数据结构的选择与算法效率
——从 IOI98 试题 PICTURE 谈起

福建师大附中 陈宏

【关键字】
数据结构的选择 线性结构 树形结构

【摘要】
算法 + 数据结构=程序。设计算法与选择合适的数据结构是程序设计中相辅相成的两方面，缺一不可。数据结构的选择一直是程序设计中的重点、难点，正确地应用数据结构，往往能带来意想不到的效果。反之，如果忽视了数据结构的重要性，对某些问题有时就得不到满意的解答。通过对 IOI98 试题 Picture 的深入讨论，我们可以看到两种不同的数据结构在解题中的应用，以及由此得到的不同的算法效率。本文以 Picture 问题为例，探讨数据结构的选择对算法效率的影响。

【正文】

引言

算法通常是决定程序效率的关键，但一切算法最终都要在相应的数据结构上实现，许多算法的精髓就是在于选择了合适的数据结构作为基础。在程序设计中，不但要注重算法设计，也要正确地选择数据结构，这样往往能够事半功倍。

在算法时间与空间效率的两方面，着重分析时间效率，即算法的时间复杂度，因为我们总是希望程序在较短的时间内给出我们所希望的输出。如果在空间上过于“吝啬”而使得时间上无法承受，对解题并无益处。

本文对 IOI98 的试题 Picture 作一些分析，通过两种不同数据结构的选择，将了解到数据结构对算法本身及算法效率的影响。

Picture 问题及算法设计

一、 Picture 问题

Picture 问题是 IOI98 的一道试题，描述如下：

墙上贴着一些海报、照片等矩形，所有的边都为垂直或水平。每个矩形可以被其它矩形部分或完全遮盖，所有矩形合并成区域的边界周长称为轮廓周长。

例如图 1 的三个矩形轮廓周长为 30：

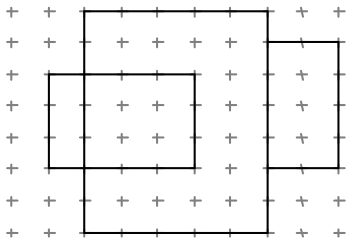


图 1

要求编写程序计算轮廓周长。

数据量限制：

0≤矩形数目<5000；

坐标数值为整数，范围是[-10000，10000]。

二、 算法描述

在算法的大体描述中，将不涉及到具体的数据结构，便于数据结构的进一步选择和比较分析。

（一）、轮廓的定义

在描述算法前，我们先明确一下“轮廓”的定义：

1、轮廓由有限条线段组成，线段是矩形边或者矩形边的一部分。

2、组成矩形边的线段不应被任何矩形遮盖。图 2 与图 3 分别是遮盖的两种情况。

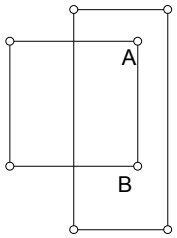


图 2
（AB 被遮盖）

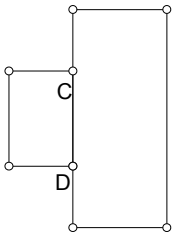


图 3
（CD 被遮盖）

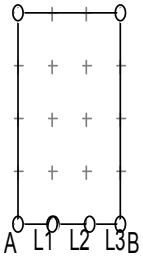


图 4

（二）、元线段

本题的一大特征是分析矩形的边，而边的端点（即矩形的顶点）坐标为整数，且坐标取值范围已经限定在[-10000，10000]之间。这样，就可以把这个平面理解成为一个网格。由于给出的坐标是整数，所以矩形边一定在网格线上。在网格中，对于一条线段我们最关心其绝对坐标。如图 4，我们认为矩形边 AB 由线段 L_1、L_2、L_3 组成。像 L_1、L_2、L_3 这样连接相邻网格顶点的基本线段，称之为“元线段”，这样就把矩形边离散化了。显然，有限的元线段覆盖了所有的网格线，且元线段是组成矩形边乃至组成轮廓的基本单位。一条元线段要么完全属于轮廓，要么完全不属于轮廓。这种定义使我们对问题的研究具体到每一条元线段，这样的离散化处理有利于问题的进一步讨论。

（三）、超元线段

元线段的引入，使问题更加具体。但也应当看到，平面中共有 20001*20000*2 条元线段，研究的对象过多，而且计算量受到网格大小的影响，

如果顶点坐标范围是[-1,000,000，1,000,000]，元线段数目将达到 8*10^12，这是天文数字。因此有必要对“元线段”进行优化。受到元线段的启发，我们定义一种改进后的元线段——“超元线段”，它将由对平面的“切割”得到。具体做法是，根据每个矩形纵向边的横坐标纵向地对平面进行 2*N 次切割、根据矩形横向边的纵坐标横向地对矩形进行 2*N 次切割（N 为矩形个数）。显然，经过切割后的平面被分成了(2*N+1)^2 个区域，如图 5 所示：

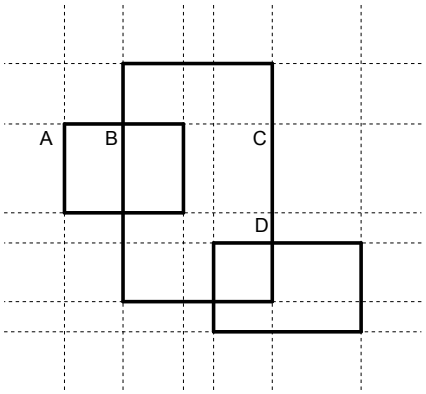


图 5

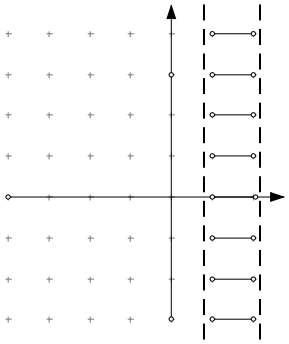


图 6

其中像横向边 AB、纵向边 CD 这样的线段就是“超元线段”。超元线段与元线段有着相似的性质，也是组成轮廓的基本单位。所不同的是，超元线段的数目较少，一般为 4*N 条左右，且超元线段数目不受网格大小的影响。

基于超元线段的优点，算法最终将研究超元线段。

(一)、 离散化及算法框架

算法的研究对象是超元线段，但这并不等于逐一枚举，那样耗时过大，而整体考虑又使得问题无从下手。有一种考虑方法是折中的，即既不研究每一条超元线段，也不同时研究所有的超元线段，而是再进一步优化问题的离散化，即将超元线段分组研究。如图 6 所示，夹在两条纵向分割边的超元线段自然地分为一组，它们的共同点是长度相同，并且端点的横坐标相同。纵向线段也可以进行类似的离散化。

这样的离散化处理后，使得问题规模降低，以此为基础，算法的框架可以基本确定为：

1、对平面进行分割。

2、累加器 ans ← 0。

3、研究每组超元线段，检测其中属于轮廓的部分的长度，并把这一长度累加入 ans。

4、输出 ans 的值。

以上只是算法的基本框架，还很粗糙，求精部分有赖于数据结构的具体选择。

三、 Picture 问题的数据结构选择之一：线性结构

(一)、 映射结构的建立

算法的基础是问题的离散化，要进行平面“分割”，一般需要记录分割点，通常采用映射来记录分割点。直观的做法是采用一维数组形式，下标表示分割点的编号，数组元素表示分割点的坐标。利用下标与数组元素的自然对应，实现映射应该说，这样表示是比较自然的，实现也比较方便。数组的优点主要是存取方便且可以在 O(NlogN)时间内排序。映射结构定义如下：

```
Type
  Mapped_TYPE = Object
    Len    : 0..Max;                {记下分割点的个数}
    Coord  : array[1..Max] of integer;      {记下分割点坐标}
    Procedure  Creat;                {映射初始化}
    Procedure  Insert(X : integer);        {插入分割坐标 X}
    Procedure  Sort;                 {对坐标排序}
  End
```

以下是三个过程的描述与解释：

```
Procedure  Mapped_TYPE.Creat
1   Len ← 0
{Creat 用于初始化该映射}

Procedure  Mapped_TYPE.Insert(X : Integer)
1   Len  ←  Len + 1
2   Coord[Len]  ←  X
{Insert 用于插入一个分割坐标，此时坐标之间是无序的}

Procedure  Mapped_TYPE.Sort
略
{Sort 用于将 Len 个坐标排序。由于 Coord 是一维数组，Sort 容易实现，例如
快速排序。设 N = Len，Sort 效率可达 O(NlogN)。针对整数，也可以采用筒
排序得到更好的效率，但这不是问题的关键部分。}

Var
  X_map,  Y_map  :  Mapped_TYPE      {分别记录横纵坐标的映射}
```

以横坐标为例，在程序处理时，首先执行 X_map.Creat 初始化映射。而后通过 X_map.Insert 将每个矩形纵向边的横坐标作为分割坐标插入 X_map.Coord，最后执行 X_map.Sort 进行排序。

至此，映射建立完毕。

应该说，这一部分完全可以满足算法要求，且执行效率较高。三个过程中的 Creat 与 Insert 耗时均为 O(1)，Sort 耗时为 O(NlogN)，但它只需执行一次。

(二)、 线性结构的建立

映射建立后，相当于完成了对平面的切割。现在的主要问题是如何描述一组超元线段的状态。由于最终要计算轮廓周长，我们最关心的是一组超元线段中究竟有多少条属于轮廓。由分组的方法可知，每组超元线段长度相同。以下均以横向超元线段为例进行说明。设：

超元线段组编号 1——N*2-1(N 是矩形数目)

编号为 S 的超元线段组中的线段长度为 Length(S)

编号为 S 的超元线段组中属于图形轮廓的超元线段数目为 Belong(S)

则：

$$\text{轮廓横向边周长 ANS} = \sum_{S=1}^{N*2-1} Length(s) * Belong(s) \qquad \text{算式①}$$

其中 Lenth(s)容易求得。如果超元线段组编号以网格中从左到右为原则，那么 Length(s)就可以表示为： X_Map.coord[s] – X_map.Coord[s-1]，算式①只需求得 Belong（s）即可。

如图 6，可以看到在问题的离散化之后，一组横向超元线段从上到下，数目是有限的，约有 N*2 条。这使得我们很容易想到线性结构。例如用一维数组来描述这么一组线段，用数组下标表示该线段从上到下的编号。数组雏形定义如下

```
Var
  A : array[1..MaxSize] of integer
```

基于对一维数组的使用，可以得到一个称为“累计扫描”的过程，来求解 Belong(s)。累计扫描的思想是，将一维数组的元素看作计数器，计数器 A[I]的内容是覆盖超元线段 I 的矩形上边的数目 — 覆盖超元线段 I 的矩形下边的数目。形象表示如图 7：

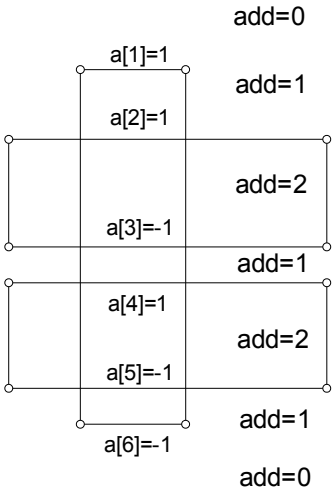


图 7

同时，设立累加器 add，从上至下扫描超元线段，累加 a[I]的值。由图 7 中可以看出，一条超元线段 I 属于轮廓的情况有两种：

1、A[I]≠0 且扫描到该超元线段未累加时 add = 0（超元线段 I 是矩形上边的情况）

2、A[I]≠0 且扫描到该超元线段累加之后 add = 0（超元线段 I 是矩形下边的情况）

这样，对于一组超元线段求解 Belong(s)可以分为两部分：

1、对 A[I]赋值，即累计过程。

2、从上至下扫描一组超元线段并累加 add，即扫描过程。

Belong(s)的值在扫描过程中得到。

至此，描述一组超元线段状态的数据结构基本确立，存储结构是线性一维数组，定义的操作包括累计与扫描两个部分。定义如下：

```
Type
  Group_TYPE = Object
    A : array[1..MaxSize] of Integer;   {线性地记录一组超元线段的信息，如图7}
    Procedure Count;          {累计的过程}
    Function Adding;          {扫描的过程，即求解 Belong(s)的过程}
  End

  Procedure Group_TYPE.Count  {累计的过程}
```

```
1   数组 A 清零
2   for I ← 1 to N
3     do if 矩形 I 跨越了超元线段组 S
                  {即矩形的左右边分别在线段两侧}
4         then A[矩形 I 的上边] ← A[矩形 I 的上边] + 1
5              A[矩形 I 的下边] ← A[矩形 I 的下边] - 1
{所谓"矩形 I 的上边"指矩形 I 上边纵坐标的映射编号，"矩形 I 的下边"同}

Function Group_TYPE.Adding {扫描的过程，函数值即为 Belong(S)的值}
1   调用 Count
2   add ← 0
3   sum ← 0
4   for I ← 1 to 纵坐标的最大映射编号
5     do if a[I] ≠ 0
6         then if add = 0
7               then sum ← sum + 1
                     {该线段是矩形的上边}
8             add ← add + a[I]
9             if add = 0
10              then sum ← sum + 1
                     {该线段是矩形的下边}
11  return sum
{Count 与 Adding 用于一组超元线段的累计扫描}
 Var
   Scan : Group_TYPE
```

数据结构确立后，Belong(s)通过调用 Scan.Adding 来计算，算式①得以实现。

以上的操作针对一维数组而设计，用于进行一组超元线段的累计扫描过程。执行 Scan.Adding 的时间复杂度为 O(N)。横向超元线段分为 2*N-1 组，固求解横向轮廓周长的算法时间复杂度为 O(N^2)。同理，求解纵向轮廓周长的复杂度也为 O(N^2)，则 Picture 问题的算法时间复杂度为 O(N^2)。虽然这是一个多项式阶但在最坏情况下（N 接近 5000 时）还有一定的计算量。

对数据结构选择的进一步分析

累计扫描过程体现了一种认识和思维方式，以一维数组作为数据结构基础，这里是否有更好的做法，我们将作进一步分析。

通过求解问题对数据结构选择作的分析中，我们注意到在选择数据结构需要考虑的几个方面：

1、数据结构要适应问题的状态描述。解决问题时需要对状态进行描述，在程序中，要涉及到状态的存储、转换等。选择的数据结构必需先适用于描述状态，并使对状态的各种操作能够明确地定义在数据结构上。在 Picture 问题中，涉及到算法的状态是关于一组"超元线段"的描述，目的是要确定该组超元线段的数目，我们选择了线性结构，采用计数扫描的方法，统计超元线段属于轮廓的数目。这种表示法直观、易于实现，可以说基本适用于描述状态。但采用一维数组效率并不高，一次扫描耗时较大。其中主要的原因是各组超元线段的扫描分别独立，后面的扫描并不能利用前面的结论。

2、数据结构应与所选择的算法相适应。数据结构是为算法服务的，其选择要充分考虑算法的各种操作，同时数据结构的选择也影响着算法的设计。我们有这样的认识和经历，如果算法是对一个队列进行堆排序，就应当选择能够迅速定

位的数据结构，如一维数组等，而不应选择像链表这样定位耗时的数据结构，反之，如果要对一个链表进行排序，则基于链表结构的基数排序应当是首选对象。Picture 问题的算法思想基于问题的离散化，需要对平面进行分割，记录分割点的坐标。通常，使用映射来记录分割点。采用数组形式，利用其下标与数组元素的自然对应，实现映射，直截了当。这样选择基本可以满足算法要求。

同时，在选择数据结构时，也要考虑其对算法的影响。数据结构对算法的影响主要在两方面：

◆ 数据结构的存储能力。如果数据结构存储能力强、存储信息多，算法将会较好设计。反之对于过于简单的数据结构，可能就要设计一套比较复杂的算法了在这一点上，经常体现时间与空间的矛盾，往往存储能力是与所使用的空间大小成正比的。

◆ 定义在数据结构上的操作。“数据结构”一词之所以不同于“变量”，主要在于数据结构上定义了基本操作，这些操作都有较强的实际意义。这些操作就好比工具，有了好的工具，算法设计也会比较轻松。Picture 问题中选择了线性结构，它定义的操作比较简单，因此无法很好地将不同组的超元线段统计联系起来。

3、数据结构的选择同时要兼顾编程的方便。许多复杂的数据结构能够得到较好的效率，但编程复杂，不易实现且容易出错。在这种情况下，如果能够选择一种我们较为熟悉的又不会过多地降低程序效率的数据结构，倒不失为一种折中的办法。如 Picture 问题中的 Group_TYPE.Count 过程的 4、5 两步，要求出某个矩形边对应的映射编号。我们定义的映射仅仅是编号→坐标值，并不是坐标值→编号。如果再实现这一映射，势必增加编程难度。所以编程求精时，可以认为以整数而不是以顶点坐标对平面进行横向切割。这样映射关系很好建立，坐标值本身就是编号，减少了编程难度。如果进一步以顶点坐标作横向切割，当然会提高程序效率，但效果并不明显——扫描计数仍需要 O(N)的时间，这是很昂贵的，所以进一步切割并不影响算法主要部分的效率，另一方面，编程难度却会大大提高，得不偿失。由此看出，在算法效率“大局已定”的情况下，有时也需要适当地牺牲程序效率来减少编程不必要的麻烦。

4、灵活应用已有知识。我们对编程都积累了一定的经验，对以后的解题有很大帮助。一个“新问题”有时与“旧问题”有许多内在的联系，往往能够将新问题转化为所学过的知识，或者由所学过的知识得到启发，从而解决问题。所谓“新”数据结构的构造，有时可以是几种基本数据结构的有机结合，或者由基本数据结构得到启发而得到。做到“温故而知新”，是对算法设计者创新意识的要求。当然对一个问题，要首先考虑现成的、经典的数据结构。如队列、栈、链表等等，其标准结构与标准运算已经有了“公论”，程序实现也经过了“千锤百炼”，效率已经很完美。如果找到一种可行的经典数据结构，那么算法实现一般来说就比较轻松。要做到这一点，要求我们有扎实的基础知识，对各种算法及数据结构了然于胸。在计数扫描过程中采用了经典的线性一维数组，是一个很自然的考虑方向，并且可以很容易上机实现，不足之处在于其效率较低。

总地来说，Picture 问题算法思想的方向还是基本正确的。Picture 问题最大数据应包含近 5000 个矩形，这样大的数据量决定了要降低规模是“大势所趋”，所以对问题的离散化处理是合理的选择。至于效率不高，应当是线性数据结构的选择造成的。由此，我们可以看到使用线性结构来实现 Picture 问题还有一些缺陷，其中最主要的是各组超元线段的统计相互独立，联系不紧，这是算法效率不高的“瓶颈”。为了解决这个问题，我们尝试用其它的数据结构来实现算法，像前面一样，这个数据结构应该符合以下的条件：

1、同线性结构一样，新数据结构要适用于描述一组超元线段的状态。至少，新结构要合理地表示一组超元线段属于轮廓的部分，或者说它要能准确地且较

快地计算出算式①中 Belong(s)的值。

2、新结构也要与基本算法相适应。新结构仍然以问题的离散化为基础，映射结构应当保留——事实上映射结构在时间效率上并没有缺点。新结构在描述超元线段组时则要设法将不同组超元线段的统计有机结合起来。

3、新结构还要兼顾编程的方便。如果选择的数据结构编程难度太大，以至于无法上机实现，或者只是理论上的"高效率"，对解题没有实际意义。这么说也许太夸张，但实际上也常常存在"鱼与熊掌不可兼得"的情况。大部分高级数据结构的实现都需要一定的编程技巧。就像问题在时间效率与空间效率上的矛盾一样，算法效率与编程难度也有矛盾，一般来说算法效率越高，编程难度也会越大。考虑新结构时希望能找到实现较为容易的数据结构。

综合以上分析，由于线性结构并不能给我们带来令人满意的效率，所以我们尝试用树形结构来描述一组超元线段的状态，实现 Picture 问题的基本算法。为了提高效率，采用的树结构必需是平衡树，我们姑且称之为"超元线段树"。

Picture 问题的深入讨论

基于对数据结构选择的进一步分析，我们来重新考虑一下 Picture 问题的数据结构的选择，即采用树形结构来描述一组超元线段的状态。

一、 线段树

受到累计扫描过程的启发，一组超元线段属于轮廓的数目，它与跨越该组超元线段的矩形的纵向边位置关系密切。不妨把矩形的纵向边投影到 Y 轴上，这样就把矩形的纵向边看作闭区间，并称之为闭区间 Q。我们以"线段树"的树形数据结构来描述闭区间 Q。作为工具，先简单研究线段树的特点。

线段树是描述单个或若干区间并的树形结构，属于平衡树的一种。使用线段树要求知道所描述的区间端点可能取到的值。换句话说，设 A[1..N]是从小到大排列的区间端点集合，对于任意一个待描述的闭区间 P=[x,y]，存在 1≤i≤j≤N 使得 x=a[i]且 y=a[j]，这里 i, j 称为 x,y 的编号。可以看到，即使是实数坐标，在线段树中也只有整数含义。以下所说的区间[x, y]如无特殊说明，x、y 均是整数，即原始区间顶点坐标的编号。

线段树是一棵二叉树，将数轴划分成一系列的初等区间[I, I+1] (I=1—N-1)。每个初等区间对应于线段树的一个叶结点。线段树的内部结点对应于形如[I, J] (J – I > 1)的一般区间。一般情况下，线段树的结点类型定义如下：

```
Type
  Lines_Tree = Object
    i, j   : integer;      {结点表示的区间的顶点标号 I, J}
    count  : integer;      {覆盖这一结点的区间数}
    leftchild, rightchild : ↑Lines_Tree;   {二叉树的两个子结点}
  end
```

关于 Lines_Tree 的其它数据域与定义的运算将陆续添加。图 8 是一棵线段树，描述的区间端点可以有 10 种取值。其中记录着一个区间[3，6]，它用红色的[3，5]及[5，6]的并采表示。图中红色结点的 count 域值为 1，黑色结点的 count 域值为 0。

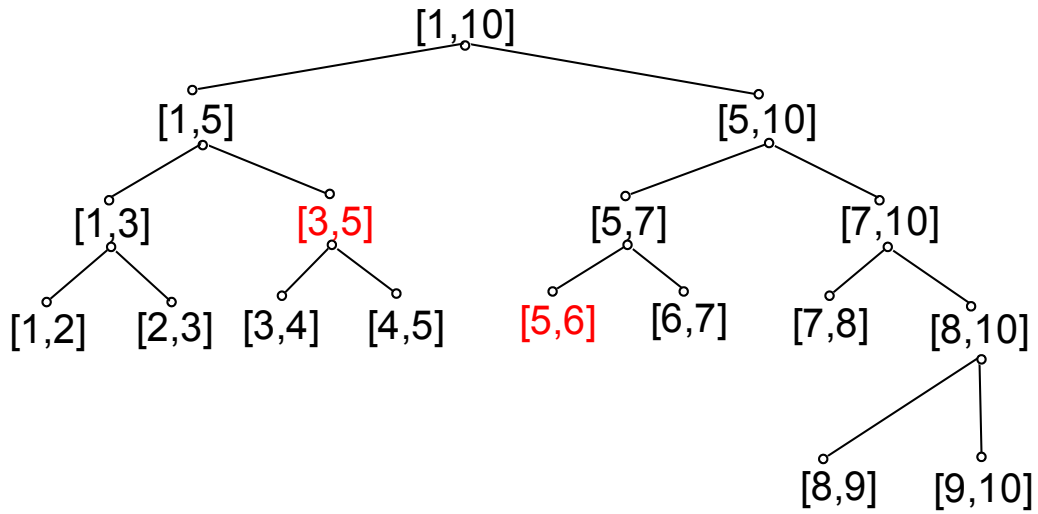


图 8

直观地看，子结点就是父结点区间平均分成两部分。设 L, R 是父结点的区间端点，我们可以增加 Lines_Tree.Build(l, r : integer)递归地定义线段树如下:

```
Procedure  Lines_tree.Build(l, r : integer)
1   I  ←  l          {左端点}
2   J  ←  r          {右端点}
3   Count  ←  0      {初始化}
4   If  r - l  >  1     {是否需要生成子结点，若 r-l=1 则是初等区间}
5     then   k  ←  (l  +  r)     {平均分为两部分}
6          new(leftchild)
7          leftchild↑.Build(l, k)   {建立左子树}
8          new(rightchild)
9          rightchild↑.Build(k, r)   {建立右子树}
10   else   leftchild  ←  nil
11        rightchild  ←  nil
```

设根结点是 Root，建树需要执行 Root.Build。

由递归定义看出，线段树是一棵平衡树，高度为 ⌈logN⌉。建立整棵树需要的时间为 O(N)。

以上着重说明了线段树的存储原理，我们还应建立线段树的基本运算。

线段树可以存储多个区间，所以支持区间插入运算 Lines_Tree.Insert(l, r : integer)，定义如下：

```
Procedure  Lines_Tree.Insert(l, r  :  integer)
{[l,  r]是待插入区间，l、r 都是原始顶点坐标}
1   if  (l  <=  a[i])  and   (a[j]  <=  r)
2     then  count  ←  count  +  1          {盖满整个结点区间}
3     else  if  l < a[(i  +  j)  div  2]  {是否能覆盖到左孩子结点区间}
4           then  leftchild↑.Insert(l, r)     {向左孩子插入}
5         if  r > a[(i  +  j) div  2 ]  {是否能覆盖到右孩子结点区间}
6           then  rightchild↑.Insert(l,  r)    {向右孩子插入}
```

类似地，线段树支持区间的删除 Lines_Tree.Delete(l, r : integer)，定义如下：

```
Procedure  Lines_Tree.Delete(l, r  :  integer)
{[l,  r]是待删除区间，l、r 都是原始顶点坐标}
```

```
1   if (l <= a[i]) and  (a[j] <= r)
2     then count ← count - 1          {盖满整个结点区间}
3     else if l < a[(i + j) div 2 ] {是否能覆盖到左孩子结点区间}
4           then leftchild↑.Delete(l, r)     {向左孩子删除}
5         if r > a[(i + j) div 2 ] {是否能覆盖到右孩子结点区间}
6           then rightchild↑.Delete(l, r)    {向右孩子删除}
```

执行 Lines_Tree.Delete(l, r : integer) 的先决条件是区间[l, r]曾被插入且还未删除。如果建树后插入区间[2，5]而删除区间[3，4]是非法的。

通过分析插入与删除的路径，可知 Lines_Tree.Insert 与 Lines_Tree.Delete 的时间复杂度均为 O(logN)。(详见[附录 1])

由于线段树给每一个区间都分配了结点，利用线段树可以求区间并后的测度与区间并后的连续段数。

(一)、 测度

由于线段树结构递归定义，其测度也可以递归定义。增加数据域 Lines_Tree.M 表示以该结点为根的子树的测度。M 取值如下：

$$M = \begin{cases} a[j] - a[i] & \text{该结点 Count>0} \\ 0 & \text{该结点为叶结点且 Count=0} \\ Leftchild\uparrow.M + Rightchild\uparrow.M & \text{该结点为内部结点且 Count=0} \end{cases}$$

据此，可以用 Lines_Tree.UpData 来动态地维护 Lines_Tree.M。UpData 在每一次执行 Insert 或 Delete 之后执行。定义如下：

```
Procedure  Lines_Tree.UpData
1   if count > 0
2     then M ← a[j] – [i]     {盖满区间，测度为 a[j] – a[i]}
3     else if j - i = 1       {是否叶结点}
4           then M ← 0        {该结点是叶结点}
5           else M ← Leftchild↑.M + Rightchild↑.M
                                {内部结点}
```

UpData 的复杂度为 O(1)，则用 UpData 来动态维护测度后执行根结点的 Insert 与 Delete 的复杂度仍为 O(logN)。

(二)、 连续段数

这里的连续段数指的是区间的并可以分解为多少个独立的区间。如[1，2]∪[2，3]∪[5，6]可以分解为两个区间[1，3]与[5，6]，则连续段数为 2。增加一个数据域 Lines_Tree.line 表示该结点的连续段数。Line 的讨论比较复杂，内部结点不能简单地将左右孩子的 Line 相加。所以再增加 Lines_Tree.lbd 与 Lines_Tree.rbd 域。定义如下：

$$lbd = \begin{cases} 1 & \text{左端点 I 被描述区间盖到} \\ 0 & \text{左端点 I 不被描述区间盖到} \end{cases}$$

$$rbd = \begin{cases} 1 & \text{右端点 J 被描述区间盖到} \\ 0 & \text{右端点 J 不被描述区间盖到} \end{cases}$$

lbd 与 rbd 的实现：

1 该结点 count > 0
lbd = 0 该结点是叶结点且 count = 0
leftchild↑.lbd 该结点是内部结点且 Count=0
1 该结点 count > 0
rbd = 0 该结点是叶结点且 count = 0
rightchild↑.rbd 该结点是内部结点且 Count=0

有了 lbd 与 rbd，Line 域就可以定义了：

1 该结点 count > 0
Line = 0 该结点是叶结点且 count = 0
Leftchild↑.Line + Rightchild↑.Line - 1
当该结点是内部结点且 Count=0，Leftchild↑.rbd=1 且 Rightchild↑.lbd=1
Leftchild↑.Line + Rightchild↑.Line
当该结点是内部结点且 Count=0，Leftchild↑.rbd 与 Rightchild↑.lbd 不都为 1

据此，可以定义 UpData'动态地维护 Line 域。与 UpData 相似，UpData'也在每一次执行 Insert 或 Delete 后执行。定义如下：

```
Procedure  Lines_Tree.UpData'
1   if  count  >  0          {是否盖满结点表示的区间}
2    then  lbd  ←  1
3         rbd  ←  1
4         Line  ←  1
5    else  if  j  -  i  =  1     {是否为叶结点}
6          then  lbd  ←  0   {进行到这一步，如果为叶结点，
                                    count = 0}
7               rbd  ←  0
8               line  ←  0
9          else  line  ←  Leftchild↑.line  +  Rightchild↑.line  -
                    Leftchild↑.rbd * Rightchild↑.lbd
                    {用乘法确定 Leftchild↑.rbd 与 Rightchild↑.lbd 是否同时为 1}
```

同时，由于增加了 Line、M 等几个数据域，在建树 Lines_Tree.Build 时要将新增的域初始化。

至此，线段树构造完毕，完整的线段树定义如下：

```
Lines_Tree = object
  i, j    :  integer;
  count   :  integer;
  line    :  integer;
  lbd, rbd  :  byte;
  m       :  integer;
  leftchild,
  rightchild  :  ↑Lines_tree;
  procedure  Build(l, r : integer);
  procedure  Insert(l, r : integer);
  procedure  Delete(l, r : integer);
  procedure  UpData;
  procedure  UpData';
end
```

有了线段树这个工具，可以考虑利用树形结构来描述一组超元线段的状态。

二、 Picture 问题的数据结构选择之二：树形结构

采用线性结构描述一组超元线段的状态并不能带来太高的效率，其中主要原因是各组超元线段联系不紧。如图 9 所示，超元线段 CD 与 EF 被矩形 AGHB 遮盖，不属于轮廓；而与之相邻 DD’与 FF’则“摆脱”了矩形的遮盖，属于轮廓的一部分。

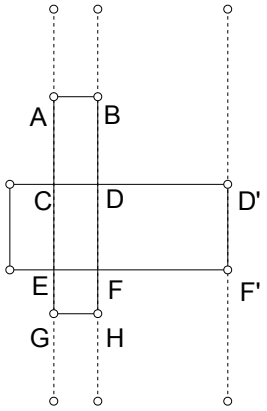


图 9

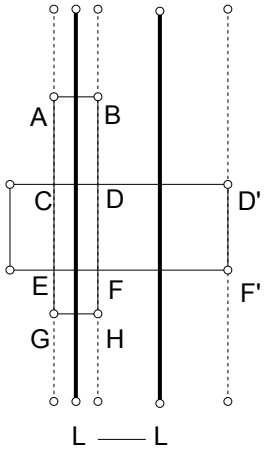


图 10

由此类推，可以看出相邻的超元线段组都有类似的问题。如图 9，DD’与 FF’不被遮盖，可以这样分析：从左往右，CD、EF 首先被遮盖，但随着 BF 的出现，对 DD’、FF’的遮盖自然消失。这一点，正是相邻超元线段组的内在联系。用线性结构无法表示出这一联系，因为各组的累计扫描过程是独立的。现在我们用树形结构来表示将较好地解决这一问题，因为线段树支持插入与删除及动态维护，可以有机联系各组超元线段的状态。我们把“从左往右”当作一个扫描的过程，若将其严格地描述，可以得到一个称为线段扫描的过程：

1. 设立扫描线段 L。
2. L 从左往右扫描，停留在每一超元线段组上。如图 10 所示。
3. L 的状态用线段树来表示，每一条纵向的矩形边看作一个待合并区间。线段树的连续段数*2 表示该组超元线段属于轮廓的线段数目。如图 10，L 的状态首先是[G,A]∪[E,C]，连续线段数是 1，所以 1*2=2 是该组超元线段属于轮廓的数目。接着 L 进一步扫描，状态改变为[E,C], 连续线段数是 1，所以该组超元线段属于轮廓的数目也是 1*2=2。这样，上文所说的“超元线段树”就用线段树来实现。为了统一起见，以后仍称线段树。
4. 扫描过程中动态地维护 L 的状态。参看图 10，L 状态的转换是在线段树中删去区间[H,B]即[G,A]造成的。归纳一下，有以下结论：

◆ L 初始化为空，即线段树刚建好时的情形。
◆ 扫描时，遇到矩形左边，将其插入（Insert）线段树。
◆ 扫描时，遇到矩形右边，将其从线段树中删除（Delete）。由于从左往右扫描，事先插入了该矩形的左边，所以删除合法。

参看算式①，以上的线段扫描过程可以得到每一组超元线段的 Belong(s)，进一步得到整个图形轮廓的横向边长。同时，线段扫描过程还可以在一次从左到右的扫描中求得图形轮廓的纵向边长。还以图 10 为例。在扫描线状态改变之前，L 是[G,A]∪[E,C]；改变状态之后，[H,F]、[D,B]就“露”了出来，成为轮廓一部分。[G,A]∪[E,C]正是 L 改变前后测度的差。如果描述相邻的扫描线状态的线段树分别为 Tree1、Tree2，则扫描过程中“露出”的纵向边长度为|Tree1.M – Tree2.M|。

在扫描过程中，遇到的插入或删除称为“事件”，待插入或删除的线段称为“事件线段”。在扫描之前，应将事件按横坐标从小到大排序。（详见[附录 2]）

通过以上分析，得到较之线性结构的累计扫描过程改进的线段扫描过程的算法：

1. 以矩形顶点坐标切割平面，实现横纵坐标的离散化并建立映射 X_Map、Y_Map。
2. 事件排序

```
3.  Root.Build(1, N*2)
4.  Nowm  ← 0
5.  NowLine ← 0
6.  Ans   ← 0
7.  for  I ← 1  to  事件的最大编号
8.    do  if I是插入事件
9.          then  Root.Insert(Y_Map.Coord[事件线段顶点 1]，
                        Y_Map.Coord[事件线段顶点 2])
10.         else  Root.Delete(Y_Map.Coord[事件线段顶点 1]，
                        Y_Map.Coord[事件线段顶点 2])
11.       nowM  ← Root.M
12.       nowLine ← Root.Line
13.       ans  ← ans  +  lastLine * 2 * (X_Map[I] – Y_Map[I-1])
14.       ans   ← ans  +  |nowM – lastM|
15.       lasM   ← nowM
16.       lastLine  ← nowLine
```

排序的时间复杂度为 O(NlogN)。事件的最大编号为 N*2，插入或删除的复杂度为 O(logN)，所以整个过程效率为 O(NlogN)。至此，以树形数据结构为基础的算法模式确立，时间效率是令人较为满意的。

三、 两种实现方法的比较

两种数据结构的不同之处不仅在于它们本身的存储差异、定义的运算差异，还在于它们对算法时间复杂度产生的影响。线性结构产生的复杂度为 O(N^2)，而树形结构产生的复杂度为 O(NlogN)。以下是一组数据，将两种数据结构实现程序的运行时间作一个对比，可以感性地认识它们之间的效率差异：

<table>
<tr><th>数据</th><th>实现一：线性结构</th><th>实现二：树形结构</th></tr>
<tr><td>Data_4_1.Txt</td><td rowspan="2">1 秒以内</td><td rowspan="5">1 秒以内</td></tr>
<tr><td>Data_4_2.Txt</td></tr>
<tr><td>Data_4_3.Txt</td><td rowspan="3">30 秒左右</td></tr>
<tr><td>Data_4_4.Txt</td></tr>
<tr><td>Data_4_5.Txt</td></tr>
</table>

注：以上程序在赛扬 300/Borland Pascal 下运行。（程序清单见[程序 1]—线性结构及[程序 2]—树形结构）

四、 Picture 问题的推广

基于对 Picture 问题特征的分析及树形结构的应用，可以使问题得到推广。

离散化思想可以使顶点坐标由整数→实数。在算法及数据结构的选择中，我们已不再使用数据类型为整数的特性。所以 Picture 问题的数据类型可以推广到实型。为了适应这一变化，要将 Mapped.Coord 的基类型改为实型，同时将 Lines_Tree.M 改为实型，线段扫描算法中的累加器 ans 等涉及到顶点坐标的类型也要改为实型。

更重要的是，Picture 问题本身也可以推广，即由 Picture 周长问题→Picture 面积问题。周长问题涉及到扫描线 L 的连续段数与测度，其中横向轮廓涉及到连续段数，纵向轮廓涉及到测度；相应地，面积问题涉及到 L 的测度。在扫描过程中，设事件点为 I，L“扫过”的一组超元线段的面积就是

测度*（X_Map.coord[I] – X_Map.coord[I-1]）。

（程序清单见[程序 3]——Picture 面积问题）

结论

Picture 问题在基于离散化思想的算法下，通过改进数据结构的选择，即由线性结构→树形结构，使得时间复杂度大大降低。改进源于数据结构的选择，更本质地，源于对问题本身与数据结构特点的较为深刻的认识。只有进一步理解问题、进一步认识问题，才能更好地选择适用于问题的数据结构；也只有对数据结构的特性充分理解，才能在各种结构中作出合适的选择。

通过上述讨论，我们进一步认识了数据结构选择的重要性，对选择数据结构的技巧也有了一定认识。其中最重要的一点，就是数据结构要适用于问题、适用于算法，只有这样，才能较为本质地描述状态、解决问题。

从某种意义上说，数据结构与算法是空间与时间的具体体现。在这个意义上它们有统一的时候，也有矛盾的时候。以 Picture 为例，线性结构可以不占用堆空间，但时间复杂度是 O(N^2)，树形结构要大量占用堆空间，却将复杂度降低到 O(NlogN)。如何协调它们的矛盾，引导其走向统一是算法设计中的一个重要环节。

【参考书目】

《算法与数据结构》　　电子工业出版社　编著：傅清祥　王晓东
《算法 + 数据结构 = 程序》　科学出版社　瑞士　N·沃思　著
　　曹德和　刘椿年　译

把握本质，灵活运用——动态规划的深入探讨

浙江省萧山中学 来煜坤

【关键字】动态规划 构思 实现

【摘要】本文讨论了动态规划这一思想的核心内容和其基本特点探讨了动态规划思想的适用范围，动态规划子问题空间和递推关系式确立的一般思路。通过例子说明在子问题确立过程中的一些问题的解决办法：通过加强命题或适当调节确定状态的变量等手段帮助建立动态规划方程，通过预处理使动态规划的过程容易实现等。接着，分析动态规划实现中可能出现的空间溢出问题及一些解决办法。总结指出，动态规划这一思想，关键还在于对不同的问题建立有效的数学模型，在把握本质的基础上灵活运用。

一、引言

动态规划是一种重要的程序设计思想，具有广泛的应用价值。使用动态规划思想来设计算法，对于不少问题往往具有高时效，因而，对于能够使用动态规划思想来解决的问题，使用动态规划是比较明智的选择。

能够用动态规划解决的问题，往往是最优化问题，且问题的最优解(或特定解)的局部往往是局部问题在相应条件下的最优解，而且问题的最优解与其子问题的最优解要有一定的关联，要能建立递推关系。如果这种关系难以建立，即问题的特定解不仅依赖于子问题的特定解，而且与子问题的一般解相关，那么，一方面难以记录下那么多的“一般解”，另一方面，递推的效率也将是很低的；此外，为了体现动态规划的高时效，子问题应当是互相重叠的，即很多不同的问题共享相同的子问题。(如果子问题不重叠，则宜使用其它方法，如分治法等。)

动态规划一般可以通过两种手段比较高效地实现，其一是通过自顶向下记

忆化的方法，即通过递归或不递归的手段，将对问题最优解的求解，归结为求其子问题的最优解，并将计算过的结果记录下来，从而实现结果的共享；另一种手段，也就是最主要的手段，通过自底向上的递推的方式，由于这种方式代价要比前一种方式小，因而被普遍采用，下面的讨论均采用这种方式实现。动态规划之所以具有高时效，是因为它在将问题规模不断减小的同时，有效地把解记录下来，从而避免了反复解同一个子问题的现象，因而只要运用得当，较之搜索而言，效率就会有很大的提高。

动态规划的思想，为我们解决与重叠子问题相关的最优化问题提供了一个思考方向：通过迭代考虑子问题，将问题规模减小而最终解决问题。适于用动态规划解决的问题，是十分广泛的。动态规划的思想本身是重要的，但更重要的是面对具体问题的具体分析。要分析问题是否具备使用动态规划的条件，确定使用动态规划解题的子问题空间和递推关系式等，以及在(常规)内存有限的计算机上实现这些算法。下面分别就构思和实现两个方面进一步探讨动态规划这一思想

二、动态规划解题的构思

当我们面对一个问题考虑用动态规划时，十分重要的一点就是判断这个问题能否用动态规划高效地解决。用动态规划构思算法时，往往要考虑到这个问题所涉及到的子问题(子问题空间)，以及如何建立递推式，并最终实现算法。其实这些过程往往是交织在一起的，子问题空间与递推关系本身就是紧密相联的，为了有效地建立起递推关系，有时就要调整子问题空间；而根据大致确定的子问题空间又可以启发我们建立递推关系式。而能否最终用一个递推关系式来联系问题与其子问题又成了判断一个问题能否使用动态规划思想解决的主要依据。因而孤立地来看这其中的每一部分，硬把思考过程人为地分成几个部分，是困难的，也是不必要的。而且动态规划这种思想方法，没有固定的数学模型，要因题而异，因而也就不可能归纳出一种"万能"的方法。但是对大多数问题而言，还是能够有一个基本的思考方向的。

首先，要大致分析一个问题是否可能用动态规划解决。如果一个问题难以确定子问题，或问题与其子问题的特殊解之间毫无关系，就要考虑使用其它方法来解决(如搜索或其它方法等)。做一个大概的判断是有必要的，可以防止在这上面白花时间。通常一个可以有效使用动态规划解决的问题基本上满足以下几方面的特性：

1、 子问题的最优解仅与起点和终点(或有相应代表意义的量)有关而与到达起点、终点的路径无关。
2、 大量子问题是重叠的，否则难以体现动态规划的优越性。

下面以 IOI'97 的"字符识别"问题为例进行分析一般情况下动态规划思路的建立。

IOI'97 的字符识别问题，题目大意是：在 FONT.DAT 中是对□(空格)、A—Z 这 27 个符号的字形说明。对每一个符号的字符点阵，用 20 行每行 20 个"0"或者"1"表示。在另一个输入文件中，描述了一串字符的点阵图象(共 N 行)，但字符

可能是“破损的”，即有些 0 变成了 1，而有些 1 变成了 0。每行固定也为 20 个“0”或“1”，但每一个字符对应的行可能出现如下情形：

- 仍为 20 行，此时没有丢失的行也没有被复制的行；
- 为 19 行，此时有一行被丢失了；
- 为 21 行，此时有一行被复制了，复制两行可能出现不同的破损。

要求输出，在一个假定的行的分割情况下，使得“0”与“1”的反相最少的方案所对应的识别结果(字符串)。

在初步确定这个问题可以用动态规划思想解决之后，我认为可以考虑用数学的方法(或观点)来刻划这个问题，比如通常的最优化问题(这也是动态规划解决的主要问题)，总会有一个最优化的标准，动态规划要通过递推来实现，就要求分析确定这个状态所需要的量。比如字符识别问题，在问题规模下相当于求 N 行的一种分割与对应方法，因而很自然地，考虑前几行就成了一个确定状态的量。最优的标准题中已经给出，即在某种假设(包括分割方法与对应识别方法)下使得“0”与“1”反相数最少。如果把这个度量标准看作一个函数，这实际上就是一个最优化函数(指标函数)，最优化函数的值依赖于自变量，即确定状态的量。自变量的个数(这里是一个，即行数，考虑前几行之意)，要因题而异，关键是要有效地确定状态，在这种状态下，因保证靠这些量已经能够确定最优化函数的值，即最优化函数在这些量确定的时候理论上应有确定的值，否则量是不够的或要使用其它量来刻划，而即使能够完全确定，但在建立递推关系式时发生困难，也要根据困难相应调整确定最优化函数值的自变量。而反过来，如果设定了过多的量来确定最优化函数值，那么，动态规划的效率将会大大下降，或者解了许多不必要解的子问题，或者将重叠子问题变成了在这种自变量条件下的非重叠子问题，从而大大降低效率，甚至完全失去动态规划的高效。在这个例子中对于前 L 行，此最优化函数显然有确定的值。

动态规划的递推的一种重要思想是将复杂的问题分解为其子问题。因而确定子问题空间及建立递推关系式是十分重要的。根据确定最优化函数值的自变量往往对子问题空间有着暗示的作用。通常，通过对最接近问题的这步进行倒推，可以得到这个问题规模减小一些的子问题，不断这样迭代考虑，就往往能够体会到问题的子问题空间。而在这个过程中，通过这种倒推分析，也比较容易得出这种递推关系。需要指出，这仅仅是对一些题目解题思考过程的总结，不同的题目原则上仍应区别对待。比如字符识别问题，考虑 n 行该最优化函数值时，注意到最终一定是按照字符分割与识别的，因而最后一个字符或者是 19 行，或者是 20 行，再或者是 21 行，仅这样三种可能情况，依次考虑这三种分割方法，对于切割下来的这一段对应于一个字符，对于每一种切割方案，当然应该选择最匹配的字符(否则，如果不使用反相情况最少的字符作为匹配结果而导致全局的最优，那么只要在这一步换成反相情况最少的字符，就得到比假定的“最优”更优的结果，从而导致矛盾)。在去除一个字符后，行数有所减少，而这些行去匹配字符显然也应当使用最优的匹配(可以用反证法证明，与前述类似)，于是得到一个与原问题相似(同确定变量，同最优化标准)但规模较小的子问题，与此同时子问题与原问题的递推关系事实上也得到了建立：

f[i]:=min{Compare19[i-19+1]+f[i-19],Compare20[i-20+1]+f[i-20],
Compare21[i-21+1]+f[i-21]}

f[i]表示对前 i 行进行匹配的最优化函数值；

Compare19[i]、Compare20[i]、Compare21[i]分别表示从 i 行开始的 19 行、20 行、21 行与这三种匹配方式下最接近的字符的反相的“0”与“1”的个数。

初始情况，f[0]=0，对于不可能匹配的行数，用一个特殊的大数表示即可。当然，本题的问题主要还不在于动态规划的基本思考上(这里只是通过这个例子，讲一下对于不少动态规划问题的一种基本的思考方向)，还有数学建模(用 2 进制表示 0、1 串)等(源程序见附录中的程序 1)。

有时虽然按上述思路得出的确定状态的量已经能够使最优化函数具有确定的值，但是在建立递推关系时发生困难，通过引入新的变量或调整已有变量，也是一条克服困难的途径。比如，NOI’97 的一题“积木游戏”，题目大意是：

积木是一个长方体，已知 N 个有编号的积木的三边(a、b、c 边)长，要求出用 N 块中的若干块堆成 M(1≤M≤N≤100)堆，使总高度最大的高度值，且满足：

- 第 K 堆中任意一块的编号大于第 K+1 堆中任意一块积木的编号；
- 任意相邻两块，下面的块的上表面要能包含上面的那块的下表面，且下面的块的编号要小于上面积木的编号。

因为题目要求编号小的堆的积木编号较大，这不太自然，在不改变结果的前提下，把题目改作编号小的堆的积木编号较小，这显然不会影响到最终的高度和，而且，此时每一种合理的堆放方法可看作，按编号递增的顺序选择若干积木，按堆编号递增的顺序逐堆放置，每堆中积木依次在前一个上面堆放而最终形成一种堆放方案。使用上面一般的分析方法，很容易看出，考虑前 i 个木块放置成前 j 堆，这样，i、j 两个量显然能够确定最优函数的值，然而递推关系却不易直接建立，稍作分析就会发现，问题主要出在第 i 块到底能否堆放到其子问题(i-1,j 作变量确定的状态)的最优解方案的最后一堆上。如果考虑增加该序列最后一块的顶部的长与宽的(最小)限制这两个变量，建立递推关系并不困难，然而，很明显，递推过程中大量结果并未被用到，这就人为地扩大了子问题空间，不仅给存储带来麻烦，而且效率也很低。其实，建立递推需要的仅仅是在子问题解最后一堆顶部能否容纳当前积木块，而题中可能产生的这种限制性的面最多仅有 3*100+1(无限制)=301 种情况，这样在多引入一个“最后一堆顶部能够容纳下第 k 种面的要求”这个量后，递推关系只要分当前块另起一堆、当前块加在前一堆上(如果可能的话)和当前块不使用这三种情况就可以了。(源程序参见所附程序 2)

此外，有些问题可能会出现仅靠这种调整递推关系仍难以建立，这时，通过增加其它量或函数来建立递推关系式也是一种思考方向(类似于数学归纳法证明时的“加强命题”)。因为，用动态规划解题的一个重要特征是通过递推，而递推是利用原有结果得到新结果的过程。如果在理论上可以证明，一个难以直接实现递推的问题可以通过引入新的递推关系，同时将两者解决，这看起来把问题复杂化了，而实际上由于对于每一步递推，在增加了解决的问题的同时也增加了条件（以前解决的值），反而使递推容易进行。举例说明，IOI’98 中的“多边形”一题，大意如下：

有一个多边形（N 边形），顶点上放整数，边上放“+”或“*”，要寻找一种逐次运算合并的顺序，通过 N-1 次运算，使最后结果最大。

如果单纯考虑用 MAX[I,L]，从 I 开始进行 L 个运算所得的最大值，则难以实现递推，而根据数学知识，引入了 MIN[I,L]为从 I 开始进行 L 个运算所得的最小值，在进行递推时，却能够有效地用较小的 I，L 来得到较大时的结果，从而事实上同时解决了最小值与最大值两个问题。

递推关系式如下：(考虑 I 从 1 到 N,L 从 1 到 N-1)

考虑 t(最后一步运算位置)从 0 到 L-1：

如果最后一步运算为"+"则：

min(i,L)=最小值{min(i,t)+min((i+t+1-1) mod N+1,L-t-1)}

max(i,L)=最大值{max(i,t)+max((i+t+1-1) mod N+1,L-t-1)}

如果最后一步运算为"*"则：

```
min(i,L)=最小值{min(i,t)*min((i+t+1-1) mod N+1,L-t-1),
          min(i,t)*max((i+t+1-1) mod N+1,L-t-1),
                    max(i,t)*min((i+t+1-1) mod N+1,L-t-1),
          max(i,t)*max((i+t+1-1) mod N+1,L-t-1)}
max(i,L)=最大值{min(i,t)*min((i+t+1-1) mod N+1,L-t-1),
          min(i,t)*max((i+t+1-1) mod N+1,L-T-1)
                    max(i,t)*min((i+t+1-1) mod N+1,L-t-1),
          max(i,t)*max((i+t+1-1) mod N+1,L-t-1)}
```

(源程序见附录中的程序 3)

此外，动态规划通过递推来实现，因而问题与子问题越相似，越有规律就越容易进行操作。因而对于某些自身的阶段和规律不怎么明显的问题，可以通过一个预处理，使其变得更整齐，更易于实现。例如，ACM'97 亚洲赛区/上海区竞赛一题"正则表达式(Regular Expression)的匹配"问题，题目大意是：

正则表达式是含有通配符的表达式，题目定义的广义符有：

- . 表示任何字符
- [c1-c2] 表示字符 c1 与 c2 间的任一字符
- [^c1-c2] 表示不在字符 c1 与 c2 间的任一字符
- * 表示它前面的字符可出现 0 或多次
- + 表示它前面的字符可出现一次或多次
- \ 表示它后面的字符以一个一般字符对待。

对一个输入串，寻找最左边的与正则表达式匹配的串(相同条件下要最长的)。这里如果不作预处理，则有时一个广义符可对应多个字符，有时又是多个广义符仅对应一个字符，给系统化处理带来很多麻烦。因而有必要对正则表达式进行标准化，使得或者某个结点仅对应一个字符，或者用一特殊标记表明它可以重复多次。定义记录类型：

```
NodeType=Record
  StartChar: Char; {开始字符}
  EndChar: Char;  {结束字符}
  Belong:  Boolean {是否属于}
  Times:  Boolean; {False: 必须一次; True:可以多次，也可以不出现}
End;
```

对输入数据预处理之后，建立递推关系就不太困难了。用 Pro[i,j]表示前 i 个正则表达式结点对以第 j 个字符为终点的子串的匹配情况(True/False)，对于为

True 的情况，同时指明此条件下最先的开始位置。如果第 i 个正则表达式结点是仅出现一次的，那么，如果它与第 j 个字符不匹配，则该值为 False，否则，它与 Pro[i-1,j-1]相同。(初始时 Pro[0,x]=True)。如果它是可重复多次的，那么它可以被解释成 0 个或多个字符。在它自身与相应位置的 0 个或多个字符匹配的条件下依次考虑这些可能情况，只要其中含 True，则 Pro[i,j]为 True，同时记录下这些达到 True 的情况中起点最先的。按此递推，直到 i 达到结点个数。(源程序见所附程序 4)

三、动态规划实现中的问题

动态规划解决问题在有了基本的思路之后，一般来说，算法实现是比较好考虑的，但有时也会遇到一些问题，而使算法难以实现。动态规划思想设计的算法从整体上来看基本都是按照得出的递推关系式进行递推，这种递推，相对于计算机来说，只要设计得当，效率往往是比较高的，这样在时间上溢出的可能性不大，而相反地，动态规划需要很大的空间以存储中间产生的结果，这样可以使包含同一个子问题的所有问题共用一个子问题解，从而体现动态规划的优越性，但这是以牺牲空间为代价的，为了有效地访问已有结果，数据也不易压缩存储，因而空间矛盾是比较突出的。另一方面，动态规划的高时效性往往要通过大的测试数据体现出来（以与搜索作比较），因而，对于大规模的问题如何在基本不影响运行速度的条件下，解决空间溢出的问题，是动态规划解决问题时一个普遍会遇到的问题。

对于这个问题，我认为，可以考虑从以下一些方面去尝试：

一个思考方向是尽可能少占用空间。如从结点的数据结构上考虑，仅仅存储必不可少的内容，以及数据存储范围上精打细算(按位存储、压缩存储等)。当然这要因题而异，进行分析。另外，在实现动态规划时，一个我们经常采用的方法是用一个与结点数一样多的数组来存储每一步的决策，这对于倒推求得一种实现最优解的方法是十分方便的，而且处理速度也有一些提高。但是在内存空间紧张的情况下，我们就应该抓住问题的主要矛盾。省去这个存储决策的数组，而改成在从最优解逐级倒推时，再计算一次，选择某个可能达到这个值的上一阶段的状态，直到推出结果为止。这样做，在程序编写上比上一种做法稍微多花一点时间，运行的时效也可能会有一些(但往往很小)的下降，但却换来了很多的空间。因而这种思想在处理某些问题时，是很有意义的。

但有时，即使采用这样的方法也会发现空间溢出的问题。这时就要分析，这些保留下来的数据是否有必要同时存在于内存之中。因为有很多问题，动态规划递推在处理后面的内容时，前面比较远处的内容实际上是用不着的。对于这类问题，在已经确信不会再被使用的数据上覆盖数据，从而使空间得以重复利用如果能有效地使用这一手段，对于相当大规模的问题，空间也不至于溢出。(为了求出最优方案，保留每一步的决策仍是必要的，这同样需要空间。)一般地说这种方法可以通过两种思路来实现。一种是递推结果仅使用 Data1 和 Data2 这样两个数组，每次将 Data1 作为上一阶段，推得 Data2 数组，然后，将 Data2 通过复制覆盖到 Data1 之上，如此反复，即可推得最终结果。这种做法有一个局限性就是对于递推与前面若干阶段相关的问题，这种做法就比较麻烦；而且，每递推一级，就需要复制很多的内容，与前面多个阶段相关的问题影响更大。另外一

种实现方法是，对于一个可能与上 N 阶段相关的问题，建立数组 Data[0..N]，其中各项即为与原 Data1/Data2 相同的内容。这样不采用这种内存节约方式时对于下标 K 的访问只要对应成对下标 K mod (N+1)的访问，就可以了。与不作这种处理的方法相比，对于程序修改的代码很少，速度几乎不受影响(用电脑做 MOD 运算是很快的)，而且需要保留不同的阶段数也都能很容易实现。这种手段对不少题目都适用，比如：NOI'98 的"免费馅饼"，题目大意是：

有一个舞台，宽度 W 格(1≤W≤99 的奇数),高度 H 格(1≤H≤100)，游戏者在时刻 0 时位于舞台正中，每个单位时间可以从当时位置向左移 2 格、向左移 1 格保持不动、向右移 1 格或者向右移 2 格，每个馅饼会告知初始下落的时间和位置以及下落速度(1 秒内下移的格子数)和分值。仅在某 1 秒末与游戏者位于同一格内的馅饼才被认为是接住的。求一种移动方案，使得分最大。注意：馅饼已按初始下落时间排序。

从问题来看，想到动态规划并不是很困难的。但是，题中规定初始下落时间从 0 到 1000，而且考虑下落到最后可能时间要到 1100 左右,而宽度可达 99，以时间-位置作为状态决定因素进行递推，速度不会慢，但如果采用初始数据经预处理后的结果(即在何时到何地可得多少分的描述数组)用一个数组，动态规划递推用一个数组，记录每步决策用一个数组，因得分题中未指出可能的大小如果采用前两个 Longint 型，最后一个 Shortint 型，所须内存约为 1100*99*9 字节，即约 957KB，这显然是不可能存得下的。但是注意到在进行递推时，一旦某一个(时间，位置)对应的最大分值一确定，这个位置的原始数据就不再有用，因而两者可以合二为一，从而只要 1100*99*5 字节，即约 532KB。这样对于题目规模的问题就勉强可以解决了。 当然，如果更进一步思考，其实这个问题中递推是仅与上一个时间有关的，而馅饼实际上仅使用了当前位置的值。由于初始下落时间已经排序，那么当读到初始下落时间晚于当前处理时间时，就不必马上读入。为了避免重复和无规律地读盘和内存开销过大，只要记录下当前之后约 100 个时间单位内的情况就可以了，使用前面所说的循环使用内存的方法，只要 101*99*4+99*2*2=40392 字节，不到 40KB，而对于每一个时间仅需 99 个 shortint 存储决策即可,就算把问题规模提高到 3000 或者 4000 个时间单位也能顺利出解。 (源程序见附录中的程序 5)

当采用以上方法仍无法解决内存问题时，也可以采用对内存的动态申请来使绝大多数测试点能有效出解(而且，使用动态内存还有一点好处，就是在重复使用内存而进行交换时，可以只对指针进行交换，而不复制数据)，这在竞赛中也是十分有效的。

四、总结

动态规划是一种重要的程序设计思想。但是，由于它没有确定的算法形式因而也就有较大的灵活性，但它的本质却具有高度的相似性。所以，学习和使用这一思想方法，关键在于在理解和把握其本质的基础上灵活运用。本文虽然谈到了一些思想方法，但这些仅是对一些较普遍问题的讨论，针对具体问题进行具体分析建立数学模型才是最重要而关键之处。

【参考资料】

1、吴文虎、王建德《实用算法的分析与程序设计》电子工业出版社 ISBN 7-5053-4402-1/TP.2036

2、吴文虎、王建德《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》清华大学出版社 ISBN 7-302-02203-8/TP.1060

3、 NOI'97、NOI'98、IOI'97、IOI'98 试题，ACM'97 亚洲赛区试题

【程序】

程序 1 IOI'97 字符识别

```
{"字符识别"的基本动态规划方程已在正文中说明，这里补充说明一下本题提高速度的关键—错位比较时提高效率。}
{注意到少一行与多一行时的比较，虽然可能出现错位，但每一行仅有与邻近的两行比较的可能，}
{先把可能的比较记录下来，再累计从端点到某一位置的非错位时反相数之和与错位时反相数之和，}
{考虑 20 种情况，仅需一重循环(不考虑比较一行的子程序内的循环)即可，效率得到很大提高}
program Character_Recognition; {"字符识别"程序}
const
  cc:array [1..27] of char=(' ','A','B','C','D','E','F','G','H','I','J','K','L',
                            'M','N','O','P','Q','R','S','T','U','V','W','X','Y','Z'); {字符常量}
var
  f,f1,f2:text; {文件变量}
  font:array [1..540] of longint; {记录字形的数组,一个 longint 数表示 20 位 2 进制数，下同}
  dd:array [1..1200] of longint; {待分析的点阵数据}
  str:string; {读入的串}
  i,j,k:integer; {辅助变量}
  t:word;
  ff:integer;
```

```
  bin:array [1..20] of longint; {2 的幂}
  pro:array [0..1200] of word; {动态规划数组}
  sta:array [0..1200] of byte; {每步分析的最优字符序号}
  bf:array [0..1200] of word; {每步分析的上一个字符的终点}
  pf:array [1..21,0..1] of word; {错位比较时用}
  n:integer;

procedure getnum(var l:longint); {String->longint 转换}
var
  i:integer;
begin
  l:=0;
  for i:=1 to 20 do
    if str[i]='1' then inc(l,bin[i]);
end;

function compare0(a,b:longint):byte; {比较 a 表示的行与 b 表示的行的反相个数}
var
  k:byte;
  i:integer;
begin
  a:=a xor b;
  k:=0;
  for i:=1 to 20 do
    if a and bin[i]<>0 then inc(k);
  compare0:=k
end;


function compare20(k:integer; var ff:integer):word; {比较 20 行的最优结果}
var
  i,j,t,s:word;
  best:word; {当前最优}
begin
  ff:=0;
  best:=maxint;
  for t:=1 to 27 do {考虑 27 个字符}
  begin
    j:=0;
    for i:=1 to 20 do
    begin
      s:=compare0(font[(t-1)*20+i],dd[i+k-1]); {比较一行}
      inc(j,s); {累计差别}
    end;
```

```
   if j<best then begin best:=j; ff:=t end; {如果更优，记录之}
  end;
  compare20:=best;
end;

function compare19(k:integer; var ff:integer):word; {返回与 k 开始 19 行最接近的字符与差别值}
var
  i,j,t,s:word;
  best:word;
  l1,l2:array [0..20] of word;
  bb,fx:integer;
begin
  ff:=0;
  best:=maxint;
  for t:=1 to 27 do {考虑 27 个字符}
  begin
    j:=0;
    fillchar(l1,sizeof(l1),0);
    fillchar(l2,sizeof(l2),0);
    for i:=1 to 19 do
      for j:=0 to 1 do
        pf[i,j]:=compare0(font[(t-1)*20+i+j],dd[k+i-1]);
        {记录 19 行中第 i 行与对应字形中第 i 行、第 i+1 行的差别}
    l1[1]:=pf[1,0]; {l1[i]为破损字形前 i 行与标准字形前 i 行匹配的差别} {}
    for i:=2 to 19 do
      l1[i]:=l1[i-1]+pf[i,0];
    l2[19]:=pf[19,1]; {l2[i]为破损字第 i 行之后与标准字形第 i+1 行之后的字形匹配的差别}
    for i:=18 downto 1 do
      l2[i]:=l2[i+1]+pf[i,1];
    bb:=maxint;
    for i:=1 to 20 do {20 种缺少方式}
      if l1[i-1]+l2[i]<bb then bb:=l1[i-1]+l2[i]; {记录最少的}
    if bb<best then begin best:=bb; ff:=t end; {如果该字符较匹配，改进 BEST}
  end;
  compare19:=best
end;

function compare21(k:integer; var ff:integer):word;
{返回与第 k 行开始的 21 行最匹配的字形与差别}
var
  i,j,t,s:word;
  best:word;
  l1,l2:array [0..22] of word;
```

```
  bb,fx:integer;
begin
  ff:=0;
  best:=maxint;
  for t:=1 to 27 do {考虑 27 个字形}
  begin
    j:=0;
    fillchar(l1,sizeof(l1),0);
    fillchar(l2,sizeof(l2),0);
    fillchar(pf,sizeof(pf),0);
    for i:=1 to 21 do
      for j:=0 to 1 do
          if not ((i=21) and (j=0)) and not ((i=1) and (j=1)) then pf[i,j]:=compare0(font[(t-1)*20+i-
j],dd[k+i-1]);
       {用破损字形第 i 行与标准字形第 i 行、第 i-1 行比较，记录差别}
    l1[1]:=pf[1,0]; {l1[i]为前 i 行与标准前 i 行匹配的差别}
    for i:=2 to 20 do
      l1[i]:=l1[i-1]+pf[i,0];
    l2[22]:=0; {l2[i]为第 i 行开始的内容与标准从 i-1 行开始的内容进行匹配的差别}
    for i:=21 downto 2 do
      l2[i]:=l2[i+1]+pf[i,1];
    bb:=maxint;
    for i:=1 to 20 do {比较 20 种方式}
      if l1[i-1]+l2[i+1]+pf[i,0]<bb then bb:=l1[i-1]+l2[i+1]+pf[i,0];
    if bb<best then begin best:=bb; ff:=t end;
  end;
  compare21:=best
end;



begin {主程序}
  assign(f,'Font.Dat');
  reset(f);
  assign(f1,'Image.dat');
  reset(f1);
  assign(f2,'Image.out');
  rewrite(f2); {文件关联}

  readln(f,k); {读入行数 540}
  bin[1]:=1;
  for i:=2 to 20 do
    bin[i]:=bin[i-1]*2; {生成 bin 数组，为 2 的幂}
```

```
for i:=1 to k do
begin
  readln(f,str);
  getnum(font[i]); {string->longint 转换}
end;

read(f1,n); {读入分析文件的各行}
for i:=1 to n do
begin
  readln(f1,str);
  getnum(dd[i]); {string->longint 转换}
end;

fillchar(pro,sizeof(pro),255); {初值 65535}
fillchar(sta,sizeof(sta),0); {每步分析出的最优字符}
fillchar(bf,sizeof(bf),255); {每步分析出的最优切割}
pro[0]:=0;
sta[0]:=0;
bf[0]:=0;
for i:=19 to n do {考虑 19 行至 n 行}
begin
  if (i>=19) and (pro[i-19]<>65535) then {如果切去一个 19 行字符后的情况可能}
  begin
    t:=compare19(i-19+1,ff); {比较从 i-19+1 起 19 行与标准结果最接近的结果与差别}
    if t+pro[i-19]<pro[i] then {如果更优，更新动态规划数组}
    begin
      pro[i]:=t+pro[i-19];
      sta[i]:=ff;
      bf[i]:=i-19;
    end;
  end;
  if (i>=20) and (pro[i-20]<>65535) then {如果切去一个 20 行字符后的情况可能}
  begin
    t:=compare20(i-20+1,ff); {比较从 i-20+1 起 20 行与标准结果最接近的结果与差别}
    if t+pro[i-20]<pro[i] then {如果更优，更新动态规划数组}
    begin
      pro[i]:=t+pro[i-20];
      sta[i]:=ff;
      bf[i]:=i-20;
    end;
  end;
  if (i>=21) and (pro[i-21]<>65535) then {如果切去一个 21 行字符后的情况可能}
  begin
    t:=compare21(i-21+1,ff); {比较从 i-21+1 起 21 行与标准结果最接近的结果与差别}
```

```
    if t+pro[i-21]<pro[i] then {如果更优，更新动态规划数组}
    begin
      pro[i]:=t+pro[i-21];
      sta[i]:=ff;
      bf[i]:=i-21;
    end;
  end;
 end;

 str:='';
 i:=n;
 if pro[n]=65535 then begin writeln(f2,'No answer.'); close(f1); close(f2); halt end; {如果无解}
 while i<>0 do {倒推求解}
 begin
  str:=cc[sta[i]]+str;
  i:=bf[i];
 end;
 writeln(f2,str); {输出}
 close(f1);
 close(f2)
end.
```

程序 2 NOI'97 积木游戏

```
{说明：}
{为防止出现内存溢出，程序采用逐级递推的方式。}
program Toy_Bricks_Game; {积木游戏}
type
 jmtype=array [0..100] of ^node;
 {动态规划过程中仅保留与当前最接近的一个阶段的情况}
 {这是存储一个阶段的量的指针类型}
 node=array [0..300] of longint;
var
 f1,f2:text; {文件变量}
 size:array [0..300,1..2] of word; {各个出现的面的记录,0 对应无面积要求}
 num:integer; {记录的不同面数}
 n,m:integer; {积木数、堆成的堆数}
 i,j,k,t:integer; {辅助变量}
 p,q,r:jmtype; {递推阶段指针数组，每个保留一个阶段（K 值），从 p 到 q，r 用于交换}
 aa:array [1..100,1..3] of word; {边长数据}
 ss:array [1..100,1..3] of word; {3 个面对应的面序号，对同样长、宽的面作了优化}
```

```
procedure sort(i:integer); {对第 i 个长方体的三边长排序}
var
  j,k,t:integer;
begin
  for j:=1 to 2 do
    for k:=j+1 to 3 do
      if aa[i,j]>aa[i,k] then
      begin
        t:=aa[i,j];
        aa[i,j]:=aa[i,k];
        aa[i,k]:=t
      end;
end;

function add(a1,a2:integer):integer; {在面标记中增添当前面，并返回相应的序号}
var
  i,j:integer;
begin
  for i:=1 to num do
    if (a1=size[i,1]) and (a2=size[i,2]) then begin add:=i; exit end;
  inc(num);
  size[num,1]:=a1;
  size[num,2]:=a2;
  add:=num;
end;

procedure preprocess; {预处理，将要处理的面记入}
var
  i,j,k:integer;
begin
  num:=0;
  for i:=1 to n do
  begin
    sort(i);
    ss[i,1]:=add(aa[i,1],aa[i,2]);
    ss[i,2]:=add(aa[i,1],aa[i,3]);
    ss[i,3]:=add(aa[i,2],aa[i,3]);
  end;
end;

procedure check(ii,nn,hh:integer);  {检查用 ii 积木的 nn 放置方式，是否有效}
var
  g,h:integer;
begin
```

```
  if (p[ii-1]^[0]>=0) and (p[ii-1]^[0]+hh>=q[ii]^[j]) then q[ii]^[j]:=p[ii-1]^[0]+hh; {考虑另成一堆}
  if (p[ii-1]^[ss[ii,nn]]>=0) and (q[ii-1]^[ss[ii,nn]]+hh>=q[ii]^[j]) then q[ii]^[j]:=q[ii-1]^[ss[ii,nn]]+hh;
  {考虑放在前一堆上}
end;

begin
  assign(f1,'ToyBrick.in');
  reset(f1);
  assign(f2,'ToyBrick.out');
  rewrite(f2); {文件关联、打开}
  readln(f1,n,m); {读入 N、M 值}
  for i:=1 to n do {读入边长}
    readln(f1,aa[i,1],aa[i,2],aa[i,3]);

  size[0,1]:=0;
  size[0,2]:=0;
  preprocess; {生成各种待处理的面}

  for i:=0 to n do {动态内存初始化}
  begin
    new(p[i]);
    new(q[i]);
    fillchar(q[i]^,sizeof(q[i]^),0);
  end;

  for t:=1 to m do {连续递推 M 阶段,分成 T 堆}
  begin
    r:=q;
    q:=p;
    p:=r; {交换 P、Q}
    for i:=0 to n do fillchar(q[i]^,sizeof(q[i]^),255); {Q 初始化}
    for i:=t to n do {考虑 T 个到 N 个积木}
    begin
      for j:=0 to num do {考虑最后"输出"的面的约束条件}
      begin
        q[i]^[j]:=q[i-1]^[j]; {当前积木不用}
        if (aa[i,1]>=size[j,1]) and (aa[i,2]>=size[j,2]) then check(i,1,aa[i,3]);
        if (aa[i,1]>=size[j,1]) and (aa[i,3]>=size[j,2]) then check(i,2,aa[i,2]);
        if (aa[i,2]>=size[j,1]) and (aa[i,3]>=size[j,2]) then check(i,3,aa[i,1]);
      {如果当前积木的某方向放置可以满足此要求，考虑按此方向放置该块作为新的一堆的底或加在前一堆上（如果可能）}
      end;
```

```
    end;
  end;
  writeln(f2,q[n]^[0]); {输出答案}
  close(f1);
  close(f2)
end.
```

程序 3 IOI'98 多边形

```
program Polygon; {"多边形"程序}
var
  f1,f2:text; {输入、输出文件变量}
  n:integer; {顶点个数}
  data:array [1..50] of integer; {原始数据-顶点}
  sign:array [1..50] of char; {原始数据-运算符}
  i,j,k,l:integer; {辅助变量}
  t,s,p:integer; {辅助变量}

  ans:set of 1..50; {可能达到最大值的第一次移动的边的序号}
  best:integer; {当前最优解}
  min,max:array [1..50,0..50] of integer;
  {动态规划表格，min[i,l]表示从第 i 个顶点开始，经过 l 个符号按合理运算所得的结果的最
小值；max 与之类似，但为最大值}
  first:boolean; {首次输出标志}

procedure init; {初始化，读入原始数据}
var
  i:integer;
  ch:char;
begin
  readln(f1,n);
  for i:=1 to n do
  begin
    repeat
      read(f1,ch);
    until ch<>' ';
    sign[i]:=ch; {sign[i]位于 data[i]与其后顶点间}
    read(f1,data[i]);
  end;
end;
```

```
begin
 {文件关联、打开}
 assign(f1,'Polygon.in');
 reset(f1);
 assign(f2,'Polygon.out');
 rewrite(f2);

 {初始化}
 init;

 {赋初值}
 best:=-maxint-1;
 ans:=[];
 fillchar(max,sizeof(max),0);
 fillchar(min,sizeof(min),0); {数组初始化}
 for j:=1 to n do
 begin
  max[j,0]:=data[j];
  min[j,0]:=data[j];
 end; {初值是不经过运算(l=0)的值}

 for l:=1 to n-1 do {考虑长度由 1 到 n-1}
  for k:=1 to n do {考虑起始点从 1 到 n}
  begin
   max[k,l]:=-maxint-1;
   min[k,l]:=maxint;
   for t:=0 to l-1 do {考虑分开前半部分经过的运算数}
   begin
    case sign[(k+t+1-1) mod n+1] of {考虑分开处的符号}
     't': {为加法}
     begin
      if max[k,t]+max[(k+t+1-1) mod n+1,l-t-1]>max[k,l] then max[k,l]:=max[k,t]+max[(k+t+1-
1) mod n+1,l-t-1];
       {最大值更新}
       if min[k,t]+min[(k+t+1-1) mod n+1,l-t-1]<min[k,l] then min[k,l]:=min[k,t]+min[(k+t+1-1)
mod n+1,l-t-1];
       {最小值更新}
     end;
     'x': {为乘法}
     begin
      for p:=1 to 4 do
      begin
       case p of
```

```
            1: s:=max[k,t]*max[(k+t+1-1) mod n+1,l-t-1];
            2: s:=max[k,t]*min[(k+t+1-1) mod n+1,l-t-1];
            3: s:=min[k,t]*max[(k+t+1-1) mod n+1,l-t-1];
            4: s:=min[k,t]*min[(k+t+1-1) mod n+1,l-t-1]; {考虑四个乘积}
          end;
          if s>max[k,l] then max[k,l]:=s;
          if s<min[k,l] then min[k,l]:=s; {更新最大最小值}
        end;
      end;
    end;
  end;
 for i:=1 to n do
  if max[i,n-1]>best then begin best:=max[i,n-1]; ans:=[i] end
  else if max[i,n-1]=best then include(ans,i); {更新全局的最大值}

 writeln(f2,best); {输出最大值}
 first:=true;
 for i:=1 to n do
  if i in ans then
  begin
      if first then first:=false
   else write(f2,' ');
   write(f2,i);
  end;
 writeln(f2); {输出首次被移动的边}
 close(f1);
 close(f2) {关闭文件}
end. {结束}
```

程序 4 ACM'97 亚洲区/上海赛题 正则表达式匹配

```
program Expression_Match; {正则表达式匹配程序}
type
 datatype=record {预处理数据类型}
  st,ed:char; {起始、结束字符}
  md:0..1; {重复方式 0：一次；1：0 或多次}
  mt:0..1; {匹配方式 0：不包含为匹配；1：包含为匹配}
 end;
var
 f1,f2:text; {文件变量}
 s1,s2:string; {正则表达式串、待匹配串}
```

```
  str:string;
  len:integer; {正则表达式预处理后的"长度"}
  dd:array [1..80] of datatype; {预处理结果}
  pro:array [0..80,0..80] of boolean; {动态规划数组}
  fr:array [0..80,0..80] of byte;
  {FR[i,j]表示以第 j 个字符为尾的与前 i 项正则表达式匹配的最前端的字符位置}
  i,j,k,l:integer; {辅助变量}
  ok:boolean; {找到标记}
  ha:boolean;
  ans:integer;
  bt,bj:integer; {当前最优值的开始位置、长度}
procedure preprocess; {预处理，生成规划的"正则表达式"表示}
var
  i,j,k:integer;
  ch,c:char;
begin
  i:=0;
  j:=0;
  while i<length(s1) do
  begin
    inc(i);
    case s1[i] of
      '.':
      begin {处理"."}
        inc(j);
        dd[j].md:=0;
        dd[j].mt:=1;
        dd[j].st:=#0;
        dd[j].ed:=#255;
      end;
      '*': dd[j].md:=1; {处理"*"}
      '+': {处理"+"}
      begin
        inc(j);
        dd[j]:=dd[j-1];
        dd[j].md:=1;
      end;
      '\': {处理"\"}
      begin
        inc(i);
        inc(j);
        dd[j].md:=0;
        dd[j].mt:=1;
        dd[j].st:=s1[i];
```

```
      dd[j].ed:=s1[i];
    end;
    '[': {处理"["}
    begin
      inc(i);
      inc(j);
      dd[j].md:=0;
      dd[j].mt:=1;
      if s1[i]='^' then begin inc(i); dd[j].mt:=0 end; {如果含"^"}
      if s1[i]='\' then inc(i);
      dd[j].st:=s1[i];
      inc(i,2);
      if s1[i]='\' then inc(i);
      dd[j].ed:=s1[i];
      inc(i);
    end
    else
    begin {处理一般字符}
      inc(j);
      dd[j].st:=s1[i];
      dd[j].ed:=s1[i];
      dd[j].mt:=1;
      dd[j].md:=0;
    end;
  end;
 end;
 len:=j
end;


begin
 assign(f1,'Match.in');
 reset(f1);
 assign(f2,'Match.out');
 rewrite(f2); {文件关联、打开}

 while true do
 begin
  readln(f1,s1);
  if s1='end' then break; {如果为 end 串，跳出}
  readln(f1,s2);
  preprocess; {预处理}
  ok:=false; {标记未找到}
  fillchar(pro,sizeof(pro),false);
```

```
  fillchar(pro[0],sizeof(pro[0]),true);
  fillchar(fr,sizeof(fr),0);
  bt:=maxint;
  bj:=0;
  for i:=0 to length(s2) do {赋初值}
   fr[0,i]:=i+1;
  for i:=1 to len do {分析前 i 项正则表达式}
   for j:=1 to length(s2) do {分析前 j 个字符}
   begin
    if dd[i].md=0 then {如果最后一个是一般字符}
     if (dd[i].mt=0) xor (s2[j] in [dd[i].st..dd[i].ed]) {如果匹配}
     then
     begin
      pro[i,j]:=pro[i-1,j-1]; {与去掉这个字母后的结果一致}
      if pro[i,j] then fr[i,j]:=fr[i-1,j-1]; {如果为真，设置起始点}
     end
     else pro[i,j]:=false {最后一个不匹配，则整个不匹配}
    else
    begin
     ha:=false;
     for k:=j downto 0 do {考虑前 i-1 项正则表达式与前若干项字符串匹配情况}
     begin
      if pro[i-1,k] then {如果某个为真}
      begin
       pro[i,j]:=true; {表示匹配}
       if (fr[i,j]=0) or (fr[i,j]>fr[i-1,k]) then fr[i,j]:=fr[i-1,k];
       {如果起点较早，更新之}
      end;
      if not ((dd[i].mt=0) xor (s2[k] in [dd[i].st..dd[i].ed]))
      {如果不匹配，则不考虑再退一格的情况}
      then begin ha:=false; break end;
     end;
    end;
    if (i=len) and pro[i,j] and (fr[i,j]<=bt) then
    {如果发现更好的，更新当前最优值}
    begin
     ok:=true;
     bt:=fr[i,j];
     bj:=j-bt+1;
    end;
   end;
    if ok then
          if bj<>0 then writeln(f2,copy(s2,1,bt-1), '(', copy(s2,bt,bj), ')',
copy(s2,bt+bj,length(s2)-bt-bj+1)) {如果找到，输出}
```

```
        else writeln(f2,'()',s2)
        {对于某些理论上讲可以与空串匹配的正则表达式，应看作与第一个字符前的空串匹配，这里作了专门处理}
    else writeln(f2,s2); {否则输出原串}
  end;
  close(f1);
  close(f2)
end.
```

程序 5 NOI'98 免费馅饼

```
{说明：}
{动态规划方程：}
{ ans[t,j]= max ( ans[t-1,j+k] ) (其中，k=-2,-1,0,1,2，且相应位置可达)}
program Pizza_For_Free {Time Limit:3000};
{免费馅饼 将允许时间扩大到 3000 秒，分值限制在 longint 范围内}
type
  link=^linetype; {指向记录一个时间单位决策数组的指针类型}
  linetype=array [1..99] of shortint; {记录一个时间单位决策的数组}
var
  f1,f2:text; {输入、输出文件变量}
  w,h:integer; {宽度、高度}
  dd:array [0..100] of array [1..99] of longint; {循环使用的数组，用于表示对应时刻、位置的得分值}
  pro:array [0..3100] of link; {记录决策信息的动态数组}
  dt:array [0..1,1..99] of longint; {循环使用的动态规划数组}
  i,j,k,t:integer; {辅助变量}
  bt,bj:integer; {最大状态记录}
  best:longint; {最大值记录}
  maxt:integer; {最大时间}
  ans:array [1..3100] of shortint; {倒推答案时用的暂存区}
  num:integer;
  pp:longint;
  _t,_j,_v:integer; {暂存初始时间、位置、速度}
  _fz:longint; {暂存分值}
  _saved:boolean; {是否暂存标志}

procedure try_reading; {读入原始数据的过程，其中使用了分段读取的方法。读入初始下落时间不大于 t 的馅饼}
var
  t0,i,j,v:integer; {初始下落时间}
  fz:longint; {分值}
```

```
begin
  fillchar(dd[(t+100) mod 101],sizeof(dd[(t+100) mod 101]),0);
  {t 及以后时间读入的块，至迟在 t+99 时间即进入可接收状态，可对 t+100(循环观点下)单元清 0，以便下次使用}
  if _saved then {如果上次有预存}
  begin
    if _t>t then exit; {如果预存结果仍不被使用到，则返回，否则用预存结果记录新馅饼}
    _saved:=false;
    if (h-1) mod _v=0 then
    begin
      inc(dd[(_t+(h-1) div _v) mod 101,_j],_fz);
      if (_t+(h-1) div _v)>maxt then maxt:=_t+(h-1) div _v;
    end;
  end;
  if eof(f1) then exit; {文件结束就返回}

  while true do {不断读取，直到初始下落时间大于当前处理时间}
  begin
    if eof(f1) then exit;
    readln(f1,t0,j,v,fz);
    if t0>t then
    begin
      _v:=v;
      _t:=t0;
      _j:=j;
      _fz:=fz;
      _saved:=true;
      exit {暂存返回}
    end;
    if (h-1) mod v<>0 then continue; {标记时间-位置与得到的分值}
    inc(dd[(t0+(h-1) div v) mod 101,j],fz);
    if t0+(h-1) div v>maxt then maxt:=t0+(h-1) div v;
  end;
end;



begin {主程序}
  assign(f1,'INPUT.TXT');
  reset(f1);
  assign(f2,'OUTPUT.TXT');
  rewrite(f2); {文件关联、打开}

  readln(f1,w,h); {读入宽、高}
```

```
for i:=0 to 3100 do {动态内存初始化}
begin
  new(pro[i]);
  fillchar(pro[i]^,sizeof(pro[i]^),0);
end;
for i:=0 to 1 do
      for j:=1 to 99 do
        dt[i,j]:=-maxlongint-1;
{赋初值}
dt[0,(1+w) div 2]:=0; {-maxlongint-1 表示该点不可达}
fillchar(dd,sizeof(dd),0);
t:=0;
maxt:=0;
_saved:=false;
try_reading;
best:=0;
bt:=0;
bj:=(w+1) div 2;
best:=dd[0,(w+1) div 2];

while true do
begin
  inc(t); {考虑下一个时间}
  try_reading; {读入数据}
  if eof(f1) and (t>maxt) and not _saved then break; {如果没有新的数据，跳出}
  for j:=1 to w do {考虑各个位置}
  begin
    dt[t mod 2,j]:=-maxlongint-1;
    for k:=-2 to 2 do {考虑 5 种移动方式}
      if (j+k>0) and (j+k<=w) and (dt[1-t mod 2,j+k]>-maxlongint-1) {如果可能}
      and (dt[1-t mod 2,j+k]+dd[t mod 101,j]>dt[t mod 2,j]) then {而且有效}
      begin
        dt[t mod 2,j]:=dt[1-t mod 2,j+k]+dd[t mod 101,j]; {更新当前最优值}
        pro[t]^[j]:=k;
      end;
    if dt[t mod 2,j]>best then {如果到目前最优，更新之}
    begin
      best:=dt[t mod 2,j];
      bt:=t;
      bj:=j;
    end;
  end;
end;
```

```
  writeln(f2,best); {输出最大值}
  num:=0;
  j:=bj;
  for t:=bt downto 1 do {倒推求解}
  begin
    ans[t]:=-pro[t]^[j];
    inc(j,pro[t]^[j]);
  end;
  for t:=1 to bt do
    writeln(f2,ans[t]);
  close(f1);
  close(f2)
end.
```

论文附录： 本文所引题目详细内容

1、字符识别 (IOI'97)

Character Recognition

This problem requires you to write a program that performs character recognition.

Details:

Each ideal character image has 20 lines of 20 digits. Each digit is a '0' or a '1'. See Figure 1a for the layout of character images in the file.

The file FONT.DAT contains representations of 27 ideal character images in this order:

□abcdefghijklmnopqrstuvwxyz

where □ represents the space character.

The file IMAGE.DAT contains one or more potentially corrupted character images. A character image might be corrupted in these ways:

- at most one line might be duplicated (and the duplicate immediately follows)
- at most one line might be missing
- some '0' might be changed to '1'
- some '1' might be changed to '0'.

No character image will have both a duplicated line **and** a missing line. No more than 30% of the '0' and '1' will be changed in any character image in the evaluation datasets.

In the case of a duplicated line, one or both of the resulting lines may have corruptions, and the corruptions may be different.

Task:

Write a program to recognise the sequence of one or more characters in the image provided in file IMAGE.DAT using the font provided in file FONT.DAT.

Recognise a character image by choosing the font character images that require the smallest number of overall changed '1' and '0' to be corrupted to the given font image, given the most favourable assumptions about duplicated or omitted lines. Count corruptions in only the least corrupted line in the case of a duplicated line. All characters in the sample and evaluation images used are recognisable one-by-one by a well-written program. There is a unique best solution for each evaluation dataset.

A correct solution will use precisely all of the data supplied in the IMAGE.DAT input file.

Input:

Both input files begin with an integer N (19 ≤ ≤ 1200) that specifies the number of lines that follow:

N
(digit1)(digit2)(digit3) … (digit20)
(digit1)(digit2)(digit3) … (digit20)
…

Each line of data is 20 digits wide. There are no spaces separating the zeros and ones.

The file FONT.DAT describes the font. FONT.DAT will always contain 541 lines. FONT.DAT may differ for each evaluation dataset.

Output:

Your program must produce a file IMAGE.OUT, which contains a single string of the characters recognised. Its format is a single line of ASCII text. The output should not contain any separator characters. If your program does not recognise a particular character, it must output a '?' in the appropriate position.

Caution: the output format specified above overrides the standard output requirements specified in the rules, which require separator spaces in output.

Scoring:

The score will be given as the percentage of characters correctly recognised.

SEE OTHER SIDE FOR SAMPLES.

Sample files:

Incomplete sample showing the *beginning* of FONT.DAT (space and 'a').

Sample IMAGE.DAT, showing an 'a' corrupted

FONT.DAT	IMAGE.DAT
540	19

00000000000000000000	00000000000000000000
00000000000000000000	00000000000000000000
00000000000000000000	00000000000000000000
00000000000000000000	00000011100000000000
00000000000000000000	00100111011011000000
00000000000000000000	00001111111001100000
00000000000000000000	00001110001100100000
00000000000000000000	00001100001100010000
00000000000000000000	00001100000100010000
00000000000000000000	00000100000100010000
00000000000000000000	00000010000000110000
00000000000000000000	00001111011111110000
00000000000000000000	00001111111111110000
00000000000000000000	00001111111111000000
00000000000000000000	00001000010000000000
00000000000000000000	00000000000000000000
00000000000000000000	00000000000001000000
00000000000000000000	00000000000000000000
00000000000000000000	00000000000000000000
00000000000000000000	
00000000000000000000	
00000000000000000000	
00000000000000000000	
00000011100000000000	
00000111111011000000	
00001111111001100000	
00001110001100100000	
00001100001100010000	
00001100000100010000	
00000100000100010000	
00000010000000110000	
00000001000001110000	
00001111111111110000	
00001111111111110000	
00001111111111000000	
00001000000000000000	
00000000000000000000	
00000000000000000000	
00000000000000000000	
00000000000000000000	
Figure 1a	Figure 1b

Sample output:

IMAGE.OUT	Explanation
A	Recognised the single character ‘A’

Figure 2

中文：

这个题目需要你编写一个字符识别程序。

具体内容：每一个假设的字符图像(字符点阵)有 20 行，每行有 20 个“0”或“1”的数字。

FONT.DAT 文件中有 27 个按照下列顺序排列的字符图像：

□abcdefghijklmnopqrstuvwxyz

□在这里表示空格符。

文件 IMAGE.DAT 包含有一个或者多个被破损的字符图像，一个字符图形可能通过以下几种途径被破损：

- 至多有一行被复制(复制的行紧接其后)
- 至多有一行丢失
- 有些“0”可能变成“1”
- 有些“1”可能变成“0”

字符图像不会同时有一行被复制而同时又丢失一行，在测试数据中，任何一个字符图像弄反“0”和“1”的比例不超过 30%。

在行被复制的情况中，复制行和被复制行都可能破损，但破损的情形可能是不同的。

任务：用 FONT.DAT 提供的字体对 IMAGE.DAT 文件中的一个或者多个字符序列进行识别。

在一种自己最满意的有关“行”被复制或丢失的假设下，根据实际字符图像和标准字符图像的比较，以“0”和“1”发生错误的总数越少越好为条件来识别给定的字符图像，题中所给的样例字符图像都会被一个好的程序所识别，对于一个被测数据组，有一个唯一的最佳解。

正确解应该准确使用由输入文件 IMAGE.DAT 所提供的所有行数。

输入：两个输入文件都由整数 N(19≤N≤1200)开始，该整数指出下面的行数。

N
(digit1)(digit2)(digit3) … (digit20)
(digit1)(digit2)(digit3) … (digit20)
…

每一行的数据都有 20 个码，码和码之间没有空格。

文件 FONT.DAT 描述字体。FONT.DAT 总是包含 541 行。每次 FONT.DAT 都可能是不同的。

输出：你的程序必须生成一个 IMAGE.OUT 文件。它应该包含一串识别出的字符。它的格式是一行 ASCII 码。输出结果不应含有任何分隔符，如果你的程序

识别不出一个字符，则在相应的位置显示“?”。

警告：上述输出格式不遵守在规则中规定的在输出的结果中留出空格的规定。

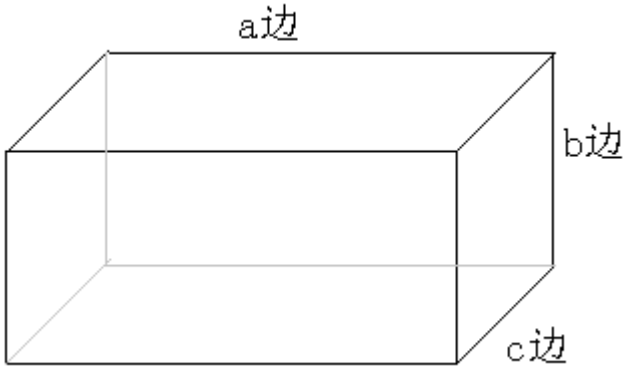
计分：根据正确识别出的字符的比例确定所得分数。

文件举例：(略，参见上面的英文试题)

2、积木游戏 (NOI’97)

SERCOI 最近设计了一种积木游戏。每个游戏者有 N 块编号依次为 1，2，3，4，...的长方体积木，它的三条不同的边分别称为“a 边”、“b 边”、“c 边”，如下图所示：

游戏规划如下：



1、 从 N 块积木中选出若干块，并将它们分成 M(1<=M<=N)堆，称为第 1 堆，第 2 堆，第 3 堆，...。每堆至少有 1 块积木，并且第 K 堆中任意一块积木的编号要大于第 K+1 堆中任意一块积木的编号(2<=K<=M)。

2、 对于每一堆积木，游戏者要将它们垂直摞成一根柱子，并要求满足下面两个条件：

(1) 除最顶上的一块积木外，任意一块积木的上表面同且仅同另一块积木的下表面接触，并且要求下面的积木的上表面能包含上面的积木的下表面，也就是说，要求下面的积木的上表面的两对边的长度分别大于等于上面的积木的两对边的长度。

(2) 对于任意两块上下表面相接触的积木，下面的积木的编号要小于上面的积木的编号。

最后，根据每人所摞成的 M 根柱子的高度之和来决出胜负。

请你编一程序，寻找一种摞积木的方案，使得你所摞成的 M 根柱子的高度之和最大。

输入输出

输入文件是 INPUT.TXT。文件的第一行有两个正整数 N 和 M(1≤M≤N≤100)，分别表示积木总数和要求摞成的柱子数。这两个数之间用一个空格符隔开。接下来 N 行依次是编号从 1 到 N 的 N 个积木的尺寸，每行有三个 1 至 1000 之间的整数，分别表示该积木 a 边、b 边和 c 边的长度。同一行相邻两个

数之间用一个空格符隔开。

输出文件是 OUTPUT.TXT。文件只有一行，为一个整数，表示 M 根柱子的高度之和。

样例

INPUT.TXT
4 2 **10 5 5** **8 7 7** **2 2 2** **6 6 6**

OUTPUT.TXT
24

3、多边形 (IOI'98)

多边形(Polygon)游戏是单人玩的游戏，开始的时候给定一个由 N 个顶点构成的多边形(图 1 所示的例子中，N=4)，每个顶点被赋予一个整数值，而每条边则被赋予一个符号：+(加法运算)或者*(乘法运算)，所有边依次用整数 1 到 N 标识。

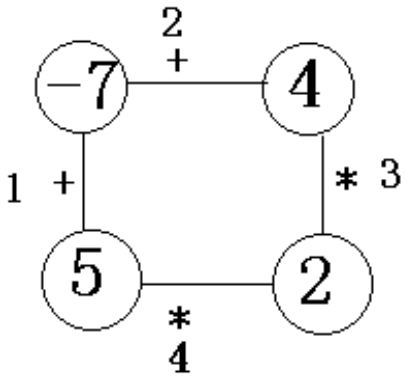


图 1 一个多边形的图形表示

首次移动(first move)，允许将某条边删除；

接下来的每次顺序移动(subsequent moves)，包括下面步骤：

1、 选出一条边 E，以及由 E 联接的顶点 V_1 和 V_2；

2、 用一个新的顶点，取代边 E 及其所联接的两个顶点 V_1 和 V_2。新顶点要赋予新的值，这个值是对 V_1 和 V_2，做由 E 所指定的运算，所得到的结果。

所有边都被删除后，只剩下一个顶点，游戏结束。游戏的得分就是该顶点的

数值。

游戏实例：略。

任务：

编写一个程序，对于任意给定的多边形，计算可能的最高得分，并且列举出所有的可以导致最高得分的被首次移动的边。

输入数据：

文件 POLYGON.IN 给出的是，由 N 个顶点构成的多边形。文件包括 2 行：

第一行记录的是数值 N；

第二行包含所有边(1，...，N)分别被赋予的符号，以及嵌入到两条边之间的顶点的数值(第一个整数值对应于与 1 号、2 号边同时相连的顶点；第二个整数对应于与 2 号、3 号边同时相连的顶点；...；等等。最后一个数值对应于与 N 号、1 号边同时相连的顶点)，符号和数值之间由一个空格分隔。边的符号有 2 种：字母 t(对应于+)，字母 x(对应于*)。

输入实例：

4

t -7 t 4 x 2 x 5

这个输入文件对应于图 1 所示的多边形。第二行的第一个字符是 1 号边的符号。

输出数据：

在文件 POLYGON.OUT 的第一行，你的程序必须输出在输入文件指定条件下可能得到的最高得分。

有些边如果在首次移动中被删除，可以导致最高得分。在输出文件的第二行要求列举出所有这样的边，而且按照升序输出，其间用一个空格分开。

输出实例：

33

1 2

4、正则表达式匹配 (ACM'97 亚洲赛区/上海赛题)

ACM International Collegiate Programming Contest
Asia Regional Contest Shanghai 1997

Problem A Pattern Matching Using Regular Expression

Input file: regular.in

A regular expression is a string which contains some normal characters and some meta characters. The meta characters include,

.

means any character

[c1-c2]

means any character between c1 and c2 (c1 and c2 two characters)

[^c1-c2]

means any character not between c1 and c2 (c1 and c2 are two characters)

*
 means the character before it can occur any times
+
 means the character before it can occur any times but at least one time
\
 means any character follow should be treated as normal character

You are to write a program to find the leftmost substring of a given string, so that the substring can match a given regular expression. If there are many substrings of the given string can match the regular expression, and the left positions of these substrings are same, we prefer the longest one.

Input

Every two lines of the input is a pattern-matching problem. The first line is a regular expression, and the second line is the string to be matched. Any line will be no more than 80 character. A line with only an "end" will terminate the input.

Output

For each matching problem, you should give an answer in one line. This line contains the string to be matched, but the leftmost substring that can match the regular expression should be bracketed. If no substring matches the regular expression, print the input string.

Sample Input

```
.*
asdf
f.*d.
sefdfsde
[0-9]+
asd345dsf
[^\*-\*]
**asdf**fasd
b[a-z]*r[s-u]*
abcdefghijklmnopqrstuvwxyz
[T-F]
dfkgjf
end
```

Output for the Sample Input

```
(asdf)
se(fdfsde)
```

asd(345)dsf
(a)sdffasd
a(bcdefghijklmnopqrstu)vwxyz
dfkgjf

中文：

ACM 国际大学生程序设计竞赛 亚洲区竞赛 上海 1997

问题 A 正则表达式匹配

输入文件：regular.in

正则表达式是一个包含一般字符与广义字符的字符串。广义符包括：

- . 代表任何字符
- [c1-c2] 代表任何位于 c1 与 c2 之间的字符 (c1 和 c2 是两个字符)
- [^c1-c2] 代表任何不位于 c1 与 c2 之间的字符 (c1 和 c2 是两个字符)
- * 代表它之前的字符可以出现任意多次，也可以不出现
- + 代表它之前的字符可以出现任意多次但至少出现一次
- \ 代表它之后的任何一个字符应看作一般字符

你要写一个程序找出所给串中最左边的能够匹配所给正则表达式的子串。如果有多个子串能匹配该正则表达式而且它们左边开始位置相同，我们要最长的。

输入

输入文件中每两行是一个正则表达式匹配问题。前一行是一个正则表达式，后一行是待匹配的串。每行长度皆不超过 80 个字符。一个仅含“end”的串表示输入文件的结束。

输出

每一个匹配问题，你应当在一行中给出你的答案。这一行包含了待匹配的串但最左边的能匹配该正则表达式的子串应当用括号括起来。如果没有子串匹配该正则表达式，原样输出待匹配的串即可。

样例输入

.*
asdf
f.*d.

sefdfsde
[0-9]+
asd345dsf
[^*-*]
asdffasd
b[a-z]*r[s-u]*
abcdefghijklmnopqrstuvwxyz
[T-F]
dfkgjf
end

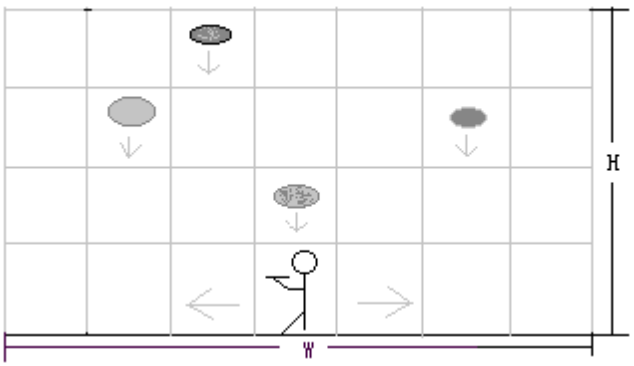
样例输出

(asdf)
se(fdfsde)
asd(345)dsf
(a)sdffasd
a(bcdefghijklmnopqrstu)vwxyz
dfkgjf

5、免费馅饼 (NOI'98)

SERKOI 最新推出了一种叫做"免费馅饼"的游戏。

游戏在一个舞台上进行。舞台的宽度为 W 格，天幕的高度为 H 格，游戏者占一格。开始时游戏者站在舞台的正中央，手里拿着一个托盘。下图为天幕的高度为 4 格时某一个时刻游戏者接馅饼的情景。



游戏开始后，从舞台天幕顶端的格子中不断出现馅饼并垂直下落。游戏者左右移动去接馅饼。游戏者每秒可以向左或向右移动一格或两格，也可以站在原地不动。

馅饼有很多种，游戏者事先根据自己的口味，对各种馅饼依次打了分。同时在 8-308 电脑的遥控下，各种馅饼下落的速度也是不一样的，下落速度以格/秒

为单位。

当馅饼在某一秒末恰好到达游戏者所在的格子中，游戏者就收集到了这块馅饼。

写一个程序，帮助我们的游戏者收集馅饼，使得所收集馅饼的分数之和最大。

输入

输入文件的第一行是用空格隔开的两个正整数，分别给出了舞台的宽度 W（1 到 99 之间的奇数）和高度 H（1 到 100 之间的整数）。

接下来依馅饼的初始下落时间顺序给出了所有馅饼的信息。每一行给出了一块馅饼的信息。由四个正整数组成，分别表示了馅饼的初始下落时刻（0 到 1000 秒）、水平位置、下落速度（1 到 100）以及分值。游戏开始时刻为 0。从 1 开始自左向右依次对水平方向的每格编号。

输入文件中同一行相邻两项之间用一个或多个空格隔开。

输出

输出文件的第一行给出了一个正整数，表示你的程序所收集的最大分数之和。其后的每一行依时间顺序给出了游戏者每秒的决策。输出 0 表示原地不动、1 或 2 表示向右移动一步或两步、-1 或-2 表示向左移动一步或两步。输出应持续到游戏者收集完他要收集的最后一块馅饼为止。

样例输入

```
3 3
0 1 2 5
0 2 1 3
1 2 1 3
1 3 1 4
```

样例输出

```
12
-1
1
1
```

搜索方法中的剪枝优化

南开中学 齐鑫

【关键字】搜索、优化、剪枝

【摘要】本文讨论了搜索方法中最常见的一种优化技巧——剪枝而且主要以剪枝判断方法的设计为核心。文章首先借助搜索树，形象的阐明了什么是剪枝；然后分析了设计剪枝判断方法的三个原则：正确、准确、高效，本文将常见的设计剪枝判断的思路分成可行性剪枝和最优性剪枝两大类，并结合上述三个原则分别以一道竞赛题为例作了说明；文章最后对剪枝方法作了一些总结。

一、 引子

搜索是人工智能中的一种基本方法，也是信息学竞赛选手所必须熟练掌握的一种方法。我们在建立一个搜索算法的时候，首要的问题不外乎两个：

1. 建立算法结构。

2. 选择适当的数据结构。

然而众所周知的是，搜索方法的时间复杂度大多是指数级的，简单的不加优化的搜索，其时间效率往往低的不能忍受，更是难以应付信息学竞赛严格的运行时间限制。

本文所讨论的主要内容就是在建立算法的结构之后，对程序进行优化的一种基本方法——剪枝。

首先应当明确的是，“剪枝”的含义是什么。我们知道，搜索的进程可以看作是从树根出发，遍历一棵倒置的树——搜索树的过程。而所谓剪枝，顾名思义，就是通过某种判断，避免一些不必要的遍历过程，形象的说，就是剪去了搜索树中的某些“枝条”，故称剪枝。

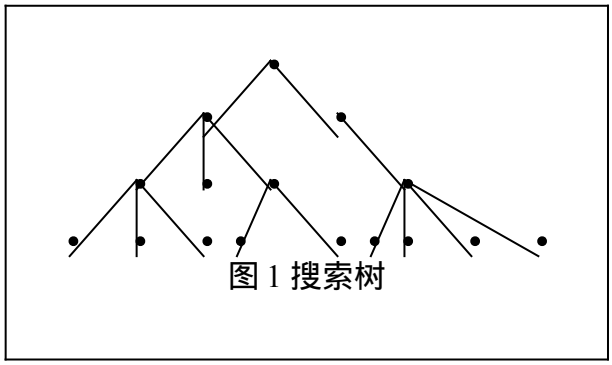


图 1 搜索树

我们在编写搜索程序的时候，一般都要考虑到剪枝。显而易见，应用剪枝优化的核心问题是设计剪枝判断方法，即确定哪些枝条应当舍弃，哪些枝条应当保留的方法。设计出好的剪枝判断方法，往往能够使程序的运行时间大大缩短；否则，也可能适得其反。那么，我们就应当首先分析一下设计剪枝判断方法的时候，需要遵循的一些原则。

二、 剪枝的原则

原则之一：正确性。

我们知道，剪枝方法之所以能够优化程序的执行效率，正如前文所述，是因为它能够“剪去”搜索树中的一些“枝条”。然而，如果在剪枝的时候，将“长有”我们所需要的解的枝条也剪掉了，那么，一切优化也就都失去了意义。所以，对剪枝的第一个要求就是正确性，即必须保证不丢失正确的结果，这是剪枝优化的前提。

为了满足这个原则，我们就应当利用“必要条件”来进行剪枝判断。也就是说，通过解所必须具备的特征、必须满足的条件等方面来考察待判断的枝条能否被剪枝。这样，就可以保证所剪掉的枝条一定不是正解所在的枝条。当然，由必要条件的定义，我们知道，没有被剪枝不意味着一定可以得到正解（否则，也就不必搜索了）。

原则之二：准确性。

在保证了正确性的基础上，对剪枝判断的第二个要求就是准确性，即能够尽可能多的剪去不能通向正解的枝条。剪枝方法只有在具有了较高的准确性的时候，才能真正收到优化的效果。因此，准确性可以说是剪枝优化的生命。

当然，为了提高剪枝判断的准确性，我们就必须对题目的特点进行全面而细致的分析，力求发现题目的本质，从而设计出优秀的剪枝判断方案。

原则之三：高效性。

一般说来，设计好剪枝判断方法之后，我们对搜索树的每个枝条都要执行一次判断操作。然而，由于是利用出解的“必要条件”进行判断，所以，必然有很多不含正解的枝条没有被剪枝。这些情况下的剪枝判断操作，对于程序的效率的提高无疑是具有副作用的。为了尽量减少剪枝判断的副作用，我们除了要下功夫改善判断的准确性外，经常还需要提高判断操作本身的时间效率。

然而这就带来了一个矛盾：我们为了加强优化的效果，就必须提高剪枝判断的准确性，因此，常常不得不提高判断操作的复杂度，也就同时降低了剪枝判断的时间效率；但是，如果剪枝判断的时间

消耗过多，就有可能减小、甚至完全抵消提高判断准确性所能带来的优化效果，这恐怕也是得不偿失。很多情况下，能否较好的解决这个矛盾，往往成为搜索算法优化的关键。

综上所述，我们可以把剪枝优化的主要原则归结为六个字：正确、准确、高效。

当然，我们在应用剪枝优化的时候，仅有上述的原则是不够的，还需要具体研究一些设计剪枝判断方法的思路。我们可以把常用的剪枝判断大致分成以下两类：

1. 可行性剪枝。
2. 最优性剪枝（上下界剪枝）。

下面，我们就结合上述的三个原则，分别对这两种剪枝判断方法进行一些讨论。

三、可行性剪枝

我们已经知道，搜索过程可以看作是对一棵树的遍历。在很多情况下，并不是搜索树中的所有枝条都能通向我们需要的结果，很多的枝条实际上只是一些死胡同。如果我们能够在刚刚进入这样的死胡同的时候，就能够判断出来并立即剪枝，程序的效率往往会得到提高。而所谓可行性剪枝，正是基于这样一种考虑。

下面我们举一个例子——Betsy 的旅行(USACO)。

题目简述：一个正方形的小镇被分成 N^2 个小方格，Betsy 要从左上角的方格到达左下角的方格，并且经过每个方格恰好一次。编程对于给定的 N，计算出 Betsy 能采用的所有的旅行路线的数目。

我们用深度优先的回溯方法来解决这个问题：Betsy 从左上角出发，每一步可以从一个格子移动到相邻的没有到过的格子中，遇到死胡同则回溯，当移动了 N^2-1 步并达到左下角时，即得到了一条新的路径，继续回溯搜索，直至遍历完所有道路。

但是，仅仅依照上述算法框架编程，时间效率极低，对 N=6 的情况无法很好的解决，所以，优化势在必行。对本题优化的关键就在于当搜索到某一个步骤时，能够提前判断出在后面的搜索过程中是否一定会遇到死胡同，而可行性剪枝正可以在这里派上用场。

我们首先从“必要条件”，即合法的解所应当具备的特征的角度分析剪枝的方法，主要有两个方向：

1. 对于一条合法的路径，除出发点和目标格子外，每一个中间格子都必然有“一进一出”的过程。所以在搜索过程中，必须保证每个尚未经过的格子都与至少两个尚未经过的格子相邻（除非当时 Betsy 就在它旁边）。这里，我们是从微观的角度分析问题。

2. 在第一个条件的基础上，我们还可以从宏观的角度分析，进一步提高剪枝判断的准确性。显然，在一个合法的移动方案的任何时刻，都不可能有孤立的区域存在。虽然孤立区域中的每一个格子也可能都有至少两个相邻的空的格子，但它们作为一个整体，Betsy 已经不能达到。我们也应当及时判断出这种情况，并避免之。

以上两个剪枝判断条件都是正确的，其准确度也比较高。但是，仅仅满足这两点还不够，剪枝判断的操作过程还必须力求高效。假如我们在每次剪枝判断时，都简单的对 N^2 个格子进行一遍扫描，其效率的低下可想而知。因此，我们必须尽可能的简化判断的过程。

实际上，由于 Betsy 的每一次移动，只会影响到附近的格子，所以每次执行剪枝判断时，应当只对她附近的格子进行检查：

对于第一个剪枝条件，我们可以设一个整型标志数组，分别保存与每个格子相邻的没被经过的格子的数目，Betsy 每次移动到一个新位置，都只会使与之相邻的至多 4 个格子的标志值发生变化，只要检查它们的标志值即可。

而对于第二个剪枝条件，处理就稍稍麻烦一些。但我们仍然可以使用局部分析的方法，即只通过对 Betsy 附近的格子进行判断，就确定是否应当剪枝，下图简要说明了剪枝的原理：

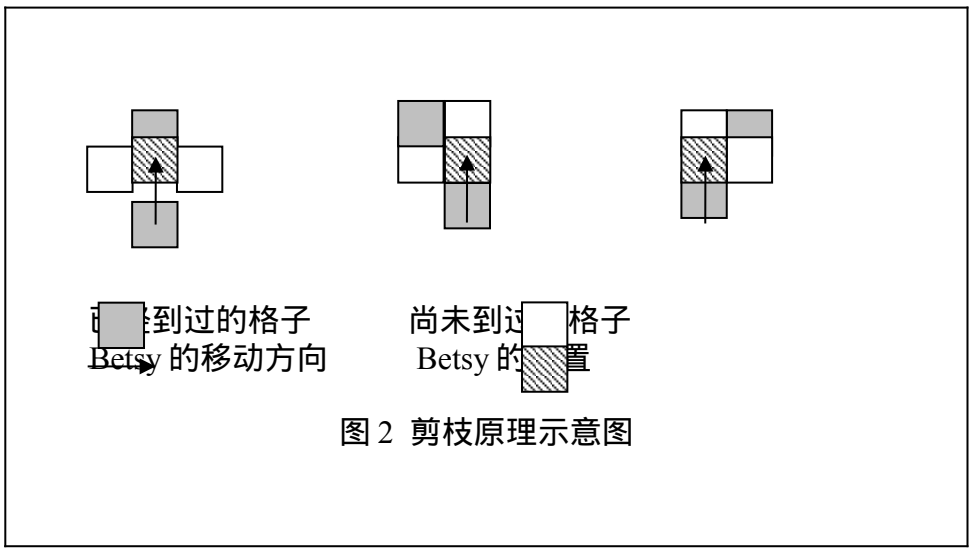


图 2 剪枝原理示意图

上图给出了可以剪枝的三种情况。由于 Betsy 到过的所有格子都一定是四连通的，所以每种情况下的两个白色的格子之间必定是不连通的，它们当中必然至少有一个是属于某个孤立区域的，都一定可以剪枝。

经过上述的优化，程序的时间效率有了很大的提高（参见附录）。

一般说来，可行性剪枝多用于路径搜索类的问题。除本例外，如 Prime Circle (ACM Asian Regional 96)等问题，也都可以使用这种剪枝方法。

在应用可行性剪枝的时候，首先要多角度全面分析问题的特点（本题就是从微观和宏观两个角度设计剪枝方法），找到尽可能多的可以剪枝的情况；同时，还必须注意提高剪枝的时间效率，所以我们使用了“局部判断”的方法，特别是在处理第二个剪枝条件时，更是通过局部判断来体现整体性质（是否有孤立区域），这一技巧不仅在设计剪枝方法的时候能够发挥作用，在其他方面也有着极为广泛的应用。

三、 最优性剪枝

在我们平时遇到的问题中，有一大类是所谓最优化问题，即所要求的结果是最优解。如果我们使用搜索方法来解决这类问题，那么最优性剪枝是一定要考虑到的。

为了表述的统一，首先要作一些说明：我们知道，解的优劣一般是通过一个评价函数来评判的。这里定义一个抽象的评价函数——“优度”，它的值越大，对应的解也就越优（对于具体的问题，我们可以认为“优度”代表正的收益或负的代价等）。

然后，我们再来回顾一下搜索最优解的过程：一般情况下，我们需要保存一个“当前最优解”，实际上就是保存解的优度的一个下界。在遍历到搜索树的叶子节点的时候，我们就能得到一个新的解，当然也就得到了它的评价函数值，与保存的优度的下界作比较，如果新解的优度值更大，则这个优度值就成为新的下界。搜索结束后，所保存的解就是最优解。

那么，最优性剪枝又是如何进行的呢？当我们处在搜索树的枝条上时，可以通过某种方法估算出该枝条上的所有解的评价函数的上界，即所谓估价函数 h。显然，h 大于当前保存的优度的下界，是该枝条上存在最优解的必要条件，否则就一定可以剪枝。所以，最优性剪枝也可以称为“上下界剪枝”。同时，我们也可以看到，最优性剪枝的核心问题就是估价函数的建立。

下面举一个应用最优性剪枝的典型例题——最少乘法次数。

题目简述：由 x 开始，通过最少的乘法次数得出 x^n，其中 n 为输入数据。（参见参考书目 1）

因为两式相乘等于方幂相加，所以本题可以等效的表示为：构造一个数列 $\{a_i\}$，满足

$$a_i=\begin{cases}1 & (i=1)\\ a_j+a_k\ (1<=j,k<i) & (i>1)\end{cases}$$

要求 $a_t=n$，并且使 t 最小。

我们选择回溯法作为本程序的主体结构：当搜索到第 i 层时，a_i 的取值范围在 $a_{i-1}+1$ 到 $a_{i-1}*2$ 之间，为了尽快接近目标 n，应当从 $a_{i-1}*2$ 开始从大到小为 a_i 取值，当然，选取的数都不能大于 n。当搜索到 n 出现时，就得到了一个解，与当前保存的解比较取优。最终搜索结束之后，即得到最终的最优解。

如果按上述结构直接进行搜索，效率显然很低。因此，我们可以考虑使用最优性剪枝进行优化：

优化之一：当然首先要建立估价函数。由于使数列中的最大数加倍无疑是最快的增长方式，所以一种最简单的估价函数为（设数列中当前的最大者是a_i，即当前搜索深度为i）：

$$h=\left\lceil \log_2 \frac{n}{a_i} \right\rceil$$

然而，这个估价函数的准确性太差，特别是当a_i大于$\frac{n}{2}$时，h只能等于1，根本不能发挥剪枝的作用。因此，无论是深度优先的回溯法还是宽度优先的A*搜索方法，只要还使用这个估价函数，其优化效果都比较有限。

下面，我们再讨论一些进一步的优化手段——

优化之二：着眼于估价函数的"生命"——准确性。我们可以利用加法在奇偶性上的特点的推广，使估价函数在某些情况下的准确性得到一定的提高（具体改进请参见附录）。

优化之三：我们新建立的这个估价函数虽然准确性稍有提高，但时间复杂度也相应提高。为了提高剪枝的效率，我们可以使用一种"逐步细化"的技巧，即先使用一次原先的估价函数（快而不准）进行"过滤"，再使用新的估价函数（稍准但较慢）减少"漏网之鱼"，以求准确性和运行效率的平衡。

优化之四：我们可以在搜索之前将n分解质因数，对每个质因数先使用上述搜索方法求解（如某个质因数仍太大，还可以将其减1，再分解），即相当于将构造数列的过程，划分成若干阶段处理。这样得到的结果虽然不一定是全局的最优解，却可以作为初始设定的优度下界，使后面的全局搜索少走很多弯路。事实上，该估计方法相当准确，在很多情况下都能直接得到最优解，使程序效率得到极大提高。

优化之五：当数列中的当前最大数a_i超过$\frac{n}{2}$后，原估价函数就难以发挥作用了。但是，此后的最优方案，实际上就等价于从a_1到a_i这i个数中，选出尽可能少的数（可重复），使它们的和等于n。这个问题已经可以使用动态规划方法来解决。这样得到的"估价函数"不但完全准确，甚至直接就可以代替后面的搜索过程。这里也体现出了搜索和动态规划的优势互补。

这道题中所体现的最优性剪枝技巧是很多的，它们的优化效果也是非常显著的（优化效果的分析请参见附录）。

由本题，并结合以前的经验，我们可以简单的总结一些在应用

最优性剪枝时需注意的问题：

1. 估价函数的设计当然首先得满足正确的原则，即使用“必要条件”来剪枝。在此基础上，就要注意提高估价的准确性。本题的优化之二就是为了这个目的。

2. 与其他剪枝判断操作一样，最优性剪枝的估价函数在提高准确性的同时，也必须注意使计算过程尽量高效的原则。由于剪枝判断在运行时执行极为频繁，所以对其算法进行精雕细琢是相当必要的，有时甚至还可以使用一些非常手法，如前述的“逐步细化”技巧，就是一种寻求时间效率和精确度相平衡的方法。

3. 在使用最优性剪枝时，一个好的初始“优度”下界往往是非常重要的。在搜索开始之前，我们可以使用某种高效方法（如贪心法等）求出一个较优解，作为初始下界，经常可以剪去大量明显不可能的枝条。本题使用划分阶段的搜索方法进行初始定界，就带来了大幅度的优化。

4. 本节所举的是在深度优先搜索中应用最优性剪枝的例子。当然，在广度优先搜索中，也是可以使用最优性剪枝的，也就是我们常说的分枝定界方法。本题也可以使用分枝定界结合 A*算法的方法来解决（用改进的估价函数作为优度上界估计，即 h*函数；前述的动态规划方法可以作为优度下界估计），但时间效率和空间效率都要差一些。不过，在有些问题中，分枝定界和 A*算法的结合以其较好的稳定性，还是有用武之地的。

四、 总结

搜索方法，因其在时间效率方面“先天不足”，所以人们才有针对性的研究出了很多优化技巧。本文所论述的“剪枝”就是最常见的优化方法之一，几乎可以说，只要想使用搜索算法来解决问题，就必须考虑到剪枝优化。

另外需要说明的是，本文所介绍的可行性和最优性两种剪枝判断，其分类只是依据其不同的应用对象，而前面阐述的剪枝方法的三个原则——正确、准确和高效，才是贯穿于始终的灵魂。

本文还介绍了一些剪枝中的常用技巧，如“局部分析”、“逐步细化”等。恰当的使用它们，也能够使程序的效率（主要是时间效率）得到相当的提高。

但是，剪枝方法无论多么巧妙，都不能从本质上降低搜索算法的时间复杂度，这是不争的事实。因此，我们在动手设计一个搜索算法之前，不妨先考虑一下是否存在着更为有效的方法。

而且，在信息学竞赛中，还有一个编程复杂度的问题。我们对一个搜索算法使用了很多优化技巧，虽然可能使程序的时间效率得到一定的提高，但却往往要消耗大量的编程时间，很容易造成“拣了芝麻，丢了西瓜”的结果。

总之，我们在实际设计程序的过程中，不能钻牛角尖，而更应当充分发挥思维的灵活性，坚持“具体问题具体分析”的思想方法。

【参考书目】

1. 《青少年国际信息学（计算机）奥林匹克竞赛指导——人工智能搜索与程序设计》刘福生 王建德 编著
2. 《Informatics Olympic》

【附录】

一、“Betsy 的旅行”的测试情况：（单位：秒）

N 值	路径数	程序运行时间			
		无剪枝	第一种剪枝	第二种剪枝	两种剪枝
2	1	<0.01	<0.01	<0.01	<0.01
3	2	<0.01	<0.01	<0.01	<0.01
4	8	<0.01	<0.01	<0.01	<0.01
5	86	0.17	<0.01	<0.01	<0.01
6	1770	41.25	0.44	0.33	0.11
7	88418	Very long	28.06	26.86	8.51

（测试环境：Pentium II 266MHz/64MB）

可见，使用前述的可行性剪枝判断方法，能够使程序的时间效率得到相当大的提高。

二、“最少乘法次数”的估价函数的改进：

最初的估价函数的设计思路实际上是在当前数列 $a_1,\cdots,a_i$ 的基础上理想化的构造 $2a_i,4a_i,\cdots,2^p a_i$，当 $2^p a_i < n < 2^{p+1} a_i$ 时，原估价方法认为只需再进行一次加法，即找一个数与 $2^p a_i$ 相加就够了。

然而，这只是保证了能够得到大于等于 n 的数，并不一定恰好得到 n。我们可以作如下分析：

首先，任何一个自然数 i 都可以表示成 $2^k(2m+1)\quad (k,m\in Z^-)$ 的形式，我们可以设 $k(i)$ 表示一个自然数 i 所对应的 k 值。显然，对于任何两个自然数 a 和 b，都有 $k(a+b)\geq\min\{k(a),k(b)\}$（我们由此可以很容易的联想到“奇+奇=偶，偶+偶=偶，奇+偶=奇”的性质）。

然后，我们再研究前述估价时所构造的数列：

$a_1,a_2,\cdots,a_i,2a_i,4a_i,\cdots,2^p a_i$　（其中，$2^p a_i < n < 2^{p+1} a_i$）

在应用新的剪枝判断之前，我们应当先检验 $\lceil \log_2(n/(a_i+a_{i-1}))\rceil \geq p$，这个条件可以保证只有构造上述数列才是最优的。

若存在自然数 $j\ (1\leq j\leq p)$，使得 $k(2^j a_i) > k(n)$，由 $k(a+b)\geq\min\{k(a),k(b)\}$，

则有 $k(2^t a_i + 2^p a_i)\geq k(2^t a_i)\geq k(2^j a_i) > k(n)\qquad (j\leq t\leq p)$

$\therefore 2^t a_i + 2^p a_i \neq n$　（$k(a)=k(b)$ 是 $a=b$ 的**必要条件**）

即 $2^j a_i,\cdots,2^p a_i$ 中的任何一个数与 $2^p a_i$ 相加都不可能得到 n 。

所以，如果 $n - 2^p a_i > 2^{j-1} a_i$，则在得到 $2^p a_i$ 后，至少还需要再进行两次加法才有可能得到 n 。

虽然上述改进可以使估价函数在某些情况下的准确性略有提高，但是其本身的时间效率却比较差，这是因为新的估价函数不但包括了原先的估价函数（构造数列），而且还增加了诸如求 $k(a_i)$ 这样的操作。因此，尽管新的估价函数只是原估价函数的改进，而不象在“Betsy 的旅行”中那样是互相补充的两个方面，但在实际的程序中，仍然要连续调用两个估价函数，即使用前文所述的“逐步细化”技巧。

最后需要说明的是，本文所提及的估价函数中所有使用对数运算的部分在实际的程序中都必须改用循环累加的方式来实现。这是因为实数函数在计算机内部是通过级数展开式计算的，实际上也是循环结构，而且更为复杂低效。

当然，实际的程序中，也还有其他的一些细节上的优化。

三、“最少乘法次数”的测试情况及分析：

部分测试数据的运行时间：（单位：秒）

N 值	乘法次数	运行时间						
		程序 1	程序 2	程序 3	程序 4	程序 5	程序 6	程序 7
127	10	0.22	0.22	0.05	0.06	0.06	<0.01	0.16
191	11	3.52	3.90	0.83	0.55	0.55	0.27	1.75
382	11	13.02	13.29	1.05	0.94	0.66	0.33	2.69
511	12	Very long	Very long	0.82	5.66	0.33	0.27	1.81
635	13	46.96	40.37	20.65	19.61	10.06	8.13	Overflow
719	13	45.59	39.11	45.64	13.45	39.11	6.64	27.35
1002	13	11.81	10.65	2.74	3.46	1.10	0.99	6.26
1023	13	Very long	Very long	2.66	Very long	1.04	0.88	6.15
1357	13	9.51	6.98	5.93	5.44	3.40	2.74	13.35
1894	14	44.87	37.24	13.89	15.49	6.37	4.61	23.68

（测试环境：Pentium Pro 233MHz/64MB）

当 n 在 1000 以内的时候，程序 6 的运行时间都不超过 10 秒，其中只有十几个测试数据的运行时间在 5 秒以上。

程序说明：

	程序特征要点

程序编号	简单的剪枝判断	改进的剪枝判断	初始定界(分段搜索)	动态规划的结合使用	A*算法结合分枝定界
1	√				
2	√	√			
3	√		√		
4	√			√	
5	√	√	√		
6	√	√	√	√	
7					√

测试结果分析：

首先，我们比较一下三个主要的优化点——改进的剪枝判断（逐步细化）、初始定界和结合动态规划——单独应用的优化效果：

1. 单纯加入改进的剪枝判断方法（程序 2），程序时间效率的提高不大，这主要是因为新的估价函数的准确性只是稍有提高，而其本身的时间效率却比较低，这在一定程度上抵消了估价准确性提高所能带来的优化效果。
2. 本题的初始定界方法（程序 3），由于准确性非常高（经常能直接得到最优解），所以能够使程序的效率得到比较明显的改善，在前四个程序中，只有它能够在一分钟以内完成表中所列出的每个测试数据，特别是 n=1023 时，效果尤为显著。但是，如果初始定界的准确性稍有下降，则程序的优化效果就会极不明显（如 n=719）。
3. 加入动态规划后的程序（程序 4），实际上使得搜索过程主要集中在[n/2]以下，在相当程度上避免了对[n/2]到 n 的估价函数几乎没有作用的部分的搜索，所以程序 4 几乎在每个测试点上都比较明显的优于程序 1。

然后，我们再来看看程序 5，由于使用了初始定界，使搜索前就已经有了比较准确的优度下界，所以新的准确性较高的估价函数发挥作用的概率就比较高，因此优化效果相当明显。与程序 3 一样，在初始定界不甚准确的时候（n=719），优化效果大打折扣。

这 7 个程序中最好的当然是程序 6，由于综合了各种优化方法，使它们优势互补，所以获得了最稳定的优化效果。

最后，我们还可以看到，在本题的特定情况下，经过充分优化的深度优先的回溯法搜索无论是在时间上，还是在空间上都明显优于 A*算法（程序 7 使用保护模式编译，除了 A*算法外，还结合使用了分枝定界的方法，否则时空效率

更差）。

【程序】

一、“Betsy 的旅行”的程序（使用两种剪枝）：

```
{$A+,B-,D-,E+,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 65520,0,655360}
program Betsy;{IOI'99 集训队 论文例题 1：Betsy 的旅行}
{说明:
   1.为了便于测试计时，本程序采用命令行输入的方式。
   2.为了处理的方便，程序中在地图的最外层补了一圈标志格。
   3.程序中，将逻辑上相对独立的程序段用空行分开。
}
const
 max=7;{本程序所能处理的最大的数据规模}
 delta:array[1..4,1..2]of shortint=((-1,0),(0,1),(1,0),(0,-1));{方向增量}

var
 map:array[0..max+1,0..max+1]of integer;
 {用于标记 Betsy 的移动路线的地图：
    没有到过的位置标记-1，
    最外层的标志格标记 0，
    其它格子标记 Betsy 到达该格子时的移动步数
 }
 left:array[0..max+1,0..max+1]of shortint;
 {标志数组：记录每个格子相邻的四个格子中尚未被 Betsy 经过的格子的数目}
 n:1..max;{地图边长}
 n2:integer;{N*N}
 s:longint;{累加器,记录 Betsy 的移动路线的总数}

procedure init;{读入数据，初始化}
 var
  i:integer;{循环变量}
  temp:integer;{临时变量}
 begin
  val(paramstr(1),n,temp);{从命令行读入 n}
  n2:=n*n;
```

```
  fillchar(map,sizeof(map),255);
  for i:=0 to n+1 do begin
   map[0,i]:=0;
   map[i,0]:=0;
   map[n+1,i]:=0;
   map[i,n+1]:=0;
  end;
  map[1,1]:=1;
  {以上程序段为对地图的初始化}

  fillchar(left,sizeof(left),4);
  for i:=2 to n-1 do begin
   left[1,i]:=3;left[n,i]:=3;
   left[i,1]:=3;left[i,n]:=3;
  end;
  left[1,1]:=2;left[1,n]:=2;left[n,1]:=3;left[n,n]:=2;
  dec(left[1,2]);dec(left[2,1]);
  {以上程序段是对标志数组的初始化}

  s:=0;{累加器清零}
 end;

procedure change(x,y,dt:integer);{给(x,y)相邻的四个格子的 left 标志值加上 dt}
{设标志时，dt 取-1；回溯时，dt 取 1}
 var
  k:integer;{循环变量}
  a,b:integer;{临时坐标变量}
 begin
  for k:=1 to 4 do begin
   a:=x+delta[k,1];b:=y+delta[k,2];
   inc(left[a,b],dt);
  end;
 end;

procedure expand(step,x,y:integer);{搜索主过程：搜索第 step 步，扩展出发位置(x,y)}
 var
```

```
  nx,ny:integer;{Betsy 的新位置}
  dir:byte;{Betsy 的移动方向}

 function able(x,y:integer):boolean;{判断 Betsy 是否可以进入(x,y)}
  begin
   able:=not((map[x,y]<>-1)or((step<>n2)and(x=n)and(y=1)));
  end;

 function cut1:boolean;{剪枝判断 1}
  var
   i:integer;{循环变量}
   a,b:integer;{临时坐标变量}
  begin
   for i:=1 to 4 do begin
    a:=x+delta[i,1];b:=y+delta[i,2];
    if (map[a,b]=-1)and((a<>nx)or(b<>ny))and(left[a,b]<=1) then begin
     cut1:=true;
     exit;
    end;
   end;
   cut1:=false;
  end;

 function cut2:boolean;{剪枝判断 2}
  var
   d1,d2:integer;{相对于当前移动方向的"左右"两个方向}
   fx,fy:integer;{Betsy 由当前位置，沿原方向，再向前移动一步的位置}
  begin
   if (dir=2)or(dir=4) then begin
    d1:=1;d2:=3;
   end
   else begin
    d1:=2;d2:=4;
   end;
   fx:=nx+delta[dir,1];fy:=ny+delta[dir,2];
```

```
    if (map[fx,fy]<>-1)and(map[nx+delta[d1,1],ny+delta[d1,2]]=-1)
    and(map[nx+delta[d2,1],ny+delta[d2,2]]=-1) then begin
      cut2:=true;
      exit;
    end;
    if (map[fx+delta[d1,1],fy+delta[d1,2]]<>-1)and(map[nx+delta[d1,1],ny+delta[d1,2]]=-1)
    and(map[fx,fy]=-1) then begin
      cut2:=true;
      exit;
    end;
    if (map[fx+delta[d2,1],fy+delta[d2,2]]<>-1)and(map[nx+delta[d2,1],ny+delta[d2,2]]=-1)
    and(map[fx,fy]=-1) then begin
      cut2:=true;
      exit;
    end;
    {以上程序段中的三个条件判断分别对应论文剪枝原理图中所列的三种情况}

    cut2:=false;
  end;

 begin{Expand}
  if (step>n2)and(x=n)and(y=1) then begin{搜索到最底层,累加器加 1}
    inc(s);
    exit;
  end;
  for dir:=1 to 4 do begin{循环尝试 4 个移动方向}
    nx:=x+delta[dir,1];ny:=y+delta[dir,2];
    if able(nx,ny) then begin
      if cut1 then continue;{调用剪枝判断 1}
      if cut2 then continue;{调用剪枝判断 2}
      change(nx,ny,-1);
      map[nx,ny]:=step;
      expand(step+1,nx,ny);{递归调用下一层搜索}
      map[nx,ny]:=-1;
      change(nx,ny,1);
    end;
```

```
  end;
 end;

begin{主程序}
 init;{初始化}
 expand(2,1,1);{调用搜索过程}
 writeln('The number of tours is ',s);{输出结果}
end.
```

二、“最少乘法次数”的程序（即附录中的程序６）：

```
{$A+,B-,D-,E+,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 65520,0,655360}
program LeastMultiply;{IOI'99 集训队 论文例题 2：最少乘法次数}
{说明:
   1.为了测试计时的方便,本程序从命令行读入 n 值。
   2.程序结束后，在 output.txt 中给出执行乘法的方式，并给出总的乘法次数。
   3.在搜索过程中，由于与动态规划结合，所以在没有搜索到底层的时候，就可以得到最优解的数列长度(但此时没有得到完整的最优幂次数列)，所以在搜索结束后，程序中调用 formkeep 过程在 keep 中生成完整的构造数列。
   4.由于程序中的搜索过程生成的是最优幂次数列，而没有直接给出乘法的进行方式，所以在输出结果的过程 output 中，对其进行了转换。
      5.为了尽可能的提高程序的时间效率，程序中有几处细节上的优化，请参见程序内的注释。
      6.程序中，逻辑上相对独立的程序段用空行分开。
}

const
 max=20;{数列最大长度}
 maxr=2000;{动态规划计数数组的最大长度(输入的 n 不能超过 maxr 的 2 倍)}
 mp=100;{预处理估价时,可以直接搜索处理的数的范围上限}
 power2:array[0..12]of integer= (1,2,4,8,16,32,64,128,256,512,1024,2048,4096); {2 的方幂}

type
 atype=array[0..max]of integer;{用于记录构造的幂次数列的数组类型，0 号元素记录数列长
```

度}

```
var
 n:integer;{读入的目标数字}
 time:array[0..maxr]of integer;
 {动态规划计数数组，time[i]表示在当前构造数列的基础上，组成数 i 至少需要的加数个数}
 range:integer;{使用动态规划处理的范围：range=[n/2]}
 a:atype;{搜索中记录数列}
 kp:atype;{预处理估界时，记录结果数列的临时数组}
 keep:atype;{记录最优结果数列的数组}
 best:integer;{当前最优解}
 f:text;{输出文件}

procedure init;{初始化}
 var
  temp:integer;{临时变量}
 begin
  val(paramstr(1),n,temp);{从命令行读入 n}
  keep[0]:=1;
  keep[1]:=1;
  best:=maxint;{最优数列长度的初值}
  assign(f,'output.txt');{连接输出文件}
 end;

procedure search(n:integer;var best:integer;var keep:atype);{搜索主过程}
{搜索之前，给出的搜索目标为 n；
      在 best 中存放搜索前已经给出的优度下界；
      在 keep 中存放初始优度下界对应的最优幂次数列。
 搜索结束之后，在 best 中给出的是构造的最优幂次数列的长度,即最少乘法次数加 1；
        在 keep 中给出所构造的最优幂次数列。
}
 var
  kn:integer;{n 所含的 2 的方幂的次数}
  i:integer;{循环变量}
```

```
function getk(num:integer):integer;{求 num 所含的 2 的方幂次数,即论文中所设的 k(num)}
  var
    i:integer;{循环变量}
  begin
    i:=0;
    while not odd(num) do begin
      num:=num shr 1;
      inc(i);
    end;
    getk:=i-1;
  end;

procedure find(step:integer);{递归搜索过程}
  var
    i:integer;{循环变量}
    k:integer;{本层搜索的循环范围上限}

  function ok(num:integer):boolean;{判断数 num 能否在当前被构造出来}
  {为了提高程序的效率，这里利用了动态规划的结果}
    var
      i,j:integer;{循环变量}
    begin
      if num<=range then begin{待判断数 num 在[n/2]以内}
        ok:=(time[num]=2);{直接利用最少需要的加数是否为 2 来判断}
        exit;
      end;
      for i:=step-1 downto 1 do begin
        if a[i]+a[i]<num then break;
        if time[num-a[i]]=1 then begin
        {time[t]=1 表明数 t 在已有数列中出现过，这样可以避免使用循环判断}
          ok:=true;
          exit;
        end;
      end;
      ok:=false;
    end;
```

```
procedure evltime;{动态规划子过程}
  var
    i,j:integer;{循环变量}
    p:integer;{本次动态规划递推的上限}
  begin
    p:=k;
    if p>range then p:=range;
    for i:=a[step-1]+1 to p do begin
      time[i]:=time[i-a[step-1]]+1;{目标函数赋初值}
      for j:=step-2 downto 2 do{决策枚举}
        if time[i-a[j]]+1<time[i] then time[i]:=time[i-a[j]]+1;
    end;
  end;

function h(num:integer):byte;{最初的简单估价函数 h}
  var
    i:integer;{循环计数变量}
  begin
    i:=0;
    while num<n do begin
      num:=num shl 1;
      inc(i);
    end;
    h:=i;
  end;

function cut1:boolean;{最初的剪枝判断，直接调用估价函数 h}
  begin
    if h(a[step])+step>=best then cut1:=true else cut1:=false;
  end;

function cut2:boolean;{使用改进的估价函数的剪枝判断}
  var
    pt:integer;{k(a[step])的值，即 a[step]中所含的 2 的幂的次数}
    rest:integer;{新构造的数列中，满足 k(rest)=k(n)的数}
```

```
  i:integer;{循环计数变量}
 begin
  if h(a[step]+a[step-1])+step+1<best then begin
  {如果新构造数列的第一步选择 a[step]+a[step-1]，而不是 2*a[step]，就必然导致剪枝，
   这是新的估价函数起作用的必要条件。
  }
   cut2:=false;
   exit;
  end;

  pt:=getk(a[step]);{求 k(a[step])}
  if pt>kn then rest:=a[step-1]
  else
   if pt=kn then rest:=a[step]
   else rest:=maxint;
     {给 rest 赋初值，以防新构造的数列中的所有数的 k 值都小于 k(n)}

  i:=0;
  repeat
   a[step+i+1]:=a[step+i] shl 1;
   inc(i);
   if pt+i=kn then rest:=a[step+i];
  until a[step+i]>n;
  dec(i);
  {以上程序段为构造数列的过程}

  if (n-a[step+i]>rest)and(step+i+2>=best) then cut2:=true else cut2:=false;{剪枝判断}
 end;

begin{Find}
 if a[step-1]+a[step-1]>=n then begin
 {数列中的当前最大数已经超过[n/2]，则直接引用动态规划的结果}
  if time[n-a[step-1]]+step-1<best then begin{判断出解}
   best:=time[n-a[step-1]]+step-1;
   keep:=a;
  end;
```

```
    exit;
  end;

  k:=a[step-1]+a[step-1];{计算 a[step]的可选范围的上限}
  evltime;{调用动态规划子过程}

  inc(a[0]);
  for i:=k downto a[step-1]+1 do
    if ok(i) then begin
      if i<=range then time[i]:=1;
      a[step]:=i;
      if cut1 then break;{调用剪枝判断 1}
      if cut2 then continue;{调用剪枝判断 2}
      find(step+1);{递归调用下一层搜索}
    end;
  dec(a[0]);
end;

procedure formkeep;{生成最优数列 keep}
{由于在搜索时，如果 a[step]>[n/2]则直接引用动态规划的结果，所以最优结果数列的最后一
 部分实际上并未得到，所以需要在本过程中将其补充完整。
 这里还需要使用递归和回溯（实际上就是一个小规模的搜索），不过，由于动态规划给出的结
 果表示的是从已有数列中，选出最少的数求和而得到 n，所以对这些数是可以定序的。
}
  var
    found:boolean;{找到最优数列的标志}

  procedure check(from,left:integer);{回溯法生成最优数列的最后一部分}
    {from 表示当前层内循环的上界(用于定序)，left 表示剩余的需要通过求和而得到的数}
    var
      i:integer;{循环变量}
    begin
      if keep[0]=best then begin
        if left=0 then found:=true;{找到最优数列}
```

```
      exit;
    end;

    inc(keep[0]);
    for i:=from downto 1 do{循环枚举数列中的数}
     if keep[i]<=left then begin
       keep[keep[0]]:=keep[keep[0]-1]+keep[i];
       check(i,left-keep[i]);{调用下一层搜索，但所使用的数不能再超过keep[i](定序)}
       if found then exit;
     end;
    dec(keep[0]);
  end;

 begin{FromKeep}
  found:=false;
  check(keep[0],n-keep[keep[0]]);{调用搜索过程}
 end;

begin{Search}
 kn:=getk(n);{计算k(n)}
 time[0]:=0;
 time[1]:=1;
 a[0]:=1;
 a[1]:=1;
 range:=n div 2;
 {以上程序段为搜索前的初始化}

 find(2);{调用搜索过程}
 formkeep;{搜索结束后，在keep中生成完整的构造数列}
end;

function guess(n:integer):integer;{递归方式实现的预处理估界}
{说明：
  1.子程序guess运行结束后，返回的值即为对n估价的结果；同时，在keep数组中得到对应的数列。
  2.为了能够使估价尽可能准确，guess中同时考虑了两种分解策略，即使用了回溯结构。
```

3.由于每次递归调用guess，其最终结果都必须记入keep数组，所以每次guess运行都只操作keep数组的一部分，同时还要借助于kp,kpt等临时数组。

}

```
 var
   num:integer;{将n中的所有2的因子都提出后剩下的数}
   pfact:integer;{表示num的素因子的变量}
   best:integer;{向下调用时记录返回的估价值}
   b2:integer;{记录num中的2的因子的数目}
   s:integer;{对n的估价步数}
   i:integer;{循环变量}
   sq:integer;{num的平方根的整数部分，分解素因子时作为上界}
   s1,k2,k:integer;{临时变量}
   kpt:atype;{临时记录最优结果数列}

 begin
   num:=n;
   b2:=0;
   while not odd(num) do begin
     num:=num div 2;
     inc(b2);
     inc(keep[0]);
     keep[keep[0]]:=power2[b2];
   end;
   s:=b2+1;
   k2:=keep[0];
   {以上程序段的作用是将n中所含的2的因子全部提出，剩下的部分即num}

   if num<=mp then begin{如果num足够小(小于等于mp)，则直接调用搜索过程处理}
     best:=maxint;
     search(num,best,kp);{对n调用搜索过程，得到的数列存放在kp中}
     inc(s,best-1);
     for i:=2 to best do keep[i+keep[0]-1]:=kp[i];
     inc(keep[0],best-1);
   end
   else begin{使用估价方法处理}
     k:=keep[0];
```

```
    best:=guess(num-1);{递归调用 guess}
    keep[keep[0]+1]:=keep[keep[0]]+1;
    inc(keep[0]);
    inc(s,best);
    {以上程序段为估价的第一种策略：将 num 减 1 后，对 num-1 进行估价}

    sq:=trunc(sqrt(num));
    pfact:=3;
    while (pfact<=sq)and(num mod pfact>0) do inc(pfact,2);
    if pfact<=sq then begin
      kpt:=keep;
      keep[0]:=k2;
      s1:=b2+1;
      {以上程序段为使用第二种策略前的初始化}

      k:=keep[0];
      best:=guess(pfact);
      inc(s1,best-1);
      {以上程序段中递归调用 guess，对质因数 pfact 进行估价}

      k:=keep[0];
      best:=guess(num div pfact);
      for i:=k+1 to keep[0] do keep[i]:=pfact*keep[i];
      inc(s1,best-1);
      {以上程序段对[num/pfact]进行估价}

      if s1<s then s:=s1 else keep:=kpt;{比较两种策略结果的优劣}
    end;
    {以上程序段是估价的第二种策略：将 num 分解出一个较小的质因数后再处理。
     估价的结果存放在 s1 中，使用 kpt 临时存放第一种策略所构造的数列。
    }
  end;

  for i:=k2+1 to keep[0] do keep[i]:=keep[i]*power2[b2];
  guess:=s;{返回估价结果}
end;
```

```
procedure output;{输出结果}
  var
    i,j,k:integer;{循环变量}
  begin
    rewrite(f);
    writeln(f,'X1');
    for i:=2 to best do begin
      for j:=1 to i-1 do begin
        for k:=j to i-1 do
          if keep[j]+keep[k]=keep[i] then break;
        if keep[j]+keep[k]=keep[i] then break;
      end;
      writeln(f,'X',i,'=X',j,'*X',k);
    end;
    writeln(f,'Times of multiplies=',best-1);
    close(f);
  end;

begin{主程序}
  init;{读入数据，初始化}
  best:=guess(n);{调用预处理估界}
  search(n,best,keep);{调用搜索主过程}
  output;{输出结果}
end.
```

数学模型的建立、比较和应用

苏州中学 邵铮

关键字: 数学模型 算法 母函数

【摘要】

数学模型是解决实际问题的一种基本工具。将实际问题抽象成一个数学模型，运用数学工具进行求解，并将结果应用于具有相同特征的一类问题中，是解决问题的一种基本的途径。本文首先介绍了数学模型的一些性质，然后建立了三种不同的数学模型来求解一个问题，将三种数学模型相互比较，得出数学模型抽象性与高效性之间的关系，再将数学模型推广应用于另两个问题的求解，得出数学模型抽象性与可推广性之间的关系，最后总结全文，揭示出有关数学模型的一些普遍规律。

一、引论

实际问题往往是纷繁而复杂的，而其中的规律也是隐藏着的，要想直接用计算机来求解实际问题往往有一定的困难。计算机擅长的是解决数学问题。因此我们有必要将实际问题抽象成数学模型，然后再用计算机来对数学模型进行求解。

与实际问题相比，数学模型有以下几个性质：

1. 抽象性：数学模型是实际问题的一种抽象，它去除了实际问题中与问题的求解无关的部分，简明地体现了问题的本质。这一点是下面两个性质的基础。

2. 高效性：数学模型中各个量之间的关系更为清晰，容易从中找到规律，从而提高求解的效率。由于这一点是由数学模型的抽象性决定的，因此数学模型的抽象化程度对数学模型效率的高低有重要的影响，这一点将在第二部分中详细阐述。

3. 可推广性：数学模型可以推广到具有相同性质的一类问题中。换句话说，解决了一个数学模型就解决了一类实际问题。这里的"相同性质"是指相同的本质表面看似毫不相干的问题可能有着相同的本质。由于这一点也是由数学模型的抽象性决定的，因此数学模型的抽象化程度对数学模型的推广范围也有重要的影响，这一点将在第三部分中详细阐述。

二、数学模型的建立和比较

由于考虑问题的角度不同，面对同一个实际问题，可能建立起各种各样的数学模型。在各种数学模型中，我们要寻找的是效率高的模型。模型的效率同模型的抽象化程度有关，下面从一个实例中来分析它们之间的具体关系。

【多边形分割问题】将一个凸 n 边形用 n-3 条互不相交的对角线分割为 n-2 个三角形,求分割方案的总数.

如:n=5 时,有以下几种分割方案:

这道题可用以下几种方法来求解:

<1>.搜索法:

这种方法的思路是将各种分割方案全都列举出来。

显然，一组 n-3 条互不相交的对角线对应于一种分割方案，因此可把问题看作是求不同的对角线组的数目。

将 n 边形的 n 个顶点按顺时针方向编号为 1、2、3……n，则一条对角线可表示为一个数对(a1,a2)，a1、a2 分别表示对角线两端顶点的序号，a1<a2，a1 为对角线的始端，a2 为末端。

对角线在对角线组中的顺序是无关紧要的，因此，一个对角线组是一个集合，它的元素是对角线。

判断两条对角线是否相交是一个必须解决的问题。设两条对角线分别为(a1,a2)与(b1,b2)，若把表示对角线的数对看作开区间，那么两条对角线不相交的充要条件是两个区间有包含关系或他们的交集为空集。

于是，我们建立起解决本问题的第一个数学模型：

已知：n 的值,

一个集合由(n-3)个不同的开区间(i,j)组成,

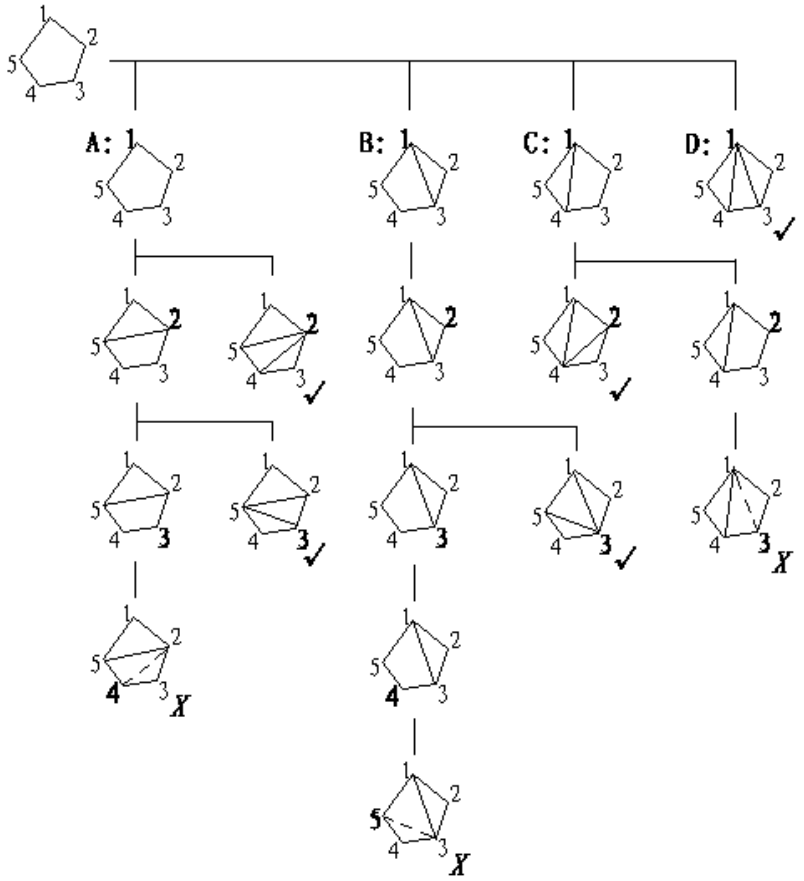
i∈{1..n-2}, j∈{i+2..n},(i≠1)或(j≠n)

同一个集合中任两个不同的开区间(i1,j1),(i2,j2)满足：

((i1,j1)∩(i2,j2)=空集)或((i1,j1)包含(i2,j2))或((i2,j2)包含(i1,j1))

求：不同的集合的个数

搜索时，先考虑以顶点 1 为始端的对角线，可以不连任何对角线(图一中 A)，也可以连(1,3)(图一中 B)，或连(1,4)(图一中 C)，或同时连(1,3)(1,4) (图一中 D)。对于每一种情况，再考虑以顶点 2 为始端的对角线，依此类推。当得到 n-3 条互不相交的对角线时，便找到了一种方案(参见图一)。



图一

在考虑以顶点 i 为始端的对角线时，有以下几条规则必须遵循：

1.与原有对角线相交的对角线不得选取。

2.当 i>=3 时，若顶点 i-1 为始端的对角线一条都未连，则对角线(i-2,i)必须是已经连的。

3.对角线的末端顶点序号必须大于 i。否则，顶点 i 将成为对角线的末端，另一个顶点 j(j<i)成为对角线的始端，这条对角线已在考虑以顶点 j 为始端的对角线时考虑过了，再考虑将引起重复。

按照以上三条规则，即可得到如图一的搜索树(图中打√的叶结点为不同的分割方案)。

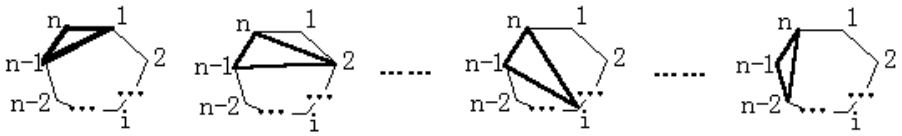
搜索法的数学模型较为复杂，用它可以求出具体方案，但它的抽象化程度不高，导致了求解时的低效率。为了使用上面的规则 2 来提高效率，求解过程还是从多边形及其对角线本身来考虑的，数学模型的作用仅体现在判断对角线是否相交上。用该方法编制的程序在 n 稍大时速度就很慢。(n=12 时已需运行时间 16.2 秒(486DX2/80)，测试结果见附表一。)

<2>.递推法:

上一种方法的数学模型中有很多与问题的要求无关的内容(如对角线的表示、对角线组的表示、每种具体方案)。在递推法建立的数学模型中，我们只考虑凸 n 边形的分割方案总数。

设 k 边形的分割方案总数为 A_k，于是得到 A 数列：$A_3,A_4,A_5...$

考虑 n 边形的分割方案总数 A_n。任取 n 边形的一条边，不妨取边(n-1,n), 若在某一种分割方案中，边(n-1,n)属于三角形(i,n-1,n),那么就将分割方案归入第 i 类，如图二所示。



图二

设第 i 类方案总数为 Bi，则

$$A_n = \sum_{i=1}^{n-2} B_i \quad ①$$

计算 Bi 可用如下方法：

对于第 i 类的方案，原 n 边形已被分割为一个 i+1 边形与一个(n-i)边形，下面的工作分为两步，第一步是将 i+1 边形分割为三角形，有 A_{i+1} 种方案，第二步是将(n-i)边形分割为三角形，有 A_{n-i} 种方案。为了表达的方便，令 $A_2=1$，于是有

$$B_i = A_{i+1} * A_{n-i} \quad ②$$

将②代入①得：

$$A_n = \sum_{i=1}^{n-2} (A_{i+1} * A_{n-i}) = \sum_{i=2}^{n-1} (A_i * A_{n+1-i}) \quad ③$$

于是，问题的数学模型即为：

已知：n 的值及数列 A_2,A_3,A_4……，

该数列满足：

$$A_2=1$$

$$A_j = \sum_{i=2}^{j-1} (A_i * A_{j+1-i}),\ j>2$$

求：A_n

利用这个模型，我们即可很方便地依次求出 $A_3,A_4 \ldots A_n$。

递推法的数学模型比搜索法的简明，抽象化程度更高，效率也高得多。用递推法编制的程序已能应付中等数据，在 n<100 时不超过一秒。但当 n 很大时仍然很慢，n=250 时需 18.7 秒(486DX2/80)，测试结果见附表一。

<3>.母函数法：

上一种方法的数学模型中已去除了很多与问题的要求无关的内容，但同时，问题只要求 An,而上述方法却将 $A_3 \sim A_n$ 都求出了。能否不求 $A_3 \sim A_{n-1}$ 而直接由 n 求出 An 呢？下面用母函数这种数学模型来解决这个问题。

将 A_2,A_3,A_4……作为幂级数的系数，令

$$Y(x) = \sum_{i=2}^{+\infty} A_i * x^i = A_2 * x^2 + A_3 * x^3 + \ldots\ldots \quad ④$$

如果能解出 Y(x),那么也就求出了 An.

为了求 Y(x),我们来看一下 Y(x)^2 的值:

$$Y^2(x) = \sum_{i=4}^{+\infty} [\sum_{j=2}^{i-2} (A_j * A_{i-j}) * x^i]$$

而 $\sum_{j=2}^{i-2}(A_j * A_{i-j}) = A_{i-1}$,因此有:

$Y^2(x) = A_3 * x^4 + A_4 * x^5 + \ldots\ldots$ ⑤

⑤-④*x 得:

$Y^2(x) - x * Y(x) + A_2 * x^3 = 0$

将 $A_2 = 1$ 代入，解出 $Y(x)$：

$$Y(x) = \frac{x \pm \sqrt{x^2 - 4x^3}}{2} = x * \frac{1 \pm \sqrt{1 - 4x}}{2}$$

$\sqrt{1-4x} = 1 - 2x - 2x^2 - 4x^3 \ldots\ldots$ 各项系数均为负数，而 Ai>0,故：

$$Y(x) = x * \frac{1 - \sqrt{1 - 4x}}{2}$$ ⑥，

其中 $\sqrt{1-4x} = \sum_{k=0}^{+\infty}\left[\frac{1}{2}\right]_k * (-4x)^k$, $\left[\frac{1}{2}\right]_k = \frac{\frac{1}{2} * (\frac{1}{2} - 1) * (\frac{1}{2} - 2) * \ldots * (\frac{1}{2} - k + 1)}{k!}$

于是，$$Y(x) = x * \frac{1 - \sum_{k=0}^{+\infty}\left[\frac{1}{2}\right]_k (-4x)^k}{2}$$

$$= x * \frac{1 - (1 + \left[\frac{1}{2}\right]_1 (-4x) + \left[\frac{1}{2}\right]_k (-4x)^2 + \ldots + \left[\frac{1}{2}\right]_k (-4x)^k + \ldots)}{2}$$

$$= -\frac{(-4)\left[\frac{1}{2}\right]_1 x^2}{2} - \frac{(-4)^2\left[\frac{1}{2}\right]_2 x^3}{2} - \ldots - \frac{(-4)^{k-1}\left[\frac{1}{2}\right]_{k-1} x^k}{2} - \ldots$$

所以，$$A_k = -\frac{(-4)^{k-1}\left[\frac{1}{2}\right]_{k-1}}{2}$$

$$= \frac{\frac{1}{2}(\frac{1}{2} - 1)(\frac{1}{2} - 2)\ldots(\frac{1}{2} + 2 - k)}{(k-1)!} * (-1)^k * 2^{2k-3}$$

$$= \frac{1 * (-1) * (-3) * \ldots * (5 - 2k)}{(k-1)!} * (-1)^k * 2^{k-2}$$

$$= \frac{1 * 3 * 5 * \ldots * (2k - 5)}{(k-1)!} * 2^{k-2}$$

$$= \frac{1 * 3 * 5 * \ldots * (2k-5) * 2 * 4 * 6 * \ldots * (2k-4)}{(k-1)!(k-2)!}$$

$$= \frac{(2k-4)!}{(k-1)!(k-2)!}$$

$$= \frac{1}{k-1} * C_{2k-4}^{k-2}$$

于是得出了由 n 直接求出 An 的数学模型：

已知：n 的值，

$$\mathbf{A_n = \frac{C_{2n-4}^{n-2}}{n-1}}$$ ⑦

求：A_n

求解时用公式⑦可直接计算 A_n。

在三个数学模型中，这一个表达最为简洁，抽象化程度最高。用它来求解的效率也最高。n=1000 时不超过 1 秒,n=5000 时也仅需 14.7 秒(486DX2/80)，测试结果见附表一与附表二。

搜索法作为一种最基本的方法，建立在一个较为复杂的数学模型之上，它的特点是可以求出每一种分割方法，但用这种方法来求方案总数显然针对性不强，因此效率很低。递推法是建立在数列这个数学模型之上的，由于去除了很多不必要的因素，效率大为提高，对于 n<=300 时有较好的效果。利用母函数这种数学模型求解，是对递推法的一种数学优化，得出了更为简洁的结论，当 n>300 时充分显示出其优势。

从以上的分析中可以得出这样的结论：数学模型的抽象化程度越高，它的效率越高。这个结论很容易理解，因为抽象化程度越高，数学模型中与问题无关的成分就越少，于是效率也就越高。相反的，若抽象化程度不够高，则数学模型中含有较多的与问题无关的成分，那么，效率也就要低一些。

三、数学模型的推广和应用

数学模型具有可推广性。

数学模型是建立在问题本质的基础上的，而不是建立在问题的表面现象上的。因此，虽然两个问题表面毫无关系，只要它们有相同的本质，就可以用相同的数学模型求解。然而，要看到两个问题有相同的本质并不是一件容易的事。这需要我们抛开问题的表面现象，仔细地比较分析，在问题之间建立对应关系。

数学模型的可推广性与数学模型的抽象化程度有着密切的关系。为解决同一个问题而建立起的不同的数学模型可能具有不同的可推广性。

下面将由母函数建立起的数学模型应用于另一些问题的求解。

【树的计数问题】求具有 n 个结点的二叉树的数目。

设具有 k 个结点的的二叉树的数目为 Dk，则

- 当 k=0 时，是一棵空树，只有一种。
- 当 k>0 时，二叉树可分为根结点、具有 i 个结点的左子树与具有 k-1-i 个结点的右子树。于是具有 k 个结点的二叉树的数目等于具有 i 个结点的二叉树的数目与具有 k-1-i 个结点的二叉树的数目的乘积。

将以上的分析写成公式，就是：

$$D_o = 1$$

$$D_k = \sum_{i=0}^{k-1} (D_i * D_{k-1-i})$$

比较上文中 A 数列与这里的 D 数列可知 $D_n = A_{n+2}$,于是将上文中的数学模

型(⑦式)稍加变换，即得：

$$D_n = \frac{C_{2n}^n}{n+1} \quad ⑧$$

至此，我们已将这个问题用上面的数学模型解决了。这个问题与[多边形分割问题]具有相同的本质，即它们计数的规律是一致的，因此，它们可用相同的数学模型求解。

为了将这种数学模型进一步推广，我们再将上一个问题分析一下：将每棵二叉树的 n 个结点一一编号，使每棵二叉树的前序序列都是 1,2,3…,n。由于前序序列与中序序列可唯一确定一棵二叉树，所以每棵二叉树的中序序列与其它的二叉树都是不同的。(一旦相同，那么这两棵二叉树就是同一棵二叉树了。)

另外，对于一棵前序序列确定的二叉树，它的中序序列可以由前序序列进栈与出栈生成。于是该数学模型又可直接用于下面问题的求解。

【火车进出栈问题】一列火车 n 节车厢，依次编号为 1,2,3,…,n。每节车厢有两种运动方式，进栈与出栈，问 n 节车厢出栈的可能排列方式有多少种。

将这个问题与上一个问题比较一下：列车原始的排列状态(1,2,3,…,n)正是二叉树的前序序列；列车车厢的进栈与出栈对应于二叉树结点的进栈与出栈；列车出栈后的排列状态正是二叉树的中序序列。

将两道题对应起来看，不难发现，列车出栈后的可能排列方式的数目就是二叉树的中序序列的数目，也就是二叉树的数目。

设 n 节车厢的排列方式有 En 种，则

$$E_n = \frac{C_{2n}^n}{n+1} \quad ⑨$$

于是，我们又用相同的数学模型解决了这个问题。

将数学模型推广到[树的计数问题]时，我们只是比较了相似的递推公式，可以说是一种简单的推广。而推广到[火车进出栈问题]时，则是从[树的计数问题]出发，将两个问题对应起来看，进行了很多逻辑分析，相比之下要复杂一些。事实上，很多数学模型的推广应用都需要进行仔细的分析。

从一个问题[多边形分割问题]出发建立起的数学模型 $X_n = \frac{C_{2n}^n}{n+1}$，公式中已完全略去了分割的具体内容，只留下了问题的本质：计数。由于公式表达了计数方法的实质内涵（ $X_0 = 0; X_k = \sum_{i=0}^{k-1} (X_i * X_{k-1-i})$ ）,于是就给了它进一步广泛应用于一类问题求解（外延）的可能。

再考虑一下[多边形分割问题]中搜索法的数学模型。在这两个问题中，搜索法的数学模型显然是不适用的。它包含着多边形的每一种具体的分割方案，没有

很好的体现问题计数的本质，因此影响了这种数学模型的可推广性。在这两个问题中，没有相应的概念对应于多边形的分割方案，于是，搜索法的数学模型便对这两个问题无能为力了。而数列与母函数两种方法的数学模型仍能应用于这两个问题。这正是由于后两种方法的数学模型更加抽象，所以更有利于它们的推广

四、总结

以上三个实例充分说明了数学模型的高效性、可推广性以及它们与抽象性之间的关系。

数学模型具有高效性。从实际问题中建立起来的数学模型可以去除无关的内容，关系清晰，有利于效率的提高。

数学模型具有可推广性。从实际问题中建立求解的数学模型，一个数学模型建立后，往往能将其应用于一类实际问题中。乍一看[分割多边形]与[火车进出栈]没有什么联系，但通过对模型的理解可以发现两个问题有着密切的内在联系：它们是可以用相同的数学模型来求解的。

数学模型的高效性与其抽象性是紧密联系的。数学模型越是抽象，它的效率也就越高。数学模型的可推广性与其抽象性也是紧密联系的。数学模型越是抽象它也就越容易被广泛应用。

【附件】

附表一:按以上三种数学模型设计的程序的运行时间的比较

n	运行时间(秒)(486/80)			结果
	搜索法	递推法	母函数法	
5	0.0	0.0	0.0	5
8	0.1	0.0	0.0	132
10	0.6	0.0	0.0	1430
11	3.2	0.0	0.0	4862
12	16.8	0.0	0.0	16796
20		0.0	0.0	477638700
30		0.1	0.0	263747951750360
40		0.1	0.0	176733862787006701400
50		0.1	0.0	131327898242169365477991900
70		0.3	0.0	862189239989602857261856406637011085 00
100		0.8	0.0	5774335806960135778218770060804285633402073162475661100 0
150		3.1	0.0	3959313147057001992888490018878757680451363792611793474 902551970920541958964206938 7800
200		8.3	0.1	3249701714469247204061030419891100129328703540371004596 972565531458474030562929950769133018913041197185787156 7302000
250		18.7	0.1	2942109465114274900932013291224718543203864499126811120 331716878369694940021100392829583154627202225799961741 962546561436757757673959435471617 2000
300		37.1	0.1	2833615951128645491952141298699350894649246764901164418 208859862469151903255965070803736549992753202965439344 706962132218771245433367832310452689722580702922416256 3399190436400

附表二:母函数法在大数据下的运行时间与结果

n	运行时间 (秒,486/80)	结果
500	0.2	3392161481258547533436328676042834151806671371051948224646322289022038948 4961666016394087658935561710828507304323072584219401165213694125111808676 7433831117255251093952216676752181549754039254079009518747834466367785846 2596953622134102520200986176150660638780745851952786342361786271048679068 0000
1000	0.6	1282659123120329389992418907864563882089032052318971850610071333658659691 7650636508971392269636384330131169530183206423519818758891241594708333432 8015864565893550987852108368101709312590228295938009433241620526247192140 9995611627511580863150740180266996754602824885598197887476283926138637334 4807651604630045903721337776816153681064661207184425277845724281859993280 9910085516200019191596126573215746216258241717105949596945086725752677939 9630744193300422624617858972825189717428936822555053619495392164363039745 4492990445740264272985056960596626000194076108609551836149060341111411420 3867955760000
5000	14.7	1990533983292846325166735064542780033892729043055532801808728231002712118 7676340856613205258009823911061689822819966058800117396543598135427586955 9792778429614014690786606171171528320083045044671489924972047501523334246 8138724573840895739739033811420784581508456866514843728085684023096771538 1553578655967556776543982717552905537280360991836287529653206624959870630 8121895633339730969725161266875649077021947002631530501156605401466085759 7536352915369221824992631352492185831978570219673534486780746350434887270 3806682765573357532768717863477686122111338669153041998736992669734521138 4136884623921590225108813107833859413755075491924954296242363596594466667 1446673358771831246640688142040703856338776629977766659654255268159378975 1044519117369900869457690210352595228237709489430574007459436019028016411 0038827696035585915387351199097375301331556293580890493918098622380755233 3892680521693312997068069144010511282780445912971074861274051992834639631 0561216451805165334432417634618203226858919012500057666732553341830111147 0819533812120988010445061003160557018172554817337108394059108173915553582 1737987646875652182998581745666283678405193386287633644237910478216042546 3668803378393982395552054129650108630734033416898205271951432275542407693 1819721455040492976792101947671393514768860471507412891459327520118017269 6786877749972031468886291116599954859230193707272332890105705243614350673 5132885611033291722431843274758902459024223795911169335414024073046732892 7395645459224329052247990854644972024029972827331377313107196086221062765 0188202833681955696454854968868373243539755357093842908311125187443209114 0686683008816625997915385029705586103119596051190592857213107922653357349 7461860596111378869832344698856089798444316515490995417460293855758521378 0659674972152322112734313586718289148701156895171132685679870431003809667 4186956424692388366951410846587759001196659372267179038623797279196316562 3988309844128732444707420528465376060417840530884098880455141863912466417 6032592842849445923263161630959760172685607996743155249441458740184540341 6372467040622320018510892266507167328210681172963043341422470770335588144 8203281065661300736427314789935878982943581444410274031000848491803374826 2914234177331202318998592244539101401954666570026383034960883253619564875 1981624029939353157525782759785276380074202722356998363216687878275696045 9799214540286045928686781434155385008768086213261980678561187332233776048 5664946553533366401151012142801380925362116746164301057451528028965946453 9972118062862177079566215629850386783557684491180752622175613789741648825 9143313542109632709368336593742684914689724272085005469649013886496227350 8257492244688170498625618918459291917844876832932013180312437483376087705 8196194803111496823543186375259372960899823055880506318903220600017289601 4818245773092045746539907057236677380540322691698615959422148103261747037 1581495719930597851695669555189979332882416754183716357001209090944294741 4739729976104880382195071324370634650548082582843292378198930180928100582 81353800000

【程序】

1.多边形分割问题 搜索法 (sousuo.pas)

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N-,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 16384,0,655360}
Program SouSuo;
Const
  Max=30;
Type
  TPara=record l,r:integer;end; {区间类型}
Var
   method:Longint; {方案总数}
   Para:array[1..Max]of TPara; {区间组}
   n:integer;
   time:Longint;

Function M(a,b:integer):integer;{高精度整数类型}
begin if a<b then m:=b else m:=a;end;

Procedure Make; {搜索多边形的所有分割方案}
Var i,j:integer;
    sp,lp1,lp2:integer;
  Function Connect:boolean; {判断新加入区间组的区间是否与原有的区间有冲突}
  var i,j,k:integer;b1,b2:boolean;
  begin
    j:=para[sp].l;k:=para[sp].r;
    Connect:=true;
    for i:=1 to sp-1 do
      begin
        if (j=para[i].l)or(j=para[i].r) then continue;
        if (k=para[i].l)or(k=para[i].r) then continue;
        if ((j>para[i].l)and(j<para[i].r))xor
           ((k>para[i].l)and(k<para[i].r))
          then exit;
      end;
    Connect:=false;
  end;

  Function PreFalse:boolean; {检验是否有其它的冲突}
  var i:integer;j,k:integer;
  begin
```

```
  prefalse:=false;j:=para[sp].l;
  if j<=2 then exit;
  for i:=1 to sp do
    if (para[i].l=j-1)or(para[i].r=j-1) then exit;
  k:=j;j:=k-2;
  for i:=1 to sp do
    if (para[i].l=j)and(para[i].r=k) then exit;
  PreFalse:=true;
 end;

 Function Pop:boolean;forward;

 Function Push:boolean; {入栈}
 begin
  inc(sp);
  Para[sp].l:=lp1;Para[sp].r:=lp2;
  Push:=((lp1=1)and(lp2=n))or(lp1>n)or(lp2>n)or connect;
  if prefalse then
    begin Push:=true;pop;exit;end;
  inc(lp2);
  if lp2>n then
    begin inc(lp1);lp2:=lp1+2;end;
 end;

 Function Pop:boolean; {出栈}
 begin
  if sp=0 then
    begin dec(sp);pop:=false;exit;end;
  lp1:=Para[sp].l;lp2:=para[sp].r;dec(sp);
  inc(lp2);
  if lp2>n then
    begin inc(lp1);lp2:=lp1+2;end;
  Pop:=(lp1>n)or(lp2>n);
 end;
begin
 sp:=0; {栈顶指针置 0}
 lp1:=1;lp2:=3;
 method:=0;
```

```
  while (sp>=0) do
  begin
    if Push then while pop do;
    if sp=n-3 then {获得了一种方案}
      begin
        method:=method+1;
        while pop do;
      end;
  end;
  writeln('Total: ',method);
end;

var i:integer;s:string;
BEGIN
  write('Input N: ');
  readln(n);{输入多边形边数}
  time:=MemL[$40:$6c];
  if n<3 then writeln('Total: 0')
    else if n=3 then writeln('Total: 1')
    else MAKE; {搜索多边形的所有分割方案}
  writeln('Time: ',(Meml[$40:$6c]-time)/18.2:5:1);
    {输出所用的时间}
END.
```

2.多边形分割问题 递推法 (ditui.pas)

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 16384,0,655360}
Program DiTui;
Const
     Len=100;Max=300;
Type
    Th=array[-1..Len+1]of integer;{高精度整数类型}
Var
   method:array[1..Max]of Th; {i 边形分割方案总数为 method[i]}
   n:integer; {要求的多边形的边数}
   time:Longint;

Function M(a,b:integer):integer; {取最大值}
```

```
begin if a<b then m:=b else m:=a;end;

Procedure Add(var a:Th;b:Th); {a:=a+b;a,b 为高精度整数类型}
var i,j:integer;
begin
  j:=0;
  a[-1]:=m(a[-1],b[-1]);
  for i:=0 to a[-1] do
    begin inc(j,a[i]+b[i]);
          a[i]:=j mod 10000; {每位 integer 存 4 位十进制数}
          j:=j div 10000;
    end;
  if j<>0 then
    begin inc(a[-1]);a[a[-1]]:=j;end;
end;

Procedure Mul(a,b:Th;var c:Th); {c:=a*b;a,b,c 为高精度整数类型}
var i,j:integer;k:Longint;
begin
  fillchar(c,sizeof(Th),0);
  for i:=0 to a[-1] do
    begin
      k:=0;
      for j:=0 to b[-1] do
        if i+j<=Len then
        begin
          inc(k,longint(a[i])*b[j]+c[i+j]);
          c[i+j]:=k mod 10000;
          k:=k div 10000;
        end;
      inc(c[i+b[-1]+1],k);
    end;
  c[-1]:=a[-1]+b[-1];
  if c[c[-1]+1]<>0 then inc(c[-1]);
end;

Procedure OutHigh(a:Th); {输出高精度整数}
var s:string[4];i,j:integer;
```

```
begin
  write('Total: ');
  j:=a[-1];write(a[j]);
  for i:=j-1 downto 0 do
    begin
      str(a[i],s);while s[0]<#4 do s:='0'+s;
      write(s);
    end;
  writeln;
end;
Procedure Make; {递推计算多边形分割总数}
var i,j:integer;a:Th;
begin
  fillchar(method,sizeof(method),0);
  method[2,0]:=1;
  method[3,0]:=1;
  fillchar(a,sizeof(a),0);
  for i:=4 to N do
    for j:=1 to i-2 do
    begin
      mul(method[j+1],method[i-j],a);
      Add(method[i],a);
    end;
  OutHigh(method[n]);
end;

var i:integer;s:string;
BEGIN
  write('Input N: ');
  readln(n); {输入多边形边数}
  time:=MemL[$40:$6c];
  if n<3 then writeln('Total: 0')
    else MAKE;{递推计算多边形分割总数}
  writeln('Time: ',(Meml[$40:$6c]-time)/18.2:5:1);
    {输出所用的时间}
END.
```

3.多边形分割问题 母函数法 见 muhanshu.Pas

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 16384,0,655360}
Program MuHanShu;
Const
    Len=1400;Max=6000;
Type
   Th=array[0..Len+1]of integer;{高精度整数类型 1,按位存储}
   Ty=array[0..Max]of integer;{高精度整数类型 2,按因数存储}
Var
  fi,fo:text;fin,fon:string;
  n:integer;
  time:Longint;

Procedure Mul(var a:Th;b:integer); {a:=a*b;a 为高精度整数类型 1}
Var i:integer;k:Longint;
begin
 k:=0;
 for i:=1 to a[0] do
   begin
     k:=k+a[i]*longint(b);
     a[i]:=k mod 10000;
     k:=k div 10000;
   end;
 if k<>0 then
   begin inc(a[0]);a[a[0]]:=k;end;
end;

Function MaxPublic(a,b:integer):integer; {a,b 的最大公因数}
var i:integer;
begin
 repeat
   a:=a mod b;
   if a=0 then break;
   b:=b mod a;
 until b=0;
 MaxPublic:=a+b;
end;
```

```
Procedure Divide(var k:Ty;h:integer);{k:=k div h;k 为高精度整数类型 2}
Var i,j:integer;
begin
  for i:=1 to k[0] do
    if k[i] mod h =0 then
      begin k[i]:=k[i] div h;
           if k[i]=1 then begin k[i]:=k[k[0]];dec(k[0]);end;
           exit;
      end;
  for i:=k[0] downto 1 do
    if MaxPublic(k[i],h)>1 then
     begin
      j:=MaxPublic(k[i],h);
      h:=h div j;k[i]:=k[i] div j;
      if k[i]=1 then begin k[i]:=k[k[0]];dec(k[0]);end;
      if h=1 then exit;
     end;
end;

Procedure translate(k:Ty;var a:Th);{a:=k;a 为高精度整数类型 1,k 为高精度整数类
型 2}
Var i:integer;
begin
  a[1]:=1;a[0]:=1;
  for i:=1 to k[0] do  mul(a,k[i]);
end;

Procedure Make;{按公式计算多边形分割总数}
Var i,j:integer;k:Ty;a:Th;s:string[4];
begin
  k[0]:=n-2;
  for i:=1 to n-2 do k[i]:=(2*n-3-i);
  for i:=n-2 downto 2 do divide(k,i);
  divide(k,n-1);

  translate(k,a);

  write('Total: ');
```

```
  j:=a[0];write(a[j]);
  for i:=j-1 downto 1 do
    begin
      str(a[i],s);while s[0]<#4 do s:='0'+s;
      write(s);
    end;
  writeln;
end;

var i:integer;s:string;
BEGIN
  write('Input N(<=',Max,'): ');
  readln(n); {输入多边形边数}
  time:=MemL[$40:$6c];
  if n<3 then writeln('Total: 0')
    else MAKE; {按公式计算多边形分割总数}
  writeln('Time: ',(Meml[$40:$6c]-time)/18.2:5:1);
    {输出所用的时间}
END.
```

4.树的计数问题、火车进出栈问题 (tuiguang.pas)

```
{$A+,B-,D-,E-,F-,G+,I-,L-,N+,O-,P-,Q-,R-,S-,T-,V+,X+,Y-}
{$M 16384,0,655360}
Program TuiGuang;
Const
     Len=1400;Max=5002;
Type
    Th=array[0..Len+1]of integer;{高精度整数类型 1,按位存储}
    Ty=array[0..Max]of integer;{高精度整数类型 2,按因数存储}
Var
   fi,fo:text;fin,fon:string;
   n:integer;
   time:Longint;

Procedure Mul(var a:Th;b:integer); {a:=a*b;a 为高精度整数类型 1}
Var i:integer;k:Longint;
begin
  k:=0;
```

```
  for i:=1 to a[0] do
    begin
      k:=k+a[i]*longint(b);
      a[i]:=k mod 10000;
      k:=k div 10000;
    end;
  if k<>0 then
    begin inc(a[0]);a[a[0]]:=k;end;
end;

Function MaxPublic(a,b:integer):integer; {a,b 的最大公因数}
var i:integer;
begin
  repeat
    a:=a mod b;
    if a=0 then break;
    b:=b mod a;
  until b=0;
  MaxPublic:=a+b;
end;

Procedure Divide(var k:Ty;h:integer);{k:=k div h;k 为高精度整数类型 2}
Var i,j:integer;
begin
  for i:=1 to k[0] do
    if k[i] mod h =0 then
       begin k[i]:=k[i] div h;
             if k[i]=1 then begin k[i]:=k[k[0]];dec(k[0]);end;
             exit;
       end;
  for i:=k[0] downto 1 do
    if MaxPublic(k[i],h)>1 then
     begin
       j:=MaxPublic(k[i],h);
       h:=h div j;k[i]:=k[i] div j;
       if k[i]=1 then begin k[i]:=k[k[0]];dec(k[0]);end;
       if h=1 then exit;
     end;
```

```
end;

Procedure translate(k:Ty;var a:Th);{a:=k;a 为高精度整数类型 1,k 为高精度整数类
型 2}
Var i:integer;
begin
  a[1]:=1;a[0]:=1;
  for i:=1 to k[0] do  mul(a,k[i]);
end;

Procedure Make;{按公式计算}
Var i,j:integer;k:Ty;a:Th;s:string[4];
begin
  k[0]:=n-2;
  for i:=1 to n-2 do k[i]:=(2*n-3-i);
  for i:=n-2 downto 2 do divide(k,i);
  divide(k,n-1);

  translate(k,a);

  write('Total: ');
  j:=a[0];write(a[j]);
  for i:=j-1 downto 1 do
    begin
      str(a[i],s);while s[0]<#4 do s:='0'+s;
      write(s);
    end;
  writeln;
end;

var i:integer;s:string;
BEGIN
  write('Input N(<=',Max-2,'): ');
  readln(n); {输入 N}
  inc(n,2);
  time:=MemL[$40:$6c];
  MAKE; {按公式计算}
  writeln('Time: ',(Meml[$40:$6c]-time)/18.2:5:1);
```

```
  {输出所用的时间}
END.
```

【参考书目】

1. 《信息学奥林匹克》1998.1-2 中国计算机学会普及工作委员会 、TSINGHUA UNIVERSITY ACM STUDENT CHAPTER 主办，第 87 页、第 93-94 页；

2. 《数据结构》（第二版），严蔚敏、吴伟民编著，清华大学出版社 1992 年 6 月，第 150-154 页；

3. 《中学生数学建模读本》，孔凡海编著，江苏教育出版社 1998 年 1 月。

隐蔽化、多维化、开放化
——论当今信息学竞赛中数学建模的灵活性

杭州外国语学校 石润婷

【关键字】 数学建模 隐蔽化 多维化 开放化

【摘要】

数学建模是信息学奥林匹克竞赛的有机组成部分。当今信息学竞赛越来越追求数学建模的灵活性。其表现大致有模型的隐蔽化、多维化和开放化三条。本文通过对这"三化"的含义及表现的探讨，研究相应的解题策略。

一、引子

数学建模作为信息学奥林匹克竞赛的一个不可或缺的组成部分，自该竞赛诞生以来，一直在进化，在完善，在发展。当今信息学竞赛越来越追求数学建模的灵活性。也正是这种灵活性，使数学建模的魅力毕现，从而赋予信息学竞赛以无限的生命力和广阔的发展前景。

通过对一系列新兴竞赛题的考察和研究，我发现当今信息学竞赛中数学建模的灵活性可以概括为模型的隐蔽化、多维化和开放化这三条。下面，让我们通过对这"三化"的含义及其表现的探讨，以获得相应的解题策略。

二、主体

（一）隐蔽化

1、定义

"隐蔽"的本意是"借旁的事物来遮掩"。而具体落实到信息学竞赛中，"旁的事物"和被"遮掩"的对象便有了特定的指代。显然，从我们的论题便可一目了然：被"遮掩"的对象即数学模型；而"旁的事物"在这里指的是扑朔迷离的现实情景。

这样，信息学竞赛中数学模型隐蔽化的定义便显而易见了，即借扑朔迷离的现实情景来遮掩数学模型。

2、表现

隐蔽化在信息学竞赛中的一大表现就是"老模型，新面孔"也就是说，沿用我们都熟悉的模型，而制造出全新的场景来容纳此模型，从而给原本赤裸裸的模型披上了新装，将它"掩护"起来。因而相同的模型，在不同竞赛题中的表现往往变幻莫测，如《最佳旅行路线》（NOI97）和《新型导弹》两题，就是典型的例子。题目请参阅附录一、二。

这两题前者描述的是一个由"林荫道"、"旅游街"组成的街道网格，而后者描述却的是一个"导弹爆炸"问题。因此，单从表面看，两者应该是风马牛不相及的然而，两种截然不同的表面现象背后，恰恰蕴藏着相同的原型——求一维数列中"最大"（元素和最大）连续子序列的问题，即已知数列

$A=\{a_1, a_2, a_3, \ldots\ldots, a_m\};$

求 $Maxsum=\max\{\sum_{k=i}^{j} a_k\}, 1\le i\le j\le N;$

这两题有一个共同点，即题目本身并没有直截了当地将数学模型展现出来，而是通过对复杂的实际情景的具体描绘，要求选手自己从实际情景中归纳、抽象出数学模型。因而，它们充分地体现了信息学竞赛中数学模型的隐蔽化特点。

3、策略——“拨开迷雾”法

虽然扑朔迷离的现实情景往往给观者以一种雾里看花的朦胧感，但是，只要我们能以敏锐的目光，透过这纷繁复杂的表面现象去观察并很好地把握模型的实质，问题往往就能迎刃而解。

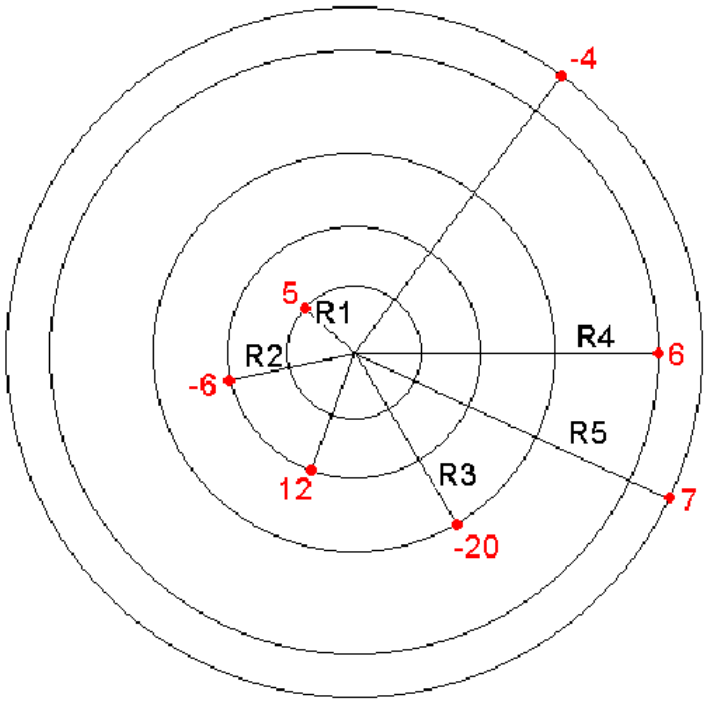
下面，让我们通过对《新型导弹》一题的分析，具体看一看我们拨开迷雾、挖掘问题本质的思维历程。我们首先不能被所谓的“屏蔽半径”和“攻击半径”、“居民点”和“碉堡”这些表象所迷惑。我们应该注意到以下事实：

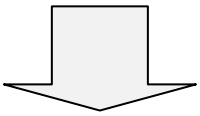
① 可以将居民点的分值修改为它的相反数，则爆炸总利益 = 所有居民点的分值和 + 所有碉堡的分值和；于是我们就能将居民点和碉堡统一起来看待。

② 一旦确定了“屏蔽半径”和“攻击半径”后，某一个建筑物是否被炸毁只与它与圆心（爆炸点）之间的距离有关，而与其在平面内（战场上）的具体位置无关。因此，我们在读入数据的时候，就可以只储存各点与圆心之间的距离，而摒弃具体的 x,y 坐标。

③ 可以将这些点按照离圆心由近到远的顺序排序，同时将与圆心等距的点合并成一个代表点，其分值为这些点的总分值。

如图：





5	6	－２０	6	3

于是，我们的任务就变成了求上图所示的一维表的最大子序列问题，即模型的实质。（附程序《新型导弹》missile.pas）

总之，我们在拿到题目时，不能急于动手编程，而首先应该冷静地去思考分析问题，并从与之关联的各种信息中，正确地“过滤”掉迷惑人的无用的部分，“提炼”出关键的部分，从而很好地把握这“新面孔”背后隐藏的“老模型”。俗话说，磨刀不误砍柴功。只有经过了周密深入的思考，我们才有可能透过现象洞察到问题的本质。而此时再着手编程，就能胸有成竹，事半功倍。

（二）多维化

模型的多维化是信息学竞赛中数学建模灵活性的另一个体现。多维化大致可分为“实”的和“虚”的两类。

1、“实”的多维化

(1) 含义

所谓“实”的多维化，顾名思义，就是指实实在在的，“看得见，摸得着”的多维化。这是它的内涵。而它的外延就是模型由线向面扩展，由面向空间扩展。

我们来看如下两个模型，从中来领略一下“由线向面向空间”的具体含义：

模型 1:已知平面内的若干个点，求覆盖这些点的最小圆。

模型 2:已知空间内的若干个点，求覆盖这些点的最小球。

我们可以看出，模型 2 是在模型 1 的基础上进行多维化而得到的产物。

事实上，这类多维化模型已屡次在竞赛题和练习题中出现。上述模型 2 只是其中小小的一例。

(2) 策略——"降维"法

这种"实"的多维化趋势增添了我们所要考虑的空间因素，进而加大了模型求解的难度。

解这类题目时，我们往往需要追寻出题者的思路，也来一个循序渐进，先从低维的问题出发，在找到低维问题的合适解法后，再加以引申和推广，从而得到相应多维问题的解法。这实际上就是一种"降维"的思想，其优点在于它先化繁为简，有利于我们找出分析思考的着手点。而在我们把握了低维模型的解法后再由简返难，就如囊中探物般地获得多维问题的解法。

"降维"思想在不同的题目中有着不同的运用方式。

i. 类比法

让我们以 NOI97《卫星覆盖》一题为例（请参阅附录三）。此题在分析过程中的第一大障碍也许要数空间难度了。但是此时，如果摆在面前的不是一个三维情景，而是简单的二维情景，我们也许就会信心百倍了。那么，何不先尝试着去求解此二维模型，或许还能从中获得一点启示。事实上，该题的二维模型，即求若干个可能有重叠的共面矩形所覆盖的总面积的问题，在第五届 IOI 中《求图形面积》一题（请参阅附录四）已经出现过，因此为我们所熟悉。其算法描述如下：

第一步，预处理。删去所有被包含矩形。

第二步，平面离散化。

第三步，统计所有被覆盖离散平面格的总面积。

然后，通过类比，我们顺利地推出相应的三维模型的求解方法：

第一步，预处理。删去所有被包含立方体。

第二步，空间离散化。

第三步，统计所有被"覆盖"离散立方格的总体积。

该三维模型求解方案与二维模型求解方案大同小异，只是考虑到效率问题，在统计的时候还需要做一些优化工作。

就这样，通过运用"降维"的思想，我们在相应二维模型求解方案的启示下，圆满地完成了三维模型的求解。（附程序《卫星覆盖》cover.pas）

通过上面的例子可以看到，多维模型从低维模型中诞生，因而难免"遗传"了低维模型的某些特征，使得我们有可能通过类比法直接套用低维模型的求解模式，来为较为复杂的多维问题"接生"。

ii.落到实处的"降维"法

从上述类比法中，我们看到，"降维"思想的整个运用过程其实是在我们的脑海中完成的，而程序的实际操作对象始终都是多维模型。因此，"降维"并没有真正在我们的程序中体现，即没有落到实处。

虽然有不少"多维化"竞赛题可以运用类比法直接套用现成的低维求解模式，但是我们也应该看到，这一招并不是时时处处都能够左右逢源的。我们来看

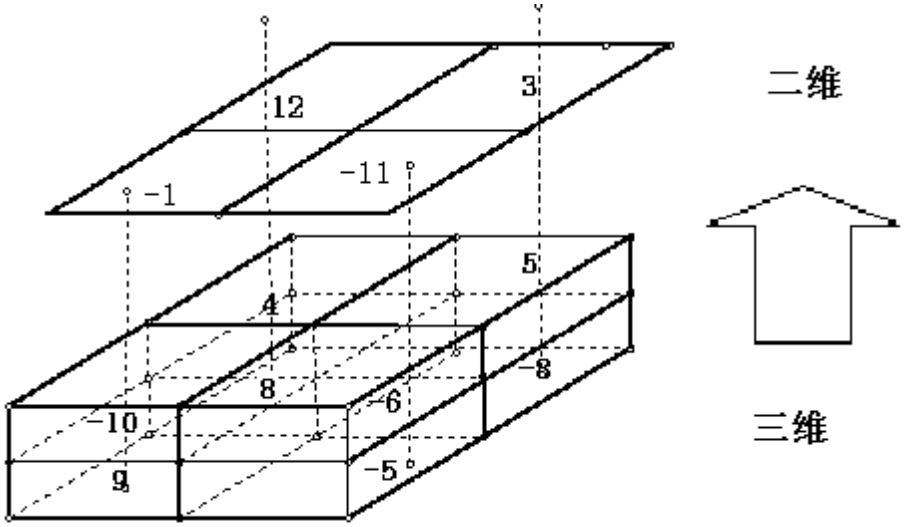
《宇宙探险》一题（请参阅附录五）。

该题的一维模型是上面已经提到过的求一维数列中最大（元素和最大）连续子序列的问题。但是，该一维模型的求解模式（一重循环法）并无法直接套用到三维空间来。

对于此题，比较容易想到的就是穷举法，但是其效率奇低。因为为了确定一个子长方体，共需要 6 个变量，即长方体的左下前角坐标（a1,b1,c1），以及长方体的右上后角坐标（a2,b2,c2）。因而需要六重循环。而即使不考虑这六重循环内为统计该长方体分值和而带来的新的循环，算法的时间复杂度也已经达到了 N^6（N<=50）规模。这显然是个庞大的数字。因此，这样的算法是不可取的。

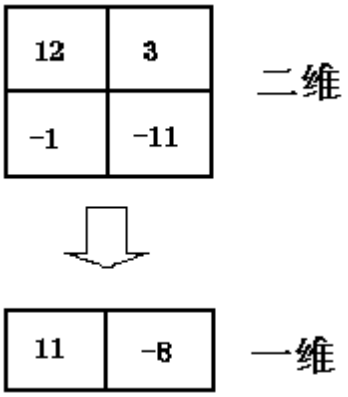
这里，我们可以采用一种落到实处的"降维"方法来解该题。

具体的方法是：假设我们当前所搜索的长方体是自上往下第 a1--a2 层的，为了找出夹在 a1--a2 层之间分值最高的长方体，即进一步确定 b1,b2,c1,c2，我们可以先将该 a1--a2 层从三维压缩到二维的，然后再加以讨论。如图：



现在我们已经将问题转化为了一个二维模型——如何在一张二维表内找出一个分值最大的矩形。我们很容易发现，这个"最大"矩形还原到三维，就是夹在 a1--a2 层间的"最大"长方体。

为了求解上述二维模型，我们可以用同样的方法先确定 b1,b2，然后将 b1--b2 层从二维压缩到一维，然后，就可以运用已知的一维模型求解模式进



行最终求解。如图。

这样，我们不仅在最后求解一维模型时，利用了现成的优秀算法而节省了一重循环，从而将算法的时间复杂度降低到 N^5，更重要的是，我们可以利用“降维”过程减少重复的求和工作。当我们已经完成对 a1--a2 层的搜索，而着手进行对 a1--a2+1 层压缩时，我们可以把累加工作建立在已有的 a1--a2 层压缩表上，即只需把 a2+1 层对应地往 a1--a2 层压缩表上累加。这样，我们就有效地利用了已经获得的信息而避免了大量的重复计算；二维到一维的压缩过程类似。与穷举法相比，该算法的统计工作是分散地进行的，即分布在各层循环之间进行，而非嵌套在最后一重循环内部。这样，就有效地控制了算法复杂度的急剧增长趋势。（附程序《宇宙探险》explore.pas）

在此题的求解过程中，我们将“降维”过程物理地落实到了程序中去。这就是我们所说的“落到实处”的降维。通过与穷举法的对比，可以看到，这种“落到实处”的降维，不仅为思考分析问题提供了清晰的思路，而且往往还能收到一些意想不到的奇妙效果。

由于多维化题自身的多样性，“降维”思想的运用方式还有很多。关键是要针对每题的独特性灵活运用。由于篇幅关系，在此只介绍较为常见的两种运用方式，希望能起到抛砖引玉的作用。

2、“虚”的多维化

(1) 定义

在上述“实”的多维化中，“维”沿用了它的本义，即构成“构成空间的因素”。而在“虚”的多维化中，“维”的含义可以引申为广义的“构成数学模型的因素”。

由此，“虚”的多维化的含义就是“构成数学模型的因素”的增加。

(2) 表现和策略

区别于“实”的多维化，“虚”的多维化的是以增加阶段参数或状态参数等形式体现在我们的数学模型和程序当中的。这既是“虚”的多维化的表现形式，也是相应的解题策略。

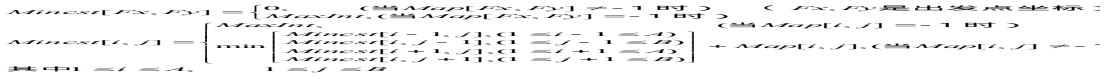
其中最为典型的就是动态规划模型的多维化。（图论中的标号法可以看成是动态规划的优化，因此，这里把标号法也归入动态规划。）

我们知道，经典的动态规划是由阶段、状态、决策三重循环构成的。但是，如今的动态规划，常常不再局限于这陈旧的三重循环模式，而是借助于阶段、状态变量的增加，将模型建筑到了多重循环之上。具有代表性的试题有第七届 IOI《商店购物》、NOI97《积木游戏》以及 IOI98 中国队组队赛《罗杰游戏》等题。题目请参阅附录六、七、八。

为了说明问题，我们以《罗杰游戏》一题为例来具体看一看“虚”的多维化的表现形式。在看该题之前，我们首先来看一个熟悉的模型，以区别比较：

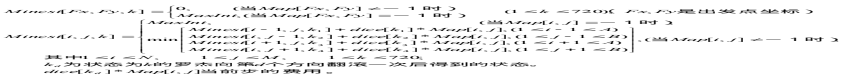
有一个 A*B 的二维表格，表中每一格都被赋予一个整数值-1 或 0--255。现在，我们要从表中(x1,y1)格出发，走到(x2,y2)，途中不能经过-1 格。把沿途经过的所有格中的数累加起来，称为该路线的费用。求所有可能路线的最小费用。

众所周知，该模型是一个经典的动态规划模型。其动态规划方程可以表示为



现在，我们来看一下《罗杰游戏》一题是怎样在上述模型的基础上进行“多维化”的。

我们看到，同样是从(x1,y1)格出发，走到(x2,y2)，与第一个模型相比，《罗》一题由于“罗杰”自身六个面上的数字位置不同而引进了成千上万种不同状态。因此，为了体现这些状态间的区别，新的动态规划方程势必要引入新的参量即新的“维”：



（附程序《罗杰游戏》roger.pas）

3、小结

无论是“实”的多维化，还是“虚”的多维化，都着重考查了选手的思维素质，和知识的引申、迁移能力。因此我们在平时的练习中，特别要注意联想与比较，学会用发展的眼光来看信息学竞赛。不要满足于对某个具体问题的练习，而要在解决该问题后，注意引申和拓宽，去主动地开展多维化，而不是被动地应付多维化。只有这样，才能更上一层楼。

（三）开放化

1、 含义

“开放”的本义是“打破禁令、封锁、束缚”，在信息学竞赛中，我们将它引申为“打破原有的、经典的模型的束缚”。

如果说上文所述的隐蔽化、多维化过程是建立在现有模型的基础上的，那么开放化就是一个彻底的推陈出新，因为它完全打破了信息学竞赛中数学模型的

陈旧模式。

2、表现

竞赛中不断涌现出来的开放化模型，往往给人以一种焕然一新的感觉；同时，也常常使得那些"有名有姓"的算法无用武之地。由于失去了经典模型的借鉴，解题者往往难以找到入手点。这就充分地考验了选手们的创造力。它不但要求我们能够灵活运用已知的模型来解各种题目，更要求我们在遇到超越了经典模型的能力范围的新题时，学会创造性地分析新情况，解决新问题。因此，它是一类颇具挑战性的题目。

比如说 IOI98 中国队组队赛中《电阻网络》一题，就是典型的一例。题目请参阅附录九。

这是一个来源于物理电学的题目。它出现在信息学竞赛中，从头到尾都是崭新的，没有任何的经验模型可以借鉴。如果说该题有什么独特的算法的话，那就是利用简单串并联电路的几个基本物理公式，按部就班地两两合并能够合并的电阻，直至最后只剩下唯一的电阻，即外电路总电阻——这是"顺推"；然后，我们再沿着"顺推"的脚步逐渐回溯，直到将该总电阻再扩展回到原来的电阻网络，并在回溯过程中沿途求得所有电阻上的电流和电压——这是逆推。但是，要实现这"两步曲"，并不是件容易的事。最大的障碍就是没有专门用来描述电路的现成的数据结构可以依赖。因此，此题的关键就在于如何利用有限的程序语言来储存、表示以及操作这张纵横交错的电阻网络，才能既方便快捷又保证不产生歧义。而如何设计出一套漂亮的数据结构模型，也为我们提供了发挥自己的创造与想象能力的广阔天地。而考查选手们运用知识的灵活程度和创造性思维，恐怕也正是该题命题者的意图所在吧。

3、策略——对症下药

由于开放化所引进的模型是千变万化的，没有什么固定的解题模式可寻，因此，只有随机应变，对症下药，才是解决这类题唯一通用的方法。（附程序《电阻网络》resistor.pas）

4、小结

开放化模型在竞赛中层出不穷，是信息学与实践日益结合的必然产物。客观世界的丰富多采决定了我们在通过程序设计以解决实际问题的工作中，所遇到的问题是各种各样的。因此，从没有什么"万能"的模型可以屡试不爽。我们也只有从特定问题的特殊性出发，具体情况具体分析，才能够探求到该问题独一无二的最优解法。

三、总结

模型的隐蔽化、多维化以及开放化，是信息学竞赛中数学建模灵活性的三大体现。我们应该看到，这"三化"并不是彼此独立地存在的，而是水乳交融地贯穿于竞赛题当中的。譬如我们在论述隐蔽化的过程中提到过的《求最佳旅行路线》一题。命题者在隐蔽模型的过程中，实际上将一维的模型用了二维的现象来描述这虽然不同于前文所述的多维化，因为此"维"事实上是非真实的，但是，我们应该承认，这当中同样寄寓着多维的思想。因此，我们应当用联系的而非隔离的目光来看待三者。往往也正是三者的珠联壁合，为我们带来了更新、更妙、更富

有创意的竞赛题。

而作为参赛者，信息学竞赛中数学建模的灵活性无疑对我们的全面素质提出了更高的要求。当然，我们应该意识到，这远不止是竞赛对我们的要求，更是时代对我们青少年一代的殷切期望和热切召唤。参加信息学竞赛，积极主动地培养自己各方面的能力与素质，尤其是数学建模能力和程序设计能力，也正是我们不断追求自我发展与完善，努力为未来做准备的重要实际行动。

【参考书目】

1、国际国内青少年信息学（计算机）竞赛试题解析（1994～1995）
2、国际国内青少年信息学（计算机）竞赛试题解析（1992～1993）

准确性、全面性、美观性
——测试数据设计中的三要素

杭州外国语学校　杨帆

【关键字】　测试数据　准确性　全面性　美观性

【摘　要】　测试数据是当今信息学竞赛不可或缺的有机组成部分，其作用已经越来越被人们所重视。本文提出了测试数据设计中信、达、雅，即准确性，全面性，美观性三要素的观点，并就此进行了逐一论述。

一、引论

国际信息学奥林匹克竞赛(IOI)自 1989 年创办以来已经举办了整整十届，而全国信息学奥林匹克竞赛(NOI)也已有了 15 年的历史。在这些年的发展过程中，信息学竞赛不断进行着自我完善，探索出了一套有自身特点的方法。信息学竞赛和其他学科竞赛有着很大的不同，其中之一，就是评分的方法不同：一般的学科竞赛，采用的是分步给分的办法，即每一步中间过程均有相应的分数；而信息学竞赛，从早年的 IOI 开始，就实行了黑箱测试法，即不对源程序进行阅读、分析，而仅仅根据源程序对于所给定的测试数据得出的结果的正确性进行评分，所有这一切均由电脑自动完成。与相比原先的白箱测试，黑箱测试显得更为客观、公正、高效，因此目前被普遍采用。在黑箱测试过程中，给定的测试数据起着至关重要的作用，其重要程度，甚至不亚于题目本身。一道题目，即使再优秀，如果没有好的测试数据，其价值将大打折扣；同样，一道看似平凡，或是毫不起眼的题，如果配上卓越的测试数据，往往就能够起到令人拍案叫绝的效果。同时，测试数据也是评判题目难易的一个重要关卡，一道题目，采用不同的测试数据其难易程度会有很大的差别。由此可见，测试数据是信息学竞赛题中不可或缺的有机组成部分。因此，对测试数据的讨论是重要的，也是必要的。

著名学者严复曾经在《天演论·译例言》中提出“译事三难信达雅”，把信、达、雅作为了翻译的标准。尽管文学翻译和信息学竞赛并没有必然联系，但是严复的这个三字标准同样可以运用在测试数据的设计上。信，是真实、确切的意思；达，是透彻、通达的意思，可引申为完备；雅，是高尚、美丽的意思。显然，信、达、雅分别对应了测试数据设计的三要素：准确性、全面性和美观性。任何优秀的测试数据，必定是这三者的完美结合。下面，本文将对测试数据设计的信、达、雅三要素进行一一论述。

二、本论

（1）信——测试数据的准确性

显然，测试数据的准确性的重要程度是至高无上的，如果失去了准确性，其他方面就无从谈起，对于一道信息学竞赛题来说，如果包含了错误的测试数据，那么它的竞赛价值就等于零。具体地说，测试数据的准确性体现在以下几个方面：一、测试数据的输入输出格式与题目要求符合；二、测试数据在题目限制的范围之内；三、测试数据的输出结果是正确的。这三点，缺一不可。

首先，测试数据的输入输出格式必须与题目要求符合。输入和输出格式的重要性在实行了计算机自动测试之后充分地显示出来，因为格式的错误和算法的错误造成的后果是相同的，计算机在自动测试过程中只判别选手的输出文件和标准输出文件是否有区别，如果有，则判选手程序错误。因此，如果测试数据中的标准输出文件错误，将会造成所有选手正确的输出均被判错。同时，如果输入数据的格式有误，或者没有按题目要求进行排序等，也会使选手的正确程序无法读出正确数据，从而得到错解。在竞赛中出现这样的事故，后果将不堪设想。

其次，测试数据一定要在题目的限制范围内。所谓题目的限制范围，有很多含义，比如题目规定的数据规模限制，数据的上界和下界，数据的类型等等。例如 IOI'98《天外来客》一题，“输入文件大小可达 2M”是对数据规模的限制，“$0<n\leq 20$，$0<a\leq b\leq 12$”是各数据的上限和下限，而“以 2 表示结尾的 0,1 序列”就是数据的类型。在这点上，任何测试数据都不能越雷池一步。在平时练习时，经常出现为了测试某个程序的运算速度而采用大规模数据的情况，在解搜索题中这种现象更为普遍。这样的测试方法就给测试数据的正确性埋下了很大的隐患。因为这样的数据规模很有可能会超出题目限制，或者表面上符合题目要求，但在程序运算过程中出现了数据越界的情况——这是最容易发生，也是最难被查出的。例如，IOI'98《多边形》一题，题目规定顶点数值总在[-32768,32767]的范围内，而一般的规模稍大的测试数据往往会出现运算结果在范围内，中间数值却在范围外的情况，显然这样的数据就不是符合要求的数据。

最后，也是最重要的，就是测试数据的标准输出结果，即俗称的“标准答案”必须正确无误。这不仅是对信息学竞赛题的要求，而且是对所有学科竞赛题的共同要求，甚至可以说是对所有题目的要求。要实现这点，对于信息学竞赛来说，必须保证标准程序的准确性，因为测试数据的标准输出结果是由标准程序产生的，标准程序的错误将直接导致测试数据标准输出的错误。在设计测试数据的时候，同样应该对这方面加以足够重视，决不能有麻痹思想。

综上所述，只有注意了以上三个方面，测试数据才能保证其必须的正确性。正确的测试数据才是合格的测试数据。

（2）达——测试数据的全面性

严复说过：“顾信而不达，虽译犹不译也。”翻译如此，测试数据的设计同样如此。如果只考虑准确性，还远远不能称得上是好的测试数据，充其量只是符合最低要求罢了。因此，在做到测试数据的准确性之后，必须考虑它的全面性，只有这样才能区分程序的优劣，达到竞赛的目的。

测试数据的全面性，大致体现在两个方面：一是对特殊情况的考查；二是对算法的效率和程序的时空承受能力的考查。这两点缺一不可。下面分别对全面性的这两个方面进行论述。

对特殊情况的考查是十分重要的。所谓特殊情况，又可以分为两种，一种是题目的边界条件，另一种则是题目没有明文规定禁止出现，而又不合常情的情况。这两种情况都是很容易被忽略的，尤其是后者。

先来讨论对边界情况的考查。每道题都有自己的限制条件，即边界条件。显然，当测试数据的范围超过边界条件时，该测试数据失去了准确性。但是，当测试数据的范围恰好在题目的边界条件上时，就达到了对边界条件考查的目的。这样的测试数据，不但是准确的，而且是优秀的。

一般说来，题目的限制范围有两个，即上限和下限。对边界条件的考查，一般情况下都是对题目要求的下限的考查，因为对上限的考查往往需要规模较大的数据，对算法的效率和时空承受能力有较高的要求，可以归为对算法的效率和时空承受能力的考查一类。所以，对边界情况的考查就是对题目限制范围下限的考查。例如 IOI'98《夜空繁星》一题规定"0≤星座总数目≤500"，这样就可以设计一个全空的星图，作为对星座数目等于 0 的边界情况的考查。同样是 IOI'98，《圆桌骑士》一题规定"0≤骑士数目≤63"，可以依此设计没有骑士的数据，考查选手考虑问题的全面性。

科学界有一句名言：非绝对禁止者，皆不无可能。同样，在信息学竞赛中，只要是题目没有明文规定禁止出现的情况，都有可能、且有必要出现在测试数据中。例如 NOI'97《文件匹配》的第 16 个测试数据中就出现了对同一个文件，既要求进行操作，又要求不进行操作的情况。由于题目中并没有明文规定不允许出现这种情况，而这个数据又恰好对此进行了考查，因此是一个漂亮的测试数据。这些特殊情况，既是选手考虑问题时容易忽视的地方，也同样是设计测试数据时容易忽视的地方。对于这些细节，无论选手还是命题者，都必须加以足够重视。

对算法的效率和程序的时空承受能力的考查同样是十分重要的。

一道题目，不同的选手在解答过程中，会建立不同的数学模型，采用不同的算法，并且，这些算法对于本题来说都是可行的。基于这点，在测试时必须从算法效率上区分选手算法的优劣，而这个任务，就理所当然地落到了测试数据身上。

对算法的效率的考查一般使用的方法是设计大规模的测试数据，比如 IOI'98《多边形》一题，数据规模稍大，采用搜索法的程序，运算时间就会成几何级数增长，而基于动态规划算法的程序，效率就比较稳定。显然，通过这样的测试，就达到了区分算法优劣的目的。

必须注意到，选手即使采用了完全相同的算法，在编程时，也会因为各种原因，用不同的数据结构去实现相同的算法，这就必须对程序的时空承受能力进行考查。

在这里，程序的时间承受能力不仅与算法的效率有关，还与程序的优化以及预处理的程度等各种因素有关。这点，在搜索算法中显得尤为突出。为了对算法效率相同的程序进行优劣区分，就必须设计一些测试数据，使进行优化或预处理的程序能够在规定时限内出解，反之则不能，例如 IOI'98《图形周长》一题，可以设计一个矩形完全包含的数据，这样，对此进行预处理的程序显然就占有很大优势。

而程序的空间承受能力与程序采用的数据结构密切相关。采用好的数据结构，往往能够节省内存开销。显然这些在运算时间上不能得到体现。因此，就必须针对这些情况，设计一些测试数据，区分程序数据结构的优劣。比方说，对于搜索题，可以设法增加搜索层数；对于需要处理大量数据的题可以增大数据容量；对于运算量大的题，可以增大数据规模。这样，假如程序采用的数据结构不是最优的，就有可能出现程序执行出错，如栈溢出、堆溢出、超过储存范围等等，或者干脆造成死机。这样一来，对于不同的程序，空间承受能力立刻得到了区分。只有算法效率高，而且数据结构选用恰当的程序才能通过这些测试数据。这样的数据无疑是有极大作用的。

综上所述，要做到全面测试，死板地规定 5 个或 10 个的测试数据显然不行，测试数据的最佳数量是由题目本身决定的，和题目的难度以及要求的数据规模密切相关。少了，不能做到全面测试；多了，又起不到太大作用。例如 IOI'98《圆桌骑士》一题，题目本身比较简单，最大规模的数据也只有 63 个骑士，因此只需少量数据就可以实现全面测试了；而像《图形周长》，不仅数据规模大，而且情况繁多，使用不同的算法效率也将有较大区别，因此就必须采用较多的测试数据进行测试。

同时，测试数据的难度往往能够决定一道竞赛题的难度，这也是需要十分注意的。一般情况下，可以把测试数据的分值作以下分配：

简单数据（规模较小）	25%
特殊情况	20%
对算法效率考查（规模较大）	20%
对程序时空承受能力考查（规模较大）	35%

必须指出的是，对程序时空承受能力考查的测试数据同样也考查了算法效率，因此上面的四部分的划分并不截然分明。按如上比例划分测试数据，显得比较平均，适合一般竞赛。但是，对于层次较高，或是层次较低的竞赛，就应该重新设计测试数据难度比例。请看以下两种划分比例：

	一	二
简单数据（规模较小）	55%	5%
特殊情况	10%	15%
对算法效率考查（规模较大）	20%	25%
对程序时空承受能力考查（规模较大）	15%	55%

显然，对于同一道题来说，第一种测试数据难度比例降低了题目难度，而第二种则提高了题目难度。这样，测试数据就起到了调节题目难度的作用。这样的例子屡见不鲜。例如 NOI'98 第一题《个人所得税》就极为典型。这道题本身并不算难，算法一目了然。它之所以出现在全国竞赛上，而且得分率仅为 11%，就是测试数据在起调节作用。本题的 10 个标准测试数据有 9 个是对程序时空承受能力的考查，只有使用高精度运算才能得出正确解。在竞赛时，只有极少数选手考虑到了这点，通过了 8 到 9 个测试数据，其余选手无一幸免。不妨在此做一个假设：假如当时比赛时并没有采用这批测试数据，而用规模较小的数据，这样，这道题的得分率肯定在 90%以上。即使在分区联赛中，这样的题也只能算是简单题了。

由此可见，测试数据的难度比例是极其重要的。在做到了全面测试之后，测试点分值比例分配将直接影响题目的难度和得分率。当然，这些都是建立在全面的测试数据上的。没有全面性，测试数据就很难影响题目的难度。因此，归根结底，测试数据的全面性才是真正的要点所在。

（3）雅——测试数据的美观性

仅仅做到了准确性和全面性，能不能算得上好的测试数据？答案是否定的。“子曰辞达而已，又曰言之无文，行之不远。三者乃文章正轨，亦即为译事楷模。故信达而外，求其而雅。”同样，对于测试数据来说，“信达而外，求其而雅”也是不可缺少的。所谓测试数据的雅，就是指测试数据的美观性，它和文学的“雅”是有很大区别的。后者指的是语言文字的润色，能使文章更为生动。显然，测试数据只是简简单单的字符、数字的组合，要做到生动的确是勉为其难了，因此，它的“雅”，它的美观，是有自己独特的含义的。但是历来信息学竞赛的命题者对此都没有引起足够的重视。

常言道：规范是美，和谐是美。测试数据的美正是建立在这两点的基础上的。测试数据不仅仅是给电脑用来进行自动测试、自动评分的，它最终还是要给人看的。

选手或是教练在赛后对题目进行分析、总结时，往往要查看测试数据。此时，一个和谐、规范的测试数据起到的作用就远远超过一个杂乱无章的测试数据。

那么，究竟怎么样测试数据才是规范的，才是和谐的呢？不妨先看一个例子，以 IOI'98《图形周长》为例，给出下面五个测试数据（为了表述方便、直观，测试数据均转换成原始图形）：

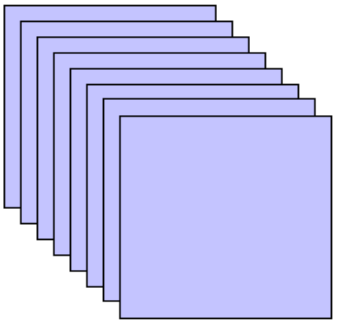


图 1

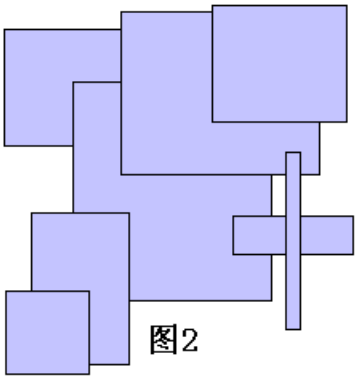


图 2

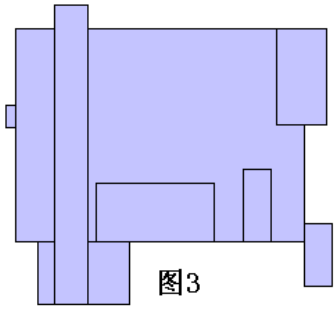


图 3

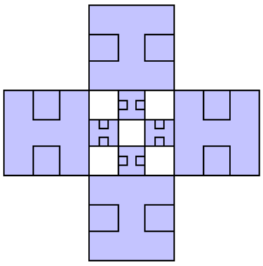


图 4

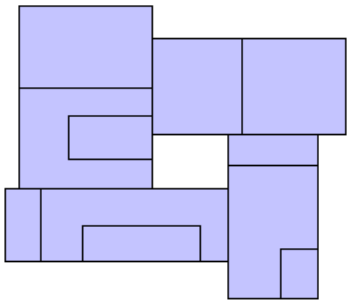


图 5

显然，图 1、图 4 给人的感觉要远远好于图 2、图 3 给人的感觉，而图 5 则介于两者之间。为什么？这就是规范、和谐的作用。图 2、图 3 给人杂乱无章的感觉，已经转换成直观的图形尚且如此，就更不用说原数据了。由此可以看出，规范、和谐都是相对概念，因此没有绝对的规范和绝对的和谐。所谓规范、和谐只是人的一种感觉罢了。如果真要下个定义，可以说假如只看测试数据，就能在脑中形成一个直观的图形，那么这个测试数据就是美的。

对比这个标准，在历年竞赛的测试数据中，能够称得上“雅”的凤毛麟角，更多的数据是随机产生的。不可否认，随机产生的数据具有一般性，在每道题的数据中搀杂几个随机数据的确必要，甚至是达到全面性的必须，但过多过滥的随机数据就不但不是必须，而且破坏了整道题应有的美感。不仅如此，当选手在赛后分析题目的时候，可以想象，如果遇到的都是随机产生的测试数据，他将不可能对程序进行很好的分析，因为他读不懂这些数据；相反，假如他遇到的是美观的数据，那么他就可以更好地，更彻底地分析程序，人脑、电脑一起发动，自然事半功倍。因此，单就这点而言，测试数据的美观性就是极为必要的，因为一道竞赛题的价值不仅仅只在于竞赛，更重要的是在赛后，选手对它分析、总结，从而提高自己的水平。如果没有美观的测试数据，那么题目就不能发挥它在赛后的作用，它的价值也就打了折扣。

由此可见，测试数据的美观性同样显得重要。在今后的竞赛中，对这点的重视程度必将逐年提高。

三、结论

综上所述，信、达、雅是测试数据设计中的三要素，测试数据只有具备了准确性、全面性、美观性，才能称得上是优秀的。但是，这三方面的重要性又互不相同。显然，“信”是基础，“达”是关键，“雅”是提高。没有“信”，测试数据就失去了存在的根本依据，“达”和“雅”就无从谈起，因此，“信”是基础；测试数据光有准确性不够，还必须有全面性，当然全面性不是一个数据就能达到的，需要一组测试数据来实现，只有全面的测试数据才能在竞赛中起到科学测试的作用，因此“达”是关键；一道好的竞赛题，它的价值不仅仅体现在竞赛中，更重要的是能够使选手在竞赛后的分析总结里得到提高，因此测试数据在这个过程中就起到了关键作用，美观的测试数据能最大限度发挥题目潜能，所以“雅”是提高。但是显然，对于一组优秀的测试数据来说，在不顾此失彼的情况下，三者均不可缺少。只有做到了信、达、雅的有机结合，测试数据才是优秀的。

一组优秀的测试数据在题目中起到的作用是不可忽视的，它往往能够对题目的难度产生直接的影响，有时还能够化腐朽为神奇，使一道毫不起眼的题产生惊人的效果。由此可见，测试数据是信息学竞赛题中不可或缺的有机组成部分，必须加以足够重视，进行深入研究。以上只是我个人的一些看法，仅以此抛砖引玉，有关测试数据的设计这个话题，还有待选手和教练们进行更进一步的探讨。

论随机化算法的原理与设计

上海市控江中学 周咏基

[关键字]

随机化算法，稳定性

[摘要]

本文提出了一种新的解决信息学问题的算法——随机化算法，并讨论了其原理与设计方法。论文首先给出随机化算法的定义，说明了由于“运气”的影响，必须对随机化算法的稳定性进行分析。然后分“随机不影响算法的执行结果”,“随机影响执行结果的正确性”,“随机影响执行结果的优劣”三种情况，以从基本算法到竞赛试题中用随机化算法有效解决问题的例子，详细分析了三种情况的随机化算法的原理与设计方法。最后总结出随机化算法的基本原理和共同性质，提出设计随机化算法的一般方法，并指出随机化算法的适用范围和一个有效的随机化算法应具备的特点。

[正文]

1．引论

在这篇论文中，我们将研究一种新概念的算法——随机化算法。顾名思义，随机化就是指使用了随机函数。这里的随机函数不妨是 Borland Pascal(或 Turbo Pascal)中的 RANDOM(N)，其返回值为[0,N-1]中的某个整数，且返回每个整数

都是等概率的[1]。

一个含有随机函数的算法很可能[2]受到不确定因素的支配。人们通常认为，一个受到不确定因素支配的算法肯定不是一个有效的算法——正是在这种思维方式的支配下，随机化算法一直被冷落——但是，在接下来的讨论中，我们将看到完全相反的事情发生：对于一些特定的问题，随机化算法恰恰成了十分有效的解题工具，有时甚至比一般的非随机化算法做得更好。

随机化算法的定义

随机化算法是这样一种算法，在算法中使用了随机函数，且随机函数的返回值直接或间接地影响了算法的执行流程或执行结果。

根据这个定义，并不是所有的用了随机函数的算法都可称为随机化算法。例如，某个算法包含

i←RANDOM(N)，

但变量 i 除了在这里被赋予一个随机值之外，在其它地方从未出现过。显然，如果这个算法没有在其它地方用过随机函数，上面这条语句就无法影响执行的流程或结果，这个算法就不能称为随机化算法。

另一方面，若一个算法是随机化算法，则它执行的流程或结果就会受其中使用的随机函数的影响。我们按影响的性质和程度分三种情况：

1．随机不影响执行结果。这时，随机必然影响了执行的流程，其效应多表现为算法的时间效率的波动。

2．随机影响执行结果的正确性。在这种情况中，原问题要求我们求出某个可行解，或者原问题为判定性问题[3]，随机的效应表现为执行得到正确解的概率。

3．随机影响执行结果的优劣。这时，随机的效应表现为实际执行结果与理论上的最优解或期望结果的差异。

第 2，3 种情况中，随机的影响还可能伴随有对执行流程的影响。

我们后面的讨论就分这三种情况进行。在讨论之前，我们还要澄清一个问题

随机化和"运气"

由于随机化算法的执行情况受到不确定因素的支配，因此即使同一个算法在多次执行中用同样的输入，其执行情况也会不同，至少略有差异。差异表现为出解速度快慢，解正确与否，解的优劣等等。例如：一个随机化算法可能在两次执行中，前一次得到的解较优，后一次的较劣。现在的问题是：在大多情况中，尤其是竞赛时，对于同样的输入，只允许程序运行一次，根据运行结果判定算法的好坏。如此一来，我们就会把出劣解的一次运行归咎于运气不佳，反之亦然然而，比赛比的是谁的算法更有效，而不是谁的运气更好。

既然我们使用了随机函数，我们就无法摆脱运气的影响，所以我们的目标是尽量将运气的影响降到最低。也就是说，我们必须使算法的执行情况较为稳定因此，在接下来的对算法的分析中，我们将从以下四方面分析算法的性能。

1．时间效率；

2．解的正确性；

3．解的优劣程度(解与最优解的接近程度)；

4．**稳定性**，即算法对同样的输入的执行情况的变化。变化越小则越稳定。非随机化算法的稳定性为 100%，随机化算法的稳定性属于区间(0%,100%)。

通常，只要算法的程序实现所用的空间不超过内存限制，我们就不必刻意提高算法的空间效率，所以我们省去了空间效率这项分析。上面第 4 项的“稳定性”可以是算法的平均时间复杂度，也可以是执行算法得到正确解的概率，还可以是实际解达到某一优劣程度的概率。“稳定性”这一项是评判随机化算法好坏的一个重要指标。

2．执行结果确定的随机化算法

在这一节中，我们以快速排序和它的随机化版本为例，讨论执行结果确定的随机化算法。根据引言中的分析，一个随机化算法的执行结果确定，则它的执行流程必会受随机的影响，影响多表现在算法的时间效率上。所以在下面的讨论中，我们省去了对算法执行结果正确性和优劣的分析。

快速排序算法

快速排序是一种我们常用的排序方法，它的基本思想是递归式的：将待排序的一组数划分为两部分，前一部分的每个数不大于后一部分的每个数，然后继续分别对这两部分作划分，直到待划分的那部分数只含一个数为止。算法可由以下伪代码描述。

```
QUICKSORT(A,lo,hi)
1  if lo < hi
2    p←PARTITION(A,lo,hi)
3    QUICKSORT(A,lo,p)
4    QUICKSORT(A,p+1,hi)
```

如果待排序的 n 个数存入了数组 A，则调用 QUICKSORT(A,1,n)就可获得升序排列的 n 个数。以上的快速排序的算法依赖于 PARTITION(A,lo,hi)划分过程。该过程在$\Theta(n)$的时间内，把 A[lo..hi]划分成不大于 x=A[lo]，和不小于 x=A[lo]的两部分。这两部分分别存入 A[lo..p]和 A[p+1,hi]。而在 QUICKSORT(A,lo,hi)过程中递归调用 QUICKSORT()，对 A[lo..p]和 A[p+1..hi]继续划分。

可以证明[4]，快速排序在最坏情况下(如每次划分都使 p=lo)的时间复杂度为$\Theta(n^2)$，在最坏情况下的时间复杂度为$\Theta(n\log_2 n)$。如果假设输入中出现各种排列都是等概率的(但实际情况往往不是这样)，则算法的平均时间复杂度为$O(n\log_2 n)$。

随机化的快速排序

经分析我们看到，快速排序是十分有效的排序法，其平均时间复杂度为$O(n\log_2 n)$。但是在最坏情况下，它的时间复杂度为$\Theta(n^2)$，当 n 较大时，速度就很慢(见本节后部的算法性能对照表)。其实，如果照前面的假设，输入中出现各种排列都是等概率的，那么出现最坏情况的概率小到只有$\Theta(1/n!)$，且在$\Theta()$中隐含的常数是很小的。这样看来，快速排序还是相当有价值的。

但是实际情况往往不符合该假设，可能对某个问题来说，我们遇到的输入大部分都是最坏情况或次坏情况。一种解决的办法是不用 x=A[lo]划分 A[lo..hi]，而用 x=A[hi]或 x=A[(lo+hi) div 2]或其它的 A[lo..hi]中的数来划分 A[lo..hi]，这要看具体情况而定。但这并没有解决问题，因为我们可能遇到的这

样的输入：有三类，每一类出现的概率为 1/3，且每一类分别对于 x=A[lo]，x=A[hi]，x=A[(lo+hi) div 2]为它们的最坏情况，这时快速排序就会十分低效。

我们将快速排序随机化后可克服这类问题。随机化快速排序的思想是：每次划分时从 A[lo..hi]中随机地选一个数作为 x 对 A[lo..hi]划分。只需对原算法稍作修改就行了。我们只是增加了 PARTITION_R 函数，它调用原来的 PARTITION()过程。QUICKSORT_R()中斜体部分为我们对 QUICKSORT 的修改。

```
PARTITION_R(A,lo,hi)
1  r←RANDOM(hi-lo+1)+lo
2  交换 A[lo]和 A[r]
3  return PARTITION(A,lo,hi)

QUICKSORT_R(A,lo,hi)
1  if lo < hi
2    p←PARTITION_R(A,lo,hi)
3    QUICKSORT_R(A,lo,p)
4    QUICKSORT_R(A,p+1,hi)
```

分析随机化快速排序算法

随机化没有改动原来快速排序的划分过程，故随机化快速排序的时间效率依然依赖于每次划分选取的数在排好序的数组中的位置，其最坏，平均，最佳时间复杂度依然分别为$\Theta(n^2)$，$O(n\log_2 n)$，$\Theta(n\log_2 n)$，只不过最坏情况，最佳情况变了。最坏，最佳情况不再由输入所决定，而是由随机函数所决定。也就是说我们无法通过给出一个最坏的输入来使执行时出现最坏情况(除非我们运气不佳)。

正如引论中所提到的，我们现在来分析随机化快速排序的稳定性。按各种排列的出现等概率的假设(该假设不一定成立)，快速排序遇到最坏情况的可能性为$\Theta(1/n!)$。假设 RANDOM(n)产生 n 个数的概率都相同(该假设几乎一定成立)，则随机化快速排序遇到最坏情况的可能性也为$\Theta(1/n!)$。如果 n 足够大，我们就有多于 99%的可能性会“交好运”。也就是说，随机化的快速排序算法有很高的稳定性。

下面是原来的快速排序和随机化后的快速排序的性能对照表。

分析项目		原算法	随机化后的算法
理论时间效率	最坏情况	$\Theta(n^2)$	$\Theta(n^2)$
	最佳情况	$\Theta(n\log_2 n)$	$\Theta(n\log_2 n)$
	平均情况	$O(n\log_2 n)$	$O(n\log_2 n)$
	稳定性	$\Theta(1)$	$\Theta(1-1/n!)$
实际运行情况	随机输入(n=30000)	0.22s	0.27s
	最坏输入(n=30000)	66s	0.22s
	稳定性(n 足够大)	100%	>99%
结论	最坏情况的起因	最坏输入	随机函数返回值不佳
	时间效率对输入的依赖	完全依赖	完全不依赖

对以上表格有几点说明：

1．程序运行环境为 Pentium 100MHz，BP7.0 编译。

2．随机化算法的相应程序的运行时间均为 1000 次运行的平均值。

3．测试随机化算法的稳定性时，相应程序对不同输入各运行了 1000 次。

4．程序代码见 QSORT.PAS。

小结

从以上分析看出，执行结果确定的随机化算法原理是：用随机函数全部或部分地抵消最坏输入的作用，使算法的时间效率不完全依赖于输入的好坏。

通过对输入的适当控制，使得执行结果相对稳定，这是设计这一类随机化算法的常用方法。例如，在随机化快速排序算法中，我们每次随机地选取 x 来划分 A[lo..hi]。这一方法的效应等价于在排序前先随机地将 A 中的数打乱。又如在建立查找二叉树时，可先随机地将待插入的关键字的顺序打乱，然后依次插入树中，以获得较平衡的查找二叉树，提高以后查找关键字的效率。

3．执行结果可能偏离正确解的随机化算法

在这一节中我们讨论第 2 种情况的随机化算法。这种随机化算法甚至会输出错误的结果，但它依然是很有效的。我们以判定素数的算法为例。

朴素的素数判定算法

对于较小的 n，我们可以用"筛数法"判定 n 是否为素数。对于稍大一点的 n，我们可以先求出[2,⌊sqrt(n)⌋]内的所有素数，再用这些素数试除 n。这两种方法都要借助于大数组，如果 n 足够大，就不再适用了。这时，我们只能用 2,3,...,⌊sqrt(n)⌋试除 n，一旦除尽，n 必然是合数，否则为素数。算法描述如下：

```
ISPRIME_NAIVE(n)
1  for a←2 to ⌊sqrt(n)⌋
2    if a|n
3      return FALSE
4  return TRUE
```

实现时，我们可以先判断 n 是否为偶数，然后用 3,5,7,9，...试除 n，以加快程序运行速度。经管如此，当遇到较大的素数 n 时，这一算法还是会显得十分慢的。其最坏情况时间复杂度为$\Theta(n^{1/2})$。

随机化的素数判定算法

换一个角度，由 Fermat 定理我们知道：若 n 是素数，a 不能整除 n，则

$$a^{n-1}\equiv 1 \pmod n$$

必然成立。我们将它改成：若 n 是素数，对于 a=1,2,...,n-1，有 $a^{n-1}\equiv 1 \pmod n$。所以，若存在整数 $a\in[1,n-1]$，使得 $a^{n-1}\neq 1 \pmod n$，则 a 必为合数。我们考虑以下算法：

```
ISPRIME_R(n, s)
1  for i←1 to s
```

```
2   a ←RANDOM(n-1)+1
3   if a^(n-1) ≠ 1 (mod n)
4     return FALSE
5  return TRUE
```

该算法随机地选取 s 个 a 值，检查 $a^{n-1}\equiv 1 \pmod n$是否成立。若发现某个 a，使得该式不成立，则算法肯定地判决“n 是合数”；若选取的 a 都使 $a^{n-1}\equiv 1 \pmod n$成立，则算法提出假设“n 是素数”。

这个算法只产生一种错误，即选取的 s 个 a 值均满足 $a^{n-1}\equiv 1 \pmod n$，而 n 是合数时，算法会认为 n 是合数的证据不足，判其为素数。

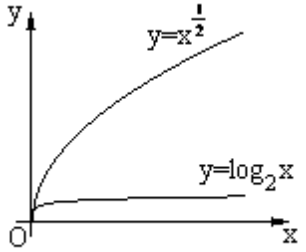
该算法伪代码第 3 行中得求 $a^{n-1} \bmod n$。我们只要$\lfloor \log_2(n-1) \rfloor$次循环就可以求出该值。设 $n-1=b_k2^k+b_{k-1}2^{k-1}+\ldots\ldots+b_02^0$，$b_kb_{k-1}\ldots b_0$ 为 n-1 的二进制表示，则 $k=\lfloor \log_2(n-1) \rfloor$。这样

$$a^{n-1} \bmod n=a^{(\ldots(((bk)\cdot 2+bk-1)\cdot 2+bk-2)\cdot 2+\ldots\ldots b1)\cdot 2+b0} \bmod n$$

记<a>=a mod n，则上式可改写为

$$<<<\ldots<<<<<<a^{bk}>^2><a^{bk-1}>>^2><a^{bk-2}>>^2>\ldots\ldots>^2><a^{b0}>>$$

用 k 次循环即可求出该值。因此随机化的素数判定算法的最坏情况时间复杂度为 $\Theta(s\log_2 n)$。如果视 s 为常数，则其时间复杂度为$\Theta(\log_2 n)$。



比较两种算法

随机化素数判定算法的稳定性取决于，对合数 n，在[1,n-1]内满足 $a^{n-1}\equiv 1 \pmod n$的 a 值的个数。记该个数为 H(n)，则算法判定 n 的稳定性为 $(1-H(n)/n)^s$。事实上，大多数 n 的 H(n)/n 都可达到 98%，几乎所有 n 的 H(n)/n 都不小于 50%，在 10000 内，H(n)/n 小于 50%的不超过 10 个。我们有理由相信，只要 s 取适当的值就能使算法有很高的稳定性。如取 s=50，则算法对几乎所有 n 的稳定性至少为 $1-2^{-50}>99\%$。即使取较小的 s(如 s=5)，算法也未必会得到错误的结果。

随机化素数判定算法的时间效率比朴素的素数判定算法高许多，这可从它们各自的时间复杂度看出。实际运用中，取 s=50，n=761838257287(是素数)，将两种算法对应程序各运行 100 次，前者需 20 秒，后者需 102 秒[5]。其实素数判定通常只是作为一个子程序被调用，其实际使用次数可能还不止 100 次。可见这两种算法的差距是明显的。

小结

执行结果可能偏离正确解的随机化算法基于这样的原理：一个近似正确的算法的近似程度受某个参数的影响。当该参数取某个特点值时，算法能得到正确解。因此，只要将算法执行多次，每次参数取指定范围内的随机值，算法就可能得到正确结果。

通常我们可以这样设计此类随机化算法：先选一个近似算法(这里用了以 Fermat 定理为基础的判定算法)，然后在算法中加入随机化控制(这里用了 a)，最后加外循环控制，多次执行近似算法(这里用了 s)，如此构成随机化算法。可通过重复执行近似算法来控制算法的稳定性，使之达到期望的水平。

4．执行结果有优劣变化的随机化算法

最后我们讨论第 3 种情况的随机化算法。这种随机化算法大多针对要求较优解的问题，例如 NOI'98 中的问题《并行计算》。

若干贪心算法

《并行计算》问题的描述大意为：输入一个由+，-，×，÷，(，)，变量(大写字母)组成的四则运算表达式，输出一段指令，控制有两个运算器的并行计算机在尽量短的时间内正确计算表达式的值。程序的得分将取决于其所能找到的最优解与标准答案(未必是最优解)相比较的优劣程度。

如果用搜索算法来解决这个问题，则每次扩展搜索树必须确定当前的操作数、运算符和运算器，搜索量大得惊人。由于问题并未要求我们求最优解，我们可以用贪心算法求较优解。贪心标准有多种选择，如每次先选取耗时长的运算符又如每次选取最早空闲的运算器。我们目前尚未找到一种普遍适用的贪心标准，而且找到这种标准的可能性不大。另一方面，现在的每种标准都只可在一定程度上对某些输入取得较好的结果，我们完全可以对各种标准分别设计出不符合其贪心方式的输入，使它输出不理想的结果。囿于上述限制，用纯粹的贪心算法无法有效地解决《并行计算》问题。

一种解决的方法是：由于贪心算法有很高的时间效率，我们可在同一个程序中将各种贪心标准全都试一次，但这样无疑极大地增加了"编程复杂度"。下面的随机化贪心算法给出了一个较好的解决方式。

随机化的贪心算法

随机化贪心算法的基本思想是：设置贪心程度 rate%(rate∈[0,100])，选一种较好的贪心标准为基础，每次求局部最优解的过程改为每次求在该贪心标准下贪心程度不小于 rate%的某个局部较优解。这一修改可由以下伪代码描述：

```
PARTOPTIMIZE(rate)
1  for A←局部最优解 to 局部最差解
2    if RANDOM(101) <= rate
3      return A
4  return 局部最差解
```

对于目前任一种贪心标准，都存在不符合它的输入，也就是说，存在输入使算法在不是每次都选局部最优解的情况下，得到的解比每次都选局部最优解所得到的解更优，而上述随机化贪心算法能覆盖这种情况。同时上述随机化贪心算法在 rate=100 时得到的结果就是原来未加修改的贪心算法的结果，所以上述随机化的算法至少不比非随机化的差。

上述思想较简单，所以实现起来不困难。而且贪心算法都有很高的时间效率多次贪心消耗的时间也不会很长。程序代码见 PARALLEL.PAS。在实现中，我们还加了些其它的优化，如对重复项只计算一次。

随机化贪心算法的性能

由于问题对解的约束不多，解的种类和个数就可能很多，因此要从理论上分析随机化贪心算法的性能较为困难。不过我们可以用当时比赛的测试数据对算

法进行测试。这样做还是有一定说服力的。下表为各种贪心算法对应程序的运行结果与标准答案的对照表。

输入文件	一般贪心算法	随机化贪心算法	标准答案	随机化贪心算法与标准答案的差距
INPUT.001	14	14	14	O
INPUT.002	50	50	50	O
INPUT.003	55	55	55	O
INPUT.004	22	21	21	O
INPUT.005	53	47(95%) 46(5%)	46	+1
INPUT.006	42	23	22	+1
INPUT.007	130	100	100	O
INPUT.008	39	33	33	O
INPUT.009	3800	3700	3700	O
INPUT.010	230	210	210	O
INPUT.011	10	12	10	+2
INPUT.012	6300	6300	6300	O
INPUT.013	234	234(99%) 235(1%)	234	O
INPUT.014	3597	3474(42%) 3486(30%) 3462(24%) 3459(4%)	3498	-24
INPUT.015	412	353(52%) 254(22%) 356(16%) 357(6%) 359(3%) 360(1%)	370	-17
INPUT.016	1250	1150(98%) 1250(2%)	1150	O
INPUT.017	580	559(96%) 580(4%)	559	O
INPUT.018	3108	3108	3108	O
INPUT.019	0	0	0	O
INPUT.020	192	192	171	+21

对以上表格有几点说明：

1．程序运行环境为 Pentium 100MHz，BP7.0。
2．“一般贪心算法”一列中的数为各种贪心标准下的算法所得到的较优解。
3．随机化算法的相应程序运行 100 次后确定其运行结果。上表“随机化贪心算法”这一列中，数旁的括号内有得到该结果的概率。若无括号，则表示 100 次运行均为此结果。
4．随机化贪心算法与标准答案的差距 = 随机化贪心算法得到的较优解 - 标准答案。
5．程序代码见 PARALLEL.PAS。

我们从上表看出，随机化贪心算法对大部分输入都能得到与标准答案同样优的结果，甚至对某几个输入(INPUT.014，INPUT.015)能得到比标准答案更优的结果。同时，该算法也有较高的稳定性(注意，这里的稳定性是指获得某一范

围内的解的概率，如得到与标准答案同样优的结果的概率，又如得到比标准答案稍好一点的结果的概率)。另外，其时间效率自然不会差。可见，这里的随机化贪心算法是解决《并行计算》问题的一个有效算法。

小结

执行结果有优劣变化的随机化算法的原理与上节中的基本相同：一个近似算法的近似程度受某个参数的影响，当参数变化时，算法的执行结果会有优劣变化。只要将该算法执行多次，每次参数取指定范围内的随机值，并在执行后及时更新当前最优解，当前解就能不断逼近理论最优解。

设计这样的算法通常以贪心算法为基础，进行像 PARTOPTIMIZE()函数这样的改造。其实 PARTOPTIMIZE()完全可以作为算法框架来套用，该过程外部的算法可以用以下伪代码表述：

```
GREEDY_R()
1  bA←最差解
2  for rate←0 to 100
3    A←{}
4    while 未得到全局解 do
5      a←PARTOPTIMIZE(rate)
6      A←A∪{a}
7    if A 优于 bA
8      bA←A
9  return bA
```

我们从《并行算法》问题的例子中看出，在解决只需求较优解的问题中(如 IOI'97 中的《有害的千足虫》、《在地图上标地名》等问题，CTSC'98 中的《设置站牌》)，执行结果有优劣变化的随机化算法不失为一种有效算法。

5．总结

经过以上对三种情况的随机化算法的分析，我们作如下总结。

随机化算法的原理与设计

随机化算法的基本原理是：*当某个决策中有多个选择，但又无法确定哪个是好的选择，或确定好的选择需要付出较大的代价时，如果大多数选择是好的，那么随机地选一个往往是一种有效的策略*。通常当一个算法需要作出多个决策，或需要多次执行一个算法时，这一点表现得尤为明显。

这个原理使得随机化算法有一个共同的性质：*没有一个特别的输入会使算法执行出现最坏情况*。最坏情况可以表现在执行流程中(主要是时间效率)，也可以表现在执行结果中。

根据这个原理，我们设计随机化算法时，通常一某个算法为母板，加入随机因素，使得算法在难以作出决策时随机地选择。在设计执行结果受随机影响的算法时，我们还可以多次执行算法，使算法不断逼近正确解或最优解[6]。

以上只是设计随机化算法的基本方式。实践中，随机化算法的设计没有公式可套。我们必须具体问题具体分析，深入研究，设计出好的随机化算法。同时在

设计中，我们还需特别注意一个问题——

随机化算法的适用范围和有效性

最后我们考虑对于随机化算法的一个至关重要的问题——随机化算法的适用范围和有效性。

一个问题对算法的所有要求除了问题本身的描述之外，还有诸如内存空间限制，时间限制，稳定性限制等要求。如果一个问题对算法不强求 100%的稳定，即对于同样的输入，不必每次运行的情况都相同(当然这种不稳定性不能太大)，同时作为补偿的，又要求算法在其它某些方面有较好的性能(如出解迅速)，而这些性能是一般非随机化算法无法达到的，那么此时，随机化算法可能就是能有效解决问题的候选算法之一。否则，随机化算法便不适用。

当一个随机化算法适用于解决某个问题，且该算法有较高的稳定性，同时它在其它某些方面有突出表现(如速度快，代码短等)，能比一般非随机化算法做得更出色，那么这个随机化算法就是一个行之有效的算法。

[附录]

[1] 严格地说，RANDOM(N)返回的数是混沌数，而不是随机数。

[2] 这里不能说"一定"。《随机算法的定义》一目中解释了这一点。

[3] **判定性问题**是这样的问题，它要求判定输入数据(问题无输入的话，可把问题中的已知条件看作输入)是否满足某一特定的条件。

[4] 证明过程如下：

记 T(n)为 QUICKSORT(A,lo,hi=lo+n-1)的时间复杂度。显然 T(n)依赖于划分时用的 x=A[lo]在排序后的 A[lo..hi]中的位置。若 PARTITION(A,lo,hi)的返回值为 p，则 T(n)=T(p-lo+1)+T(hi-p)+Θ(n)，其中Θ(n)是 PARTITION(A,lo,hi)的时间复杂度。

在最坏情况下，

$$T(n) = \max_{q=1,2,\ldots,n-1}\{T(q) + T(n-q)\} + \Theta(n)$$

可以证明 $T(n)<=cn^2$，其中 c 为常数，而且当每次划分都使 q=1 时，$T(n)=\Theta(n^2)$，所以在最坏情况下，快速排序的时间复杂度为$\Theta(n^2)$。

证明 $T(n)<=cn^2$ 可以用数学归纳法。以下为简要过程。

简证：$T(n) = \max_{q=1,2,\ldots,n-1}\{T(q) + T(n-q)\} + \Theta(n)$

由归纳假设，

$$T(n) <= \max_{q=1,2,\ldots,n-1}\{cq^2 + c(n-q)^2\} + \Theta(n)$$

$$= c \cdot \max_{q=1,2,\ldots,n-1}\{q^2 + (n-q)^2\} + \Theta(n)$$

而 max{}中的关于 q 的函数在 q=1,n-1 时取得最大值，故

$T(n)<=cn^2-2c(n-1)+\Theta(n)<=cn^2$

其中选择适当的 c，可消去$-2c(n-1)+\Theta(n)$。

证毕。

在最佳情况下，每一次划分都将 A 分成相同大小的两部分。这时，

$T(n)=2T(n/2)+\Theta(n)$

由于排序过程中形成的递归树有$\Theta(\log_2 n)$层，每层的时间代价均为$\Theta(n)$，因此有 $T(n)=\Theta(n\log_2 n)$。上面的分析中忽略了 n/2 不是整数的情况，但这对分析结果没有影响。

我们已经假设输入中出现各种排列都是等概率的。对于平均情况，

$$T(n) = \frac{1}{n}\sum_{q=1}^{n-1}(T(q) + T(n-q)) + \Theta(n)$$

$$= \frac{2}{n}\sum_{q=1}^{n-1} T(q) + \Theta(n)$$

可以证明 $T(n)<=an\log_2 n+b$，其中 a,b 为常数，所以快速排序的平均时间复杂度

为 $O(n\log_2 n)$。

证明 $T(n)<=an\log_2 n+b$ 可以用数学归纳法。以下为简要过程。

$$T(n)=\frac{2}{n}\sum_{q=1}^{n-1}T(q)+\Theta(n)$$

简证：

由归纳假设，

$$T(n)<=\frac{2}{n}\sum_{q=1}^{n-1}(aq\log_2 q+b)+\Theta(n)$$

$$=\frac{2a}{n}\sum_{q=1}^{n-1}q\log_2 q+\frac{2b(n-1)}{n}+\Theta(n)$$

第一项中的

$$\sum_{q=1}^{n-1}q\log_2 q=\sum_{q=1}^{\lceil n/2\rceil-1}q\log_2 q+\sum_{q=\lceil n/2\rceil}^{n-1}q\log_2 q$$

等号右边的第一个和式中的 $\log_2 q<=\log_2(n/2)=\log_2 n-1$，第二个和式中的 $\log_2 q<=\log_2 n$。这样，

$$\sum_{q=1}^{n-1}q\log_2 q<=(\log_2 n-1)\sum_{q=1}^{\lceil n/2\rceil-1}q+\log_2 n\sum_{q=\lceil n/2\rceil}^{n-1}q$$

$$=\log_2 n\sum_{q=1}^{n-1}q-\sum_{q=1}^{\lceil n/2\rceil-1}q$$

$$<=\frac{n(n-1)}{2}\log_2 n-\frac{1}{2}\cdot\frac{n}{2}(\frac{n}{2}-1)$$

$$<=\frac{n^2}{2}\log_2 n-\frac{n^2}{8}$$

由此可得，

$$T(n)<=an\log_2 n+b+(\Theta(n)+b-an/4)$$
$$<=an\log_2 n+b$$

其中选择适当的 a,b，可消去 $\Theta(n)+b-an/4$。

证毕。

[5] 运行环境为 Pentium 100MHz，BP7.0 编译。

[6] 如果将执行结果确定的随机化算法执行多次，时间效率可能很低，所以这类随机化算法一般只执行一次。

[参考书目]

1．《实用算法与程序设计》 吴文虎等著 电子工业出版社
2．《计算机数据结构和实用算法大全》 北京希望电脑公司 谋仁主编
3．《组合数学》 吴文虎等著 电子工业出版社
4．《组合数学》 卢开澄著 清华大学出版社

5．《数据结构》 施伯乐等编 复旦大学出版社
6．《中学数学竞赛导引》 上海教育出版社