

## 组合算法的选择与应用

**【关键字】** 组合算法    评价依据    复杂性    选择  
应用

### **【摘要】**

本文提出了在组合算法设计和组合算法选择方面所应当遵循的三个原则，即通用性、可计算性和较少的信息冗余量，并初步分析了它们之间的相互关系。这三个原则是整个组合算法设计的主导思想，也是数学建模和算法优化的方向。通过对三个问题的分析，提示了组合算法的设计方法，改进方向和优化技术，是对一系列组合数学原理的实际应用，也是对组合算法设计的初步研究。

### **【Abstract】**

The disquisition brings forward three principle in combination arithmetic designing and combination arithmetic choice. There is currency, countability and less information redundancy. And the disquisition analyses the relation of them. The three principle is the dominant ideology in combination arithmetic designing. Also it is the aspect of making mathematics modeling and optimizing arithmetic. Then the disquisition analyses three questions, prompts the devisal's method, betterment's way and the technique optimizing arithmetic. That is actual appliance to a catena of combination mathematics elements and it is also initial research in combination arithmetic designing.

## 【正文】

### 一、引 论

组合数学是一个古老的数学分支，也是当代数学中非常重要的数学分支。它发源于有趣的数学游戏，许多古典的组合数学问题，无论在理论数学或应用数学上都有很重要的研究价值。

今天，一方面，极为成熟的组合计数理论为物理学、化学、生物学的发展奠定了坚实的基础，另一方面，由于计算机软硬件的限制，组合计数理论的计算机实践又必然涉及到基于多项式时间内的算法优化问题。本文正是基于以上情况，对一系列组合问题的算法设计做了一些初步探索。

## 二、组合算法的评价依据

任何事物都有好坏之分，算法也不例外。众所周知，时间复杂度与空间复杂度是算法评价的主要依据。那么，除了这两点外，组合算法的设计还应遵循怎样的原则呢？

### 1. 通用性

通用性即可移植性。一个算法，是只适合于一个特殊问题，还是可以适用于一类问题，这是组合算法评价的一个主要依据，有些组合数学问题，许多信息学竞赛或数学建模竞赛选手一看到题目后往往使用模拟法或构造产生式系统<sup>1</sup>，然后用深度优先搜索（DFS），或广度优先搜索（BFS）求解，用这些方法设计的程序往往受到时间或空间的限制，而且由于在综合数据库中信息存储的数据结构不同，其算法实现时的规模<sup>2</sup>也不同，这必然影响到算法的通用性问题。解决问题的办法是对原问题进行数学抽象，取其精华，去其糟粕。只有对原问题的数学模型仔细分析，抓住本质，才能建立高效的组合算法，只有这种经过数学抽象的算法，才能具有较好的通用性，能够应付较大的规模。

另外，在大规模组合算法的设计过程中，强调通用性的好处是显而易见的，它便于算法的优化和升级。在实际应用中的某些条件改变时，可以重写较少的代码。从软件工程学的角度来说，通用性是必需的。

### 2. 可计算性

这里指的可计算性，是指能够在多项式时间内得出结果。一般来说，对于递归函数来说，由于计算机系统空间一定，因此对问题的规模有较严格的限制（例如在 Turbo Pascal 7.0 系统

---

<sup>1</sup> 见参考文献[6]第一章

<sup>2</sup> 在本论文中，我们将规模定义为在一定时间内程序可以运行完毕的情况下输入数据的最大量。

中, 栈的最大空间只有 65520 字节) 因此说, 对于大多数的递归函数具有较差的可计算性。通过组合方法, 求递归函数的非递归形式是解决这类问题有主要方法, 但并不是所有的递归函数都可转化为非递归形式, 双递归函数 (如 Ackermann 函数<sup>3</sup>) 便不能转化为非递归形式, 这类函数具有较小的可计算性。

当然, 对于某些递归函数, 通过寻找函数本身的组合意义进而将其转化为非递归函数也是一种方法。这种方法的应用读者将在后文中见到。

### 3、信息冗余量

在组合数学的建模过程中, 大量的信息冗余是制约组合算法效率的一个重要因素, 例如在递归程序运行的过程中, 实际上产生了一棵解答树<sup>4</sup>, 同一棵子树的结点间的信息不相互关联, 这样便产生了大量的信息冗余, 解决的方法之一便是引入记忆机制, 将已得出的信息记录下来。这种机制实际上起到了剪枝的作用, 但它严格受到空间的限制, 实际上是时空矛盾在算法设计中的体现。这便是我们在组合算法设计中倡导函数非递归化的原因。它可以达到零信息冗余。

当然, 组合算法的通用性、可计算性与信息冗余程度在一定程度上是对立的。例如双递归函数作为一种数学模型, 能够反映一类问题, 具有通用性, 但却具有较差的可计算性和较大的信息冗余量, 而有些问题虽具有较好的可计算性和较低的信息冗余量, 却具有较差的通用性。总之, 算法的时间复杂度, 空间复杂度, 通用性, 可计算性和信息冗余量应是衡量组合算法的几个主要标准。

## 三、组合算法的选择实例

<sup>3</sup> Ackermann 函数可用递推关系如下定义

$$A(m, 0) = A(m-1, 0) \quad m=1, 2, \dots$$

$$A(m, n) = A(m-1, A(m, n-1)) \quad m=1, 2, \dots \quad n=1, 2, \dots$$

初始条件为

$$A(0, n) = n+1, \quad n=0, 1, \dots$$

<sup>4</sup> 见参考文献[6]第二章 (产生式系统的搜索策略)

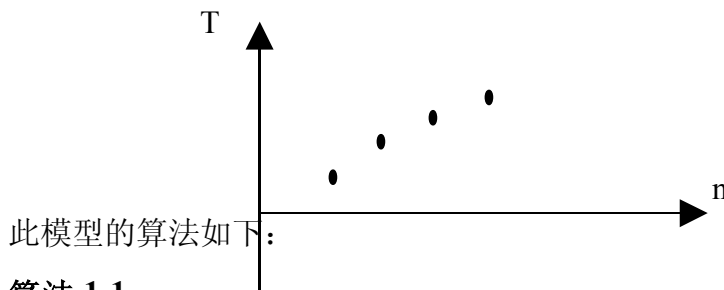
组合算法的评价依据同时也是建立数学模型和优化算法的指导思想。那么应该如何设计高效，通用的组合算法呢？下面我们通过几个实际的组合问题来初步研究。

**例 1.** 核反应堆中有  $\alpha$  和  $\beta$  两种粒子，每秒钟内一个  $\alpha$  粒子可以反应产生 3 个  $\beta$  粒子，而一个  $\beta$  粒子可以反应产生 1 个  $\alpha$  粒子和 2 个  $\beta$  粒子。若在  $t=0$  时刻的反应堆中只有一个  $\alpha$  粒子，求在  $t$  时刻的反应堆中  $\alpha$  粒子和  $\beta$  粒子数。

这是一个物理学中的简单问题。我们通过对两种算法的比较来说明组合算法的通用性。

**模型 I:** 本题中共涉及两个变量，设在  $i$  时刻  $\alpha$  粒子数为  $n_i$ ， $\beta$  粒子数为  $m_i$ ，则有： $n_0=1, m_0=0, n_i=m_{i-1}, m_i=3n_{i-1}+2m_{i-1}$

本题便转化为求数列  $N$  和  $M$  的第  $t$  项，我们可用递推的方法求得  $n_t$  和  $m_t$ ，此模型的空间需求较小，时间复杂度为  $O(n)$ ，但随着  $n$  的增大，所需时间越来越大，即：



#### 算法 1-1

```

Prog Arithmtic1_1;
Begin
  Read(t);
  n[0]:=1;    //初始化操作
  m[0]:=0;
  for i:=1 to t do //进行 t 次递推

```

```

begin
  n[i]:=m[i-1];
  m[i]:=3*n[i-1]+2*m[i-1];
end;
write(n[t]); //输出结果
write(m[t]);
End. Arithmtic1_1

```

模型 II: 设在  $t$  时刻的  $\alpha$  粒子数为  $f(t)$ ,  $\beta$  粒子数为  $g(t)$ , 依题可知:

$$\begin{cases} g(t)=3f(t-1)+2g(t-1) & (1) \\ f(t)=g(t-1) & (2) \\ g(0)=0, f(0)=1 \end{cases}$$

下面求解这个递归函数的非递归形式

$$\text{由 (2) 得 } f(t-1)=g(t-2) \quad (3)$$

将 (3) 代入 (1) 得

$$g(t)=3g(t-2)+2g(t-1) \quad (t \geq 2) \quad (4)$$

$$g(0)=0, g(1)=3$$

(4) 式的特征根方程为:

$$x^2 - 2x - 3 = 0$$

其特征根为  $x_1=3, x_2=-1$

所以该式的递推关系的通解为

$$g(t)=C_1 \cdot 3^t + C_2 \cdot (-1)^t$$

代入初值  $g(0)=0, g(1)=3$  得

$$C_1 + C_2 = 0$$

$$3C_1 - C_2 = 3$$

$$\text{解此方程组得 } C_1 = \frac{3}{4}, C_2 = -\frac{3}{4}$$

所以该递推关系的解为

$$g(t) = \frac{3}{4} \cdot 3^t - \frac{3}{4} \cdot (-1)^t$$

$$\therefore f(t) = g(t-1) = \frac{\tau}{\xi} \cdot \tau^{t-1} - \frac{\tau}{\xi} \cdot (-1)^{t-1}$$

$$\text{即 } f(t) = \frac{\tau^t}{\xi} + \frac{\tau}{\xi} \cdot (-1)^t$$

### 算法 1—2

Prog Arithmetic1\_2;

Begin

  read(t);

  n:=trunc(exp(t\*ln(3)));

  m:=trunc(exp((t+1)\*ln(3)));

  if odd(t) then begin //判断 $(-1)^t$

    n:=n-3;

    m:=m+3;

  end

  else begin

    n:=n+3;

    m:=m-3;

  end;

  n:=trunc(n/4); //  $4|n$

  m:=trunc(m/4); //  $4|m$

  Write(n);

  Write(m);

End. Arithmetic1\_2

在模型 II 中, 我们运用组合数学的方法建立了递归函数并转化为非递归函数。它的优点是算法的复杂性与问题的规模无关。针对某一具体数据, 问题的规模对时间的影响微乎其微。

通过以上两个模型我们可以看出, 模型 II 抓住了问题的本质, 尤其成功地运用了组合数学中关于常系数线性齐次递推关系求解的有关知识, 因而使算法本身既具有通用性和可计算性, 同时达到了零信息冗余。

**例 2.** 在平面直角坐标系中, 有  $n$  个圆心坐标为整点的单位圆, 求它们所覆盖区域的总面积。

这是一道计算几何学的问题, 关于图形并的问题, 较为常用

的方法是离散化，但是如果借助于组合数学的有关理论，是否可以设计出更好的算法呢？我们先来看几个简单的例子。

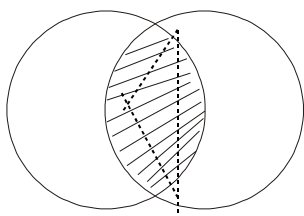
(1) 两个圆的交（交集不为  $\phi$ ）

设圆  $i$  的圆心坐标为  $(x_i, y_i)$ ，我们定义一个异型函数（dissimilarity function）如下：

$$\varphi(i, j) = |x_i - x_j| + |y_i - y_j|$$

在讨论两个圆的交的问题时，设两圆为圆 1 与圆 2，它们的交有两种情况

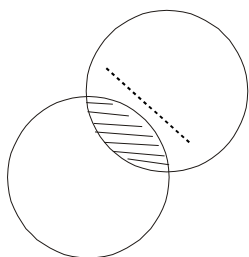
①  $\varphi(1, 2) = 1$



设阴影部分面积为  $S$ ，则

$$\begin{aligned} S &= r \cdot \left( \frac{1}{r} \pi r^2 - \frac{1}{r} \cdot \frac{1}{r} \cdot \sqrt{r} \right) \\ &= \frac{\pi}{r} \pi r^2 - \frac{\sqrt{r}}{r} \\ &= \frac{2}{3} \pi - \frac{\sqrt{3}}{2} \end{aligned}$$

②  $\varphi(1, 2) = 2$



设阴影部分面积为  $S$ ，则

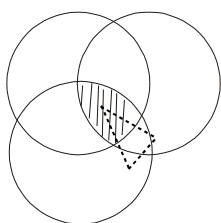
$$\begin{aligned} S &= r \cdot \left( \frac{1}{r} \pi r^2 - \frac{1}{r} r^2 \right) \\ &= \frac{\pi}{r} r^2 - r^2 \\ &= \frac{\pi}{2} - 1 \end{aligned}$$

交集问题是圆最简单的交问题。所以我们规定  $\varphi(i, j) = 1$  的交为  $\alpha$  型交， $\varphi(i, j) = 2$  的交为  $\beta$  型交，这个规定将在下文的讨论中用到。



## 2、三个圆的交（交集不为 $\phi$ ）：

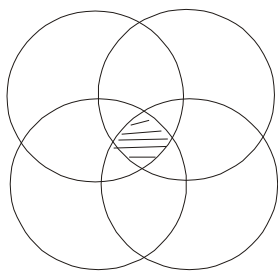
经过分析易证，若三个圆的交集不为空，则三个圆中任意两圆的交集一定不为空，反之亦成立。且在任意两圆相交所组成的三个交中，一定有 2 个  $\alpha$  型交，1 个  $\beta$  型交。如图所示，阴影部分面积为  $S$ ，则有：



$$\begin{aligned} S &= 2 \cdot \left( \frac{1}{6} \cdot \pi \cdot r^2 - \frac{\sqrt{3}}{4} r^2 \right) + \frac{\pi}{12} r^2 \\ &= \frac{\pi}{3} - \frac{\sqrt{3}}{2} + \frac{\pi}{12} = \frac{\pi}{4} - \frac{\sqrt{3}}{2} \end{aligned}$$

## 3、四个圆的交（交集不为 $\phi$ ）

经分析可证，若四个圆的交集不为  $\phi$ 。则四个圆的圆心一定围成一个边长为 1 的正方形，这四个圆心按照顺时针（或逆时针方向）一定形成 4 个  $\alpha$  型交，四个圆的交集如图中阴影部分所示，设其阴影部分面积为  $S$ ，则：



$$\begin{aligned} S &= 4 \cdot \left( \frac{1}{2} r^2 \sin^2 15^\circ + \frac{\pi}{12} r^2 - \frac{1}{2} \sin 30^\circ r^2 \right) \\ &= 4 \sin^2 15^\circ + \frac{\pi}{12} - \frac{1}{4} \end{aligned}$$

由此可知，任意 5 个以上互不重合的单位圆的交集必为  $\phi$ 。

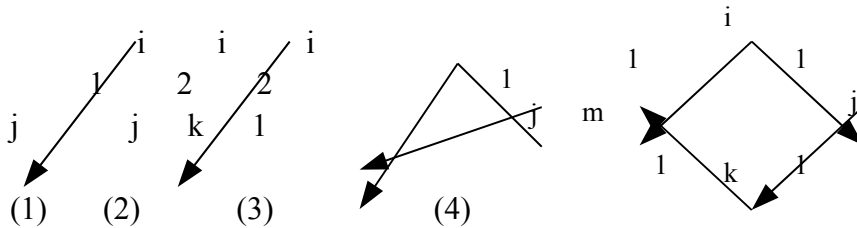
分析至此，我们可以知道，任意多个单位圆的交集都可以通过 2、3、4 个圆的交而获得，那么任意多个单位圆的并集呢？由交集到并集，这使我们想起了容斥原理，于是得出：

$$|A_1 \cup A_2 \cup \dots \cup A_n| = \sum_{i=1}^n |A_i| - \sum_{i=1}^{n-1} \sum_{j=i+1}^n |A_i \cap A_j| + \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} \sum_{k=j+1}^n |A_i \cap A_j \cap A_k| - \sum_{i=1}^{n-3} \sum_{j=i+1}^{n-2} \sum_{k=j+1}^{n-1} \sum_{m=k+1}^n |A_i \cap A_j \cap A_k \cap A_m|$$

模型有了，但是平面上的位置关系如何来表示呢？我们用带权有向图来有表示单位圆间的关系，边上的权函数定义如下：

$$C(i, j) = \begin{cases} 0 & A_i \cap A_j = \emptyset \\ 1 & A_i \text{ 与 } A_j \text{ 为 } \alpha \text{ 型交} \\ 2 & A_i \text{ 与 } A_j \text{ 的 } \beta \text{ 型交} \end{cases}$$

于是所有单位圆之间的信息便可由一个三角矩阵表示出来。两个圆、三个圆、四个圆相交的情况可由下图表示：



(1) 图表示两圆为  $\alpha$  型交的圆；(2) 图表示两圆为  $\beta$  型为圆；(3) 图表示 3 个圆相交的图，在 3 边中有边权为 2，其余两边权为 1；(4) 图为四个圆相交时的图。

题目标分析至此，我们便可轻松地设计出算法。

## 算法 2

**Func** dissimilarity\_function(k1, k2:integer):integer;

**Begin**

l:=abs(x[k1]-x[k2])+abs(y[k1]-y[k2]);

//计算异型函数的值

```

        if l>2 then return(0)
            else return(1);
    End; dissimilaruty_function
Proc Arithmetic2;
Begin
    count1:=0; //count1 为  $\alpha$  型交的个数
    count2:=0; //count2 为  $\beta$  型交的个数
    area:=n*pi; //当所有圆都不相交时的面积值
    for i:=1 to n-1 do
        for j:=i+1 to n do
            begin
                list[i,j]:=dissimilaruty_function(i,j);
                if list[i,j]=1 then count1:=count1+1; //两圆为  $\alpha$ 
型交
                                else if list[i,j]=2 then
count2:=count2+1;
                                //两圆为  $\beta$  型交
            end;//判断两个圆的相交情况
        area:=area-count1*s1-count2*s2;
        count1:=0;
        for i:=1 to n-2 do
            for j:=i+1 to n-1 do
                for k:=j+1 to n do
                    begin
                        check:=true;
                        p:=list[i,j]+list[j,k]+list[i,k];
                        if (list[i,j]=0) or (list[j,k]=0) or
(list[i,k]=0)
                            then check:=false;
                        if (p=4) and check then //三边的权值都不为 0 且权
值之和为 4
                            begin
                                count1:=count1+1; //三个圆的交不为空的个数
                                if list[i,j]=2 then info[i,k]:=2
                                    else if list[j,k]=2 then info[j,k]:=2
                                        else if list[i,k]=2 then
info[i,k]:=2;

```

```

        end; //info 供判断四个圆的交的情况时使用
    end; //判断三个圆的交的情况
    area:=area+s3*count1;
    count1:=0;
    for i:=1 to n-2 do
        for j:=i+1 to n-1 do
            for k:=j+1 to n do
                if (j<>k) and (info[i,j]=2) and (list[j,k]=1) and
                    (list[i,k]=1) then count1:=count1+1;
                //四个圆的交的个数
            end;
        end;
    end;
    area:=area-s4*count1;
End; Arithmetic2

```

这种算法建立在对问题进行深入分析，数学抽象的基础之上的，因而无论在时间上还是在空间上都是较优的。更为重要的是，这种算法比离散化算法更精确，更具一般性，能够解决诸如图形并等一系列问题。且算法的实质是一种计数问题，具有较强的可计算性。

**例 3.** 一场激烈的足球赛开始前，售票工作正在紧张的进行中，每张球票为 50 元。现有  $2n$  个人排队等待购票，其中有  $n$  个人手持 50 元的钞票，另外  $n$  个人手持 100 元的钞票，假设开始售票时售票处没有零钱。问这  $2n$  个人有多少种排队方式，使售票处不至出现找不开钱的局面？

这是一道典型的组合计数问题。从表面上看很难找出规律，下面我们基于本题建立几个模型，最终揭示问题的本质。

### I. 搜索模型

我们用深度优先搜索（DFS）算法来直观地模拟所有情况。

算法中指定一变量  $k$  记录售票处有 50 元钞票的张数，初始时令  $k=0$ ，若某人手持 100 元钞票且  $k=0$  时则回溯，否则继续递归。若

第  $2n$  个人购完票即递归进行到第  $2n$  层时计数器累加 1。递归结束后，计数器中的值便为排队总数。

### 算法 3-1

```

Proc DFS(i:integer); //I 为递归层数
Begin
  for j:=0 to 1 do //j=0 表示某人手持 50 元的钞票，j=1 表示某人手持 100 元钞票
    begin
      if (j=0) then begin
        k:=k+1; //k 表示计数器
        m:=m+1; //m 表示有多少人手持 50 元钞票
        购票
        if (m=n) then total:=total+1 //若已有
        n 个人手持 50 元钞票购票，那么其余手持 100 元钞票购票的人一定能
        找开钱。
        else dfs(i+1);
        k:=k-1;
        m:=m-1;
      end
    else begin //表示手持 100 元钞票时
      if k>0 then
        begin
          k:=k-1;
          dfs(i+1);
          k:=k+1;
        end;
      end;
    endfor;
  End; dfs

```

由于本算法的实质是模拟，因此算法实现起来时间复杂度较高，为指数级，这种算法严格限制了问题的规模，因而不是一个好的算法。

## II. 栈模型

通过对问题的分析我们可以得出这样的结论：在任何时刻，

若第  $n$  个人手持 100 元的钞票买票，则在此之前，定有  $m$  个人手持 50 元的钞票买票，使得  $m \geq n$ ，我们通过分析还将得出：售票处收到的 50 元的钞票最终将全部找出，售票处收到的 100 元的钞票最终将全部留下，且一旦收到一张面值为 100 元的钞票，则一定找出一张面值为 50 元的钞票。由此我们想到了用栈来表示这一过程：若某人手持一张 50 元的钱币购票，相当于一个元素进栈；若某人手持一张 100 元的钱币购票，相当于一个元素出栈。则问题转化为：若  $1 \sim n$  共  $n$  个元素依次进栈，问共有多少种出栈顺序。

$n$  个元素的全排列共有  $n!$  个，那么这  $n!$  种方案是否都是可能的出栈顺序呢？答案是否定的，我们可以证明，若  $a_1, a_2, \dots, a_n$  是可能的依次出栈顺序，则一定不存在这样的情况：使得  $i < j < k$  且  $a_j < a_k < a_i$ ，证明如下：

若  $i < j < k$ ，说明  $a_i$  最先出栈， $a_j$  次之， $a_k$  最后出栈，下面分两种情况讨论：

(i) 如果  $a_i > a_j$ ，那么当  $a_j$  出栈时，如果  $a_k$  已在栈中，则  $a_k$  比  $a_j$  先入栈，由输入  $a$  序列知：  $a_k < a_j$ ，所以有  $a_k < a_j < a_i$ ；当出栈时，如果  $a_k$  尚未入栈，则由输入序列知  $a_j < a_i < a_k$

(ii) 如果  $a_i < a_j$ ，那么当  $a_j$  出栈时，如果  $a_k$  已入栈，则由输

入序列知  $a_k < a_j$ , 而  $a_i$  与  $a_k$  的关系取决于  $a_i$  与  $a_k$  哪个先入栈。但无论怎样,  $a_i$  与  $a_k$  均小于  $a_j$ , 当  $a_i$  出栈时, 如果  $a_k$  尚未入栈, 则由输入序列知  $a_i < a_j < a_k$

因此, 输出序列中不可能出现当  $i < j < k$  时,  $a_j < a_k < a_i$

通过以上分析, 我们得出栈模型的算法, 算法先产生  $1 \sim n$  共  $n$  个数的全排列, 对于每种排列, 若符合前面所讲的出栈规则, 那么这  $n$  个排列便是一个可能的出栈序列, 计数器加 1, 当  $n$  个元素的全排列列举结束时, 计数器的值便是问题的解。

在此思想的指导下, 为了与模型 I 的算法进行比较, 我们在这里采用递归技术来产生  $n$  个元素的全排列, 若在每产生一个排列后进行该排列是否为可能输出栈序列的判定, 则算法的时间复杂度为  $O(n^n)$ , 与模型 I 的算法比较起来, 我们发现模型 II 中递归的深度降低, 栈的使用空间减小, 但在构造解答树的过程中, 每层扩展的结点数则大量增加, 而有些结点的增加是无意义的, 所以我们在实际的算法设计中可以一边生成排列一边进行可能输出序列的判定性检验, 若不满足条件, 则及时剪枝, 因而在  $n$  较大时该算法的时间复杂度应小于  $O(n^n)$

### 算法 3-2

```
Func check(s:integer):boolean; //判断 1~s 共 s 个元素的出栈
                               序列是否为可能的栈输出序列
begin
  for a:=1 to s-2 do
    for b:=a+1 to s-1 do
      if (data[b]<data[s]) and (data[s]<data[a]) then
```

```
return(false);
  return(true);
end;  check
Proc stack(i:integer); //产生 n 个元素的全排列
begin
  for j:=1 to n do
    if not(j in flag) then
      begin
        data[i]:=j;
        if check(i) then
          begin
            flag:=flag+[j];
            if i=n then total:=total+1  //计数器加 1
              else stack(i+1);
            flag:=flag-[j];
          end;
        endfor;
      end;  stack
```

但我们应该明确地看到，模型 I 与模型 II 在算法实现时解答树中的结点数目都是很多的，结点是栈所储存的信息，大量结点的出现必然影响算法可运行数据的规模，在模型 III 中，我们着重思考如何对问题进行数学抽象。

### III 递归算法：

令  $f(m,n)$  表示有  $m$  个人手持 50 元的钞票， $n$  个人手持 100 元的钞票时共有的方案总数。我们分情况来讨论这个问题。

(1)  $n=0$

$n=0$  意味着排队购票的所有人手中拿的都是 50 元的钱币，那么这  $m$  个人的排队总数为 1，即  $f(m,0)=1$



(2)  $m < n$ 

若排队购票的 $(m+n)$ 个人中有  $m$  个人手持 50 元的钞票,  $n$  个人手持 100 元的钞票, 当  $m < n$  时, 即使把  $m$  张 50 元的钞票都找出去, 仍会出现找不开钱的局面, 所以这时排队总数为 0, 即  $f(m,n)=0$

## (3) 其它情况

我们思考  $(m+n)$  个人排队购票的情景, 第 $(m+n)$ 个人站在第  $(m+n-1)$  个人的后面, 则第  $(m+n)$  个人的排队方式可由下列两种情况获得:

① 第  $(m+n)$  个人手持 100 元的钞票, 则在他之前的  $(m+n-1)$  个人中有  $m$  个人手持 50 元的钞票, 有  $(n-1)$  个人手持 100 元的钞票, 此种情况共有  $f(m,n-1)$

② 第  $(m+n)$  个人手持 50 元的钞票, 则在他之前的  $(m+n-1)$  个人中有  $(m-1)$  个人手持 50 元的钞票, 有  $n$  个人手持 100 元的钞票, 此种情况共有  $f(m-1,n)$

由加法原理得  $f(m,n)=f(m-1,n)+f(m,n-1)$

于是我们得到  $f(m,n)$  的计算公式:

$$f(m,n)=\begin{cases} 0 & m < n \\ 1 & n=0 \\ f(m,n-1)+f(m-1,n) & \end{cases} \quad (*)$$

于是我们可以根据 (\*) 式编写递归算法

### 算法 3-3

```

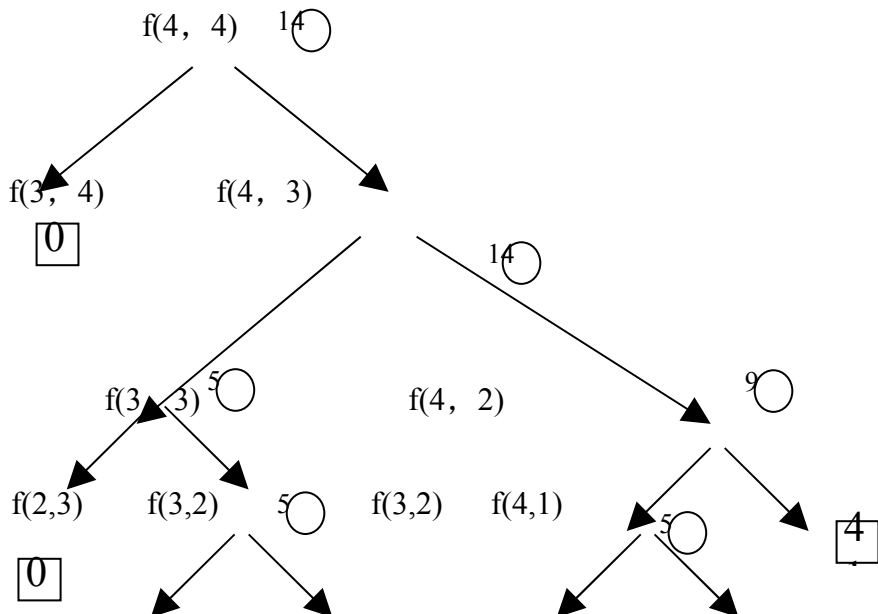
Func f(a,b:integer):longint;
begin
  if a<b then f:=0
    else if b=0 then f:=1
      else f:=f(a-1,b)+f(a,b-1);
end;  f

```

#### IV 递推算法

递归算法是由终止条件向初始条件推导，而递推算法是由初始条件向终止条件推导。可以说，它们本质上是相同的。那么，把递归算法改为递推算法的意义何在呢？我们运用 (\*) 式求解

$f(4,4)$ ，递归程序执行时构造的解答树如下：





```
for a:=2 to n do
    for b:=2 to a do data[a,b]:=data[a-1,b]
+data[a,b-1]; //递推
write(data[n,n]);
End. Arithmetic3_4
```

由此，本题的递推关系便建立起来，这个算法的时间复杂度为  $O(n^2)$ ，它与模型 III 的递归算法比较起来最大的优点在于它充分利用了已经得到的信息，从而使算法的时间复杂度大大降低，算法本身能够接受的规模也大大增加，达到了零信息冗余，可以说，这是一个较优化的算法。

### V 组合算法

我们下面用一种崭新的模型——二叉树来反映本题，我们依据以下原则建立一棵具有  $n$  个结点的二叉树。

(1) 若结点  $i$  是结点  $j$  的儿子结点，则  $i > j$ ，若结点  $i$  是结点  $k$  的左儿子，结点  $j$  是结点  $k$  的右儿子，则  $i < j$ 。

(2) 若结点  $i$  是结点  $j$  的儿子且  $i$  比  $j$  先出栈，则结点  $i$  是结点  $j$  的左儿子；若结点  $i$  比结点  $j$  后出栈，则结点  $i$  是结点  $j$  的右儿子。

由 (1) 可知，这棵具有几个结点的二叉树的先序遍历序列一定为  $1 \sim n$ ，由 (2) 可知，这棵树最左边的叶结点一定最先出栈，最后边的叶结点一定最后出栈。所以说，对于任意一棵具有几个结点的二叉树，它的前序遍历顺序便为  $1 \sim n$ ，即  $n$  个元素的入栈顺序，那么它的中序遍历顺序便是这  $n$  个元素的出栈顺序。

即  $2n$  个人的排队方案总数即为具有  $n$  个结点的二叉树的个数，

又因为具有  $n$  个结点的二叉树个数为  $C_n = \frac{1}{n+1} \binom{2n}{n}$ ，即 Catalan

数，所以本题的不同排列总数为  $\frac{1}{n+1} \binom{2n}{n}$

### 算法 3-5<sup>5</sup>

```
Prog Arithmetic3_5;
Begin
  read(n);
  total:=1;
  a:=n+2;
  b:=2;
  while (a<=2*n) do
  begin
    total:=total*a;
    while (total mod b=0) and (b<=n) do
    begin
      total:=total div b;
      b:=b+1;
    end;
    a:=a+1;
  end;
  while b<n do
  begin
    total:=total div b;
    inc(b);
  end;
  write(total);
End.  Arithmetic3_5
```

本算法的时间复杂度为  $O(n)$ ，从建模方式看，组合算法的模型最抽象，也最不易理解，但这个模型却能抓住问题的本质，

---

<sup>5</sup> 由于该算法涉及除法运算，为了保证在程序执行过程的中间结果在长整型之内，此算法在求组合数时进行了优化。

因而具有极大的可计算性，达到了零信息冗余。

## 四 总 结

组合算法作为当代组合数学研究的重要组成部分，在基础理论研究和实践中发挥着越来越重要的作用，本文着重讨论组合算法的评价依据，初步揭示了组合算法的设计和优化的基本问题。总之，只有掌握好组合算法的通用性，可计算性和信息冗余量的组合算法评价原则，才能设计出高效的组合算法。

### 【附 录】

#### 【参考文献】

- [1] 《组合数学》 卢开澄 清华大学出版社（1999）
- [2] 《组合数学引论》 孙淑玲 许胤龙 中国科学技术大学出版社（1999）
- [3] 《青少年国际和全国信息学（计算机）奥林匹克竞赛指导——组合数学的算法与程序设计》 吴文虎 王建德 清华大学出版社（1997）
- [4] 《离散数学》 Richard Johnsonbaugh 电子工业出版社（1999）
- [5] 《算法与数据结构》 傅清祥 王晓东 电子工业出版社（1999）
- [6] 《人工智能导论》 林尧瑞 马少平 清华大学出版社（1999）

### 【算法比较实验】

为了更好地反映组合算法设计中的三原则对算法效率的影响，我们对“球迷购票问题”的五个模型进行了实验，其总结如下：

#### 1、 系统设置：

CPU: Intel 633 Celeron

RAM: 128MB

OS: Windows Me

算法运行环境: Turbo Pascal 7.0

## 2、 规模确定:

由于此实验的目的是确定模型的优劣, 所以测试数据所得结果控制在长整型以内。由计算得到  $1 \leq n \leq 17$ 。为了更好地反映算法的效率, 尤其是信息冗余对算法效率的影响, 在进行  $n$  值选取时, 我们选的是不均匀的。

## 3、 时间测定算法:

**Begin**

t:=meml[\$40:\$6c];

主程序;

t:=(meml[\$40:\$6c]-t)/18.2;

out(t)

**end.**

## 4、 实验结果

N	结果	模 型 1 运 行 时 间	模 型 2 运 行 时 间	模 型 3 运 行 时 间	模 型 4 运 行 时 间	模 型 5 运 行 时 间
5	42	0.0000	0.0000	0.0000	0.0000	0.0000
10	16796	0.0000	1.1538	0.0000	0.0000	0.0000
13	7429000	0.1099	>60	0.2747	0.0000	0.0000
15	9694845	1.1538	>60	3.6813	0.0000	0.0000
16	35357670	4.2308	>60	13.5165	0.0000	0.0000
17	12964479 0	15.3846	>60	49.5055	0.0000	0.0000

(时间单位: s)

**【源程序】****[1] 算法 1—1 的源程序**

```
Program Arithmtic1_1;
Var n,m:array[0..100] of longint;
    t,i:integer;
Begin
    write('Please input t:');
    readln(t);
    n[0]:=1;
    m[0]:=0;
    for i:=1 to t do
    begin
        n[i]:=m[i-1];
        m[i]:=3*n[i-1]+2*m[i-1];
    end;
    writeln('N=',n[t]);
    writeln('M=',m[t]);
End.
```

**[2] 算法 1—2 的源程序**

```
Program Arithmetic1_2;
var t:integer;
    n,m:longint;
begin
    write('Please input t:');
    readln(t);
    n:=trunc(exp(t*ln(3)));
    m:=trunc(exp((t+1)*ln(3)));
    if odd(t) then begin
        n:=n-3;
        m:=m+3;
    end
    else begin
```



```
        n:=n+3;
        m:=m-3;
    end;
n:=trunc(n/4);
m:=trunc(m/4);
writeln(' N=',n);
writeln(' m=',m);
end.
```

### [3] 算法 2 的源程序

```
Program Arithmetic2;
Const InFile=' input.txt' ;
      OutFile=' output.txt' ;
      pi=3.1415926535;
      s1=2/3*pi-1.732/2;
      s2=pi/2-1;
      s3=5/12*pi-1.732/2;
Var list,info:Array[1..100,1..100] of shortint;
    x,y:      Array[1..100] of integer;
    n:        Integer;
    area,s4:  real;
Procedure init;
Var f:Text;
    a:integer;
Begin
    assign(f,InFile);
    reset(f);
    readln(f,n);
    for a:=1 to n do
        read(f,x[a],y[a]);
    close(f);
    s4:=4*sin(pi/12)*sin(pi/12)+pi/12-1/4;
End;
Function dissimilaruty_function(k1,k2:integer):integer;
Var l:integer;
Begin
    l:=abs(x[k1]-x[k2])+abs(y[k1]-y[k2]);
```

```
        if l>2 then dissimilaruty_function:=0
            else dissimilaruty_function:=1;
    End;
Procedure done;
    var i, j, k, p, count1, count2: integer;
        check:                      boolean;
Begin
    count1:=0;
    count2:=0;
    area:=n*pi;
    for i:=1 to n-1 do
        for j:=i+1 to n do
            begin
                list[i, j]:=dissimilaruty_function(i, j);
                if list[i, j]=1 then inc(count1)
                    else if list[i, j]=2 then inc(count2);
            end;
        area:=area-count1*s1-count2*s2;
        count1:=0;
        for i:=1 to n-2 do
            for j:=i+1 to n-1 do
                for k:=j+1 to n do
                    begin
                        check:=true;
                        p:=list[i, j]+list[j, k]+list[i, k];
                        if (list[i, j]=0) or (list[j, k]=0) or (list[i, k]=0)
                            then check:=false;
                        if (p=4) and check then
                            begin
                                inc(count1);
                                if list[i, j]=2 then info[i, k]:=2
                                    else if list[j, k]=2 then info[j, k]:=2
                                        else if list[i, k]=2 then info[i, k]:=2;
                            end;
                    end;
                end;
            area:=area+s3*count1;
            count1:=0;
```

```

    for i:=1 to n-2 do
      for j:=i+1 to n-1 do
        for k:=j+1 to n do
          if (j<>k) and (info[i,j]=2) and (list[j,k]=1) and
(list[i,k]=1) then
            inc(count1);
          area:=area-s4*count1;
        End;
      Procedure out;
        Var f:text;
        Begin
          assign(f,OutFile);
          rewrite(f);
          writeln(f,area:0:4);
          close(f);
        End;
      Begin
        Init;
        Done;
        Out;
      End.

```

#### [4] 算法 3—1 的源程序

```

{$A+, B-, D-, E+, F-, G+, I-, L-, N-, O-, P-, Q-, R-, S-, T-, V+, X+}
{$M 65520, 0, 655360}
Program Arithmetic3_1;
Var n,k,m:integer;
    total:longint;
Procedure DFS(i:integer);
  Var j:integer;
  Begin
    for j:=0 to 1 do
      begin
        if (j=0) then begin
          inc(k);
          inc(m);
          if (m=n) then inc(total)

```

```
                else dfs(i+1);
            dec(k);
            dec(m);
        end
    else begin
        if k>0 then
            begin
                dec(k);
                dfs(i+1);
                inc(k);
            end;
        end;
    end;
end;
End;
Begin
    read(n);
    m:=0;
    k:=0;
    dfs(1);
    writeln(total);
End.
```

### [5]算法 3—2 的源程序

```
{ $A+, B-, D+, E+, F-, G-, I+, L+, N-, O-, P-, Q-, R-, S+, T-, V+, X+ }
{ $M 65520, 0, 655360 }
program Arithmetic3_2;
var n:integer;
    data:Array[1..100] of integer;
    flag:set of byte;
    total:longint;
function check(s:integer):boolean;
var a,b:integer;
begin
    check:=false;
    for a:=1 to s-2 do
        for b:=a+1 to s-1 do
            if (data[b]<data[s]) and (data[s]<data[a]) then exit;
```

```
    check:=true;
end;
procedure stack(i:integer);
var j:integer;
begin
    for j:=1 to n do
        if not(j in flag) then
            begin
                data[i]:=j;
                if check(i) then
                    begin
                        flag:=flag+[j];
                        if i=n then inc(total)
                            else stack(i+1);
                        flag:=flag-[j];
                    end;
                end;
            end;
    end;
begin
    read(n);
    stack(1);
    writeln(total);
end.
```

[6]算法 3—3 的源程序

```
{ $A+, B-, D+, E+, F-, G-, I+, L+, N-, O-, P-, Q-, R-, S+, T-, V+, X+ }
{ $M 65520, 0, 655360 }
program Arithmetic3_3;
var n:integer;
function f(a,b:integer):longint;
begin
    if a<b then f:=0
        else if b=0 then f:=1
            else f:=f(a-1,b)+f(a,b-1);
end;
begin
    read(n);
```

```
writeln(f(n,n));  
end.
```

#### [7] 算法 3—4 的源程序

```
Program Arithmetic3_4;  
Var data :array[1..20,1..20] of longint;  
    a,b,n:integer;  
Begin  
    readln(n);  
    for a:=1 to n do data[a,1]:=a;  
    for a:=2 to n do  
        for b:=2 to a do data[a,b]:=data[a-1,b]+data[a,b-1];  
    writeln(data[n,n]);  
End.
```

#### [8] 算法 3—5 的源程序

```
Program Arithmetic3_5;  
Var n,a,b:integer;  
    total:longint;  
Begin  
    readln(n);  
    total:=1;  
    a:=n+2;  
    b:=2;  
    while (a<=2*n) do  
        begin  
            total:=total*a;  
            while (total mod b=0) and (b<=n) do  
                begin  
                    total:=total div b;  
                    inc(b);  
                end;  
            inc(a);  
        end;  
    while b<n do  
        begin
```

```
total:=total div b;  
inc(b);  
end;  
writeln(total);  
End.
```

## 贪心策略的特点与在信息学竞赛 中的应用

**【关键字】** 贪心策略      特点      理论基础      应用

## 【摘要】

本文着重探讨的是贪心策略的数学模型、理论基础（“矩形胚”结构）和贪心策略的特点。（贪心选择性质和局部最优解）介绍了 3 种体现“贪心”思想的图形算法：Dijkstra 算法、Prim 算法和 Kruskal 算法，并着重给出了近几年来在各级各类程序设计竞赛中出现的一些题目。

## 【正文】

### 一、 引 论

信息，人类社会发展的标志。人类对信息的记载，可以追溯到原始社会。在漫长的人类社会发展过程中，伴随着科学技术的发展，人类对客观世界的认识不断加深，现实世界的信息量急剧增大。为了满足人们对大数据量信息处理的渴望，1946 年世界上第一台电子数字计算机 ENIAC 应运而生。在此后的半个世纪中，为解决各种实际问题，计算机算法学得到了飞速的发展。线形规划、动态规划等一系列运筹学模型纷纷运用到计算机算法学中，解决了诸如经济决策等一系列现实问题。在众多的计算机解题策略中，贪心策略可以算得上是最接近人们日常思维的一种解题策略，正基于此，贪心策略在各级各类信息学竞赛、尤其在对 NPC 类问题的求解中发挥着越来越重要的作用。

### 二、 贪心策略的定义

**【定义 1】** 贪心策略是指从问题的初始状态出发，通过若干



次的贪心选择而得出最优值(或较优解)的一种解题方法。

其实，从“贪心策略”一词我们便可以看出，贪心策略总是做出在当前看来是最优的选择，也就是说贪心策略并不是从整体上加以考虑，它所做出的选择只是在某种意义上的局部最优解，而许多问题自身的特性决定了该题运用贪心策略可以得到最优解或较优解。

### 三、贪心算法的特点

通过上文的介绍，可能有人会问：贪心算法有什么样的特点呢？我认为，适用于贪心算法解决的问题应具有以下 2 个特点：

#### 1、贪心选择性质：

所谓贪心选择性质是指应用同一规则  $f$ ，将原问题变为一个相似的、但规模更小的子问题、而后的每一步都是当前看似最佳的选择。这种选择依赖于已做出的选择，但不依赖于未做出的选择。从全局来看，运用贪心策略解决的问题在程序的运行过程中无回溯过程。关于贪心选择性质，读者可在后文给出的贪心策略状态空间图中得到深刻地体会。

#### 2、局部最优解：

我们通过特点 2 向大家介绍了贪心策略的数学描述。由于运用贪心策略解题在每一次都取得了最优解，但能够保证局部最优解得不一定是贪心算法。如大家所熟悉得动态规划算法就可以满足局部最优解，在广度优先搜索（BFS）中的解题过程亦可以满足局部最优解。

在遇到具体问题时，许多选手往往分不清哪些题该用贪心策略求解，哪些题该用动态规划法求解。在此，我们对两种解题策略进行比较。


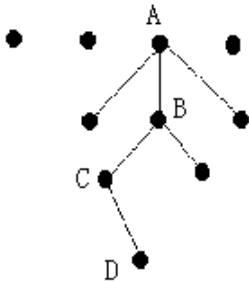
	贪心策略	动态规划
相同点	1 时间效率较高、适用于对问题的求解有严格时间限制的题目。 2 均能保证局部最优解。	
不同点	所占空间较小	所占空间较大
	某次具体选择必是最优选择	某次具体选择不一定是最优选择
	解题大体框架：  线形    A—B—C—D为最优选择，每一中间结点B、C也为最优选择	解题大体框架：  图或树    A—B—C—D为最优解，B、C不一定是最优选择

图 1

【引例】 在一个  $N \times M$  的方格阵中，每一格子赋予一个数（即为权）。规定每次移动时只能向上或向右。现试找出一条路径，使其从左下角至右上角所经过的权之和最大。

我们以  $2 \times 3$  的矩阵为例。

3	4	6
1	2	10

若按贪心策略求解，所得路径为：1→3→4→6；  
图 二

若按动态规划法求解，所得路径为：1→2→10→6。

由于贪心策略自身的特点，使得数字 10 所在的格子成为一个“坏格子”，即运用贪心策略找不到它，而运用动态规划法求解的第一步（1→2）并不是最优选择，但却保证了全局最优解；运用贪心策略求解的第一步（1→3）保证了

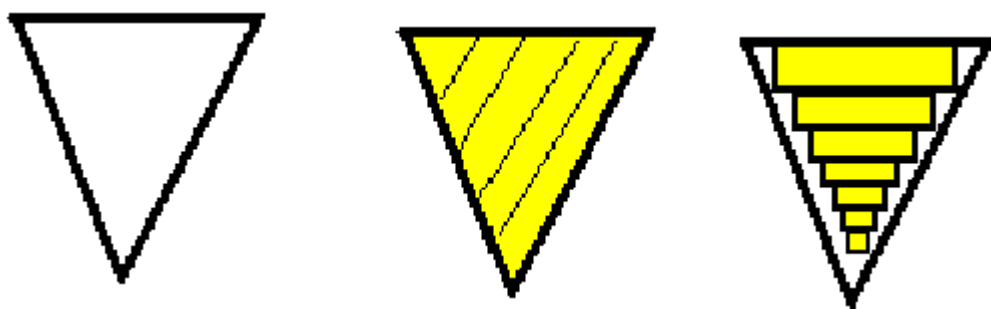
$a_{11}$	$a_{12}$	$a_{13}$	$a_{14}$
$a_{21}$	$a_{22}$	$a_{23}$	$a_{24}$
$a_{31}$	$a_{32}$	$a_{33}$	$a_{34}$

图 三

局部最优解，却无法保证全局最优解。我们若用图 3 所示的  $N \times M$  的矩阵表示一组数，设运用与引例同样的移动规则后得到了由若干个元素组成的数列  $A$ 。可得如下结论：

若  $a_{n+1,1} > a_{n,2}$ ，则  $a_{n,2}$ 、 $a_{n,3}$ 、 $\dots$ 、 $a_{n,m}$  一定不在数列  $A$  中。对于一元素  $a_{n,p}$  ( $2 \leq p \leq m$ )，设  $a_{n,p} = \infty$ ，若保证全局最优解，则  $a_{n,p}$  必在数列  $A$  中，但运用贪心策略求解时  $a_{n,p}$  不在数列  $A$  中。

由此可见，贪心策略并不到达问题状态的全部空间。若用空间图来表示贪心算法和动态规划算法（如下图），我们可以清楚地看到，贪心算法



问题状态空间

动态规划所用空间

贪心策略所用空间

图 4

是一种对输入数据进行不断收缩的过程。它并不到达问题的全部状态空间。这是由本文所述的贪心策略的线形解题框架所决定的。

## 四、 贪心策略的理论基础

### ——矩阵胚

正如前文所说的那样，贪心策略是最接近人类认知思维的一种解题策略。但是，越是显而易见的方法往往越难以证明。下面我们就来介绍贪心策略的理论——矩阵胚。

“矩阵胚”理论是一种能够确定贪心策略何时能够产生最优解的理论，虽然这套理论还很不完善，但在求解最优化问题时发挥着越来越重要的作用。

**【定义 3】** 矩阵胚是一个序对  $M=[S, I]$ ，其中  $S$  是一个有序非空集合， $I$  是  $S$  的一个非空子集，成为  $S$  的一个独立子集。

如果  $M$  是一个  $N \times M$  的矩阵的话，即

$$M = \begin{pmatrix} a_{11}, a_{12}, \dots, a_{1n} \\ \dots \dots \dots \\ a_{i1}, a_{i2}, \dots, a_{in} \\ \dots \dots \dots \\ a_{n1}, a_{n2}, \dots, a_{nn} \end{pmatrix}$$

$S$  是  $M$  的各个行,  $S = (a_1, a_2, \dots, a_n)$ ,  $I$  是线形无关的若干行  $a_i, a_j, a_p, \dots$

若  $M$  是无向图  $G$  的矩阵胚的话, 则  $S$  为图的边集,  $I$  是所有构成森林的一组边的子集。

如果对  $S$  的每一个元素  $X (X \in S)$  赋予一个正的权值  $W(X)$ , 则称矩阵胚  $M = (S, I)$  为一个加权矩阵胚。

适宜于用贪心策略来求解的许多问题都可以归结为在加权矩阵胚中找一个具有最大权值的独立子集的问题, 即给定一个加权矩阵胚,  $M = (S, I)$ , 若能找出一个独立且具有最大可能权值的子集  $A$ , 且  $A$  不被  $M$  中比它更大的独立子集所包含, 那么  $A$  为最优子集, 也是一个最大的独立子集。

我们认为, 针对绝大多数的信息学问题, 只要它具备了“矩阵胚”的结构, 便可用贪心策略求解。矩阵胚理论对于我们判断贪心策略是否适用于某一复杂问题是十分有效的。

## 五、 几种典型的贪心算法

贪心策略在图论中有着极其重要的应用。诸如 Kruskal、Prim、Dijkstra 等体现“贪心”思想的图形算法更是广泛地应用于树与

图的处理。下面就分别来介绍 Kruskal 算法、Prim 算法和 Dijkstra 算法。

### I、库鲁斯卡尔 (Kruskal) 算法

**【定义 4】** 设图  $G=(V, E)$  是一简单连通图,  $|V|=n$ ,  $|E|=m$ , 每条边  $e_i$  都给以权  $W_i$ ,  $W_i$  假定是边  $e_i$  的长度 (其他的也可以),  $i=1, 2, 3, \dots, m$ 。求图  $G$  的总长度最短的树, 这就是最短树问题。

kruskal 算法的基本思想是: 首先将赋权图  $G$  的边按权的升序排列, 不失一般性为:  $e_1, e_2, \dots, e_m$ 。其中  $W_i \leq W_{i+1}$ , 然后在不构成回路的条件下择优取进权最小的边。

其流程如下:

(1) 对属于  $E$  的边进行排序得  $e_1 \leq e_2 \leq \dots \leq e_m$ 。

(2) 初始化操作  $w \leftarrow 0$ ,  $T \leftarrow \phi$ ,  $k \leftarrow 0$ ,  $t \leftarrow 0$ ;

(3) 若  $t=n-1$ , 则转 (6), 否则转 (4)

(4) 若  $T \cup \{e_k\}$  构成一回路, 则作

**【 $k \leftarrow k+1$ , 转 (4)】**

(5)  $T \leftarrow T \cup \{e_k\}$ ,  $w \leftarrow w + W_k$ ,  $t \leftarrow t+1$ ,  $k \leftarrow k+1$ , 转

(3)

(6) 输出  $T$ ,  $w$ , 停止。

下面我们对这个算法的合理性进行证明。

设在最短树中, 有边  $\langle v_i, v_j \rangle$ , 连接两顶点  $v_i, v_j$ , 边  $\langle v_i, v_j \rangle$  的权为  $w_p$ , 若  $\langle v_i, v_j \rangle$  加入到树中不能保证树的总长度最短, 那么一定有另一条边  $\langle v_i, v_j \rangle$  或另两条边  $\langle v_i$

,  $\langle v_i, v_k \rangle$ 、 $\langle v_k, v_j \rangle$  , 且  $W(\langle v_i, v_j \rangle) < W_p$  或  $W(\langle v_i, v_k \rangle) + W(\langle v_k, v_j \rangle) < W_p$ , 因为  $\langle v_i, v_q \rangle$ 、 $\langle v_q, v_p \rangle$  不在最短树中, 可知当  $\langle v_i, v_q \rangle$ 、 $\langle v_q, v_p \rangle$  加入到树中时已构成回路, 此时程序终止。因为  $\langle v_i, v_k \rangle \in T$ ,  $\langle v_k, v_j \rangle \in T$  且  $W(\langle v_i, v_k \rangle) + W(\langle v_k, v_j \rangle) < W_p$ , 与程序流程矛盾。

## II、普林 (Prim) 算法:

Kruskal 算法采取在不构成回路的条件下, 优先选择长度最短的边作为最短树的边, 而 Prim 则是采取了另一种贪心策略。

已知图  $G = (V, E)$ ,  $V = \{v_1, v_2, v_3, \dots, v_n\}$ ,  $D = (d_{ij})_{n \times n}$  是图  $G$  的矩阵, 若  $\langle v_i, v_j \rangle \in E$ , 则令

Prim 算法的基本思想是: 从某一顶点 (设为  $v_1$ ) 开始, 令  $S \leftarrow \{v_1\}$ , 求  $V \setminus S$  中点与  $S$  中点  $v_1$  距离最短的点, 即从矩阵  $D$  的第一行元素中找到最小的元素, 设为  $d_{1j}$ , 则令  $S \leftarrow S \cup \{v_j\}$ , 继续求  $V \setminus S$  中点与  $S$  的距离最短的点, 设为  $v_k$ , 则令  $S \leftarrow S \cup \{v_k\}$ , 继续以上的步骤, 直到  $n$  个顶点用  $n-1$  条边连接起来为止。  
流程如下:

(1) 初始化操作:  $T \leftarrow \phi$ ,  $q(1) \leftarrow -1$ ,  $i$  从 2 到  $n$  作

【  $p(i) \leftarrow -1$ ,  $q(i) \leftarrow d_{i1}$  】 ,  $k \leftarrow 1$

(2) 若  $k \geq n$ , 则作 【输出  $T$ , 结束】

否则作  $[\min \leftarrow \infty, j \text{ 从 } 2 \text{ 到 } n \text{ 作}$

$[\text{若 } 0 < q(i) < \min \text{ 则作}$

$[\min \leftarrow q(i) \text{ } h \leftarrow j]$

$]$

$]$

(3)  $T \leftarrow T \cup \{h, p(h)\}, q(h) \leftarrow -1$

(4)  $j \text{ 从 } 2 \text{ 到 } n \text{ 作}$

$[\text{若 } d_{hj} < q(j) \text{ 则作 } [q(j) \leftarrow d_{hj}, p(j) \leftarrow h]]$

(5)  $k \leftarrow k + 1$ , 转(2)

算法中数组  $p(i)$  是用以记录和  $v_i$  点最接近的属于  $S$  的点,  $q(i)$  则是记录了  $v_i$  点和  $S$  中点的最短距离,  $q(i) = -1$  用以表示  $v_i$  点已进入集合  $S$ 。算法中第四步:  $v_n$  点进入  $S$  后, 对不属于  $S$  中的点  $v_j$  的  $p(j)$  和  $q(j)$  进行适当调整, 使之分别记录了所有属于  $S$  且和  $S$  距离最短的点和最短的距离, 点  $v_1, v_2, \dots, v_n$  分别用  $1, 2, \dots, n$  表示。

III、戴克斯德拉 (Dijkstra) 算法:

已知图  $G = (V, E)$ ,  $V = \{v_1, v_2, \dots, v_n\}$ , 起始顶点  $v_o$ , 求  $v_o$  点到其他各点的最短路径。

算法如下:

令  $S$  表示已求出最短路径的顶点集合。对于不在  $S$  中的顶点



$w$ , 令  $d(w)$  表示从  $v_o$  开始且通过  $S$  中的顶点到达  $w$  的最短路径的长度。 $S$  的初态为空, 若  $v_o$  通过  $S$  中的顶点到  $w$  有路径, 则  $d(w)$  为  $v_o$  到  $w$  的最短的一条路径的长度; 若  $v_o$  没有经  $S$  中的顶点到  $w$  的路径, 则  $d(w) = \infty$ , 初始时设  $d(v_n) = 0$ , 除  $v_o$  外的所有顶点  $v_i, v_j$  ( $v_i \in V$  且  $v_i \neq v_j$ ),  $d(v_i) = \infty$ 。

## 六、 贪心策略的应用

在现实世界中, 我们可以将问题分为两大类。其中一类被称为  $P$  类问题, 它存在有效算法, 可求得最优解; 另一类问题被称为  $NPC$  类问题, 这类问题到目前为止人们尚未找到求得最优解的有效算法, 这就需要每一位程序设计人员根据自己对题目的理解设计出求较优解的方法。下面我们着重分析贪心策略在求解  $P$  类问题中的应用。

### § 6.1 贪心策略在 $P$ 类问题求解中的应用

#### § 6.1.1 贪心策略在求 $P$ 类最优解问题中的应用

在现实生活中,  $P$  类问题是十分有限的, 而  $NPC$  类问题则是普遍的、广泛的。在国际信息学奥林匹克竞赛的发展过程中, 由于受到评测手段的限制, 在 1989 年至 1996 年的 8 年赛事中, 始终是以  $P$  类问题为主的, 且只允许求最优解。在这些问题中, 有

的题目可以用贪心策略来直接求解，有的题目运用贪心策略后可以使问题得到极大的简化，使得程序对大信息量的处理提供了可能。

### [例 1]删数问题

**试题描述** 键盘输入一个高精度的正整数  $N$ ，去掉其中任意  $S$  个数字后剩下的数字按左右次序组成一个新的正整数。对给定的  $N$  和  $S$ ，寻找一种删数规则使得剩下得数字组成的新数最小。

**试题背景** 此题出自 NOI94

**试题分析** 这是一道运用贪心策略求解的典型问题。此题所需处理的数据从表面上看是一个整数。其实，大家通过对此题得深入分析便知：本题所给出的高精度正整数在具体做题时将它看作由若干个数字所组成的一串数，这是求解本题的一个重要突破。这样便建立起了贪心策略的数学描述。

### [例 2]数列极差问题

**试题描述** 在黑板上写了  $N$  个正整数作成的一个数列，进行如下操作：每一次擦去其中的两个数  $a$  和  $b$ ，然后在数列中加入一个数  $a \times b + 1$ ，如此下去直至黑板上剩下一个数，在所有按这种操作方式最后得到的数中，最大的  $\max$ ，最小的为  $\min$ ，则该数列的极差定义为  $M = \max - \min$ 。

编程任务：对于给定的数列，编程计算出极差  $M$ 。

**试题背景** 这是 1997 年福建队选拔赛的一道题目。

**试题分析** 当看到此题时，我们会发现求  $\max$  与求  $\min$  是两个相似的过程。若我们把求解  $\max$  与  $\min$  的过程分开，着重探讨求

max 的问题。

下面我们以求 max 为例来讨论此题用贪心策略求解的合理性。

讨论：假设经  $(N-3)$  次变换后得到 3 个数：a,b,max'

$(\max' \geq a \geq b)$ ，其中  $\max'$  是  $(N-2)$  个数经  $(N-3)$  次  $f$  变换后所得的最大值，此时有两种求值方式，设其所求值分别为  $Z_1, Z_2$ ，则有： $Z_1 = (a \times b + 1) \times \max' + 1$ ， $Z_2 = (a \times \max' + 1) \times b + 1$  所以  $Z_1 - Z_2 = \max' - b \geq 0$  若经  $(N-2)$  次变换后所得的 3 个数为： $m, a, b$  ( $m \geq a \geq b$ ) 且  $m$  不为  $(N-2)$  次变换后的最大值，即  $m < \max'$  则此时所求得的最大值为： $Z_3 = (a \times b + 1) \times m + 1$  此时  $Z_1 - Z_3 = (1 + a \cdot b) (\max' - m) > 0$  所以此时不为最优解。

所以若使第  $k$  ( $1 \leq k \leq N-1$ ) 次变换后所得值最大，必使  $(k-1)$  次变换后所得值最大（符合贪心策略的特点 2），在进行第  $k$  次变换时，只需取在进行  $(k-1)$  次变换后所得数列中的两最小数  $p, q$  施加  $f$  操作： $p \leftarrow p \times q + 1, q \leftarrow -\infty$  即可（符合贪心策略特点 1），因此此题可用贪心策略求解。

讨论完毕。

在求  $\min$  时，我们只需在每次变换的数列中找到两个最大数  $p, q$  施加作用  $f$ ： $p \leftarrow p \times q + 1, q \leftarrow -\infty$  即可。原理同上。

这是一道两次运用贪心策略解决的一道问题，它要求选手有较高的数学推理能力。

[例 3] 最优乘车问题

**试题描述** H城是一个旅游胜地，每年都有成千上万的人前来观光。为方便游客，巴士公司在各个旅游景点及宾馆、饭店等地都设置了巴士站，并开通了一些单向巴士线路。每条单向巴士线路从某个巴士站出发，依次途径若干个巴士站，最终到达终点巴士站。

阿昌最近到H城旅游，住在C U P饭店。他很想去看S公园游玩。听人说，从C U P饭店到S公园可能有也可能没有直通巴士。如果没有，就要换乘不同线路的单向巴士，还有可能无法乘巴士到达。

现在用整数 1, 2, ..., n 给H城的所有巴士站编号，约定C U P饭店的巴士站编号为 1，S 公园巴士站的编号为N。写一个程序，帮助阿昌寻找一个最优乘车方案，使他在从C U P饭店到S 公园的过程中换车的次数最少。

**试题背景** 出自NOI 97

**试题分析** 此题看上去很像一道搜索问题。在搜索问题中，我们所求的使经过车站数最少的方案，而本题所求解的使换车次数最少的方案。这两种情况的解是否完全相同呢？我们来看一个实例：

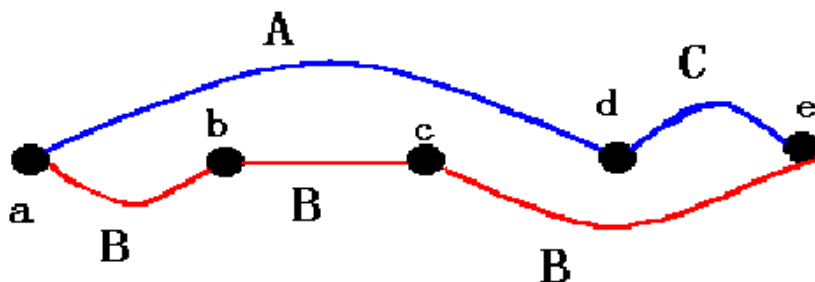


图 5

如图5所示：共有5个车站（分别为a、b、c、d、e），共有3条巴士线（线路A：a → d；线路B：a → b → c → e；线路C：d → e）。此时要使换车次数最少，应乘坐线路B的巴士，路线为：a → b → c → e，换车次数为0；要使途经车站数

最少，乘坐线路应为  $a \rightarrow d \rightarrow e$ ，换车次数为 1。所以说使换车次数最少的路线和使途经车站数最少的方案不一定相同。这使不能用搜索法求解此问题的原因之一。

原因之二，来自对数学模型的分析。我们根据题中所给数据来建立一个图后会发现该图中存在大量的环，因而不适合用搜索法求解。

题目分析到这里，我们可以发现此题与 NOI93 的求最长路径问题有相似之处。其实，此题完全可以套用上文所提到的 Dijkstra 算法来求解。

以上三道题只是使用了单一的贪心策略来求解的。而从近几年的信息学奥林匹克竞赛的命题方向上看，题目更加灵活，同时测试数据较大，规定的出解时间较短。在一些问题中，我们采用贪心策略对问题化简，从而使程序具有更高的效率。

#### [例 4] 最佳浏览路线问题

**试题描述** 某旅游区的街道成网格状（见图），其中东西向的街道都是旅游街，南北向的街道都是林荫道。由于游客众多，旅游街被规定为单行道。游客在旅游街上只能从西向东走，在林荫道上既可以由南向北走，也可以从北向南走。

阿隆想到这个旅游区游玩。他的好友阿福给了他一些建议，用分值表示所有旅游街相邻两个路口之间的道路值得浏览得程度，分值从 -100 到 100 的整数，所有林荫道不打分。所有分值不可能全是负值。

例如下图是被打过分的某旅游区的街道图：

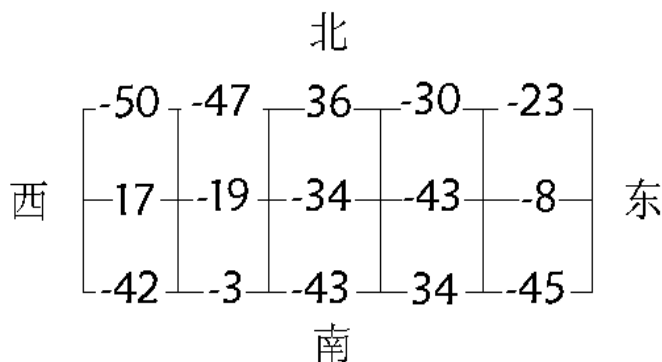


图 6

阿隆可以从任一路口开始浏览,在任一路口结束浏览。请你写一个程序,帮助阿隆寻找一条最佳的浏览路线,使得这条路线的所有分值总和最大。

**试题背景** 这道题同样出自 NO I 9 7, ' 9 7 国际大学生程序设计竞赛的第二题(吉尔的又一个骑车问题)与本题是属于本质相同的题目。

**试题分析** 由于林荫道不打分,也就是说,无论游客在林荫道中怎么走,都不会影响得分。因题可知,若游客需经过某一系列的旅游街,则他一定要经过这一列的  $M$  条旅游街中分值最大的一条,才会使他所经路线的总分值最大。这是一种贪心策略。贪心策略的目的是降维,使题目所给出的一个矩阵便为一个数列。下一步便是如何对这个数列进行处理。在这一步,很多人用动态规划法求解,这种算法的时间复杂度为  $O(n^2)$ ,当林荫道较多时,效率明显下降。其实在这一步我们同样可以采用贪心法求解。这时的时间复杂度为  $O(n)$ 。

### § 6.1.2 贪心策略在求 P 类较优解问题中的应用

正如其他学科奥林匹克竞赛一样,国际信息学奥赛的发展同样经历了一个逐步成熟的发展过程。回顾十余年赛事的发展,我们不妨将国际信息学奥赛的发展分为两个阶段:第一阶段是 1989—1996 年,这一时期奥赛题目的特点是:试题全部为 P 类问题,且只允许求最优解,题目的设计强调对选手基本算法的掌握。第二阶段为 1997 年至今。在南非举行的 IOI97 中,命题方向一举突破传统模式, NPC 类问题在竞赛中大量出现,每道题目到具有一定的实际背景,引进了崭新的程序评测机制。在求解 P

类问题时允许得出较优解并得到相应的分数。这些变化无疑更好地考察了选手的综合素质。在对 P 类较优解问题的求解过程中，贪心策略无疑扮演着重要角色。IOI97 中的障碍物探测器问题便是运用贪心策略来求得较优解的 P 类问题。

### [例 5] 障碍物探测器问题

**试题描述** 有一个登陆舱（POD），里面装有许多障碍物探测车（MEV），将在火星表面着陆，着陆后，探测车离开登陆舱向相距不远的先期到达的传送器（Transmitter）移动。MEV 一边移动，采集岩石（ROCK）标本，岩石由第一个访问到它的 MEV 所采集，每块岩石只能被采集一次，但是这以后，其他 MEV 可以从该处通过。探测车 MEV 不能通过有障碍的地面。

本题限定探测车 MEV 只能沿着格子向南或向东从登陆处向传送器 transmitter 移动，允许多个探测车 MEV 在同一时间占据同一位置。

警告：如果某个探测车 MEV 在到达传送器以前不能在继续合法前进时，则车中的石块必定不可挽回地全部丢失。

任务：计算机探测车的每一步移动，使其送到传送器的岩石标本的数量尽可能多。这两项都做到会使你的得分最高。

输入：火星表面上登陆舱 POD 和传送器之间的位置用网格 P 和 Q 表示，登陆舱 POD 的位置总是在 (1, 1) 点，传送器的位置总是在 (P, Q) 点。

火星上的不同表面用三中不同的数字符号来表示：

- 0 代表平坦无障碍
- 1 代表障碍
- 2 代表石块

输入文件的第一行为探测车的个数，第二行为  $P$  的值，第三行为  $Q$  的值。接下来的  $Q$  行为一个  $Q \times P$  的矩阵。

输出：表示 MEV 移向 transmitter 的行动序列。每行包含探测车号和一个数，0 或 1，这里 0 表示向南移动，1 表示向东移动。

得分：分数的计算将根据收集的岩石样本（取到传送器上）的数目，MEV 到达传送器和不到达传送器的数目有关

- 非法移动将导致求解无效，并记作零分，当 MEV 的障碍物上移动或移出网格，即视为非法。
- 得分 = (收集的样品并取到传送器上的数目 + MEV 到达传送器上的数目 - MEV 没有到达传送器上的数目) 与应得的最大的数目之比 (%)
- 最高分为 100%，最低分为 0%

**试题背景** IOI'97 中的第一试第一题。国际信息学奥赛中出现的  
第一道 NPC 类问题。1997 年美国的探测器再次到达火星。火星及  
太空搜索引起了人们的广泛关注，此题便是以此为素材而创作的。

**试题分析** 关于迷宫问题相信每一个参加信息学奥赛的选手都不会陌生。对于不同的迷宫，我们可用搜索策略或动态规划进行求解。在本题中，无论运用哪种解题策略均不能得到问题的最优解，



我们的任务是合理选择一种解题策略，使我们运用该策略得到的较优解尽可能地接近最优解。我们先来看一个例子（如图 7 所示）。对于一个探测车而言，我们运用动态规划的方法使探测车经过岩石最多的一条路线便可得到问题的最优解（如图 8 所示），这时共可收集到岩石 10 个。

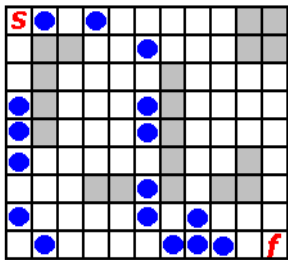


图 7

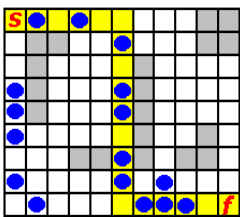


图 8

当有 2 个探测车时，我们让第 2 辆探测车在图 8 的基础上从地图的起点 S 行进至终点 f（如图 9 所示），这时我们共收集到岩石 15 个。而实际上两辆探测车可收集到地图中的全部岩石（共 16 个），

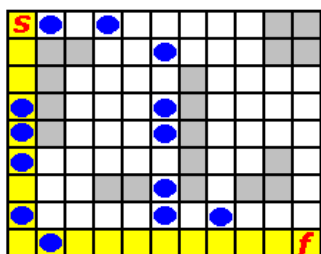


图 9

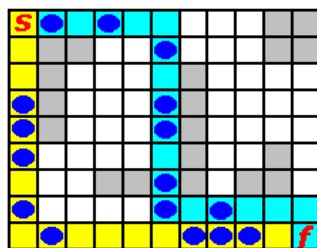


图 10

当探测车数量为 3 时，我们可以收集到全部的 16 个岩石。

我们可让从起点出发的每一辆探测车都收集到尽可能多的岩石，这实际上是一种贪心策略。对于本题而言，贪心策略并不能保证所得结果全部为最优解，但由于每一辆探测车都收集尽可能多的岩石，而对于由计算机随机产生的测试数据而言，岩石是比较均匀地分布在地图中的，于是我们认为：

## 探测车收集岩石数 $\approx$ 探测车所游历的地图空间

让每一辆探测车收集尽可能多的岩石，也就是让探测车经过尽可能大的地图空间。所以在探测车数量逐渐增多时，所有探测车所经过的地图空间越多，收集到的岩石也就越多，此时也就越接近最优解。

此题是否存在最优解呢？其实，我们可以用网络流的算法来解决此题。但实践证明，用网络流算法去求解本题所占空间较大，编程复杂度较高且程序调试起来较为困难，因此在实际比赛中，在限定的时间内用贪心策略完成对题目的求借不失为上策。

### § 6.2 关于运用贪心策略求解 NPC 类问题的讨论

正如前面所讲的那样，在南非举行的第九届国际奥林匹克信息学竞赛中首次引入了 NPC 类问题，在杭州举行的 NOI98 中引入了 NOI 发展史上的第一道 NPC 类问题——并行计算。可以说，NPC 类问题正在日益引起人们的兴趣。它要求选手根据题意自己建立适当的模型，使程序的解尽量逼近最优解。目前，信息学竞赛所涉及到的少量 NPC 类问题主要是运用贪心策略或随机化算法去求较优解的。但是对于同一道 NPC 类问题来说，运用不同的贪心选择所求得的最优解是不同的，不同的贪心选择针对不同的测试数据所得解与最优解的逼近程度也是不同的。所以有关 NPC 类问题的众多特性以及哪些问题运用贪心策略求得的较优解逼近于最优解仍是需要我们花费大量精力去研究的。信息学奥林匹克的精华——激励创新也许在求解 NPC 类问题时会得到最大程度的体现。

## 七、 总结

通过对贪心策略的分析，大家可以看出：贪心策略作为一种高效算法，广泛地应用与信息学奥林匹克竞赛中。即使表面上看起来与贪心策略关系甚微的题目，运用贪心算法也可使程序的运行效率大大提高。因此，深刻理解贪心策略的数学模型、特点、理论基础、尤其是运用其基本思想解决具体问题是十分重要的。希望本文能给参赛选手以一定的启发。

### 【附 录】

#### 【参考书目】

1 《实用算法的分析与程序设计》

吴文虎，王健德编著，电子工业出版社，ISBN 7-5053-4402-1

## 2 《计算机算法导引》

卢开澄编著，清华大学出版社，ISBN 7-302-02277-1

## 3、《国际大学生程序设计竞赛试题解析》

王建德，柴晓路编著，复旦大学出版社  
ISBN 7-309-02141-X/T·210

### 【部分试题及源程序】

#### 1、 吉尔的又一个乘车问题：

Input file:jill.in

Jill likes to ride her bicycle, but since the pretty city of Greenhills where she lives has grown, Jill often uses the excellent public bus system for part of her journey. She has a folding bicycle which she carries with her when she uses the bus for the first part of her trip. She follows the bus route until she reaches her destination of she comes to a part of the city she does not like. In the latter event she will board the bus to finish her trip.

Through years of experience, Jill has rated each road on an integer scale of “niceness”. Positive niceness values indicate roads Jill likes; negative values are used for roads she does not like. Jill plans where to leave the bus and start bicycling, as well as where to stop bicycling and re-join the bus, so that the sum of niceness values of the roads she bicycles on is maximized. This means that she will sometimes cycle along a road she does not like, provided that it joins up two other parts of her journey involving roads she likes enough to compensate. It may be that no part of the route is suitable for cycling so that Jill takes the bus for its entire route. Conversely, it may be that the whole route is so nice Jill will not use the bus at all.

Since there are many different bus routes, each with several stops at which Jill could leave or enter the bus, she feels that a computer program could help her identify the best part to cycle for each bus route.

INPUT

The input file contains information on several bus routes. The first line of the file is a single integer  $b$  representing the number of route descriptions in the file. The identifier for each route ( $r$ ) is the sequence number within the data files,  $1 \leq r \leq b$ . Each route description begins with the number of stops on the route : an integer  $s$ ,  $2 \leq s \leq 20,000$  on a line by itself. The number of stops is followed by  $s-1$  lines, each line  $i(1 \leq i \leq s)$  is an integer  $ni$  representing Jill's assessment of the niceness of the road between the two stops  $i$  and  $i+1$ .

### OUTPUT

For each route  $r$  in the input file, your program should identify the beginning bus stop  $i$  and the ending bus stop  $j$  that identify the segment of the route which yields the maximal sum of niceness  $m = ni + ni+1 + \dots + nj-1$ . If more than one segment is maximally nice, choose the one with longest cycle ride (largest  $j-i$ ). To break ties in longest maximal segments, choose the segment that begins with the earliest stop (lowest  $i$ ). For each route  $r$  in the input file, print a line in the form:

The nicest part of route  $r$  is between stops  $i$  AND  $j$ .

However, if the maximal sum is not positive, your program should print:

Route  $r$  has no nice parts.

### INPUT SAMPLE

```

3
3
- 1
6
10
4
5
4
3
4
4
4
```

4  
- 5

4  
2  
3  
4

#### OUTPUT SAMPLE

The nicest part of route 1 is between stops 2 and 3

The nicest part of route 2 is between stops 3 and 9

Route 3 has no nice parts

#### 2、求最长路径问题（NOI93）：

对一个不存在回路的有向图，编程求出途经结点数最多的一条路径。有向图存放在一个文本文件中，第 0 行为一个数字，为该图的结点总数 N，其下还有 N 行，每行有 N 个非 0 即 1 的数字。若第 i 行第 j 列的数字为 1，则表示结点 i 到结点 j 存在由 i 指向 j 的边，否则该数为 0。

#### 3、删数问题的源程序：

输入数据：一个高精度正整数 N，所删除的数字个数 S。

输出数据：去掉的数字的位置和组成的新的正整数。

```
Program Delete_digit;
Var n:string; {n 是由键盘输入的高精度正整数}
    s, a, b, c:byte; {s 是所要删除的数字的个数}
    data:array[1..200] of 0..9; {记录删除的数字所在位置}
begin
  readln(n);
  readln(s);
  for a:=1 to s do
    for b:=1 to length(n) do if n[b]>n[b+1] then {贪心选择}
      begin
        delete(n, b, 1);
```

```
        data[a]:=b+a-1; {记录所删除的数字的位置}
    break;
end;
while n[1]='0' do delete(n,1,1); {将字符串首的若干个“0”
去掉}
writeln(n);
for a:=1 to s do writeln(data[a],' ');
end.
```

#### 4、最优乘车问题

输入数据：输入文件 INPUT.TXT。文件的第行是一个数字  $M$  ( $1 \leq M \leq 100$ ) 表示开通了  $M$  条单向巴士线路，第 2 行是一个数字  $N$  ( $1 < N \leq 500$ )，表示共有  $N$  个车站。从第 3 行到第  $M+2$  行依次给出了第一条到第  $M$  条巴士线路的信息。其中第  $i+2$  行给出的是第  $i$  条巴士线路的信息，从左至右依次按行行驶顺序给出了该线路上的所有站点，相邻两个站号之间用一个空格隔开。

输出数据：输出文件是 OUTPUT.TXT。文件只有一行，为最少换车次数（在 0, 1, ...,  $M-1$  中取值），0 表示不需换车即可达到。

如果无法乘车达到 S 公园，则输出 “NO”。

```
Program Travel;
var m:1..100;      {m 为开通的单向巴士线路数}
    n:1..500;      {n 为车站总数}
    result:array[1..501] of -1..100; {到某车站的最少换车数}
    num:array[1..500,1..50] of 1..500; {从某车站可达的所有车站
    序列}
    sum:array[1..500] of 0..50;      {从某车站可达的车站总数}
    check:array[1..500] of Boolean; {某车站是否已扩展完}
Procedure Init;
var fl:text;
```

```
a, b, c, d:byte;
data:array[1..100] of 0..100;
begin
  assign(f1, 'input.txt');
  reset(f1);
  readln(f1, m);
  readln(f1, n);
  result[501]:=100;
  for a:=1 to m do
  begin
    for b:=1 to 100 do data[b]:=0;
    b:=0;
    repeat
      inc(b);
      read(f1, data[b]);
    until eoln(f1);
    for c:=1 to b-1 do
      for d:=c+1 to b do
      begin
        inc(sum[data[c]]);
        num[data[c], sum[data[c]]]:=data[d];
      end;
    end;
  end;
end;
Procedure Done;
var min, a, b, c, total:integer;
begin
  fillchar(result, sizeof(result), -1);
  result[1]:=0;
  for c:=1 to sum[1] do result[num[1, c]]:=0;
  b:=data[1, 1];
  repeat
    for c:=1 to sum[b] do
      if (result[num[b, c]]=-1) then result[num[b, c]]:=result[b]
        +1;
    min:=501;
    for c:=1 to n do if (result[c]<>-1) and
```



```
(result[c]<result[min])
    then min:=c;
    b:=min;
until result[n]<>-1;
writeln(result[n]);{到达 S 公园的最少换车次数}
end;
begin
    Init;
end.
```

## 5、最佳游览路线问题

输入数据：输入文件是 INPUT.TXT。文件的第一行是两个整数 M 和 N，之间用一个空格符隔开，M 表示有多少条旅游街 ( $1 \leq M \leq 100$ )，N 表示有多少条林荫道 ( $1 \leq N \leq 20000$ )。接下里的 M 行依次给出了由北向南每条旅游街的分值信息。每行有 N-1 个整数，依次表示了自西向东旅游街每一小段的分值。同一行相邻两个数之间用一个空格隔开。

输出文件：输出文件是 OUTPUT.TXT。文件只有一行，是一个整数，表示你的程序找到的最佳浏览路线的总分值。

```
Program Tour;
var m,n:integer; {M 为旅游街数, N 为林荫道数}
    data:array[1..20000] of -100..100; {data 是由相邻两条林荫道所
    分}
procedure Init; {隔的旅游街的最大分值}
var a,b,c:integer;
    fl:text;
begin
    assign(fl,'input.txt');
    reset(fl);
    read(fl,m,n);
    for a:=1 to n-1 do read(fl,data[a]); {读取每一段旅游街的分值}
```

```
for a:=2 to m do
  for b:=1 to n-1 do
    begin
      read(f1, c);
      if c>data[b] then data[b]:=c;    {读取每一段旅游街的分值,
        并选择}
      end;          {到目前位置所在列的最大分值记入数组 data}
    close(f1);
  end;
procedure Done;
var a, sum, result, c:integer;
    f2:text;
begin
  result:=0;
  sum:=0;
  a:=0;
  while (a<n) do
    begin
      inc(a);          {从数组的第一个数据开始累加, 将累
        加所}
      sum:=sum+data[a];    {得到的最大分值记入 result}
      if sum>result then result:=sum;
      if sum<0 then sum:=0;    {若当前累加值为负数, 则从当前状态起
        从新}
    end;          {累加}
    assign(f2, 'output.txt');
    rewrite(f2);
    writeln(f2, result);
    close(f2);
  end;
begin
  Init;
  Done;
end.
```