x+1,则同时调整p2和p3值*/
else if（2*m[p2]<3*m[p3]）
/*如果2x+1<3x+1,2x+1的排头送入数组，只调整p2值*/
{m[i]=2*m[p2]+1;p2++;}
else
/*如果2x+1>3x+1,3x+1的排头送入数组，只调整p3值*/
{m[i]=3*m[p3]+1;p3++;}
printf（"\n"）；
for（i=1;i<=n;i++）
{printf（"%4d",m[i];
if（！（i%10））printf（"\n"）；
}
}
```

由上述程序我们可以发现，每一个m的元素都是由这个元素前面的元素递推得到的。初始时m[1]=1,由m[1]依次递推得到后面的各个m的元素。

版权方授权希赛网发布，侵权必究

阶乘计算

24.4.2 阶乘计算

问题描述：编写程序，对给定的n（n≤100），计算并输出k的阶乘k!（k=1,2,…，n）的全部有效数字。

要求得阶乘k!的值，必定已经求得了（k-1）！的值，依次递推，当k=2时，要求得的1!=1为已知。求得（k-1）！的值后，对（k-1）！连续累加k-1次后即可求得k!值。例如，已知5!=120,计算6!,可对原来的120累加5次120后得到720.

由于k!可能大大超出一般整数的位数，因此程序用一个一维数组存储长整数，存储长整数数组的每个元素只存储长整数的一位数字。如有m位长整数N用数组a[ ]存储：

$$N=a[m]\times10^{m-1}+a[m-1]\times10^{m-2}+\cdots+a[2]\times10^{1}+a[1]\times10^{0}$$

并用a[0]存储长整数N的位数m,即a[0]=m.按上述约定，数组的每个元素存储k的阶乘k!的一位数字，并从低位到高位依次存于数组的第二个元素、第三个元素……例如，6!=720,在数组中的存储形式为：

a[0]	a[1]	a[2]	a[3]	……
3	0	2	7	……

a[0]=3表示长整数是一个3位数，接着从低位到高位依次是0、2、7,表示成整数720.

该问题算法程序实现见程序24-8.

【程序24-8】

```
# include <stdio.h>
# include <malloc.h>
# define  MAXN1000
void pnext ( int a[ ],int k )                    /*已知a中的（k-1）！,求k! */
{int *b,m=a[0],i,j,r,carry;
b= ( int * ) malloc ( sizeof ( int ) * ( m+1 ) ) ;
for ( i=1;i<=m;i++ ) b[i]=a[i];
for ( j=1;j<k;j++ )                    /*控制累加k-1次*/
{for ( carry=0,i=1;i<=m;i++ )
{r= ( i<=a[0]?a[i]+b[i]:a[i] ) +carry;
a[i]=r%10;
carry=r/10;
}
if ( carry ) a[++m]=carry;
}
free ( b ) ;
```

```
a[0]=m;
}
void  write ( int *a,int k )
{int i;
printf ( "%4d!=",k ) ;
for  ( i=a[0];i>0;i-- ) printf ( "%d",a[i] ) ;
printf ( "\n\n" ) ;
}
void main ( )
{int a[MAXN],n,k;
printf ( "Enter the number n:  " ) ;
scanf ( "%d",&n ) ;
a[0]=1;
a[1]=1;
write ( a,1 ) ;
for  ( k=2;k<=n;k++ )
{pnext ( a,k ) ;
write ( a,k ) ;
getchar ( ) ;
}
}
```

版权方授权希赛网发布，侵权必究

第 24 章：常用算法设计　作者：希赛教育软考学院　来源：希赛网　2014年01月27日

递归法

24.5 递归法

递归是一种特别有用的工具，不仅在数学中广泛应用，在日常生活中也常常遇到。例如一个画家画的如图24-4所示的画便是一种递归的图形。

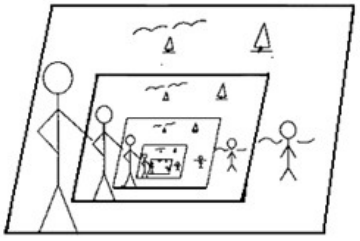


图24-4 递归图形

递归是设计和描述算法的一种有力的工具，由于它在复杂算法的描述中被经常采用，能采用递归描述的算法通常有这样的特征：为求解规模为N的问题，设法将它分解成规模较小的问题，然后从

这些小问题的解方便地构造出大问题的解，并且这些规模较小的问题也能采用同样的分解和综合方法，分解成规模更小的问题，并从这些更小问题的解构造出规模较大问题的解。特别地，当规模N=1时，能直接得解。

递归算法包括"递推"和"回归"两部分。递推就是为得到问题的解，将它推到比原问题简单的问题的求解。如f（n）=n!,为计算f（n），将它推到f（n-1），即f（n）=nf（n-1），这就是说，为计算f（n），将问题推到计算f（n-1），而计算f（n-1）比计算f（n）简单，因为f（n-1）比f（n）更接近于已知解0!=1.

使用递推时应注意以下条件。

递推应有终止的时候。例如n!,当n=0时，0!=1为递推的终止条件。所谓"终止条件"就是在此条件下问题的解是明确的，缺少终止条件便会使算法失效。

"简单问题"表示离递推终止条件更为接近的问题。简单问题与原问题解的算法是一致的，其差别主要反映在参数上。如，f（n-1）与f（n）其参数相差1.参数变化，使问题递推到有明确解的问题。

回归是指当"简单问题"得到解后，回归到原问题的解上来。如，当计算完f（n-1）后，回归计算nf（n-1），即得n!的值。

使用回归应注意以下问题。

递归到原问题的解时，算法中所涉及的处理对象应是关于当前问题的，即递归算法所涉及的参数与局部处理对象是有层次的。当解一问题时，有它的一套参数与局部处理对象。当递推进入一"简单问题"时，这套参数与局部对象便隐蔽起来，在解"简单问题"时，又有它自己的一套。但当回归时，原问题的一套参数与局部处理对象又活跃起来了。

有时回归到原问题以得到问题解，回归并不引起其他动作。

例如计算n!,其公式为：

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

程序片段见程序24-9.

【程序24-9】

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

图的深度优先搜索、二叉树的前序、中序和后序遍历等可采用递归实现。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

第 24 章：常用算法设计　作者：希赛教育软考学院　来源：希赛网　2014年01月27日

斐波那契（Fibonacci）数列

24.5.1 斐波那契（Fibonacci）数列

问题描述：编写计算斐波那契（Fibonacci）数列，数列大小为n.

无穷数列1,1,2,3,5,8,13,21,35,…，称为斐波那契数列，其递归定义如下：

$$F(n)=\begin{cases}1 & n=0\\ 1 & n=1\\ F(n-1)+F(n-2) & n>1\end{cases}$$

由斐波那契数列的递归定义可知，当n大于1时，这个数列的n项的数值是它前面两项之和。递归算法的执行过程分递推和回归两个阶段。在递推阶段，程序把较复杂的问题（规模为n）的求解推到比原问题简单一些的问题（规模小于n）的求解；在回归阶段，程序由在规模很小时求得的解得到较复杂问题的解。因此，对于斐波那契数列的求解，要得到数列第n项的值，就必须求得数列第n-1项的值，同理，要求得数列n-1的值，就必须求得数列n-2项的值，依次递推下去，最后需要求得数列第1项和第0项的值，递推部分结束。又当n等于0和1时，其数值为1,根据这些值可求出数列第二项的值，再根据数列第二项的值可得到第三项的值，依次回归，最后可依次得到数列第n-2、n-1、n项的值。由这个例子，我们可以很清楚地了解递归的形式和过程。该问题程序实现见程序24-10.

【程序24-10】

```
# include< stdio.h >
int  F ( int n )
{if ( n= =0 ) return  1;
if ( n= =1 ) return  1;
if ( n>1 ) return  F ( n-1 ) +F ( n-2 ) ;        /*递归*/
}
main ( ) {
int i,n,m;
printf ( "please input n: \n" ) ;
scanf ( "%d",&n ) ;
printf ( "the Fibonacci is : " ) ;
for ( i=0;i<=n;i++ ) {
m=F ( i ) ;
printf ( "%d,",m ) ;
}
}
```

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

字典排序问题

24.5.2 字典排序问题

问题描述：本程序将一段以"*"结束的文本中的单词按字典序打印出来，并且打印出该单词在正文中的出现次数。

考虑到二叉排序树可以很容易地以递归方式实现，我们使用二叉排序树来实现文本中单词按字典序排序。程序中使用二叉排序树存入单词，即每次从文件中读出一个单词，都将其按单词的字典顺序存入一个二叉排序树，第一个存入的单词为二叉排序树的根。读完文件中的单词后，中序遍历打印出二叉排序树中存放的各个单词。

为了使存储空间使用更加合理且能够处理任意长度的单词，程序设立了一个数组text,所有读入的单词都放在text数组中。函数getword 完成读入单词的操作，并返回所读入的单词的长度。函数insert完成在二叉排序树中插入一个新结点的操作并返回指向二叉树根结点的指针。

该问题算法程序实现见程序24-11.

【程序24-11】

```
# include< stdio.h >
# include< malloc.h >
char ch=' ';
typedefine struct node * tree
struct node
{ char * data ;
int  count ;
tree  lchild ;
tree  rchild ;
} ;
getword （ word ）                /*读取单词子程序，返回单词字符个数*/
char * word ;
{ int i = 0 ;
if （ ch = = '*' ） return （ 0 ） ;
while （ ch = = ''||ch = ='\ t' | |ch= ='\n' ）
ch = getchar （ ） ;               /*去掉前导空白符或其他非法操作*/
while （ ch! = ' '&& ch !='\ t'&&ch !='\n'&& ch !='#' ）
{ word[i++]=ch;                  /*输入单词字符存入数组word*/
ch = getchar （ ） ;              /*输入下一个字符*/
}
word [i] = 0;                   /*单词末尾用字符0表示结束*/
return （ i ） ;                  /*返回单词个数*/
}
tree insert （ root , x ）              /*将数组x中的单词插入到二叉排序树中*/
tree root;
char * x;
{ tree p ;
int res;
if （ root = = NULL ）              /*若排序树根结点为空，则置x为根结点*/
{ p = （ tree ） malloc （ sizeof （ * p ） ） ;
```

```
p–>data = x;
p–>count=1;
p–>lchild = NULL ; p–>rchild = NULL;
return （ p ）； }                    /*根结点返回*/
else if （ （ res = strcmp （ x, root–>data ） ） < 0 ） /*若x小于根结点数据域单词，则搜索其左子树*/
root–>lchild=insert ( root–>lchild,x ）；  /*递归搜索左子树*/
else if ( res>0 )
root–>rchild=insert ( root–>rchild,x ）；  /*递归搜索其右子树*/
else
root–>count++;                /*若x等于根结点，则根结点计数器加1*/
return （root ）；
}
print_tree （root ）；                    /*打印二叉排序树*/
tree root;
{
if ( root!=NULL )
{ print_tree ( root–>lchild ）；
printf ( "%d %s \n", root–>count, root–>data ）；
print_tree ( root–>rchild ）；
}
}
main （ ）
{  int len;
char *word,*text;
tree  root;
root=NULL;
word=text;          /*输入的所有单词要求放到text中，故初始word=text */
while （ ( len=getword ( word ）） ！=0 )
{ root = insert ( root , word ）； /*输入单词，若不空则将其插入到二叉排序树*/
word + =（len+1 ）；      /*调整下一个将要输入的单词的储存位置*/
}
print_tree ( root ）；          /*打印输出*/
}
```

本程序由4个部分组成，3个子程序和一个主程序。子程序getword用来输入单词；子程序insert用来将单词插入到中序排序二叉树中；子程序print_tree用来中序遍历二叉排序树，并打印。单词存放在数组text中，且单词和单词之间用字符"0"分开。初始时数组word和text的首地址相等。

版权方授权希赛网发布，侵权必究

 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

贪婪法

24.6 贪婪法

贪婪法是一种重要的算法设计技术，它总是做出在当前来说是最好的选择，而并不从整体上加以考虑，它所做的每步选择只是当前步骤的局部最优选择，但从整体来说不一定是最优的选择。由于它不必为了寻找最优解而穷尽所有可能解，因此其耗费时间少，一般可以快速得到满意的解。当然，我们也希望贪婪算法所得到的最终解是整体的最优解。对贪婪法的理解如下。

贪婪法不追求最优解，只求可行解，通俗点讲就是不求最好，只求可好。

每一步都按贪婪准则找一个解，故到n步后（n为问题的规模）得到问题的所有解。如找不到所有解，则修改贪婪准则，放宽贪婪条件或修改算法某些细节，重新从头开始找解。

每一步所找到的解是这一步中的最优解（按贪婪准则来说），但每步最优解所形成的整体解并不一定最优。

算法思想如下：

在贪婪算法中采用逐步构造最优解的方法。在每个阶段，都做出一个看上去最优的决策（在一定的贪婪标准下），决策一旦做出，就不可再更改。做出贪婪决策的依据称为贪婪准则。

应用贪婪法求解问题，首要的问题就是要弄清楚贪婪准则。但是我们应该看到，虽然在每个阶段做出的都是看上去最优的决策，但这些决策的集合从整体来说有可能并不是最优的。

例如，一个小孩买了价值少于1美元的糖，并将1美元钱交给售货员。售货员希望用数目最少的硬币找给小孩。假设提供了数目不限的面值为25美分、10美分、5美分和1美分的硬币。售货员分步骤组成要找的零钱数，每次加入一个硬币。

选择硬币时所采用的贪婪准则如下：每一次选择应使零钱数尽量增大。为保证解法的可行性（即所给的零钱等于要找的零钱数），所选择的硬币不应使零钱总数超过最终所需的数目。假设需要找给小孩67美分，首先入选的是两枚25美分的硬币，第三枚入选的不能是25美分的硬币，否则硬币的选择将不可行（零钱总数超过67美分），第三枚应选择10美分的硬币，然后是5美分的，最后加入两个1美分的硬币。可以证明采用上述贪婪算法找零钱时所用的硬币数目的确最少，这是一个得到最优解的例子。

同样是找零钱问题，如果假设提供的是数目不限的面值为8美分、5美分和1美分的硬币。若需要找给小孩20美分，按照上述贪婪准则，首先入选的是两枚8美分硬币，然后是4枚1分硬币，共找回6枚硬币。显然这不是最优的解，最优解应该是4枚5美分的硬币。虽然在找零钱的每步都应用了最优找法，但后一个例子的解却不是最优解。

所以我们说贪婪法是一种不求最优解，只是希望得到满意解的方法，即不求最好，只求可好，求第一次满足条件的解。一般来说，这个解却是最优解的很好的近似解。当然，贪婪法所求得的解也有可能是最优解，而且对范围相当广的许多问题它能产生整体最优解。下面介绍一些应用贪婪算法的典型问题。

版权方授权希赛网发布，侵权必究

背包问题

24.6.1 背包问题

问题描述：给定n种物品和一个背包。物品i的重量是wi,其价值为vi,背包的容量为C,问应该如何选择装入背包的物品，使得装入背包中的物品的总价值最大？

背包问题可分为两种：

0-1背包问题：对于每种物品i装入背包只有一种选择，即要么装入背包或不装入背包，不能装入多次或只装入部分。

部分背包问题：对于每种物品i可以只装入部分。

我们可以采用贪婪算法来解决背包问题。先对每种物品计算其单位重量价值vi/wi,然后按单位价值单调递减的顺序对所有的物品进行排序。按照贪婪准则，开始时放入背包的物品单位重量价值尽可能大，基于这种思想，我们可以将按单位重量价值排序好的物品依次放入到背包中。当某个物品装入过程中，若其重量大于背包所剩余装载重量时，对于0-1背包问题，此时装入过程完成，得到问题的解，很显然这不是问题的最优解。对于部分背包问题，可以将部分的最后物品装入背包，使得背包的重量容量被装满，显然此时背包物品的总价值要大于0-1背包，是问题的一个最优解。上述两个背包问题的举例如图24-5所示。

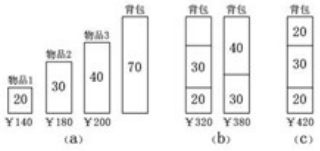


图24-5 背包问题的一个例子

图24-5例子中，图24-5（b）为0-1背包问题，可知由贪婪算法所得解中背包物品总价值只有320,而最优解为380.图24-5（c）为部分背包问题，由贪婪算法所得解中物品总价值为420,为最优解。

对于0-1背包问题，贪婪法之所以不能得到最优解是因为它无法保证最终能将背包容量装满，背包空间的闲置使得背包所装物品的总价值降低了。

0-1背包问题的算法简单描述如下：

输入物品个数n,每个物品的重量wi和价值vi.

对物品按单位重量价值wi/vi从大到小进行排序。

将排序后的物品依次装入背包。对于当前物品i,若背包剩余可装重量大于或等于wi,则将物品i装入背包，继续考虑下一个物品i+1,重复步骤（3），否则得到问题的解，输出。

0-1背包问题的算法程序实现见程序24-12.

【程序24-12】

```
#include<stdio.h>
#define MAX 100                    /*最多物品数*/
sort（int n,float a[MAX],float b[MAX]）    /*对储存物品重量和价值的数组进行排序 */
{                                  /*采用冒泡法排序*/
```

```
int j,p,h,k;
float t1,t2,t3,c[MAX];
for（k=1;k<=n;k++）                /*求物品单位重量价值*/
c[k]=a[k]/b[k];
for（h=n;h>1;h=p）{
for（p=j=1;j<h;j++）
if（c[j]<c[j+1]）{
t1=a[j];a[j]=a[j+1];a[j+1]=t1;
t2=b[j];b[j]=b[j+1];b[j+1]=t2;
t3=c[j];c[j]=c[j+1];c[j+1]=t3;
p=j;
}
}
}
/*背包装载物品子程序，limitw为背包可装载重量*/
knapsack（int n,float limitw,float v[MAX],float w[MAX],int x[MAX]）
{  float c1;                /*c1为背包剩余可装载重量*/
int i;
sort（n,v,w）；               /*物品按单位重量大小降序排序*/
c1=limitw;
for（i=1;i<=n;i++）{
if（w[i]>c1）break;
x[i]=1;               /*x储存物品选择情况，当x[i]为1时，物品i在解中*/
c1-=w[i];
}
/*对于部分背包问题，此行需添加语句 if（i<=n）x[i]=c1/w[i];*/
}
main（）
{
int n,i,x[MAX];
float v[MAX],w[MAX],totalv=0,limitw;
printf（"please input n and limitw:"）；
scanf（"%d,%f",&n,&limitw）；
for（i=1;i<=n;i++）x[i]=0;      /*物品选择情况表初始化为0*/
for（i=1;i<=n;i++）
{
printf（"please input %d thing's value and weight:\n",i）；
scanf（"%f,%f",&v[i],&w[i]）；
}
```

```
knapsack（n,limitw,v,w,x）；
printf（"the selection is:\n"）；
for（i=1;i<=n;i++）
{
printf（"%d,",x[i]）；
totalv+=v[i]*x[i];       /*背包所装载总价值*/
}
printf（"\n"）；
printf（"the total value is: %f",totalv）；
}
```

贪婪法应用于0-1背包问题往往得不到最优解，下面是一种获得0-1背包问题最优解的算法。算法思想见递归法部分，同样的算法思想，我们考虑非递归的程序解。为了提高找解速度，算法简单来说就是考虑每个物品对候选解的影响来形成有效的临时候选解。一个有效临时候选解是通过依次考查每个物品形成的，对物品i的考查有这样两种情况：当考虑该物品i包含在候选解中时，如果其依旧满足解的总重量的限制，则应该将该物品包含在候选解中；反之，如果不满足解的总重量限制，则该物品不应该包括在当前正在形成的候选解中。第二种是当考虑物品i不包含在候选解中时，且有可能找到比目前临时最佳解更好的候选解时（此时条件为期望的总价值减去当前物品i的价值后仍大于目前临时最佳解），则应该将该物品不包含在候选解中；反之，该物品不包括在当前候选解中的方案也不应继续考虑，则回退。对于任一值得继续考虑的方案，程序就去进一步考虑下一个物品。程序实现见程序24-13.

【程序24-13】

```
# include<stdio.h>
# defineN100
doublelimitW;
intcop[N];       /*临时最佳候选解的选择方案，当cop[i]为1,物品i在解中*/
struct ele{
doubleweight;
doublevalue;
} a[N];     /*储存物品重量和价值的结构*/
intk,n;
struct{int flg;    /*物品的考虑状态，0:不选，1:将被考虑，2:曾被选中*/
double tw;   /*背包中已经装入的总重量*/
double tv;   /*期望达到的总价值*/
}twv[N];     /*当前候选解中各个物品的考虑和选择状态*/
void next（int i,double tw,double tv）/*将考虑物品i是否可以放入背包*/
{twv[i].flg=1; twv[i].tw=tw; twv[i].tv=tv; }
double find（struct ele *a,int n）
{int i,k,f;
double maxv,tw,tv,totv;
```

```
maxv=0;
for（totv=0.0,k=0;k<n;k++） totv+=a[k].value; /*初始时背包期望能装载总价值为所有物品
总价值*/
next（0,0.0,totv）;                    /*0号物品将被考虑*/
i=0;
While（i>=0）
{ f=twv[i].flg; tw=twv[i].tw; tv=twv[i].tv;
switch（f）
{ case 1: twv[i].flg++;              /*先考虑选中的情况*/
if（tw+a[i].weight<=limitW）     /*选中是否满足条件*/
if（i<n-1）                    /*是否是最后一个物品*/
{ next（i+1,tw+a[i].weight,tv）; /*当前物品i被选中，继续考虑下一个物品*/
i++;
}
else {                  /*是一个更好的有效候选解*/
maxv=tv;
for（k=0;k<n;k++）
cop[k]=twv[k].flg!=0;
}
break;
case 0:i--; break;       /*回退*/
default:twv[i].flg=0;/*f= =2的情况，即被考虑选中的物品i不满足重量条件*/
if（tv-a[i].value>maxv）/*不选物品i可行吗*/
if（i<n-1）       /*是否是最后一个物品*/
{/*当前物品i被考虑不选中，继续考虑下一个物品*/
next（i+1,tw,tv-a[i].value）;
i++;
}
else {            /*是一个更好的有效候选解*/
maxv=tv-a[i].value;
for（k=0;k<n;k++）
cop[k]=twv[k].flg!=0;
}
break;
}
}
```

```
return maxv;
}
void main ( )
{double maxv;
printf ( "输入物品种数\n" ) ; scanf ( "%d",&n ) ;
printf ( "输入限制重量\n" ) ; scanf ( "%lf",&limitW ) ;
printf ( "输入各物品的重量和价值\n" ) ;
for ( k=0;k<n;k++ )
scanf ( "%lf%lf",&a[k].weight,&a[k].value ) ;
maxv=find ( a,n ) ;
printf ( "\n选中的物品为\n" ) ;
for ( k=0;k<n;k++ )
if ( cop[k] ) printf ( "%4d",k ) ;
printf ( "\n总价值为%2f\n",maxv ) ;
}
```

版权方授权希赛网发布，侵权必究

上一节 本书简介 下一节

 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

装箱问题

24.6.2 装箱问题

问题描述：设有编号为0,1,…，n-1的n种物品，体积分别为v0、v1、…、vn-1.将这n种物品装到容量都为V的若干箱子里。约定这n种物品的体积均不超过V,即对于0≤i<n,有0<vi≤V.可知选择不同的装箱方案所需要的箱子数目可能不同。装箱问题要求使装尽这n种物品的箱子数要少。

要寻找该问题的最优解，可以将n种物品划分为小于或等于n的子集，考查所有的划分，可以找出满足条件且箱子数最少的划分，即为最优解。但要穷尽所有可能划分的总数则太大。对于大到一定程度的n,找出所有可能的划分要花费的时间是无法承受的。为此，对装箱问题采用寻找最优解的近似算法，即贪婪法，可以很快地找到最优解的近似解。该算法贪婪准则是：依次将物品放到它第一个能放进去的箱子中，若当前箱子装不下当前物品，则启用一个新的箱子装该物品，直到所有的物品都装入了箱子。该算法虽不能保证找到最优解，但还是能找到非常好的解而不失一般性。设n件物品的体积是按从大到小排好序的，即有v0≥v1≥…≥vn-1.如不满足上述要求，只要先对这n件物品按它们的体积从大到小排序，然后按排序结果对物品重新编号即可。

算法简单描述：

{输入箱子的容积；

输入物品种数n;

按体积从大到小顺序，输入各物品的体积；

预置已用箱子链为空；

```
预置已用箱子计数器box_count为0;
for（i=0;i<n;i++）
{从已用的第一只箱子开始顺序寻找能放入物品i 的箱子j;
if （已用箱子都不能再放物品i）
{新启用一个箱子，并将物品i放入该箱子；
box_count++;
}
else
将物品i放入箱子j;
}
}
```

上述算法一次就能求出需要的箱子数box_count,并能求出各箱子所装物品，但该算法不一定能找到最优解。如下面例子所示，设有6种物品，它们的体积分别为：60、45、35、20、20和20单位体积，箱子的容积为100个单位体积。按上述算法计算，需3只箱子，各箱子所装物品分别为：第一只箱子装物品1、3;第二只箱子装物品2、4、5;第三只箱子装物品6.而最优解为两只箱子，分别装物品1、4、5和2、3、6.该算法也能够找到最优解。下面的例子说明了这种情况，设有6种物品，它们的体积分别为：60、35、25、20、20和20单位体积，箱子的容积为100个单位体积。按上述算法计算，可以得到最优解为两个箱子，分别装物品1、2和3、4、5、6.

该问题算法程序实现见程序24-14.

【程序24-14】

```
# include<stdio.h>
# include<stdlib.h>
typedef  struct  ele{        /*物品结构的信息*/
int  vno;                /*物品号*/
struct  ele  *link;     /*指向下一物品的指针*/
}ELE;
typedef  struct  hnode{  /*箱子结构信息*/
int  remainder;             /*箱子的剩余空间*/
ELE  *head;          /*箱子内物品链的首元指针*/
struct  hnode  *next;  /*箱子链的后继箱子指针*/
}HNODE;
void  main（ ）
{int  n, i, box_count, box_volume, *a;
HNODE  *box_h,  *box_t,  *j;
ELE  *p,  *q;
printf（"输入箱子容积\n"）；scanf（"%d",&box_volume）；
printf（"输入物品种数\n"）；scanf（"%d",&n）；
a=（int *）malloc（sizeof（int）*n）；
printf（"请按体积从大到小顺序输入各物品的体积："）；
```

```
for（i=0;i<n;i++）scanf（"%d",a+i）；/*数组a按从大到小顺序存放各物品的体积信息*/
box_h=box_t=NULL;  /*box_h为箱子链的首元指针，box_t为当前箱子的指针，初始为空*/
box_count=0;          /*箱子计数器初始也为0*/
for（i=0;i<n;i++）          /*物品i按下面各步开始装箱*/
{p=（ELE *）malloc（sizeof（ELE））；
p->vno=i;          /*指针p指向当前待装物品*/
/*从第一只箱子开始顺序寻找能放入物品i的箱子j*/
for（j=box_h;j!=NULL;j=j->next）
if（j->remainder>=a[i]）break; /*找到可以装物品i的箱子，贪婪准则的体现*/
if（j==NULL）{          /*已使用的箱子都不能装下当前物品i*/
j=（HNODE *）malloc（sizeof（HNODE））；/*启用新箱子*/
j->remainder=box_volume-a[i];  /*将物品i放入新箱子j*/
j->head=NULL;          /*新箱子内物品链首元指针初始为空*/
if（box_h==NULL）box_h=box_t=j; /*新箱子为第一个箱子*/
elsebox_t=boix_t->next=j;  /*新箱子不是第一个箱子*/
j->next=NULL;
box_count++;
}
elsej->remainder-=a[i];       /*将物品i放入已用过的箱子j*/
/*物品放入箱子后要修改物品指针链*/
for（q=j->head;q!=NULL&&q->link!=NULL;q=q->link）；
if（q==NULL）{          /*新启用的箱子插入物品*/
p->link=j->head; j->head=p;  /*p为指向当前物品的指针*/
}
else{                  /*已使用过的箱子插入物品*/
p->link=NULL; q->link=p;   /*q为指向箱子内物品链顶端的物品*/
}
}
printf（"共使用了%d只箱子", box_count）；
printf（"各箱子装物品情况如下："）；
for（j=box_h,i=1;j!=NULL;j=j->next,i++）     /*输出i只箱子的情况*/
{printf（"第%2d只箱子，还剩余容积%4d,所装物品有；\n",i,j->remainder）；
for（p=j->head;p!=NULL;p=p->link）
printf（"%4d",p->vno+1）；
printf（"\n"）；
}
}
```

装箱问题所采用的数据结构为链表。用链表将启用的箱子链接起来，而且每个箱子所装入的物品也用一个链表将它们链接起来，这样就有两个链表：箱子链和物品链。程序将物品按体积从大到

小依次装入箱子。对每一个物品，从箱子链中第一个箱子开始顺序寻找能放入该物品的箱子（箱子剩余体积大于或等于当前物品体积），将物品放入最先找到的箱子。装箱的贪婪准则是：如果一旦找到能装下当前物品的箱子，就将当前物品放入，而不考虑其最优解情况；若所有启用的箱子都装不下当前物品，则开启一个新箱子。

程序中箱子链首元指针为box-h,当前箱子指针为box-t,箱内物品链首元指针为j->head.装入物品5的过程如图24-6所示。

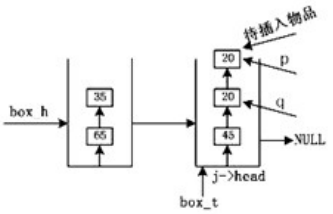


图24-6 物品装入过程图

版权方授权希赛网发布，侵权必究

第 24 章：常用算法设计 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

马踏棋盘问题

24.6.3 马踏棋盘问题

问题描述：在8×8方格的棋盘上，从任意指定的方格出发，为马寻找一条走遍棋盘每一格并且只经过一次的一条路径。

熟悉国际象棋的人都知道，马在某个方格，可以在一步内到达的不同位置最多有8个，如图24-7所示。如用二维数组board[ ][ ]表示棋盘，其值记录马经过该位置时的步骤号。对马的8种可能走法设定一个顺序，如当前位置在棋盘的（i,j）方格，下一个可能的位置依次为（i+2, j+1）、（i+1, j+2）、（i-1, j+2）、（i-2, j+1）、（i-2, j-1）、（i-1, j-2）、（i+1, j-2）和（i+2, j-1），实际可以走的位置很明显仅仅限于还未走过的和不越出边界的那些位置。这里我们定义马在一步内实际可以走的位置数为马在当前位置的出口数。此外，为便于程序的统一处理，这里引入两个数组deltai_i和deltai_j,分别存储各种可能走法对马所在当前位置下标的纵横增量，如图24-7所示。

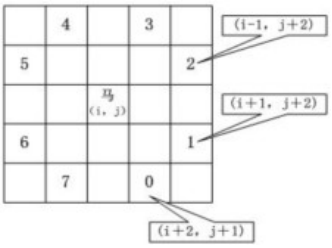


图24-7 马所有可能的走法示意图

本题可以采用回溯法求解，这里采用Warnsdoff策略求解，这是一种贪婪法。当马处于某一个位置时，其选择下一个位置的贪婪准则为：从马当前位置所允许走的所有位置中，选择出口数最少的那个位置。如马的当前位置（i, j）只有3个实际可走出口，它们是位置（i+2, j+1）、（i+1, j+2）和（i-1, j+2），这3个位置的出口数分别为4、2、3,则程序就选择让马走向（i+1, j+2）位置。

算法简单描述：

{马从棋盘第一行第一列位置开始出发；

预设着法选择顺序控制变量start为0;

do{

棋盘数组初始化为0;

设置棋盘数组马起始位置数组值为1;

i=马起始位置数组行下标；j=马起始位置数组列下标；

for（step=2;step<64;step++）{

if（马当前位置没有出口）break;

i+=马所选择的下一位置相对当前位置行下标的增量；

j+=马所选择的下一位置相对当前位置列下标的增量；

board[i][j]=step;}

if（找到解）终止do 循环；

else start++;

} while（没有找到解）；

输出棋盘数组board[][]值；

上述算法采用贪婪法，整个找解过程一直向前，没有回溯，所以能非常快地找到解。但是，对于某些开始位置，实际有解而程序可能第一次找不到解，则程序只要改变8种可能出口的选择顺序，就能找到解。一般情况下，程序是从0号出口开始选择出口的，依次比较0号出口至8号出口的出口数（假设马当前位置有8个出口），找出出口数最小的那个出口为马的下一个位置。假设0号出口和3号出口的出口数相等且最小，由于是从0号出口开始选择，0号出口后面没有比它出口数更小的出口，故程序选择0号出口为下一个位置，如果程序是从2号出口开始选择，程序将选择3号出口为下一个位置。考虑到这种情况，程序引入变量start,用于控制8种可能着法的选择顺序。开始时为0,当不能找到解时，就让start增1,重新找解。

该问题算法程序实现见程序24-15.

【程序24-15】

```
# include<stdio.h>
int delta_i[ ]={2,1,-1,-2,-2,-1,1,2};    /*存放马各个出口位置相对当前位置行下标的增量数组*/
int delta_j[ ]={1,2,2,1,-1,-2,-2,-1};    /*存放马各个出口位置相对当前位置列下标的增量数组*/
int board[8][8];
int exitn（int i,int j,int s,int a[ ]）       /*求（i, j）的出口数子程序*/
{int i1,j1,k,count;
for（count=k=0;k<8;k++）      /*从s号出口开始依次考查8个出口是否可走*/
{i1=i+delta_i[（s+k）%8];
j1=j+delta_j[（s+k）%8];
if（i1>=0&&i1<8&&j1>=0&&j1<8&&board[i1][j1]= =0）/*出口没有越界且马没有走过，则出口数加1*/
a[count++]=（s+k）%8;   /*数组a记录（i, j）所有可以走的出口号*/
}
```

```
return count;                    /*返回出口数*/
}
/*求（i,j）的下一个出口子程序，即求各个出口中出口数最小的出口，从s号出口开始考查*/
int next（int i,int j,int s）
{int m,k,kk,min,a[8],b[8],temp;
m=exitn（i,j,s,a）；                  /*求（i, j）的出口数给m*/
if （m= =0）return -1;       /*没有出口*/
for （min=9,k=0;k<m;k++）          /*逐一考查（i, j）各出口*/
{temp=exitn（i+delta_i[a[k]],j+delta_j[a[k]],s,b）；  /*求（i, j）第a[k]号出口的出口数*/
if （temp<min）
{min=temp;          /*求（i, j）的出口中出口数最小的出口*/
kk=a[k];
}
}
return  kk;                    /*返回（i, j）下一个出口*/
}
void main（）                          /*求解*/
{ int sx,sy,i,j,step,no,start;
for （sx=0;sx<8;sx++）
for （sy=0;sy<8;sy++）{          /*求棋盘64个不同位置为起始位置的解*/
start=0;                       /*初始时从0号出口开始考查*/
do {
for （i=0;i<8;i++）
for （j=0;j<8;j++） board[i][j]=0;  /*棋盘初始化，即清棋盘*/
board[sx][sy]=1;                /*马开始位置步骤为1*/
i=sx; j=sy;
for （step=2;step<64;step++）       /*马开始行走，棋盘数组board记录行走轨迹*/
{  if （（no=next（i,j,start））= =-1） break;  /*无出口可走*/
i+=delta_i[no];              /*走到下一个出口*/
j+=delta_j[no];
board[i][j]=step;
}
if （step>=64） break;             /*找到一个解*/
start++;                    /*求解失败，修改顺序选择着法的开始序号*/
} while（step<=64）
for （i=0;i<8;i++）                    /*输出解*/
{ for （j=0;j<8;j++）
printf（"%4d",board[i][j]）；
printf（"\n\n"）；
```

```
}
scanf ( "%*c" ) ;
}
}
```

马所能走的最多的出口号为0、1、2、3、4、5、6、7,实际情况有些出口不能走。数组a[]记录所有能走的出口号，如（i, j）位置，马实际能走的出口号为1、4、6,则a[0]=1,a[1]=4,a[2]=6.再通过数组delta来表示马各个出口位置相对当前位置（i, j）下标的增量。上例中，假设（i, j）下一步选择4号出口，则4号出口的坐标为（i+delta_i[a[1]], j+delta_j[a[1]].程序运行所得一个解如表24-1所示。

表24-1 起始位置为（0,0）的一个解

1	34	3	18	49	32	13	16
4	19	56	33	14	17	50	31
57	2	35	48	55	52	15	12
20	5	60	53	36	47	30	51
41	58	37	46	61	54	11	26
6	21	42	59	38	27	64	29
43	40	23	8	45	62	25	10
22	7	44	39	24	9	28	63

版权方授权希赛网发布，侵权必究

 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

货郎担问题

24.6.4 货郎担问题

问题描述：所谓货郎担问题是指，给定一个无向图，并已知各边的权，在这样的图中，要找到一个闭合回路，使回路经过图中的每一个点，而且回路各边的权之和为最小。

例如有A、B、C、D、E、F 6个点，已知各点的坐标分别为（0,0）、（4,3）、（1,7）、（15,7）、（15,4）、（18,0），设两点间各边的权为边的长度，如图24-8所示回路，该回路各边的权之和为最小，这是最优解。

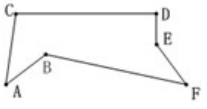


图 24-8 回路示意图

应用贪婪法求解该问题，并不一味追求最优解，但求解应使人满意。程序先计算由各点构成的所有边的长度，按长度大小对各边进行排序后，按贪婪准则从排序后的各边中选择边组成回路的边，贪婪准则使得边的选择按各边长度从小到大选择。

算法简单描述如下。

（1）先计算各点间距离，即各边的长度，如表24-2所示。

表24-2 距离关系表

	A	B	C	D	E	F
A		5	sqrt(50)	Sqrt(274)	Sqrt(241)	18
B			5	Sqrt(137)	Sqrt(122)	Sqrt(205)
C				14	Sqrt(205)	Sqrt(338)
D					3	Sqrt(58)
E						5
F						

ABCDEF

A5sqrt（50）Sqrt（274）Sqrt（241）18

B5Sqrt（137）Sqrt（122）Sqrt（205）

C14Sqrt（205）Sqrt（338）

D3Sqrt（58）

E5

F

（2）6个点所构成边的条数m==15条，根据表24-3对边进行排序，形成边排序表，如表24-3所示。

表24-3 边排序表

边长度	边端点 1	边端点 2	边长度	边端点 1	边端点 2
3	D	E	14	C	D
5	A	B	Sqrt(205)	B	F
5	B	C	Sqrt(205)	C	E
5	E	F	Sqrt(241)	A	E
Sqrt(50)	A	C	Sqrt(274)	A	D
Sqrt(58)	D	F	18	A	F
Sqrt(122)	B	E	Sqrt(338)	C	F
Sqrt(137)	B	D			

边长度边端点1边端点2边长度边端点1边端点2

3DE14CD

5ABSqrt（205）BF

5BCSqrt（205）CE

5EFSqrt（241）AE

Sqrt（50）ACSqrt（274）AD

Sqrt（58）DF18AF

Sqrt（122）BESqrt（338）CF

Sqrt（137）BD

（3）应用贪婪准则选择构成回路的各个边，入选的边应符合如下3个条件：

从边排序表中依次按长度从小到大选取各边。

新选择的边加入后，不能使得某个端点联系两条以上的边。

新选择的边加入后，不能使入选的边形成回路，除非入选正好是边数等于端点数，即得到解。为此引入端点关系表，如表24-4所示。

表24-4 端点关系表

	A	B	C	D	E	F
度数	2	2	2	2	2	2
联系端	B	A	A	C	D	B
联系端	C	F	D	E	F	E

ABCDEF

度数222222

联系端BAACDB

联系端CFDEFE

（4）如果由（3）得不到解，应调整距离关系表中距离相同的边的次序，再试。

（5）若有解，则按端点关系表给出回路的轨迹。如表24-5所示。

表24-5 回路轨迹表

1	2	3	4	5	6	7
A	B	F	E	D	C	A

按上述贪婪算法，程序第一次可能找不到形成回路的所有边，此时需要修改边排序表中长度相等的边的排列次序，使得长度相等的边入选回路的先后顺序发生变化。

以上各表在程序中表示如下。

边排序表：设为dr,其类型为结构tdr的数组，tdr类型为struct{float x,int p1,p2;}

端点关系表：设为r,其类型为结构tr的数组，tr类型为struct{int n,p1,p2}.

端点坐标，设为pd,其类型为结构tpd的数组，tpd类型为struct{float x,y}.回路轨迹表为locus,其类型为数组类型tl.

该问题算法程序实现见程序24-16.

【程序24-16】

```
#define maxm 100
typedef struct {float x;int p1,p2;}tdr; /*x为两端点p1、p2之间的距离，即p1、p2所组成边的长度 */
typedef struct {int n,p1,p2;}tr;      /*p1、p2为和端点相联系的两个端点，n为端点的度数*/
typedef struct {float x,y;}tpd;      /*给出两点坐标 */
typedef int tl[maxm];
int n=10;                    /*n为点的个数*/
float distance（tpd a,tpd b）
{  /*给出两点坐标，计算距离*/  }
void sortarr（tdr a[maxm],int m）
{  /*将已经计算好的距离关系表按距离从小到大排序，形成边排序表，m为边的条数*/  }
int iscircuit（tr r[maxm],int i,int j）
{  /*判断边（i, j）选入端点关系表r[maxm]后，是否形成回路*/  }
void selected（tr r[maxm],int i,int j）
{  /*将边（i, j）选入端点关系表r[maxm]*/  }
void course（tr r[maxm],tl l[maxm]）
{  /*从端点关系表r得出回路轨迹表*/   }
void exchange（tdr a[maxm],int m,int b）
{  /*调整边排序表，b表示是否可调，即是否有边长度相同的边存在*/  }
void travling（tpd pd[maxm],int n,float dist,tl locus[maxm]）
{ tdr dr[maxm];              /*距离关系表*/
tr r[maxm];                 /*端点关系表*/
int i,j,k,h,m;
int b;                    /*b标识是否有长度相等的边*/
k=0;                     /*k为边的条数计数器*/
```

```
for（i=1;i<n;i++）          /*以下双重循环计算距离关系表中各边长度*/
for（j=i+1;j<=n;j++）
{k++;
dr[k].x=distance（pd[i],pd[j]）；
dr[k].p1=i; dr[k].p2=j;   /*形成距离关系表*/
}
m=k; sortarr（dr,m）；          /*按距离大小给距离关系表排序生成边排序表*/
do{
b=1;dist=0;             /*dist为总路程计数器*/
k=h=0;       /*k为边排序表中边的序号；h为选入端点关系表中的边数*/
do{
k++;       /*k初始为1,即开始时加入边排序表中第一条边，以后依次加入其余边*/
i=dr[k].p1;j=dr[k].p2;
if（（r[i].n<=1）&&（r[j].n<=1））/*端点度数不能大于2*/
/* 若边（i,j）加入端点关系表r后形成回路，则该边不能加入r */
if（！iscircuit（r,i,j））{             selected（r,i,j）；      /*边（i,j）加入到端点关系表r*/
h++;
dist+=dr[k].x;}
else if（h==n-1）{      /*若最后一边选入r成回路，则该边必须加入r,且得到解*/
selected（r,i,j）；
h++;
dist+=dr[k].x ;}
}while（（k!=m）&&（h!=n））；
if（h==n）              /*最后一边选入r成回路，完成结果输出*/
course（r,locus）；
else
/*找不到解，调整dr,交换表中边长相同的边在表中的顺序，并将b置0*/
exchange（dr,m,b）；
}while（！b）；
}
```

进一步细化前面各个子程序如下。

将边（i,j）选入r中的子程序selected（）：

```
void selected（tr r[maxm],int i,int j）
{
r[i].n++;r[j].n++;
if（r[i].n==1） r[i].p1=j; /*规定和端点联系的第一个端点为p1,第二个端点p2*/
else r[i].p2=j;
if（r[j].n==1） r[j].p1=i
else r[j].p2=i
```

```
}
```

判断边（i,j）选入r后是否构成回路的子程序iscircuit（ ）：当将边（i,j）选入r后，若去掉r中度数为1的端点，剩下的为2的度数项多于3个，我们就说其构成了回路。

```
int iscircuit（tr r[maxm],int i,int j）
{int k;
selected（r,i,j）；              /*注意r是iscircuit的局部量*/
k=1;
while（k<=n）{
if（r[k].n= =1）
{
i=r[k].p1; r[k].n=0;
if（r[i].n= =2）
if（r[i].p1= =k） r[i].p1=r[i].p2; /*保证和某一点相联系的端点只有一个时用p1存放*/
r[i].n--;}
if（i<k） k=i-1;      /*保证回路以外的度数为2的端点也要去掉*/
k++
}
i=0;
for（k=1;k<=n;k++）
if（r[k].n= =2） i++;
if（i>=3）return（1）；
else return（0）；
}
```

其他的4个函数（distance、sortarr、course、exchange）较简单，请读者自行细化。

上述各表中所列数据是最优解情况时的各项数据。贪婪法并不是刻意求最优解，其能较快速地得到令人满意的最优解的近似解。通过贪婪算法思想所得的回路轨迹实际应为A→F→E→D→C→B→A,也即1→6→5→4→3→2→1,这进一步体现了贪婪法不求最优解，但求快速满意解的思想。回路选择边的过程如图24-9所示。

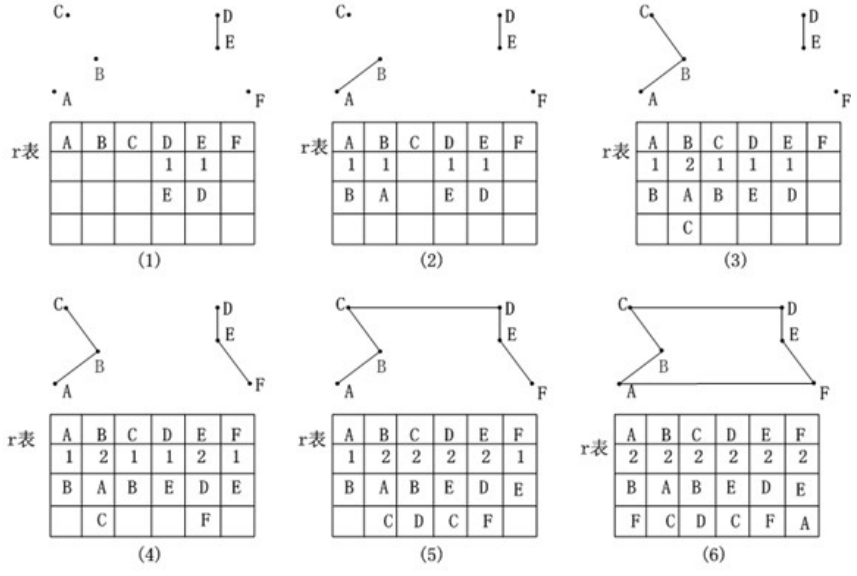


图24-9 回路中边的选择过程图

版权方授权希赛网发布，侵权必究

 作者：希赛教育软考学院 来源：希赛网 2014年03月13日

哈夫曼编码问题

24.6.5 哈夫曼编码问题

问题描述：在数据通信中，一般需要将传送的文字转换成由二进制字符0、1组成的二进制串，称为编码。例如，假设要传送的电文为AABCADC,电文中只含有A、B、C、D 4种字符，若这4种字符采用表24-6（a）所示的编码，则电文的代码为00000001010000011110O,长度为21.在传送电文时，我们总是希望传送时间尽可能短，这就要求电文代码尽可能短，显然，这种编码方案产生的电文代码不够短。表24-6（b）所示为另一种编码方案，用此编码对上述电文进行编码所建立的代码为00000110001110,长度为14.在这种编码方案中，4种字符的编码均为两位，是一种等长编码。如果在编码时考虑字符出现的频率，让出现频率高的字符采用尽可能短的编码，出现频率低的字符采用稍长的编码，构造一种不等长编码，则电文的代码就可能更短。如当字符A、B、C、D采用表24-6（c）所示的编码时，上述电文的代码为0011010011110,长度仅为13.构造一种编码方案，使得电文的编码总长度最短。

字符	编码
A	000
B	010
C	100
D	111

（a）

字符	编码
A	00
B	01
C	10
D	11

（b）

字符	编码
A	0
B	110
C	10
D	111

（c）

字符	编码
A	01
B	010
C	001
D	10

（d）

表24-6 字符的4种不同的编码方案

我们采用哈夫曼编码方案，即应用哈夫曼树构造使电文的编码总长最短的编码方案。假设有n个权值$\{w_1,w_2,\ldots,w_n\}$,构造有n个叶子结点的二叉树，每个叶子结点带权为wi,则二叉树带权路径长度WPL为树中所有叶子结点到树根之间的路径长度与结点上权的乘积之和，WPL最小的二叉树为哈夫曼树。应用哈夫曼树编码的方法如下：设需要编码的字符集合为$\{d_1,d_2,\ldots,d_n\}$,它们在电文中出现的次数集合相应为$\{w_1,w_2,\ldots,w_n\}$,以$d_1,d_2,\ldots,d_n$作为叶结点，$w_1,w_2,\ldots,w_n$作为它们的权值，构造一棵哈夫曼树，规定哈夫曼树中的左分支代表0,右分支代表1,则从根结点到每个叶结点所经过的路径分支组成的0和1的序列便为该结点对应字符的编码，称为哈夫曼编码。

在哈夫曼编码树中，树的带权路径长度WPL含义是各个字符的码长与其出现次数的乘积之和，也就是电文的代码总长，所以采用哈夫曼树构造的编码是一种能使电文代码总长最短的不等长编码。

此外，在建立不等长编码时，必须使任何一个字符的编码都不是另一个字符编码的前缀，这样才能保证译码的唯一性。例如表24-6（d）的编码方案，字符A的编码01是字符B的编码010的前缀部分，这样对于代码串0101001,既是AAC的代码，也是ABD和BDA的代码，因此，这样的编码不能保证译码的唯一性，称为具有二义性的译码。

然而，采用哈夫曼树进行编码，则不会产生上述二义性问题。因为，在哈夫曼树中，每个字符结点都是叶结点，它们不可能在根结点到其他字符结点的路径上，所以一个字符的哈夫曼编码不可能是另一个字符的哈夫曼编码的前缀，从而保证了译码的非二义性。

算法简单描述如下。

这是一种构造最优无前缀码的贪婪算法，用于求解某个字符串的哈夫曼编码，分为以下两步。

1.构造哈夫曼树

构造哈夫曼树的的算法为哈夫曼算法，其过程如下。

①根据给定的n个权值{w_1,w_2,…，w_n}构造含n棵二叉树的集合F={T_1,T_2,…，T_n},其中，T_i只有一个带权为w_i的根结点，其左右子树为空。

②以Ti为子树逐步合并形成哈夫曼树。根据贪婪准则，在F中选取两棵根结点权值最小的树作为左右子树形成一个新二叉树，且新二叉树根结点权值为其左右子树根结点的权值之和。同时在F中用新二叉树替代它的左右子树。

③重复上述步骤，直到F只含有一棵树为止。这棵树即为哈夫曼树。

我们可以设置一个结构数组HuffNode保存哈夫曼树中各结点的信息。根据二叉树的性质可知，具有n个叶子结点的哈夫曼树共有2n-1个结点，所以数组HuffNode的大小设置为2n-1,数组元素的结构形式如下：

weight	lchild	rchild	parent

其中，weight域保存结点的权值，lchild和rchild域分别保存该结点的左、右孩子结点在数组HuffNode中的序号，从而建立起结点之间的关系。为了判定一个结点是否已加入到要建立的哈夫曼树中，可通过parent域的值来确定。初始时parent的值为-1,当结点加入到树中时，该结点parent的值为其双亲结点在数组HuffNode中的序号，就不会是-1了。

2.在哈夫曼树上求叶结点的编码

该过程实质上就是在已建立的哈夫曼树中，从叶结点开始，沿结点的双亲链域回退到根结点，每回退一步，就走过了哈夫曼树的一个分支，从而得到一位哈夫曼码值。由于一个字符的哈夫曼编码是从根结点到相应叶结点所经过的路径上各分支所组成的0、1序列，因此先得到的分支代码为所求编码的低位码，后得到的分支代码为所求编码的高位码。我们可以设置一个结构数组HuffCode用来存放各字符的哈夫曼编码信息，数组元素的结构如下：

bit	start

其中，分量bit为一维数组，用来保存字符的哈夫曼编码，start表示该编码在数组bit中的开始位置。所以，对于第i个字符，它的哈夫曼编码存放在HuffCode[i].bit中的从HuffCode[i].start到n的分量上。

该问题算法程序实现见程序24-17.

【程序24-17】

```
#define MAXBIT 10                    /*定义哈夫曼编码的最大长度*/
#define MAXVALUE 10000               /*定义最大权值*/
#define MAXLEAF 30                   /*定义哈夫曼树中最多叶子结点个数*/
#define MAXNODE  MAXLEAF*2-1     /*哈夫曼树最多结点数*/
typedef struct {                     /*哈夫曼编码信息的结构*/
```

```
int bit[MAXBIT];
int start;
}HCodeType;
typedef struct {                    /*哈夫曼树结点的结构*/
int weight;
int parent;
int lchild;
int rchild;
}HNodeType;
void HuffmanTree（HNodeType HuffNode[MAXNODE],int n）    /*构造哈夫曼树的函数
*/
{
int i,j,m1,m2,x1,x2;
for（i=0;i<2*n-1;i++）          /*存放哈夫曼树结点的数组HuffNode[ ]初始化*/
{   HuffNode[i].weight=0;
HuffNode[i].parent=-1;
HuffNode[i].lchild=-1;
HuffNode[i].rchild=-1;
}
for（i=0;i<n;i++）              /*输入n个叶子结点的权值*/
{
printf（"please input %d character's weight\n",i）；
scanf（"%d",&HuffNode[i].weight）；
}
for（i=0;i<n-1;i++）            /*该循环开始构造哈夫曼树*/
{ m1=m2=MAXVALUE;
x1=x2=0;
for（j=0;j<n+i;j++）
{ if（HuffNode[j].weight<m1&&HuffNode[j].parent= =-1）
{ m2=m1; x2=x1;m1=HuffNode[j].weight;  x1=j;}
else if（HuffNode[j].weight<m2&&HuffNode[j].parent= =-1）
{ m2=HuffNode[j].weight; x2=j; }
}
HuffNode[x1].parent=n+i;     /*以下代码将找出的两棵子树合并为一棵子树*/
HuffNode[x2].parent=n+i;
HuffNode[n+i].weight=HuffNode[x1].weight+HuffNode[x2].weight;
HuffNode[n+i].lchild=x1;
HuffNode[n+i].rchild=x2;
}
```

```
}
void main ( )
{
HNodeType HuffNode[MAXNODE];
HCodeType HuffCode[MAXLEAF],cd;
int i,j,c,p,n;
printf ( "please input n:\n" ) ;
scanf ( "%d",&n ) ;                    /*输入叶子结点个数*/
HuffmanTree ( HuffNode,n ) ;          /*建立哈夫曼树 */
for ( i=0;i<n;i++ )        /*该循环求每个叶子结点对应字符的哈夫曼编码*/
{  cd.start=n-1;   c=i;
p=HuffNode[c].parent;
while ( p!=-1 )                  /*由叶结点向上直到树根*/
{ if ( HuffNode[p].lchild= =c ) cd.bit[cd.start]=0;
else  cd.bit[cd.start]=1;
cd.start--;  c=p;
p=HuffNode[c].parent;
}
/*保存求出的每个叶结点的哈夫曼编码和编码的起始位*/
for （ j=cd.start+1;j<n;j++ ）
HuffCode[i].bit[j]=cd.bit[j];
HuffCode[i].start=cd.start;
}
for ( i=0;i<n;i++ )                  /*输出每个叶子结点的哈夫曼编码*/
{   printf ( "%d character is: ",i ) ;
for ( j=HuffCode[i].start+1;j<n;j++ )
printf ( "%d",HuffCode[i].bit[j] ) ;
printf ( "\n" ) ;
}
}
```

构造哈夫曼树时，首先将由n个字符形成的n个叶结点存放到数组HuffNode的前n个分量中，然后根据前面介绍的哈夫曼方法的基本思想，不断将两棵小子树合并为一个较大的子树，每次构成的新子树的根结点顺序放到HuffNode数组中的前n个分量的后面。电文AABCADC中字符A、B、C、D的哈夫曼编码过程如图24-10所示，图中结点圆内的数字为结点权值，叶结点权值为相应字符在电文中出现的次数。

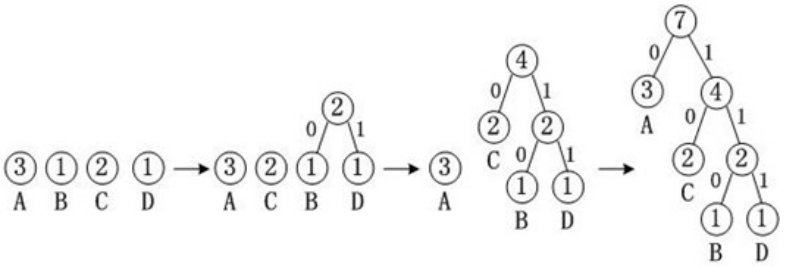


图24-10 哈夫曼编码过程图

版权方授权希赛网发布，侵权必究

第 24 章：常用算法设计 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

回溯法

24.7 回溯法

回溯法是一种选优搜索法，按选优条件向前搜索，以达到目标。但当搜索到某一步时，发现原先选择并不优或达不到目标，就退回一步重新选择。这种走不通就退回再走的技术就是回溯法，而满足回溯条件的某个状态的点称为"回溯点".

可用回溯法求解的问题P,通常要能表达为：对于已知的由n元组（$x_1,x_2,\ldots,x_n$）组成的一个解空间E={（$x_1,x_2,\ldots,x_n$）|xi∈S ,i=1,2,…，n},给定关于n元组中分量的一个约束集D,问题P需要求出E中满足D的所有n元组，其中S是分量x_i的定义域，且 |S| 有限，i=1,2,…，n.我们称E中满足D的任一n元组为问题P的一个解。

解问题P的最朴素的方法就是穷举法，即对E中的所有n元组逐一地检测其是否满足D的全部约束，若满足，则为问题P的一个解，但显然，其计算量是相当大的。

我们发现，对于许多问题，只要存在0≤j≤n-1,使得（$x_1,x_2,\ldots,x_j$）违反D的约束，则以（$x_1,x_2,\ldots,x_j$）为前缀的任何n元组（$x_1,x_2,\ldots,x_j,x_{j+1},\ldots,x_n$）一定也违反D的约束，n≥i>j。因此，可以肯定，一旦检测断定某个j元组（$x_1,x_2,\ldots,x_j$）违反D的约束，就可以肯定，以（$x_1,x_2,\ldots,x_j$）为前缀的任何n元组（$x_1,x_2,\ldots,x_j,x_{j+1},\ldots,x_n$）都不会是问题P的解，因而就不必去搜索它们、检测它们。回溯法正是针对这类问题，利用这类问题的上述性质而提出来的比穷举法效率更高的算法。

回溯法首先将问题P的n元组的解空间E表示成一棵高为n的带权有序树T（称为解空间树），把在E中求问题P的所有解转化为在T中搜索问题P的所有解。例1说明了T的建立和利用T求解的过程。

【例1】n=5，r=3的所有组合为：

（1）1、2、3 （6）1、4、5

（2）1、2、4 （7）2、3、4

（3）1、2、5 （8）2、3、5

（4）1、3、4 （9）2、4、5

（5）1、3、5 （10）3、4、5

则该问题的解空间为：

E={（X1,X2,X3）|Xi∈S,i=1,2,3 },其中Xi的定义域为：S={1,2,3,4,5},约束集D为：X1<X2<X3.

则建立的问题解空间树T如图24-14所示。

如图24-11所示，组合问题解空间树的每层路径表示解空间{（X1、X2、X3）}的一个分量，路径的权表示该分量的所有取值。求解从T的根结点出发，按深度优先的策略，系统地搜索以该结点为根的子树中可能包含着解的所有状态结点，而跳过对肯定不含解的所有子树的搜索，以提高搜索效率。在组合问题中，从T的根出发深度优先遍历该树。当遍历到结点（1,1）时，它不满足约束条件D,则遍历跳过该结点的所有子树，回溯至该结点的父结点，遍历该父结点的下一个子树；当遍历到结点（1,2）时，虽然它满足约束条件，但还不是解结点，则应继续深度遍历；当遍历到叶子结点（1,2,1）时，它不满足约束条件D,则同样需要回溯；当遍历到叶子结点（1,2,3）时，由于它已是一个解结点，则保存（或输出）该结点，并需要回溯到其父结点，继续深度遍历该父结点的下一个子树；当遍历到结点（1,5）时，由于它不是叶子结点，但不满足约束条件，故也需回溯。按同样方法依次遍历完整棵解空间树，就能找到所有的解结点，输出。

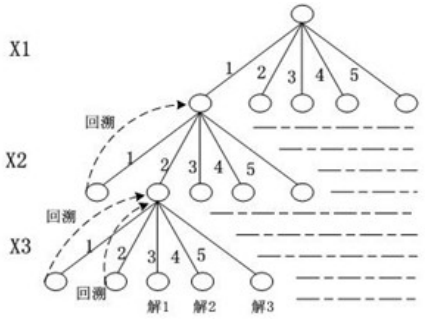


图24-11 组合问题的状态空间树T示意图

由上可以看出，回溯法首先放弃关于规模大小的限制，并将问题的候选解按某种顺序逐一试探，故其又称为试探法。试探过程中发现当前候选解不可能是解，且该规模下还有其他可选候选解时，就顺序试探下一个候选解；试探过程中发现当前候选解不可能是解，且该规模下没有其他可选候选解时，就缩小规模，试探该规模的下一个候选解；如果当前候选解除了不满足问题规模之外，满足其他要求，则扩大规模，继续试探；如果当前候选解满足包括问题规模在内的所有要求时，则该候选解就是问题的一个解。

对于回溯法，我们要搞清楚回溯的条件规则。该算法由两部分组成，如下所示：

试探部分：满足除规模之外的所有条件，则扩大规模。
（扩大规模）

回溯部分：
（缩小规模）
1.当前规模解不是合法解时回溯（不满足约束条件 D）。
2.求完一个解，要求下一个解时，也要回溯。

下面介绍一些应用回溯法的典型问题。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

作者：希赛教育软考学院　来源：希赛网　2014年01月27日

组合问题

24.7.1 组合问题

问题描述：找出从自然数1,2,…，n中任取r个数的所有组合。

采用回溯法找问题的解，将找到的组合以从小到大的顺序存于a[0],a[1],…，a[r-1]中，组合的元素满足以下性质：

la[i+1]>a[i],后一个数字比前一个大。

la[i]-i<=n-r+1.

算法简单描述如下。

按回溯法的思想，由以下步骤得到问题的解。

（1）首先设置a[0]=1,此时候选解的规模为1,候选解为1,且满足除问题规模之外的全部条件，则下一步应扩大规模，考虑a[1]的赋值。

（2）设置a[1]= a[0]+1=2,此时候选解规模为2,候选解为1、2,且仍不满足问题的规模，则继续扩大规模；设置a[2]= a[1]+1=3,此时候选解规模为3,候选解为1、2、3,该候选解满足包括问题规模在内的全部条件，因而是一个解，输出。

（3）再考虑该规模3下是否还有其他解。选下一个候选解，因此令a[2]加1调整为4,以及以后再加1调整为5都满足问题的全部要求，得到解1、2、4和1、2、5,输出。

（4）规模3情况下的候选解已经考查完，下一步应该回溯考虑规模2情况下的下一个候选解。则令a[1]加1为3,候选解为1、3,此时规模不满足，继续扩大规模，设置a[2]= a[1]+1=4,此时候选解规模为3,候选解为1、3、4,该候选解满足包括问题规模在内的全部条件，因此为一个解，输出。重复上述向前试探和向后回溯，直至要从a[0]再回溯时，说明已经找完问题的全部解。由于数组a的元素始终按递增顺序增加，故其始终满足上述第一条条件。

该问题算法实现见程序24-18.

【程序24-18】

```
# defineMAXN100
int a[MAXN];
void comb（int m,int r）  /*求从自然数1到m中任取3个数的所有组合子程序*/
{int i,j;
i=0; a[i]=1;       /*初始规模为1时，a[0]为1*/
do {
if （a[i]-i<=m-r+1）/*还可以向前试探*/
{if （i= =r-1）/*当前候选解的规模满足问题的规模要求，找到一个解*/
{for （j=0;j<r;j++）
printf（"%4d",a[j]）；
printf（"\n"）；
}
a[i]++;    /*考查当前规模的下一个候选解*/
continue;
}
i++;
a[i]=a[i-1]+1;
}
```

```
else            /*当前规模的候选解已经全部考查完，则应回溯，缩小规模*/
{if （i= =0）  /*回溯至初始规模，则已经全部找到了解*/
return;
a[--i]++;  /*缩小规模，考查下一个候选解*/
}
}while （1）
}
main（ ）
{comb（5,3）；}
```

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

子集和问题

24.7.2 子集和问题

问题描述：给定由n个不同正数组成的集合W={w1,w2,…，wn}和正数M,要求找出N={1,2,…，n}的所有使得 $\sum_{i\in s} w_i = M$ 的子集S.例如，给定n=4,W={11,13,24,7}和M=31,则相应的子集和问题的解是{3,4}和{1,2,4}.

这个问题还可表述为：求所有使得 $\sum_{i\in s} w_i x_i = M$ 的n元组（x1,x2,…，xn），其中xi∈{0,1},1≤i≤n,以及得到的n元组相对应的原问题的解S={i|xi=1,i∈{1,2,…，n}}.解向量的元素xi或者为0或者为1,这取决于子集中是否包含了wi。

根据题意，我们知道，若条件

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geqslant M$$

不成立，则（x1,x2,…，xk）就不可能成为解的一部分。此外，如果假定wi按升序排列，如果条件

$$\sum_{i=1}^{k} w_i x_i + w_{k+1} \leqslant M$$

不成立，则（x1,x2,…，xk）也不可能成为解的一部分。故要想使得（x1,x2,…，xk）有可能成为解的一部分，则必须满足以下两个条件：

$$\sum_{i=1}^{k} w_i x_i + \sum_{i=k+1}^{n} w_i \geqslant M \quad 和 \quad \sum_{i=1}^{k} w_i x_i + w_{k+1} \leqslant M$$

由上面我们可以看到，子集和问题的解空间树T是一棵高度为n的二叉树，其中深度为k的一个状态结点对应于一个k元组（x1,x2,…，xk）。我们可以约定（x1,x2,…，xk-1,1）和（x1,x2,…，xk-1,0）所对应的状态结点分别是（x1,x2,…，xk-1）所对应的状态结点的左儿子和右儿子。初始时，我们要求输入W时，从小到大输入，使得数组W中的元素有序。

该问题算法实现见程序24-19.

【程序24-19】

```
#include <stdio.h>
#define MAX 100
int w[MAX],x[MAX],m,n;
void sumofsub ( float s,int k,float r )
{               /*求子集和函数，s和r表示如程序附后说明*/
int i;
x[k]=1;                    /*试探x[k]包含在解向量中的情况*/
if ( s+w[k]= =m )              /*找到一个解，输出*/
{
for ( i=1;i<=k;i++ ) printf ( "%4d",x[i] ) ;
printf ( "\n" ) ;
}
else{
if ( s+w[k]+w[k+1]<=m )        /* x[k]包含在解向量中是否可行 */
sumofsub ( s+w[k],k+1,r-w[k] ) ; /* 可行则扩大规模*/
if ( s+r-w[k]>=m&&s+w[k+1]<=m ){/*试探x[k]不包含在解向量中的情况是否可行 */
x[k]=0;                    /* 可行则扩大规模*/
sumofsub ( s,k+1,r-w[k] ) ;      /* 可行则扩大规模*/
}
}
}
void main ( )
{ int i,k;
float r,s;
r=s=0;
k=1;
for ( i=0;i<=MAX;i++ ) x[i]=0;
scanf ( "%4d,%4d",&n,&m ) ;
printf ( "please input values of array w by ascend:\n" ) ;
for ( i=1;i<=n;i++ ) scanf ( "%4d",&w[i] ) ; /* 按升序输入w数组的值*/
for ( i=1;i<=n;i++ )
r+=w[i];                    /* 计算r的初始值，为w数组所有元素的和*/
sumofsub ( s,k,r ) ;                /* 递归求所有全部解*/
}
```

程序中 $\sum_{i=1}^{k} w_i x_i$ 和 $\sum_{i=k+1}^{n} w_i x_i$ 分别保存在变量s和r中。该算法没有明显地使用测试条件k>n去终止递归，其原因在于过程每次调用开始时s≠M,s+r≥M,因此，r≠0,从而k也不可能大于n.而且如果s+w[k]=M,则x[k+1],x[k+2],…，x[n]应该为0,这些0不包含在解的输出中。而且，程序假定x[1]≤M,w[1]+x[2]+…+x[k]≥M.图24-12是n=4,M=31,W={7,11,13,24}时的解空间树。从树中可以

看到，该问题的解为（1,1,1）和（1,0,0,1）。

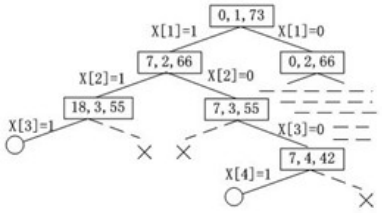


图24-12 子集和问题解空间树

版权方授权希赛网发布，侵权必究

 作者：希赛教育软考学院 来源：希赛网 2014年01月27日

八皇后问题

24.7.3 八皇后问题

问题描述：求出在一个n×n的棋盘上，放置n个不能互相捕捉的国际象棋"皇后"的所有布局。

这是来源于国际象棋的一个问题。皇后可以沿着纵、横和两条斜线4个方向相互捕捉。如图24-13所示，一个皇后放在棋盘的第4行第3列位置上，则棋盘上凡打"×"的位置上的皇后就能与这个皇后相互捕捉。

1	2	3	4	5	6	7	8
		×			×		
×		×		×			
	×	×	×				
×	×	Q	×	×	×	×	×
	×	×	×				
×		×		×			
		×			×		
		×				×	

图24-13 皇后互相捕捉示意图

从图24-13中可以得到以下启示：一个合适的解应是在每列、每行上只有一个皇后，且一条斜线上也只有一个皇后。

求解过程从空配置开始。在第1列至第m列为合理配置的基础上，再配置第m+1列，直至第n列配置也是合理时，就找到了一个解。接着改变第n列配置，希望获得下一个解。另外，在任一列上，可能有n种配置。开始时配置在第1行，以后改变时，顺次选择第2行、第3行……直到第n行。当第n行配置也找不到一个合理的配置时，就要回溯，去改变前一列的配置。

比较直观的方法是采用一个二维数组，但仔细观察就会发现，这种表示方法给调整候选解及检查其合理性带来困难。更好的方法乃是尽可能直接表示那些常用的信息。对于本题来说，"常用信息"并不是皇后的具体位置，而是"一个皇后是否已经在某行和某条斜线合理地安置好了".因在某一列上恰好放一个皇后，引入一个一维数组（col[ ]），值col[i]表示在棋盘第i列、col[i]行有一个皇后。

例如，col[3]=4,就表示在棋盘的第3列、第4行上有一个皇后。另外，为了使程序在找完了全部解后回溯到最初位置，设定col[0]的初值为0,当回溯到第0列时，说明程序已求得全部解，结束程序运行。

为使程序在检查皇后配置的合理性方面简易方便，引入以下3个工作数组：

数组a[ ],a[k]表示第k行上还没有皇后。

数组b[ ],b[k]表示第k列右高左低斜线上没有皇后。

数组c[ ],c[k]表示第k列左高右低斜线上没有皇后。

棋盘中同一右高左低斜线上的方格，它们的行号与列号之和相同；同一左高右低斜线上的方格，它们的行号与列号之差均相同。

初始时，所有行和斜线上均没有皇后，从第1列的第1行配置第一个皇后开始，在第m列col[m]行放置了一个合理的皇后后，准备考查第m+1列时，在数组a[ ]、b[ ]和c[ ]中为第m列，col[m]行的位置设定有皇后标志；当从第m列回溯到第m-1列，并准备调整第m-1列的皇后配置时，清除在数组a[ ]、b[ ]和c[ ]中设置的关于第m-1列，col[m-1]行有皇后的标志。一个皇后在m列，col[m]行方格内配置是合理的，由数组a[ ]、b[ ]和c[ ]对应位置的值都为1来确定。由于每次只在某列配置一个皇后，故不可能有两个皇后在同一列的情况。

我们将棋盘的列数m称为问题的规模，配置某列m的皇后位置称为在规模m下设置皇后的位置。

该问题算法程序实现见程序24-20.

【程序24-20】

```
# include<stdio.h>
# include<stdlib.h>
# defineMAXN20
int n,m,good;
int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];
void main（ ）
{int j;
char awn;
printf（"Enter n: "）；scanf（"%d",&n）；
for（j=0;j<=n;j++）a[j]=1;          /*初始时每列的各行都能放置皇后*/
for（j=0;j<=2*n;j++）b[j]=c[j]=1;  /*初始时各个斜角也都能放置皇后*/
m=1;col[1]=1;good=1;col[0]=0;
do {                          /*循环找解*/
if（good）
if（m= =n）              /*找到一个解*/
{printf（"列\t行"）；
for（j=1;j<=n;j++）
printf（"%3d\t%d\n",j,col[j]）； /*输出每列放置皇后的位置*/
printf（"Enter a character（Q/q for exit）！\n"）； /*是否找下一个解*/
scanf（"%c",&awn）；
if（awn= ='Q'||awn= ='q'）exit（0）；
```

```
while （ col[m]= =n ）    /*问题当前规模m是否还有其他解*/
{m--;          /*当前规模m无其他解则回溯，缩小规模m*/
a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1;  /*回溯后清除皇后标志*/
}
col[m]++;          /*试探当前规模的下一个解*/
}
else
/*在m列、col[m]行位置设置皇后标志*/
{ a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=0;
/*扩大规模，试探该规模下的第一个可能解，即皇后从第一行开始配置*/
col[++m]=1;
}
else   /*good为0时的情况，即当前规模下试探解失败，需回溯或调整皇后位置*/
{while （ col[m]= =n ） /*是否当前规模下所有可能解试探都失败*/
{m--;       /*当前规模下所有可能解试探都失败，需回溯*/
a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1; /*回溯后清除皇后标志*/
}
col[m]++;    /*当前规模下可能还有其他解，调整当前规模下的皇后位置*/
}
/*当前规模m下放置的皇后是否合法*/
good=a[col[m]]&&b[m+col[m]]&&c[n+m-col[m]];
} while （ m!=0 ）；
}
```

八皇后问题是典型的回溯法求解的问题，程序实现的关键点为：

试探条件good=a[col[m]]&&b[m+col[m]]&&c[n+m-col[m]].

当满足条件，不满足规模时，扩大规模，col[++m]=1.

当不满足条件时，则判断当前规模下是否还有其他可能解while（m= =n），有则试探下一个可能解col[m]++;没有则缩小规模回溯后试探下一个可能解m--、col[m]++.

该算法程序充分体现了回溯法的思想，回溯各部分说明如下：

```
do {
    if (good)
        if (m= =n)
        {
            ……
            while (col[m]= =n)
                {   m--;
                  a[col[m]]=b…=1;
                }
            col[m]++;
        }
        else
            {  a[col[m]]=b…=0;
               col[++m]=1;
            }
    else
        {   while (col[m]= =n)
            {   m--;
                a[col[m]]=b…=1;
            }
            col[m]++;
        }
```

满足条件和规模，输出解。若要求下一个解时就回溯

满足条件，但不满足规模，扩大规模

满足条件

不满足条件，回溯

good=a[col[m]]&&b…；

} while （ m!=0 ）；

}

运行程序，得到八皇后问题的一个解，如图24-14所示。

1	2	3	4	5	6	7	8
				×			
×							
							×
					×		
		×					
						×	
	×						
			×				

图24-14 八皇后问题的一个解

版权方授权希赛网发布，侵权必究

上一节 本书简介 下一节

作者：希赛教育软考学院 来源：希赛网 2014年01月27日

迷宫问题

24.7.4 迷宫问题

问题描述：这是实验心理学中的一个经典问题。心理学家把一只老鼠从一个无顶盖的大盒子的入口处赶进迷宫。迷宫中设置很多隔壁，对前进方向形成了多处障碍，心理学家在迷宫的唯一出口处放置了一块奶酪，吸引老鼠在迷宫中寻找通路以到达出口。

求解过程采用回溯法。从入口出发，按某一方向向前探索，若能走通且未走过的，即某处可以到达，则到达新点，否则试探下一方向；若所有的方向均没有通路，则沿原路返回前一点，换下一个方向再继续试探，直到所有可能的通路都探索到，或找到一条通路，或无路可走又返回到入口

点。

在求解过程中，为了保证在到达某一点后不能向前继续行走（无路）时，能正确返回前一点以便继续从下一个方向向前试探，则需要用一个栈保存所能够到达的每一点的下标及从该点前进的方向。

1.表示迷宫的数据结构

设迷宫为m行n列，利用数组maze[m][n]来表示一个迷宫。maze[i][j]=0或1.其中0表示通路，1表示不通。当从某点向下试探时，中间的点有8个方向可以试探（见图24-14），而4个角点只有3个方向，而其他边缘点有5个方向。为使问题简单化，我们用maze[m+2][n+2]来表示迷宫，而迷宫的四周的值全部为1.这样做使问题简单了，每个点的试探方向全部为8,不用再判断当前点的试探方向有几个，同时与迷宫周围是墙壁这一实际问题相一致。

如图24-15所示的迷宫是一个6×8的迷宫。入口坐标为（1,1），出口坐标为（6,8）。

入口（1,1）

0123456789

01111111111

11011101111

21101011111

31010000011

41011101111

51100110001

61011001101

71111111111

出口（6,8）

	0	1	2	3	4	5	6	7	8	9
0	1	1	1	1	1	1	1	1	1	1
1	1	0	1	1	1	0	1	1	1	1
2	1	1	0	1	0	1	1	1	1	1
3	1	0	1	0	0	0	0	0	1	1
4	1	0	1	1	1	0	1	1	1	1
5	1	1	0	0	1	1	0	0	0	1
6	1	0	1	1	0	0	1	1	0	1
7	1	1	1	1	1	1	1	1	1	1

图 24-15 数组maze[m+2][n+2]表示的迷宫

迷宫的定义如下：

```
#define  m   6        /* 迷宫的实际行 */
#define  n   8        /* 迷宫的实际列 */
int maze [m+2][n+2] ;
```

2.试探方向

在上述表示迷宫的情况下，每个点有8个方向可以试探。如当前点的坐标为（x,y），与其相邻的8个点的坐标都可根据与该点的相邻方位而得到，如图24-15所示。因为出口在（m,n），因此试探顺序规定为：从当前位置向前试探的方向为从正东沿顺时针方向进行。为了简化问题，方便地求出新点的坐标，将从正东开始沿顺时针方向进行的这8个方向的坐标增量放在一个结构数组move [8]中，在move 数组中，每个元素由两个域组成，x为横坐标增量，y为纵坐标增量。move数组如图

24-16所示。

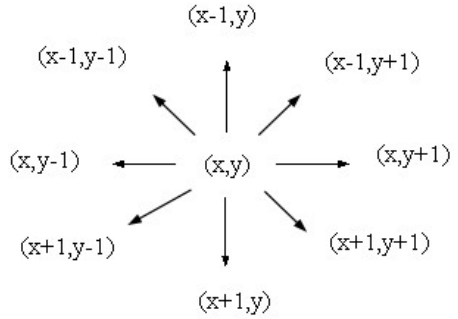


图24-16 点（x,y）相邻的8个点及坐标及增量数组move

move数组定义如下：

typedef struct

{

int x,y

} item ;

item move[8] ;

这样对move的设计会很方便地求出从某点（x,y）按某一方向v （0≤v≤7） 到达的新点（i,j）的坐标。

	x	Y
0	0	1
1	1	1
2	1	0
3	1	-1
4	0	-1
5	-1	-1
6	-1	0
7	-1	1

xY

001

111

210

31-1

40-1

5-1-1

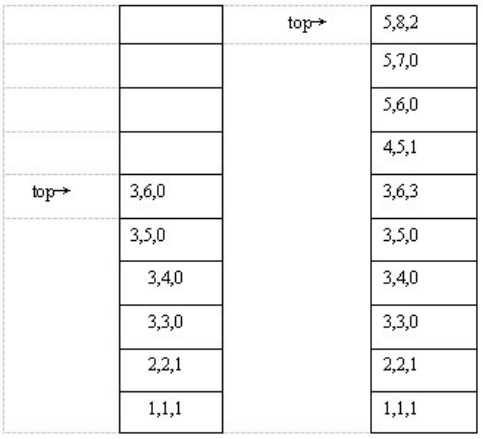
6-10

7-11

可知，试探点的坐标（i,j）可表示为i=x+move[v].x ;j=y+move[v].y.

3.栈的设计

到达了某点而无路可走时需返回前一点，再从前一点开始向下一个方向继续试探。因此，压入栈中的不仅是顺序到达的各点的坐标，而且还要有从前一点到达本点的方向。对于如图24-14所示迷宫，依次入栈如下：



top→5,8,2

5,7,0

5,6,0

4,5,1

top→3,6,03,6,3

3,5,03,5,0

3,4,03,4,0

3,3,03,3,0

2,2,12,2,1

1,1,11,1,1

栈中每一组数据是所到达的每点的坐标及从该点沿哪个方向向下走的，对于如图24-14所示的迷宫，走的路线为：（1,1）1→（2,2）1→（3,3）0→（3,4）0→（3,5）0→（3,6）0（下脚标表示方向）；当从点（3,6）沿方向0到达点（3,7）之后，无路可走，则应回溯，即退回到点（3,6），对应的操作是出栈，沿下一个方向即方向1继续试探；方向1、2试探失败，在方向3上试探成功，因此将（3,6,3）压入栈中，即到达了（4,5）点。

栈中元素是一个由行、列、方向组成的三元组，栈元素的设计如下：

```
typedef struct
{int x , y , d ;            /* 横坐标和纵坐标及方向*/
}datatype ;
```

栈的设计如下：

```
#define MAXSIZE  1024   /*栈的最大深度*/
typedef  struct
{datatype  data[MAXSIZE];
int  top;            /*栈顶指针*/
}SeqStack
```

4.如何防止重复到达某点，以避免发生死循环

一种方法是另外设置一个标志数组mark[m][n],它的所有元素都初始化为0,一旦到达了某一点（i,j）之后，使mark[i][j]置1,下次再试探这个位置时就不能再走了。另一种方法是当到达某点（i,j）后使maze[i][j]置-1,以便区别未到达过的点，同样也能起到防止走重复点的目的。本书采用后者方法，算法结束前可恢复原迷宫。

算法简单描述如下：

栈初始化；

将入口点坐标及到达该点的方向（设为-1）入栈

while（栈不空）{

栈顶元素=>（x，y，d）

出栈；

求出下一个要试探的方向d++;

while（还有剩余试探方向时）{

if （d方向可走）

then {（x，y，d）入栈；

求新点坐标（i, j）；

将新点（i，j）切换为当前点（x，y）；

if （（x,y）= =（m,n））结束；

else 重置 d=0；

}

else d++；

}

}

该问题算法程序实现见程序24-21.

【程序24-21】

```
#include <stdio.h>
#define m 6                    /* 迷宫的实际行 */
#define n 8                    /* 迷宫的实际列 */
#define MAXSIZE 1024           /* 栈的最大深度 */
int maze[m+2][n+2];            /* 迷宫数组，初始时为0 */
typedef struct item{           /* 坐标增量数组 */
int x,y;
}item;
item move[8];                  /* 方向数组 */
typedef struct datatype{       /* 栈结点数据结构 */
int x,y,d;                     /* 横坐标和纵坐标及方向 */
}datatype;
typedef struct SeqStack{       /* 栈结构 */
datatype data[MAXSIZE];
int top;                       /* 栈顶指针 */
}SeqStack;
SeqStack *s;
datatype temp;
int path（int maze[m+2][n+2],item move[8]）{
int x,y,d,i,j;
```

```
temp.x=1;temp.y=1;temp.d=-1;
Push_SeqStack（s,temp）；  /* 辅助变量temp表示当前位置，将其入栈 */
while（！Empty_SeqStack（s））
{ Pop_SeqStack（s,&temp）；/* 若栈非空，取栈顶元素送temp */
x=temp.x;y=temp.y;d=temp.d+1;
while（d<8）          /* 判断当前位置的8个方向是否为通路 */
{ i=x+move[d].x;j=y+move[d].y;
if（maze[i][j]= =0）
{ temp.x=x;temp.y=y;temp.d=d;
Push_SeqStack（s,temp）；
x=i;y=j;maze[x][y]=-1;
if（x= =m&&y= =n）return 1; /* 迷宫有路 */
else d=0;
}
else d++;
}                    /* while （d<8）*/
}                     /* while */
return 0 ;                  /* 迷宫无路 */
}
int Empty_SeqStack（SeqStack *s）     /* 判断栈空函数 */
{ if （s->top= =-1）return 1;
else return 0;
}
int Push_SeqStack（SeqStack *s, datatype x）  /* 入栈函数 */
{ if（s->top= =MAXSIZE-1）return 0;   /* 栈满不能入栈 */
else {s->top++;
s->data[s->top]=x;
return 1;
}
}
int Pop_SeqStack（SeqStack *s,datatype *x）    /* 出栈函数 */
{ if（Empty_SeqStack（s））return 0;      /* 栈空不能出栈 */
else{*x=s->data[s->top];
s->top--;return 1; }          /* 栈顶元素存入*x,返回 */
}
void main（）{
int i,j,t;
move[0].x=0;move[0].y=1;
move[1].x=1;move[1].y=1;
```

```
move[2].x=1;move[2].y=0;
move[3].x=1;move[3].y=-1;
move[4].x=0;move[4].y=-1;
move[5].x=-1;move[5].y=-1;
move[6].x=-1;move[6].y=0;
move[7].x=-1;move[7].y=1;
for ( i=0;i<=n+1;i++ ) {
maze[0][i]=1;
maze[m+1][i]=1;
}
for ( i=1;i<=m;i++ ) {
maze[i][0]=1;
maze[i][n+1]=1;
}
printf ( "please input maze\n" ) ;
for ( i=1;i<=m;i++ )
for ( j=1;j<=n;j++ ) scanf ( "%d",&maze[i][j] ) ;
t=path ( maze,move ) ;
if ( t= =1 ) {
printf ( "the track is :\n" ) ;
while ( ! Empty_SeqStack ( s ) )
{ Pop_SeqStack ( s,&temp ) ;            /*若栈非空，则打印输出 */
printf ( "%d,%d,%d\n",temp.x,temp.y,temp.d ) ;
}
}
}
```

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

分治法

24.8 分治法

对于一个规模为n的问题，若该问题可以容易地解决（比如说规模n较小）则直接解决；否则将其分解为k个规模较小的子问题，这些子问题互相独立且与原问题形式相同，递归地解这些子问题，然后将各子问题的解合并得到原问题的解。这种算法设计策略叫做分治法。

我们知道，任何一个可以用计算机求解的问题所需的计算时间都与其规模有关。问题的规模越小，越容易直接求解，解题所需的计算时间也越少。例如，对于n个元素的排序问题，当n=1时，不

需任何计算。n=2时，只要作一次比较即可排好序。n=3时只要做3次比较即可……而当n较大时，问题就不那么容易处理了。要想直接解决一个规模较大的问题，有时是相当困难的。而分治法的设计思想是，将一个难以直接解决的大问题，分割成一些规模较小的相同问题，以便各个击破，分而治之。如果原问题可分割成k个子问题，1<k≤n,且这些子问题都可解，并可利用这些子问题的解求出原问题的解，那么这种分治法就是可行的。由分治法产生的子问题往往是原问题的较小模式，这就为使用递归技术提供了方便。在这种情况下，反复应用分治手段，可以使子问题与原问题类型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。这自然导致递归过程的产生。因此，我们说分治与递归像一对孪生兄弟，经常同时应用在算法设计之中，并由此产生许多高效算法。

分治法所能解决的问题一般具有以下几个特征：

该问题的规模缩小到一定的程度就可以容易地解决。

该问题可以分解为若干个规模较小的相同问题，即该问题具有最优子结构性质。

利用该问题分解出的子问题的解可以合并为该问题的解。

该问题所分解出的各个子问题是相互独立的，即子问题之间不包含公共的子子问题。

上述的第一条特征是绝大多数问题都可以满足的，因为问题的计算复杂性一般是随着问题规模的增加而增加的。第二条特征是应用分治法的前提，它也是大多数问题可以满足的，此特征反映了递归思想的应用。第三条特征是关键，能否利用分治法完全取决于问题是否具有第三条特征，如果具备了第一条和第二条特征，而不具备第三条特征，则可以考虑贪心法或动态规划法。第四条特征涉及到分治法的效率，如果各子问题是不独立的，则分治法要做许多不必要的工作，重复地解公共的子问题。

分治法在每一层递归上都有3个步骤：

①分解：将原问题分解为若干个规模较小、相互独立、与原问题形式相同的子问题。

②解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题。

③合并：将各个子问题的解合并为原问题的解。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

二分法查找

24.8.1 二分法查找

在对线性表的操作中，经常需要查找某一个元素在线性表中的位置。此问题的输入是待查元素x和线性表L,输出为x在L中的位置或者x不在L中的信息。

比较自然的想法是一个一个地扫描L的所有元素，直到找到x为止。这种方法对于有n个元素的线性表在最坏情况下需要n次比较。一般来说，如果没有其他的附加信息，在有n个元素的线性表中查找一个元素在最坏情况下都需要n次比较。

下面我们考虑一种简单的情况。假设该线性表已经排好序了，不妨设它按照主键的递增顺序排列（即由小到大排列）。在这种情况下，我们是否有改进查找效率的可能呢？如果线性表里只有一

个元素，则只要比较这个元素和x就可以确定x是否在线性表中。因此这个问题满足分治法的第一个适用条件。同时我们注意到对于排好序的线性表L有以下性质：比较x和L中任意一个元素L[i],若x=L[i],则x在L中的位置就是i;如果x<L[i],由于L是递增排序的，因此假如x在L中的话，x必然排在L[i]的前面，所以我们只要在L[i]的前面查找x即可；如果x>L[i],同理我们只要在L[i]的后面查找x即可。无论是在L[i]的前面还是后面查找x,其方法都和在L中查找x一样，只不过是线性表的规模缩小了。这就说明了此问题满足分治法的第二个和第三个适用条件。很显然此问题分解出的子问题相互独立，即在L[i]的前面或后面查找x是独立的子问题，因此满足分治法的第四个适用条件。

于是我们得到利用分治法在有序表中查找元素的算法。其程序片断见程序24-22. 【程序24-22】

```
function Binary_Search（L,a,b,x）；
{ if（a>b）return（-1）；
else
{ m=（a+b）/2;
if（x= =L[m]）return（m）；
else if（x>L[m]）
return（Binary_Search（L,m+1,b,x））；　　/*递归实现*/
else return（Binary_Search（L,a,m-1,x））；　　/*递归实现*/
}
}
```

在以上算法中，L为排好序的线性表，x为需要查找的元素，b、a分别为x的位置的上下界，即如果x在L中，则x在L[ab]中。每次我们用L中间的元素L[m]与x比较，从而确定x的位置范围，然后递归地缩小x的范围，直到找到x。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

汉诺塔问题

24.8.2 汉诺塔问题

问题描述：设A、B、C是3个塔座。开始时，在塔座A上有一叠n个圆盘，这些圆盘自下而上，由大到小地叠在一起，各圆盘从大到小编号为1,2,…，n,如图24-17所示。

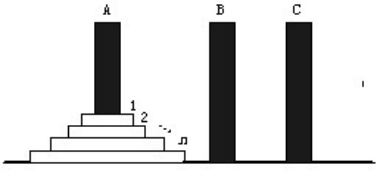


图24-17 汉诺塔问题的初始状态

现要求将塔座A上的这一叠圆盘移到塔座B上，并仍按同样顺序叠置。在移动时应遵守以下移动规则：

（1）每次只能移动一个圆盘。（2）任何时刻都不容许将较大的圆盘压在较小的圆盘之下。（3）在满足移动规则（1）和（2）的前提下，可将圆盘移至A、B、C中任意一个塔座上。

在遵守上述规则的情况下，借助C塔座，将n张圆盘从A塔座移到B塔座可以采用递归实现（如图24-18（a）所示）。递归方法较为简单，先将n-1张圆盘借助B塔座从A塔座移到C塔座（如图24-18（b）所示），然后将A塔座上第n张圆盘移到B塔座（如图24-18（c）所示），再将C塔座上的n-1张圆盘借助A塔座移到B塔座（如图24-18（d）所示），这样，A塔座上的n张圆盘就移到了B塔座上。这实际上是一种分治的方法，把整个n张圆盘的移动分为第n张圆盘的移动和其上面n-1张圆盘的移动。过程如图24-18所示。

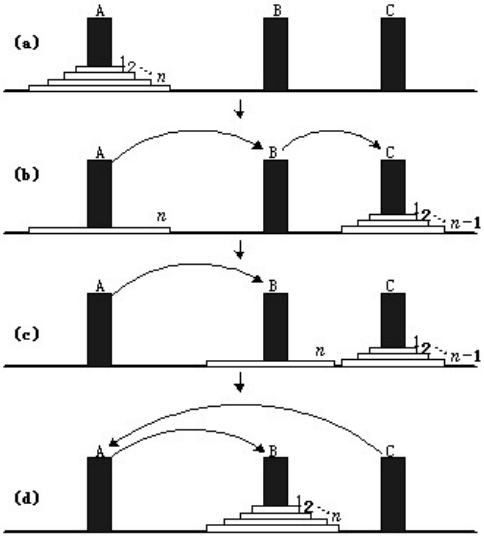


图24-18 汉诺塔问题圆盘移动过程示意图

该问题算法程序实现见程序24-23.

【程序24-23】

```
#include〈stdio.h〉
void move（int n,char getone,char putone） /*将一个圆盘从getone塔座移到putone塔座*/
{
printf（"no.%d plate %c-->%c\n",n,getone,putone）；
}
void Hanoi（int n,char A,char B,char C） /*借助塔座C,将n个圆盘从A塔座移到B塔座*/
{
if（n>0）{
Hanoi（n-1,A,C,B）； /*借助塔座B,将n-1个圆盘从A塔座移到C塔座*/
Move（n,A,B）； /*将一个圆盘从A塔座移到B塔座*
Hanoi（n-1,C,B,A）； /*借助塔座A,将n-1个圆盘从C塔座移到B塔座*/
}
}
main（）
{
int m;
printf（"input the number of disks:"）；
```

```
scanf ( "%d",&m ) ;
printf ( "the step to moving %3d disks :\n",m ) ;
Hanoi ( m,'A','B','C' ) ;
}
```

版权方授权希赛网发布，侵权必究

其他典型例程汇集

24.9 其他典型例程汇集

本节主要介绍其他典型例程汇集。

版权方授权希赛网发布，侵权必究

有序链表的合并

24.9.1 有序链表的合并

设有两个单链表A、B,其中元素递增有序，编写算法将A、B归并成一个按元素值递减（允许有相同值）并有序的链表C,要求用A、B中的原结点形成，不能重新申请结点。

算法思路：利用A、B两表有序的特点，依次进行比较，将当前值较小者摘下，插入到C表的头部，得到的C表则为递减有序的。

程序实现见程序24-24.

【程序24-24】

```
LinkList  merge ( LinkList A,LinkList B )  /*设A、B均为带头结点的单链表*/
{ LinkList C;  LNode  *p,*q;
p=A->next;q=B->next;
C=A;                        /*C表的头结点*/
C->next=NULL;
free ( B ) ;
while  ( p&&q )
{ if  ( p->data<q->data )
{ s=p;p=p->next; }
else
{s=q;q=q->next;}            /*从原A和B表上摘下较小者*/
```

```
s->next=C->next;              /*插入到C表的头部*/
C->next=s;
}
if （p= =NULL） p=q;
while （p）          /*将剩余的结点一个个摘下，插入到C表的头部*/
{ s=p;p=p->next;
s->next=C->next;
C->next=s;
}
}
```

版权方授权希赛网发布，侵权必究

第 24 章：常用算法设计　作者：希赛教育软考学院　来源：希赛网　2014年01月27日

链表多项式加法

24.9.2 链表多项式加法

用链表表示的两个多项式相加函数，一个多项式pn（x）=anxn+an-1xn-1+…+a1x+a0可用一个链表来表示。其中表元是由幂次、系数和后继表元指针3个成分组成的结构。

函数addpoly（ ）实现l（x）+k（x）→l（x）从高次项到低次项，逐项考查多项式l（x）和k（x）当前项的幂次。若l（x）多项式已经结束，则在l（x）的链表的末尾接上一项，其幂次与系数均与k（x）相同。在l（x）多项式还未结束的情况下，如l（x）多项式当前项的幂次与k（x）多项式当前项的幂次相等，则将k（x）当前项的系数累加到l（x）当前项，并当和为零时，l（x）当前项应从链表中删去。若l（x）当前项的幂次大于k（x）当前项的幂次，则l（x）当前项不变。若l（x）当前项的幂次小k（x）当前项的幂次，则在l（x）链表中插入一项，其幂次与系数均与k（x）当前项的相同。重复以上处理过程，直至k（x）结束。

程序实现见程序24-25.

【程序24-25】

```
#include <stdio.h>
#define EPSILON 1.0e-5
#include <math.h>
struct node {
int  power;
doudle  coef;
struct  node *link;
};
void addpoly（l,y）          /*建立多项式链表和多项式链表相加*/
{ struct node  *l,*k;
```

```
struct node *p, *lpt, *kpt, *q;
p=l;
lpt=l–>link;
kpt=k–>link;
while ( kpt ) {
if ( lpt= =null )              /*l多项式当前项默认*/
{                    /*生成与k多项式当前项值相同的新结点*/
q= ( struct node * ) malloc ( sizeof ( struct node ) ) ;
q–>power=kpt–>power;
q–>coef=kpt–>coef;
p–>link=q;           /*新结点接在l多项式链表的末尾*/
q–>link=null;
p=q;
kpt=kpt–>link;   /*让k多项式当前项的后继项为当前项*/
}
else if ( lpt–>power= =kpt–>power ) /*l多项式和k多项式当前项幂次相等*/
{
lpt–>coef+=kpt–>coef;   /*等幂次项的系数相加*/
if ( fabs ( lpt–>coef ) <=EPSILON )
{              /*若l多项式的当前项系数为0,则从链表中删除*/
p–>link=lpt–>link;      /*p为当前项的前驱指针*/
free ( lpt ) ;
}
else p=p–>link;
lpt=p–>link;                /*调整l多项式的当前项*/
kpt=kpt–>link;
}
else if ( lpt–>power>kpt–>power ) /*l多项式当前项幂次大于k多项式当前项幂次*/
{                          /*跳过 ( *lpt ) */
p=lpt;
lpt=lpt–>link;
}
else                        /*l多项式当前项幂次小于k多项式当前项幂次*/
{                         /*复制 ( *kpt ) , 生成新结点*/
q= ( struct node * ) malloc ( sizeof ( struct node ) ) ;
q–>power=kpt–>power;
q–>coef=kpt–>coef;
p–>link=q;
q–>link=lpt;
```

```
p=q;
kpt=kpt–>link;
}
}
return;
}
struct node *create_list（ ）          /*输入多项式的幂次和系数*/
{ struct node  *u,*w,*p,*h;
int v; double e;
h=（struct node *）malloc（sizeof（struct node））；/*建立空的链表*/
h–>link=null;
h–>power=-1;
printf（"input data"（<0:finish）\n"）；
scanf（"%d",&v）；              /*输入幂次*/
while（v>=0）
{
p=（struct node *）malloc（sizeof（struct node））；
p–>power=v;
scanf（"%1f",&e）；          /*输入系数*/
p–>coef=e;
w=h;
for（u=h–>link;u!=null&&v<u–>power;w=u,u=u–>link）；
w–>link=p;
p–>link=u;
scanf（"%d",&v）；          /*输入幂次*/
}
return h;
}
main（ ）
{ struct node  *h1,*h2,*p,*q;
h1=creat_list（ ）；
h2=creat_list（ ）；
addpoly（h1,h2）；
q=h1–>link;
while（q）{                    /*输出*/
printf（"%d%f\n",q–>power,q–>coef）；
q=q–>link; }
q=h1;                        /*释放空间*/
while（q）{
```

```
p=q->link;
ree ( q ) ;
q=p; }
q=h2;
while ( q ) {
p=q->link;
free ( q ) ;
q=p; }
}
```

版权方授权希赛网发布，侵权必究

约瑟夫环问题

24.9.3 约瑟夫环问题

约瑟夫环问题：设编号为1,2,3,…，n的n（n>0）个人按顺时针方向围坐一圈，每个人持有一个正整数密码。开始时任选一个正整数作为报数上限m,从第一个人开始顺时针方向自1起顺序报数，报到m则停止报数，报m的人出列，将他的密码作为新的m值，从他的下一个人开始重新从1报数。如此下去，直到所有人全部出列为止。令n最大值取30.要求设计一个程序模拟此过程，求出出列编号序列。

程序实现见程序24-26.

【程序24-26】

```
#include <stdlib.h>
#include <malloc.h>
struct node
{
int number;                /* 人的序号 */
int cipher;               /* 密码 */
struct node *next;          /* 指向下一个结点的指针 */
};
struct node *CreatList ( int num )  /* 建立循环链表 */
{
int i;
struct node *ptr1,*head;
if ( ( ptr1= ( struct node * ) malloc ( sizeof ( struct node ) ) ) = =NULL )
{
perror ( "malloc" ) ;
```

```
return ptr1;
}
head=ptr1;
ptr1->next=head;
for ( i=1;i<num;i++ )
{
if ( ( ptr1->next= ( struct node * ) malloc ( sizeof ( struct node ) ) ) = =NULL )
{
perror ( "malloc" ) ;
ptr1->next=head;
return head;
}
ptr1=ptr1->next;
ptr1->next=head;
}
return head;
}
main ( )
{
int i,n=30,m;                /* 人数n为30个 */
struct node *head,*ptr;
randomize ( ) ;
head=CreatList ( n ) ;
for ( i=1;i<=30;i++ )
{
head->number=i;
head->cipher=rand ( ) ;
head=head->next;
}
m=rand ( ) ;                  /* m取随机数 */
/*因为我没办法删除head指向的结点，只会删除head的下一结点，所以只能从0数起*/
i=0;
while ( head->next!=head )       /* 当剩下最后一个人时，退出循环 */
{
if ( i= =m )
{
ptr=head->next;     /* ptr记录数到m的那个人的位置 */
printf ( "number:%d\n",ptr->number ) ;
printf ( "cipher:%d\n",ptr->cipher ) ;
```

```
m=ptr->cipher;     / *让m等于数到m的人的密码 */
head->next=ptr->next; /*让ptr从链表中脱节，将前后两个结点连接起来*/
head=hea/d->next;   /* head移向后一个结点 */
free（ptr）；            /*释放ptr指向的内存*/
i=0;               /*将i重新置为0,从0再开始数*/
}
else
{
head=head->next;
i++;
}
}
printf（"number:%d\n",head->number）；
printf（"cipher:%d\n",head->cipher）；
free（head）；                /*让最后一个人也出列*/
}
```

版权方授权希赛网发布，侵权必究

旅行线路问题

24.9.4 旅行线路问题

旅行线路选择。设有n个城市（或景点），今从某市出发遍历各城市，使之旅费最少（即找出一条旅费最少的路径）。

设矩阵元素aij表示从第i号城市到第j号城市的旅费，并设城市间往返旅费可以不等（即aij不等于aji）。aii是没有意义的，由于问题是求最小，因此aii不应为0,可以设一个足够大的值MAX.各城市间旅费如下：

$$\begin{pmatrix} MAX & 17 & 13 & 24 & 10 \\ 10 & MAX & 20 & 9 & 6 \\ 17 & 29 & MAX & 21 & 28 \\ 12 & 10 & 22 & MAX & 19 \\ 12 & 18 & 31 & 20 & MAX \end{pmatrix}$$

问题算法是在表每行中找最小元素，并用该数减该行非MAX元素。再对每列也施以同样的工作，形成一个新表（保证每行、每列不少于1个为0），所有减数累加为min（其含义为旅费下界，即旅费不会少于min）。旅行路程因成环路，故可以设起点是第0号城市。若选第i号到第j号程序，则表上bij表示还需旅费。同时由于选择了i→j,则i不可能再选其他城市，则第i行全填MAX.同理，由于j已由i过来，则第j程序不可能再由其他城市过来，第j列也全填上MAX.对新矩阵施以上述同样操作，

找出余下城市遍历所需旅费下界mj.对于不同的j,比较mj+bij以最小的一个为选定从i到达的城市，并将选择路径记下、如此重复到选完。初始时，表处理如下：

$$\begin{pmatrix} MAX & 7 & 0 & 11 & 0 \\ 4 & MAX & 11 & 0 & 0 \\ 0 & 12 & MAX & 1 & 11 \\ 2 & 0 & 9 & MAX & 9 \\ 0 & 6 & 16 & 5 & MAX \end{pmatrix}$$

程序实现见程序24-27.

【程序24-27】

```
#include<stdio.h>
#define  n  5
#define  max  1000
typedef  int  array[n][n];
array  a, b, c, d;
int  path[n],s[n];
int  p ( array b, int *m )              /*表的调整，m为所有减数的累加和*/
{int  pi, pj, pk;
*m=0;
for ( pi=0;pi<n; pi++ )          /*寻找每行最小的元素*/
{pk=max;
for  ( pj=0;pj<n; pj++ )
if ( b[pi][pj]<pk )
pk=b[pi][pj];       /*pk为当前行中最小的元素*/
if ( ( pk>0 ) && ( pk!=max ) )
{*m=*m+pk;           /*减数累加*/
for ( pj=0;pj<n; pj++ )
if ( b[pi][pj]!=max )
b[pi][pj]= b[pi][pj]-pk;/*当前行中的每个元素减该行中最小的元素*/
}
}
for ( pj=0;pj<n; pj++  )             /*寻找每列最小的元素*/
{pk=max;
for ( pi=0;pi<n; pi++ )
if ( b[pi][pj]<pk )
pk=b[pi][pj];              /*pk为当前列中最小的元素*/
if ( ( pk>0 ) && ( pk!=max ) )
{ *m=*m+pk;           /*减数累加*/
for ( pi=0;pi<n;pi++ )
if ( b[pi][pj]!=max )
b[pi][pj]=b[pi][pj]-pk; /*当前列中的每个元素减该列中最小的元素*/
}
```

```
}
return;
}
main（ ）
{int i, j, k, min, ml, m, jj, kk, si, sj;
printf（ "\n Input traveling expenses table:" ）
printf（ "\n" ） ;
for（ i=0;i<n; i++ ）
for（ j=0;j<n; j++ ）
scanf（ "%d",&a[i][j] ） ;          /*输入旅费表数组a*/
for（ i=0;i<n;i++ ）a[i][i]=max;       /*旅费aii没有意义，全部置max*/
k=0; path[0]=0; i=0;
s[0]=max;             /*s数组的i元素标记为max表示已经到过城市i,从0号城市出发*/
for（ j=1;j<n;j++ ）s[j]=0;    /*s数组的i元素标记为0表示i城市还没有被访问*/
for（ si=0;si<n;si++ ）
for（ sj=0;sj<n;sj++ ）
b[si][sj]=a[si][sj];      /*将旅费表数组a备份到b数组*/
p（ b,&min ） ;                    /*调整旅费表备份数组b,使得b为初始表*/
do{                          /*从0号城市出发，选择下一个城市*/
m1=max;
for（ j=0;j<n j++ ）         /*从i城市出发，依次试探没有到达过的城市，选择减数累加和
最小且没有到达过的城市为下一个要到达的城市*/
if（ （ s[j]= =0&&（ b[i][j]!=max ） ）
{  for （ si=0;si<n; si++ ）
for（ sj=0;sj<n; sj++ ）
c[si][sj]=b[si][sj];   /*将初始表送工作数组c*/
for（ kk=0;kk<n;kk++ ）
{ c[i][kk]=max;
c[kk][j]=max;     /*从i城市出发到j,将旅费初始表i行j列全置max*/
}
p（ c,&m ） ;                /*调整置max后的工作数组*/
if（ m+b[i][j]<m1 ）           /*判断最小减数累加和*/
{ m1=m+b[i][j]; jj=j;     /*jj保存旅费表中减数累加和最小的城市号*/
for（ si=0;si<n; si++ ）
for（ sj=0;sj<n; sj++ ）
d[si][sj]=c[si][sj];     /*中间数组d保存减数累加和最小的旅费表数组*/
}
}
for（ si=0;si<n; si++ ）
```

```
for（sj=0;sj<n; sj++）
/*将减数累加和最小旅费表数组送b,使jj成为当前所到达的城市号*/
b[si][sj]=d[si][sj];
min=min+m1;i=jj;
k=k+1;path[k]=jj;       /*数组path记录旅行的路径*/
s[jj]=max; sj=max;       /*到达城市jj,设置到达标识*/
for（si=0;si<n;si++）       /*sj赋值用来控制while语句的条件转移*/
if（s[si]!=max）sj=0;   /*如果还有没有到达的城市，则令sj为0,否则sj为max*/
}while（sj!=max）；              /*若sj为0则还有没到达的城市，继续下一个循环*/
printf（"\n\n the route path is:"）；
for（i=0,i<=k,i++）
printf（"%4d–>",path[i]）；
printf（"0"）；
printf（"\n total of traveling expenses :"）；
printf（"%5d",min）；
}
```

算法过程如下所示（M代表MAX）。

从0号城市出发的4种可能：

0→1 min=16

M	M	M	M	M
4	M	4	0	0
0	M	M	1	11
0	M	0	M	7
0	M	9	5	M

0→2 min=0

M	M	M	M	M
4	M	M	0	0
0	12	M	1	11
2	0	M	M	9
0	6	M	5	M

0→3 min=20

M	M	M	M	M
4	M	2	M	0
0	12	M	M	11
2	0	0	M	9
0	6	7	M	M

0→4 min=9

M	M	M	M	M
4	M	2	0	M
0	12	M	1	M
2	0	0	M	M
0	6	7	5	M

从中选择应是0→2.再从2出发，有3种选择：

2→1 min=14

M	M	M	M	M
4	M	M	0	0
M	M	M	M	M
0	M	M	M	7
0	M	M	5	M

2→3 min=1

M	M	M	M	M
4	M	M	M	0
M	M	M	M	M
2	0	M	M	9
0	6	M	M	M

2→4 min=11

M	M	M	M	M
4	M	M	0	M
M	M	M	M	M
2	0	M	M	M
0	6	M	5	M

从中应选2→3.再从3出发，有2种选择：

3→1 min=0

M	M	M	M	M
4	M	M	M	0
M	M	M	M	M
M	M	M	M	M
0	M	M	M	M

3→4 min=19

M	M	M	M	M
0	M	M	M	M
M	M	M	M	M
M	M	M	M	M
0	0	M	M	M

从中选3→1.再从1出发可选：

1→4　min=0

$$\begin{pmatrix} M & M & M & M & M \\ M & M & M & M & M \\ M & M & M & M & M \\ M & M & M & M & M \\ 0 & M & M & M & M \end{pmatrix}$$

最后的结果应该为0→2→3→1→4→0.

版权方授权希赛网发布，侵权必究

　作者：希赛教育软考学院　来源：希赛网　2014年01月27日

迷宫最短路径问题

24.9.5 迷宫最短路径问题

求迷宫的最短路径：现要求设计一个算法找一条从迷宫入口到出口的最短路径。

本算法要求找一条迷宫的最短路径，算法的基本思想为：从迷宫入口点（1,1）出发，向四周搜索，记下所有一步能到达的坐标点；然后依次再从这些点出发，再记下所有一步能到达的坐标点，……，依次类推，直到到达迷宫的出口点（m,n）为止，然后从出口点沿搜索路径回溯直至入口。这样就找到了一条迷宫的最短路径，否则迷宫无路径。

有关迷宫的数据结构、试探方向、如何防止重复到达某点以避免发生死循环的问题与前面算法部分的迷宫问题处理相同，不同的是如何存储搜索路径。在搜索过程中必须记下每一个可到达的坐标点，以便从这些点出发继续向四周搜索。由于先到达的点先向下搜索，故引进一个"先进先出"数据结构--队列来保存已到达的坐标点。到达迷宫的出口点（m,n）后，为了能够从出口点沿搜索路径回溯直至入口，对于每一点，记下坐标点的同时，还要记下到达该点的前驱点，因此，用一个结构数组sq[num]作为队列的存储空间。因为迷宫中每个点至多被访问一次，所以num至多等于m*n.sq的每一个结构有3个域：x、y和pre.其中x、y分别为所到达的点的坐标，pre为前驱点在sq中的坐标，是一个静态链域。除sq外，还有队头、队尾指针front和rear用来指向队头和队尾元素。

队的定义如下：

```
typedef  struct
{  int x,y;
int pre;
} sqtype;
sqtype sq[num];
int front,rear;
```

初始状态，队列中只有一个元素sq[1]记录的是入口点的坐标（1,1）。因为该点是出发点，因此没有前驱点，pre域为-1,队头指针front和队尾指针rear均指向它，此后搜索时都是以front所指点为搜索的出发点。当搜索到一个可到达点时，即将该点的坐标及front所指点的位置入队，不但记下了到达点的坐标，还记下了它的前驱点。front所指点的8个方向搜索完毕后，则出队，继续对下一点搜索。搜索过程中遇到出口点则成功，搜索结束，打印出迷宫最短路径，算法结束。或者当前队空

即没有搜索点了，表明没有路径算法也结束。

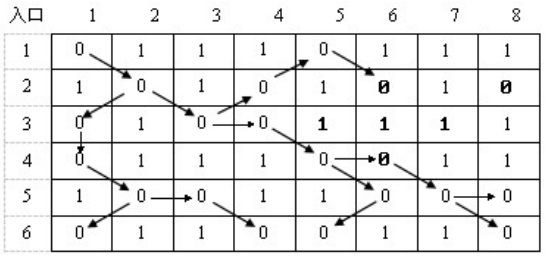
程序实现见程序24-28.

【程序24-28】

```
typedef  struct
{  int x,y;
int pre;
} sqtype;
sqtype sq[num];
int front,rear;
void path（maze,move）
int maze[m][n];     /*迷宫数组*/
item move[8];        /*坐标增量数组，其定义和值见算法部分迷宫问题*/
{ sqtype sq[NUM];
int front,rear;
int x,y,i,j,v;
front=rear=0;
sq[0].x=1; sq[0].y=1; sq[0].pre=-1;  /*入口点入队*/
maze[1,1]=-1;
while （front<=rear）            /*队列不空*/
{  x=sq[front].x ;  y=sq[front ].y ;
for （v=0;v<8;v++）
{ i=x+move[v].x;  j=x+move[v].y;
if （maze[i][j]= =0）
{  rear++;
sq[rear].x=i; sq[rear].y=j;  sq[rear].pre=front;
maze[i][j]=-1;
}
if （i= =m&&j= =n）
{  printpath（sq,rear）；  /*打印迷宫*/
restore（maze）；    /*恢复迷宫*/
return 1;
}
}                  /*for v*/
front++;                /*当前点搜索完，取下一个点搜索 */
}                     /*while循环结束*/
return 0;
}                             /*path子程序结束*/
void printpath（sqtype sq[],int rear）/*打印迷宫路径*/
{  int   i;
```

```
i=rear;
do { printf ( " ( %d,%d ) ←",sq[i].x , sq[i].y ) ;
i=sq[i].pre;            /*回溯*/
} while ( i!=-1 ) ;
}                              /*printpath子程序结束*/
```

对于图24-19（a）所示的迷宫的搜索过程如图24-19（b）所示。



（a）用二维数组表示的迷宫

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19…
x	1	2	3	3	3	2	4	4	1	5	4	5	2	5	6	5	6	6	5	6
y	1	2	3	1	4	4	1	5	5	2	6	6	6	3	1	7	5	4	8	8
pre	-1	0	1	1	2	2	3	4	5	6	7	7	8	9	9	10	11	13	15	15

（b）队列中的数据

图24-19 迷宫搜索过程

运行结果：（6,8）←（5,7）←（4,6）←（4,5）←（3,4）←（3,3）←（2,2）←（1,1）。

在上面的例子中，不能采用循环队列。因为在本问题中，队列中保存了探索到的路径序列，如果用循环队列，则会把先前得到的路径序列覆盖掉。而在有些问题中，如持续运行的实时监控系统中，监控系统源源不断地收到监控对象顺序发来的信息，如报警，为了保持报警信息的顺序性，就要按顺序一一保存。而这些信息是无穷多个，不可能全部同时驻留内存，可根据实际问题，设计一个适当大的向量空间，用做循环队列，最初收到的报警信息一一入队。当队满之后，又有新的报警到来到时，新的报警则覆盖掉了旧的报警。内存中始终保持当前最新的若干条报警，以便满足快速查询。

版权方授权希赛网发布，侵权必究

上一节　本书简介　下一节

例题分析

24.10 例题分析

例题（2008年12月试题5）

阅读下列说明和C函数，将应填入（n）处的字句写在答题纸的对应栏内。

【说明】

已知集合A和B的元素分别用不含头结点的单链表存储，函数Difference（ ）用于求解集合A与B的差集，并将结果保存在集合A的单链表中。例如，若集合A={5,10,20,15,25,30},集合B={5,15,35,25},如图24-20（a）所示，运算完成后的结果如图24-20（b）所示。

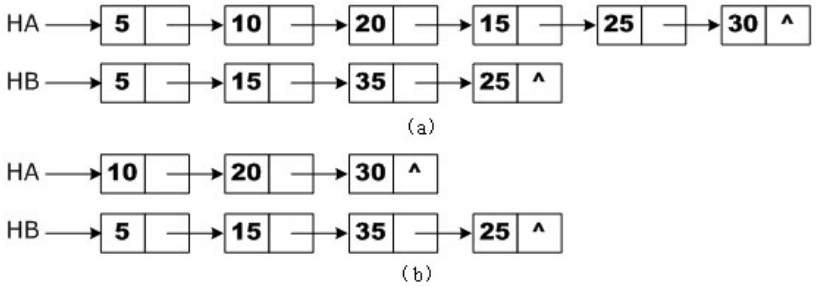


图24-20 集合A、B运算前后示意图

链表结点的结构类型定义如下：

```
typedef struct Node{
ElemType elem;
struct Node *next;
}NodeType;
```

【C 函数】

```
void Difference ( NodeType **LA, NodeType *LB )
{
NodeType *pa, *pb, *pre, *q;
pre = NULL;
（1）;
while （pa）{
pb = LB;
while （ （2） ）
pb = pb->next;
if （ （3） ）{
if （ ! pre )
*LA = （4）;
else
（5） = pa->next;
q = pa;
pa = pa->next;
free ( q ) ;
}
else  {
（6）;
pa = pa->next;
}
}
}
```

例题分析：

该题是一道C语言程序题。题目考查数据结构当中的链表操作。程序要实现的功能比较简单，即从链表A中，去除链表A和链表B均有的公共元素。其中的填空主要是对链表的一些基本操作。

算法的基本原理是：遍历链表A,对于A中的每一个结点，都在B链表中进行查询，若在B中查到相同值的结点，则将链表A中的该结点删除。

下面分析程序：

1.程序中的（1）空显然是对pa赋初值，结合上面的算法分析可知pa应指向链表A的首指针，所以该空应填：pa=*LA.

2.接下来第（2）空所在的while循环的作用是在B链表中查找与A链表当前结点值相等的结点，所以循环结束条件有两个（满足一个条件即退出循环）：其一、B链表已查询完毕（也就是pb为NULL）；其二、找到与A链表当前结点值相等的结点（即pb->elem==pa->elem）。所以本空应填：pb&&pb->elem!=pa->elem.

3.第（3）、（4）、（5）空，从程序代码及之前的算法分析可以得知，（3）所在if语句的真分支是将链表A当前结点删除的操作，这表明此时在B中找到了与A链表相等的结点，结合第（2）空的分析可知（3）空应填pb.第（4）空与第（5）空是处理删除结点的两种情况：当需要删除的结点是链表首结点时，删除该结点只要将链表头指针直接指向当前结点的next域即可；当需要删除的结点是链表的中间结点时，则将当前结点的前趋结点next域，指向当前结点的后继结点。所以（4）填：pa->next,（5）填：pre->next.

```
if （ ！pre）
*LA = （4）；
else
（5） = pa->next;
q = pa;
pa = pa->next;
free（q）；
```

4.前面我们用到了pa的前趋结点pre进行相应操作，为了保障pre始终指向pa的前趋结点。pre的值是随pa的值变化而变的，当pa指向下一个结点时，pre应指向pa的当前结点。所以（6）应填：pre=pa.

例题参考答案：

（1）pa=*LA（2分）

（2）pb&&pb->elem!=pa->elem,或其等价表示（3分）

（3）pb（2分）

（4）pa->next,或（*pa）。next,或其等价表示（3分）

若考生解答为*pa.next,则给2分

（5）pre->next,或（*pre）。next,或其等价表示（2分）

若考生解答为*pre.next,则给1分

（6）pre=pa（3分）

版权方授权希赛网发布，侵权必究