第5章 回 溯 法

5.1 回溯法概述

回溯法(back track method)是一种比枚举“聪明”的常用算法。

本章介绍回溯设计及其应用,并在案例的回溯求解中比较回溯相对于枚举的特点与优势。

5.1.1 回溯的概念

有许多问题,当需要找出它的解集,或者要求回答什么解是满足某些约束条件的最佳解时,往往使用回溯法。

回溯法有“通用解题法”的美称,是一种比枚举“聪明”的效率更高的搜索技术。回溯在搜索过程中动态地产生问题的解空间,系统地搜索问题的所有解。如果需要,可通过比较,在所有解中找出满足某些约束条件的最佳解。

回溯法是一种试探求解的方法:通过对问题的归纳分析,找出求解问题的一个线索,沿着这一线索往前试探,若试探成功,即得到解;若试探失败,就逐步往回退,换其他路线再往前试探。因此,回溯法可以形象地概括为“向前走,碰壁回头”,这样可大大缩减无效操作,提高搜索效率。

回溯法的试探搜索是一种组织得井井有条的、能避免一些不必要搜索的枚举式搜索。回溯法在问题的解空间树中,从根结点出发搜索解空间树,搜索至解空间树的任意一个结点,先判断该结点是否包含问题的解;如果肯定不包含,则跳过对该结点为根的子树的搜索,逐层向其父结点回溯;否则,进入该子树,继续搜索。

从解的角度理解,回溯法将问题的候选解按某种顺序进行枚举和检验。当发现当前候选解不可能是解时,就选择下一个候选解。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。如果当前候选解除了不满足问题规模要求外,满足所有其他要求,则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,则该候选解就是问题的一个解。

与枚举法相比,回溯法的“聪明”之处在于能适时“回头”,若再往前走不可能得到解,就回溯,退一步另找线路,这样可省去大量的无效操作。因此,与枚举相比,回溯更适宜于问题规模比较大,候选解比较多的案例求解。

5.1.2 回溯描述

1. 回溯的一般方法

下面简要阐述回溯的一般方法。

回溯求解的问题 P 通常要能表达为:对于已知的由 n 元组$(x_1,x_2,\cdots,x_n)$组成的一个状态空间 $E=\{(x_1,x_2,\cdots,x_n)\mid x_i\in s_i,i=1,2,\cdots,n\}$,给定关于 n 元组中的一个分量的约

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,s_i 是分量 x_i 的定义域,$i=1,2,\cdots,n$。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,即对 E 中的所有 n 元组逐一地检验其是否满足 D 的全部约束。若满足,则为问题 P 的一个解。显然,当 P 的数量规模比较大时,枚举的计算量是相当大的。

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$(x_1,x_2,\cdots,x_j)$违反 D 中仅涉及 $x_1,x_2,\cdots,x_j$ 的一个约束,就可以肯定以$(x_1,x_2,\cdots,x_j)$为前缀的任何 n 元组$(x_1,x_2,\cdots,x_j,x_{j+1},\cdots,x_n)$都不会是问题 P 的解,因而就不必去搜索它们,省略了对部分元素$(x_{j+1},\cdots,x_n)$的操作与测试。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效率更高的算法。

2.4 皇后问题的回溯举例

为了具体说明回溯的实施过程,先看一个简单实例。

如何在 4×4 的方格棋盘上放置 4 个皇后,使皇后互不攻击,即任意两个皇后不允许处在同一横排或同一纵列,也不允许处在与棋盘边框成 45°角的同一斜线上。

图 5-1 所示为应用回溯的实施过程,其中方格中的数字表示皇后所在位置(列),方格中的×表示由于受前面已放置的皇后的攻击而放弃的位置。

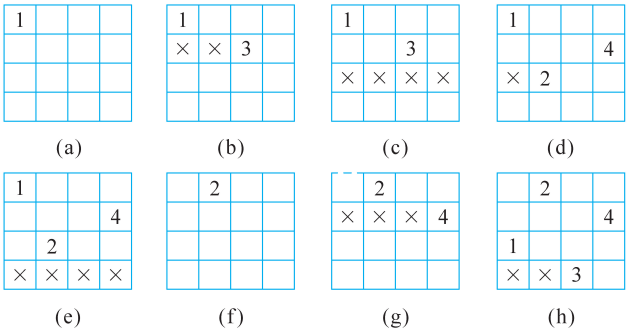

图 5-1 4 皇后问题回溯实施

图 5-1(a)为在第 1 行第 1 列放置一个皇后的初始状态。

图 5-1(b)中,第 2 个皇后不能放置在第 1、2 列,因而放置在第 3 列。

图 5-1(c)中,表示第 3 行的所有各列均不能放置皇后,则回溯至第 2 行,第 2 个皇后需后移。

图 5-1(d)中,第 2 个皇后后移到第 4 列,第 3 个皇后放置在第 2 列。

图 5-1(e)中,第 4 行的所有各列均不能放置皇后,则回溯至第 3 行;第 3 个皇后后移的所有位置均不能放置皇后,则回溯至第 2 行;第 2 个皇后已无位可退,则回溯至第 1 行;第 1 个皇后需后移。

图 5-1(f)中,第 1 个皇后后移至第 2 格。

图 5-1(g)中,第 2 个皇后不能放在第 1、2、3 列,因而放置在第 4 列。

图 5-1(h)中,第 3 个皇后放在第 1 列;第 4 个皇后不能放置在第 1、2 列,于是放置在第 3 列。

经以上探索与回溯,得到 4 皇后问题的一个解:2413(第 1 行皇后在第 2 列,第 2 行皇

后在第 4 列，第 3 行皇后在第 1 列，第 4 行皇后在第 3 列）。

继续探索与回溯，可得 4 皇后问题的另一个解：3142。

继续探索与回溯，直到第 1 行的皇后至第 4 格后无位可退，探索结束。

3. 回溯法框架描述

1）回溯描述

对于一般含参量 m、n 的搜索问题，回溯法框架描述如下：

```
输入正整数 n,m,(n≥m)
i=1;a[i]=<元素初值>;
while(1)
{
    for(g=1,k=i-1;k>=1;k--)
        if(<约束条件 1>) g=0;                    //检测约束条件,不满足则返回
    if(g &&<约束条件 2>)
         printf(a[1: m]);                        //输出一个解
    if(i<n && g)
         {i++;a[i]=<取值点>;continue;}
    while(a[i]==<回溯点>&& i>1) i--;             //向前回溯
    if(a[i]==n && i==1) break;                   //退出循环,结束
    else a[i]=a[i]+1;
}
```

具体求解问题的探索搜索范围与要求不同，在应用回溯设计时，需根据问题的具体情况确定数组元素的初值、取值点与回溯点，同时需把问题中的约束条件进行必要的分解，以适应上述回溯流程。

其中，实施向前回溯的循环

```
while(a[i]==<回溯点>&& i>1) i--;
```

是向前回溯一步还是回溯两步或更多步，完全根据 $a[i]$ 是否达到回溯点来确定。例如，回溯点是 n，$i=6$，当 $a[6]=n$ 时回溯到 $i=5$；若 $a[5]=n$ 时回溯到 $i=4$；以此类推。

以上回溯由迭代式 $i--$；（即 $i=i-1$；）实现，因而又称为迭代回溯。

图 5-1 所示的 4 皇后问题迭代回溯过程描述如下：

```
i=1;a[i]=1;
while(1)
{
    g=1;for(k=i-1;k>=1;k--)
    if(a[i]=a[k] || abs(a[i]-a[k])=i-k)
        g=0;                                     //检测约束条件,不满足则返回
    if(g && i==4)
        printf(a[1: 4]);                         //输出一个解
    if(i<4 && g) {i++;a[i]=1;continue;}
    while(a[i]==4 && i>1) i--;                   //向前回溯
    if(a[i]==4 && i==1) break;                   //退出循环结束探索
    else a[i]=a[i]+1;
}
```

2）递归回溯

在第 4 章应用递归实现排列组合的设计中，我们已经知道递归也能实现回溯。递归回

溯通过递归尝试遍历问题的各个可能解的通路。当发现此路不通时,回溯到上一步,继续尝试别的通路。

递归回溯描述如下:

```
int put(int k)
{  int i,j,u;
   if( k<=<规模>)
   {  u=0;
      if(<约束条件>) u=1;          //当 u=1 时不可操作
      if(u==0)                     //当 u=0 时可操作
      {  if(k==<规模>)             //若已满足规模,则打印出一个解
           printf(<一个解>);
         else  put(k+1);           //调用 put(k+1)
      }
   }
}
```

在调用 put(k)时,当检测约束条件知不可操作(记 u=1),即再往前不可能得解,此时当然不可能输出解,也不调用 put(k+1),而是回溯,返回调用 put(k)之处。这就是递归回溯的机理。

如果是主程序调用 put(1),最后返回到主程序调用 put(1)的后续语句,完成递归。

图 5-1 所示的 4 皇后问题迭代回溯过程描述如下:

```
int put(int k)
{  int i,j,u;
   if(k<=4)
   {  for(i=1;i<=4;i++)                    //探索第 k 行从第 1 格开始放皇后
      {  a[k]=i;
         for(u=0,j=1;j<=k-1;j++)
            if(a[k]==a[j] || abs(a[k]-a[j])==k-j)
              u=1;                         //若第 k 行第 i 格放不下,则置 u=1
         if(u==0)                          //若第 k 行第 i 格可放,则检测是否满 4 行
         {  if(k==4)                       //若已放满 4 行,则打印出一个解
            {  s++; printf(" ");
               for(j=1;j<=4;j++)
                  printf("%d",a[j]);
            }
            else  put(k+1);                //若没放满 4 行,则放下一行: put(k+1)
         }
      }
   }
}
```

4. 回溯法的效益分析

应用回溯设计求解实际问题,由于解空间的结构差异,很难精确计算与估计回溯产生的结点数,这是分析回溯法效率时遇到的主要困难。回溯法实际产生的结点数通常只有解空间所有结点数的一小部分,这也是回溯法的探索效率大大高于枚举的原因所在。

回溯求解过程实质上是遍历一棵“状态树”的过程,只是这棵树不是遍历前预先建立的。回溯法在搜索过程中,只要所激活的状态结点满足终结条件,应该把它输出或保存。由于在回溯法求解问题时,一般要求输出问题的所有解,因此,在得到并输出一个解后并不终止,还

要进行回溯，以便得到问题的其他解，直至回溯到状态树的根且根的所有子结点均已被搜索过为止。

组织解空间便于算法在求解集时更易于搜索，典型的组织方法是图或树。一旦定义了解空间的组织方法，这个空间即可从开始结点进行搜索。

回溯法的时间通常取决于状态空间树上实际生成的那部分问题状态的数目。对于元组长度为 n 的问题，若其状态空间树中结点总数为 $n!$，则回溯算法的最坏情形的时间复杂度可达 $O(p(n)n!)$；若其状态空间树中结点总数为 2^n，则回溯算法的最坏情形的时间复杂度可达 $O(p(n)2^n)$，其中 $p(n)$ 为 n 的多项式。

对于不同的实例，回溯法的计算时间有很大的差异。对于数量规模比较大的求解实例，应用回溯法一般可在较短的时间内求得其解，可见回溯法不失为一种快速有效的算法。

对于某一具体情况问题的回溯求解，常通过计算实际生成结点数的方法即蒙特卡罗方法(Monte Carlo)来评估其计算效率。蒙特卡罗方法的基本思想是：在状态空间树上随机选择一条路径($\boldsymbol{x}_0,\boldsymbol{x}_1,\cdots,\boldsymbol{x}_{n-1}$)，设 X 是这一路径上部分向量($\boldsymbol{x}_0,\boldsymbol{x}_1,\cdots,\boldsymbol{x}_{k-1}$)的结点，如果在 X 处不受限制的子向量数是 m_k，则认为与 X 同一层的其他结点不受限制的子向量数也都是 m_k。也就是说，若不受限制的 $\boldsymbol{x}_0$ 取值有 m_0 个，则该层上有 m_0 个结点；若不受限制的 $\boldsymbol{x}_1$ 取值有 m_1 个，则该层上有 m_0m_1 个结点；以此类推。由于认为在同一层上不受限制的结点数相同，因此，该路径上实际生成的结点数估计为

$$m=1+m_0+m_0m_1+m_0m_1m_2+\cdots$$

计算路径上结点数 m 的蒙特卡罗算法描述如下：

```
//已知随机路径上取值数据 m0,m1,…,mk-1
m=1;t=1;
for(j=0;j<=k-1;j++)
{   t=t*m[j];
    m=m+t;
}
printf("%ld",m);
```

把所求得的随机路径上的结点数(或若干条随机路径的结点数的平均值)与状态空间树的总结点数进行比较，由其比值可以初步看出回溯设计的效益。

5.2 桥本分数式与10数字分数式

桥本分数式是一个填数趣题，本节从桥本分数式及其引申10数字分数式这两个难度不高的典型数式案例的求解入手，看看回溯求解的具体实现。

5.2.1 桥本分数式

1. 案例背景

日本数学家桥本吉彦教授于1993年10月在我国山东举行的中日美三国数学教育研讨会上向与会者提出以下填数趣题：把1～9这9个数字填入下式的9个方格中(数字不得重复)，使分数等式成立。

$$\frac{\square}{\square\square}+\frac{\square}{\square\square}=\frac{\square}{\square\square}$$

桥本教授当即给出了一个解答。这一填数趣题的解是否唯一？如果不唯一，究竟有多少个解？试求出所有解答（等式左边两个分数交换次序只算一个解答）。

2. 回溯设计

我们采用回溯法逐步调整探求。把式中 9 个□规定一个顺序后，先在第一个□中填入一个数字(从 1 开始递增)，然后从小到大选择一个不同于前面□中的数字填在第二个□中，以此类推，把 9 个□都填入没有重复的数字后，检验是否满足等式。若等式成立，打印所得的解。

可见，问题的解空间是 9 位的整数组，其约束条件是 9 位数中没有相同数字且必须满足分式的要求。

为此，设置 a 数组，式中每一□位置用一个数组元素来表示：

$$\frac{a[1]}{a[2]a[3]}+\frac{a[4]}{a[5]a[6]}=\frac{a[7]}{a[8]a[9]}$$

同时，记式中的 3 个分母分别为

$$m_1=a[2]a[3]=a[2]\times 10+a[3]$$
$$m_2=a[5]a[6]=a[5]\times 10+a[6]$$
$$m_3=a[8]a[9]=a[8]\times 10+a[9]$$

所求分数等式等价于整数等式 $a[1]m_2m_3+a[4]m_1m_3=a[7]m_1m_2$ 成立。这一转换可以把分数的测试转换为整数测试。

注意到等式左侧两个分数交换次序只算一个解，为避免解的重复，设 $a[1]<a[4]$。

式中 9 个□各填一个数字，不允许重复。为判断数字是否重复，设置中间变量 g，先赋值 $g=1$；若出现某两数字相同(即 $a[i]=a[k]$)或 $a[1]>a[4]$，则赋值 $g=0$(重复标记)。

首先从 $a[1]=1$ 开始，逐步给 $a[i](1\leqslant i\leqslant 9)$赋值，每一个 $a[i]$赋值从 1 开始递增至 9，直至 $a[9]$赋值，判断：

(1) 若 $i=9$，$g=1$，$a[1]m_2m_3+a[4]m_1m_3=a[7]m_1m_2$ 同时满足，则为一组解，用 n 统计解的个数后，输出这组解。

(2) 若 $i<9$ 且 $g=1$，表明还不到 9 个数字，则下一个 $a[i]$从 1 开始赋值继续。

(3) 若 $a[9]=9$，则返回前一个数组元素 $a[8]$增 1 赋值(此时，$a[9]$又从 1 开始)再试。若 $a[8]=9$，则返回前一个数组元素 $a[7]$增 1 赋值再试。以此类推，直到 $a[1]=9$ 时，已无法返回，意味着已全部试毕，求解结束。

按以上所描述的回溯的参量：$m=n=9$。

元素初值：$a[1]=1$，数组元素初值取 1。

取值点：$a[i]=1$，各元素从 1 开始取值。

回溯点：$a[i]=9$，各元素取值至 9 后回溯。

约束条件 1：$a[i]==a[k]\ ||\ a[1]>a[4]$，其中 $i>k$。

约束条件 2：$i=9\ \&\&\ a[1]m_2m_3+a[4]m_1m_3=a[7]m_1m_2$。

3. 桥本分数式回溯程序设计

```
//桥本分数式回溯实现,c521
//把 1～9 填入□/□□+□/□□=□/□□
#include<stdio.h>
void main()
{  int g,i,k,s,a[10];
   long m1,m2,m3;
   i=1;a[1]=1;s=0;
   while(1)
   {  g=1;
      for(k=i-1;k>=1;k--)
        if(a[i]==a[k]) {g=0;break;}                    //两数相同,标记 g=0
      if(i==9 && g==1 && a[1]<a[4])
      {  m1=a[2]*10+a[3];
         m2=a[5]*10+a[6];
         m3=a[8]*10+a[9];
         if(a[1]*m2*m3+a[4]*m1*m3==a[7]*m1*m2)         //判断等式
         {  s++;printf("(%2d) ",s);
            printf("%d/%ld+%d/",a[1],m1,a[4]);
            printf("%ld=%d/%ld   ",m2,a[7],m3);
            if(s%2==0) printf("\n");
         }
      }
      if(i<9 && g==1)
        {i++;a[i]=1;continue;}                         //不到 9 个数,往后继续
      while(a[i]==9 && i>1) i--;                       //往前回溯
      if(a[i]==9 && i==1) break;
      else a[i]++;                                     //至第 1 个数为 9 时结束
   }
   printf("  共以上%d 个解。\n",s);
}
```

4. 程序运行结果与说明

```
(1) 1/26+5/78=4/39   (2) 1/32+5/96=7/84
(3) 1/32+7/96=5/48   (4) 1/78+4/39=6/52
(5) 1/96+7/48=5/32   (6) 2/68+9/34=5/17
(7) 2/68+9/51=7/34   (8) 4/56+7/98=3/21
(9) 5/26+9/78=4/13   (10) 6/34+8/51=9/27
共以上 10 个解。
```

关于桥本分数式求解,已有应用程序设计得到 9 个解,显然遗失了一个解。可见在程序设计求解时,如果程序中结构欠妥或参量设置不当,也可能造成解的遗失。

5.2.2　10 数字分数式

本节推出的 10 数字分数式是前面桥本分数式的引申,并添加“最简真分数”的条件限制。

1. 案例提出

把 0,1,2,…,9 这 10 个数字填入下式的 10 个方格中。

$$\frac{\square}{\square\square}+\frac{\square}{\square\square\square}=\frac{\square}{\square\square}$$

要求：

(1) 各数字不得重复。

(2) 数字 0 不得填在各分数的分子或分母的首位。

(3) 式中各分数为最简真分数，即分子、分母没有大于 1 的公因数。

这一分数等式填数趣题究竟有多少个解答？试求出所有解答。

2. 回溯设计

设置 a 数组表示式中的 10 个数字，即

$$\frac{a[1]}{a[2]a[3]}+\frac{a[4]}{a[5]a[6]a[7]}=\frac{a[8]}{a[9]a[10]}$$

同时，记式中的 3 个分母分别为

$m_1=a[2]a[3]=a[2]\times 10+a[3]$

$m_2=a[5]a[6]a[7]=a[5]\times 100+a[6]\times 10+a[7]$

$m_3=a[9]a[10]=a[9]\times 10+a[10]$

在上述回溯设计的基础上修改若干参数：

(1) 数字从 9 个增加到 10 个，因而 $i<9$ 改为 $i<10$；$i==9$ 改为 $i==10$。

(2) 数组元素取值修改为从 0 开始，即 $a[1]=0$，$a[i]=0$。

(3) 数字 0 不得在各分数的分子与分母的首位，即 0 只能在 $a[3]$、$a[6]$、$a[7]$与 $a[10]$ 这 4 个数字中，因而在输出解的条件中增加 $a[3]a[6]a[7]a[10]=0$。

此外，需增加判断 3 个分数是否为最简真分数的测试循环。

3. 10 数字分数式程序实现

```
//10 数字分数式,c522
#include<stdio.h>
void main()
{  int g,i,k,s,t,u,a[11]; long m1,m2,m3;
   i=1;a[1]=0;s=0;
   while(1)
   {  g=1;
      for(k=i-1;k>=1;k--)
        if(a[i]==a[k]) {g=0;break;}                    //两数相同,标记 g=0
      if(i==10 && g==1 && a[3]*a[6]*a[7]*a[10]==0)
      {  m1=a[2]*10+a[3];
         m2=a[5]*100+a[6]*10+a[7];
         m3=a[9]*10+a[10];
         if(a[1]*m2*m3+a[4]*m1*m3==a[8]*m1*m2)   //判断等式
         {  t=0;
            for(u=2;u<=9;u++)                          //测试 3 个分数是否为真分数
            {  if(a[1]%u==0 && m1%u==0) {t=1;break;}
               if(a[4]%u==0 && m2%u==0) {t=1;break;}
               if(a[8]%u==0 && m3%u==0) {t=1;break;}
```

```
            }
            if(t==0)
            {  printf("  %d/%ld+%d/",a[1],m1,a[4]);
               printf("%ld=%d/%ld\n ",m2,a[8],m3);
            }
         }
      }
      if(i<10 && g==1)
        {i++;a[i]=0;continue;}                    //不到 10 个数,往后继续
      while(a[i]==9 && i>1) i--;                  //往前回溯
      if(a[i]==9 && i==1) break;
      else a[i]++;                                //至第 1 个数为 9 时结束
   }
}
```

4. 程序运行结果与变通

```
4/19+5/608=7/32
```

以上 10 数字分数式求解是在桥本分数式设计基础上变通所得的，结构完全相同。请比较以上两个回溯设计的参数变化。

如果把问题要求中的第(3)点去掉，即不要求各分数为最简真分数，程序应如何修改？

以上桥本分数式与 10 数字分数式的回溯设计都能快捷地通过上机求解，速度明显快于枚举设计。

5.3　直尺与串珠

本节应用回溯探索涉及直尺刻度分布的“古尺神奇”与环序列覆盖的“数码串珠”两个有趣的案例。

5.3.1　古尺神奇

1. 案例提出

有一把年代不明的古尺长 36 寸(1 寸≈3.33 厘米)，因使用日久，尺上的刻度只剩下 8 条，其余刻度均已不复存在。神奇的是，用该尺仍可一次性度量 1～36 的任意整数寸长度。

试确定古尺上 8 条刻度分布的位置。

2. 回溯设计

这是一个新颖且有一定难度的实用案例。

我们探索更具一般性的尺长 s，刻度数为 n(s、n 均为正整数)的完全度量问题。

为了确定实现尺长 s 完全度量的 n 条刻度的分布位置，设置 a、b 两个数组。

a 数组元素 $a[i]$为第 i 条刻度距离尺左端线的长度，约定 $a[0]=0$ 以及 $a[n+1]=s$ 对应尺的左右端线。注意到尺的两端至少有一条刻度距端线为 1(否则长度 $s-1$ 不能度量)，不妨设 $a[1]=1$，其余的 $a[i](i=2,3,\cdots,n)$在 $2\sim s-1$ 中取数。不妨设

$$2\leqslant a[2]<a[3]<\cdots<a[n]\leqslant s-1$$

从 $a[2]$取 2 开始，以后 $a[i]$从 $a[i-1]+1$ 开始递增 1 取值，直至 $s-(n+1)+i$ 为止。

当 $i=n$ 时,n 条刻度连同尺的两条端线共 $n+2$ 条,从 $n+2$ 取 2 的组合数为 $C(n+2,2)$,记为 m,显然有

$$m=C(n+2,2)=\frac{(n+1)(n+2)}{2}$$

m 种长度赋给 b 数组元素 $b[1],b[2],\cdots,b[m]$。为判定某种刻度分布能否实现完全度量,设置特征量 u,对于 $1\leqslant d\leqslant s$ 的每一个长度 d,如果在 $b[1]\sim b[m]$中存在某一元素等于 d,特征量 u 值增 1。

若 $u=s$,说明从 1 至尺长 s 的每一个整数 d 都有一个 $b[i]$相对应,即达到完全度量,于是输出直尺的 n 条刻度分布位置。

(1) 若 $i<n$,i 增 1,$a[i]=a[i-1]+1$,继续探索。

(2) 若 $i>1$,$a(i)$增 1,至 $a[i]=s-(n+1)+i$ 时回溯。

3. 古尺刻度回溯程序设计

```
//尺长 s,寻求 n 条刻度分布回溯探索,c531
#include<stdio.h>
void main()
{  int d,i,j,k,t,u,s,m,n,a[30],b[300];
   printf("  尺长 s,寻求 n 条刻度分布,请确定 s,n: ");
   scanf("%d,%d",&s,&n);
   a[0]=0;a[1]=1;a[n+1]=s;
   m=(n+2) * (n+1)/2;
   i=2;a[i]=2;
   while(1)
   {  if(i<n)
      {i++; a[i]=a[i-1]+1; continue;}
      else
      {  for(t=0,k=0;k<=n;k++)
         for(j=k+1;j<=n+1;j++)
            {t++;b[t]=a[j]-a[k];}              //序列部分和赋值给 b 数组
         for(u=0,d=1;d<=s;d++)
         for(k=1;k<=m;k++)
            if(b[k]==d) {u+=1;k=m;}            //检验 b 数组取 1~s 有多少个
         if(u==s)                              //b 数组值包括 1~s 所有整数
         {  if((a[n]!=s-1) || (a[n]==s-1) && (a[2]<=s-a[n-1]))
            {  printf("┌");                    //输出尺的上边
               for(k=1;k<=s-1;k++) printf("—");
               printf("┐\n");
               printf("| ");
               for(k=1;k<=n+1;k++)             //输出尺的数字标注
               {  for(j=1;j<=a[k]-a[k-1]-1;j++) printf("  ");
                  if(k<n+1) printf("%2d",a[k]);
                  else printf("|\n");
               }
               printf("└");                    //输出尺的下边与刻度
               for(k=1;k<=n+1;k++)
```