多串匹配算法及其启示

南京市外国语学校　朱泽园

[关键字]　模式串　单词前缀树　后缀树　串匹配

[摘要]

字符串处理在实际应用中具有重要地位，其看似简单，但随着研究的深入，各类思想精华涌现其中，难度也变得深不可测。因此信息学竞赛中常以字符串处理为题，锻炼选手的创新能力。

本文第一章提出问题并进行朴素的分析，第二、三、五章分别介绍三个辅助算法：KMP 模式匹配算法、自创的单词前缀树算法，以及后缀树算法。另外基于 KMP 算法的核心思想，在第四章中，面向“多串匹配”问题提出一个线性算法。但本文并没有满足于线性时间复杂度，接着在第六章提出了平均性能更好的算法。最后第七章对算法的构思进行了剖析，并将这种思想方法上升到理论高度。

[目录]

[正文]

§1　问题的提出

§1.1　问题描述

所谓多串匹配，就是给定一些模式串，在一段文章（只出现小写 **a** 到 **z** 这 **26** 个字母）中，找出第一个出现的任意一个模式串的位置。具体来说就是：

给定 **m** 个长度分别为 L_1、L_2……L_m 的模式串数组 $P_1[1..L_1]$、$P_2[1..L_2]$……$P_m[1..L_m]$，假设正文为一个长度为 **n** 的数组 **T[1..n]**，限定 $\sum L \le 100K, m \le 1000, n \le 900K$。我们要找到一个最小的整数 $s \in [1,n]$，满足

$\exists a \in [1,m]$ 使得 $\forall x \in [1,L_a]$ 都有 $T[s+x\text{-}1] = P_a[x]$

注：如模式串为 **cdefg** 与 **efg**，正文为 **abcdefgh** 时，会造成匹配目标的不明确，因此我们一般将“求所有模式串的所有出现位置”这一任务模糊，转而求“第一个”（不过接下来将介绍的算法，可以在不改变复杂度的情况下完全接受此任务）。

含逻辑关键字的搜索引擎是这个问题的实际应用。医学家们在 **DNA** 序列中，搜索可能为变异的几种模式，也是这类问题的典型。因此用有效算法解决该问题能大大提高各行各业的工作效率！

§1.2　最初想法

最朴素的想法是：

```
For i ß 1 to n Do
    For j ß 1 to m Do   If   i+Lj-1<=n   Then
        If   T[i..i+Lj-1]=Pj[1..Lj]   Then
            输出位置 i，并退出循环
```

该算法从小到大枚举每一个位置，并且进行检查。最坏情况下时间复杂度为 $O(n \cdot \sum L)$。

另一个有效的优化是：

```
For   i ß 1 to m   Do
    Xi ß Pi 在 T 中第一次出现的位置，如果没出现返回∞
write   min(X)
```

其中 P_i 在 **T** 中第一次出现的位置，可以通过 **kmp** 算法（下一章将提到），在 $O(n+L_i)$ 内完成。因此总复杂度为 $O(n \cdot m + \sum L)$。

可惜这两种方法面对我们即将解决的数据量，是力不从心的，我们应该努力想出一种线性,或者更优秀的算法。为此，我们要先介绍一个预备算法——**kmp**（**Knuth-Morris-Pratt**）单串匹配算法，和一个预备数据结构——单词前缀树。

§2　Knuth-Morris-Pratt 算法

该算法为 **D.E.Knuth**、**J.H.Morris** 和 **V.R.Pratt** 同时发现的，被称作“克努特——莫里斯——普拉特操作”，简称 **kmp** 算法。

§2.1　定义

给定一个长度为 **m** 的模式串 **P[1..m]**，和一个长度为 **n** 的正文 **T[1..n]**，找到所有的整数 $s \in [1, n-m+1]$，满足：

对于 $\forall x \in [1, m]$ 都有 $T[s+x\text{-}1] = P[x]$。

§2.2　模式串的前缀函数（Prefix Function）

对 $1 \le i \le m$ 有前缀函数 $\pi(i) = \max_{0 \le j \le i}\{j \mid P[1..j] = P[i\text{-}j+1..i]\}$，如**错误！链接无效。**：

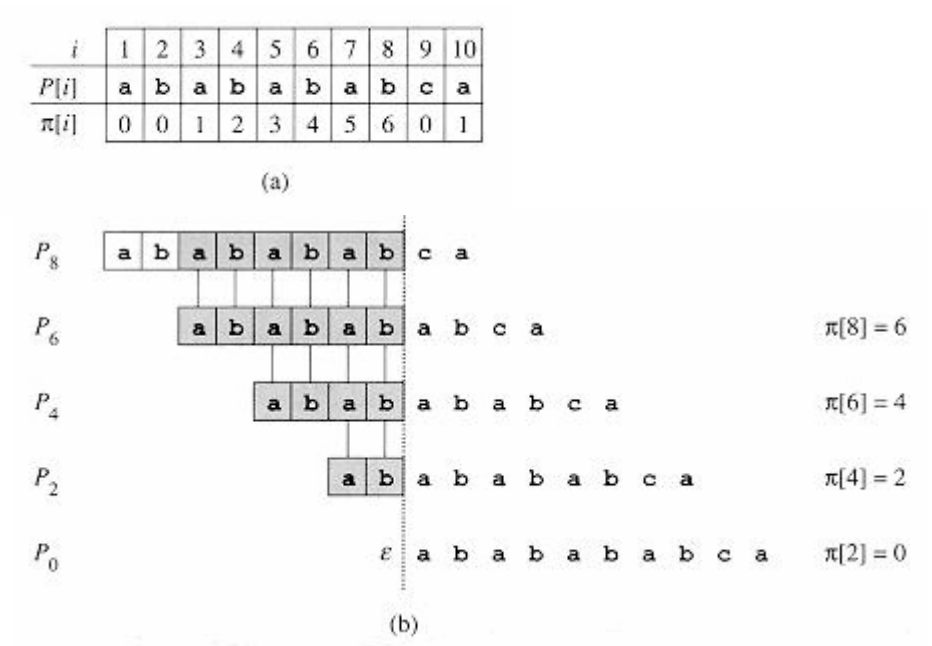

图 1

该函数可以通过如下程序段在数组中完成预处理：

```
k ß 0
π[1] ß 0
For i ß 2 to m do
    While  k>0 and P[k+1]≠P[i]  Do  k ß π[k];
    If  P[k+1]=P[i]  Then  k ß k+1;
    π[i] ß k;
End For
```

因为 k 的增加值最多为 m-1（最多进行 m-1 个“k←k+1”），所以“k←π[k]”的执行数量不会超过 m-1 次。该程序段的复杂度为 O(m)。

§2.3　kmp 主算法

因为前缀函数记录了模式串自身的位移匹配信息，所以在串匹配算法中，遇到了不匹配的字符，就可以根据前缀函数进行适当的调整，而不需要进行重新的比对。

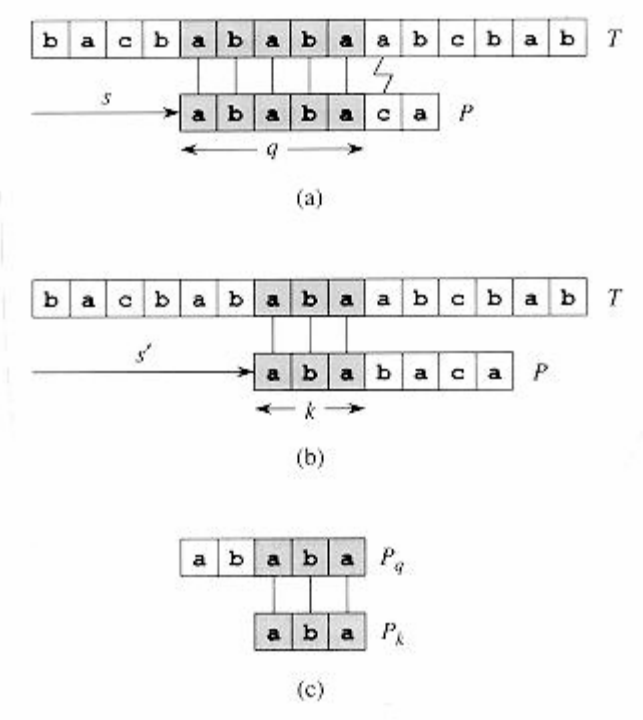

图 2

如**错误！链接无效。**，当我们在串中匹配到了“↯”的时候，可以根据 π[5]=3，将模式串右移 5-3=2 格，再接着比对（如果还不匹配，再根据 π 进行右移……）。

类似于计算前缀数组的方法，我们也不难写出这段主算法的伪代码：

```
q←0
For i ← 1 to n Do
    While  q>0 and P[q+1]≠T[i]  Do  q←π[q]
    If  P[q+1]=T[i]  Then  q←q+1
    If  q=m  Then
        write “在 i-m 处出现模式串”
        q←π[q]
    End If
End For
```

与计算前缀函数的算法相同，q←π[q]的执行次数不会超过 n 次。因此，整个 kmp 算法的时间复杂度为 O(n＋m)

§3　单词前缀树

§3.1　单词查找树（Trie）的定义

所谓单词前缀树，其实就是一棵单词查找树（简称单词树）。理想情况下单词树是一棵无限延伸的 26 叉树。每个结点的 26 个通向子结点的边，被分别命名为 a,b……z。

对于每一个结点 p，从根结点至 p 的路径上的所有字母，可以连成一个字符串，我们定义这个字符串为 S_p。反之，对于一个字符串 S，我们根据该字符串上的字母，可以在单词查找树中找到相对应的路径，以及路径上的最后一个结点 p_s。所以，单词树上的结点，与字符串之间存在**一一对应**的关系。

§3.2　单词树的建立

当然实际情况下我们不能让这棵单词查找树无限延伸。因此我们在一棵初始状态为空的树中，不断插入所涉及，所需要的单词，如**错误！链接无效。**：

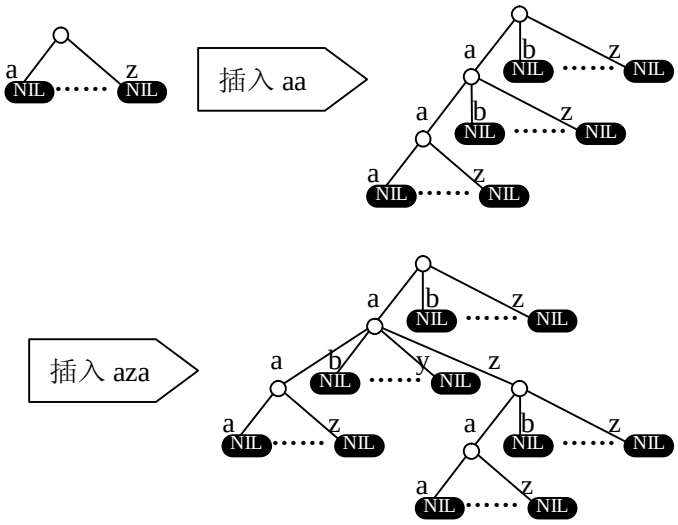

图 3

对一个长度为 n 的字符串 S[1..n]的插入过程是这样的：

```
p = Root;
For i ß 1 to n do
    If   p->Child[S[i]]=NIL   Then
         NEW p->Child[S[i]]; 并对其初始化
    p  ß  p->Child[S[i]]
End For
```

其中 p->Child[ch](ch∈[a..z])代表结点 p 的，标记为 ch 的边所指向的子结点。

§3.3　前缀指针的定义

单词前缀树之所以不同于单词树，是因为它的每一个非根结点上都有一个前缀指针（**Prefix Pointer**）。对于结点 **p** 的前缀指针定义是这样的：

找到最小的整数 $k \in [2,|S_p|]$，使得 $S_p[k..|S_p|]$ 在树中存在。

那么 **p** 结点的前缀指针定义为 $p_{S_p[k..|S_p|]}$。如**错误！链接无效。**（**NIL** 结点在图中被省略，前缀指针用虚箭头表示）：

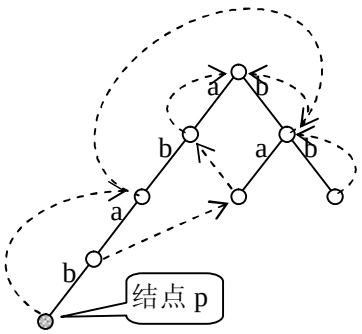

图 4

考虑阴影结点 **p**。S_p=**abab**。我们先找 **bab**，不过这个字符串没有在单词树中出现，因此我们再找 **ab**，这一次找到了。

§3.4　前缀指针的生成

但是前缀指针怎么生成呢？如果和前面一样，基于定义再进行枚举，时间复杂度是可想而知的。仿照 **kmp** 算法中的前缀函数，我们不难想到，可以通过**父结点**的前缀指针，找到当前结点的前缀指针。

定义一个结点的**前缀指针**所指向的结点为该结点的**前缀结点**。

设当前结点为 **p**，此时所有深度比 **p** 小的结点的前缀指针均已处理完毕，则可以通过如下程序段，来完成 **p** 结点的前缀指针的计算：

```
q ß p->Father
ch ß 此时 q->p 的边上的字母
q ß q->Prefix
While  q<>Root and q->Child[ch] = NIL  Do
       q ß q->Prefix
If  q->Child[ch] = NIL  Then  p->Root
       Else  p->Prefix ß q->Child[ch]
```

其中 **p->Prefix** 代表 **p** 结点的前缀指针。

以**错误！链接无效。**为例：

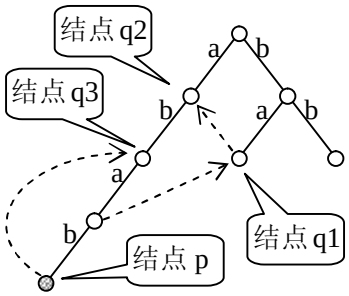

图 5

我们在求结点 **p** 的前缀指针时，首先找到 **p** 结点的父结点的前缀结点 **q1**，可惜 **q1** 没有标记为字符 **b** 的通向子结点的边，因此不得不找到 **q1** 的前缀结点 **q2**，这次 **q2** 拥有了一条通向 **q3** 的标记为 **b** 的边，因此 **q3** 为 **p** 结点的前缀结点。

§4　主算法一（线性算法）

§4.1　kmp 算法的启发

kmp 算法的精髓是减少重复的计算，根据自身的位移匹配，确定模式串的右移量。当然我们很想效仿这个算法，对文章进行仅一次扫描。那么就需要当我们遇到不匹配的字符时，找到当前前缀的尽可能长的后缀。不过其不一定是当前模式的前缀，是任意一个模式的前缀均可，如**错误！链接无效。**：

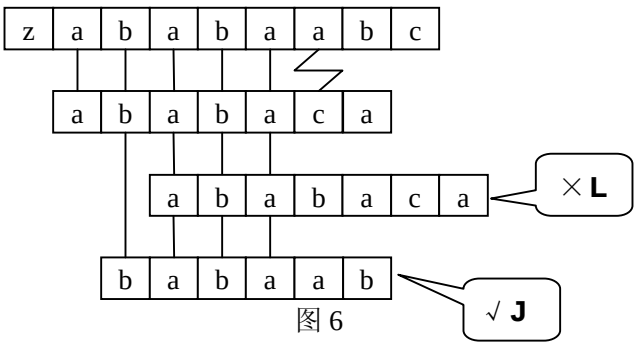

图 6

正文无法匹配 **ababaca**，但并不是一定需要将模式后移 **2** 位，可以用另一个模式“**babaab**”来匹配，只需要后移 **1** 位，后移的位数尽可能少，这才是必要的！

考虑到模式之间的联系紧凑，为了尽可能降低程序的复杂度，树形结构是一个无奈的，也是很好的选择。再加上 **kmp** 的前缀函数的启发，我们就设计出了§**3** 所提到的单词前缀树。

§4.2　单词前缀树的使用及附加标记

我们现在明确了单词前缀树的使用目的，也就不难得出模式串是构成单词前缀树的基本元素。根据§**3** 提到的单词前缀树的建立方法，我们可以把所有模式串插入一棵空的树，并且计算好它们的前缀。

但是，什么时候才真正找到了某个模式 P_a 呢？一个最简单的办法是在所有的 p_{P_a} 处建立一个**附加标记**（后面称此标记为 **Okay**），标记 **Okay=a**，记录这里

是模式 **a** 的终点，没有标记的地方设定 **Okay=0**。可惜这种方法是不成功的，比如存在 **abcd** 和 **bc** 两个模式，如**错误！链接无效。**：

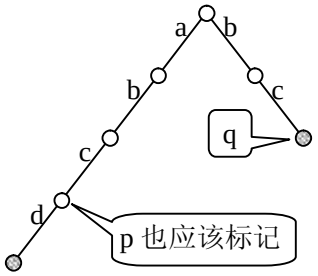

图 7

阴影部分为依照前面的方法标记 Okay 的结点，而事实上我们并没有标记完全。这个过程怎么弥补呢？不难发现 q 结点是 p 结点的前缀结点。因此我们想到了在生成前缀指针时，对 Okay 标记进行传递，即如果当前结点的前缀有 Okay 标记，那么对当前结点也设定 Okay 标记。

§4.3　单词前缀树的时间复杂度

在前缀链接的生成过程中，用到了 **kmp** 前缀数组生成的思想，可想而知，我们可以用平摊分析中的势能方法计算其时间复杂度。

对一个长度为 **n** 的模式串 **P[1..n]**，设 p_i 表示子串 **P[1..i]**在单词树中对应的结点。那么我们将证明 p_0**(Root),**$p_1 \cdots\cdots p_n$ 的前缀指针计算复杂度之和为 $O(n)$。

证明

设 $c_i(0 \le i \le n)$ 为计算 p_i 结点的前缀指针的代价（$c_0 = 1$）。势能函数 Φ 将结点 p_i 映射为一个实数 $\Phi(i)(0 \le i \le n)$，即结点 p_i 的势，在此我们定义 $\Phi(0) = 0$，$\Phi(i)(1 \le i \le n)$ 为 p_i 结点的前缀结点在树中的深度。显然地，当 $0 \le i < n$ 时，p_i 结点为 p_{i+1} 结点的父结点。回到 §**3.4** 生成前缀指针的伪代码：

```
q ß p->Father
ch ß 此时 q->p 的边上的字母
q ß q->Prefix
While  q<>Root and q->Child[ch] = NIL  Do
       q ß q->Prefix          ç
If  q->Child[ch] = NIL  Then  p->Root
       Else  p->Prefix ß q->Child[ch]
```

p_{i+1} 的前缀结点从 p_i 的前缀结点开始搜索，将在“ç”处不断进行调整。不难发现“ç”每执行一次，指针 **q** 的深度至少减小 **1**，所以“ç”的执行次数至多为 $\Phi(i)+1-\Phi(i+1)$，因此计算 p_{i+1} 的前缀指针的代价 c_{i+1}**=1+**“ç”的执行次数 $\le \Phi(i)+2-\Phi(i+1)$。对所有 $0 \le i < n$，将 c_{i+1} 进行累加可得：

$$\sum_{i=1}^{n} c_i \le 2n + \Phi(0) - \Phi(n) \le 2n$$

因此这 n 个结点的前缀指针计算时间复杂度为 $O(n)$。

证毕！

由此不难得出，整个单词前缀树的构造时间复杂度为 $O(\sum L)$。

§4.4　主过程

仿照 kmp 算法的主过程，我们的多串匹配算法也可以大功告成了：

```
Algorithm Multi-Pattern Maching 1;
    预处理  建立单词前缀树
    q ß Root
    For i ß 1 to n Do
        While  q≠Root and q->Child[T[i]] = NIL  Do
            q ß q->Prefix
        If  q->Child[T[i]]≠NIL  Then
            q ß q->Child[T[i]]
        If  q->Okay  Then
            write “在(i - L_{q->Okay})处出现模式串”
    End For
End Algorithm
```

注意，与 kmp 主算法相同，q ß q->Prefix 的次数不会超过 q ß q->Child[T[i]]的总次数，因此这一段程序的时间复杂度为 $O(n)$。

可是这段程序输出的就是模式串的第一个出现位置吗？并不是的。如**错误！链接无效。**：

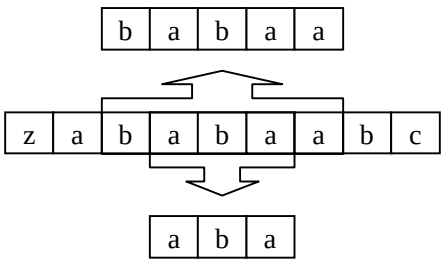

图 8

下面一个模式串其实会先于上面一个模式串找到，但是上面一个模式串应该是第一个出现的。这里增加一些琐碎的判断即可。

§4.5　时空复杂度

整个主程序分为单词前缀树的生成与正文检索两部分。

总时间复杂度为这两部分的时间复杂度之和，为 $O(\sum L + n)$。

空间复杂度是可想而知的，因为单词前缀树上的每一个结点都有 **26** 个指针，所以在单词树中需要使用 $O(26\cdot\sum L)$ 的空间。对正文进行扫描不需要占用额外空间，因此整个程序的空间复杂度为 $O(26\cdot\sum L)$。

§4.6　该算法的一些扩展

在时空复杂度中我们看到了，如果正文长度并不很大，空间复杂度将成为时间的瓶颈。另外如果字符种类增多，那么空间复杂度有可能会吃不消。

这时候存在两种解决办法，一种是通过将所有字符转化成二进制来处理（如**错误！链接无效。**），整个单词前缀树变成了二叉树，但这种算法的优化程度不是很理想。

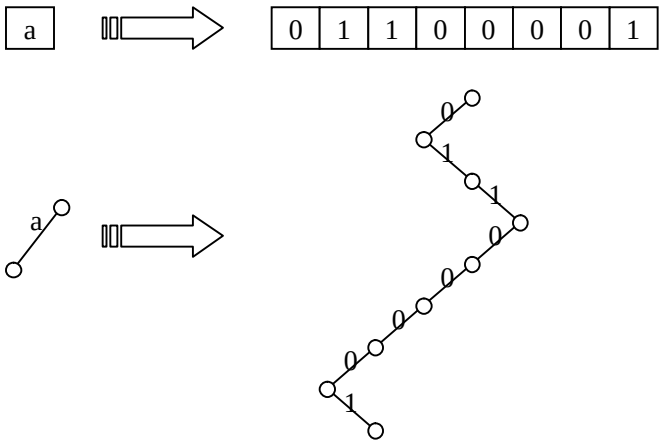

图 9

另一种是动态分配每个结点的子结点，再将其进行有序存储。调用子结点时通过二分查找。这样一来程序的时间复杂度将略有下降，但空间复杂度降为 $O(\sum L)$。

另外如果求“所有模式串的所有位置”，本算法仍然是能胜任的。只是在 Okay 处，不一定只有一个已经匹配的串，可能出现多个。

其实如果最短的模式串很长，并且正文趋近随机，那么有一个平均性能更好的算法。为此我们有需要介绍一个预备算法——后缀树。

§5　后缀树和 McCreight 算法

§5.1　数据结构

后缀树的基本结构，是由一个单词的所有后缀组成的单词树（关于单词树，我们在前面已经介绍过了）。对于单词“ababc”，如**错误！链接无效。**：

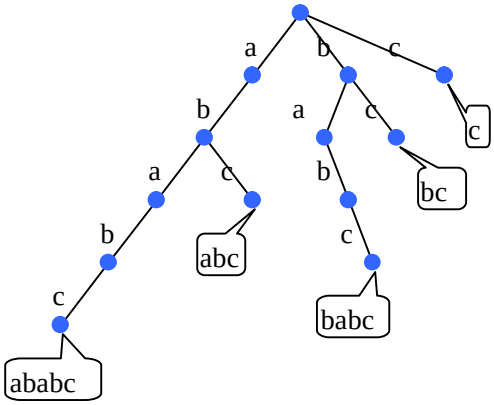

图 10

一共 5 个叶结点，分别代表了 5 个不同后缀（我们称后缀对应的结点为“终结点”）。由于出现的结点数为 $O(N^2)$级别（N 为单词的长度），那么其势必会影响程序的运行效率。改进的思想是进行路径的压缩，如**错误！链接无效。**：

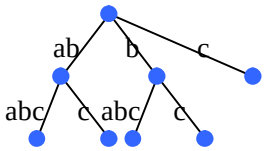

图 11

该压缩方案使得每一个结点最多只有 26 个子结点，并且通向子结点的边上的字符串首字母各不相同。因为每插入一个后缀，最多只会增加一条边，结点数为 O(N)级别。

但是有一个问题，对于上图，增加一个 aba 的边，会出现什么情况呢？会导致在某条边的中间出现了一个“终结点”！这是很麻烦的，因此我们在原字符串的末尾增加一个字符串中未出现的字符（如“$”），来避免该情况的出现。

§5.2　一些定义

设给定的单词为 S[1..n]，suf_i=S[i..n]，代表 S 的第 i 长的后缀。

对后缀树上的结点的定义：

Tpoint 为结点指针。	
p:Tpoint->Father	p 结点的父结点指针
p:Tpoint->Suffix	p 结点的后缀链接（后面我们将提到）
p:Tpoint->Child[‘a’..’z’,’$’]	p 结点的子结点指针，下标为边上字符串的首字母
p:Tpoint->String[‘a’..’z’,’$’]	p 结点通向子结点的边上的字符串，下标为首字母 注：实际操作时可以用指向 S 的**头尾**两个指针表示。

§5.3　建立后缀树（初步）

McCreight 算法用 n 步建立后缀树 T，T 初始为空，第 i 步将 suf_i 插入 T。

定义 $head_i$ 为 suf_i 和 suf_j（$1 \leqslant j < i$）的最长公共前缀中最长的。并定义 $suf_i=head_i+tail_i$。

这样后缀树的建立可以通过如下程序段完成：

```
For i ß 1 to n Do
    找到 head_i 对应的结点 p
    NEW p->Child[tail_i[1]]; 并对其初始化
    p->String[tail_i[1]] ß tail_i;
End For
```

例如 S=“abab$”，后缀树的构建过程如**错误！链接无效。**：

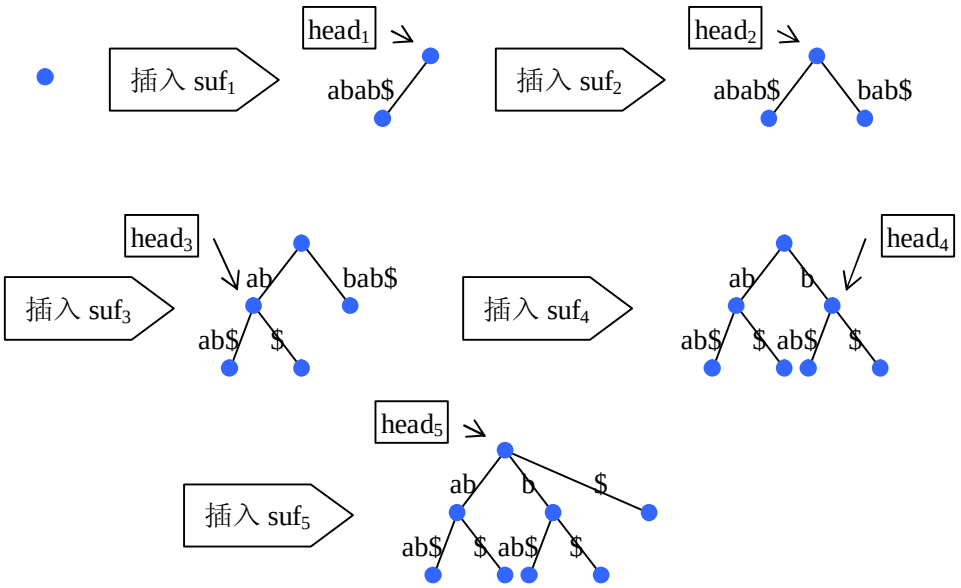

图 12

显然，插入 $tail_i$ 的时间复杂度仅为 O(1)，因此找到 $head_i$ 对应的结点位置，其时间复杂度成了问题的瓶颈。如果从根结点开始不断检索，那么总复杂度势必非常高。

§5.4　后缀链接

性质 1：如果 $head_{i-1}$ 可以写成 x*d* 的形式，其中 x 是一个字符，*d* 是一个可能为空的字符串，那么 *d* 必定为 $head_i$ 的前缀。

证明：这个性质是显然的，因为后缀树在 $head_{i-1}$ 对应的结点 p 处分叉，所以至少有一个 j（j<i-1），满足 x*d* 是 suf_j 的前缀，那么 *d* 为 suf_{j+1} 的前缀。又因为 *d* 为 suf_i 的前缀，因此 *d* 必定为 $head_i$ 的前缀。

后缀链接的定义：

对于每一个非叶结点 p，其对应的字符串可以写成 x*d* 的形式，其中 x 是一个字符，*d* 是一个可能为空的字符串。**定义：p 结点的后缀指针 p->Suffix 指向 *d* 在树中的位置**（如果 *d* 为空串，则指向根结点）。定义根结点的后缀链接指向自身。

如**错误！链接无效。**：

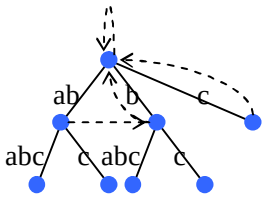

图 13

后缀链接的建立：

建立过程中我们使其满足：

性质 2：每一个非叶结点在被创建后，其后缀链接立即被计算。

另外由性质 1，不难得：

性质 3：所有非叶结点的后缀链接在建立时，其后缀链接对应的字符串在树中已出现。

这样与单词前缀树类似，建立 p 结点的后缀链接时，可以从 **p 的父亲结点**的后缀链接开始，**向下检索**：

```
Procedure Count_Suffix_Link ( p:Tpoint )
    q ß p->Father->Suffix;
    s ß p->Father 到 p 的边上的字符串
    If q = Root then s ß s[2..|s|];
    While s≠'' do
        s1 ß q->String[s[1]]
        L ß min{|s|,|s1|};                    (*)
        If L=|s| and L≠|s1| Then
            NEW r:Tpoint; 并对其初始化
            r->String[s1[L+1]] ß s1[L+1..|s1|];
            r->Child[s1[L+1]] ß q->Child[s[1]];
            q->String[s[1]] ß s;
            q->Child[s[1]] ß r;
            p->Suffix ß r;
            Count_Suffix_Link(r); Exit Procedure;
        End If
        q ß q->Child[s[1]];
        s ß s[L+1..|s|];
    End While
End Procedure
```

正是由于性质 3，(*)处只需要将长度进行比较，即可求出 s 和 s1 的最长公共部分。

§5.5　建立后缀树

这样每次寻找 $head_i$ 时，只需从 $head_{i-1}$ 对应结点的前缀指针开始，**向下检索**即可。回顾一下算法框架：

```
For i ß 1 to n Do
1    找到 head_{i-1} 对应的结点 p
2    从 p->Suffix 开始向下检索到 head_i 对应的结点 q
3    NEW q->Child[tail_i[1]]; 并对其初始化
4    q->String[tail_i[1]] ß tail_i;
5    If p->Suffix≠q Then Count_Suffix_Link(q);
End For
```

时间复杂度分析：

语句 1、3、4 的复杂度为 O(1)，共 O(N)。

语句 5，即所有非叶结点后缀链接的计算，因为 Count_Suffix_Link 的(*)处直接将长度进行比较，因此总复杂度与结点数成正比，即 O(N)。

语句 2 的时间复杂度为 O(|$head_i$|-|$head_{i-1}$|+1)，总复杂度亦为 O(N)。

因此 McCreight 算法建立后缀树的总时间复杂度为 O(N)。

§6　主算法二（平均性能更好的算法）

§6.1　单词前缀树的使用和扩展

与主算法一类似，我们使用一个单词前缀树 **TreeA**，其由所有的模式串构成。

为了适用主算法二，还需要在每一个结点增设一个参数 Shift。它记录每一个结点到达任意一个 Okay 结点（自身除外）的最短路径（既可以通过树中的边，也可以通过前缀指针），其中树中的边的权值为 1，前缀指针的权值为 0。这个 Shift 参数代表如果指针指向该结点，那么至少需要读入多少字符，才有可能得到一个模式匹配。

对于模式串 “abab”“ba”“bb”，如**错误！链接无效。**：

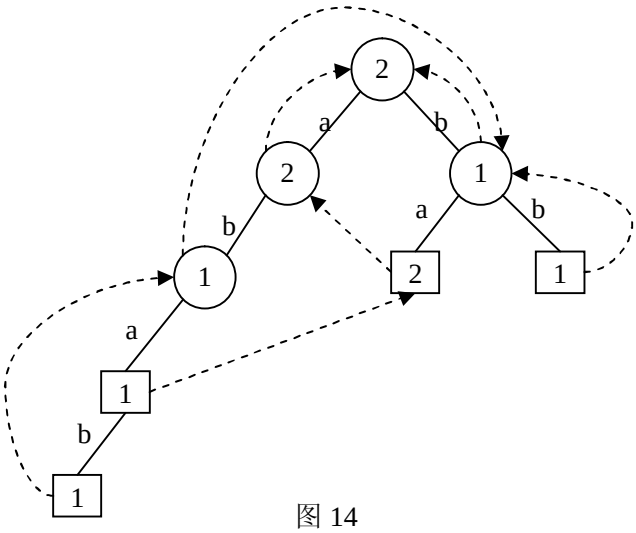

图 14

其中矩形代表 Okay 结点，结点上的数字代表 Shift 参数。

Shift 参数的建立是很容易想到的，插入模式串 P_i 的第 j 个字符时，设：

$$new = \begin{cases} L_i - j & (1 \le j < L_i) \\ L_i & (1 = L_i) \end{cases}$$

那么：

- 如果建立了一个新结点 p，那么 p->Shift　ß　new。
- 如果达到了一个已经存在的结点 p，那么 p->Shift　ß　min{p->Shift,new}

仅仅这样当然是不够的，因为还没有考虑前缀链接。注意到前缀链接总是从下层指向上层，因此我们进行一次按照层号从小到大的遍历，并且对每个结点 p：

- p->Shift　ß　min{p->Shift, p->Prefix->Shift }

遍历的结点总数为 N，因此 TreeA 的建立以及维护操作总时间复杂度 O(N)。

§6.2　后缀树的使用和扩展

主算法二中，使用了一个后缀树 TreeB，其由所有模式串倒置后的所有后缀组成。

当模式串为“abab”“ba”“bb”时，它们的倒置分别为：“baba”“ab”和“bb”，这三个字符串的所有后缀组成的后缀树如**错误！链接无效。**所示：

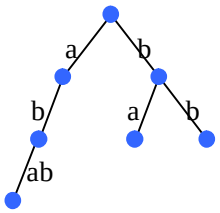

图 15

注：与§5 所述不同，在图中，没有出现含“$”的边。这是因为我们根据需要，不需要将它们表示出来，“$”完全是为了建立的方便。

这样的 TreeB，可以使我们在 O(N)的时间内，从后向前地查看一段长度为 N 的字符，检测它是否为任意一个模式串的子串。

§6.3　TreeA 和 TreeB 上的两个函数

Function ScanA(L,R,point);
　在 TreeA 中，从 point 结点开始扫描 T[L..R]，point 置为最后到达的结点
　While　point->Shift　<　最短的模式串的长度 div 2　**Do**
　　从正文中再读入一个字符，并且在 TreeA 中，point 继续移动
　End While
　write　过程所有遇到的所有 Okay（遇到的所有匹配）
　return　正文最后扫描到的字符位置，以及 point 指针的最后位置
End Function

注：

1、所谓在 TreeA 中，根据正文进行扫描，实际上就是§4.4 的步进操作，即顺着 TreeA 往下走，无法继续时遵循前缀指针回退，平摊时间复杂度为扫描的字符数。

2、为了叙述方便，这里转求所有模式串在正文中出现的所有位置。

```
Function ScanB(L,R);
    在 TreeB 中，将 T[L..R]从右向左进行扫描
        直到没有符合的路径，或者到达 L。
    return 正文最后扫描到的字符位置
End Function
```

这个过程如果返回了 L，说明 T[L..R]是某一个模式串的子串，否则不是。当 T[L..R]是某个模式串的子串时，我们称其为“**有效的**”，反之称其为“**无效的**”。

§6.4　主过程

```
Algorithm Multi-Pattern Maching 2;
    预处理：建立 TreeA 和 TreeB
    L ß 0;    R ß 最短的模式串的长度;
    point ß TreeA.root
    While R<=n Do
        pos ß ScanB(L+1, R);
        If  pos>L+1  Then  point ß TreeA.root;
        (L,point) ß ScanA(pos, R, point) ;
        R ß L+point->Shift ;
    End While
End Algorithm
```

该算法的基本思想是：

当前的搜索起点为 L+1，如果在此处出现一个模式串，那么模式串串尾应该在 R 之后。我们从 R 开始反向检索（ScanB），设 ScanB 在 pos 处停止：

- 如果 pos=L+1，那么 T[L+1..R]为“**有效的**”，因此我们可以正向（ScanA）从 L+1 继续往后检索。
- 如果 pos>L+1，那么 T[L+1..pos-1]的每一个字符，均不可能成为某个匹配的模式串的串头（反之，若 L+1<=x<=pos-1 处出现模式串，那么 T[x..R]必为“有效的”，而 ScanB 的检索终止在 pos 处，矛盾）。
  T[pos..R]是“有效的”，因此我们可以将 point 重置为 TreeA 的根结点，并且从 pos 往后检索（ScanA）。

再看 ScanA，ScanA 的“直到 point->Shift >= 最短的模式串的长度 div 2”才停止循环，是为了确保下一个 R-L>=最短的模式串的长度 div 2。

错误！链接无效。可以清楚地表现出主算法二的工作顺序：

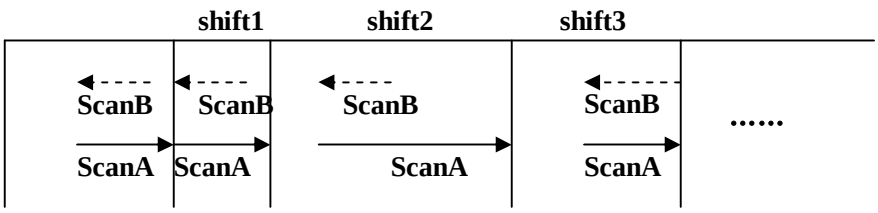

图 16

§6.5　一个例子

模式串为“abcd”和“bcde”，建立单词前缀树 TreeA：

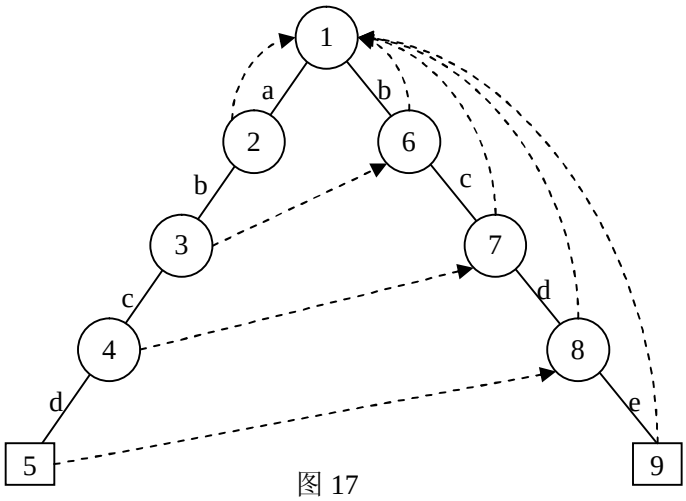

图 17

其中每个结点上的数字代表编号，矩形为 Okay 结点，他们的 Shift 参数如下表所示：

编号	1	2	3	4	5	6	7	8	9
Shift	4	3	2	1	1	3	2	1	4

后缀树 TreeB 在此省略，因为我们可以直接观察任意一段字符是否为“**有效的**”

设正文 T=“abcabcde”，模拟主算法二的进程：

阶段 1：L=0, R=4, point=1

从 R 到 L 逆向进行 ScanB，因为只有“a”为“**有效的**”，“ca”为“**无效的**”，所以 pos=4。因为 pos>L+1，因此 1..3 的正文位置上，不可能出现模式的匹配，否则如**错误！链接无效。**：

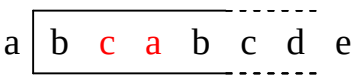

图 18

若出现某个匹配的模式，那么必然包含了“ca”，而“ca”是“**无效的**”，不被任何模式包含。

在 TreeA 上，从 point=1 开始，走过字符[pos..R]=“a”，point=2。因为 Shift[2]=3 >= 最短的模式长度 div 2，所以 ScanA 在 point=2 处停止。

第一个阶段的扫描完成，未发现模式串匹配。

阶段 2：L=4, R = L+Shift[point] = 7, point=2

从 R 到 L 逆向进行 ScanB，“bcd”为“**有效的**”，ScanB 在 pos=5 处停止，pos=L+1，因此不用修改 point 值，直接正向对 T[pos..R]进行 ScanA 扫描，point 到达 5，发现一个模式匹配“abcd”。然而 Shift[5]=1<最短的模式长度 div 2，因此 ScanA 继续扫描字符“e”，point 从 5 到达 8 在到达 9，又发现一个匹配。

字符扫描完毕，算法结束。

§6.6　时间复杂度分析

单词前缀树和后缀树的建立时间复杂度为$O(26\cdot\sum L)$这是不容置疑的（注意正如§4.6 所述，空间复杂度仍有可能成为时间的瓶颈），我们重点考虑对正文检索的主过程的时间复杂度。

设最短的模式串长度为 θ。

最坏情况下，ScanA 将会因为 Shift 始终<$\frac{q}{2}$，而将所有字符扫描一遍。因此主算法最坏情况下将耗费 O(N)的时间。

当所有的模式串长度均为 θ，且 θ 足够大时，若正文随机，并且字母表足够大，因为 ScanB 将所有的 T[L+1..R]的字符扫描完毕的概率并不大，因此平均复杂度将远远小于 O(N)，下面将完成平均复杂度的计算和证明：

设所有的模式串长度均为 θ，设$M=\sum L=\theta^k$，因为 θ 比较大，因此 k 为一个较小的常数。设 a 为字母表（alphabet）的大小（26 或者其它），因为 θ 和 a 足够大，可令其满足$\log_\alpha M^3\le\frac{\theta}{2}$即$3k\cdot\log_\alpha q\le\frac{\theta}{2}$。

另外定义一次 ScanB 和一次 ScanA 的循环为**一个阶段**。

命题 1　每一个阶段中，R-L 的数量级为$O(q)$。

证明　在主算法中，第一次进入循环时，L=0，R=θ，显然满足。因为 ScanA 的终止条件是 Shift≥θ div 2，因此接下来每次 R≥L+θ div 2，所以 R-L 的数量级为$O(q)$。

命题 2　一共有最多$O(n/q)$的阶段。

由命题 1，这是显然的。

命题 3　对于模式串集合 P，有不超过q^{2k}个子串（即“**有效的**”字符串）。

证明　对每一个模式串，有不超过q^2个子串，而一共有 θ^{k-1} 个模式串，因此有不超过q^{2k}个“**有效的**”字符串。

命题 4　存在一个常数 C，使得每读入$C\cdot\log_a q$ 个字符，其组成的字符串，有不超过$\frac{1}{M}$的机率成为模式串集合 P 中某个模式的子串。

证明　由命题 3，有不超过q^{2k}个“**有效的**”字符串。而长度为$C\cdot\log_a q$ 的

字符串有 $a^{C\cdot\log_a q}$ 种可能，每一种有相同的机率被读入，有不超过 $\frac{q^{2k}}{a^{C\cdot\log_a q}}$ 的机率读入一个“**有效的**”字符串。因此 $\frac{q^{2k}}{a^{C\cdot\log_a q}}\geq\frac{1}{M}=\frac{1}{q^{k}}$，可解得：$C\leq 3k$，是一个常数。根据条件有 $C\cdot\log_a q\leq\frac{q}{2}$。

命题 5　ScanA 中，处理至 TreeA 上的深度为 x 的结点 p 时，若对应正文字符 T[y]，那么 T[y-x+1..y]必为“**有效的**”。

证明　其实这也是很显然的，因为 T[y-x+1..y]其实就是 TreeA.Root 与 p 结点在树上的路径组成的字符串，而这个字符串显然是某个模式串的前缀，因此其为“**有效的**”。

命题 6　ScanA 中，读到 Shift$\geqslant\frac{q}{2}$的结点并结束扫描，需要在[L..R]之外，多读入字符数的期望值为 $O(\log_a q)$。

证明　若在读入前 $C\cdot\log_a q$ 个字符中，ScanA 退出，那么字符数为 $O(\log_a q)$。

若在读入 $C\cdot\log_a q$ ~$2C\cdot\log_a q$ 的字符中，ScanA 退出，那么处理完前 $C\cdot\log_a q$ 个字符后，指向 TreeA 的指针 point 的深度必然大于等于 $\frac{q}{2}$（若小于 $\frac{q}{2}$，那么 point->Shift 显然$\geqslant\frac{q}{2}$），而 $C\cdot\log_a q\leq\frac{q}{2}$，由命题 5 可得这 $C\cdot\log_a q$ 个字符组成的字符串是“**有效的**”，而其概率由命题 4，应小于等于 $\frac{1}{M}$。

同理可得，在读入 $xC\cdot\log_a q$ ~$(x+1)C\cdot\log_a q$ 的字符中，ScanA 退出的概率为 $\frac{1}{M^{x}}$。

因此期望值小于 $\sum_{x=0}^{\infty}(x+1)\cdot C\cdot\log_a q\cdot\frac{1}{q^{xk}}=O(\log_a q)$。

命题 7　每一个阶段读入字符数量的期望值为 $O(\log_a q)$。

证明　分两种情况

1、ScanB 读入的字符数小于等于 $C\cdot\log_a q$，由命题 4，其发生的概率不小于 $1-\frac{1}{M}$。ScanA 将从 TreeA 的根结点开始，读入最多 $C\cdot\log_a q$ 个字符后，因为

$C\cdot\log_a q \le \frac{q}{2}$，Shift 参数必然大于等于$\frac{q}{2}$，ScanA 退出。这种情况下最多将读入$2C\cdot\log_a q$ 个字符。

2、ScanB 读完第$C\cdot\log_a q$ 个字符时并未停止，由命题 4，其发生的概率不大于$\frac{1}{M}$。这种情况下读入字符数的期望值，将不超过 ScanB 的q，加上 ScanA 的q，以及 ScanA 因为 Shift 参数不满足而附加读入的$C\cdot\log_a q$ 个字符（由命题 6）。

因此每一个阶段读入字符数量的期望值为：

$$(1-\frac{1}{M})\cdot 2C\cdot\log_a q+\frac{1}{M}\cdot(2q+C\log_a q)=O(\log_a q)$$

结论　由命题 2 和命题 7 可得本算法的平均复杂度为：

$$O(\frac{n\cdot\log_a q}{q})$$

因为其常数因子是比较大的，所以在 θ 与 α 足够大时，该算法的优越性才显现出来。

§7　启示和总结

§7.1　算法分析

主算法二中使用了$\frac{q}{2}$这一个数，它汇聚了双重定义于一身：

- 是 ScanA 退出的标志。
- 是每一个阶段的 R-L 的长度下限。

如果$\frac{q}{2}$变大，那么 ScanA 的退出将变得更困难。

如果$\frac{q}{2}$变小，那么阶段的数量将会增加。

这两者都会使得平均复杂度变大！

因此$\frac{q}{2}$是一个恰好使得平均复杂度取得最小值的**中间点**！

§7.2　启示

我从该算法中得到的启示有两点：

- 多种算法并用（比如本题的单词前缀树和后缀树）
- 优劣得所思想（选取恰当的中间点）

本题中使用了**算术平均数**来选取**中间点**，其实更常见的是**几何平均数**。举一个典型的例子：

[例]（NOI2003 第一天第二题 Editor）　在一个长度最多为 2M 的字符数组中，不断进行插入、删除、取址、读取等操作，要求模拟这一过程。

[分析]

这道题如果使用数组储存，那么插入和删除的复杂度为 O(2M)，取址的复杂度为 O(1)。

如果使用链表储存，插入和删除的复杂度为 O(1)，取址的复杂度为 O(2M)。

可惜两种方法面临给定的数据量（各操作的次数限制）都是力不从心的，因此我们“**两种算法并用**”，使用一个块状链表，如**错误！链接无效。**。

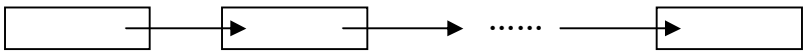

图 19

其中每一块是一个数组。设块的大小为 X，块的数量为 Y，那么理想情况下 $X \times Y$ 应该为正文的大小。这样插入、删除时，需要在个别块中进行琐碎的操作 O(X)，再将整个链表进行指针转移等操作 O(1)，复杂度为 O(X)；定位时需要先找到所在块的位置 O(Y)，再找到块中的具体位置 O(1)，复杂度为 O(Y)。

这时可以取 $X = Y = \sqrt{2M}$，找到“**中间点**”，使得算法总复杂度降到最低。

§7.3　总结

对于主算法二，程序的实现难度已远超出 OI 要求，况且读入字符数组的时间复杂度已经超过处理复杂度，那么有了简单易行的主算法一，该算法就没有实际意义了吗？

不！首先该算法使用的几个预备算法，都已在信息学竞赛中成为主流，熟练掌握这些算法是信息人必备的素质。其次，主算法二的时间复杂度在另一个程度上反映出，硬件和软件之间矛盾的天平，随着历史巨轮的航行，再一次倾向了软件一边，因此，更高效的储存介质和储存算法等待着人们的进一步开发。

最最令人拍案叫绝的是算法构思中融入的“综合使用、从最弱的点打开去”的思想。一条铁链的强度，决定于最弱的铁环的强度；一个水桶的水量，决定于最短的竹片的长度。只要不断地从瓶颈处突破，解题将会“有山就有路，有河就能渡”！

信息学是一门博大精深的学科，其本质不在于低层次的程序设计技术，而在于人的思维创新，这也是学科竞赛的宗旨。

[感谢]

感谢江苏省青少年信息学(计算机)奥赛委员会教练组全体老师对我的指导。
感谢我的母校江苏省南京市外国语学校的老师给我的支持。
感谢辽宁省的辛韬同学对我的大力帮助。
感谢我的家人给我的关怀。
感谢所有的信息学好友!

[参考书目]

i. 《Introduction to Algorithms》
by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein
ii. 《The Art of Computer Programming》
by Donald E. Knuth
iii. 《The Computer Science and Engineering Handbook》
by Allen B. Tucker, etc.
iv. 《On-line Construction of Suffix Trees》
by E. Ukkonen 1995
v. 《McCreight's Suffix Tree Construction Algorithm》
by Barbara König
vi. 《实用算法的分析和程序设计》
by 吴文虎 王建德

优化，再优化！

——从《鹰蛋》一题浅析对动态规划算法的优化

安徽省芜湖市第一中学 朱晨光

目录

关键字

优化　动态规划　模型

摘要

本文就 Ural 1223 《鹰蛋》这道题目介绍了五种性能各异的算法，并在此基础上总结了优化动态规划算法的本质思想及其一般方法。全文可以分为四个部分。
第一部分引言，阐明优化动态规划算法的重要性；
第二部分给出本文讨论的题目；
第三部分详细讨论这道题目五种不同的动态规划算法，并阐述其中的优化思想；
第四部分总结全文，再次说明对于动态规划进行优化的重要性，并分析优化动态规划的本质思想与一般方法。

正文

引言

在当今的信息学竞赛中，动态规划可以说是一种十分常用的算法。它以其高效性受到大家的青睐。然而，动态规划算法有时也会遇到时间复杂度过高的问题。因此，要想真正用好用活动态规划，对于它的优化方法也是一定要掌握的。

优化动态规划的方法有许多，例如四边形不等式、斜率优化等。但是这些方法只能对某些特定的动态规划算法进行优化，尚不具有普遍的意义。本文将就《鹰蛋》这道题目做较为深入的分析，并从中探讨优化动态规划的本质思想与一般方法。

问题

有一堆共 M 个鹰蛋，一位教授想研究这些鹰蛋的坚硬度 E。他是通过不断从一幢 N 层的楼上向下扔鹰蛋来确定 E 的。当鹰蛋从第 E 层楼及以下楼层落下时是不会碎的，但从第（E+1）层楼及以上楼层向下落时会摔碎。如果鹰蛋未摔碎，还可以继续使用；但如果鹰蛋全碎了却仍未确定 E，这显然是一个失败的实验。教授希望实验是成功的。

例如：若鹰蛋从第 1 层楼落下即摔碎，E=0；若鹰蛋从第 N 层楼落下仍未碎，E=N。

这里假设所有的鹰蛋都具有相同的坚硬度。给定鹰蛋个数 M 与楼层数 N。要求最坏情况下确定 E 所需要的最少次数。

样例：

M=1，N=10

ANS=10

样例解释：为了不使实验失败，只能将这个鹰蛋按照从一楼到十楼的顺序依次扔下。一旦在第（E+1）层楼摔碎，E 便确定了。（假设在第（N+1）层摔鹰蛋会碎）

分析

算法一

乍一看这道题，算法并不十分明晰，因为这并不是简单的二分查找，还有对鹰蛋个数的限制。但由于这题是求最优值，我们便自然想到了动态规划。

状态定义即为用 i 个蛋在 j 层楼上最坏情况下确定 E 所需要的最少次数，记为 f(i,j)。

很显然，当层数为 0 时 f(i,j)=0，即 f(i,0)=0 (i>=0); 当鹰蛋个数为 1 时，为了不使实验失败，只能从下往上依次扔这个鹰蛋以确定 E，即 f(1,j)=j (j>=0).

下面是状态转移：

假设我们在第 w 层扔下鹰蛋，无外乎有两种结果：

①鹰蛋摔碎了，此时必有 E<w,我们便只能用（i-1）个蛋在下面的（w-1)层确定 E，并且要求最坏情况下次数最少，这是一个子问题，答案为 f(i-1,w-1),总次数便为 f(i-1,w-1)+1；

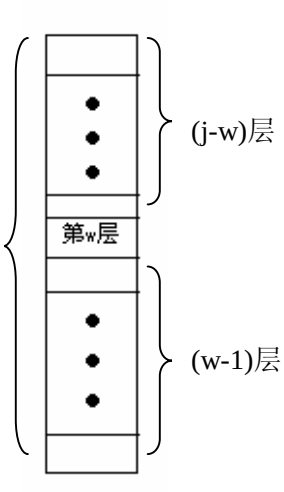

图 1

②鹰蛋没摔碎，此时必有 E>=w .我们还能用这 i 个蛋在上面的（j-w)层确定 E。注意，这里的实验与在第 1~(j-w)层确定 E 所需次数是一样的，因为它们的实验方法与步骤都是相同的，只不过这(j-w)层在上面罢了。完全可以把它看成是对第 1~(j-w)层进行的操作。因此答案为 f(i,j-w),总次数便为 f(i,j-w)+1。（如图 1）

题目要求最坏情况下的最小值，所以这两种情况的答案须取较大值，且又要在所有决策中取最小值，所以有

$f(i,j)=\min\{\max\{f(i-1,w-1),f(i,j-w)\}+1 \mid 1<=w<=j\}$　　①

很显然，所需鹰蛋个数必不会大于 N，因此当 M>N 时可令 M=N，且并不影响结果。所以这个算法的时间复杂度是 $O(MN^2)=O(N^3)$,空间复杂度是 O(N)（可用滚动数组优化）。

算法二

这个算法的时间复杂度太高，有没有可以优化的余地呢？答案是肯定的。首先，这题很类似于二分查找，即每次根据扔鹰蛋所得信息来决定下一步操作的区间，只不过对鹰蛋碎的次数有限制罢了。假设我们对于鹰蛋的个数不加限制，那么根据判定树的理论[1]，叶子结点个数共有(n+1)个，（E 的取值只可能是 0,1,2,…,n 共（n+1）种情况），则树的高度至少为$\lceil \log_2(n+1) \rceil$+1,即比较次数在最坏情况下需要$\lceil \log_2(n+1) \rceil$次。而我们又知道，在 n 个排好序的数里进行二分查找最坏情况下需要比较$\lceil \log_2(n+1) \rceil$次（在这个问题中，若未查找到可视为 E=0）。这两点便决定了$\lceil \log_2(n+1) \rceil$是下限而且是可以达到的下限。换句话说，对于一个确定的 n，任意 M 所得到的结果均不会小于$\lceil \log_2(n+1) \rceil$。一旦 M>=$\lceil \log_2(n+1) \rceil$,该题就成了求二分查找在最坏情况下的比较次数，可以直接输出$\lceil \log_2(n+1) \rceil$。因此时间复杂度立即降为 $O(N^2 \log_2 N)$.

由此可见，对于动态规划的优化是十分必要的。算法二仅通过考察问题自身的性质便成功地减少了状态总数，从而降低了算法一的时间复杂度，大大提高了算法效率。

算法三

然而优化还远未结束。经实践证明，M 的大小已经不能再加限制了，因此我们不妨从动态规划方程本身入手，探寻新的优化手段。

在实际操作中，我们发现 f(i,j)>=f(i,j-1) (j>=1)总是成立的。那么是否可以对该性质进行证明呢？是的。这里用数学归纳法加以证明。

当 i=1 时，由于 f(1,j)=j (j>=0),显然有 f(i,j)>=f(i,j-1)(j>=1)成立。
当 i>=2 时[2],f(i,0)=0,f(i,1)=max{f(i-1,0),f(i,0)}+1=1,即当 j=1 时，f(i,j)>=f(i,j-1)
现在假设当 j=k-1 时,f(i,j)>=-f(i,j-1)成立(k>1) ,则当 j=k 时,
f(i,j)=f(i,k)=min{max{f(i-1,w-1),f(i,k-w)}+1|1<=w<=k}
f(i,j-1)=f(i,k-1)=min{max{f(i-1,w-1),f(i,k-1-w)}+1|1<=w<=k-1}
当 1<=w<=k-1 时,∵k-1-w<k-w<=k-1,根据归纳假设，有 f(i,k-w)>=f(i,k-1-w),
∴max{f(i-1,w-1),f(i,k-w)}+1>=max{f(i-1,w-1),f(i,k-1-w)}+1

[1] 有关判定树的理论详情请见参考文献[1]第 292~293 页。
2 这里，我们按照 i 依次增大的顺序进行证明，即先证明 i =2 的情况，再证明 i=3 的情况，……，依此类推。

令 t=min{ max{f(i-1,w-1),f(i,k-w)}+1|1<=w<=k-1},
则有 t>=min{max{f(i-1,w-1),f(i,k-1-w)}+1|1<=w<=k-1}=f(i,k-1)　　(1)
又 f(i,k)=min{t,max{f(i-1,k-1),f(i,0)}+1}=min{t,max{f(i-1,k-1),0}+1}
∵对于 1<=i<=n,0<=j<=m,f(i,j)>=0
∴f(i,k)=min{t,f(i-1,k-1)+1}
在(1)式中,令 w=k-1,
=>f(i,k-1)<=max{f(i-1,k-2),f(i,0)}+1=f(i-1,k-2)+1<=f(i-1,k-1)+1
即 f(i-1,k-1)+1>=f(i,k-1)，又 t>=f(i,k-1) ,
∴f(i,k)=max{t,f(i-1,k-1)+1}>=f(i,k-1)
即 f(i,j)>=f(i,j-1),命题得证。
所以有 f(i,j)>=f(i,j-1)　(j>=1)　②
由此结论，可以在状态转移中使用二分法：

i) 若 f(i-1,w-1)<f(i,j-w),则对于 w'<w,必有 f(i,j-w')>=f(i,j-w)
∴max{f(i-1,w'-1),f(i,j-w')}+1>=f(i,j-w')+1>=f(i,j-w)+1=max{f(i-1,w-1),f(i,j-w)}+1
∴决策为 w'必无决策为 w 好；

ii) 若 f(i-1,w-1)>=f(i,j-w),则对于 w'>w,必有 f(i-1,w'-1)>=f(i-1,w-1),
∴max{f(i-1,w'-1),f(j,j-w')}+1>=f(i-1,w'-1)+1>=f(i-1,w-1)+1=max{f(i-1,w-1),f(i,j-w)}+1
∴决策为 w'必无决策为 w 好；

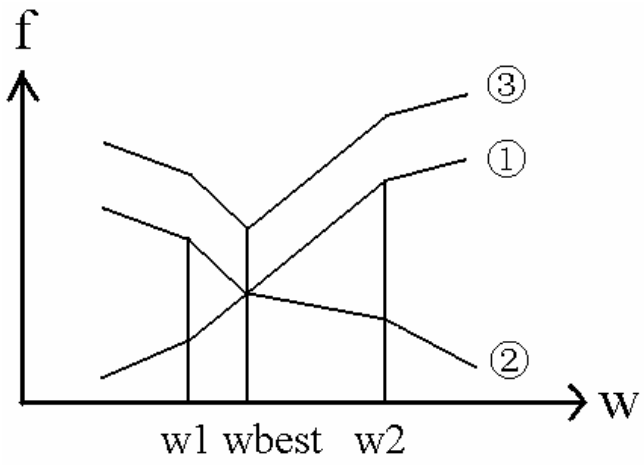

图 2

（如图 2，令①为 f(i-1,w-1)图象，②为 f(i,j-w)图象（皆用线段连结相邻两点），③即为 max{f(i-1,w-1),f(i,j-w)}+1 的图象）

可见 w1 对应情况 i),w2 对应情况 ii)，最优取值即为 w=wbest 时的取值.

因此，在状态转移中，可以像二分查找那样，每次将 w 的取值范围缩小一半，这样就能在$\lceil \log_2(n+1) \rceil$次之内找到最佳的决策 wbest。

这样，根据 f(i,j)的单调性，我们又成功地将状态转移降为 $O(\log_2 N)$级，从而

使算法的时间复杂度降至 $O(N\log_2^2 N)$级，已经可以解决 N 较大时的情况了。

从算法二到算法三的优化利用了动态规划函数 f(i,j)的单调性，成功地降低了状态转移部分的时间复杂度，使算法的效率又提高了一步。这说明研究算法本身同样可以找到优化的空间，为我们找到更低阶的算法创造了条件。

算法四

在对算法三进行研究之后，我们会萌生一个想法：既然现在 f(i,j)都需要求出，要想找到更高效的算法就只能从状态转移入手，因为这一步是 $O(\log_2 N)$，仍然不够理想。

这一步优化需要我们进一步挖掘状态转移方程：

f(i,j)=min{max{f(i-1,w-1),f(i,j-w)}+1|1<=w<=j}

很显然，f(i,j)<=max{f(i-1,w-1),f(i,j-w)}+1 (1<=w<=j)

令 w=1，则 f(i,j)<=max{f(i-1,0),f(i,j-1)}+1=f(i,j-1)+1

即 f(i,j)<=f(i,j-1)+1(j>=1)　③

又据②式，有 f(i,j-1)<=f(i,j)<=f(i,j-1)+1 (j>=1)④

这是一个相当重要的结论，由此可以得到一个推理:

若某个决策 w 可使 f(i,j)=f(i,j-1),则 f(i,j)=f(i,j-1)

若所有决策 w 都不能使 f(i,j)=f(i,j-1),则 f(i,j)=f(i,j-1)+1（且必存在这样的 w 使 f(i,j)=f(i,j-1)+1）

由此，我们设一指针 p,使 p 始终满足：

f(i,p)<f(i,j-1)且 f(i,p+1)=f(i,j-1)

很显然，f(i,p)=f(i,j-1)-1,

f(i,p+1)=f(i,p+2)=……=f(i,j-1)　(2)

在计算 f(i,j)时,我们令 p=j-w,则 w=j-p

令 tmp=max{f(i-1,w-1),f(i,j-w)}+1

则 tmp=max{f(i-1,j-p-1),f(i,p)}+1

若 f(i,p)>=f(i-1,j-p-1),则 tmp=f(i,p)+1=f(i,j-1)-1+1=f(i,j-1)

这说明当前决策 w 可以使 f(i,j)=f(i,j-1), ∴f(i,j)=f(i,j-1)

若 f(i,p)<f(i-1,j-p-1),则：

i) 当 p'<p 时,必有 f(i-1,j-p'-1)>=f(i-1,j-p-1)>f(i,p),

∴max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i-1,j-p'-1)+1>=f(i-1,j-p-1)+1>f(i,p)+1=f(i,j-1)

即 max{f(i,p'),f(i-1,j-p'-1)}+1>f(i,j-1)

max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i,j-1)+1,无法使 f(i,j)=f(i,j-1)

ii) 当 p'=p 时,

max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i-1,j-p'-1)+1>f(i,p')+1=f(i,p)+1=f(i,j-1)

同样无法使 f(i,j)=f(i,j-1)

iii) 当 p'>p 时,必有 f(i,p')>f(i,p),此时 f(i,p')=f(i,j-1)　(据(2)式)
所以 max{f(i,p'),f(i-1,j-p'-1)}+1>=f(i,p')+1=f(i,j-1)+1,还是无法使 f(i,j)=f(i,j-1)

综上所述，当 f(i,p)<f(i-1,j-p-1)时，无论任何决策都不能使 f(i,j)=f(i,j-1),所以此时 f(i,j)=f(i,j-1)+1。

因此，我们只需根据 f(i,p)与 f(i-1,j-p-1)的大小关系便可直接确定 f(i,j)的取值[3],使状态转移成功地降为 O(1),算法的时间复杂度随之降至 $O(N\log_2 N)$。

从算法三到算法四，我们是从尚未完全优化的部分（即状态转移）入手，通过进一步挖掘动态规划方程中的特性，恰当地找到了一个可以用来求 f(i,j)的剖分点 p，使得状态转移部分的时间复杂度降为 O(1)，最终将算法的效率又提高了一步。

小结

从算法一到算法四，我们一共进行了三步优化，让我们先来小结一下：

算法一建立动态规划模型，这也是后面几个算法进行优化的基础；

算法二将动态规划中 M 的取值限定在$\lceil \log_2(n+1) \rceil$以内，这样就从状态总数方面优化了这个动态规划算法；

算法三利用 f(i,j)的单调性，改进了动态规划中的状态转移部分，提高了算法效率；

算法四挖掘出 f(i,j)所具备的另一个特殊性质，让状态转移部分的时间复杂度变为 O(1),把原来算法中不尽人意的地方进行了进一步的优化与改进。

这时我们会发现，经过了数次优化的动态规划模型已经不可能再有所改进了，对这题的讨论似乎可以到此为止了。但是，经过进一步思考，我们又找到了另一种动态规划模型，在这种模型下的算法五，可以将时间复杂度降为 $O(\sqrt{N})$。让我们来看一看算法五的精彩表现吧！

算法五

这里，我们需要定义一个新的动态规划函数 g(i,j)，它表示用 j 个蛋尝试 i 次在最坏情况下能确定 E 的最高楼层数。下面具体讨论 g(i,j).

很显然，无论有多少鹰蛋，若只试 1 次就只能确定一层楼，即 g(1,j)=1 (j>=1)

[3] 有两点注意事项：
①f(i,1)需特殊处理，即令 f(i,1)=1,因为 p 初值为 0，并不满足 f(i,p)=f(i,j-1)-1
②若 f(i,j)=f(i,j-1)+1,则将 p 赋值为 j-1

而且只用 1 个鹰蛋试 i 次在最坏情况下可在 i 层楼中确定 E，即 g(i,1)=i (i>=1)

状态转移也十分简单，假设第一次在某一层楼扔下一只鹰蛋，且碎了，则在后面的(i-1)次里，我们要用(j-1)个蛋在下面的楼层中确定 E。为了使 g(i,j)达到最大，我们当然希望下面的楼层数达到最多，这便是一个子问题，答案为 g(i-1,j-1); 假设第一次摔鹰蛋没碎，则在后面(i-1)次里，我们要用 j 个蛋在上面的楼层中确定 E，这同样需要楼层数达到最多，便为 g(i-1,j) (见图 3)。

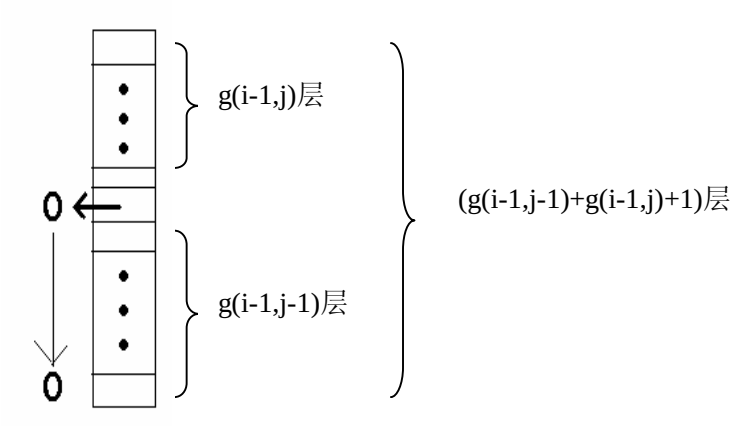

图 3

因此，有如下等式成立：

g(i,j)=g(i-1,j-1)+g(i-1,j)+1 ⑤

我们的目标便是找到一个 x，使 x 满足 g(x-1,M)<N 且 g(x,M)>=N ⑥，答案即为 x. [4]

这个算法乍一看是 $O(N\log_2 N)$的，因为 i 是次数，最大为 N；j 为鹰蛋数，最大为 M, 即 $\log_2 N$，状态转移为 O(1)，所以时间复杂度与状态数同阶，为 i*j，即 $O(N\log_2 N)$.但实际情况却并非如此，下面予以证明。

经过观察，我们很快会发现，函数 g(i,j)与组合函数 C_i^j 有着惊人的相似之处，让我们来比较一下（见下表）：

g(i,j)	C_i^j
g(1,j)=1(j>=1)	C_1^1=1, C_1^j=0 (j>=2)
g(i,1)=i(i>=1)	C_i^1=i (i>=1)
g(i,j)=g(i-1,j-1)+g(i-1,j)+1	$C_i^j = C_{i-1}^{j-1} + C_{i-1}^j$ (j<=i) , C_i^j=0 (j>i)

4 若 x=1,须特殊判断

根据边界条件与递推公式，我们可以很容易用数学归纳法证明对于任意 i,j (i>=1,j>=1) 总有 g(i,j)>= C_i^j

又据⑥式，可以得到

C_{x-1}^{M} <=g(x-1,M)<N

即 C_{x-1}^{M} <N, $\frac{(x-1)(x-2)...(x-M)}{M!} < N$ (3)

这里介绍一个引理：当 1<=N<=3 或 N>=17 时，$(\log_2 N)^2$<N （用函数图象可以证明）

注：当 4<=n<=16 时，$(\log_2 N)^2$>=N，但由于相差很小，并不影响对于渐近复杂度[5]的分析

若 x<M，则 $xM<M^2<=(\log_2 N)^2<N$

若 x>=M，则根据(3)式,有$(x-M)^M<=(x-1)(x-2)...(x-M)<N \cdot M!$

$x-M<=\sqrt[M]{NM!}<=\sqrt[M]{NM^M}=M\sqrt[M]{N}$

$\therefore x<=M+M\sqrt[M]{N}=M(1+\sqrt[M]{N})$

$\therefore xM<=M^2(1+\sqrt[M]{N})$ (4)

又$\because N^{\frac{M-1}{M}} \approx N$,且 $N>(\log_2 N)^2>=M^2$ (引理)

$\therefore N^{\frac{M-1}{M}}>=M^2$ (此性质对于 M=1 仍成立)

$\log_2 N^{\frac{M-1}{M}}>=\log_2 M^2$

$\frac{M-1}{M}\log_2 N>=\log_2 M^2$

$\log_2 N-\frac{1}{M}\log_2 N>=\log_2 M^2$

$\frac{1}{M}\log_2 N+\log_2 M^2<=\log_2 N$

$\log_2 N^{\frac{1}{M}}+\log_2 M^2<=\log_2 N$

又 $\log_2 N^{\frac{1}{M}} \approx \log_2 (N^{\frac{1}{M}}+1)$

$\therefore \log_2 (N^{\frac{1}{M}}+1)+\log_2 M^2<=\log_2 N$

$\log_2[(N^{\frac{1}{M}}+1)M^2]<=\log_2 N$

5 关于渐近复杂度的理论，详情请见参考文献[3]第 41~49 页。

$\therefore (N^{\frac{1}{M}}+1)$ • M^2<= N,即 $M^2(1+\sqrt[M]{N})$ <=N

又据(4)式，得 xM<= $M^2(1+\sqrt[M]{N})$<=N

综上所述，xM<=N

这就说明了 g(i,j)函数的实际运算量是 O(N)级的，那么又是怎样变为 O($\sqrt{N}$)的呢？

观察 $\frac{(x-1)(x-2)...(x-M)}{M!} < N$，可得 (x-1)(x-2)…(x-M)<NM!

这可以大致得出当 M 不太大时，x 与 $\sqrt[M]{N}$ 是同阶的。在实际情况中，可以发现只有当 M=1 时，x=N, xM=N ;当 M>1 时，xM 立即降至($\sqrt{N}$)的级别。因此，只需要在 M=1 时特殊判断一下就可以使算法的时间复杂度降为 O($\sqrt{N}$)了，空间复杂度可用滚动数组降为 O(M),即 O($\log_2 N$)。

在新的动态规划模型之下，我们找到了一个比前几种算法都优秀得多的方法。这就提醒我们不要总是拘泥于旧的思路。换个角度来审视问题，往往能收到奇效。倘若我们仅满足于算法四，就不能打开思路，找到更高效的解题方法。可见多角度地看问题对于动态规划的优化也是十分重要的。

总结

本文就《鹰蛋》一题谈了五种性能各异的算法，这里做一比较：（见下表）

算法编号	时间复杂度	空间复杂度	优化方法
算法一	$O(N^3)$	O(N)	
算法二	$O(N^2\log_2 N)$	O(N)	考察问题自身的性质，减少状态总数
算法三	$O(N\log_2^2 N)$	O(N)	研究动态规划方程，找出其中的特性，优化状态转移部分
算法四	$O(N\log_2 N)$	O(N)	进一步挖掘动态规划方程的特性，从而再次降低状态转移部分的时间复杂度
算法五	O($\sqrt{N}$)	$O(\log_2 N)$	建立新的动态规划模型，从另一个角度重新审视问题

从这张表格中，我们可以很明显地看出优化能显著提高动态规划算法的效率。并且，优化动态规划的方法也是多种多样的。这就要求我们在研究问题时必须深入探讨，大胆创新，永不满足，不断改进，只有这样才能真正将优化落到实

处。在实际问题中，尽管优化手段千变万化，但万变不离其宗，其本质思想都是找到动态规划算法中仍然不够完美的部分，进行进一步的改进；或是另辟蹊径，建立新的模型，从而得到更高效的算法。而在具体的优化过程中，需要我们从减少状态总数、挖掘动态规划方程的特性、降低状态转移部分的时间复杂度以及建立新模型等几方面入手，不断完善已知算法。这便是优化动态规划算法的一般方法。

当然，世上的任何事物都是既有普遍性，又有特殊性的。当我们用一般方法难以解决的时候，使用特殊方法（如四边形不等式、斜率优化等）也是不错的选择。因此，只有将一般方法与特殊方法都灵活掌握，才能真正高效地解决动态规划的优化问题。

结束语

本文仅是讨论了对于动态规划算法进行优化的本质思想及一般方法，实际上，优化思想极其重要，也无处不在。优化可以使原本效率低下的算法变为一个非常优秀的算法，可以使能够解决的问题规模扩大几十倍，几百倍，乃至成千上万倍。而更重要的是，优化思想的应用是极为广泛的。无论是在信息学竞赛中，还是在日常的生产生活中，优化思想都发挥着十分重要的作用。具备了优化思想，我们就不会满足于现有的方法，而会不断地开拓、创新，去创造出更好、更优秀的方法。

优化，再优化，就是为了让算法得到进一步的完善，同时也开阔了我们的思维，锻炼了我们深入分析研究问题的能力，培养了我们不断进取的精神，对今后进一步的科学研究也是大有益处的。

本文仅对一道信息学竞赛的题目做了粗浅的分析，希望能够起到抛砖引玉的作用，让我们进一步了解优化思想的重要作用，用好用活优化思想，以便更好、更高效地解决问题。

参考文献

[1] 严蔚敏 吴伟民 ，1992，《数据结构（第二版）》。 北京：清华大学出版社。

[2] 吴文虎 王建德，1997，《信息学奥林匹克竞赛指导——组合数学的算法与程序设计(PASCAL 版)》。 北京：清华大学出版社。

[3] Thomas H.Cormen Charles E.Leiserson Ronald L.Rivest Clifford Stein, 2001 , 《Introduction to Algorithms, Second Edition》. The MIT Press.

[4] Ural Online Judge acm.timus.ru

原题网页 http://acm.timus.ru/problem.aspx?space=1&num=1223

（本文为了讨论方便，对题目某些部分进行了改动）

附录

1、本文讨论原题：

Chernobyl' Eagle on a Roof

Time Limit: 1.0 second

Memory Limit: 1 000 КБ

Once upon a time an Eagle made a nest on the roof of a very large building. Time went by and some eggs appeared in the nest. There was a sunny day, and Niels Bohr was walking on the roof He suddenly said: "Oops! All eggs surely have the same solidity, thus there is such non-negative number *E* that if one drops an egg from the floor number *E*, it will not be broken (and so for all the floors below the *E*-th), but if one drops it from the floor number *E*+1, the egg will be broken (and the same for every floor higher, than the *E*-th). Now Professor Bohr is going to organize a series of experiments (i.e. drops). The goal of the experiments is to determine the constant *E*. It is evident that number *E* may be found by dropping eggs sequentially floor by floor from the lowest one. But there are other strategies to find *E* for sure with much less amount of experiments. You are to find the least number of eggs droppings, which is sufficient to find number *E* for sure, even in the worst case. Note that dropped eggs that are not broken can be used again in following experiments.

The number of floors is a positive integer and a number *E* is a non-negative one. They both do not exceed 1000. The floors are numbered with positive integers starting from 1. If an egg hasn't been broken even being dropped from the highest floor, one can suppose, that *E* is also determined and equal to the total number of floors.

Input

Input contains multiple test cases. Each line is a test case. Each test case consists of two numbers separated with a space: first the number of eggs, and then the number of floors. Tests will end with the line containing a single zero.

Output

For each test case output in a separate line the minimal number of experiments, which Niels Bohr will have to make even in the worst case.

Sample Input

1 10

2 5

0

Sample Output

10

3

2、相关程序

算法一　eagle_1.cpp　(由于时间复杂度过高，在 Ural Online Judge 上超时)
算法二　eagle_2.cpp
算法三　eagle_3.cpp
算法四　eagle_4.cpp
算法五　eagle_5.cpp
} 均在 Ural Online Judge 上测试通过

1）eagle_1.cpp

```
#include<iostream.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
  else return(b);
}

void work()
{
 long i,j,w,temp;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    {
```

```
    f[now][j]=maxnum;
    for (w=1; w<=j; w++)
     {
      temp=max(f[old][w-1],f[now][j-w])+1;
      if (temp<f[now][j])
       f[now][j]=temp;
     }
   }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
{
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   init();
   work();
   output();
  }
 return 0;
}
```

2）eggle_2.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
```

```
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
  else return(b);
}

void work()
{
 long i,j,w,temp;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    {
     f[now][j]=maxnum;
     for (w=1; w<=j; w++)
      {
       temp=max(f[old][w-1],f[now][j-w])+1;
       if (temp<f[now][j])
        f[now][j]=temp;
      }
    }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
{
 long temp;
 while (1)
```

```
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         work();
         output();
        }
  }
 return 0;
}
```

3) eggle_3.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

long max(long a,long b)
{
 if (a>b)
  return(a);
  else return(b);
}

void calc(long i,long j)
{
```

```
 long w,temp,start,stop,mid;
 f[now][j]=maxnum;
 start=1; stop=j;
 while (start<=stop)
  {
   mid=(start+stop)/2;
   if (f[old][mid-1]>f[now][j-mid])
    {
     if (f[old][mid-1]+1<f[now][j])
      f[now][j]=f[old][mid-1]+1;
     stop=mid-1;
    }
    else if (f[old][mid-1]<f[now][j-mid])
         {
          if (f[now][j-mid]+1<f[now][j])
           f[now][j]=f[now][j-mid]+1;
          start=mid+1;
         }
         else {
              f[now][j]=f[now][j-mid]+1;
              return;
             }
  }
}

void work()
{
 long i,j;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
   now=1-now;
   f[now][0]=0;
   for (j=1; j<=n; j++)
    calc(i,j);
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
```

```
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         work();
         output();
        }
  }
 return 0;
}
```

4) eggle_4.cpp

```
#include<iostream.h>
#include<math.h>
#define maxn 1100
#define maxnum 1000000000

long n,eggnum,now,old,f[2][maxn+1];

void init()
{
 long i;
 now=1;
 f[now][0]=0;
 for (i=1; i<=n; i++)
  f[now][i]=i;
}

void work()
{
 long i,j,p;
 for (i=2; i<=eggnum; i++)
  {
   old=now;
```

```
   now=1-now;
   p=f[now][0]=0;
   f[now][1]=1; //special!!! In case of mistake
   for (j=2; j<=n; j++)
    if (f[now][p]>=f[old][j-p-1])
     f[now][j]=f[now][j-1];
     else {
          f[now][j]=f[now][j-1]+1;
          p=j-1;
         }
  }
}

void output()
{
 cout<<f[now][n]<<endl;
}

int main()
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         work();
         output();
        }
  }
 return 0;
}
```

5) eggle_5.cpp

```
#include<iostream.h>
#include<math.h>
#define maxlogn 20
```

```
#define maxnum 1000000000

long n,eggnum,now,old,g[maxlogn+1];

void init()
{
 long i;
 now=1;
 for (i=1; i<=eggnum; i++)
  g[i]=1;
}

void work()
{
 long i,j,p;
 for (i=2; i<=n; i++)
  {
   for (j=eggnum; j>=2; j--)
    {
     g[j]=g[j-1]+g[j]+1;
     if ((j==eggnum)&&(g[j]>=n))
      {
       cout<<i<<endl;
       return;
      }
    }
   g[1]=i;
   if ((eggnum==1)&&(g[1]>=n))
    {
     cout<<i<<endl;
     return;
    }
  }
}

int main()
{
 long temp;
 while (1)
  {
   cin>>eggnum;
   if (eggnum==0)
    break;
    else cin>>n;
```

```
   temp=long (floor(log(n+0.0)/log(2.0))+1.0);
   if (eggnum>=temp)
    cout<<temp<<endl;
    else {
         init();
         if(g[eggnum]>=n)
          cout<<1<<endl;
          else if (eggnum==1)
               cout<<n<<endl;
               else work();
        }
  }
 return 0;
}
```

浅谈数形结合思想在信息学竞赛中的应用

安徽省芜湖一中　周源

目录

摘要

数与形是数学中两个最古老而又最基本的对象，数形结合又是一种重要的数学思想。

本文主要以当今信息学奥赛中几道试题为例，从以形助数和以数助形两个侧重点讨论了数形结合思想在信息学竞赛解题中广阔的应用前景。

最后文章分析指出数形结合思想的两个重要特性并由此提出“数形结合”重在有机的结合，希望对同学们在实际比赛中灵活的运用数形结合思想有一些帮助。

关键字

信息学竞赛 数学思想 数形结合思想

以数助形 以形助数

辩证矛盾 多元性 个体差异性

思维、编程、时间、空间复杂度

引子

数与形是数学中两个最古老而又最基本的对象，数形结合又是一种重要的数学思想。

在当今信息学竞赛中，某些纷繁复杂的试题背后，往往蕴含着丰富的几何背景，而计算几何类问题却又需要借助计算机强大的实数运算能力。正如华罗庚先生所说的“数形结合千般好”，在算法和程序设计中，巧妙地运用数形结合思想，可以顺利的破解问题，化难为易，找到问题的解题思路。

数形结合思想常包括以形助数、以数助形两个方面。

以形助数

正如前文所述，一些试题中繁杂的代数关系身后往往隐藏着丰富的几何背景，而借助背景图形的性质，可以使那些原本复杂的数量关系和抽象的概念，显得直观，从而找到设计算法的捷径。

[例一]Raney 引理的证明

[题意简述]

设整数序列 $A = \{A_i, i=1, 2, \ldots, N\}$，且部分和 $S_k=A_1+\ldots+A_k$，序列中所有的数字的和 $S_N=1$。

证明：在 A 的 N 个循环表示[1]中，有且仅有一个序列 B，满足 B 的任意部分和 S_i 均大于零。

[分析]

先来看一个例子，若有序列 A = <1, 4, -5, 3, -2, 0>，其 6 个循环表示为

1. <1, 4, -5, 3, -2, 0>
2. <4, -5, 3, -2, 0, 1>
3. <-5, 3, -2, 0, 1, 4>
4. <3, -2, 0, 1, 4, -5>
5. <-2, 0, 1, 4, -5, 3>
6. <0, 1, 4, -5, 3, -2>

其中只有第 4 个序列，部分和为 3, 1, 1, 2, 6, 1，满足成为序列 B 的条件。

若要用一般的代数或是组合方法来证明这个有趣的结论，似乎无从下手，但若要想到了用“形”来帮忙，问题就简单多了。

目标图形化

周期性的推广 A 序列，得到一个无穷序列，便于观察其循环表示，得到：

[1] 先设一个序列是环状的，则从其任意一个字符处断开以后形成的非环序列即为该序列的一个循环表示。

$<A_1, A_2, \ldots, A_N, A_1, A_2, \ldots, A_N, \ldots>$

同时计算这个序列的部分和 S_i，因为这个序列是周期性的，因此对于所有的 $k>0$，均有 $S_{k+N}=S_k+1$。如果做出这个函数的图像，则可以说函数有一个“平均斜率”为 $\frac{1}{N}$：每沿横轴正方向走 N 个单位，函数值就增加 1。于是如下图所示，可以用两条斜率为 $\frac{1}{N}$ 的直线“夹住”函数包含的所有点：

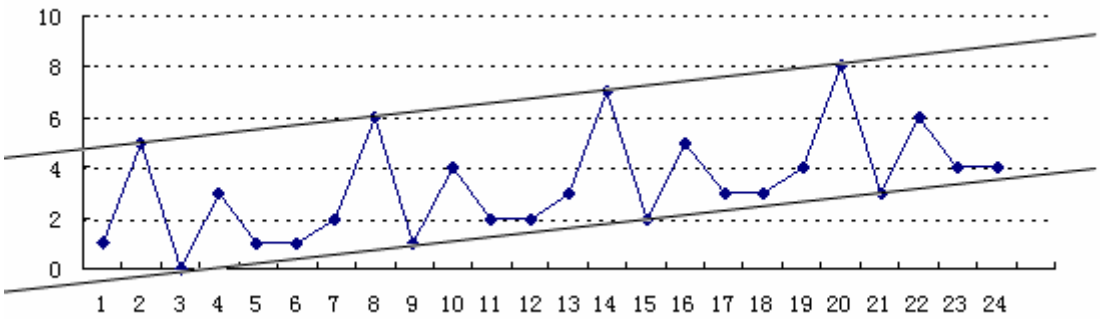

图 1 无穷序列的部分和函数图像

图示中 N=6，且使用了上文举的例子。注意较低的那条直线，在每连续的 N 个单位长度中，它与函数图像有且仅有一个交点，这是因为斜率是 $\frac{1}{N}$ 的直线在每 N 个单位长度中最多到达一次整数点。这个交点是在这以后的 N 个点中的最低值，因此由此处的后一个位置导出的循环表示的所有部分和均为正数。而同时每连续 N 个单位长度仅有一个交点也证明了解的唯一性。

小结

一个简单的几何论证就证明了著名的 Raney 引理，其简练是其他方法不能企及的。

Raney 引理有很广泛的应用，Catalan 数以及扩展 Catalan 数的组合公式就可以用该引理轻松解决。比如今年上海市选拔赛第二天比赛中的序列(Sequence)以及 OIBH 练习赛中的项链，使用 Raney 引理都是最简单的方法之一。[2]

用几何图形辅助思考，不只停留在组合计数这一类中，更渗透在算法设计和优化的每一个分支中，近年来流行的“斜率优化”法是另一个很好的例子。

[例二]最大平均值问题(USACO 2003 March Open)

[题意简述]

读入一列正数，$a_1, a_2, \ldots, a_N$，以及一个数 F。定义 $ave(i, j)=\frac{a_i+\ldots+a_j}{j-i+1}$，$i \leqslant j$。

[2] 用 Raney 引理解答 Sequence 的过程，详见附录。

求 Max{ave(a, b), 1≤a, b≤N，且 a≤b-F+1}，即求一段长度大于等于 F 且平均值最大的子串。

范围：F≤N≤10^5。

[分析]

简单的枚举算法可以这样描述：每次枚举一对满足条件的(a, b)，即 a≤b-F+1，检查 ave(a, b)，并更新当前最大值。

然而这题中 N 很大，N^2 的枚举算法显然不能使用，但是能不能优化一下这个效率不高的算法呢？答案是肯定的。

目标图形化

首先一定会设序列 a_i 的部分和：$S_i=a_1+a_2+\ldots+a_i$，特别的定义 $S_0=0$。

这样可以很简洁的表示出目标函数 $ave(i, j)=\frac{S_j-S_{i-1}}{j-(i-1)}$！

如果将 S 函数绘在平面直角坐标系内，这就是过点 S_j 和点 S_{i-1} 直线的斜率！

于是问题转化为：平面上已知 N+1 个点，$P_i(i, S_i)$，0≤i≤N，求横向距离大于等于 F 的任意两点连线的最大斜率。

构造下凸折线

有序化一下，规定对 i<j，只检查 P_j 向 P_i 的连线，对 P_i 不检查与 P_j 的连线。也就是说对任意一点，仅检查该点与在其前方的点的斜率。于是我们定义点 P_i 的检查集合为

$$G_i = \{P_j, 0 \leqslant j \leqslant i-F\}$$

特别的，当 i<F 时，G_i 为空集。

其明确的物理意义为：在平方级算法中，若要检查 ave(a, b)，那么一定有 P_a∈G_b；因此平方级的算法也可以这样描述，首先依次枚举 P_b 点，再枚举 P_a∈G_b，同时检查 k(P_aP_b)。

若将 P_i 和 G_i 同时列出，则不妨称 P_i 为检查点，G_i 中的元素都是 P_i 的被检查点。

当我们考察一个点 P_t 时，朴素的平方级算法依次选取 G_t 中的每一个被检查点 p，考察直线 pP_t 的斜率。但仔细观察，若集合内存在三个点 P_i, P_j, P_k，且 i<j<k，三个点形成如下图所示的的关系，即 P_j 点在直线 P_iP_k 的上凸部分：k(P_i, P_j)>k(P_j, P_k)，就很容易可以证明 P_j 点是多余的。

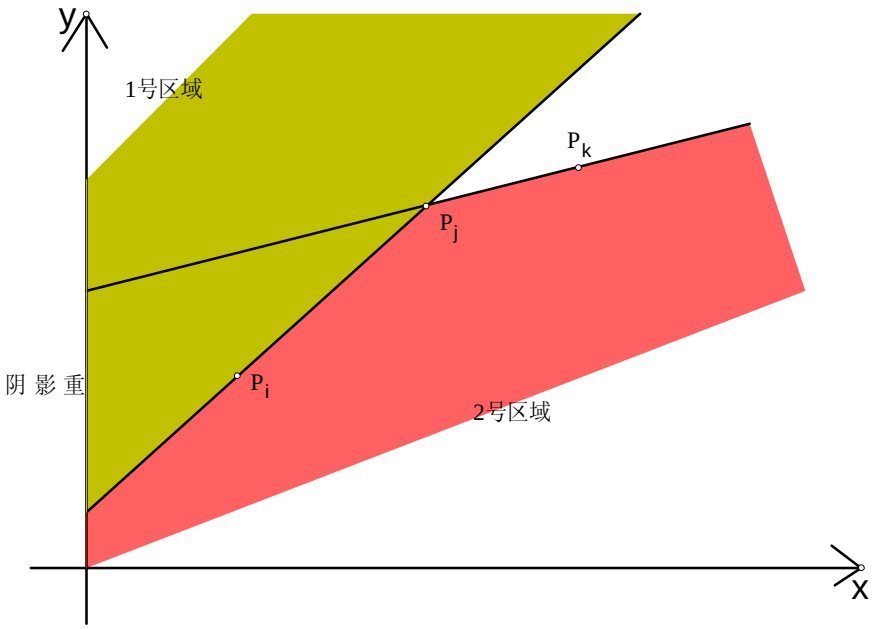

图 2

若 $k(P_t, P_j) > k(P_t, P_i)$，那么可以看出，$P_t$ 点一定要在直线 P_iP_j 的上方，即阴影所示的 1 号区域。同理若 $k(P_t, P_j) > k(P_t, P_k)$，那么 P_t 点一定要在直线 P_jP_k 的下方，即阴影所示的 2 号区域。

综合上述两种情况，若 P_tP_j 的斜率同时大于 P_tP_i 和 P_tP_k 的，P_t 点一定要落在两阴影的重叠部分，但这部分显然不满足开始时 $t>j$ 的假设。于是，P_t 落在任何一个合法的位置时，P_tP_j 的斜率要么小于 P_tP_i，要么小于 P_tP_k，即不可能成为最大值，因此 P_j 点多余，完全可以从检查集合中删去。

这个结论告诉我们，任何一个点 P_t 的检查集合中，不可能存在一个对最优结果有贡献的上凸点，因此我们可以删去每一个上凸点，剩下的则是一个下凸折线。最后需要在这个下凸折线上找一点与 P_t 点构成的直线斜率最大——显然这条直线是在与折线相切时斜率最大，如图所示。

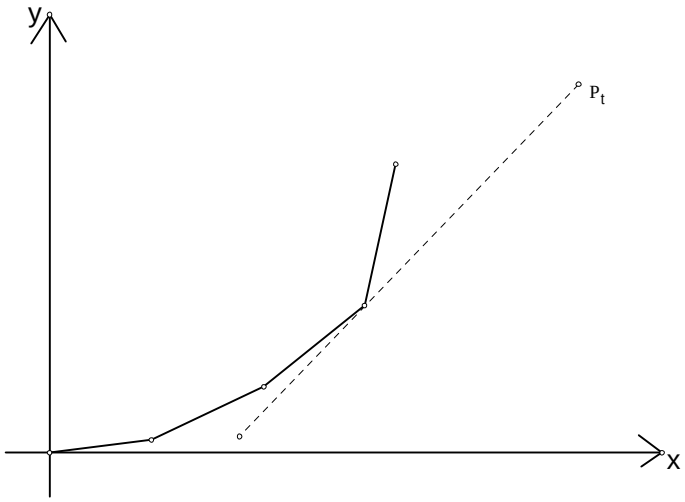

图 3

维护下凸折线

这一小节中，我们的目标是：用尽可能少的时间得到每一个检查点的下凸折

线。

算法首先从 P_F 开始执行：它是检查集合非空的最左边的一个点，集合内仅有一个元素 P_0，而这显然满足下凸折线的要求，接着向右不停的检查新的点：P_{F+1}, P_{F+2}, …, P_N。

检查的过程中，维护这个下凸折线：每检查一个新的点 P_t，就可以向折线最右端加入一个新的点 P_{t-F}，同时新点的加入可能会导致折线右端的一些点变成上凸点，我们用一个类似于构造凸包的过程依次删去这些上凸点，从而保证折线的下凸性。由于每个点仅被加入和删除一次，所以每次维护下凸折线的平摊复杂度为 O(1)，即我们用 O(*N*)的时间得到了每个检查集合的下凸折线。

最后的优化：利用图形的单调性

最后一个问题就是如何求过 P_t 点，且与折线相切的直线了。一种直接的方法就是二分，每次查找的复杂度是 $O(\log_2 N)$。但是从图形的性质上很容易得到另一种更简便更迅速的方法：由于折线上过每一个点切线的斜率都是一定的[3]，而且根据下凸函数斜率的单调性，如果在检查点 P_t 时找到了折线上的已知一个切点 A，那么 A 以前的所有点都可以删除了：过这些点的切线斜率一定小于已知最优解，不会做出更大的贡献了。

于是另外保留一个指针不回溯的向后移动以寻找切线斜率即可，平摊复杂度为为 O(1)。

至此，此题算法时空复杂度均为 O(*N*)，得到了圆满的解决。

小结

回顾本题的解题过程，一开始就确立了以平面几何为思考工具的正确路线，很快就发现了检查集合中对最优解有贡献的点构成一个下凸函数这个重要结论，之后借助计算几何中求凸包的方法维护一个下凸折线，最后还利用下凸函数斜率的单调性发现了找切线简单方法。题解围绕平面几何这个中心，以斜率为主线，整个解题过程一气呵成，又避免了令人头晕的代数式变换，堪称以形助数的经典例题。

顺便提一下：这种方法在加速决策过程，很多动态规划算法都可以运用本题“斜率优化”的方法提高算法效率。如 IOI 2002 的 batch 和 BOI 2003 的 euro 等。至于这类题目的共同特点，还是很值得研究的，但不在本文讨论范围内，因而不再讨论，但欢迎有兴趣的同学以后和我交流。

以数助形

古希腊的毕达哥拉斯认为“万物皆数”，的确，数是反映事物本质特征的最好方法之一。数学发展史上，正是在解析几何创立之后，人们才对各种繁杂的曲线有了更深入的了解。如今信息时代中，计算机处理各类事物，最终无不是归结于二进制数的基本运算，数的重要性可见一斑。

在当今信息学竞赛中，一些试题给出的描述中图形极为复杂，容易使选手陷入“迷魂阵”，在这种情况下，以数助形，一举抓住其本质特征，不失为解题的一种好方法。

[3] 由于折线没有连续性，因此更准确的应该说，过每一个点切线斜率的范围都一定的。

[例三]画室(POI oi V Stage I)

[题意简述]

定义尺寸为 0 的方阵为一个 1*1 的矩阵，在其唯一的一个方格中有一个小孔。

对于 $i>0$，递归的定义尺寸为 i 的方阵如下图所示：

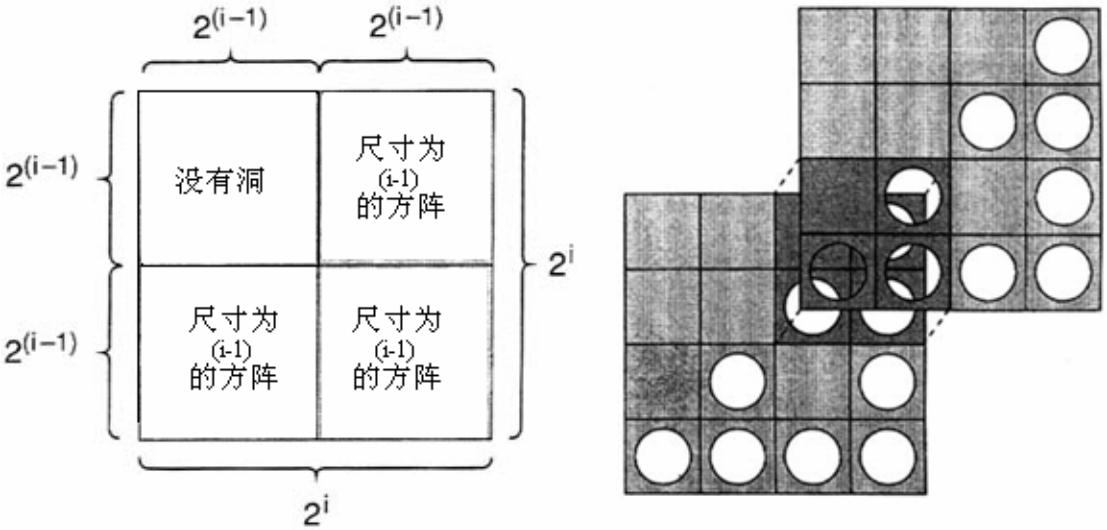

图 4

给定方阵的尺寸 N，以及另外两个参数 X 和 Y。准备两个尺寸为 N 的方阵，一个叠放在另一个上面，再将上面的方阵向右移动 X 列，同时向上移动 Y 行。

如此操作之后，求两个方阵有多少个公共的孔。

如右上图，尺寸为 2 的方阵，向右平移 2 列，向上平移 2 行。则两个方阵有 3 个公共小孔。

范围：$N\leqslant 100$。

[分析]

直接分析两个方阵相交后的情况是可行的，我曾经看过一些集训队前辈的解题报告，都是这么分析的，但是方法很繁，思考量很大。

下图是某解题报告中的一个说明附图，报告中先标出两个方阵的相交区域，再分情况讨论。显然可以看出，直接从“形”来分析本题，路子是很坎坷的。

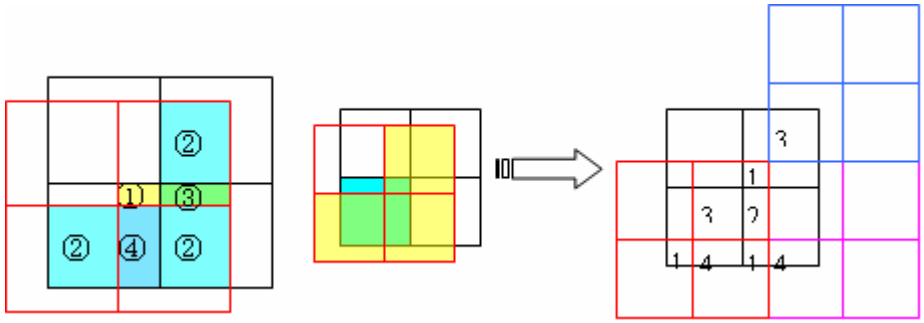

图 5

目标数值化

我们不如换至和“形”相对的另一面“数”来思考，按照下图所示的 x, y 方向为每行每列从 0 开始编号，最大至 2^N-1，于是每一个方格都有唯一的坐标(x, y)。

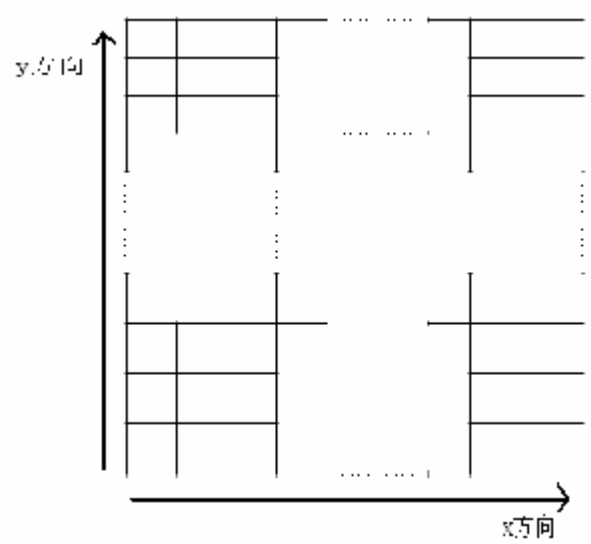

图 6

下面来研究一下在什么条件下，一个方格 P(x, y)内有小孔。由于方阵是二分递归定义的，于是我们很自然联想到将 x 和 y 化为二进制。设 x 和 y 的二进制表示分别为：

$$a_1a_2a_3\ldots a_N \text{ 和 } b_1b_2b_3\ldots b_N$$

来看两个数的第 1 位，a_1 和 b_1，如下图，它们一共有 4 种取值方法，其分布分别对应着递归定义中的左上、左下、右上、右下四块区域。显然当 a_1=0 且 b_1=1 时无论以后各位取什么数，P 点内都不会有小孔：因为其已经落在了左上无孔区。否则可以同理讨论两个数的第 2 位，第 3 位……

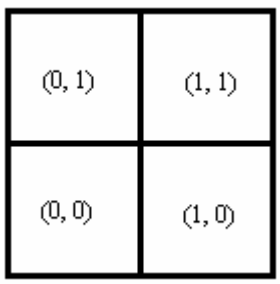

图 7 示意(a_1, b_1)的取值分布情况

得到的结论是，当且仅当不存在 $1\leqslant i\leqslant N$，满足 a_i=0 且 b_i=1 时，方格 P 内有小孔。不妨称这个为方格的**有孔性质**。

动态规划解题

后面的问题就非常简单了，题目要找的无非是这样的有序数对(x, y)的个数：$0\leqslant x, y$，$x+X, y+Y\leqslant 2^N-1$，且(x, y)，$(x+X, y+Y)$的二进制表示都满足有孔性质称这个为方格的**有公共孔性质**。

我们可以采用动态规划的方法：首先将 X, Y 也都转化成二进制形式：

$$p_1p_2p_3\ldots p_N \text{ 和 } q_1q_2q_3\ldots q_N$$

以位数为阶段，通过记录进位情况保证无后效性：$f(i, k1, k2)$表示第 i 位至第

N 位部分满足有公共孔性质的有序对总数，且要满足这一部分有序对的坐标和对应部分的 X, Y 相加进位分别是 $k1$ 和 $k2$：显然 $0 \leqslant k1, k2 \leqslant 1$。

动态规划的状态转移是非常简单的，但描述比较复杂：每一次转移需要约 $(2^2)^2=16$ 次运算，因此不再赘述，有兴趣的读者可以查看附件中的程序。

最后说明一下，题目所要的答案就是 $f(1, 0, 0)$。

算法若不算上高精度，时间复杂度为 O(N)，若使用循环数组，空间上仅需要常数个高精度数组[4]，而且实现程序也极为简单，包括高精度也不过 100 多行。对比从“形”上得出的算法，“数”的优越性是不言而喻的。

小结

回顾解题过程，当分析发现两方阵相交情况较复杂，不宜讨论时，我们决定避开“形”的正面冲突，而从“数”这方面下手，很快便取得了令人满意的效果：方格的有孔性质和有公共孔性质使题目的要求显得简单了许多。到此就可以套用经典的动态规划算法了。

可以说本题是一个较好的例子，但类似以数助形的例题似乎比较罕见。事实上，正如前文所述，一般的计算机都是以数为基础的，同学们在写各类程序的时候，最终还是要归结到“数”来实现，对数的重要作用多少有些熟视无睹了。

而从上例又可以看出，如果试题加以适当的“误导”，选手们背离“数”的捷径，南辕北辙也不是没有可能的。因此，在遇到如同上例的题目时，面对多元化的复杂图形，化形归数，往往是抓住题目要害的好方法。

总结

数与形是现实世界中客观事物的抽象和反映，是数学的基石，也是信息学竞赛命题涉及的两个主要方面。数形结合是一种古老的数学思想，新兴的信息学奥林匹克竞赛又赋予她新的活力。

上文举了三个实例，大体上来说，都巧妙的运用了数形结合思想。但从细节上分析，它们之间仍略有差异。

其一，三者从两个不同的侧重点阐述了数形结合思想的内涵，即以形助数和以数助形。但在实际问题中，数和形决没有明确的界限，数形结合思想也并不仅仅局限于文中提出的两个方面。更多的情况下，数与形互相促进、互相包含，在一定条件下互相转化，可以用“数形互助”一词来形容。

这，体现了数形的**辩证矛盾关系**和数形结合思想的**多元性**。

其二，用“形”来解例题二，似乎是唯一的出路，但在例一和例三中，并不是仅仅能用文中提到的方法解题，其他精彩解法我也略知一二。但相比而言，巧妙的使用数形结合思想会大大降低思考和编程复杂度，为我们在短短的竞赛时间中迅速解题开辟了一条便捷的道路。

需要指出的是，不同的人有不同的知识结构，比赛经验等，他们对某一算法难度系数的感觉也是不同的。因而对同一题而言，不同的人可能会选择不同的数形之路解题。这，体现了数形结合思想的**个体差异性**。

而本文提出的三个例子，都是选择了大多数人能够接受的算法，却并不能说是每位读者心目中最简单的算法。但醉翁之意不在酒，几个小例子仅作抛砖引玉，重点在于探讨如何在信息学竞赛中运用数形结合思想。

[4] 直接用“形”的方法做出的程序，空间复杂度是 O(N)的，而且程序很长，详见附录。

在信息学竞赛中运用数形结合思想，就是在处理问题时，斟酌问题的具体情形，善于抓住问题的主要矛盾，使数量关系的问题借助于几何图形直观而形象化，或者使图形问题借助于数量关系而本质化。

数形结合，重在“结合”二字。灵活的运用数形结合思想，需要重视思想的个体差异性，根据各人的现有知识水平和思维方式，有机的将抽象的数学、计算机语言与直观的图形结合起来，将抽象思维与形象思维结合起来，实现抽象概念与具体形象的联系和转化，更快更好更简单的解决实际问题。

附录

关于 2003 年上海市选拔赛题 Sequence

[题意简述]

一个序列{A_i, i=0, 1, 2, …, 3N}由 3N+1 项组成，每一项要么为 1，要么为 -2。定义部分和 $S_K=A_0+A_1+\ldots+A_K$，求所有满足性质 P 的序列 A 的数目，性质 P 为：S_{3N} = 1 且对于所有的 K = 0, 1, 2, …, 3N-1, 3N，有 $S_K>0$。即所有项的和为 1，且所有部分和为正。

例如 N=2 的时候，共有 3 组这样的序列：

1, 1, 1, -2, 1, 1, -2，

1, 1, 1, 1, -2, 1, -2，

1, 1, 1, 1, 1, -2, -2。

范围：$N\leqslant 1000$。

[分析]

[引理]任一序列 A，它的任何一种循环表示都不与自身相同。

[证明]若相同，根据循环串的性质，其必定可以分成 d>1 个完全相同部分。设每部分和为 s，显然有 $s*d$=1，而 d>1，则 s 一定不是整数，这与序列中所有项都是整数矛盾。

因此，A 的任意循环表示都不等于 A。Q.E.D.

[定理]满足性质 P 的序列个数为 $\frac{1}{3N+1}C_{3N+1}^{N}$。

[证明]列出所有的 A 序列，一共有 C_{3N+1}^{N} 个。

根据其循环表示分类，由于[引理]的成立，每一类中一定都有 3N+1 个序列，即一共 $\frac{1}{3N+1}C_{3N+1}^{N}$ 类。又因为 Raney 引理成立，所以每一类中有且只有一个序列满足性质 P。

即满足性质 P 的序列总数为 $\frac{1}{3N+1}C_{3N+1}^{N}$。Q.E.D.

同样的方法，可以推出 Catalan 数的公式，这里不再赘述。

论文附件

POI oi V Stage I Painter’s Studio 一题，集训队前辈的解题报告：

画室解题报告.doc

特别感谢湖南长郡中学的金恺提供这份报告。

POI oi V Stage I Painter’s Studio 一题，数形结合思想算法的程序：

Mal.pas

参考文献

CONCRETE MATHEMATICS by Ronald L. Graham & Donald E. Knuth & Oren Patashnik

USA Computing Olympiad : http://ace.delos.com/usacogate

Polish Olympiad in Informatics : http://www.oi.edu.pl/

上海市 NOI’2003 选拔赛（SHTSC 2003）试题

数形结合思想在数学教学中的妙用 from 教育教学论文网

伸展树的基本操作与应用

安徽省芜湖一中　杨思雨

目录

【关键字】

伸展树　基本操作　应用

【摘要】

本文主要介绍了伸展树的基本操作以及其在解题中的应用。全文可以分为以下四个部分。

第一部分引言，主要说明了二叉查找树在信息学竞赛中的重要地位，并且指出二叉查找树在某些情况下时间复杂度较高，进而引出了在时间复杂度上更为优秀的伸展树。

第二部分介绍了伸展树的基本操作。并给出了对伸展树时间复杂度的分析和证明，指出伸展树的各种基本操作的平摊复杂度均为 O(log n)，说明伸展树是一种较平衡的二叉查找树。

第三部分通过一个例子介绍了伸展树在解题中的应用，并将它与其它树状数据结构进行了对比。

第四部分指出了伸展树的优点，总结全文。

【引言】

二叉查找树（Binary Search Tree）能够支持多种动态集合操作。因此，在信息学竞赛中，二叉排序树起着非常重要的作用，它可以被用来表示有序集合、建立索引或优先队列等。

作用于二叉查找树上的基本操作的时间是与树的高度成正比的。对一个含 n 各节点的完全二叉树，这些操作的最坏情况运行时间为 O(log n)。但如果树是含 n 个节点的线性链，则这些操作的最坏情况运行时间为 O(n)。而有些二叉查找树的变形，其基本操作在最坏情况下性能依然很好，比如红黑树、AVL 树等等。

本文将要介绍的伸展树（Splay Tree），也是对二叉查找树的一种改进，虽然它并不能保证树一直是“平衡”的，但对于伸展树的一系列操作，我们可以证明其每一步操作的平摊复杂度都是 O(log n)。所以从某种意义上说，伸展树也是一种平衡的二叉查找树。而在各种树状数据结构中，伸展树的空间要求与编程复杂度也都是很优秀的。

【伸展树的基本操作】

伸展树是二叉查找树的一种改进，与二叉查找树一样，伸展树也具有有序性。即伸展树中的每一个节点 x 都满足：该节点左子树中的每一个元素都小于 x，而其右子树中的每一个元素都大于 x。与普通二叉查找树不同的是，伸展树可以自我调整，这就要依靠伸展操作 Splay(x,S)。

伸展操作 Splay(x,S)

伸展操作 Splay(x,S)是在保持伸展树有序性的前提下，通过一系列旋转将伸展树 S 中的元素 x 调整至树的根部。在调整的过程中，要分以下三种情况分别处理：

情况一：节点 x 的父节点 y 是根节点。这时，如果 x 是 y 的左孩子，我们进行一次 Zig（右旋）操作；如果 x 是 y 的右孩子，则我们进行一次 Zag（左旋）操作。经过旋转，x 成为二叉查找树 S 的根节点，调整结束。如图 1 所示

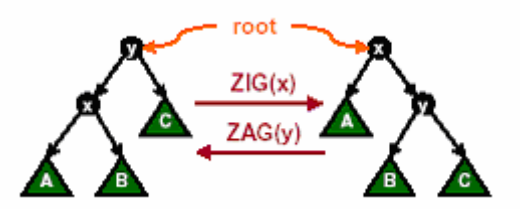

图 1

情况二：节点 x 的父节点 y 不是根节点，y 的父节点为 z，且 x 与 y 同时是各自父节点的左孩子或者同时是各自父节点的右孩子。这时，我们进行一次 Zig-Zig 操作或者 Zag-Zag 操作。如图 2 所示

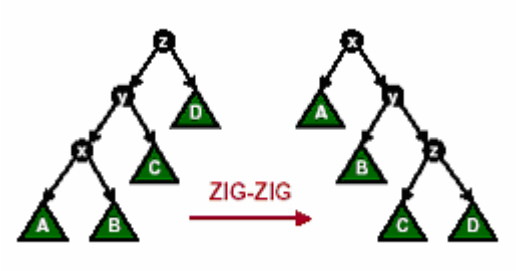

图 2

情况三：节点 x 的父节点 y 不是根节点，y 的父节点为 z，x 与 y 中一个是其父节点的左孩子而另一个是其父节点的右孩子。这时，我们进行一次 Zig-Zag 操作或者 Zag-Zig 操作。如图 3 所示

图 3

如图 4 所示，执行 Splay(1,S)，我们将元素 1 调整到了伸展树 S 的根部。再执行 Splay(2,S)，如图 5 所示，我们从直观上可以看出在经过调整后，伸展树比原来“平衡”了许多。而伸展操作的过程并不复杂，只需要根据情况进行旋转就

可以了，而三种旋转都是由基本得左旋和右旋组成的，实现较为简单。

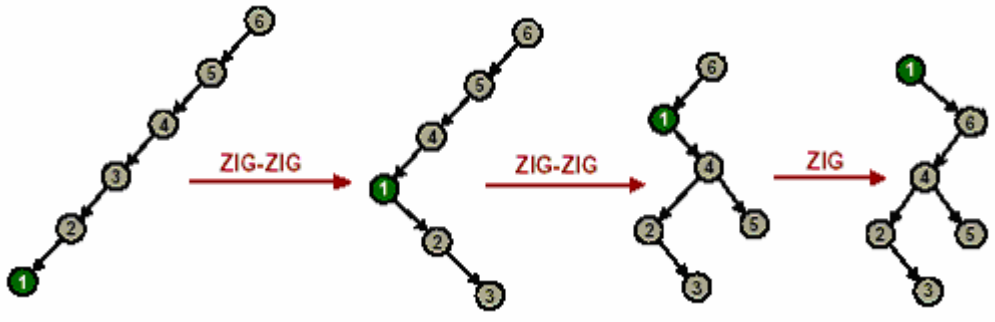

图 4　Splay(1,S)

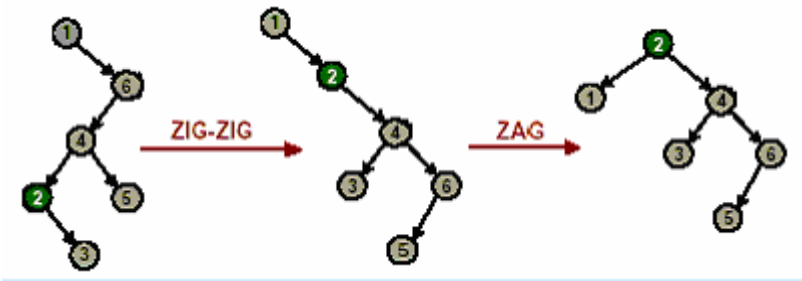

图 5　Splay(2,S)

伸展树的基本操作

利用 Splay 操作，我们可以在伸展树 S 上进行如下运算：

(1)Find(x,S)：判断元素 x 是否在伸展树 S 表示的有序集中。

首先，与在二叉查找树中的查找操作一样，在伸展树中查找元素 x。如果 x 在树中，则再执行 Splay(x,S)调整伸展树。

(2)Insert(x,S)：将元素 x 插入伸展树 S 表示的有序集中。

首先，也与处理普通的二叉查找树一样，将 x 插入到伸展树 S 中的相应位置上，再执行 Splay(x,S)。

(3)Delete(x,S)：将元素 x 从伸展树 S 所表示的有序集中删除。

首先，用在二叉查找树中查找元素的方法找到 x 的位置。如果 x 没有孩子或只有一个孩子，那么直接将 x 删去，并通过 Splay 操作，将 x 节点的父节点调整到伸展树的根节点处。否则，则向下查找 x 的后继 y，用 y 替代 x 的位置，最后执行 Splay(y,S)，将 y 调整为伸展树的根。

(4)Join(S1,S2)：将两个伸展树 S1 与 S2 合并成为一个伸展树。其中 S1 的所有元素都小于 S2 的所有元素。

首先，我们找到伸展树 S1 中最大的一个元素 x，再通过 Splay(x,S1)将 x 调整到伸展树 S1 的根。然后再将 S2 作为 x 节点的右子树。这样，就得到了新的伸展树 S。如图 6 所示

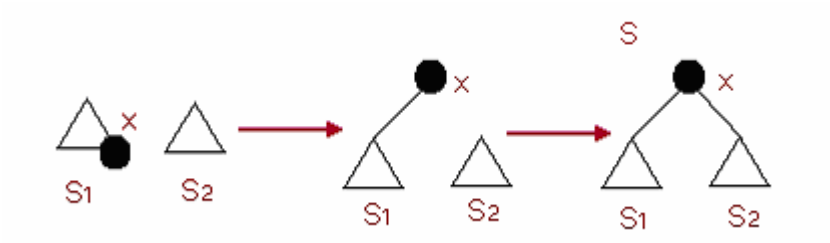

图 6

(5)Split(x,S)：以 x 为界，将伸展树 S 分离为两棵伸展树 S1 和 S2，其中 S1 中所有元素都小于 x，S2 中的所有元素都大于 x。

首先执行 Find(x,S)，将元素 x 调整为伸展树的根节点，则 x 的左子树就是 S1，而右子树为 S2。如图 7 所示

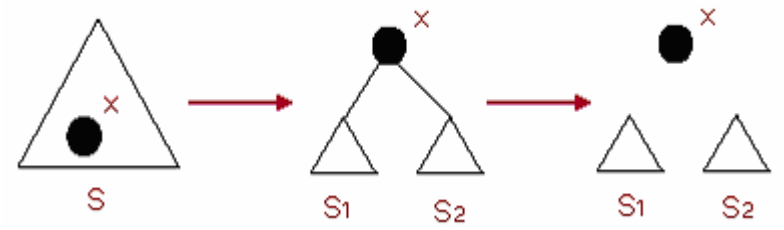

图 7

除了上面介绍的五种基本操作，伸展树还支持求最大值、求最小值、求前趋、求后继等多种操作，这些基本操作也都是建立在伸展操作的基础上的。

时间复杂度分析

由以上这些操作的实现过程可以看出，它们的时间效率完全取决于 Splay 操作的时间复杂度。下面，我们就用会计方法来分析 Splay 操作的平摊复杂度。

首先，我们定义一些符号：S(x)表示以节点 x 为根的子树。|S|表示伸展树 S 的节点个数。令 μ (S) = [log|S|]， μ (x)= μ (S(x))。如图 8 所示

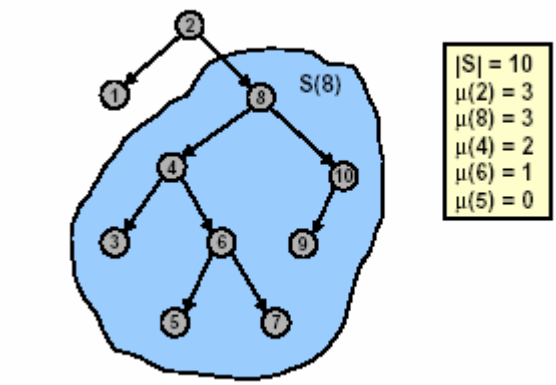

图 8

我们用 1 元钱表示单位代价（这里我们将对于某个点访问和旋转看作一个单位时间的代价）。定义**伸展树不变量**：在任意时刻，伸展树中的任意节点 x 都至少有 $\mu(x)$元的存款。

在 Splay 调整过程中，费用将会用在以下两个方面：

(1)为使用的时间付费。也就是每一次单位时间的操作，我们要支付 1 元钱。

(2)当伸展树的形状调整时，我们需要加入一些钱或者重新分配原来树中每个节点的存款，以保持不变量继续成立。

下面我们给出关于 Splay 操作花费的定理：

定理：在每一次 Splay(x,S)操作中，调整树的结构与保持伸展树不变量的总花费不超过 $3\mu(S)+1$。

证明：用 $\mu(x)$和 $\mu'(x)$分别表示在进行一次 Zig、Zig-Zig 或 Zig-Zag 操作前后节点 x 处的存款。

下面我们分三种情况分析旋转操作的花费：

情况一：如图 9 所示

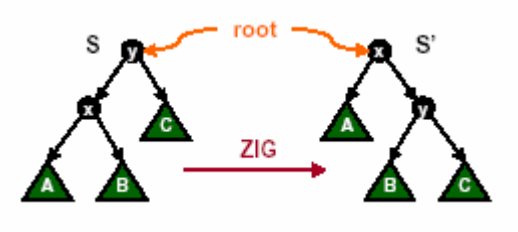

图 9

我们进行 Zig 或者 Zag 操作时，为了保持伸展树不变量继续成立，我们需要花费：

$$\begin{aligned}\mu'(x)+\mu'(y)-\mu(x)-\mu(y) &= \mu'(y)-\mu(x)\\ &\le \mu'(x)-\mu(x)\\ &\le 3(\mu'(x)-\mu(x))\\ &= 3(\mu(S)-\mu(x))\end{aligned}$$

此外我们花费另外 1 元钱用来支付访问、旋转的基本操作。因此，一次 Zig 或 Zag 操作的花费至多为 $3(\mu(S)-\mu(x))$。

情况二：如图 10 所示

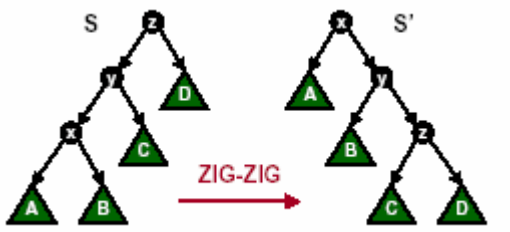

图 10

我们进行 Zig-Zig 操作时，为了保持伸展树不变量，我们需要花费：

$$\begin{aligned}\mu'(x)+\mu'(y)+\mu'(z)-\mu(x)-\mu(y)-\mu(z) &= \mu'(y)+\mu'(z)-\mu(x)-\mu(y)\\ &= (\mu'(y)-\mu(x))+(\mu'(z)-\mu(y))\end{aligned}$$

$$\leqslant (\mu'(x) - \mu(x)) + (\mu'(x) - \mu(x))$$
$$= 2(\mu'(x) - \mu(x))$$

与上种情况一样，我们也需要花费另外的 1 元钱来支付单位时间的操作。

当 $\mu'(x) < \mu(x)$ 时，显然 $2(\mu'(x) - \mu(x)) + 1 \leqslant 3(\mu'(x) - \mu(x))$。也就是进行 Zig-Zig 操作的花费不超过 $3(\mu'(x) - \mu(x))$。

当 $\mu'(x) = \mu(x)$ 时，我们可以证明 $\mu'(x) + \mu'(y) + \mu'(z) < \mu(x) + \mu(y) + \mu(z)$，也就是说我们不需要任何花费保持伸展树不变量，并且可以得到退回来的钱，用其中的 1 元支付访问、旋转等操作的费用。为了证明这一点，我们假设 $\mu'(x) + \mu'(y) + \mu'(z) > \mu(x) + \mu(y) + \mu(z)$。

联系图 9，我们有 $\mu(x) = \mu'(x) = \mu(z)$。那么，显然 $\mu(x) = \mu(y) = \mu(z)$。于是，可以得出 $\mu(x) = \mu'(z) = \mu(z)$。令 $a = 1 + |A| + |B|$，$b = 1 + |C| + |D|$，那么就有

$$[\log a] = [\log b] = [\log (a+b+1)]。 \quad ①$$

我们不妨设 $b \geqslant a$，则有

$$[\log (a+b+1)] \geqslant [\log (2a)]$$
$$= 1+[\log a]$$
$$> [\log a] \quad ②$$

①与②矛盾，所以我们可以得到 $\mu'(x) = \mu(x)$ 时，Zig-Zig 操作不需要任何花费，显然也不超过 $3(\mu'(x) - \mu(x))$。

情况三：与情况二类似，我们可以证明，每次 Zig-Zag 操作的花费也不超过 $3(\mu'(x) - \mu(x))$。

以上三种情况说明，Zig 操作花费最多为 $3(\mu(S)-\mu(x))+1$，Zig-Zig 或 Zig-Zag 操作最多花费 $3(\mu'(x)-\mu(x))$。那么将旋转操作的花费依次累加，则一次 Splay(x,S) 操作的费用就不会超过 $3\mu(S)+1$。也就是说对于伸展树的各种以 Splay 操作为基础的基本操作的平摊复杂度，都是 O(log n)。所以说，伸展树是一种时间效率非常优秀的数据结构。

【伸展树的应用】

伸展树作为一种时间效率很高、空间要求不大的数据结构，在解题中有很大的用武之地。下面就通过一个例子说明伸展树在解题中的应用。

例：营业额统计 Turnover（湖南省队 2002 年选拔赛）

题目大意

Tiger 最近被公司升任为营业部经理，他上任后接受公司交给的第一项任务便是统计并分析公司成立以来的营业情况。Tiger 拿出了公司的账本，账本上记录了公司成立以来每天的营业额。分析营业情况是一项相当复杂的工作。由于节假日，大减价或者是其他情况的时候，营业额会出现一定的波动，当然一定的波动是能够接受的，但是在某些时候营业额突变得很高或是很低，这就证明公司此时的经营状况出现了问题。经济管理学上定义了一种**最小波动值**来衡量这种情况：

该天的最小波动值= min { | 该天以前某一天的营业额-该天的营业额 | }

当最小波动值越大时，就说明营业情况越不稳定。而分析整个公司的从成立

到现在营业情况是否稳定，只需要把每一天的最小波动值加起来就可以了。你的任务就是编写一个程序帮助 Tiger 来计算这一个值。

注：第一天的最小波动值为第一天的营业额。

数据范围：天数 n≤32767，每天的营业额 ai≤1,000,000。最后结果 T≤2^{31}。

初步分析

题目的意思非常明确，关键是要每次读入一个数，并且在前面输入的数中找到一个与该数相差最小的一个。

我们很容易想到 $O(n^2)$的算法：每次读入一个数，再将前面输入的数一次查找一遍，求出与当前数的最小差值，记入总结果 T。但由于本题中 n 很大，这样的算法是不可能在时限内出解的。而如果使用线段树记录已经读入的数，就需要记下一个 2M 的大数组，这在当时比赛使用 TurboPascal 7.0 编程的情况下是不可能实现的。而前文提到的红黑树与平衡二叉树虽然在时间效率、空间复杂度上都比较优秀，但过高的编程复杂度却让人望而却步。于是我们想到了伸展树算法。

算法描述

进一步分析本题，解题中，涉及到对于有序集的三种操作：插入、求前趋、求后继。而对于这三种操作，伸展树的时间复杂度都非常优秀，于是我们设计了如下算法：

开始时，树 S 为空，总和 T 为零。每次读入一个数 p，执行 Insert(p,S)，将 p 插入伸展树 S。这时，p 也被调整到伸展树的根节点。这时，求出 p 点左子树中的最右点和右子树中的最左点，这两个点分别是有序集中 p 的前趋和后继。然后求得最小差值，加入最后结果 T。

解题小结

由于对于伸展树的基本操作的平摊复杂度都是 O(log n)的，所以整个算法的时间复杂度是 O(nlog n)，可以在时限内出解。而空间上，可以用数组模拟指针存储树状结构，这样所用内存不超过 400K，在 TP 中使用动态内存就可以了。编程复杂度方面，伸展树算法非常简单，程序并不复杂。虽然伸展树算法并不是本题唯一的算法，但它与其他常用的数据结构相比还是有很多优势的。下面的表格就反映了在解决这一题时各个算法的复杂度。从中可以看出伸展树在各方面都是优秀的，这样的算法很适合在竞赛中使用。

	顺序查找	线段树	AVL 树	伸展树
时间复杂度	$O(n^2)$	O(nlog a)	O(nlog n)	O(nlog n)
空间复杂度	O(n)	O(a)	O(n)	O(n)
编程复杂度	很简单	较简单	较复杂	较简单

【总结】

由上面的分析介绍，我们可以发现伸展树有以下几个优点：

(1)时间复杂度低，伸展树的各种基本操作的平摊复杂度都是 O(log n)的。在树状数据结构中，无疑是非常优秀的。

(2)空间要求不高。与红黑树需要记录每个节点的颜色、AVL 树需要记录平衡因子不同，伸展树不需要记录任何信息以保持树的平衡。

(3)算法简单，编程容易。伸展树的基本操作都是以 Splay 操作为基础的，而 Splay 操作中只需根据当前节点的位置进行旋转操作即可。

虽然伸展树算法与 AVL 树在时间复杂度上相差不多，甚至有时候会比 AVL 树慢一些，但伸展树的编程复杂度大大低于 AVL 树。在竞赛中，使用伸展树在编程和调试中都更有优势。

在信息学竞赛中，不能只一味的追求算法有很高的时间效率，而需要在时间复杂度、空间复杂度、编程复杂度三者之间找到一个“平衡点”，合理的选择算法。这也需要我们在平时对各种算法反复琢磨，深入研究，在竞赛中才能够游刃有余的应用。

【参考书目】

[1]傅清祥，王晓东.《算法与数据结构》.电子工业出版社.1998.01
[2]严蔚敏，吴伟民.《数据结构(第二版)》.清华大学出版社.1992.06
[3]《Introduction to Algorithms,Second Edition》.2001

【附录】

(一)　伸展操作和其他各种基本操作，其实现参见：Splay Tree.doc
(二)文中提到的伸展树的基本操作，具体过程可参照动画：Splay Tree.htm
(三)针对文中例题，作者用伸展树算法编写了程序：Turnover.pas