数据结构经典问题和算法分析（一）-迭代法

来源:　作者:　2007-5-30 21:17:53　字体:[大 中 小]

一、迭代法

迭代法是用于求方程或方程组近似根的一种常用的算法设计方法。设方程为 f(x)=0，用某种数学方法导出等价的形式 x=g(x)，然后按以下步骤执行：

（1）　选一个方程的近似根，赋给变量 x0；

（2）　将 x0 的值保存于变量 x1，然后计算 g(x1)，并将结果存于变量 x0；

（3）　当 x0 与 x1 的差的绝对值还小于指定的精度要求时，重复步骤（2）的计算。

若方程有根，并且用上述方法计算出来的近似根序列收敛，则按上述方法求得的 x0 就认为是方程的根。上述算法用 C 程序的形式表示为：

【算法】迭代法求方程的根

```
{   x0=初始近似根；
  do {
    x1=x0；
    x0=g(x1)；  /*按特定的方程计算新的近似根*/
    } while ( fabs(x0-x1)>Epsilon)；
  printf(“方程的近似根是%f\n”，x0)；


}
```

迭代算法也常用于求方程组的根，令

X=（x0，x1，…，xn-1）

设方程组为：

xi=gi(X)　(I=0，1，…，n-1)

则求方程组根的迭代算法可描述如下：

【算法】迭代法求方程组的根

```
 {   for (i=0;i<n;i++)
       x[i]=初始近似根;
    do {
      for (i=0;i<n;i++)
        y[i]=x[i];
      for (i=0;i<n;i++)
        x[i]=gi(X);
      for (delta=0.0,i=0;i<n;i++)

        if (fabs(y[i]-x[i])>delta)      delta=fabs(y[i]-x[i])；
      } while (delta>Epsilon)；
    for (i=0;i<n;i++)
      printf(“变量 x[%d]的近似根是 %f”，I，x[i])；
```

```
    printf("\n")；
  }
```

具体使用迭代法求根时应注意以下两种可能发生的情况：

（1） 如果方程无解，算法求出的近似根序列就不会收敛，迭代过程会变成死循环，因此在使用迭代算法前应先考察方程是否有解，并在程序中对迭代的次数给予限制；

（2） 方程虽然有解，但迭代公式选择不当，或迭代的初始近似根选择不合理，也会导致迭代失败。

数据结构经典问题和算法分析（二） 穷举搜索法

来源: 作者: 2007-5-30 21:26:51 字体:[大 中 小]

二、穷举搜索法

穷举搜索法是对可能是解的众多候选解按某种顺序进行逐一枚举和检验，并从众找出那些符合要求的候选解作为问题的解。

【问题】 将 A、B、C、D、E、F 这六个变量排成如图所示的三角形，这六个变量分别取[1，6]上的整数，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。如图就是一个解。

程序引入变量 a、b、c、d、e、f，并让它们分别顺序取 1 至 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。细节见下面的程序。

【程序 1】

```
# 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==e+f+a))   {
printf("%6d,a);
   printf("%4d%4d",b,f);
   printf("%2d%4d%4d",c,d,e);
```

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

按穷举法编写的程序通常不能适应变化的情况。如问题改成有 9 个变量排成三角形，每条边有 4 个变量的情况，程序的循环重数就要相应改变。

对一组数穷尽所有排列，还有更直接的方法。将一个排列看作一个长整数，则所有排列对应着一组整数。将这组整数按从小到大的顺序排列排成一个整数，从对应最小的整数开始。按数列的递增顺序逐一列举每个排列对应的每个整数，这能更有效地完成排列的穷举。从一个排列找出对应数列的下一个排列可在当前排列的基础上作部分调整来实现。倘若当前排列为 1，2，4，6，5，3，并令其对应的长整数为 124653。要寻找比长整数 124653 更大的排列，可从该排列的最后一个数字顺序向前逐位考察，当发现排列中的某个数字比它前一个数字大时，如本例中的 6 比它的前一位数字 4 大，这说明还有对应更大整数的排列。但为了顺序从小到大列举出所有的排列，不能立即调整得太大，如本例中将数字 6 与数字 4 交换得到的排列 126453 就不是排列 124653 的下一个排列。为了得到排列 124653 的下一个排列，应从已经考察过的那部分数字中选出比数字大，但又是它们中最小的那一个数字，比如数字 5，与数字 4 交换。该数字也是从后向前考察过程中第一个比 4 大的数字。5 与 4 交换后，得到排列 125643。在前面数字 1，2，5 固定的情况下，还应选择对应最小整数的那个排列，为此还需将后面那部分数字的排列顺序颠倒，如将数字 6，4，3 的排列顺序颠倒，得到排列 1，2，5，3，4，6，这才是排列 1，2，4，6，5，3 的下一个排列。按以上想法编写的程序如下。

【程序 2】

```
# include <stdio.h>
# define SIDE_N   3
# define LENGTH   3
# define VARIABLES   6
int A,B,C,D,E,F;
int *pt[]={&A,&B,&C,&D,&E,&F};
int *side[SIDE_N][LENGTH]={&A,&B,&C,&C,&D,&E,&E,&F,&A};
int side_total[SIDE_N];
main{}
{   int i,j,t,equal;
  for (j=0;j<VARIABLES;j++)
     *pt[j]=j+1;
  while(1)
  {   for (i=0;i<SIDE_N;i++)
     {   for (t=j=0;j<LENGTH;j++)
```

```
        t+=*side[i][j];
      side_total[i]=t;

    }
    for (equal=1,i=0;equal&&i<SIDE_N-1;i++)
      if (side_total[i]!=side_total[i+1]   equal=0;
    if (equal)
    {   for (i=1;i<VARIABLES;i++)
        printf("%4d",*pt[i]);
      printf("\n");
      scanf("%*c");
    }
    for (j=VARIABLES-1;j>0;j--)
      if (*pt[j]>*pt[j-1])   break;
    if (j==0)   break;

    for (i=VARIABLES-1;i>=j;i--)
      if (*pt[i]>*pt[i-1])   break;
    t=*pt[j-1];* pt[j-1] =* pt[i]; *pt[i]=t;
    for (i=VARIABLES-1;i>j;i--,j++)
    {   t=*pt[j]; *pt[j] =* pt[i]; *pt[i]=t;   }
  }
}
```

从上述问题解决的方法中，最重要的因素就是确定某种方法来确定所有的候选解。下面再用一个示例来加以说明。

【问题】　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。

设 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。考虑一个 n 元组（x0，x1，…，xn-1），其中 xi=0 表示第 i 个物品没有选取，而 xi=1 则表示第 i 个物品被选取。显然这个 n 元组等价于一个选择方案。用枚举法解决背包问题，需要枚举所有的选取方案，而根据上述方法，我们只要枚举所有的 n 元组，就可以得到问题的解。

显然，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。

【算法】

```
maxv=0;
for (i=0;i<2n;i++)
{   B[0..n-1]=0;
  把 i 转化为二进制数，存储于数组 B 中;
```

```
  temp_w=0;
  temp_v=0;
  for (j=0;j<n;j++)
  {   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;
        保存该 B 数组；

     }
  }
}
```

数据结构经典问题和算法分析（三）递推法

来源:　作者:　2007-5-30 21:31:13　字体:[大 中 小]

三、递推法

递推法是利用问题本身所具有的一种递推关系求问题解的一种方法。设要求问题规模为 N 的解，当 N=1 时，解或为已知，或能非常方便地得到解。能采用递推法构造算法的问题有重要的递推性质，即当得到问题规模为 i-1 的解后，由问题的递推性质，能从已求得的规模为 1，2，…，i-1 的一系列解，构造出问题规模为 I 的解。这样，程序可从 i=0 或 i=1 出发，重复地，由已知至 i-1 规模的解，通过递推，获得规模为 i 的解，直至得到规模为 N 的解。

【问题】　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。

由于要求的整数可能大大超出一般整数的位数，程序用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 位成整数 N 用数组 a[ ]存储：

N=a[m]×10m-1+a[m-1]×10m-2+ … +a[2]×101+a[1]×100

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，5！=120，在数组中的存储形式为：

3　0　2　1　……

首元素 3 表示长整数是一个 3 位数，接着是低位到高位依次是 0、2、1，表示成整数 120。

计算阶乘 k！可采用对已求得的阶乘(k-1)！连续累加 k-1 次后求得。例如，已知 4！=24，计算 5！，可对原来的 24 累加 4 次 24 后得到 120。细节见以下程序。

```
# include <stdio.h>
# include <malloc.h>
```

```
# define  MAXN   1000
void  pnext(int a[ ],int 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++)
   {   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();
   }
```

数据结构经典问题和算法分析（四）递归

来源:　作者:　2007-5-30 21:35:56　字体:[大 中 小]

四、递归

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，为此在进一步介绍其他算法设计方法之前先讨论它。

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。

【问题】　编写计算斐波那契（Fibonacci）数列的第 n 项函数 fib（n）。

斐波那契数列为：0、1、1、2、3、……，即：

fib(0)=0;

fib(1)=1;

fib(n)=fib(n-1)+fib(n-2)　　（当 n>1 时）。

写成递归函数有：

```
int fib(int n)
{   if (n==0)      return  0;

   if (n==1)      return  1;
   if (n>1)      return  fib(n-1)+fib(n-2);
}
```

递归算法的执行过程分递推和回归两个阶段。在递推阶段，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。例如上例中，求解 fib(n)，把它推到求解 fib(n-1)和 fib(n-2)。也就是说，为计算 fib(n)，必须先计算 fib(n-1)和 fib(n-2)，而计算 fib(n-1)和 fib(n-2)，又必须先计算 fib(n-3)和 fib(n-4)。依次类推，直至计算 fib(1)和 fib(0)，分别能立即得到结果 1 和 0。在递推阶段，必须要有终止递归的情况。例如在函数 fib 中，当 n 为 1 和 0 的情况。

在回归阶段，当获得最简单情况的解后，逐级返回，依次得到稍复杂问题的解，例如得到 fib(1)和 fib(0)后，返回得到 fib(2)的结果，……，在得到了 fib(n-1)和 fib(n-2)的结果后，返回得到 fib(n)的结果。

在编写递归函数时要注意，函数中的局部变量和参数知识局限于当前调用层，当递推进入“简单问题”层时，原来层次上的参数和局部变量便被隐蔽起来。在一系列“简单问题”层，它们各有自己的参数和局部变量。

由于递归引起一系列的函数调用，并且可能会有一系列的重复计算，递归算法的执行效率相对较低。当某个递归算法能较方便地转换成递推算法时，通常按递推算法编写程序。例如上例计算斐波那契数列的第 n 项的函数 fib(n)应采用递推算法，即从斐波那契数列的前两项出发，逐次由前两项计算出下一项，直至计算出要求的第 n 项。

【问题】　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。例如 n=5，r=3 的所有组合为：　（1）5、4、3　　（2）5、4、2　　（3）5、4、1　（4）5、3、2　　（5）5、

3、1　　（6）5、2、1　　（7）4、3、2　　（8）4、3、1　　（9）4、2、1　　（10）3、2、1

分析所列的 10 个组合，可以采用这样的递归思想来考虑求组合函数的算法。设函数为 void comb(int m,int k)为找出从自然数 1、2、……、m 中任取 k 个数的所有组合。当组合的第一个数字选定时，其后的数字是从余下的 m-1 个数中取 k-1 数的组合。这就将求 m 个数中取 k 个数的组合问题转化成求 m-1 个数中取 k-1 个数的组合问题。设函数引入工作数组 a[ ]存放求出的组合的数字，约定函数将确定的 k 个数字组合的第一个数字放在 a[k]中，当一个组合求出后，才将 a[ ]中的一个组合输出。第一个数可以是 m、m-1、……、k，函数将确定组合的第一个数字放入数组后，有两种可能的选择，因还未去顶组合的其余元素，继续递归去确定；或因已确定了组合的全部元素，输出这个组合。细节见以下程序中的函数 comb。

【程序】

```
# include   <stdio.h>
# define   MAXN   100
int   a[MAXN];
void   comb(int m,int k)
{   int i,j;
  for (i=m;i>=k;i--)
  {   a[k]=i;   // a[m]---a[1] store the combination
    if (k>1)
      comb(i-1,k-1);
    else
    {   for (j=a[0];j>0;j--)
        printf("%4d",a[j]);
      printf("\n");
    }
  }
}

void main()

{   a[0]=3;
  comb(5,3);
}
```

【问题】　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。

设 n 件物品的重量分别为 w0、w1、…、wn-1，物品的价值分别为 v0、v1、…、vn-1。采用递归寻找物品的选择方案。设前面已有了多种选择的方案，并保留了其中总价值最大的方

案于数组 option[ ]，该方案的总价值存于变量 maxv。当前正在考察新方案，其物品选择情况保存于数组 cop[ ]。假定当前方案已考虑了前 i-1 件物品，现在要考虑第 i 件物品；当前方案已包含的物品的重量之和为 tw；至此，若其余物品都选择是可能的话，本方案能达到的总价值的期望值为 tv。算法引入 tv 是当一旦当前方案的总价值的期望值也小于前面方案的总价值 maxv 时，继续考察当前方案变成无意义的工作，应终止当前方案，立即去考察下一个方案。因为当方案的总价值不比 maxv 大时，该方案不会被再考察，这同时保证函数后找到的方案一定会比前面的方案更好。

对于第 i 件物品的选择考虑有两种可能：

（1） 考虑物品 i 被选择，这种可能性仅当包含它不会超过方案总重量限制时才是可行的。选中后，继续递归去考虑其余物品的选择。

（2） 考虑物品 i 不被选择，这种可能性仅当不包含物品 i 也有可能会找到价值更大的方案的情况。

按以上思想写出递归算法如下：

```
try(物品 i，当前选择已达到的重量和，本方案可能达到的总价值 tv)
{   /*考虑物品 i 包含在当前方案中的可能性*/
  if(包含物品 i 是可以接受的)
  {   将物品 i 包含在当前方案中；
    if (i<n-1)
      try(i+1,tw+物品 i 的重量,tv);
    else
      /*又一个完整方案，因为它比前面的方案好，以它作为最佳方案*/
以当前方案作为临时最佳方案保存;
      恢复物品 i 不包含状态；

    }
    /*考虑物品 i 不包含在当前方案中的可能性*/
    if (不包含物品 i 仅是可男考虑的)
      if (i<n-1)
        try(i+1,tw,tv-物品 i 的价值)；
      else
        /*又一个完整方案，因它比前面的方案好，以它作为最佳方案*/
以当前方案作为临时最佳方案保存;
  }
```

为了理解上述算法，特举以下实例。设有 4 件物品，它们的重量和价值见表：

物品	0	1	2	3
重量	5	3	2	1
价值	4	4	3	1

写一个能解决背包问题的程序,任意给定背包的容量以及一系列物品的重量,设把这些重量值存在一个数组中.

假定背包重 20 磅,有 5 个要以选择放入的数据项,它们的重量依次为 11 磅,8 磅,7 磅,6 磅,5 磅.

解答如下:

```
/*Knapsick.java*/

 public class Knapsack {
    private static int[] heft = {11, 8, 7, 6, 5};   //重量值
    public static int knapsick(int index, int heftSum) {
        if (index > heft.length - 1) {
            return 0;
        }
        if(heftSum >= heft[index]) {
            int result = knapsick(index + 1, heftSum - heft[index]);
            if (result == heftSum - heft[index]) {
                System.out.print(heft[index] + " ");
                return heftSum;
            }
        }
        return knapsick(index + 1, heftSum);
    }

    public static void main(String args[]) {
        knapsick(0, 20);
    }
}
```

non-递归算法学习系列之经典背包问题

1.引子

我们人类是一种贪婪的动物,如果给您一个容量一定的背包和一些大小不一的物品，装到背包里面的物品就归您，遇到这种好事大家一定不会错过，用力塞不一定是最好的办法，用脑子才行，下面就教您如何解决这样的问题，以获得更多的奖品。

2.应用场景

在一个物品向量中找到一个子集满足条件如下 ：

1）这个子集加起来的体积大小不能大于指定阀值

2） 这个物品子集加起来价值大小是向量V中所有满足条件1的子集中最大的

3.分析

背包问题有好多版本，本文只研究0/1版本，即对一个物体要么选用，要么就抛弃，不能将一个物体再继续细分的情况。这种问题最简单的方法就是找出这个向量 的所有子集，如同找出幂集中的子集一样，但这种遍历的方法恐怕并不会被聪明的我们所使用，现在举办这些活动的电视台也非常聪明，他们不但要求您能将物品装 进去，而且指定操作时间，这样当您慢慢腾腾的装进去倒出来的时候，时间恐怕早就到了，最终您可能一无所获，这可不是我们希望的结果，我们需要使用一些策 略：第一次我们可以从大小小于背包容量的物品中随意挑取一个，这样可以尽量争取时间，选取第一个后的每一个我们希望其都是最优的，这样能节省一定的时间。 假设有这么一组物品，其大小和价值如下表所示：

物品编号	大小	价值
1	2	1
2	3	4
3	4	3
4	5	6
5	6	8

给我们一个capcity为12的背包，让我们装上面这些物品，我们可以用下面的方法来解决寻找最优组合的问题

建立一个二围数组，数组包括n个行(n为物品数量)和capcity+1列

首先我们对第一个物品进行取舍，因为物品1大小为2，先将物品1加入背包，物品1的大小为2，则cap>=2的时候能容纳item1，这时候背包里面物品的价值为item1.Value=1,得到以下数组

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	1	1	1	1	1	1	1	1	1	1

接下来处理物品1和物品2的子集，item2的大小为3，则只有cap=3的时候才能容纳item2,当cap=3的时候讲好能容纳item2,此 时背包里面价值item2.value=4,且剩余空间为0，当cap=4的时候，能容纳item2，且剩余空间为1,不能容item1,当cap=5的 时候,可以容纳item1+item2,此时的价值为1+4=5，得到第二行

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	4	4	5	5	5	5	5	5	5	5

下面分析物品三，物品二，物品一的子集，物品三的大小为 4，当 cap=4 的时候就能容纳 item3,但此时背包里面的价值为 3，明显小于上一行中的 cap=4 的价值（3<4),所以 cap=4 时不能将 item3 放进去，所以第三行的 4 位置应该和第二行的 4 位置一致，当 cap=5 的时候能够容纳 item3,且剩余空间为 1，和 cap=4 情况一样,拷贝上一行同一位置的值,当 cap=6,放置 item3 后剩余 2,能容 item1 和 item4,二者 的总价值:1+3=4<5，故拷贝上一行同位置的值，cap=7 的时候，能容 item2+item3，总价值大小为 7,大于>5，故 cap= 8 的时的值为 7，cap=9 的时候仍能容难 item3+item2,value=7，cap=8 的时候，能容纳 item1+item2+item3,且总 价值大小为 8，大于上一行同位置的值，故 cap>=9 时候，总价值大小为 8,第三行：

0	1	2	3	4	5	6	7	8	9	10	11	12
0	0	1	4	4	5	5	7	7	8	8	8	8

按照这样的逻辑可以得到下面两列，最后二围数组是

	0	1	2	3	4	5	6	7	8	9	10	11	12
1	0	0	1	1	1	1	1	1	1	1	1	1	1
2	0	0	1	4	4	5	5	5	5	5	5	5	5
3	0	0	1	4	4	5	5	7	7	8	8	8	8
4	0	0	1	4	4	6	6	7	10	10	11	11	13
5	0	0	1	4	4	6	8	8	10	12	12	14	14

得到这样的数组之后，我们需要作的是根据这个二围数组来产生最优物品子集，方法为

从第 len 行开始,比较最后一行 cap 索引位置的值是否大于上一行同一位置的值，如先比较第五行位置 12 的值（14）与第四行位置 12 的值（13)，因为 14!=13，所以 item5 放置到最优集合中，item5 的大小为 6，故比较第四行 cap-6=6 的位置上的值与上一行同一位置上值得大 小，因为 6！= 5,所以 item4 能放置到最优集合，下一步要比较的位置 cap = 6-item4.Size=6-5=1,第三行位置 1 与第二行位置 1 相同，故 item3 不能放置到最优集合，第二行和第一行第一个位置上的值也一样，所以 item2 也不能放置进去，最后判断 item1 是否应该在最优集合，item5+item4 后，剩余空间为 1，不能容纳 item1,故最优集合为 {item4,item5};

综合上面的分析，我们可以得到这样的一个处理流程

1）首先建立一个 nx(cap+1)的二围数组

2）第一行从尝试选择第一个物品开始

3）对于以后的行，对于每个容量 1<=cap<=capacity，首先拷贝上一行同一位置的值下来，如果 itemi.Size<=cap 并且上一行(cap-itemi.Size)位置上的值与 itemi.Value 的 和(tempMax)大于拷贝下来的值的话，就将拷贝下来的值替换为上一行(cap-itemi.Size)位置上的值与 itemi.Value 的 和（tempMax）

4）得到完整数组之后，我们既可以根据数组来确定最优集合了，首先从最后一样最后位置开始，和上一行的同一位置进行比较，如果相同，则该行对应索引的物品不能 放到背包中，否则放到背包，并且开始比较上一行与 上上一行在当前背包剩余空间索引出的值，如不等，则对应物品可放置，如此，直到处理到第二行和第一行的比对完成，然后根据当前背包剩余容量与第一个物品的 大小比对来确定物品一是否能放置到背包中

数据结构经典问题和算法分析（五）回溯法

来源:　作者:　2007-5-30 21:43:35　字体:[大 中 小]

五、回溯法

回溯法也称为试探法，该方法首先暂时放弃关于问题规模大小的限制，并将问题的候选解按某种顺序逐一枚举和检验。当发现当前候选解不可能是解时，就选择下一个候选解；倘若当前候选解除了还不满足问题规模要求外，满足所有其他要求时，继续扩大当前候选解的规模，并继续试探。如果当前候选解满足包括问题规模在内的所有要求时，该候选解就是问题的一个解。在回溯法中，放弃当前候选解，寻找下一个候选解的过程称为回溯。扩大当前候选解的规模，以继续试探的过程称为向前试探。

1、回溯法的一般描述

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（x1，x2，…，xn）组成的一个状态空间 E={（x1，x2，…，xn） | xi∈Si ，i=1，2，…，n}，给定关于 n 元组中的一个分量的一个约束集 D，要求 E 中满足 D 的全部约束条件的所有 n 元组。其中 Si 是分量 xi 的定义域，且 |Si| 有限，i=1，2，…，n。我们称 E 中满足 D 的全部约束条件的任一 n 元组为问题 P 的一个解。

解问题 P 的最朴素的方法就是枚举法，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，所给定的约束集 D 具有完备性，即 i 元组（x1，x2，…，xi）满足 D 中仅涉及到 x1，x2，…，xi 的所有约束意味着 j（j<i）元组（x1，x2，…，xj）一定也满足 D 中仅涉及到 x1，x2，…，xj 的所有约束，i=1，2，…，n。换句话说，只要存在 0≤j≤n-1，使得（x1，x2，…，xj）违反 D 中仅涉及到 x1，x2，…，xj 的约束之一，则以（x1，x2，…，xj）为前缀的任何 n 元组（x1，x2，…，xj，xj+1，…，xn）一定也违反 D 中仅涉及到 x1，x2，…，xi 的一个约束，n≥i>j。因此，对于约束集 D 具有完备性的问题 P，一旦检测断定某个 j 元组（x1，x2，…，xj）违反 D 中仅涉及 x1，x2，…，xj 的一个约束，就可以肯定，以（x1，x2，…，xj）为前缀的任何 n 元组（x1，x2，…，xj，xj+1，…，xn）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枚举法效率更高的算法。

回溯法首先将问题 P 的 n 元组的状态空间 E 表示成一棵高为 n 的带权有序树 T，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。树 T 类似于检索树，它可以这样构造：

设 Si 中的元素可排成 xi(1) ，xi(2) ，…，xi(mi-1) ，|Si| =mi，i=1，2，…，n。从根开始，让 T 的第 I 层的每一个结点都有 mi 个儿子。这 mi 个儿子到它们的双亲的边，按从左到右的次序，分别带权 xi+1(1) ，xi+1(2) ，…，xi+1(mi) ，i=0，1，2，…，n-1。照这种构造方式，E 中的一个 n 元组（x1，x2，…，xn）对应于 T 中的一个叶子结点，T 的根到这个叶子结点的路径上依次的 n 条边的权分别为 x1，x2，…，xn，反之亦然。另外，对于任意的 0≤i≤n-1，E 中 n 元组（x1，x2，…，xn）的一个前缀 I 元组（x1，x2，…，xi）对应于 T 中的一个非叶子结点，T 的根到这个非叶子结点的路径上依次的 I 条边的权分别为 x1，x2，…，xi，反之亦然。特别，E 中的任意一个 n 元组的空前缀（），对应于 T 的根。

因而，在 E 中寻找问题 P 的一个解等价于在 T 中搜索一个叶子结点，要求从 T 的根到该叶子结点的路径上依次的 n 条边相应带的 n 个权 x1，x2，…，xn 满足约束集 D 的全部约束。在 T 中搜索所要求的叶子结点，很自然的一种方式是从根出发，按深度优先的策略逐步深入，即依次搜索满足约束条件的前缀 1 元组（x1i）、前缀 2 元组（x1，x2）、…，前缀 I 元组（x1，x2，…，xi），…，直到 i=n 为止。

在回溯法中，上述引入的树被称为问题 P 的状态空间树；树 T 上任意一个结点被称为问题 P 的状态结点；树 T 上的任意一个叶子结点被称为问题 P 的一个解状态结点；树 T 上满足约束集 D 的全部约束的任意一个叶子结点被称为问题 P 的一个回答状态结点，它对应于问题 P 的一个解。

【问题】 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。

例如 n=5，r=3 的所有组合为：

（1）1、2、3 （2）1、2、4 （3）1、2、5

（4）1、3、4 （5）1、3、5 （6）1、4、5

（7）2、3、4 （8）2、3、5 （9）2、4、5

（10）3、4、5

则该问题的状态空间为：

E={（x1，x2，x3）丨xi∈S ，i=1，2，3 } 其中：S={1，2，3，4，5}

约束集为： x1<x2<x3

显然该约束集具有完备性。

问题的状态空间树 T：

2、回溯法的方法

对于具有完备约束集 D 的一般问题 P 及其相应的状态空间树 T，利用 T 的层次结构和 D 的完备性，在 T 中搜索问题 P 的所有解的回溯法可以形象地描述为：

从 T 的根出发，按深度优先的策略，系统地搜索以其为根的子树中可能包含着回答结点的所有状态结点，而跳过对肯定不含回答结点的所有子树的搜索，以提高搜索效率。具体地说，当搜索按深度优先策略到达一个满足 D 中所有有关约束的状态结点时，即“激活”该状态结点，以便继续往深层搜索；否则跳过对以该状态结点为根的子树的搜索，而一边逐层地向该状态结点的祖先结点回溯，一边“杀死”其儿子结点已被搜索遍的祖先结点，直到遇到其儿子结点未被搜索遍的祖先结点，即转向其未被搜索的一个儿子结点继续搜索。

在搜索过程中，只要所激活的状态结点又满足终结条件，那么它就是回答结点，应该把它输出或保存。由于在回溯法求解问题时，一般要求出问题的所有解，因此在得到回答结点后，同时也要进行回溯，以便得到问题的其他解，直至回溯到 T 的根且根的所有儿子结点均已被搜索过为止。

例如在组合问题中，从 T 的根出发深度优先遍历该树。当遍历到结点（1，2）时，虽然它满足约束条件，但还不是回答结点，则应继续深度遍历；当遍历到叶子结点（1，2，5）时，由于它已是一个回答结点，则保存（或输出）该结点，并回溯到其双亲结点，继续深度遍历；当遍历到结点（1，5）时，由于它已是叶子结点，但不满足约束条件，故也需回溯。

3、回溯法的一般流程和技术

在用回溯法求解有关问题的过程中，一般是一边建树，一边遍历该树。在回溯法中我们一般采用非递归方法。下面，我们给出回溯法的非递归算法的一般流程：

在用回溯法求解问题，也即在遍历状态空间树的过程中，如果采用非递归方法，则我们一般要用到栈的数据结构。这时，不仅可以用栈来表示正在遍历的树的结点，而且可以很方便地表示建立孩子结点和回溯过程。

例如在组合问题中，我们用一个一维数组 Stack[ ]表示栈。开始栈空，则表示了树的根结点。如果元素 1 进栈，则表示建立并遍历（1）结点；这时如果元素 2 进栈，则表示建立并遍历（1，2）结点；元素 3 再进栈，则表示建立并遍历（1，2，3）结点。这时可以判断它满足所有约束条件，是问题的一个解，输出（或保存）。这时只要栈顶元素（3）出栈，即表示从结点（1，2，3）回溯到结点（1，2）。

【问题】 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顺序存于 a[0]，a[1]，…，a[r-1]中，组合

的元素满足以下性质：
（1） a[i+1]>a[i]，后一个数字比前一个大；
（2） a[i]-i<=n-r+1。
按回溯法的思想，找解过程可以叙述如下：
首先放弃组合数个数为 r 的条件，候选组合从只有一个数字 1 开始。因该候选解满足除问题规模之外的全部条件，扩大其规模，并使其满足上述条件（1），候选组合改为 1，2。继续这一过程，得到候选组合 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。在该解的基础上，选下一个候选解，因 a[2]上的 3 调整为 4，以及以后调整为 5 都满足问题的全部要求，得到解 1，2，4 和 1，2，5。由于对 5 不能再作调整，就要从 a[2]回溯到 a[1]，这时，a[1]=2，可以调整为 3，并向前试探，得到解 1，3，4。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。按上述思想写成程序如下：
【程序】

```
# define MAXN 100
int a[MAXN];
void comb(int m,int r)
{ int i,j;
 i=0;
 a[i]=1;
 do {
 if (a[i]-i<=m-r+1
 { if (i==r-1)
 { for (j=0;j<r;j++)
 printf("%4d",a[j]);
 printf("\n");
 }
 a[i]++;
 continue;
 }
 else
 { if (i==0)
 return;
 a[--i]++;
 }
 } while (1)
}

main()
{ comb(5,3);
}
```

【问题】 填字游戏

问题描述：在 3×3 个方格的方阵中要填入数字 1 到 N（N≥10）内的某 9 个数字，每个方格填一个整数，似的所有相邻两个方格内的两个整数之和为质数。试求出所有满足这个要求的各种数字填法。

可用试探发找到问题的解，即从第一个方格开始，为当前方格寻找一个合理的整数填入，并在当前位置正确填入后，为下一方格寻找可填入的合理整数。如不能为当前方格找到一个合理的可填证书，就要回退到前一方格，调整前一方格的填入数。当第九个方格也填入合理的整数后，就找到了一个解，将该解输出，并调整第九个的填入的整数，寻找下一个解。

为找到一个满足要求的 9 个数的填法，从还未填一个数开始，按某种顺序（如从小到大的顺序）每次在当前位置填入一个整数，然后检查当前填入的整数是否能满足要求。在满足要求的情况下，继续用同样的方法为下一方格填入整数。如果最近填入的整数不能满足要求，就改变填入的整数。如对当前方格试尽所有可能的整数，都不能满足要求，就得回退到前一方格，并调整前一方格填入的整数。如此重复执行扩展、检查或调整、检查，直到找到一个满足问题要求的解，将解输出。

回溯法找一个解的算法：

```
{ int m=0,ok=1;
 int n=8;
 do{
 if (ok) 扩展;
 else 调整;
 ok=检查前 m 个整数填放的合理性;
 } while ((!ok||m!=n)&&(m!=0))
 if (m!=0) 输出解;
 else 输出无解报告；
}
```

如果程序要找全部解，则在将找到的解输出后，应继续调整最后位置上填放的整数，试图去找下一个解。相应的算法如下：

回溯法找全部解的算法：

```
{ int m=0,ok=1;
 int n=8;
 do{
 if (ok)
{ if (m==n)
{ 输出解；
调整；
}
else 扩展;
 }
 else 调整;
```

```
 ok=检查前 m 个整数填放的合理性;
 } while (m!=0);
}
```

为了确保程序能够终止，调整时必须保证曾被放弃过的填数序列不会再次实验，即要求按某种有许模型生成填数序列。给解的候选者设定一个被检验的顺序，按这个顺序逐一形成候选者并检验。从小到大或从大到小，都是可以采用的方法。如扩展时，先在新位置填入整数 1，调整时，找当前候选解中下一个还未被使用过的整数。将上述扩展、调整、检验都编写成程序，细节见以下找全部解的程序。

【程序】

```
# include <stdio.h>
# define N 12
void write(int a[ ])
{ int i,j;
 for (i=0;i<3;i++)
 { for (j=0;j<3;j++)
 printf("%3d",a[3*i+j]);
 printf("\n");
 }
 scanf("%*c");
}

int b[N+1];
int a[10];
int isprime(int m)
{ int i;
 int primes[ ]={2,3,5,7,11,17,19,23,29,-1};
 if (m==1||m%2=0) return 0;
 for (i=0;primes[i]>0;i++)
 if (m==primes[i]) return 1;
 for (i=3;i*i<=m;)
 { if (m%i==0) return 0;
 i+=2;
 }
 return 1;
}


int checkmatrix[ ][3]={ {-1},{0,-1},{1,-1},{0,-1},{1,3,-1},
 {2,4,-1},{3,-1},{4,6,-1},{5,7,-1}};
int selectnum(int start)
```

```
{ int j;
 for (j=start;j<=N;j++)
 if (b[j]) return j
 return 0;
}

int check(int pos)
{ int i,j;
 if (pos<0) return 0;
 for (i=0;(j=checkmatrix[pos][i])>=0;i++)
 if (!isprime(a[pos]+a[j])
 return 0;
 return 1;
}

int extend(int pos)
{ a[++pos]=selectnum(1);
 b[a][pos]]=0;
 return pos;
}

int change(int pos)
{ int j;
 while (pos>=0&&(j=selectnum(a[pos]+1))==0)
 b[a[pos--]]=1;
 if (pos<0) return –1
 b[a[pos]]=1;
 a[pos]=j;
 b[j]=0;
 return pos;
}

void find()
{ int ok=0,pos=0;
 a[pos]=1;
 b[a[pos]]=0;
 do {
 if (ok)
 if (pos==8)
 { write(a);
```

```
 pos=change(pos);
 }
 else pos=extend(pos);
 else pos=change(pos);
 ok=check(pos);
 } while (pos>=0)
}

void main()
{ int i;
 for (i=1;i<=N;i++)
 b[i]=1;
 find();
}
```

【问题】 n 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 × × × × ×
 × × ×
× × ×
 × ×
 × ×
```

从图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。得到求解皇后问题的算法如下：

```
{ 输入棋盘大小值 n；
 m=0;
 good=1;
```

```
do {
if (good)
if (m==n)
{ 输出解；
改变之，形成下一个候选解;
}
else 扩展当前候选接至下一列；
else 改变之，形成下一个候选解；
good=检查当前候选解的合理性；
} while (m!=0);
}
```

在编写程序之前，先确定边式棋盘的数据结构。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，引入以下三个工作数组：

（1） 数组 a[ ]，a[k]表示第 k 行上还没有皇后；

（2） 数组 b[ ]，b[k]表示第 k 列右高左低斜线上没有皇后；

（3） 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，他们的行号与列号之和相同；同一左高右低斜线上的方格，他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，准备考察第 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。细节见以下程序：

【程序】

```
# include <stdio.h>
# include <stdlib.h>
# define MAXN 20
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;
 for (j=0;j<=2*n;j++) cb[j]=c[j]=1;
 m=1; col[1]=1; good=1; col[0]=0;
 do {
 if (good)
 if (m==n)
 { printf(“列\t 行”);
 for (j=1;j<=n;j++)
 printf(“%3d\t%d\n”,j,col[j]);
 printf(“Enter a character (Q/q for exit)!\n”);
 scanf(“%c”,&awn);
 if (awn==’Q’||awn==’q’) exit(0);
 while (col[m]==n)
 { m--;
 a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1;
 }
 col[m]++;
 }
 else
 { a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=0;
 col[++m]=1;
 }
 else
 { while (col[m]==n)
 { m--;
 a[col[m]]=b[m+col[m]]=c[n+m-col[m]]=1;
 }
 col[m]++;
 }
 good=a[col[m]]&&b[m+col[m]]&&c[n+m-col[m]];
 } while (m!=0);
}
```

试探法找解算法也常常被编写成递归函数，下面两程序中的函数 queen_all()和函数 queen_one()能分别用来解皇后问题的全部解和一个解。

【程序】

```
# include <stdio.h>
# include <stdlib.h>
# define MAXN 20
```

```
int n;
int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];

void main()
{ int j;
 printf("Enter n: "); scanf("%d",&n);
 for (j=0;j<=n;j++) a[j]=1;
 for (j=0;j<=2*n;j++) cb[j]=c[j]=1;
 queen_all(1,n);
}

void queen_all(int k,int n)
{ int i,j;
 char awn;
 for (i=1;i<=n;i++)
 if (a[i]&&b[k+i]&&c[n+k-i])
 { col[k]=i;
 a[i]=b[k+i]=c[n+k-i]=0;
 if (k==n)
 { printf("列\t 行");
 for (j=1;j<=n;j++)
 printf("%3d\t%d\n",j,col[j]);
 printf("Enter a character (Q/q for exit)!\n");
 scanf("%c",&awn);
 if (awn=='Q'||awn=='q') exit(0);

 }
 queen_all(k+1,n);
 a[i]=b[k+i]=c[n+k-i];
 }
}
```

采用递归方法找一个解与找全部解稍有不同，在找一个解的算法中，递归算法要对当前候选解最终是否能成为解要有回答。当它成为最终解时，递归函数就不再递归试探，立即返回；若不能成为解，就得继续试探。设函数 queen_one()返回 1 表示找到解，返回 0 表示当前候选解不能成为解。细节见以下函数。

【程序】

```
# define MAXN 20
 int n;
 int col[MAXN+1],a[MAXN+1],b[2*MAXN+1],c[2*MAXN+1];
 int queen_one(int k,int n)
```

```
{ int i,found;
i=found=0;
While (!found&&i<n)
{ i++;
if (a[i]&&b[k+i]&&c[n+k-i])
{ col[k]=i;
a[i]=b[k+i]=c[n+k-i]=0;
if (k==n) return 1;
else
found=queen_one(k+1,n);
a[i]=b[k+i]=c[n+k-i]=1;
}
}
return found;
}
```

数据结构经典问题和算法分析（六）贪婪法

来源:　作者:　2007-5-30 21:47:46　字体:[大 中 小]

六、贪婪法

贪婪法是一种不追求最优解，只希望得到较为满意解的方法。贪婪法一般可以快速得到满意的解，因为它省去了为找最优解要穷尽所有可能而必须耗费的大量时间。贪婪法常以当前情况为基础作最优选择，而不考虑各种可能的整体情况，所以贪婪法不要回溯。

例如平时购物找钱时，为使找回的零钱的硬币数最少，不考虑找零钱的所有各种发表方案，而是从最大面值的币种开始，按递减的顺序考虑各币种，先尽量用大面值的币种，当不足大面值币种的金额时才去考虑下一种较小面值的币种。这就是在使用贪婪法。这种方法在这里总是最优，是因为银行对其发行的硬币种类和硬币面值的巧妙安排。如只有面值分别为 1、5 和 11 单位的硬币，而希望找回总额为 15 单位的硬币。按贪婪算法，应找 1 个 11 单位面值的硬币和 4 个 1 单位面值的硬币，共找回 5 个硬币。但最优的解应是 3 个 5 单位面值的硬币。

【问题】 装箱问题

问题描述：装箱问题可简述如下：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。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。

若考察将 n 种物品的集合分划成 n 个或小于 n 个物品的所有子集，最优解就可以找到。但所有可能划分的总数太大。对适当大的 n，找出所有可能的划分要花费的时间是无法承受的。为此，对装箱问题采用非常简单的近似算法，即贪婪法。该算法依次将物品放到它第一个能放进去的箱子中，该算法虽不能保证找到最优解，但还是能找到非常好的解。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。装箱算法简单描述如下：

```
{ 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）
 { 另用一个箱子，并将物品 i 放入该箱子；
 box_count++；
 }
 else
 将物品 i 放入箱子 j；
 }
}
```

上述算法能求出需要的箱子数 box_count，并能求出各箱子所装物品。下面的例子说明该算法不一定能找到最优解，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，需三只箱子，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。

若每只箱子所装物品用链表来表示，链表首结点指针存于一个结构中，结构记录尚剩余的空间量和该箱子所装物品链表的首指针。另将全部箱子的信息也构成链表。以下是按以上算法编写的程序。

【程序】

```
# include <stdio.h>
# include <stdlib.h>
typedef struct ele
{ int vno;
 struct ele *link;
} ELE;
typedef struct hnode
{ int remainder;
 ELE *head;
 Struct hnode *next;
} 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);

 Box_h=box_t=NULL;
 Box_count=0;
 For (i=0;i<n;i++)
 { p=(ELE *)malloc(sizeof(ELE));
 p->vno=i;
 for (j=box_h;j!=NULL;j=j->next)
 if (j->remainder>=a[i]) break;
 if (j==NULL)
 { j=(HNODE *)malloc(sizeof(HNODE));
 j->remainder=box_volume-a[i];
 j->head=NULL;
 if (box_h==NULL) box_h=box_t=j;
 else box_t=boix_t->next=j;
 j->next=NULL;
 box_count++;
 }
 else j->remainder-=a[i];
 for (q=j->next;q!=NULL&&q->link!=NULL;q=q->link);
 if (q==NULL)
 { p->link=j->head;
 j->head=p;
 }
 else
 { p->link=NULL;
 q->link=p;
 }
 }
 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++)
 { 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”);
 }
}
```

【问题】 马的遍历

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。

马在某个方格，可以在一步内到达的不同位置最多有 8 个，如图所示。如用二维数组 board[ ][ ]表示棋盘，其元素记录马经过该位置时的步骤号。另对马的 8 种可能走法（称为着法）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）、（i+2，j-1），实际可以走的位置尽限于还未走过的和不越出边界的那些位置。为便于程序的同意处理，可以引入两个数组，分别存储各种可能走法对当前位置的纵横增量。

```
4 3
5 2
 马
6 1
 7 0
```

对于本题，一般可以采用回溯法，这里采用 Warnsdoff 策略求解，这也是一种贪婪法，其选择下一出口的贪婪标准是在那些允许走的位置中，选择出口最少的那个位置。如马的当前位置（i，j）只有三个出口，他们是位置（i+2，j+1）、（i-2，j+1）和（i-1，j-2），如分别走到这些位置，这三个位置又分别会有不同的出口，假定这三个位置的出口个数分别为 4、2、3，则程序就选择让马走向（i-2，j+1）位置。

由于程序采用的是一种贪婪法，整个找解过程是一直向前，没有回溯，所以能非常快地找到解。但是，对于某些开始位置，实际上有解，而该算法不能找到解。对于找不到解的情况，程序只要改变 8 种可能出口的选择顺序，就能找到解。改变出口选择顺序，就是改变有相同出口时的选择标准。以下程序考虑到这种情况，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。细节以下程序。

【程序】

```
# include <stdio.h>
int delta_i[ ]={2,1,-1,-2,-2,-1,1,2};
int delta_j[ ]={1,2,2,1,-1,-2,-2,-1};
int board[8][8];
int exitn(int i,int j,int s,int a[ ])
```

```
{ int i1,j1,k,count;
 for (count=k=0;k<8;k++)
 { i1=i+delta_i[(s+k)%8];
 j1=i+delta_j[(s+k)%8];
 if (i1>=0&&i1<8&&j1>=0&&j1<8&&board[I1][j1]==0)
 a[count++]=(s+k)%8;
 }
 return count;
}

int next(int i,int j,int s)
{ int m,k,mm,min,a[8],b[8],temp;
 m=exitn(i,j,s,a);
 if (m==0) return –1;
 for (min=9,k=0;k<m;k++)
 { temp=exitn(I+delta_i[a[k]],j+delta_j[a[k]],s,b);
 if (temp<min)
 { min=temp;
kk=a[k];
 }


 }
 return kk;
}

void main()
{ int sx,sy,i,j,step,no,start;
 for (sx=0;sx<8;sx++)
 for (sy=0;sy<8;sy++)
 { start=0;
 do {
 for (i=0;i<8;i++)
 for (j=0;j<8;j++)
 board[i][j]=0;
 board[sx][sy]=1;
 I=sx; j=sy;
 For (step=2;step<64;step++)
 { if ((no=next(i,j,start))==-1) break;
 I+=delta_i[no];
```

```
j+=delta_j[no];
board[i][j]=step;
}
if (step>64) break;
start++;
} while(step<=64)
for (i=0;i<8;i++)
{ for (j=0;j<8;j++)
printf("%4d",board[i][j]);
printf("\n\n");
}
scanf("%*c");
}
}
```

据结构经典问题和算法分析（七）分治法

来源:　作者:　2007-5-30 21:49:11　字体:[大 中 小]

七、分治法

1、分治法的基本思想

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 N 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 时只要作 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。

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。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。

2、分治法的适用条件

分治法所能解决的问题一般具有以下几个特征：

（1）该问题的规模缩小到一定的程度就可以容易地解决；

（2）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；

（3）利用该问题分解出的子问题的解可以合并为该问题的解；

（4）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；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，此时虽然可用分治法，但一般用动态规划法较好。
3、分治法的基本步骤
分治法在每一层递归上都有三个步骤：
（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。
它的一般的算法设计模式如下：
Divide_and_Conquer（P）
if |P|≤n0
then return（ADHOC（P））
将 P 分解为较小的子问题 P1、P2、…、Pk
for i←1 to k
do
yi ← Divide-and-Conquer（Pi） △ 递归解决 Pi
T ← MERGE（y1，y2，…，yk） △ 合并子问题
Return（T）
其中 |P| 表示问题 P 的规模；n0 为一阈值，表示当问题 P 的规模不超过 n0 时，问题已容易直接解出，不必再继续分解。ADHOC（P）是该分治法中的基本子算法，用于直接解小规模的问题 P。因此，当 P 的规模不超过 n0 时，直接用算法 ADHOC（P）求解。
算法 MERGE（y1，y2，…，yk）是该分治法中的合并子算法，用于将 P 的子问题 P1、P2、…、Pk 的相应的解 y1、y2、…、yk 合并为 P 的解。
根据分治法的分割原则，原问题应该分为多少个子问题才较适宜？各个子问题的规模应该怎样才为适当？这些问题很难予以肯定的回答。但人们从大量实践中发现，在用分治法设计算法时，最好使子问题的规模大致相同。换句话说，将一个问题分成大小相等的 k 个子问题的处理方法是行之有效的。许多问题可以取 k=2。这种使子问题规模大致相等的做法是出自一种平衡子问题的思想，它几乎总是比子问题规模不等的做法要好。
分治法的合并步骤是算法的关键所在。有些问题的合并方法比较明显，有些问题合并方法比较复杂，或者是有多种合并方案；或者是合并方案不明显。究竟应该怎样合并，没有统一的模式，需要具体问题具体分析。
【问题】 大整数乘法
问题描述：
通常，在分析一个算法的计算复杂性时，都将加法和乘法运算当作是基本运算来处理，即将执行一次加法或乘法运算所需的计算时间当作一个仅取决于计算机硬件处理速度的常数。这个假定仅在计算机硬件能对参加运算的整数直接表示和处理时才是合理的。然而，在某些

情况下，我们要处理很大的整数，它无法在计算机硬件能直接表示的范围内进行处理。若用浮点数来表示它，则只能近似地表示它的大小，计算结果中的有效数字也受到限制。若要精确地表示大整数并在计算结果中要求精确地得到所有位数上的数字，就必须用软件的方法来实现大整数的算术运算。

请设计一个有效的算法，可以进行两个 n 位大整数的乘法运算。

设 X 和 Y 都是 n 位的二进制整数，现在要计算它们的乘积 XY。我们可以用小学所学的方法来设计一个计算乘积 XY 的算法，但是这样做计算步骤太多，显得效率较低。如果将每 2 个 1 位数的乘法或加法看作一步运算，那么这种方法要作 O(n2)步运算才能求出乘积 XY。下面我们用分治法来设计一个更有效的大整数乘积算法。

图 6-3 大整数 X 和 Y 的分段

我们将 n 位的二进制整数 X 和 Y 各分为 2 段，每段的长为 n/2 位（为简单起见，假设 n 是 2 的幂），如图 6-3 所示。

由此，X=A2n/2+B，Y=C2n/2+D。这样，X 和 Y 的乘积为：

XY=(A2n/2+B)(C2n/2+D)=AC2n+(AD+CB)2n/2+BD （1）

如果按式(1)计算 XY，则我们必须进行 4 次 n/2 位整数的乘法(AC，AD，BC 和 BD)，以及 3 次不超过 n 位的整数加法（分别对应于式（1）中的加号），此外还要做 2 次移位（分别对应于式（1）中乘 2n 和乘 2n/2）。所有这些加法和移位共用 O（n）步运算。设 T（n）是 2 个 n 位整数相乘所需的运算总数，则由式（1），我们有：

（2）

由此可得 T（n）=O（n2）。因此，用（1）式来计算 X 和 Y 的乘积并不比小学生的方法更有效。要想改进算法的计算复杂性，必须减少乘法次数。为此我们把 XY 写成另一种形式：

XY=AC2n+[(A-B)(D-C)+AC+BD]2n/2+BD （3）

虽然，式（3）看起来比式（1）复杂些，但它仅需做 3 次 n/2 位整数的乘法（AC，BD 和（A-B）（D-C）），6 次加、减法和 2 次移位。由此可得：

（4）

用解递归方程的套用公式法马上可得其解为

T(n)=O(nlog3)=O(n1.59)。利用式（3），并考虑到 X 和 Y 的符号对结果的影响，我们给出大整数相乘的完整算法 MULT 如下：

```
function MULT(X，Y，n); {X 和 Y 为 2 个小于 2n 的整数，返回结果为 X 和 Y 的乘积 XY}
begin
S=SIGN(X)*SIGN(Y); {S 为 X 和 Y 的符号乘积}
X=ABS(X);
Y=ABS(Y); {X 和 Y 分别取绝对值}
if n=1 then
if (X=1)and(Y=1) then return(S)
else return(0)
else begin
```

A=X 的左边 n/2 位;
B=X 的右边 n/2 位;
C=Y 的左边 n/2 位;
D=Y 的右边 n/2 位;
ml=MULT(A,C,n/2);
m2=MULT(A-B,D-C,n/2);
m3=MULT(B,D,n/2);
S=S*(m1*2n+(m1+m2+m3)*2n/2+m3);
return(S);
end;
end;
上述二进制大整数乘法同样可应用于十进制大整数的乘法以提高乘法的效率减少乘法次数。
【问题】 最接近点对问题
问题描述：
在应用中，常用诸如点、圆等简单的几何对象代表现实世界中的实体。在涉及这些几何对象的问题中，常需要了解其邻域中其他几何对象的信息。例如，在空中交通控制问题中，若将飞机作为空间中移动的一个点来看待，则具有最大碰撞危险的 2 架飞机，就是这个空间中最接近的一对点。这类问题是计算几何学中研究的基本问题之一。下面我们着重考虑平面上的最接近点对问题。
最接近点对问题的提法是：给定平面上 n 个点，找其中的一对点，使得在 n 个点的所有点对中，该点对的距离最小。
严格地说，最接近点对可能多于 1 对。为了简单起见，这里只限于找其中的一对。
这个问题很容易理解，似乎也不难解决。我们只要将每一点与其他 n-1 个点的距离算出，找出达到最小距离的两个点即可。然而，这样做效率太低，需要 O(n2)的计算时间。我们能否找到问题的一个 O (nlogn)算法。
这个问题显然满足分治法的第一个和第二个适用条件，我们考虑将所给的平面上 n 个点的集合 S 分成 2 个子集 S1 和 S2，每个子集中约有 n/2 个点，然后在每个子集中递归地求其最接近的点对。在这里，一个关键的问题是如何实现分治法中的合并步骤，即由 S1 和 S2 的最接近点对，如何求得原集合 S 中的最接近点对，因为 S1 和 S2 的最接近点对未必就是 S 的最接近点对。如果组成 S 的最接近点对的 2 个点都在 S1 中或都在 S2 中，则问题很容易解决。但是，如果这 2 个点分别在 S1 和 S2 中，则对于 S1 中任一点 p，S2 中最多只有 n/2 个点与它构成最接近点对的候选者，仍需做 n2/4 次计算和比较才能确定 S 的最接近点对。因此，依此思路，合并步骤耗时为 O(n2)。整个算法所需计算时间 T(n)应满足：

T(n)=2T(n/2)+O(n2)
它的解为 T(n)=O(n2)，即与合并步骤的耗时同阶，显示不出比用穷举的方法好。从解递归方程的套用公式法，我们看到问题出在合并

步骤耗时太多。这启发我们把注意力放在合并步骤上。
为了使问题易于理解和分析，我们先来考虑一维的情形。此时 S 中的 n 个点退化为 x 轴上的 n 个实数 x1、x2、…、xn。最接近点对即为这 n 个实数中相差最小的 2 个实数。我们显然可以先将 x1、x2、…、xn 排好序，然后，用一次线性扫描就可以找出最接近点对。这种方法主要计算时间花在排序上，因此如在排序算法中所证明的，耗时为 O(nlogn)。然而这种方法无法直接推广到二维的情形。因此，对这种一维的简单情形，我们还是尝试用分治法来求解，并希望能推广到二维的情形。

假设我们用 x 轴上某个点 m 将 S 划分为 2 个子集 S1 和 S2，使得 S1={x∈S | x≤m}；S2={x∈S | x>m}。这样一来，对于所有 p∈S1 和 q∈S2 有 p<q。
递归地在 S1 和 S2 上找出其最接近点对{p1，p2}和{q1，q2}，并设 δ=min{|p1-p2|，|q1-q2|}，S 中的最接近点对或者是{p1，p2}，或者是{q1，q2}，或者是某个{p3，q3}，其中 p3∈S1 且 q3∈S2。如图 1 所示。

图 1 一维情形的分治法
我们注意到，如果 S 的最接近点对是{p3，q3}，即 | p3-q3 | < δ，则 p3 和 q3 两者与 m 的距离不超过 δ，即 | p3-m | < δ，| q3-m | < δ，也就是说，p3∈(m-δ，m)，q3∈(m，m+δ)。由于在 S1 中，每个长度为 δ 的半闭区间至多包含一个点（否则必有两点距离小于 δ），并且 m 是 S1 和 S2 的分割点，因此(m-δ，m)中至多包含 S 中的一个点。同理，(m，m+δ)中也至多包含 S 中的一个点。由图 1 可以看出，如果(m-δ，m)中有 S 中的点，则此点就是 S1 中最大点。同理，如果(m，m+δ)中有 S 中的点，则此点就是 S2 中最小点。因此，我们用线性时间就能找到区间(m-δ，m)和(m，m+δ)中所有点，即 p3 和 q3。从而我们用线性时间就可以将 S1 的解和 S2 的解合并成为 S 的解。也就是说，按这种分治策略，合并步可在 O(n)时间内完成。这样是否就可以得到一个有效的算法了呢？

还有一个问题需要认真考虑，即分割点 m 的选取，及 S1 和 S2 的划分。选取分割点 m 的一个基本要求是由此导出集合 S 的一个线性分割，即 S=S1∪S2 ，S1∩S2=Φ，且
S1 {x | x≤m}；S2 {x | x>m}。容易看出，如果选取 m=[max（S）+min（S）]/2，可以满足线性分割的要求。选取分割点后，再用 O(n)时间即可将 S 划分成 S1={x∈S | x≤m}和 S2={x∈S | x>m}。然而，这样选取分割点 m，有可能造成划分出的子集 S1 和 S2 的不平衡。例如在最坏情况下，|S1|=1，|S2|=n-1，由此产生的分治法在最坏情况下所需的计算时间 T（n）应满足递归方程：
T（n）=T（n-1）+O（n）
它的解是 T（n）=O（n2）。这种效率降低的现象可以通过分治法中“平衡子问题”的方法加以解决。也就是说，我们可以通过适当选择分割点 m，使 S1 和 S2 中有大致相等个数的点。自然地，我们会想到用 S 的 n 个点的坐标的中位数来作分割点。在选择算法中介绍的选取中位数的线性时间算法使我们可以在 O（n）时间内确定一个平衡的分割点 m。

至此，我们可以设计出一个求一维点集 S 中最接近点对的距离的算法 pair 如下。

```
Float pair（S）;
{ if | S | =2 δ= | x[2]－x[1] | /*x[1..n]存放的是 S 中 n 个点的坐标*/
else
{ if ( | S | =1) δ=∞
 else
{ m=S 中各点的坐标值的中位数;
 构造 S1 和 S2，使 S1={x∈S | x≤m}，S2={x∈S | x＞m};
δ1=pair(S1);
 δ2=pair(S2);
 p=max(S1);
 q=min(S2);
 δ=min(δ1，δ2，q-p);
}
return(δ);
}
```

由以上的分析可知，该算法的分割步骤和合并步骤总共耗时 O(n)。因此，算法耗费的计算时间 T(n)满足递归方程：

解此递归方程可得 T(n)=O(nlogn)。

【问题】循环赛日程表

问题描述：设有 n=2k 个运动员要进行网球循环赛。现要设计一个满足以下要求的比赛日程表：

（1）每个选手必须与其他 n-1 个选手各赛一次；

（2）每个选手一天只能参赛一次；

（3）循环赛在 n-1 天内结束。

请按此要求将比赛日程表设计成有 n 行和 n-1 列的一个表。在表中的第 i 行，第 j 列处填入第 i 个选手在第 j 天所遇到的选手。其中 1≤i≤n，1≤j≤n-1。

按分治策略，我们可以将所有的选手分为两半，则 n 个选手的比赛日程表可以通过 n/2 个选手的比赛日程表来决定。递归地用这种一分为二的策略对选手进行划分，直到只剩下两个选手时，比赛日程表的制定就变得很简单。这时只要让这两个选手进行比赛就可以了。

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

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

1 2 3 4 1 2 7 8 5 6 3 2 1 4
2 1 4 3 2 1 8 7 6 5 4 3 2 1
（1） （2） （3）
图 1 2 个、4 个和 8 个选手的比赛日程表
图 1 所列出的正方形表（3）是 8 个选手的比赛日程表。其中左上角与左下角的两小块分别为选手 1 至选手 4 和选手 5 至选手 8 前 3 天的比赛日程。据此，将左上角小块中的所有数字按其相对位置抄到右下角，又将左下角小块中的所有数字按其相对位置抄到右上角，这样我们就分别安排好了选手 1 至选手 4 和选手 5 至选手 8 在后 4 天的比赛日程。依此思想容易将这个比赛日程表推广到具有任意多个选手的情形。

数据结构经典问题和算法分析（八）动态规划法

来源: 作者: 2007-5-30 21:53:24 字体:[大 中 小]

八、动态规划法
经常会遇到复杂问题不能简单地分解成几个子问题，而会分解出一系列的子问题。简单地采用把大问题分解成子问题，并综合子问题的解导出大问题的解的方法，问题求解耗时会按问题规模呈幂级数增加。
为了节约重复求相同子问题的时间，引入一个数组，不管它们是否对最终解有用，把所有子问题的解存于该数组中，这就是动态规划法所采用的基本方法。以下先用实例说明动态规划方法的使用。
【问题】 求两字符序列的最长公共字符子序列
问题描述：字符序列的子序列是指从给定字符序列中随意地（不一定连续）去掉若干个字符（可能一个也不去掉）后所形成的字符序列。令给定的字符序列 X="x0，x1，…，xm-1"，序列 Y="y0，y1，…，yk-1"是 X 的子序列，存在 X 的一个严格递增下标序列<i0，i1，…，ik-1>，使得对所有的 j=0，1，…，k-1，有 xij=yj。例如，X="ABCBDAB"，Y="BCDB"是 X 的一个子序列。
给定两个序列 A 和 B，称序列 Z 是 A 和 B 的公共子序列，是指 Z 同是 A 和 B 的子序列。问题要求已知两序列 A 和 B 的最长公共子序列。
如采用列举 A 的所有子序列，并一一检查其是否又是 B 的子序列，并随时记录所发现的子序列，最终求出最长公共子序列。这种方法因耗时太多而不可取。

考虑最长公共子序列问题如何分解成子问题，设 A="a0，a1，…，am-1"，B="b0，b1，…，bn1"，并 Z="z0，z1，…，zk-1"为它们的最长公共子序列。不难证明有以下性质：
（1） 如果 am-1=bn-1，则 zk-1=am-1=bn-1，且"z0，z1，…，zk-2"是"a0，a1，…，am-2"和"b0，b1，…，bn-2"的一个最长公共子序列；
（2） 如果 am-1!=bn-1，则若 zk-1!=am-1，蕴涵"z0，z1，…，zk-1"是"a0，a1，…，am-2"

和“b0，b1，…，bn-1”的一个最长公共子序列；
（3） 如果 am-1!=bn-1，则若 zk-1!=bn-1，蕴涵“z0，z1，…，zk-1”是“a0，a1，…，am-1”和“b0，b1，…，bn-2”的一个最长公共子序列。
这样，在找 A 和 B 的公共子序列时，如有 am-1=bn-1，则进一步解决一个子问题，找“a0，a1，…，am-2”和“b0，b1，…，bm-2”的一个最长公共子序列；如果 am-1!=bn-1，则要解决两个子问题，找出“a0，a1，…，am-2”和“b0，b1，…，bn-1”的一个最长公共子序列和找出“a0，a1，…，am-1”和“b0，b1，…，bn-2”的一个最长公共子序列，再取两者中较长者作为 A 和 B 的最长公共子序列。
定义 c[i][j]为序列“a0，a1，…，ai-2”和“b0，b1，…，bj-1”的最长公共子序列的长度，计算 c[i][j]可递归地表述如下：
（1）c[i][j]=0 如果 i=0 或 j=0；
（2）c[i][j]= c[i-1][j-1]+1 如果 I，j>0，且 a[i-1]=b[j-1]；
（3）c[i][j]=max（c[i][j-1]，c[i-1][j]） 如果 I，j>0，且 a[i-1]!=b[j-1]。
按此算式可写出计算两个序列的最长公共子序列的长度函数。由于 c[i][j]的产生仅依赖于 c[i-1][j-1]、c[i-1][j]和 c[i][j-1]，故可以从 c[m][n]开始，跟踪 c[i][j]的产生过程，逆向构造出最长公共子序列。细节见程序。

```
# include <stdio.h>
# include <string.h>
# define N 100


char a[N],b[N],str[N];

int lcs_len(char *a, char *b, int c[ ][ N])
{ int m=strlen(a), n=strlen(b), i,j;
 for (i=0;i<=m;i++) c[i][0]=0;
 for (i=0;i<=n;i++) c[0][i]=0;
 for (i=1;i<=m;i++)
 for (j=1;j<=m;j++)
 if (a[i-1]==b[j-1])
 c[i][j]=c[i-1][j-1]+1;
 else if (c[i-1][j]>=c[i][j-1])
 c[i][j]=c[i-1][j];
 else
 c[i][j]=c[i][j-1];
 return c[m][n];
}

char *buile_lcs(char s[ ],char *a, char *b)
{ int k, i=strlen(a), j=strlen(b);
```

```
k=lcs_len(a,b,c);
s[k]='\0';
while (k>0)
if (c[i][j]==c[i-1][j]) i--;


else if (c[i][j]==c[i][j-1]) j--;
else { s[--k]=a[i-1];
i--; j--;
}
return s;
}

void main()
{ printf ("Enter two string（<%d）!\n",N);
scanf("%s%s",a,b);
printf("LCS=%s\n",build_lcs(str,a,b));
}
```

1、动态规划的适用条件

任何思想方法都有一定的局限性，超出了特定条件，它就失去了作用。同样，动态规划也并不是万能的。适用动态规划的问题必须满足最优化原理和无后效性。

（1）最优化原理（最优子结构性质）

最优化原理可这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对前面的决策所形成的状态而言，余下的诸决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。一个问题满足最优化原理又称其具有最优子结构性质。

图 2

例如图 2 中，若路线 I 和 J 是 A 到 C 的最优路径，则根据最优化原理，路线 J 必是从 B 到 C 的最优路线。这可用反证法证明：假设有另一路径 J'是 B 到 C 的最优路径，则 A 到 C 的路线取 I 和 J'比 I 和 J 更优，矛盾。从而证明 J'必是 B 到 C 的最优路径。

最优化原理是动态规划的基础，任何问题，如果失去了最优化原理的支持，就不可能用动态规划方法计算。根据最优化原理导出的动态规划基本方程是解决一切动态规划问题的基本方法。

（2）无后向性

将各阶段按照一定的次序排列好之后，对于某个给定的阶段状态，它以前各阶段的状态无法直接影响它未来的决策，而只能通过当前的这个状态。换句话说，每个状态都是过去历史的一个完整总结。这就是无后向性，又称为无后效性。

（3）子问题的重叠性

动态规划算法的关键在于解决冗余，这是动态规划算法的根本目的。动态规划实质上是一种以空间换时间的技术，它在实现的过程中，不得不存储产生过程中的各种状态，所以它的空

间复杂度要大于其它的算法。选择动态规划算法是因为动态规划算法在空间上可以承受，而搜索算法在时间上却无法承受，所以我们舍空间而取时间。
所以，能够用动态规划解决的问题还有一个显著特征：子问题的重叠性。这个性质并不是动态规划适用的必要条件，但是如果该性质无法满足，动态规划算法同其他算法相比就不具备优势。
2、动态规划的基本思想
前文主要介绍了动态规划的一些理论依据，我们将前文所说的具有明显的阶段划分和状态转移方程的动态规划称为标准动态规划，这种标准动态规划是在研究多阶段决策问题时推导出来的，具有严格的数学形式，适合用于理论上的分析。在实际应用中，许多问题的阶段划分并不明显，这时如果刻意地划分阶段法反而麻烦。一般来说，只要该问题可以划分成规模更小的子问题，并且原问题的最优解中包含了子问题的最优解（即满足最优子化原理），则可以考虑用动态规划解决。

动态规划的实质是分治思想和解决冗余，因此，动态规划是一种将问题实例分解为更小的、相似的子问题，并存储子问题的解而避免计算重复的子问题，以解决最优化问题的算法策略。
由此可知，动态规划法与分治法和贪心法类似，它们都是将问题实例归纳为更小的、相似的子问题，并通过求解子问题产生一个全局最优解。其中贪心法的当前选择可能要依赖已经作出的所有选择，但不依赖于有待于做出的选择和子问题。因此贪心法自顶向下，一步一步地作出贪心选择；而分治法中的各个子问题是独立的（即不包含公共的子子问题），因此一旦递归地求出各子问题的解后，便可自下而上地将子问题的解合并成问题的解。但不足的是，如果当前选择可能要依赖子问题的解时，则难以通过局部的贪心策略达到全局最优解；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。
解决上述问题的办法是利用动态规划。该方法主要应用于最优化问题，这类问题会有多种可能的解，每个解都有一个值，而动态规划找出其中最优（最大或最小）值的解。若存在若干个取最优值的解的话，它只取其中的一个。在求解过程中，该方法也是通过求解局部子问题的解达到全局最优解，但与分治法和贪心法不同的是，动态规划允许这些子问题不独立，（亦即各子问题可包含公共的子子问题）也允许其通过自身子问题的解作出选择，该方法对每一个子问题只解一次，并将结果保存起来，避免每次碰到时都要重复计算。

因此，动态规划法所针对的问题有一个显著的特征，即它所对应的子问题树中的子问题呈现大量的重复。动态规划法的关键就在于，对于重复出现的子问题，只在第一次遇到时加以求解，并把答案保存起来，让以后再遇到时直接引用，不必重新求解。
3、动态规划算法的基本步骤
设计一个标准的动态规划算法，通常可按以下几个步骤进行：
（1）划分阶段：按照问题的时间或空间特征，把问题分为若干个阶段。注意这若干个阶段一定要是有序的或者是可排序的（即无后向性），否则问题就无法用动态规划求解。
（2）选择状态：将问题发展到各个阶段时所处于的各种客观情况用不同的状态表示出来。当然，状态的选择要满足无后效性。
（3）确定决策并写出状态转移方程：之所以把这两步放在一起，是因为决策和状态转移有

着天然的联系，状态转移就是根据上一阶段的状态和决策来导出本阶段的状态。所以，如果我们确定了决策，状态转移方程也就写出来了。但事实上，我们常常是反过来做，根据相邻两段的各状态之间的关系来确定决策。

（4）写出规划方程（包括边界条件）：动态规划的基本方程是规划方程的通用形式化表达式。

一般说来，只要阶段、状态、决策和状态转移确定了，这一步还是比较简单的。动态规划的主要难点在于理论上的设计，一旦设计完成，实现部分就会非常简单。根据动态规划的基本方程可以直接递归计算最优值，但是一般将其改为递推计算，实现的大体上的框架如下：

标准动态规划的基本框架

```
1. 对 fn+1(xn+1)初始化; {边界条件}
for k:=n downto 1 do
for 每一个 xk∈Xk do
for 每一个 uk∈Uk(xk) do
begin
5. fk(xk):=一个极值; {∞或－∞}
6. xk+1:=Tk(xk,uk); {状态转移方程}
7. t:=φ(fk+1(xk+1),vk(xk,uk)); {基本方程(9)式}
if t 比 fk(xk)更优 then fk(xk):=t; {计算 fk(xk)的最优值}
end;
9. t:=一个极值; {∞或－∞}
for 每一个 x1∈X1 do
11. if f1(x1)比 t 更优 then t:=f1(x1); {按照 10 式求出最优指标}
12. 输出 t;
```

但是，实际应用当中经常不显式地按照上面步骤设计动态规划，而是按以下几个步骤进行：

（1）分析最优解的性质，并刻划其结构特征。

（2）递归地定义最优值。

（3）以自底向上的方式或自顶向下的记忆化方法（备忘录法）计算出最优值。

（4）根据计算最优值时得到的信息，构造一个最优解。

步骤（1）～（3）是动态规划算法的基本步骤。在只需要求出最优值的情形，步骤（4）可以省略，若需要求出问题的一个最优解，则必须执行步骤（4）。此时，在步骤（3）中计算最优值时，通常需记录更多的信息，以便在步骤（4）中，根据所记录的信息，快速地构造出一个最优解。

【问题】 凸多边形的最优三角剖分问题

问题描述：多边形是平面上一条分段线性的闭曲线。也就是说，多边形是由一系列首尾相接的直线段组成的。组成多边形的各直线段称为该多边形的边。多边形相接两条边的连接点称为多边形的顶点。若多边形的边之间除了连接顶点外没有别的公共点，则称该多边形为简单多边形。一个简单多边形将平面分为 3 个部分：被包围在多边形内的所有点构成了多边形的内部；多边形本身构成多边形的边界；而平面上其余的点构成了多边形的外部。当一个简

单多边形及其内部构成一个闭凸集时，称该简单多边形为凸多边形。也就是说凸多边形边界上或内部的任意两点所连成的直线段上所有的点均在该凸多边形的内部或边界上。
通常，用多边形顶点的逆时针序列来表示一个凸多边形，即 P=<v0，v1，…，vn-1>表示具有 n 条边 v0v1，v1v2，…，vn-1vn 的一个凸多边形，其中，约定 v0=vn 。

若 vi 与 vj 是多边形上不相邻的两个顶点，则线段 vivj 称为多边形的一条弦。弦将多边形分割成凸的两个子多边形<vi，vi+1，…，vj>和<vj，vj+1，…，vi>。多边形的三角剖分是一个将多边形分割成互不重迭的三角形的弦的集合 T。图 1 是一个凸多边形的两个不同的三角剖分。

(a) (b)
图 1 一个凸多边形的 2 个不同的三角剖分
在凸多边形 P 的一个三角剖分 T 中，各弦互不相交且弦数已达到最大，即 P 的任一不在 T 中的弦必与 T 中某一弦相交。在一个有 n 个顶点的凸多边形的三角刮分中，恰好有 n-3 条弦和 n-2 个三角形。
凸多边形最优三角剖分的问题是：给定一个凸多边形 P=<v0，v1，…，vn-1>以及定义在由多边形的边和弦组成的三角形上的权函数 ω。要求确定该凸多边形的一个三角剖分，使得该三角剖分对应的权即剖分中诸三角形上的权之和为最小。
可以定义三角形上各种各样的权函数 ω。例如：定义 ω(△vivjvk)=| vivj |+| vivk |+| vkvj |，其中，| vivj |是点 vi 到 vj 的欧氏距离。相应于此权函数的最优三角剖分即为最小弦长三角剖分。
（1）最优子结构性质
凸多边形的最优三角剖分问题有最优子结构性质。事实上，若凸（n+1）边形 P=<v0，v1 ，…，vn>的一个最优三角剖分 T 包含三角形 v0vkvn，1≤k≤n-1，则 T 的权为 3 个部分权的和，即三角形 v0vkvn 的权，子多边形<v0，v1，…，vk>的权和<vk，vk+1，…，vn>的权之和。可以断言由 T 所确定的这两个子多边形的三角剖分也是最优的，因为若有<v0，v1，…，vk>或<vk，vk+1，…，vn>的更小权的三角剖分，将会导致 T 不是最优三角剖分的矛盾。
（2）最优三角剖分对应的权的递归结构
首先，定义 t[i，j]（1≤i<j≤n）为凸子多边形<vi-1，vi，…，vj>的最优三角剖分所对应的权值，即最优值。为方便起见，设退化的多边形<vi-1，vi>具有权值 0。据此定义，要计算的凸（n+1）边多边形 P 对应的权的最优值为 t[1，n]。
t[i，j]的值可以利用最优子结构性质递归地计算。由于退化的 2 顶点多边形的权值为 0，所以 t[i，i]=0，i=1，2，…，n 。当 j 一 i≥1 时，子多边形<vi-1，vi，…，vj>至少有 3 个顶点。由最优于结构性质，t[i，j]的值应为 t[i，k]的值加上 t[k+1，j]的值，再加上△vi-1vkvj 的权值，并在 i≤k≤j-1 的范围内取最小。由此，t[i，j]可递归地定义为：

（3）计算最优值
下面描述的计算凸（n+1）边形 P=<v0，v1，…，vn>的三角剖分最优权值的动态规划算法 MINIMUM_WEIGHT，输入是凸多边形 P=<v0，v1，…，vn>的权函数 ω，输出是最优值 t[i，j]和使得 t[i，k]+t[k+1，j]+ω（△vi-1vkvj）达到最优的位置（k=）s[i，j]，1≤i≤j≤n 。

```
Procedure MINIMUM_WEIGHT(P，w)；
Begin
n=length[p]-1;
for i=1 to n do t[i,i]:=0;
for ll=2 to n do
for i=1 to n-ll+1 do
begin
j=i+ll-1;
t[i,j]=∞;
for k=i to j-1 do
begin
 q=t[i,k]+t[k+1,j]+ω(△vi-1vkvj);
if q<t[i,j] then
begin
t[i,j]=q;
s[i,j]=k;
end;
 end;
end;
 return(t,s);
end;
```

算法 MINIMUM_WEIGHT_占用 θ(n2)空间，耗时 θ(n3)。

（4）构造最优三角剖分

如我们所看到的，对于任意的 1≤i≤j≤n ，算法 MINIMUM_WEIGHT 在计算每一个子多边形<vi-1，vi，…，vj>的最优三角剖分所对应的权值 t[i，j]的同时，还在 s[i，j]中记录了此最优三角剖分中与边（或弦）vi-1vj 构成的三角形的第三个顶点的位置。因此，利用最优子结构性质并借助于 s[i，j]，1≤i≤j≤n ，凸（n+l）边形 P=<v0，v1，…，vn>的最优三角剖分可容易地在 O(n)时间内构造出来。

习题：

1、汽车加油问题：

设有路程长度为 L 公里的公路上，分布着 m 个加油站，它们的位置分别为 p[i]（i=1，2，……，m），而汽车油箱加满油后（油箱最多可以加油 k 升），可以行驶 n 公里。设计一个方案，使汽车经过此公路的加油次数尽量少（汽车出发时是加满油的）。

2、最短路径：

设有一个网络，要求从某个顶点出发到其他顶点的最短路径

3、跳马问题：

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。

4、二叉树的遍历
5、背包问题
6、用分治法实现两个大整数相乘
7、设 x1，x2，…，xn 是直线上的 n 个点，若要用单位长度的闭区间去覆盖这 n 个点，至少需要多少个这样的单位闭区间？
8、用关系“＜”和“＝”将 3 个数 A、B 和 C 依次排列时，有 13 种不同的序关系：
A＝B＝C，A＝B＜C，A＜B＝C，A＜B＜C，A＜C＜B，A＝C＜B，B＜A＝C，
B＜A＜C，B＜C＜A，B＝C＜A，C＜A＝B，C＜A＜B，C＜A＜B。
若要将 n 个数依序进行排列，试设计一个动态规划算法，计算出有多少钟不同的序关系。
9、有一种单人玩的游戏：设有 n(2<=n<=200)堆薄片，各堆顺序用 0 至 n-1 编号，极端情况，有的堆可能没有薄片。在游戏过程中，一次移动只能取某堆上的若干张薄片，移到该堆的相邻堆上。如指定
I 堆 k 张 k 移到 I-1(I>0)堆，和将 k 张薄片移至 I+1(I<n-1)堆。所以当有两个堆与 I 堆相邻 时，I 堆原先至少有 2k 张薄片；只有一个堆与 I 堆相邻 时， I 堆原先至少有 k 张薄片。
游戏的目标是对给定的堆数，和各堆上的薄片数，按上述规则移动薄片，最终使 各堆的薄片数相同。为了使移动次数较少些，移动哪一堆薄片，和移多少薄片先作以下估算：
设
ci:I 堆的薄片数(0<=I<n,0<=ci<=200);
v：每堆 的平均薄片数；
ai:I 堆的相邻堆可以从 I 堆得到的薄片数。
估算方法如下：
v=c0+a1-a0 a1=v+a0-c0
v=c1+a0+a2-2a1 a2=v+2a1-a0-c1
…….. …….
V=ci+ai-1+ai+1-2aI ai+1=v+2ai-ai-1-ci
这里并不希望准确地求出 A0 至 an-1，而是作以下处理：若令 a0 为 0，能按上述算式计算出 A1 至 an-1。程序找出 a 中的最小值，并让全部 a 值减去这最小值，使每堆移去的薄片数大于等于 0。
实际操作采用以下贪心策略：
（1）每次从第一堆出发顺序搜索每一堆，若发现可从 I 堆移走薄片，就完成一次移动。即，I 堆的相邻堆从 I 堆取走 ai 片薄片。可从 I 堆移薄片到相邻堆取于 I 堆薄片数：若 I 堆是处于两端位置(I=0 I=n-1), 要求 ci>=ai ；若 I 堆是中间堆，则要求 ci>=2ai。
（2）因在 ai>0 的所有堆中，薄片数最多的堆 在平分过程中被它的相邻堆取走的薄片数也最多。在用策略（1）搜索移动时，当发生没有满足条件（1）的可移走薄片的堆时，采用本策略，让在 ai>0 的所有堆中，薄片数最多的堆被它的相邻堆取走它的全部薄片。