回到起点——一种突破性思维

南京市外国语学校 朱泽园

[关键字] 起点 类比 分离集合森林

[摘要]

高层次的信息学竞赛已经不是单纯一个算法、一种数据结构间的较量，而是对思维方法、创新能力的考验。本文旨在深刻剖析一种突破性思维——回到起点，亦即敢于放弃当前成果，回到初步分析处展开理性思考的可贵品质。

本文第一章提出一个已被经典算法解决的普通题。第二章对这道题展开讨论，简述了经典的解决方案，而后模拟本文主线描绘的思维方式进行思考，提出一个被遗忘的简单算法，并进行优化和类比，精确计算复杂度后问题被完美解决。

第三章对本思维方式在两个例题中的应用作了对比和升华，辩证地以前进性和曲折性的统一阐述了这类思想的重要性。

[目录]

[正文]

§1 问题的提出

§1.1 问题描述 [USACO 2.1 Shaping Regions 改编]

N 个不同颜色的不透明长方形（1≤N≤3000）被放置在一张长宽分别为 A、B 的白纸上。这些长方形被放置时，保证了它们的边与白纸的边缘平行。所有的长方形都放置在白纸内，所以我们会看到不同形状的各种颜色。坐标系统的原点（0,0），设在这张白纸的左下角，而坐标轴则平行于纸边缘。

输入：

第 1 行：A ， B 和 N, 由空格分开。

第 2~N+1 行：按照从下往上的顺序，每行输入的是一个长方形的放置。为五个数 llx, lly, urx, ury, color 这是长方形的左下角坐标，右上角坐标和颜色。其中 color 为整数。0<=llx,urx<=A，0<=lly,ury<=B。所有坐标（包括 A、B）都是 0 至 10^9 的实数。

颜色 1 和底部白纸的颜色相同。

输出：

输出文件应该包含一个所有能被看到颜色连同该颜色的面积（保留小数点后三位）的列表（即使颜色的编号不是连续的）。按 color 的增序顺序输出，不要输出面积为 0 的颜色。

样例输入：

```
20 20 3
2 2 18 18 2
0 8 19 19 3
8 0 10 19 4
```

样例输出

1 91.000
2 84.000
3 187.000
4 38.000

§1.2 问题的初步分析——离散化

自 IOI1998 的 Picture 问题始，和平面放置矩形有关的问题已在信息学竞赛中屡见不鲜，其最基本的预处理操作是将平面离散化。如图 1

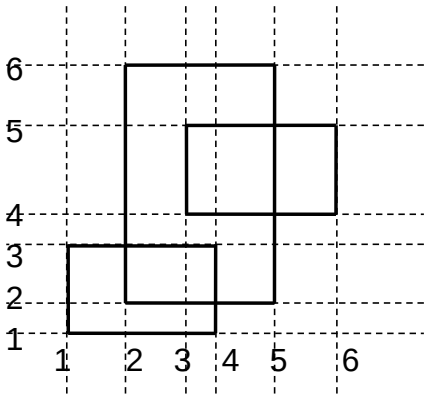


图 1

将平面理解成网格，也就是说将所有在矩形坐标中出现过的 x 坐标（y 坐标）提出，排序后删除重复，并按顺序编号。任意两个相邻格点连成的线段，称之为"元线段"；两个相邻离散过的 x 坐标（y 坐标）之间的区域，称之为"离散列"（"离散行"）；任意一个离散行和离散列的公共部分称之为"离散格"。对每个矩形坐标相应地存在一个格点坐标与之对应。

这有助于处理坐标为实数，或者范围超过 longint 的问题。离散化之后，任意矩形顶点坐标均可表示为 1 到 2N 之间的整数：利用 hash 策略可以将实数坐标在 O(1)时间内转化为对应的整点坐标，更显然地，直接读表可以在 O(1)时间内将格点坐标转化为原矩形坐标。因此后文只考虑坐标为整数，且范围在 1 到 2N 之间的 N 个格点矩形的问题。

§1.3 一个朴素的想法

最朴素算法的实现取自于简单的灌水（Floodfill）算法。就是用 2Nx2N 的数组，分别记录每个离散格的颜色，初始时全部无色。自顶至下处理每一个矩形，将其覆盖之下无色的部分全部填上颜色。

考虑到空间可能无法承受，改进的措施是每次处理一个离散行（2N 的一维数组），自顶向下处理每个矩形，如果这个矩形与当前离散行有公共部分，那么就和前面类似地，将无色部分涂色，如图 2：

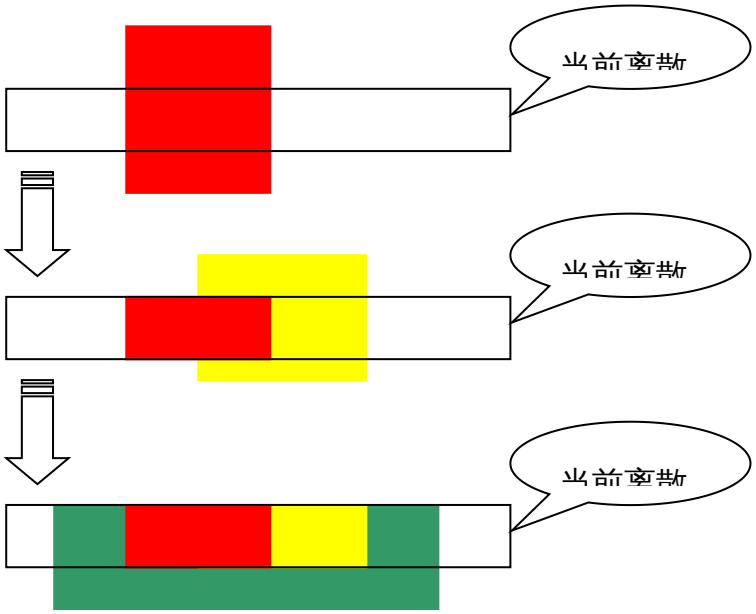


图 2

该方法基本不会增加程序运行时间，只是时间复杂度系数稍微增加了些，并不影响大局。

§2 问题的解决

§2.1 经典算法

解决这类问题的经典算法莫过于线段树。同样是对每个离散行做处理，该算法利用了 O(N)空间的树型数据结构。如图 3，构建一个线段树：

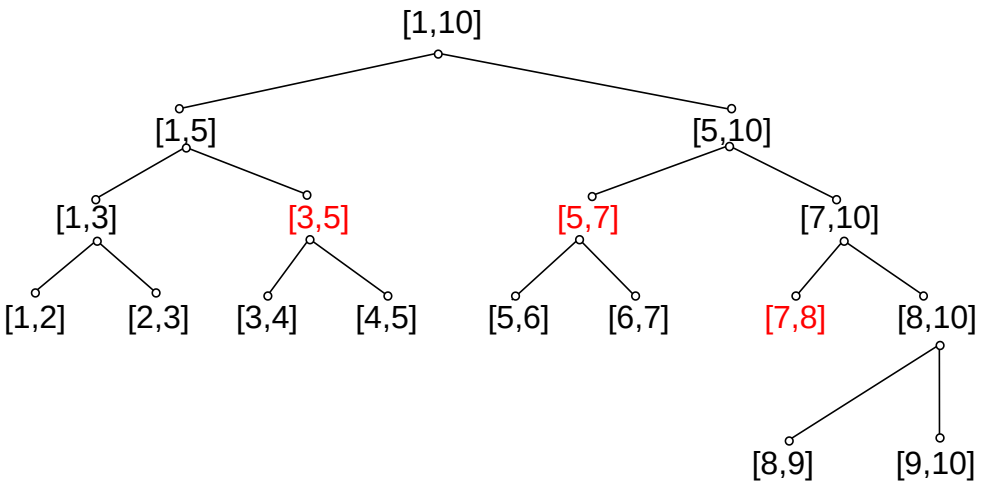


图 3

该数据结构主要支持的操作是，给定线段[l,r]和属性 X 组成的操作对(l,r,X)，意思是将区间[l,r]赋予属性 X。称"在 P 节点上进行(l,r,X)操作"为 P(l,r,X)，具体操作如下：

```
Operation P(l,r,X)
  If [l,r]与当前节点 P 对应的区间交集为空 Then Return
  If [l,r]可以完全覆盖当前结点 P 对应的区间
    Then 将属性 X 记录在该结点
```

例如图 3 中红色的部分是处理 Root(3,8,X)操作后被赋予 X 属性的结点。可以证明对每个操作 Root(l,r,X)，其时间复杂度为 O(logN)（N 是叶结点个数，也就是区间范围）。

观察本题需要这个数据结构支持怎样的特殊操作：

1、 插入一条颜色为 X 的线段[l,r]，该区间[l,r]上原有颜色不被替换，其余部分染上颜色 X。

2、 返回所有颜色当前的覆盖量。（该操作只将在该离散行处理完毕后一次性调用）

操作 2 只需将线段树全部遍历一遍即可求得，时间复杂度 O(N)。至于操作 1，对线段树上的每条线段标记颜色不是难事儿，只需要将前文所述的属性 X 定义其为 color flag，用来标记颜色。因此主要难点在于不破坏区间上的已有颜色。

其实这一点也不难实现，类似地我们给出这个操作的具体过程：

Operation P(l,r,X)

If [l,r]与当前节点 P 对应的区间交集为空 **Then Return**

If 节点 P 已经被某种颜色覆盖 **Then Return**

If [l,r]可以完全覆盖当前结点 P 对应的区间

至此该算法的空间复杂度为 O(N)，时间复杂度为 $O(N^2 \log N)$，在某种程度上是一个可以通过本题数据的优秀算法。

§2.2 另类算法

避开繁文缛节，我们回到起点，分析一下原始算法，就是"§1.3 一个朴素的想法"内的算法。有效解决问题往往有两条途径，一个是使用高级算法或数据结构，另一个就是分析原有的算法或数据结构，找出冗余进行针对性改进。

回顾一下之前的算法

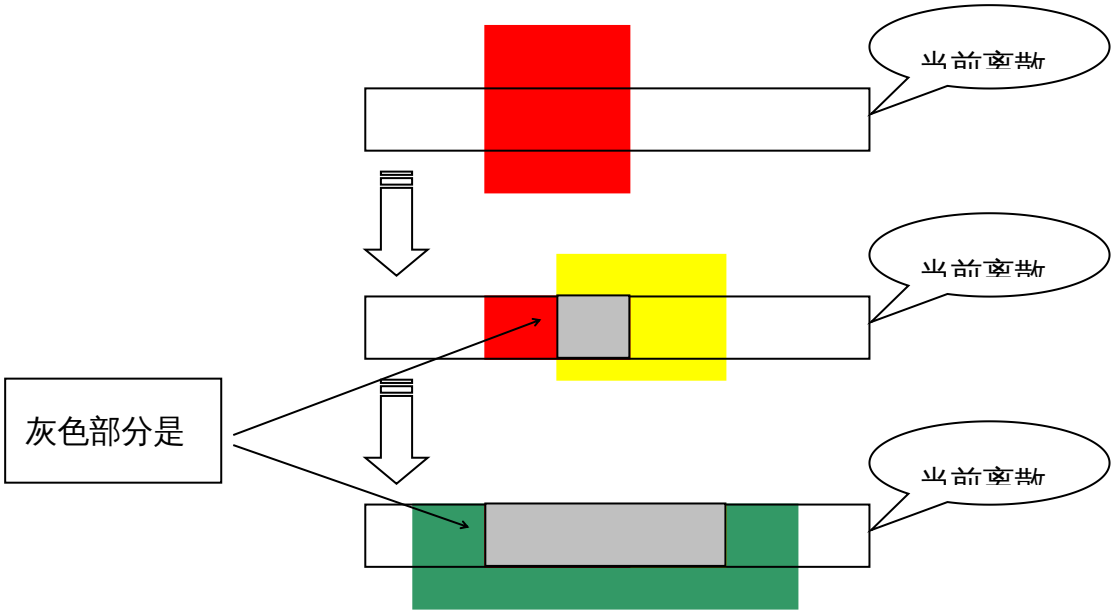


图 4

迅速找到冗余：处理当前线段（矩形与当前离散行的交集）时，对已经覆盖的线段，不能迅速避开，需要无畏的比较。改进措施如下：

设立 next 数组，初始时 next[i]=i+1；另还有 color 数组，color[i]记录该离散行的第 i 个离散格当前的颜色。设当前插入的线段为[*l*,r]，即占用编号为 *l* 至 r-1 的离散格：

1. i←*l*
2. 查看 color[i]，如果它未被着色，那么将其着色。

下面的图 5 将给出一组覆盖的例子：（插入的线段与图 4 稍有类似）

第一次用颜色 1 的线段覆盖[2,6），因为 next 数组完全是初始值，因此 i 指针从 *l*=2 一直移动到 i=r，color[2..5]被置为颜色 1，next[2..5]被置为 next[r-1]=r=6。

第二次用颜色 2 的线段覆盖[4,7)，i 指针来到 *l*=4，color[4]不被修改，i←next[i] = next[4] = 6。接下来 color[6]被修改，指针移动到 next[6]=7=r，移动完毕。指针经过了 4，因此 next[4]被置为 next[r-1]=r=7。

第三次用颜色 3 的线段覆盖[1,9)，i 指针先来到 1，修改 color[1]，i←next[i]来到 2，观察到 color[2]已经有颜色以后先后转战 i←next[2]=6，i←next[6]=7。至此可以修改 color[7]以及 color[8]，并且最终在 i=9 的时候停止。过程中 next[1]、next[2]、next[6]、next[7]都需要被修改为 next[8]=9。

i	1	2	3	4	5	6	7	8
next[i]	2	3	4	5	6	7	8	9
color[i]	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL

i	1	2	3	4	5	6	7	8
next[i]	2	6	6	6	6	7	8	9
color[i]	NIL	1	1	1	1	NIL	NIL	NIL

i	1	2	3	4	5	6	7	8
next[i]	2	6	6	7	6	7	8	9
color[i]	NIL	1	1	1	1	2	NIL	NIL

i	1	2	3	4	5	6	7	8
next[i]	9	9	6	7	6	9	9	9
color[i]	3	1	1	1	1	2	3	3

图 5

观察上述算法，其精髓在于将已染色的相邻离散格合并，或者对其“预定”合并，也就是做上标记，使其在以后的处理中自动合并——平摊思想。这一想法源于“带路径压缩按秩合并的分离集合森林”，也就是高效的俗称 “并查集”算法。注意到当前算法并未完全进行路径压缩，而且没有按秩合并，因此该算法有待完善。（其实可以证明本算法在大部分情况下时间复杂度维持在 N^2 左右，不易退化）

§2.3 通过完整的路径压缩完善算法

将相邻的已染色线段看成一个集合，这样在[l,r)的检索过程中，遇到已染色线段可以大跨步跨越。我们的目标是让这个跨越在平摊 O(1)的时间内完成。算法改进如下：

设立 next 数组，初始时 next[i]=i+1；同时设立 p 数组，初始时 p[i]=i（p 数组类似并查集算法中的父节点指针）；另还有 color 数组，color[i]记录该离散行的第 i 个离散格当前的颜色。设当前插入的线段为[l,r]，即占用编号为 l 至 r-1 的离散格：

1. i←l
2. 查看 color[i]，如果它未被着色，那么将其着色。
3. 将当前指针位置记录
4. 如果 i 指针已经越过为 r-1，跳转 7

为了建立分离集合森林，我们作如下定义：

1、 用 i 代表编号为 i 的节点

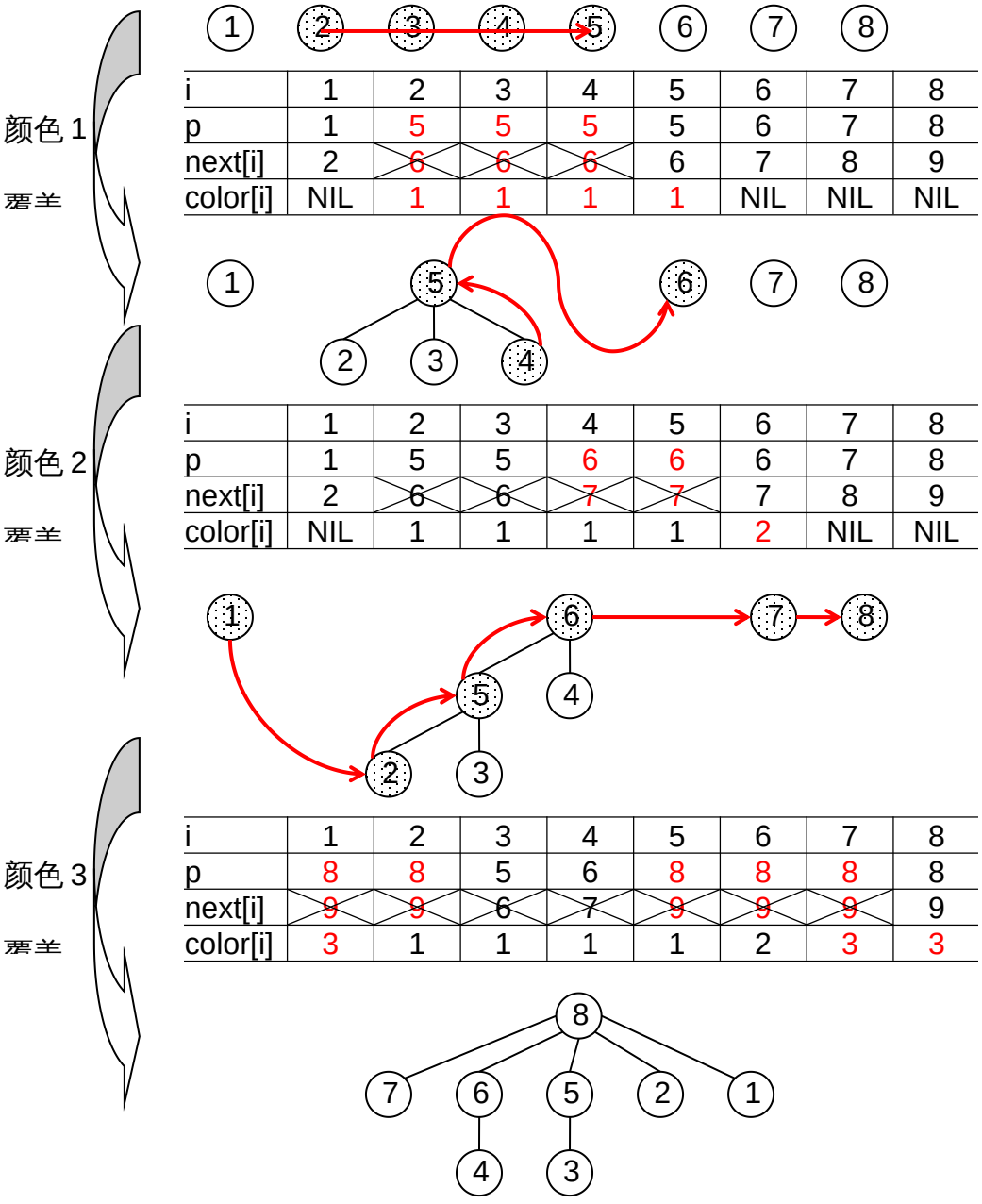
2、 将 p[i]定义为节点 i 的父节点

3、 父节点为自身的节点定义为根节点。

注意：对于 p[i]!=i 的节点，也就是分离集合森林中的非根节点，其 next[i]已经失去意义。而对于根节点，必然有 next[i] = i+1。这一点将为 2.4 节服务。

与图 5 相同，这里仍对上文数据给出图示及文字解释（因为 next/color 的定义与§2.2 类似，因此对其改变我们不多赘述，将重点放在分离集合森林的构建）：

i	1	2	3	4	5	6	7	8
p	1	2	3	4	5	6	7	8
next[i]	2	3	4	5	6	7	8	9
color[i]	NIL	NIL	NIL	NIL	NIL	NIL	NIL	NIL



i	1	2	3	4	5	6	7	8
p	1	5	5	5	5	6	7	8
next[i]	2	~~6~~	~~6~~	~~6~~	6	7	8	9
color[i]	NIL	1	1	1	1	NIL	NIL	NIL

i	1	2	3	4	5	6	7	8
p	1	5	5	6	6	6	7	8
next[i]	2	~~6~~	~~6~~	~~7~~	~~7~~	7	8	9
color[i]	NIL	1	1	1	1	2	NIL	NIL

i	1	2	3	4	5	6	7	8
p	8	8	5	6	8	8	8	8
next[i]	~~9~~	~~9~~	6	7	~~9~~	~~9~~	~~9~~	9
color[i]	3	1	1	1	1	2	3	3

图 6

第一次用颜色 1 的线段覆盖[2,6），因为所有节点自成集合，故本次覆盖操作将 2..5 节点合并，***5 为他们的根节点（代表元）***，显然 next[2..4]失去其意义（在表格中用 X 表示）。

第二次用颜色 2 的线段覆盖[4,7)，i 指针来到 I=4，根据分离集合森林的定义，需要找到其所在集合的根节点（该集合的代表元），也就是节点 p[4]=5。紧接跳转下一个集合，也就是节点 6。指针 i 移动到 6，并且修改 color[6]。最后压缩路径——前面所经过的所有点 x 的 p[x]都指向 6——5 和 6 所在的集合合并，***6 成为新的根节点***。

第三次用颜色 3 的线段覆盖[1,9)，i 指针先来到 1，修改 color[1]后跨入 next[1]=2 所在的集合，发现其已染色，通过 i←p[p[2]]=6 以及 i←next[i]=7 跨越该集合，并对 7、8 两个独立的集合染色。最后 1、2、7、8 所在集合进行总合并。***8 为新的集合的根节点（代表元）***。

§2.4 秩的建立

所谓秩，就是一个集合(树)所含的节点个数。对分离集合的合并，采取将节点数多的集合(树)，附着在节点数少的集合(树)上的策略，可以将集合的合并复杂度降至平摊的 O(1)。这一个过程需要在上文蓝色粗斜体部分，即确定合并后新根节点的部分稍加修改，选取秩最大的树根作为新的根节点（代表元）即可。注意到这时对根节点不一定有 next[i] = i+1。

一共有最多 2N 个集合，合并次数不超过 2N-1，对当前离散行的处理时间复杂度仅为 O(N)，整个算法的时间复杂度为 $O(N^2)$。

§2.5 小结

摒弃当今主流的线段树算法，回到起点，对朴素算法进行改进的思想实属不易。但其后算法的完全建立，思维过程极其缜密：首先找到冗余进行修改，旨在**量变**——某种意义上提高算法对部分数据的运行效率；发现算法的优越之

处后，展开**类比**，将类似数据结构、算法的自身优点吸取，并进行改进；最后大步伐迈向让算法复杂度**质变**的目标。

§3 总结

希腊传说弗里吉亚国国王打了一个戈尔迪之结，预言称能解开它的人将统治整个亚洲。亚历山大大帝认为它在预言中找到了自己的命运归宿，拿起长剑，径直将结劈开。有人说亚历山大将绳劈开是"作弊"的行为，但或许奥林匹斯山诸神寻找的就是如此少说多做的决断力？

面对**多元化**的世界，我们总会有这种"**答案意识过强**"的状态，就是这种倾向：对答案进行快速的设想，迅速沿那条道兴高采烈地出发，却并没在意先前设想的正确性、全面性。被自己创造的无形障碍蒙蔽实乃可惜，因此我们不能因片面的认识，或对放弃眼前利益的恐惧而裹足不前，偶尔我们也需要此类**果敢与魄力**。这正是**前进性和曲折性的统一**。

问题的表示往往比答案更重要，答案不过乃数学或实验。要提出新的问题、新的可能性、从某个新的角度考虑一个旧问题，都要求创造性的想象力，**回到起点**对问题重新定义，这才是真正的科学进步之所在。

[参考资料]

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. 《实用算法的分析和程序设计》

by 吴文虎 王建德

v. http://get-me.to/oi 辛韬同学提供的 BalticOI2004 解答

[感谢]

1、感谢辽宁省的辛韬同学对我的大力帮助（包括部分算法和证明）。

2、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。

3、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。

4、感谢我的家人给我的关怀。

5、感谢所有的信息学好友！

浅析倍增思想在信息学竞赛中的应用

安徽省芜湖市第一中学 朱晨光

目 录

摘要

倍增思想是解决信息学问题的一种独特而巧妙的思想。本文就倍增思想在信息学竞赛中两个方面的应用进行了分析。全文可以分为五个部分：

第一部分引言，简要阐述了倍增思想的重要作用以及运用方法；

第二部分介绍倍增思想的第一个应用——在变化规则相同的情况下加速状态转移；

第三部分介绍倍增思想的第二个应用——加速对区间进行的操作；

第四部分探讨了一个有趣的问题，即为什么倍增思想每次只将考虑范围扩大一倍而不是两倍、三倍等；

第五部分总结全文，再次指出倍增思想的重要性以及应该怎样灵活运用倍增思想。

关键字

倍增思想 变化规则 状态转移 区间操作

正文

引言

倍增思想是一种十分巧妙的思想，在当今的信息学竞赛中应用得十分广泛。尽管倍增思想可以应用在许多不同的场合，但总的来说，它的本质是：每次根据已经得到的信息，将考虑的范围扩大一倍，从而加速操作。大家所熟悉的归并排序实际上就是倍增思想的一个经典应用。

在解决信息学问题方面，倍增思想主要有这两个方面的应用——

一、在变化规则相同的情况下加速状态转移；

二、加速区间操作。

下文将就这两个方面进行详细的讨论。

倍增思想应用之一　在变化规则相同的情况下加速状态转移[1]

首先，让我们来看一个简单的例子——已知实数 a，计算 a^{17}。

分析：

很显然，一种最简单的方法就是令 b=a，然后重复 16 次进行操作 b=b*a.这样，为了得到 a^{17}，共进行了 16 次乘法。

现在考虑另外一种方法，令 $a_0=a,a_1=a^2,a_2=a^4,a_3=a^8,a_4=a^{16}$,可以看出，$a_i=a_{i-1}^2,(1<=i<=4)$。于是，得到 a_0,a_1,a_2,a_3,a_4 共需要 4 次乘法。而 $a^{17}=a*a^{16}=a_0*a_4$,
也就是说，再进行一次乘法就可以得到 a^{17}.这样，总共进行 5 次乘法就算出了 a^{17}.

如果将这种方法推而广之，就可以解决这样一个一般性的例题：

例 1、 已知 a ，计算 a^n:

分析：

1、 将 n 表示成为二进制形式并提取出其中的非零位，即

$n=2^{b1}+2^{b2}+\ldots\ldots+2^{bw}$，不妨设 b1<b2<……<bw.

[1] 这里是指由一种状态变化到另一种状态，并不只限于动态规划中的“状态转移”。

2、 由于已知 a ，所以也就知道了 a^{2^0} ，重复 bw 次将这个数平方并记录下来，就可以得到(bw+1)个数： a^{2^0} ， a^{2^1} ， a^{2^2} ，……， $a^{2^{bw}}$ ；

3、 根据幂运算的法则，可以推出 $a^n = a^{2^{b1}+2^{b2}+\ldots\ldots+2^{bw}} = a^{2^{b1}} * a^{2^{b2}} * \ldots\ldots * a^{2^{bw}}$ ，而这些数都已经被求出，所以最多再进行(bw+1)次操作就可以得到 a^n 。

由于 n 的二进制表示最多有 $\lfloor \log_2 n \rfloor + 1$ 个非零位，所以 bw 最大为 $\lfloor \log_2 n \rfloor$ 。也就是说，最多进行 O($\log_2 n$)次乘法就可以算出 a^n ，这比进行 O(n)次乘法效率高得多。

当然，由于得到 n 的二进制表示的过程本身就是按照从低位到高位的顺序，所以并不需要记录 a^{2^0} ， a^{2^1} ， a^{2^2} ，……， $a^{2^{bw}}$ ，只需要每次即算即用就可以了。伪代码如下（见下页）：

那么，这个算法是如何减少乘法次数的呢？显然， $a^n = a^{2^{b1}+2^{b2}+\ldots\ldots+2^{bw}} = a^{2^{b1}} * a^{2^{b2}} * \ldots\ldots * a^{2^{bw}}$ 使得求 a^n 转化为求不超过 $\lfloor \log_2 n \rfloor + 1$ 个 a 的幂的积。而序列 a^{2^0} ， a^{2^1} ， a^{2^2} ，……， $a^{2^{bw}}$ 中除了第一个数以外，每一个数都是前一个数的平方。即在从 a^{2^i} 得到 $a^{2^{i+1}}$ 的过程中，按照原始的方法需要进行 2^i 次乘法操作，而现在只需要利用已知结果 a^{2^i} 进行一次乘法操作（ $a^{2^i} * a^{2^i} = a^{2^{i+1}}$ ）即可。大大减少了操作次数，从而降低了时间复杂度。

```
long double power(long double a,long n)
{
 long double b,result;
 result=1;
 b=a;
 while (n)
  {
     if (n%2)
      result*=b;
     b*=b;
     n/=2;
  }
 return result;
}
```

算法 1.1

而在实际情况中，a 可能是一个实数，也可能是一个矩阵或是一个抽象的状态。变化规则也可能是其他操作（如矩阵乘法、动态规划的状态转移等）。但是只要符合以下两个条件，就可以应用倍增思想并采用类似于上面的方法加速计算：

1、 每次的变化规则必须相同；

2、 变化规则必须满足结合律。

具体到上面的例子，每次的变化规则都是乘法，而乘法是满足结合律的。

下面将通过另一个例子更加深入地探讨倍增思想在加速状态转移方面的应用，同时得到更精确的定义。

例 2、骰子的运动[2]

给定一个六个面的骰子，每个面上都有一定的权值（1 到 50 之间的整数）。骰子运动的范围是一个宽度为 4，向左右无限延伸的带子（如图 1.1）。

[2] CEPC 2003 Problem D Dice Contest

带子从左到右的横坐标值为……，-3，-2，-1，0，1，2，3，……，从前到后的纵坐标值依次为 1，2，3，4（这里的坐标对应的都是格子，而不是点）。这样，带子被分成了无限多个格子。每个格子恰好能与骰子的一个面完全重合。骰子每次可以向前后左右中的一个方向移动一格（但不能移出带子），花费是移动后朝上的面所附带的权值。

给定当前骰子位置的坐标与各个面的朝向，求将这个骰子移动到某个新位

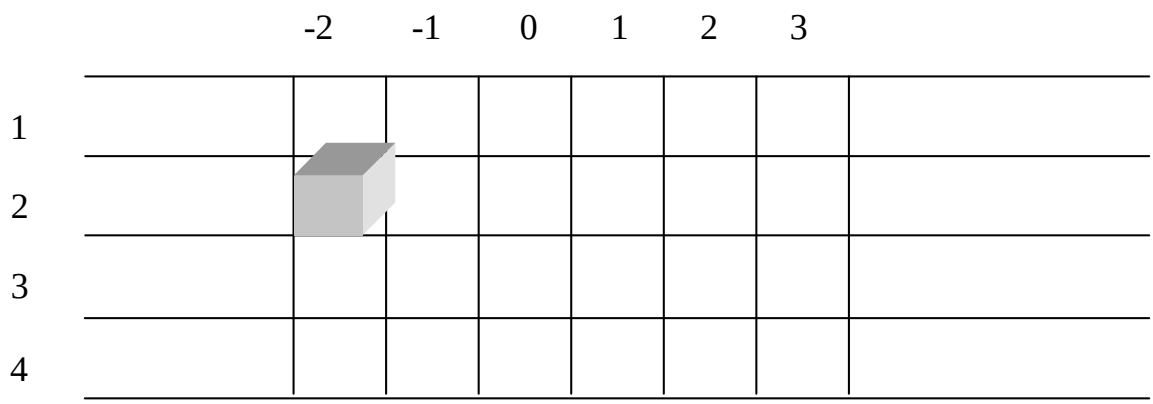


图 1.1

置所需的最小花费。（所给横坐标的绝对值小于等于 10^9）。

分析：

如果不考虑横坐标巨大的差值，本题完全可以用动态规划求解。方法是将每一格按照骰子的朝向拆分成 24 种状态，然后按列进行动态规划（本质上是一个分层图）得到最小花费。具体方法是从第一列 24*4=96 个状态推到第二列 96 个状态，再推到第三列，第四列……，一直推到终点所在的列，每次都用 Dijkstra 算法算出从一列中某个状态转移到相邻的一列中某个状态的最小花费。时间复杂度是与横坐标差值 n 是同阶的。但是，由于 n

最大可达 10^9，所以这个算法无论从时间还是空间上都难以满足要求。那么，是否可以采用倍增思想呢？答案是肯定的。

下面，我们来验证这里是否存在一种变化规则符合运用倍增思想的要求——相同并符合结合律。

设骰子从第 1 列某个状态 A 运动到第 2 列某个状态 B 需要花费代价 c，则骰子从第 2 列的状态 A（与前一个 A 的行号以及朝向均相同）运动到第 3 列的状态 B 同样需要花费代价 c！这就是一种“相同的变化规则”！

更一般地，如果骰子从第 i 列某个状态 A 运动到第(i+k) [3]列某个状态 B 需要花费代价 c，则骰子从第 j 列某个状态 A 运动到第(j+k)列某个状态 B 也需要花费代价 c，这就是相同的变化规则。

又由于前面所给出的算法是动态规划，而动态规划一个很重要的特点是具有最优子策略。所以如图 1.2 中按照 A->B->C 的顺序计算转移费用再加起来（对于一条路径来说，最小费用是它在这三个部分中的最小费用之和）与按照 B->C->A 的顺序计算转移费用再加起来完全一样，因为它们相互独立，而且合并方式是加法，所以符合结合律。

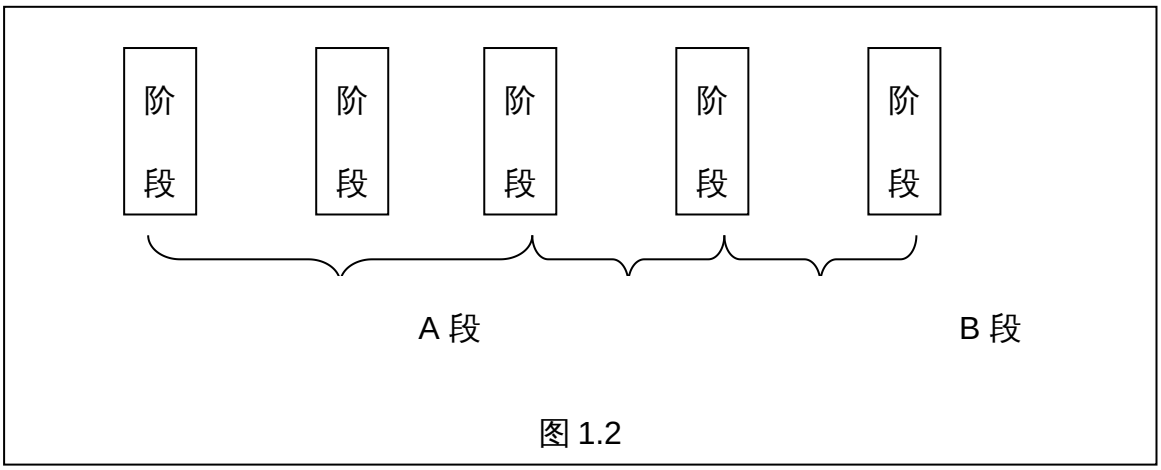


图 1.2

[3] 这里 k 可以是正数或负数，表示向右或向左运动

至此，我们证明了变化规则既满足相同性又满足结合律，可以运用倍增思想解决：

（以下默认终点在起点的右侧，如果是在左侧，处理方法类似；如果在同一列，用 Dijkstra 算法就可以得到答案）

1、 用 Dijkstra 算法得到一列中某个状态 A 转移到右边相邻一列中某个状态 B 所需最小花费。这可以推出一个 96*96 的矩阵，不妨称为 a^1;

2、 根据已经得到的变化规则 a^1 可以计算出一列中某个状态 A 转移到右边与其距离为 2 的一列中某个状态 B 所需花费——a^2。

3、 依次类推得到 a^4，a^8，a^{16}，a^{32}，……, a^w(w<=n 且 2w>n);

4、 与例 1 相类似，设 $n=2^{b1}+2^{b2}+\ldots\ldots+2^{bw}$，然后从初始状态（即起点所在的那一列）开始，不断对其施行变化规则 $a^{2^{b1}}$、$a^{2^{b2}}$、……、$a^{2^{bw}}$，最终得到从起点到终点所在那一列每个状态的最小花费。

本题中，Dijkstra 算法所耗费的时间可以看成是常数（根据题目中骰子各面权值的取值范围可以算出必须考虑的点数为常数），每次倍增的时间为 96^3，而倍增的次数为 $\log_2 n$，所以上述算法的时间复杂度为 $O(96^3 \log_2 N)$。

从上题中，我们进一步加深了对于倍增思想的理解，并且纠正了一个以前错误的认识：倍增思想所作用的对象实际上是变化规则，而并非具体的状态！倍增思想的作用是算出一个状态到另一个状态的变化量。设初始状

态为 A，目标状态为 B，A 到 B 的变化量为 c，则最终结果是 A*c（这里的乘号表示一种变化的手段）。也就是说，倍增思想计算出的量与具体的状态是无关的，而仅与状态之间的关系有关。将这个过程写成伪代码就是（见下页）：

```
Doubling Algorithm(A,n)
 {
  //a,b,c 均为变化规则，a 的初值由其他算法得到。如例 2 中 a 由 Dijkstra 算法得到
  c=b=a;
  while (n)
   {
      if (n%2)
       c=c•b;
      b=b•b;
      n/=2;
   }
  B=A*c;
```

（A 为原状态，B 为末状态。“状态*变化规则[4]”的结果表示对于某个状态施行变化规则后得到的状态，“变化规则•变化规则”的结果表示与依次施行两个变化规则等价的变化规则。）

可以看出，算法 1.2 与算法 1.1 十分相似。其实，算法 1.1 就是算法 1.2 的一个实例。

[4] 这里的规则是灵活多变的，可以指“能否达到”，也可以指“转移后新增的花费”等，要根据具体的题目制定特定的规则。

需要再次强调的是，这里的变化规则必须是相同的（这比较容易检验），并且满足结合律。一个不满足结合律的运算——除法就不能用倍增思想求解（除非转化成乘法）：$a_1/a_2/a_3/a_4\ldots\ldots/a_n \neq a_1/(a_2/a_3/a_4\ldots\ldots/a_n)$。这也提醒我们，在运用倍增思想解题之前，必须检验是否可行。如果不满足条件，就要构造出合法的变化规则或者思考其他的方法。

以上便是倍增思想的第一个应用——在变化规则相同时加速状态转移。当然，这个应用还有一些其他的实现方法。比如在计算 a^n 时,不一定要计算 $a^1,a^2,a^4,a^8\ldots\ldots$，可以按照如下的规则进行递归：

$$a^n=\begin{cases} a & n=1， \\ a^{n/2}*a^{n/2} & n\text{ 为偶数}， \end{cases}$$

很显然，这种算法的时间复杂度也是 $O(\log_2 N)$，这也说明倍增思想在实际操作中是灵活多变的，要根据实际情况选择相对简便的形式加以利用。

倍增思想应用之二 加速区间操作

在区间操作方面，有许多表现优秀的算法与数据结构，线段树便是其中的代表。这里之所以提出倍增思想，是因为它也可以胜任在区间上的一些操作，并且在某些特定的情况下可以做得更好。下面先介绍在这种情况下使用倍增思想的模式，再通过几个例题详细说明。

在区间操作中运用倍增思想的一般模式：

预处理：

对于区间中每一个点 A，记录[A,A+2^0-1],[A,A+2^1-1],[A,A+2^2-1],……[A,A+$2^{\lfloor \log_2 N \rfloor}$-1][5],这($\lfloor \log_2 N \rfloor$+1)个区间的性质。在记录的过程中，按照区间长度依次求解（也可以按照点的顺序求解，下文中求树上最近公共祖先的例题中将会有所介绍）。设要求[A,A+2^{2i}-1]的性质，则可以通过已知结果[A,A+2^i-1]与[A+2^i, (A+2^i)+ 2^i-1]得到有关性质。

取用：

在取用区间[A,B]时，有两种适用范围不同的方法：

1、 如果区间的重叠对于所需结果无影响（如两段区间重叠对于求最大值没有影响），则令 k=$\lfloor \log_2(B-A+1) \rfloor$. 根据区间[A,A+$2^k$-1]与[B-$2^k$+1,B]就可以

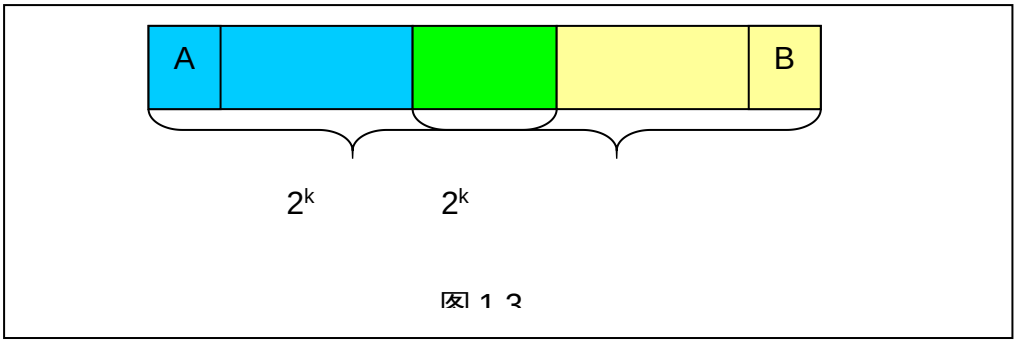


图 1.3

得到所需结果（如图 1.3），时间复杂度为 O(1)；

2、 如果区间的重叠对于所需结果有影响（如两段区间重叠对于计数问题有影响），则将[A,B]划分成为[A,A+$2^{\lfloor \log_2(B-A+1) \rfloor}$-1]，[A+$2^{\lfloor \log_2(B-A+1) \rfloor}$，A+$2^{\lfloor \log_2(B-A+1) \rfloor}$+$2^{\lfloor \log_2(B-(A+2^{\lfloor \log_2(B-A+1) \rfloor})+1) \rfloor}$-1]，……[x,B]；（x 为某个数）每次用对数运算直接得到划分点（如图 1.4）。可以证明每次至少将区

[5] 如果某些区间[A，B]中的 B 已经大于 N，则取[A，N]代替。

间减半，所以最多分成了$\lfloor \log_2(B-A+1) \rfloor$+1 个区间。因此这样处理的时间复杂度为 O(log$_2$N).

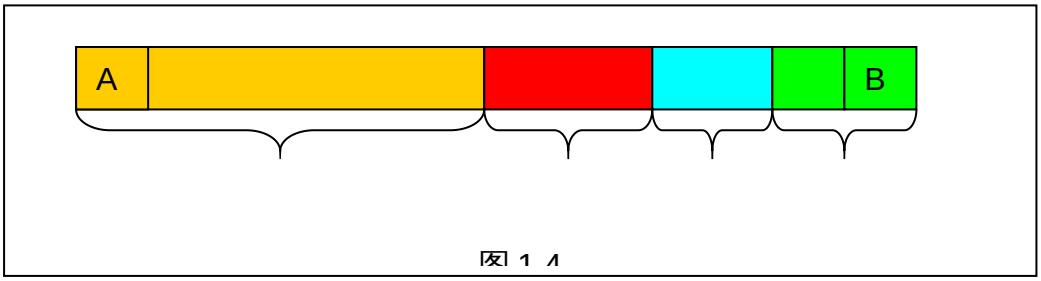


图 1.4

综上所述，预处理时间复杂度为 O(Nlog$_2$N)，处理一个区间的时间复杂度为 O(1)或 O(log$_2$N)（视处理方法而定）。空间复杂度为 O(Nlog$_2$N)

例 3　一般 RMQ 问题[6]

给定一个长度为 n 的数组 A，每次询问？？i,j(i<=j)，要求得到 A[i…j]中最小数的下标。

分析：

构造数组 B，使得 B[i,j]表示 A[i…i+2^j-1]中最小数的下标。根据一般模式，可以在 O(Nlog$_2$N)的时间内得到这个数组。

然后对于每对 i,j，可以利用一般模式中第一种取用区间的方法，令 k=$\lfloor \log_2(j-i+1) \rfloor$，则比较 A[B[i,k]]与 A[B[j-2^k+1,k]]的大小就可以得到结果了。时间复杂度为 O(1).

[6] 经典问题

这便是巧妙地运用了倍增思想降低了时间复杂度，整个算法也十分简洁明了。

例 4 求树上最近公共祖先问题[7]

给定一棵树，节点数为 n. 每次询问两点 i,j 在树上最近的公共祖先 k，询问次数为 m.

分析：

如果每次直接去寻找两点的公共祖先，时间复杂度为 O(MN)，不能满足要求。那么，如何用倍增思想来解决这道题目呢？

对于每个节点 A，记录 A，A 的第 2^0 层祖先（即父亲），A 的第 2^1 层祖先，A 的第 2^2 层祖先，……，A 的第 $2^{\lfloor \log_2 levelA \rfloor}$ 层祖先。如果按照先序遍历的顺序来记录，就可以利用以前的结果（即利用祖先的计算结果）。根据一般模式，可以得出预处理的时间复杂度是 $O(N\log_2 N)$.

然后，对于每对节点 A，B，所要做的就是求 A 的所有祖先与 B 的所有祖先中层次最小的相同节点。做法是这样的：

二分地找到一个 B 的祖先 X，如果 A 的与 X 同层的祖先即为 X[8]，则可以把范围缩小到[B,B 的父亲，……，X]，否则缩小到[X 的父亲，……，根]。

1、二分地找到 B 的祖先 X

[7] USACO 2004 February Contest Green Problem 3 Distance Queries

[8] 如果 X 的层次比 A 还要大，认为 A 的与 X 同层的“祖先”不等于 X。

首先，令上界为根，下界为 B。因为已经记录了下界 B 的第 2^0 层祖先（即父亲），B 的第 2^1 层祖先，B 的第 2^2 层祖先，……，B 的第 $2^{\lfloor \log_2 levelB \rfloor}$ 层祖先，所以可以令 X 为 B 的第 $2^{\lfloor \log_2 levelB \rfloor}$ 层祖先。当 A 的与 X 同层的祖先 Y 即

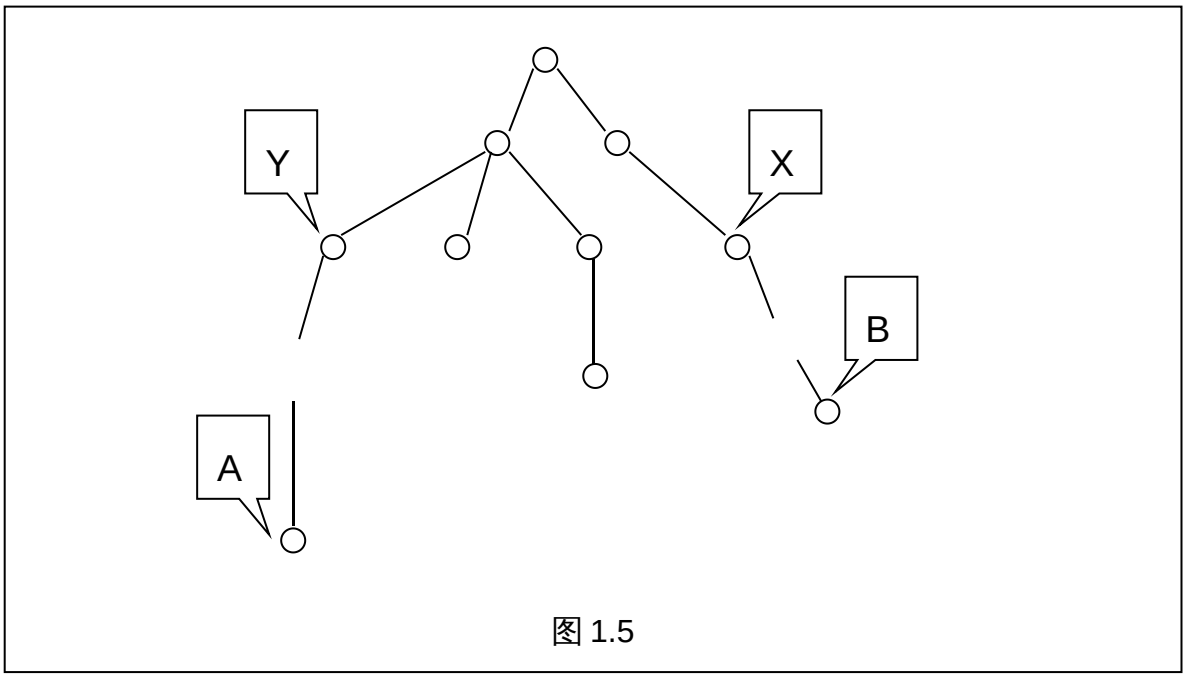


图 1.5

为 X 时，将上界变为 X，否则将下界变为 X 的父亲。因此，这个步骤的时间复杂度为 $O(\log_2 N)$。

2、找到 A 的与 X 同层的祖先

上文提到了如何在区间重叠对所需结果有影响的情况下取用区间[A,B],这里只不过是把 B 换成了 A 的与 X 同层的祖先。所以可以在 $O(\log_2 N)$的时间内找到这个祖先。

步骤 2 中可以加上这样一个优化，即如果本次找到的祖先并非 X，则根据步骤 1，下一次的 X 一定是这一次的 X 的祖先。那么，下一次步骤 2 就可以不从

A 开始找，而是从本次的 X 的父亲开始找。但是这个优化并不能降低最坏情况下的时间复杂度。

综上所述，步骤 1 与步骤 2 各需要 O($\log_2 N$)时间，所以这个算法回答每次询问的时间复杂度是 O($\log_2^2 N$). 预处理的时间复杂度为 O($N\log_2 N$)。总的时间复杂度为 O($N\log_2 N+M\log_2^2 N$)，空间复杂度为 O($N\log_2 N$).

参考文献[1]中给出了一种转化为±1RMQ 求解的方法，可以使得预处理的时间复杂度降为 O(N)，每次回答询问的时间复杂度降为 O(1)，但是操作比较繁琐，这里不再介绍。

以上两个例题说明了倍增思想是如何解决区间操作问题的。虽然在这两题中倍增思想都不是最好的解决办法，但是它的编程复杂度很低，思考起来也不复杂。而在构造后缀数组[9]当中，倍增思想更是发挥了巨大的作用。

在例 4 以及构造后缀数组当中，本身并没有什么明显的“区间”，而都是人为构造出来的。这也体现了倍增思想作为一种思想，并不拘泥于一种死板的模式，而是靠选手通过这种思想来得到巧妙而高效的算法。

一个有趣的探讨

倍增思想的本质是每次将考虑的范围扩大一倍，那么是否可以每次扩大两倍、三倍甚至更多呢？下面就来探讨这个问题。

[9] 详见参考文献[2]

设每次将考虑的范围扩大(k-1)倍(k>=2)的倍增思想为 k 增思想，则本文所介绍的倍增思想为 2 增思想。

对于 k 增思想，最多通过$\lfloor \log_k N \rfloor$次扩大就可以得到所有需要的值，这里为了讨论方便，简化为 $\log_k N$. 但是，为了每次扩大(k-1)倍，必须操作(k-1)次（这里不再考虑使用倍增思想）。所以时间复杂度为 $O((k-1)\log_k N)$。

如 k=3,N=30，则所需要的值（即 3 的幂）为 3 与 27 (30=3+27)。需要 3 次扩大可以得到 1,3,9,27。但是每次扩大需要进行 3-1=2 次操作。如从 3 得到 9，需要进行两次加法 3+3=6，6+3=9（尽管这里也可以通过一次乘法得到 9，但是如果是矩阵乘法或是更加复杂的操作就只能通过(k-1)次操作得到所需要的值）。

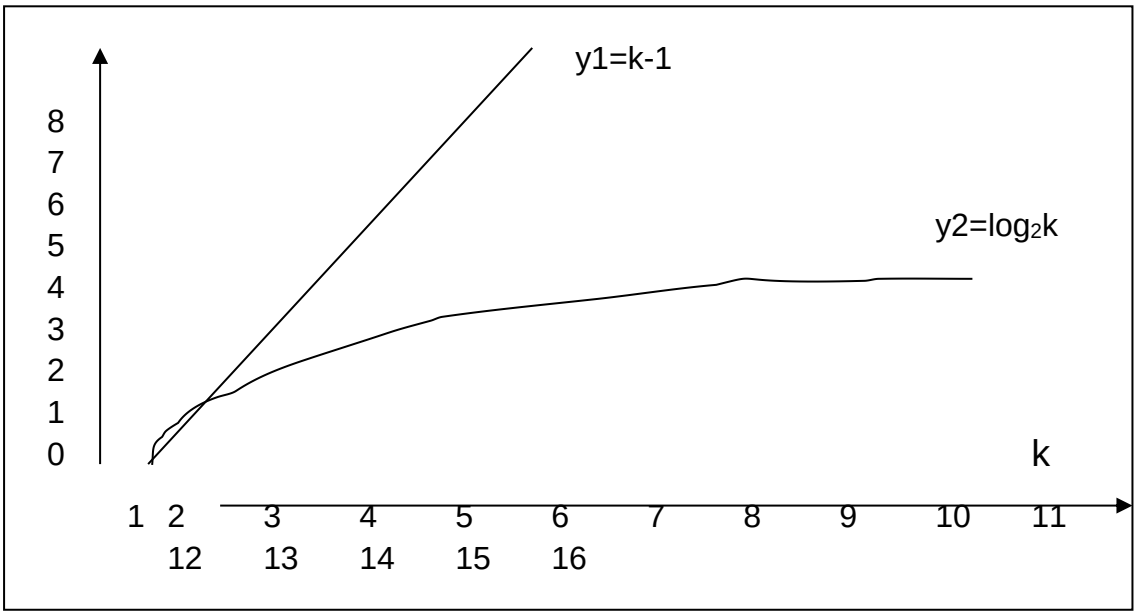
本文中 k=2，因此复杂度为$(2-1)\log_2 N= \log_2 N$.

为了将 k>=3 与 k=2 进行比较。这里采用商值比较法：

$\frac{(k-1)\log_k N}{\log_2 N} = \frac{(k-1)\frac{\log_2 N}{\log_2 k}}{\frac{\log_2 N}{\log_2 2}} = \frac{k-1}{\log_2 k}$. 令 $f(k)=(k-1)-\log_2 k$. 当 k=2 时，$f(k)=0$；当 k>=3 时，$f'(k)=1-\frac{1}{k}\log_2 e = 1-\frac{\ln e}{k\ln 2} = 1-\frac{1}{k\ln 2} >= 1-\frac{1}{3\ln 2} \approx 0.5191$，也就是说当 k>=3 时 f(k)为增函数，即恒有 f(k)>f(2)=0，即$(k-1)-\log_2 k>0$，$(k-1)>\log_2 k$.

当然，这个结论也可以根据图象得到（见下页图 1.6）。图中 $y_1=k-1$ 与 $y_2=\log_2 k$ 的图象共有两个交点(1,0)与(2,1). 而当 k>2 时，y_1 总大于 y_2，所以 $\frac{(k-1)\log_k N}{\log_2 N} = \frac{k-1}{\log_2 k} > 1$ (k>=3) ，即$(k-1)\log_k N > \log_2 N$.

这就从数学角度解释了倍增思想的高效性。



y

总结

本文简要介绍了倍增思想的重要作用以及它在两个方面的应用——在变化规则相同的情况下加速状态转移以及加速区间操作。倍增思想高效的根源在于它恰当地利用了以前的计算结果。从前文的理论分析可以得知，倍增思想将这种利用发挥到了极致。正因为如此，根据倍增思想设计出的算法中没有冗余的运算，使得它能够高效、简洁地解决许多信息学问题。灵活掌握倍增思想，可以使我们在思考问题时能独辟蹊径，快速地找到时空复杂度与编程复杂度都相对较低的算法，从而在紧张的信息学比赛中占得先机。

倍增思想本身作为一种思想，应用是十分广泛的。尽管本文给出了倍增思想的两类应用以及其一般模式，但应该认识到，倍增思想的作用决不仅在于此，其千变万化的实战运用也决非一两个模式可以涵盖。本文作为一篇浅析倍增思想的论文，只是起到抛砖引玉的作用。希望读者能够从中体会到倍增思想

博大精深的内涵，并在实际运用中逐渐积累经验，将倍增思想自然地融入到自己的思考过程当中，指导自己解决各类问题，从而在面对规模巨大的题目时能够做到举重若轻，挥洒自如。当然，这也需要长期的磨练与不断的总结和体会。毕竟，

纸上得来终觉浅，绝知此事要躬行。

感谢

感谢江涛老师阅读我的论文并提出宝贵的修改意见和建议。

感谢刘汝佳教练与许智磊同学，他们与作者就论文写作进行了深入的讨论。

参考文献

[1] 刘汝佳 黄亮，2004，《算法艺术与信息学竞赛》。 北京：清华大学出版社。

[2] 许智磊 IOI2004 国家集训队论文 《后缀数组》

[3] CEPC 2003 Problem D Dice Contest

[4] USACO 2004 February Contest Green Problems

附录

一、文中例子的原题

1、例 2（此处给出许智磊同学翻译并改编后的题目）

Farewell, My Friend!

满分：100

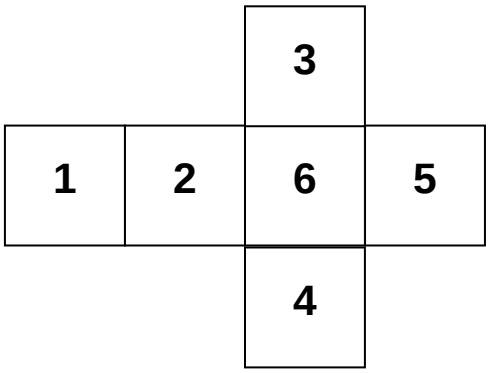
时限：5s

程序：fmf.cpp/fmf.c/fmf.pas　fmf.exe

输入：fmf.in

输出：fmf.out

骰子的空间和我们人类所处的空间很不一样，当然，骰子和人长得也不一



样。骰子是用如下的六个连在一起的面折叠后粘贴起来形成的，六个面的编号如图所示：

粘的方式是把 6 放在桌面上，2,3,4,5 都向上折，1 折到顶部（这需要一些空间想象力），然后再粘起来。骰子的每个面上不光有编号，还有分值，每个骰子六个面上的分值是给定的，分别是 f(1)到 f(6)。

骰子的空间是一个摆放在你的桌面上的带子，这个带子向左延伸至无限长，向右延伸至无限长。带子从前向后（“前”指的是向着你所坐的桌子边缘的方向）被均匀地分成四行，分别用 1 到 4 来编号，每行的宽度恰好是一个骰子的边长。同时，这条带子从左到右被划分成无限多列，这些列用连续的整数来编号，从左到右编号依次增大，每列的宽度恰好也是一个骰子的边长。于是，带子就被分成了无限多个小方格，每个方格恰好全等于骰子的一个面。骰子总

是摆放在某个方格上，且某一个面与方格表面重合。用(x,y)可以代表一个骰子的位置，其中 x 代表所处方格的列编号，y 代表所处方格的行编号，x 是整数，y 是 1,2,3,4 中的某一个。

一个骰子可以向它的前后左右四个相邻的方格移动，移动的方法是以它的底面上的一条边为轴向外滚动 90 度，如果滚动的轴是底面的左边那么它就移动到了左边的相邻格，滚动轴是前边那么它就移动到了前边的相邻格，等等。当然，如果骰子处在第 1 行那它没有前边的相邻格，处在第 4 行则没有后边的相邻格，移动的时候不能超出这两个边界。从一个旧方格移动到相邻的一个新方格的花费是移动以后这个骰子顶部那个面上的分值。

你的朋友，它是一个骰子，目前正处于(x1,y1)，它的 6 号面贴着带子，1 号面处于顶部，2 号面对着前面。它想做一次远行，移动到(x2,y2)上，这是一个冒险性的举动，除了说“再见”之外，你唯一能为它做的就是计算一下它这次远足的最小花费。

输入第一行是六个数，依次给出 f(1)到 f(6)，均在 1 到 50 范围内。第二行四个数 x1,y1,x2,y2，给出了出发地(x1,y1)和目的地(x2,y2)。x1,x2 的绝对值不超过 10^9。输出一行，单独的一个整数 M，代表从(x1,y1)移动到(x2,y2)的最小花费。

样例：

输入	输出
1 2 8 3 1 4	7

2、例 3 一般 RMQ 问题（经典问题）

给定 N 与 N 个整数 A[1…N]，M 与 M 对下标(x,y) (x<=y)，对于每对下标(x,y)求出 A[x…y]中最小数的下标。

3、 例 4 原题

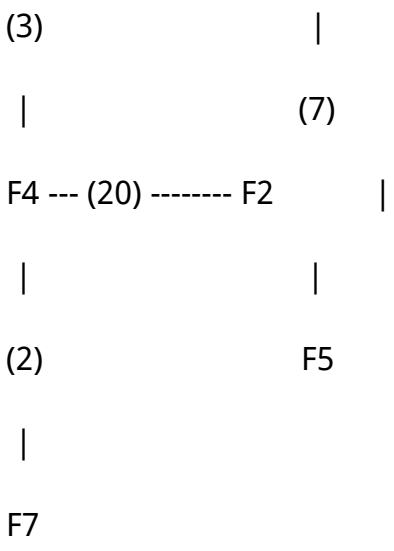
USACO 2004 February Contest Green Problem 3 Distance Queries

Distance Queries [Brian Dean, 2004]

Farmer John's pastoral neighborhood has N farms (2 <= N <= 40,000), usually numbered/labeled 1..N. A series of M (1 <= M < 40,000) vertical and horizontal roads each of varying lengths (1 <= length <= 1000) connect the farms. A map of these farms might look something like the illustration below in which farms are labeled F1..F7 for clarity and lengths between connected farms are shown as (n):

F1 --- (13) ---- F6 --- (9) ----- F3

| |



Being an ASCII diagram, it is not precisely to scale, of course.

Each farm can connect directly to at most four other farms via roads that lead exactly north, south, east, and/or west. Moreover, farms are only located at the endpoints of roads, and some farm can be found at every endpoint of every road. No two roads cross, and precisely one path (sequence of roads) links every pair of farms.

FJ lost his paper copy of the farm map and he wants to reconstruct it from backup information on his computer. This data contains lines like the following, one for every road:

There is a road of length 10 running north from Farm #23 to Farm #17
There is a road of length 7 running east from Farm #1 to Farm #17

...

Farmer John's cows refused to run in his marathon since he chose a path much too long for their leisurely lifestyle. He therefore wants to find a path of a more reasonable length. The input to this problem consists of the description of the farm and is followed by a line containing a single integer K, followed by K "distance queries". Each distance query is a line of input containing two integers, giving the numbers of two farms between which FJ is interested in computing distance (measured in the length of the roads along the path between the two farms). Please answer FJ's distance queries as quickly as possible!

PROBLEM NAME: dquery

INPUT FORMAT:

* Line 1: Two space-separated integers: N and M

* Lines 2..M+1: Each line contains four space-separated entities, F1, F2, L, and D that describe a road. F1 and F2 are numbers of two farms connected by a road, L is its length, and D is a character that is either 'N', 'E', 'S', or 'W' giving the direction of the road from F1 to F2.

* Line 2+M: A single integer, K. 1 <= K <= 10,000

* Lines 3+M..2+M+K: Each line corresponds to a distance query and
    contains the indices of two farms.

SAMPLE INPUT (file dquery.in):

7 6

1 6 13 E

6 3 9 E

3 5 7 S

4 1 3 N

2 4 20 W

4 7 2 S

3

1 6

1 4

2 6

INPUT DETAILS:

This is the farm layout drawn above.

OUTPUT FORMAT:

* Lines 1..K: For each distance query, output on a single line an
    integer giving the appropriate distance.

SAMPLE OUTPUT (file dquery.out):

13

3

36

OUTPUT DETAILS:

Farms 2 and 6 are 20+3+13=36 apart.

二、例 2——例 4 源程序

1、 例 2 程序

```
#include <stdio.h>
#include <stdlib.h>
#define infile "fmf.in"
#define outfile "fmf.out"
#define shapenum 24
#define mid 3
#define distlen 5
#define maxnum 10000000
#define df 1e12

const long long shape[shapenum+1][4]={{0,0,0,0},
                        {0,6,2,4},
                        {0,6,4,5},
                        {0,6,5,3},
                        {0,6,3,2},
                        {0,5,1,3},
                        {0,5,3,6},
                        {0,5,6,4},
                        {0,5,4,1},
                        {0,4,1,5},
                        {0,4,5,6},
                        {0,4,6,2},
                        {0,4,2,1},
                        {0,3,6,5},
                        {0,3,5,1},
                        {0,3,1,2},
                        {0,3,2,6},
```

```
                {0,2,1,4},
                {0,2,4,6},
                {0,2,6,3},
                {0,2,3,1},
                {0,1,5,4},
                {0,1,4,2},
                {0,1,2,3},
                {0,1,3,5}};

const long long opposite[7]={0,6,5,4,3,2,1};
const long long deltax[5]={0,0,1,0,-1};
const long long deltay[5]={0,-1,0,1,0};
const long long gb[5][4]={{0,0,0,0},
                {0,2,-1,3},
                {0,3,2,-1},
                {0,-2,1,3},
                {0,-3,2,1}};

long long dy[7][7][7],f[7],done[distlen+1][5][shapenum+1],
    dist[distlen+1][5][shapenum+1],zhy[5][shapenum+1][5][shapenum+1],
    now[5][shapenum+1],x1,y1,x2,y2;
FILE *fin=fopen(infile,"r"),
    *fout=fopen(outfile,"w");

void change(long long sign,long long x,long long y,long long z,long long *nx,long long
*ny,long long *nz)
{
 long long a[4],b[4],i;
 (*nx)=x+deltax[sign];
 (*ny)=y+deltay[sign];
 for (i=1; i<=3; i++)
  a[i]=shape[z][i];
 for (i=1; i<=3; i++)
  {
   if (gb[sign][i]<0)
    b[i]=opposite[a[-gb[sign][i]]];
    else b[i]=a[gb[sign][i]];
  }
 (*nz)=dy[b[1]][b[2]][b[3]];
}

void calc_dist_zhy()
//calculate the initial regulation of changes---zhy(a[1])
{
```

```
long long a,b,i,j,k,x,y,z,nx,ny,nz,xynum,min,qq,zeng;
qq=3*4*24;
for (a=1; a<=4; a++)
 for (b=1; b<=24; b++)
 {
  for (i=1; i<=distlen; i++)
   for (j=1; j<=4; j++)
    for (k=1; k<=shapenum; k++)
     {
      dist[i][j][k]=df;
      done[i][j][k]=0;
     }
  dist[mid][a][b]=0;
  xynum=0;
  while (xynum<qq)
   {
    min=df;
    for (i=1; i<=distlen; i++)
     for (j=1; j<=4; j++)
      for (k=1; k<=shapenum; k++)
       if ((done[i][j][k]==0)&&(dist[i][j][k]<min))
        {
         min=dist[i][j][k];
         x=i;
         y=j;
         z=k;
        }
    if (min==df)
     break;
    done[x][y][z]=1;
    if ((x>=mid-1)&&(x<=mid+1))
     xynum++;
    for (i=1; i<=4; i++)
     {
      change(i,x,y,z,&nx,&ny,&nz);
      if ((nx>=1)&&(nx<=distlen)&&(ny>=1)&&(ny<=4))
       if (done[nx][ny][nz]==0)
        {
         zeng=f[opposite[shape[nz][1]]];
         if (dist[x][y][z]+zeng<dist[nx][ny][nz])
          dist[nx][ny][nz]=dist[x][y][z]+zeng;
        }
     }
   }
```

```
    if (x1<x2)
     {
      for (i=1; i<=4; i++)
       for (j=1; j<=shapenum; j++)
        zhy[a][b][i][j]=dist[mid+1][i][j];
     }
    if (x1>x2)
     {
      for (i=1; i<=4; i++)
       for (j=1; j<=shapenum; j++)
        zhy[a][b][i][j]=dist[mid-1][i][j];
     }
    if (x1==x2)
     {
      if ((a==y1)&&(b==1))
       {
        min=df;
        for (i=1; i<=shapenum; i++)
         if (dist[mid][y2][i]<min)
          min=dist[mid][y2][i];
        fprintf(fout,"%Ld\n",min);
        //change!!!!!!!!!!!!!!!!!!!!
        fclose(fout);
        exit(0);
       }
     }
   }
}
void init()
{
 long long i;
 for (i=1; i<=6; i++)
  fscanf(fin,"%Ld",&f[i]);
 fscanf(fin,"%Ld%Ld%Ld%Ld",&x1,&y1,&x2,&y2);
 //change!!!!!!!
 fclose(fin);
 for (i=1; i<=shapenum; i++)
  dy[shape[i][1]][shape[i][2]][shape[i][3]]=i;
 calc_dist_zhy();
}

void zhuan1()  //to update now---the specific state
{
 long long i,j,k,h,temp[5][shapenum+1];
```

```
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   temp[i][j]=df;
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   if (now[i][j]!=df)
    for (k=1; k<=4; k++)
     for (h=1; h<=shapenum; h++)
      if ((now[i][j]+zhy[i][j][k][h]<temp[k][h]))
       temp[k][h]=now[i][j]+zhy[i][j][k][h];
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   now[i][j]=temp[i][j];
}

void zhuan2()  //to update zhy---the regulation of changing
{
 long long i,j,k,h,a,b,qq,temp[5][shapenum+1][5][shapenum+1];
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   for (k=1; k<=4; k++)
    for (h=1; h<=shapenum; h++)
     temp[i][j][k][h]=df;
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   for (k=1; k<=4; k++)
    for (h=1; h<=shapenum; h++)
     if (zhy[i][j][k][h]!=df)
      for (a=1; a<=4; a++)
       for (b=1; b<=shapenum; b++)
        if (zhy[i][j][k][h]+zhy[k][h][a][b]<temp[i][j][a][b])
         temp[i][j][a][b]=zhy[i][j][k][h]+zhy[k][h][a][b];
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   for (k=1; k<=4; k++)
    for (h=1; h<=shapenum; h++)
     zhy[i][j][k][h]=temp[i][j][k][h];
}

void work()
{
 long long x,i,j,temp;
 x=x2-x1;
 if (x<0)
```

```
  x=-x;
 for (i=1; i<=4; i++)
  for (j=1; j<=shapenum; j++)
   now[i][j]=df;
 now[y1][1]=0;
 while (x>0)
  {
   temp=x%2;
   if (temp==1)
    zhuan1();
   zhuan2();
   x/=2;
  }
}

void output()
{
 long long i,result;
 result=df;
 for (i=1; i<=shapenum; i++)
  if (now[y2][i]<result)
   result=now[y2][i];
 fprintf(fout,"%Ld\n",result);
 //change!!!!!!!!!!!!!!!!!
 fclose(fout);
}

int main()
{
 init();
 work();
 output();
 return 0;
}
```

2、 例 3 程序

```
#include <stdio.h>
#include <math.h>
#define infile "rmq.in"
#define outfile "rmq.out"
#define maxn 50000
#define maxlogn 17
```

```
long a[maxn+1],b[maxn+1][maxlogn+1],mi[maxlogn+1],logn,n,m,x,y;

FILE *fin=fopen(infile,"r"),
     *fout=fopen(outfile,"w");

void init()
{
 long i;
 fscanf(fin,"%ld",&n);
 for (i=1; i<=n; i++)
  fscanf(fin,"%ld",&a[i]);

 logn=0;
 mi[0]=1;
 while (mi[logn]*2<=n)
  {
   logn++;
   mi[logn]=mi[logn-1]*2;
  }
}

long min(long sa,long sb)
{
 if (a[sa]<a[sb])
  return (sa);
  else return (sb);
}

void calc_b()
{
 long i,j;
 for (i=1; i<=n; i++)
  b[i][0]=i;
 for (j=1; j<=logn; j++)
  for (i=1; i<=n; i++)
   if (i+mi[j-1]>n)
    b[i][j]=b[i][j-1];
    else b[i][j]=min(b[i][j-1],b[i+mi[j-1]][j-1]);
}

void deal_with_query()
{
 long i,k,result;
 fscanf(fin,"%ld",&m);
```

```
 for (i=1; i<=m; i++)
  {
   fscanf(fin,"%ld%ld",&x,&y);
   k=(long) (log2(y-x+1));
   result=min(b[x][k],b[y-mi[k]+1][k]);
   fprintf(fout,"%ld\n",result);
  }
 fclose(fin);
 fclose(fout);
}

void work()
{
 calc_b();
 deal_with_query();
}

int main()
{
 init();
 work();
 return 0;
}
```

3、 例 4 程序

```
/*
PROG: dquery
LANG: C++
*/
#include <fstream.h>
#define infile "dquery.in"
#define outfile "dquery.out"
#define maxn 41000
#define maxm 41000
#define maxlogn 20

struct anedge{
            long x,y,len;
           }edge[maxm+1];

struct qqedge{
            long node,len;
           }*g[maxn+1];
```

```
struct jl{
         long node,dist;
        }df[maxn+1][maxlogn+1];

long degree[maxn+1],xl[maxn+1],level[maxn+1],
    father[maxn+1],done[maxn+1],top[maxn+1],
    n,m,result,asknum;              //top[i] stands for the size of df[i].

ifstream qin(infile);

long calc_father(long node)
{
 long j,k,temp;
 j=node;
 while (father[j]>0)
  j=father[j];
 k=node;
 while (k!=j)
  {
   temp=father[k];
   father[k]=j;
   k=temp;
  }
 return (j);
}

void combine(long a,long b)
{
 father[b]+=father[a];
 father[a]=b;
}

void calc_xl()
//calculate the pre-order of nodes in the tree and get the level of every node
{
 long i,j,now,node,head,tail,qq;
 for (i=1; i<=n; i++)
  level[i]=father[i]=done[i]=0;
 head=0;
 tail=1;
 xl[1]=done[1]=level[1]=1;
 top[1]=0;
 while (head<tail)
```

```
  {
   head++;
   node=xl[head];
   for (j=1; j<=degree[node]; j++)
    {
     qq=g[node][j].node;
     if (done[qq]==0)
      {
       done[qq]=1;
       father[qq]=node;
       tail++;
       xl[tail]=qq;
       level[qq]=level[node]+1;

       top[qq]=1;
       df[qq][1].node=node;
       df[qq][1].dist=g[node][j].len;
       i=node;
       now=1;
       while (1)
        {
         if (now>top[i])
          break;
         top[qq]++;
         df[qq][top[qq]]=df[i][now];
         df[qq][top[qq]].dist+=df[qq][top[qq]-1].dist;
         i=df[i][now].node;
         now++;
        }
      }
    }
  }
}

void init()
{
 long i,j,x,y,fx,fy,len,nm;
 char c;
 qin>>n>>m;
 for (i=1; i<=n; i++)
  {
   father[i]=-1;
   degree[i]=0;
  }
```

```
 nm=0;
 for (i=1; i<=m; i++)
  {
   qin>>x>>y>>len>>c;
   fx=calc_father(x);
   fy=calc_father(y);
   if (fx!=fy)
    {
     nm++;
     edge[nm].x=x;
     edge[nm].y=y;
     edge[nm].len=len;
     degree[x]++;
     degree[y]++;
     combine(fx,fy);
    }
  }
 m=nm;
 for (i=1; i<=n; i++)
  {
   g[i]=new (struct qqedge [degree[i]+2]);
   g[i][0].node=0;
  }
 for (i=1; i<=m; i++)
  {
   x=edge[i].x;
   y=edge[i].y;
   len=edge[i].len;
   g[x][0].node++;
   g[x][g[x][0].node].node=y;
   g[x][g[x][0].node].len=len;
   g[y][0].node++;
   g[y][g[y][0].node].node=x;
   g[y][g[y][0].node].len=len;
  }
 calc_xl();
}

void suan(long dian,long ceng,long *node,long *len)
//calculate an ancestor of “dian” and the ancestor is at level “ceng”
{
 long now,i;
 if (ceng>level[dian])
  {
```

```
  (*node)=(*len)=-1;
  return;
 }
 if (ceng==level[dian])
 {
  (*node)=dian;
  (*len)=0;
  return;
 }
 now=dian;
 (*len)=0;
 while (level[now]>ceng)
 {
  i=1;
  while ((i<=top[now])&&(level[df[now][i].node]>=ceng))
   i++;
  i--;
  (*len)+=df[now][i].dist;
  now=df[now][i].node;
 }
 (*node)=now;
 return;
}

void calc_result(long x,long y)
//calculate the sum of the distance from x to z and from y to z(z is the nearest common
ancestor of node x and y)
{
 long l1,l2,n1,n2,start,stop,mid;
 if (x==y)
 {
  result=0;
  return;
 }
 start=1;
 stop=level[y];
 result=0;
 while (start<=stop)
 {
  mid=(start+stop)/2;
  suan(y,mid,&n1,&l1);
  suan(x,mid,&n2,&l2);
  if (n1==n2)
   start=mid;
```

```
    else {
          stop=mid;
          y=n1;
          result+=l1;
         }
   if (start==stop)
    {
     result+=l2;
     break;
    }
   if (start+1==stop)
    {
     suan(y,stop,&n1,&l1);
     suan(x,stop,&n2,&l2);
     if (n1==n2)
      {
       result+=l1+l2;
       return;
      }
     suan(y,start,&n1,&l1);
     suan(x,start,&n2,&l2);
     if (n1==n2)
      {
       result+=l1+l2;
       return;
      }
     return;
    }
  }
}

void work()
{
 long i,x,y;
 qin>>asknum;
 ofstream cout(outfile);
 for (i=1; i<=asknum; i++)
  {
   qin>>x>>y;
   calc_result(x,y);
   cout<<result<<endl;
  }
}
```

```
int main()
{
 init();
 work();
 return 0;
}
```

压去冗余 缩得精华

——浅谈信息学竞赛中的“压缩法”

安徽 周源

摘要

在信息学竞赛中，我们经常遇到这样一类问题：数据规模大，或是数据间的关系复杂，总而言之即输入数据的信息量过大。

作者在综合分析了很多信息学竞赛试题后，提出了“压缩法”这个概念：即压去输入数据中的冗余信息，保留下对解决问题有帮助的精华部分。压缩法在信息学竞赛中有着广泛的应用，但在各类问题中却有看起来截然不同的表现形式，本文即将选择多道有代表性的例题，提炼它们的共同点，提出压缩法适用问题的两个要点。最后，作者将在总结部分着重分析压缩法的工作特点，认为压缩法是在化归思想的基础上，加上了“信息化”的因素，通过合理的利用信息达到化简问题的目的。

关键字

信息学竞赛 压缩法

冗余/精华信息

可压缩性 替代法则 化归思想 信息化

目录

引子

可能不少同学都对题目中“压缩法”这个名词感到很陌生，不错，因为这是作者自己发明的一个名词☺。这篇论文就将带领大家走近我所说的“压缩法”，熟悉“压缩法”。

看到“压缩”二字，可能有的同学会想到我们常用的压缩软件，如 WinZip, WinRAR 等等，他们都是想尽办法的减小文件占用的空间来为我们服务。这正是压缩一词的本义，即“加以压力,以减小体积、大小、持续时间、密度和浓度等”[10]。

然而本文中要讨论的“压缩法”的含义则为：“除去冗余信息，留下精华，以减小问题的规模”，看得出，这正是为信息学竞赛量身定制一种好方法。

下面，就让我们看看压缩法在竞赛中的精彩表现吧！

压缩法的定义

一般来说，当我们处理问题时常常遇到一个集合 *S*，*S* 内部各个元素之间的关系对问题的结果不造成影响，而且也不会受到外部因素的干扰，那么我们可以将 *S* 看作一个内外隔绝的**包裹（package）**，忽略包裹内部的冗余信息，并将这个包裹与外部事物的联系保留，打包、**压缩后**作为一个新的元素存储下来以供以后分析处理。

这就是压缩法工作的基本流程，其好处在于略去了包裹内部种种纷扰却无用的信息，从而化简了问题，进而用更好的方法去解决问题。

[10] 摘录自《高级汉语大词典》。

压缩法的简单实例

其实我们对压缩法并不陌生，相信大家对下面两个例子都有一定的了解。

[例一]多源最短路问题（经典问题）

如下图。在一个加正权的有向图 $G = \{V, E\}$中，给出源的位置，求源到其余所有点的最短路长度。与一般最短路问题不同的是，本题中源是一个集合 S 中的所有点。而 S 到某一个点 p 的最短距离等于 S 中所有点与 p 最短距离的最小值。[11]

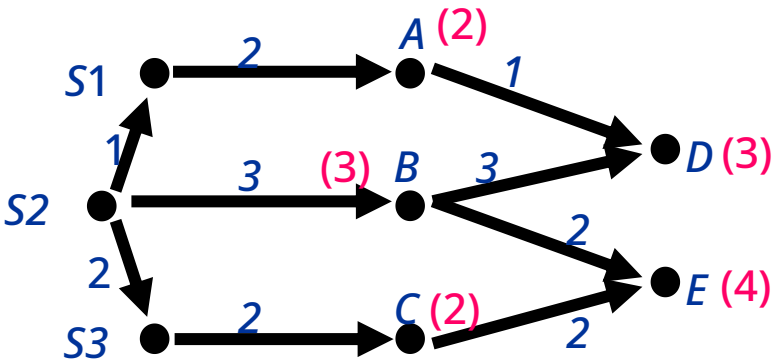


图 1

[分析]

这道题目的关键在于有多个源，而为了提高效率，只能执行一次 Dijkstra 算法。

其实这是很简单的，很多不同的方法都可以做到这一点。下面让我们看一看压缩法是怎么实现的：

可以看出，由于不可能有一条最短路径从一个源点出发却又经过另外一个源点，因此源点集合 S 中任何两点间的连边关系对答案都没有影响。如上文

[11] 图中边上的数值为边的权值，顶点后括号内的数值为到该点最短路的长度。

所述，可以将 S 视为一个内外隔绝的“包裹”，舍去包裹内的冗余信息，并将其“压缩”成为一个新的点 P_S。

另一方面，我们还需要保留 S 中节点对外的连边情况作为压缩后节点 P_S 的对外连边。

这样，我们就成功的完成了压缩流程，得到的是一张新图和一个单源最短路问题：这正是 Dijkstra 所做的。

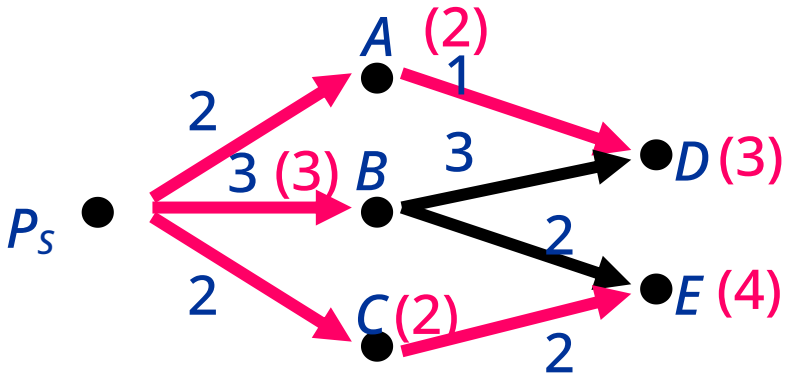


图 2

[例二]球队问题（经典问题）

给出某个篮球队的球员通讯图 $G = \{V, E\}$如下，若存在有向边(u, v)表示球员 u 可将消息及时告诉 v，若教练想将一条紧急消息告知给全体队员，利用这个通讯图，他至少要亲自通知几个队员？

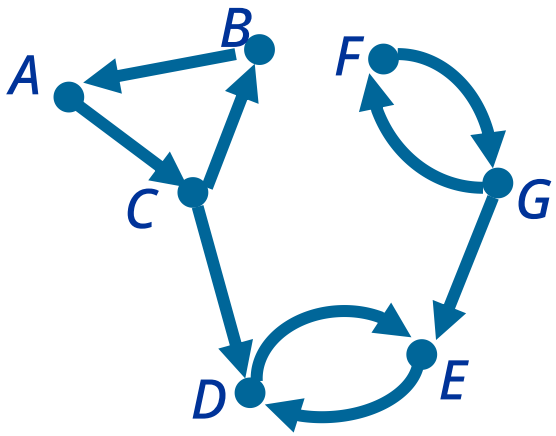


图 3

[分析]

看着这个杂乱无章的通讯图，我们决定还是使用压缩法“压去”一些不必要的关系，简化题设条件。

分析图中的每一个强连通分量 S，若 S 中的某一个球员得知了消息，那么显然 S 中其他所有的队员都可以及时获得消息。即 S 中每一个球员是否得知消息的状态都是相等的，如果有必要的话，教练最多只会通知 S 中的一个球员。此时保留 S 中的通讯边已经毫无意义，我们可以舍去它们，并将 S 压缩成为一个点 P_S，并保留 S 中每个球员的对外通讯情况作为 P_S 的连边情况。

经过对每一个强连通分量的压缩处理后，我们得到了一个全新通讯图如下。由于所有的强连通分量都被压缩成为一个点，新图中已经没有环的存在。

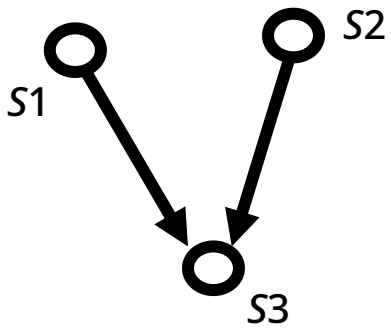


图 4

此时问题就变得很明朗了：没有入度的点（可能是一个强连通分量）是不可能被别的球员通知到的，因此教练必须亲自通知；另一方面，由于没有环的存在，每一个有入度的点，都至少可以被一个没有入度的点直接或者间接的通知到，因此教练没有必要亲自通知他们。

综上所述，教练需要通知的最少人数即为新图中没有入度的点的个数。问题得到解决。

可以看出，在上述两个例子中压缩法都有着出色的表现，而这两个例子又以图论中的“缩点法”为我们熟知。然而压缩法的应用并不仅仅局限于图论算法中，只要问题满足一定的条件，我们都可以向压缩法求助。

压缩法的要点

1. 可压缩性

可压缩性是压缩法可行性的理论保障。

如果可以将压缩法应用于一个问题，这个问题首先应当存在某一种或者多种可压缩性。可压缩性即是问题本身的某些元素集合在满足特定条件的情况下可以被压缩的性质，如上述两例中的源点集合和强连通分量。

一般来说，一个集合 S，若其内部各个元素之间的联系不会和外部元素相互作用而影响到问题的结果；则我们可以忽略 S 内的相互联系，从而将它们**压缩**为一个新的个体，即 ***S* 可压缩**。

在[例一]中，任意两个源点间的可到达关系对答案没有影响。鉴于此我们可以**舍去**源点集合 S 中两两节点间的**相互联系**，将其压缩，变成一个单一的源点考虑，**即 *S* 满足可压缩性**。

而在[例二]中，我们说过，每一个强连通分量中的球员是否得知消息的状态始终是相等的：无论这些球员相互的通讯情况如何，都不会改变这个事实。因此它们都是**可压缩的**，我们**舍去**分量内球员的**通讯情况**，将它们**压缩**成为一个个新的节点，考虑简化后的问题。

2. 替代法则

需要注意的是，压缩不是完全的忽略，可压缩性也不是可删除性。压缩法，顾名思义，“压”即为可以压去被压缩集合 S 内部各个元素的个体性质和它们的相互关系，这是冗余信息，对问题的结果并没有影响；而“缩”则是缩得精华，保留下集合 S 作为一个点与外部信息的联系。

如果说可压缩性体现了“压”的过程，那么“缩”则是替代法则的任务。

所谓替代法则，即用什么样的形式，表现出精华部分的内容。

在[例一]和[例二]中，都使用了最简单的替代法则：用一个新的点代替多个点的集合，并用相同形式的有向边代替原先每个点和集合外元素的连边。压缩前后的元素形式完全一致。而在一些复杂化的问题中，我们则需要认真考虑这个问题：压缩后保留下来了些什么，用什么形式保留？

不妨来看这样一个小例子。

[例三]模方程组的替代法则（经典问题）

我们在解一元模方程组的时候，也可以使用压缩法：通过将两两方程压缩减小问题的规模。方程压缩的可行性是毋庸置疑的，下面就让我们研究一下这里压缩的替代法则。

假设有两个方程需要压缩：

$$\begin{cases} a_1 x \equiv c_1 (\bmod b_1) \\ a_2 x \equiv c_2 (\bmod b_2) \end{cases}$$

而要求的替代法则就是用具有相同解集的单一方程来代替。

可将上面两个方程写作二元一次不定方程：

$$\begin{cases} a_1 x + k_1 b_1 = c_1 \\ a_2 x + k_2 b_2 = c_2 \end{cases}$$

将它们视为单独的方程，分别解出其中的一个解 x_1 和 x_2，则可将以上两个方程写作：

$$(*)\begin{cases} x = x_1 + p_1 \Delta_1 \\ x = x_2 + p_2 \Delta_2 \end{cases}$$

其中

$$\begin{cases} \Delta_1 = (a_1, b_1) b_1 \\ \Delta_2 = (a_2, b_2) b_2 \end{cases}$$

联立(*)式，仍是一个二元一次不定方程：

$$x_1 + p_1 \Delta_1 = x_2 + p_2 \Delta_2$$

可以得到 x 的一个解 x_0，那么就可以将 x 的通解写作：

$$x = x_0 + k[\Delta_1, \Delta_2]$$

即：

$$x \equiv x_0 (\mathrm{mod}\,[\Delta_1, \Delta_2])$$

至此，我们就用上面的一个模方程代替了压缩前的两个模方程：这就是本例中的替代法则。

至此我们已经初步熟悉了压缩法的两个要点，在很多竞赛问题中，只要我们能够找出这两个要点的具体体现，就等于找到了压缩法的着力点，进而可以用压缩法解题。

压缩法的应用

[例四]欧元兑换（BOI 2003）

你每天会收到一些欧元，可能为正数也可能是负数，银行允许你在某天将手头所有的欧元兑换成 lei。第 i 天的兑换比率是 1 euro ：i lei，同时你必须在多付出 T lei 被银行收取。在 N 天你必须兑换所有持有的欧元。要求找一个方案使第 N 天结束时能得到最多的 lei。

数据范围：

1 ≤ N ≤ 34 567
0 ≤ T ≤ 34 567

[分析]

首先让我们把这题的任务转化为较为简捷的数学语言。即给定一个费用 *T* 以及 *N* 个整数（**有可能是负数**）：

$$a_1, a_2, a_3, \ldots, a_N$$

要求一种划分方案：

$$1 \le c_1 < c_2 < \ldots < c_k = N$$

即分为连续的 k 份，第 i 份的所有欧元都在第 c_i 天兑换，不妨将 c_i 称为**兑换点**。

从而使目标函数

$$Z = (\sum_{i=1}^{c_1} a_i) * c_1 - T + (\sum_{i=c_1+1}^{c_2} a_i) * c_2 - T + \ldots\ldots + (\sum_{i=c_{k-1}+1}^{c_k} a_i) * c_k - T$$

最大化。

[动态规划的矛盾]

看上去这是一道完全的动态规划问题，与压缩法没有什么关系，因为很显然我们都会构造出这样一个动态规划算法：

令 $f(n)$表示在 1~n 天的一个子问题：即以第 n 天($n \le N$)为最后一个兑换点时的最大收益。则有边界条件

$$f(0) = 0$$

通过枚举上一个兑换点 $i<n$，得到动态规划方程

$$f(n) = \max_{0 \le i < n} \{ f(i) + (\sum_{j=i+1}^{n} a_j) * n - T \}$$

不妨设 S 为 a_i 数列的部分和，则可以简化上面的式子：

$$f(n) = \max_{0 \le i < n} \{ f(i) + (S_n - S_i) * n - T \}$$

至此，我们已经发现这个动态规划算法虽然时间复杂度是 $O(N^2)$的，并不能胜任题目的数据规模。通过一些分析，参考作者去年的论文[12]，我们知道了类似的动态规划方程式可以被优化的：如下图，在计算 $f(n)$时可将从前的每一个决策状态对应为平面上的一个个点

$$[\ S_i\ ,\ f(i)\ ]$$

[12] 具体方法参见附录以及作者去年的集训队论文。

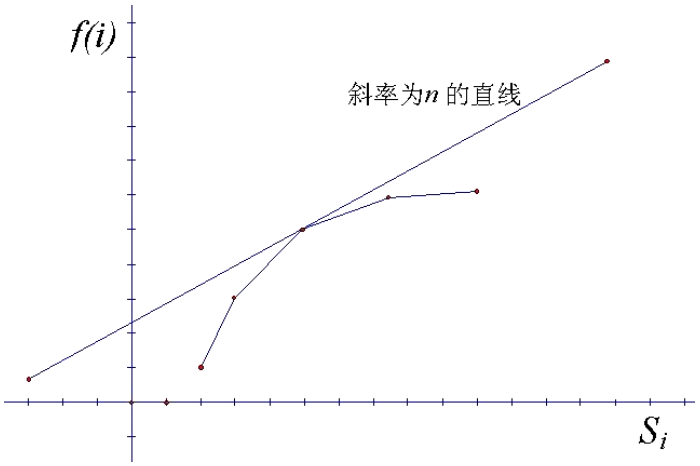


图 5

随着 n 的递增，依次将这些点加入一个下凸函数，并维护该函数，利用下凸折线斜率的单调性加速决策过程，从而可以得到一个 O(N)的方法。

然而新的问题又出现了：由于本题中的 a_i 可以是负数，也就是说部分和函数 S_i 并不是递增的，甚至不是非降的！那么在计算动态规划函数的过程中，随着 n 的递增，S_n 有可能变小，那么对应的点[S_n ， $f(n)$]将可能被加入到折线的“中间”，如下图所示，这样维护这个折线将变得异常困难！

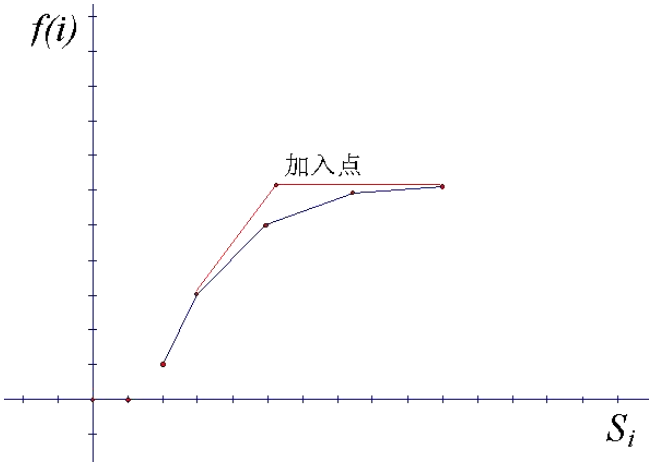


图 6

此时我们的分析就走到的本题矛盾的激化点：传统的斜率优化法要求操作函数 S_n 是单调的（至少非严格单调）；而题目中给出的条件却允许该函数任意的变化！

山重水复疑无路，两者直接的、完全的冲突似乎彻底否定了这个做法。而压缩法正是在此时大显身手，导演了一出柳暗花明又一春的好戏。

[压缩法化解矛盾]

让我们仔细的回顾一下题目，其中有这样一句话：“**第 i 天的兑换比率是 1 euro ：i lei**”，其含义则是，随着天数的递增，欧元将变得越来越值钱。那么不难想象出，如果当前手中的钱是正数，当然不急着将其兑换成 lei：过些日子将会兑换到更多；而如果当前手中的钱是负数，那么应该尽量早的将其卖出：否则过些天将要赔出更多的 lei，但尚要注意，还应保证总兑换次数较少：由于每次兑换都要花 T lei。

以上感性的认识就勾画出了我们“压缩”算法的一个大致轮廓。不难通过理论证明，得到如下定理：

[定理 4.1]

若{a_i}序列中有一段连续的子序列 a_i, a_{i+1}, …, a_j（$i<j$），满足该子序列的部分和

$$a_i,$$
$$a_i + a_{i+1},$$
$$a_i + a_{i+1} + a_{i+2},$$
$$\dots$$
$$a_i + a_{i+1} + \dots + a_{j-1}$$

均为正数，而总和

$$a_i + a_{i+1} + \dots + a_j$$

非正数。

则当在第 i 天收获了 a_i 欧元后，一定有某个最优方案不急于兑换，而是继续在第(i+1)天收得 a_{i+1} 欧元，第(i+2)天收得 a_{i+2} 欧元，直到第 j 天收到 a_j 欧元，再看情况决定是否兑换。

从感性上看，这个定理显然是正确的。但还是让我们用理论证明一下：

[证明 4.1]

用反证法，考虑最简单的情况，假设某个最优方案在[i, j)区间中，仅有一个兑换点：在第 k 天（$i \le k < j$）。[13]

那么设从上次兑换点到第(i-1)天为止，共余下了 x 欧元，如下图所示：

1. x 为正数[14]：那么显然到第 k 天的兑换点我们将余下更大的正数，取消这个兑换点，将其并入后一个兑换点一起兑换，既提高了手中欧元的价值，又省去了一次兑换手续费 T，何乐而不为？
2. x 为负数：将这个兑换点提前至第(i-1)天，提前脱手了负数的欧元，显然可以付出较少的 lei 的代价；而将[i, k]区间内的正数归入下一个兑换点，也可以得到更大的收益。

综上两种情况，无论如何我们都可以取消或者移走这个兑换点，却提高了收入，这与假设命题中方案的最有性不符，推出矛盾，从而[定理 4.1]得证。

[定理 4.1]即说明了本题中存在的可压缩性：它实际上告诉我们，如果 a_1 是正数，则第一天结束时一定不急于兑换，等到第二天有 a_2 的收入，若当前手中的欧元($a_1 + a_2$)仍是正数则还是不兑换，再等到第三天……直到第 k 天结束若尚未兑换的欧元

[13] 多次兑换的同理可以证明。

[14] 或者可以是 0。

$$a_1 + a_2 + \ldots + a_p$$

不再是正数，再考虑是否兑换。这么多天的收入在兑换问题上是等价的，它们作为一个个个体的性质是相同的，因为根据定理他们一定都在同一个兑换点被兑换。即可以把这一段数字**压缩成一个点**：令

$$b_1 = a_1 + a_2 + \ldots + a_p$$

表示我们认为这 p 天的欧元是在同一个时间点 b_1 收到的，可以作为一个压缩后的点与其它每个收入点**同样的**被处理：稍有不同的是，这些钱的兑换要在 $t_1 = p$ 天后进行，而不是原来的每天后。所以我们的替代法则为：用二元组(b_1, t_1)代替原来 p 天的收入。

这样，我们继续考察 a_{p+1} 开始的每个点，累计从该点开始收入的欧元，直到新的点 a_q 出现了非正数的和，那么就将 a_{p+1} 直至 a_q 这些$(q-p+1)$个点压缩成为一个新的收入点：

$$b_2 = a_{p+1} + a_{p+2} + \ldots + a_q$$

同时记录下 $t_2 = q$，表示压缩处理后第二个兑换点的具体时间。用(b_2, t_2)代替。

这样一次处理下去，我们可以得到以下 M 个压缩后的收入点和兑换点：

$$(b_1, t_1), (b_2, t_2), (b_3, t_3), \ldots, (b_M, t_M)$$

值得强调的是，最后处理到 a_N 时可能会有手中欧元为正数的情况，将这些钱在第 N 天（最后一天）兑换即可获得它们的最大价值。

将目光转移到压缩后的任务上，和原任务类似，我们还是要求对数列 b_i 一个最优分割兑换方案，构造它们的部分和 S'并写出新的动态规划方程：

$$g(m) = \max_{0 \le i < m} \{ g(i) + (S'_m - S'_i) * t_m - T \}$$

可以看出由于任意 $b_i \le 0$，部分和 S'_m 一定是非上升序列：优化决策过程的条件“S'_m 的非严格单调性”已经满足！

那么利用附录所述的方法，可以构造出 O(*M*)的动态规划算法，再加上之前的 O(*N*)的压缩预处理。我们得到了一个 O(*N* + *M*) = O(*N*)的线性算法！

至此，通过压缩处理，本题已经圆满解决。

[小结]

回顾一下[例四]的分析过程，我们已经掌握了一些基本的动态规划优化方法，如本题中的“斜率优化法”。解决类似的问题本应不难，然而题设中一个障碍条件“每次收入的欧元 a_i 正负任意”却与斜率优化的原理即操作函数的单调性产生了极大的冲突。看似矛盾不可调和，然而利用压缩法，将连续的部分和是正数的收入子序列“压缩”，直到成为一个不大于零的收入点，奇迹般的化解了矛盾，打开了僵局，为下一步优化营造了条件。压缩法在本题中证明了每一个待压缩收入集合中的收入点在兑换问题上的等价性，继而可以压缩，忽略它们作为个体的性质，从而达到简化问题的目的。

[例五]合并数列问题（ZOJ p1794 Merging Sequence Problem 改编）

给出 *K* 个数列，假设每个数列的长度分别为 n_1，n_2，n_3，…，n_K。而这 *K* 个数列分别为：

$$a_{1,1}, a_{1,2}, a_{1,3}, \cdots, a_{1,n_1}$$
$$a_{2,1}, a_{2,2}, a_{2,3}, \cdots, a_{2,n_2}$$
$$\vdots$$
$$a_{K,1}, a_{K,2}, a_{K,3}, \cdots, a_{K,n_K}$$

注意这里每个数都可能是负数。

将这 K 个数列**归并**为一个新的数列 S，所谓归并，即选择一个非空的输入数列，将其第一项放入 S 的尾部，并在数列中删去该项；重复这一过程，直到所有的输入数列都是空数列。

要求寻找一个归并后的数列 S，新的数列 S 的长度为 $N=n_1 + n_2 + n_3 + \ldots + n_K$。且 S 的各个部分和

$$Sum_0, Sum_1, Sum_2,\ldots,Sum_N$$

中，最大的最小。

数据范围：

$K \le N \le 10^5$。

[分析]

初看这题似乎只有 $O(K*N^K)$的动态规划算法，但这个算法太不现实了，下面就让我们有意识的尝试用压缩法来分析这题。

[观察压缩要点]

首先我们看一下这题中的可压缩性具体表现为什么：

假设某一个数列中存在一段连续的子串 u，

$$u_1, u_2, u_3, \ldots, u_p$$

如果可以证明当归并过程中选择了 u 的第一项放入 S，就一定可以连续的选至 u 的最后一项而不会丢失寻找最优解的可能性：即 u 一定会在某个最优串 S 中连续出现。根据可压缩性的定义，如果 u 满足上述性质，那么这一段子串内部各项的互相联系即它们的相互位置关系已经失去价值，也就是说，u 内各项的相互联系不会与外部因素相互作用而影响最优结果，即 u 可压缩。

再分析一下本题中的替代法则，分析如何保留这一段子列对其他数值的影响。

首先，如下图，*u* 子串中的某一个部分和可能成为 *S* 序列里最大的部分和：不过也只有 *u* 的最大部分和即它在数列中的**相对峰值**有这个可能性。

第二，只要 *u* 子串不是在 *S* 数列的末尾，其总和一定会对以后的部分和产生影响，间接的影响到部分和的最大值：这也是我们所关心的。

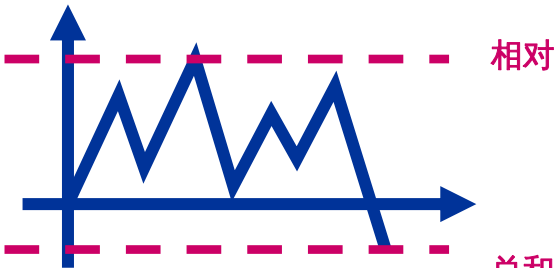


图 7

综合以上两点，我们连续的两个数的子串 *a*, *b* 陈述一个可以被压缩的子串 *u* 的所有令人感兴趣的特征，其中 *a* 为非负数，表示 *u* 的相对峰值，*b* 是一个非正的修正值，使(*a* + *b*)等于子列 *u* 的总和。从图象上看，即将上图简化为下图的形式。这就是本题中压缩的替代法则。

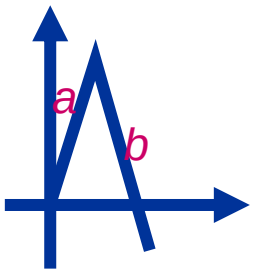


图 8

注意形如上文中含两个数字的子串 *a*, *b*，其中 *b*≤0 显然是可压缩的：即他们在最优串 *S* 中一定连续出现，不妨称之为一“对”（**couple**）。

这样的一“对”不仅可以在替代法则中代替一段待压缩的子串，在以后的定理证明中也可以代替一些未知的数列的基本特征。我们称一“对”*a*, *b* 为正，当它

对以后部分和的影响($a+b$)为正数，否则若影响为负数则称之为负；特别的，定义 a, b 为“零对”当且仅当 a+b=0。

下面我们就来分析在这题中存在着哪些可压缩性：在什么情况下，某一个数列中的一段子串可以被压缩。

[寻找可压缩性：第一阶段压缩]

首先不难发现下面的定理：

[定理 5.1]

如果在某个输入数列中，存在连续的子串

$$u_1，u_2，u_3，…，u_p$$

设其部分和为

$$Q_1，Q_2，Q_3，…，Q_p$$

且 $Q_1，Q_2，Q_3，…，Q_{p-1}$ 均是非负数，只有 $Q_p < 0$。则当可以选择 u_1 加入 S 数列时，要么暂时不选，要么将连续的 u_1 直至 u_p 一起加入 S。**即 *c* 子串可压缩**。

要证明这个定理，先来看一个引理：

[引理 5.1.1]

如果在某个输入数列中，存在连续的两“对”：

$$a_1, b_1, a_2, b_2$$

其中 a_1, b_1 非负，a_2, b_2 为负，如下图所示。则可以将它们压缩成新的一“对”。

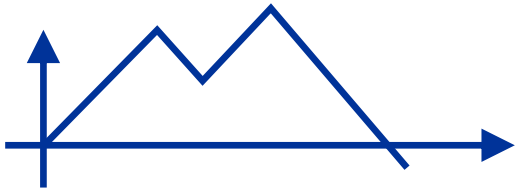


图 9

[证明 5.1.1]

使用调整法，假设在 S 中，有如下三“对”形成的子列：

$$a_1, b_1, x, y, a_2, b_2$$

其中 x, y 表示隔开 a_1, b_1 和 a_2, b_2 的一段数。

分两种情况讨论：

1. 若 $x+y\le 0$，则将子列调整为：

$$x, y, a_1, b_1, a_2, b_2$$

此时 a_1, b_1 和 a_2, b_2 连续。而由于“对”a_1, b_1 非负即 $a_1+b_1\ge 0$，调整后 x 的绝对高度不会升高；由于 $x+y\le 0$，调整后 a_1 的绝对高度也不会变得更高。

2. 若 $x+y>0$，则将子列调整为：

$$a_1, b_1, a_2, b_2, x, y$$

此时 a_1, b_1 和 a_2, b_2 连续。而由于 $a_2+b_2<0$，调整后 x 的绝对高度降低；由于 $x+y>0$，调整后 a_2 的绝对高度同样也会降低。

综上两种情况，无论如何都可以将 S 中的 a_1, b_1 和 a_2, b_2 调整到连续的位置而 S 的峰值却不会增高。因此[引理 2.1.1]成立。

利用这个引理，就可以证明定理的结论了。

[证明 5.1]

首先认为这 p 个数字是 p 个特殊的“对”，即一个数字被“压缩”后形成的“对”。那么这 p 个数字构成的数列就变成了 p 个“对”形成的数列。

根据假设，只要这 p 个“对”还没有被最终压缩成一个，那么第一个“对”总是正的，而由于 p 个数字的总和是负数，因此总能找到一个负的“对”。鉴于此，只要数列还没有被压缩成的一个“对”，我们总可以找到一个正的“对”后接一个负“对”，并根据[引理 5.1.1]将这两个”对”压缩成为一个新的“对”；重复着一个过程，直到只剩一个“对”时，就构造证明了这个定理。

现在我们可以这样依次处理每一个输入数列：从第一个数开始，累计部分和，依次向后考察每一个数，若加入部分和后部分和依然为正，则继续处理，否则若部分和为负，可将目前处理的这一段数根据[定理 5.1]“压缩”成一个“对”。将部分和清零后继续处理。

至此，我们已经完成了第一阶段的压缩。在这一阶段完成后，每一个输入的数列的前半部分都变成了许多负的“对”a_i, b_i，即每一个正数后都跟有一个绝对值更大的负数。而后半部分则是这样的一列数：它的任意部分和都是正数。

可以证明，在最优的合并数列 S 中，所有输入数列的“后半部分”将恰好是 S 的后半部分，即最优合并方案一定是将所有输入数列中的前半部分按某个顺序都归入 S 后再考虑那些任意部分和都是正数的子列[15]。

因此这个问题可以分成两个互不影响的部分：先归并所有数列的前半部分，再考虑那些部分和为正的子列。进一步的分析可以发现，问题的后一部分其实和前一部分等价，S 数列的总和确定，若将所有部分和为正的子列逆向观察，则一定也可以被压缩成多个负的“对”从而进行合并。所以我们只要专心得考虑问题的第一个部分就可以了。

[15] 证明较为简单，故略去。

[寻找可压缩性：第二阶段压缩]

[定理 5.2]

如果在某个输入数列中，存在连续的负的“对”

$$a_1, b_1, a_2, b_2$$

且峰值在 a_1，即 $a_1 \geq a_1 + b_1 + a_2$，那么这两个“对”在 S 中必定连续出现，即**可压缩**。

这个定理的证明也很简单：

[证明 5.2]

假设某一个最优方案的一个子列为

$$a_1, b_1, x, y, a_2, b_2$$

则这一段的峰值为 $\max\{a_1, a_1 + b_1 + x, a_1 + b_1 + x + y + a_2\}$，而做调整

$$a_1, b_1, a_2, b_2, x, y$$

后的峰值为

$$\max\{a_1, a_1 + b_1 + a_2, a_1 + b_1 + a_2 + b_2 + x\}$$

由于 $a_2 + b_2 \leq 0$，得到

$$\begin{cases} a_1 \leq a_1 \\ a_1 + b_1 + a_2 \leq a_1 \\ a_1 + b_1 + a_2 + b_2 + x \leq a_1 + b_1 + x \end{cases}$$

➔ $\max\{a_1, a_1 + b_1 + a_2, a_1 + b_1 + a_2 + b_2 + x\} \leq \max\{a_1, a_1 + b_1 + x, a_1 + b_1 + x + y + a_2\}$

即调整后 a_1, b_1 和 a_2, b_2 连续出现，且峰值不会变得更大：同样是一个最优方案。命题得证。

[推论 5.2.1]

如果在某个输入数列中，存在连续的多个负的“对”

$$a_1, b_1, a_2, b_2, a_3, b_3\ldots$$

且峰值在 a_1，即 $a_1 \geq a_1+b_1+a_2$, $a_1+b_1+a_2+b_2+a_3$…，那么这个子串可以压缩。

以[推论 5.2.1]为依据，我们对所有具有第二条可压缩性的子串实施压缩，将得到新的 K 个数列。每个新数列中的数字符号一定是正负交错，而绝对值是递增的，因此每一个“对”的相对峰值都是递增的。如下图所示。

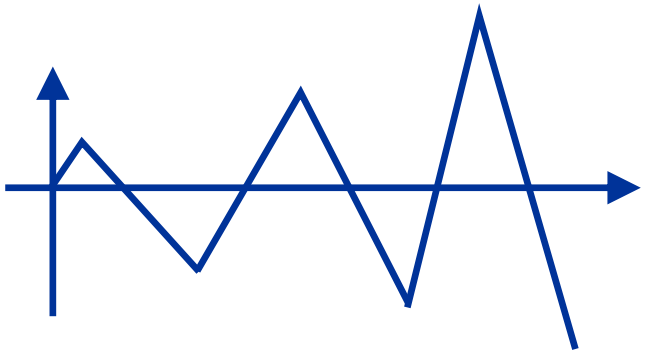


图 10

[贪心法解题]

经过上述第二阶段的压缩处理，现在这个问题就以一个十分简洁、优美的形式呈现在我们面前了：有 K 个数列，且每个数列都是由若干个总和为负的“对”组成，且它们的相对峰值单调递增，要求将它们归并成为新数列 S，且数列 S 的最大部分和尽量小。

至此，我们可以使用一个简单的贪心算法了。由于每一个“对”的总和为负，因此将相对峰值较高的“对”尽量向后放，可以尽量降低其绝对峰值，这是贪心的原则。而由于每个数列中“对”的相对峰值严格递增，因此对这些数列执行一次按相对峰值高度大小为序的归并排序，就可以得到一个新数列 S，其中“对”的相对峰值也是递增的：这正好完全符合我们的贪心原则。从感性上看

这个算法是正确的，而简单的推导也可以证明我们的观点，我们把这个问题就留给各位同学了。

算法实现上，在每次归并时，都选择供选“对”中相对峰值最小的一个加入 S，而通过一个堆，则可以做到在 $O(\log_2 K)$的时间内取最小。

因此压缩法不仅为本题找到了多项式级别的算法，而且还是一个时间复杂度仅为 $O(N\log_2 K)$的优秀算法。[16]

[小结]

在上例中，在第一阶段的压缩之后，我们手中的 K 个数列由杂乱无章变得稍有规律，其每一个数列中的正数后总会跟有一个绝对值更大的负数；但这还不足以实施贪心法，因此又有了第二阶段的压缩，进一步提炼真正有用的信息，得到的新数列中的非结尾的负数后又跟着一个绝对值更大的正数。这样数列中一个个“对”的相对峰值呈现美妙的单调性，这正是贪心法需要的条件。

可见，在很多问题中由于输入数据过于纷乱，我们可以采取多阶段压缩、甚至是迭代式压缩的方法，分层次的丢弃冗余，提炼精华，就像一个金字塔，一层一层的逼近，最终到达解决问题的顶峰。

总结

让我们先来回顾一下文中的主要例题。在[例一]单源最短路问题中，我们使用压缩法将源点集合压缩成为一个新的源，从而由多源问题得到了单源的

[16] 至于如何实现，大家可以参考论文附件中关于本题的程序。

问题进而得以解决；在[例二]球队问题中，压缩法将每个强连同分量缩成一个个点，从而将复杂的、任意的球员通讯联系图，简化成为一个有向无环图；在[例五]合并数列问题中，根据两条不同的可压缩性，符合条件的子串都可以被压缩成 a, b 两个数形成的一个“单峰”，由此 K 个任意的输入数列在压缩转化后显示出了美妙的单调性从而有了后来的贪心法解题。

在这些例题中，压缩法都是将一个个复杂的事物用较为简单的来代替，从而化简了问题。这正是数学思维中的一条经典思想：**化归思想**。

说到化归思想，大家一定对我们在做数学题时使用的换元法不陌生。例如解下面一个方程

$$x+\sqrt{x-1}=7$$

的时候，可以设 $t=\sqrt{x-1}$ 代换得到关于 t 的方程

$$t^2+1+t=7$$

从而得到原方程的解。这里的假设 $t=\sqrt{x-1}$ 就是做了一步化归：化无理为有理，化抽象为形象，进而化复杂为简单为最终解决问题铺平道路。可以看出，我们上文中讨论的压缩法也有类似的神奇特点。

然而，换元法做的仅仅是形式上的变化，是一个完全等价的变化：针对上面的例子，其换元的前提就是 t 和 $\sqrt{x-1}$ 相等。

而压缩法则不然，不妨回顾一下压缩法的两个要点：可压缩性和替代法则。可压缩性实际上承认了一个集合中冗余信息的存在性，即集合中元素的相互关系是无意义的。而替代法则则为我们提供了一个实际操作方案，通过一个现实的方法，保留下信息的精华部分。

可以看出，压缩法的两个要点都在围绕着信息的利用大做文章，而这是换元法所不具备的特性。

我们生活在一个信息爆炸的社会中，快速准确的获取信息已不是什么难事，然而如何防止信息污染却是一个日益严重的全球化问题。信息学竞赛应运而生，其宗旨在于教会我们如何合理有效的利用信息。而信息学竞赛中的压缩法继承了经典数学中的化归思想，又创新的加入了“信息化”因素，将两种思想有机的结合，通过舍去冗余的信息达到化繁为简、化难为易的目的，在很多复杂化的问题中有着广阔的应用前景。

因此，当我们遇到问题，感觉题设条件过多，信息量过大，无从下手时，不妨试一试压缩法，因为我们压缩法的目标正是：

压去冗余 缩得精华

附录

附录一：关于[例四]中的斜率优化法

考虑[例四]中的动态规划方程：

$$f(n) = \max_{0\le i<n}\{f(i) + (S_n - S_i) * n - T\}$$

当 n 确定时，不妨设

$$g(i) = f(i) + (S_n - S_i) * n - T$$

表示每一个决策，则

$$f(n) = \max_{0\le i<n}\{g(i)\}$$

对于两个不同的决策 $0 \le i, j < n$，设 $S_i > S_j$，那么 $g(i) > g(j)$的充分必要条件则为：

$$f(i) + (S_n - S_i) * n - T > f(j) + (S_n - S_j) * n - T$$

$$f(i) - f(j) > n(S_i - S_j)$$

$$\frac{f(i) - f(j)}{S_i - S_j} > n$$

可以看出，如果将每个决策看作一个对应的点

$$P_i[S_i,\ f(i)]$$

如正文中图 11 所示。

则两个决策 P_i 和 P_j 相比，若 P_i 对应的点在 P_j 的右边，那么 P_i 优于 P_j 当且仅当直线 P_iP_j 的斜率大于 n。参考作者去年论文中的[例二]最大平均值问题的优化方法，可以证明在待选的决策点中，下凸点（凹点）的存在是没有意义的：无论 n 的取值如何都不会使其变成最优的决策点。因此我们只需要维护一个上凸函数即可。

如果可以确定每次添加的点都在原先所有点的一侧（这正是正文中分析所要做的），则可以参见作者去年论文，对这个上凸折线进行平摊复杂度为 O(1)的插入维护和取最优操作。

附录二：论文附件

Baltic Olympiad in Informatics 2003 Euro（欧元兑换）英文原题

Eur o. doc

Baltic Olympiad in Informatics 2003 Euro（欧元兑换）程序，时间复杂度 O(N)：

euro.pas

[例五]合并数列问题，程序，时间复杂度 O($N\log_2 K$)：

MergeSequence.pas

附录三：关于 MergeSequence.pas 程序的输入格式

输入文件 MergeSequence.in 的第一行包含一个整数 K，表示待归并数列的个数，以下 K 行，每行描述一个数列。每行的第一个数 n_i 表示该数列中的数字数目，之后 n_i 个数，顺序描述这个数列。

参考文献

《信息学奥林匹克竞赛指导——图论的算法与程序设计(PASCAL 版)》 吴文虎 王建德 编著

《实用算法的分析与程序设计》 吴文虎 王建德 编著

《浅谈数形结合思想在信息学竞赛中的应用》 作者论文于 2004 年信息学奥林匹克竞赛冬令营

Zhejiang University Online Judge：http://acm.zju.edu.cn

用改进算法的思想解决规模维数增大的问题

广东韶关一中 张伟达

【关键字】 增大规模 改进算法 降维 分析 构造

【摘要】

我们常常会遇到一些特殊的问题，它们把我们能够解决的问题改了一改，增加了一维，或者增加了一个因素，从 1 到 2 或者是从 2 到 3，本文把它们统称规模维数增大的问题。

解决这类问题可以用改进算法的思想：本文第一部分先是概述这种思想；第二部分则从一道有趣的 IQ 题目说起，引入改进算法的基本思路和基本途径；第三部分则用多个例子解释说明如何改进算法；最后一部分是总结改进算法的思想。

【目录】

一、概述

每个学习信息学奥林匹克的选手总是要先学习一些基本的算法，然后才能把算法应用到题目当中去。但是，题目形式多种多样，这些算法往往是不能够直接套用的。有的时候我们分析到某个问题好像可以用某种方法解决，但是这个问题却与之前见过的问题有些不同，例如本文所讨论的情况：问题的规模维数比原问题大，一维的变二维的，二维的变三维的，等等。我们不妨把这类问题叫做规模维数增大的问题，解决这一类问题，往往需要观察问题的特殊性，改进原有算法，从而解决实际问题。

二、引子：从一道 IQ 题说起

【IQ 题题目】在走入正题之前，还是让我们来玩一道 IQ 题(据说是 IBM 公司招聘面试用过的题目)：有两根完全相同但分布不均匀的香（提示：不妨假设是蚊香，一圈圈的那种），每根香烧完的时间是一个小时，你能用什么方法来确定一段 45 分钟的时间？

【……】（思考中）

【思路】在公布答案之前，还是让我们来循序渐进地思考下去吧。

要确定 45 分钟的时间，而每根香烧完的时间是一个小时，现在应该有三个模型让我们思考：

A.不减小每根香的计时，两根香加在一起计时是 45 分钟；

B.只用一根香，使其计时减小，直接计时 45 分钟；

C.把两根香的计时都减小，使两根香加起来是 45 分钟。

A 模型显然是不可能成立的；B、C 模型的共同点是：我们必须使一根香的计时减小。但是怎样减，能减成什么呢？显然是不能直接把它们分开的，因为香的分布不均匀。由于这里笔者已经降低了难度，根据提示，比较容易想到香是可以两头一起烧的。这样，我们就能把一根香的计时减成半个小时。方案已经更进一步了。通过这一步，B 模型只用一根香，是很容易被排除的。

余下的选择只有“C 模型+两头烧方法”了，但是 C 模型不能直接应用“两头烧”，因为这样套用的话，0.5+0.5 仍然等于 1。我们需要对“两头烧”做进一步改进：如果一根香只烧了一头，当剩余时间为 t 的时候，我们把另一头也点燃——那么，我们就能够计出 t/2 的时间了（前提是我们已知当前剩余时间 t）。特别地，当 t=60min 时，t/2=30min。

根据这一改进后的方法，我们很容易就得出正确的解法：分别点燃第一根的两头和第二根的一头，第一根烧完的时候，已经过了 30 分钟；第二根还剩 30 分钟，点燃第二根的另一头；当第二根也烧完了，即时间又过了 15 分钟。那么我们计出的总的时间就为 45 分钟了。

【小结】上面这个例子中，我们从构造模型和改进算法两方面入手，一步一步地达到了优化的解法。主要流程是：

1. 找出原始解法和可能改进的方向（即分析成 A、B、C 模型）；
2. 分析算法的原理（由烧一根香计时半小时，引申为烧剩 t 的时候点两头就能计时 $\frac{t}{2}$）；
3. 改进算法（改进的过程中，往往不是依靠算法改进算法本身，反而是利用算法的内涵、实质，结合问题，构造算法）；
4. 解决问题（我们得出了正确的解法）。

但是如何来改进原有的算法呢，笔者认为可以有以下途径：

1. 直接增加算法的规模，解决问题
2. 用枚举处理增加的规模，从而解决问题
3. 用贪心解决增加的规模，从而解决问题
4. 以上各个途径的综合运用

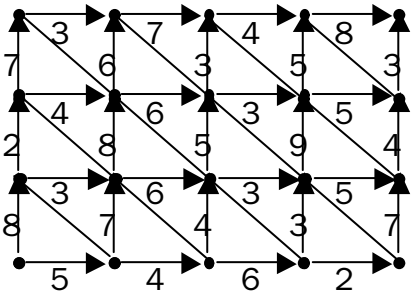
围绕这一主线，笔者将用例题做进一步的阐明。

三、改进算法的途径

(1)直接增加算法的规模，解决问题

解决简单的问题，常常可以采用直接处理的方法，但是很多时候还是需要一些技巧。

【例一】街道问题及其扩展。（经典问题）



【题目大意】〖原题〗在右图中找出从左下角到右上角的最短路径，每步只能向右方或上方走。〖扩展〗在地图中找出从左下角到右上角的两条路径，两条路径中的任何一条边都不能重叠，并且要求两条路径的总长度最短。

【问题分析】原题是一道简单而又典型的动态规划题，很显然，可以用这样的动态规划方程解题：$f_{i,j}=\min\begin{cases}f_{i-1,j}+\text{Distance}_{(i-1,j),(i,j)}\\ f_{i,j+1}+\text{Distance}_{(i,j+1),(i,j)}\end{cases}$（这里 Distance 表示相邻两点间的边长）

但是对于扩展后的题目：再用这种"简单"的方法就不太好办了。如果非得套用这种方法的话，则最优指标函数就需要有四维的下标，并且难以处理两条路径“不能重叠”的问题。

我们来分析一下原方法的实质：按照图中的斜线来划分阶段，即阶段变量 k 表示走过的步数，而状态变量 x_k 表示当前处于这一阶段上的哪一点。这时的模型实际上已经转化成了一个特殊的多段图。用决策变量 u_k=0 表示向右走，u_k=1 表示向上走，则状态转移方程为：$x_{k+1}=\begin{cases}x_k+1-u_k, k\le row\\ x_k-u_k, k>row\end{cases}$ (这里的 *row* 是地图竖直方向的行数)。通过这一步观察，我们就会发现这个问题很好解决，只需要加一维状态变量就成了。即用 $X_k=(a_k,b_k)$ 分别表示两条路径走到阶段 k 时所处的位置，相应的，决策变量也增加一维，用 $u_k=(x_k,y_k)$ 分别表示两条路径的行走方向。状态转移时将两条路径分别考虑：

$$\begin{cases}a_{k+1}=a_k+1-x_k, b_{k+1}=b_k+1-y_k, k\le row\\ a_{k+1}=a_k+x_k, b_k+1=b_k+y_k, k>row\end{cases}$$

【小结】从这个例子可以看出，改进的时候不能只依靠原始算法，还要分析原始算法的本质。

(2)用枚举处理增加的规模，从而解决问题

【例二】旅行（广东省奥林匹克竞赛 2001）

【题目大意】给出一个 N×M 的数字矩阵，要求这个矩阵的一个子矩阵，并要求这个子矩阵的数字和最大。

【问题分析】初一看题目，想到枚举每一个子矩阵，求出子矩阵的和，比较得出最大值。这样，时间复杂度达到 $O(N^4)$，显然不可以接受。因为它是两维的问题，我们可以尝试着把维数降低。

先看一维时候的情况：在数列 a_i 中找出和最大的一段。

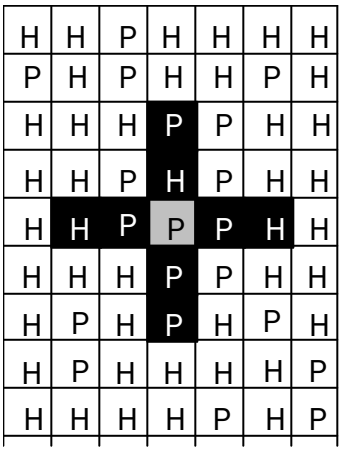
对于一维的子问题，可以这样来想：如果用最基本的方法，可以枚举每个子序列。但是如果纯粹三重循环，枚举头，枚举尾，枚举中间求和，显然是太浪费了。其实我们计算 $S_{i,j}$ ($S_{i,j}=a_j+a_{j+1}+\cdots+a_i, j<i$)时，可以由 $S_{i,j}=S_{i-1,j}+a_i$ 获得。这样，动态规划的思路就明显了，$S_{i,j}$ 与 $S_{i-1,j}$ 有直接的关系，而且我们还可以发现 $S_{i,*}$（表示从 $S_{i,1}$ 到 $S_{i,i-1}$ 构成的数列）只比 $S_{i-1,*}$ 多了一项。于是简单的动态规划就出来了，用 f_i 表示以 i 为终点的数列的最大和，有：

$$f_i=\begin{cases}\max(f_{i-1}+a_i,a_i), i>1\\ a_1, i=1\end{cases}$$

不过，我们得到的只是子问题的算法，如何改进算法能使它适用于矩阵的问题呢？我们需要找出矩阵和数列的关系：矩阵是若干条数列，但是它又不仅仅是简单的若干条数列，不同数列中的相同位置的数也构成了联系。我们可以尝试着构造上下、左右同时进行的动态规划，但是也许我们无能为力。对于这道题的数据规模，似乎没有必要构造 $O(N^2)$时间的算法。不难想到，枚举上下方向的数列和，在左右方向用动态规划求解，可以得到时间复杂度在 $O(N^3)$的算法。

【小结】在这个例题中，我们用了最简单的方法——枚举，来改进一维数列的最大子序列的解法，从而解决问题的规模的增加。但是枚举这个简单的方法，却能解决很复杂的问题。

【例三】炮兵阵地。（NOI2001）



【题目大意】在图中标为 P 的格子中放炮兵，如图灰色区域放了炮兵后，黑色区域不能放炮兵，问图中最多能放多少炮兵？

【问题分析】简化的问题：若只有一列，问最多可以放多少炮兵。

可以用贪心，尽量把炮兵往上放，也可以这样递推：用 f_i 表示前 i 行最多能放炮兵数，a_i 表示第 i 行的地形，则边界条件：

$f_1 = \begin{cases} 1, a_1 = 'P' \\ 0, a_1 = 'H' \end{cases}$，动态规划方程：$f_i = \max \begin{cases} f_{i-3} + 1, i > 3 且 a_i = 'P' \\ 1, 1 < i \le 3 且 a_i = 'P' \\ f_{i-1}, i > 1 \end{cases}$

但是当问题扩展到多列的情况，就复杂了：不仅要考虑一列是否能引用上一行的函数值，而且要多列同时考虑。但是容易观察到，m<<n，最大只有 10，又因为炮兵攻击范围很大，实际上能放炮兵的组合数很小，可以考虑枚举组合，从而得到改进的动态规划方法：以两行为一个状态，s_1 表示状态中上行的炮兵布置情况，s_2 表示下行的布置情况，（可以预先枚举所有可能的情况并给这些情况标号）

当第 i 行能不能布置 s_1 或第 i+1 行不能布置 s_2 时，或者 s_1,s_2 在同一位置布置了炮兵时，都是不符合情况的，令此时 $f_i(s_1, s_2) = 0$；

否则：$f_i(s_1, s_2) = \max(f_{i-1}(s, s_1)) + \text{count}\, s_2$，其中 counts$_2$ 表示 s_2 的炮兵个数；s 表示该 s_1,s_2 的前一行的炮兵布置情况，而且满足：不存在这样的 $j(j \le m) \,\&\, (s_2[j] = s[j] = 1)$。

这样到最后的最优解就是 $\max(f_{n-1}(s_1, s_2))$

最终复杂度分析，时间复杂度大约 $O(R^3n)$（R 是一行状态的组合数）

(3)用贪心解决增加的规模，从而解决问题

【例四】求网络的最小费用最大流。（经典问题）

显然，求最小费用最大流可以由求最大流的算法改进。求网络的最大流，简单地说来，是每次寻找一条连接源点和汇点的可增广路径并由之扩充网络流，直到不存在这样可增广路径，则得出最大流。

那么，又如何把费用也考虑进去呢？我们先来看看网络的费用的定义吧。网络的费用一般是指在每一段弧上弧的费用与弧的流量的乘积的和。（弧的费用由可能为负）所以网络的最小费用最大流是指可行流中费用最小的。不难看出，可行流每增加 1，所增加的费用都应该是最小的（事实上应该是减小得最多的）。这样可以得出一个改进：每次选取一条费用最小（而且非正）的可增广路径，直到最终不存在费用非正的可增广路径。这样用贪心的策略就能解决问题了。

(4)多种途径的综合运用

【例五】Team Selection (Balkan OI 2004 Day1)

【题目大意】IOI 要来了，BP 队要选择最好的选手去参加。幸运地，教练可以从 N 个非常棒的选手中选择队员，这些选手被标上 1 到 N($3 \le N \le 500000$)。为了选出的选手是最好的，教练组织了三次竞赛并给出每次竞赛排名。每个选手都参加了每次竞赛并且每次竞赛都没有并列的。当 A 在所有竞赛

中名次都比 B 前，我们就说 A 是比 B better。如果没有人比 A better，我们就说 A 是 excellent。求 excellent 选手的个数。

如数据：

```
10
2  5  3  8  10 7  1  6  9  4
1  2  3  4  5  6  7  8  9  10
3  8  7  10 5  4  1  2  6  9
```

则 excellent 选手是 1，2，3，5。

【原始思路】一拿到这一题，很容易会有以下的思路。

〖原始算法〗第一眼看到这道题往往想到枚举。简单地根据 better 和 excellent 的定义，现只需要枚举每一个选手 X，判断 X 是否 excellent。这可以通过另一重循环，枚举另一选手 Y，判断 Y 是否比 X better。判断是容易的，我们只需要简单地判断 X 和 Y 的三次排名。因此这一算法的时间复杂度是 $O(N^2)$，对于这道题，不能满足要求。

主要流程 1：

```
for(X 从 1 到 N)
        for(Y 从 1 到 N)
                判断 Y 是否比 X better
```

〖改进一〗原始算法是可以改进的。不难看出，如果让 X 依照第一次竞赛的名次循环，枚举 Y 时只需要枚举在第一次竞赛中排在 X 前面选手即可，因为第一次竞赛排在 X 后的选手一定不可能比 X better。不过这样小的改进提起来只是开阔开阔思路，它对时间复杂度不会有太大影响。

主要流程 2：

```
for(X 从第一次竞赛的第 1 名到第一次竞赛的第 N 名)
```

for(Y 从第一次竞赛的第 1 名到第一次竞赛的第(X-1)名)

判断 Y 是否比 X better

〖改进二〗如果我们进一步观察，我们能优化上述算法的时间复杂度。我们发现：如果 Y 不是 excellent（因为有 Z 比 Y better），当我们检查 X 是否 excellent 时，我们检查了 Z 是否比 X better 的话，可以不检查 Y。因为如果 Y 比 X better，那么 Z 一定比 X better。然后可以通过简单地修改“改进一”。在判断 X 是否 excellent 的时候，我们只需要在当前的 excellent 选手集合中取得 Y 即可。

主要流程 3：

for(X 从第一次竞赛的第 1 名到第一次竞赛的第 N 名)

for(Y 枚举当前已知的 excellent)

判断 Y 是否比 X better

这样，我们能把时间复杂度减少到 O(NK)（设 K 是 excellent 选手的个数）。由于这题 K 可能很大，算法效果仍然不是很理想。

【原始思路小结】这里的原始算法是直接根据原始模型模拟出来的，改进一和改进二都单纯地根据原始算法的设计缺陷来“改进”（这个改进没有利用问题的特殊性，不是本文所要阐述的“改进”），所以最后的时间复杂度没有质的进展。

【降维思路】本道题规模还是很明显的从二维扩充到三维。所以还是先用降维的思想，我们先解决选手之间只进行两次竞赛的问题。

两次竞赛的问题同样可以套用改进二的普遍算法。为了方便说明，我们设第一次竞赛排名依次为 A_i（表示第一次竞赛的第 i 名是 A_i），A_i 号选手在第二

次竞赛中的排名的为 B[A_i]（注意 B[A_i]与 A_i 的含义不同）。参考“主要流程 3”：先是 X=A_1，显然 A_1 是 excellent；接着 X=A_2，只需要比较 B[A_1]与 B[A_2]大小，不妨假设 B[A_2]<B[A_1]，则 A_2 也是 excellent；接下来，X=A_3，我们需要比较 B[A_1]与 B[A_3]，B[A_2]与 B[A_3]……假设 A_3 也是 excellent；X=A_4 时，我们需要比较 B[A_1]与 B[A_4]，B[A_2]与 B[A_4]，B[A_3]与 B[A_4]，假设 A_4 也是 excellent；……。也许这里有心人会发现什么，对了，比较 B[A_1]与 B[X]，B[A_2]与 B[X]，B[A_3]与 B[A_i]……的时候，只要有任何一个 B[A_j]<B[A_i]($j<i$)，A_i 就不是 excellent 的，那么，我们只需要用 min(B[A_j])与 B[A_i]比较！

〖改进三〗（适用于两次竞赛）以上我们看到了一个特殊的例子：两次排名完全相反，但是我们可以从中提炼出可行的改进算法：对于两次竞赛的情况，当 X=A_i 时，设 $best_i = \min(B[A_j])(j < i)$，则 B[$A_i$]只需要与 $best_i$ 比较即可。时间复杂度竟然达到 O(N)。

【降维思路小结】在这次分析中，我们从两次竞赛——简化后的问题出发，通过简单的观察和思考，得出了改进的算法，但是优化后的算法要应用到三维的情况还有很长的路要走。

【扩展思路】我们怎样把“改进三”应用到三维的情况从而解决问题呢？

改进三的思路能否再明确些？改进三的本质是什么？其实，X 依照第一次竞赛的名次循环，X=A_i 时，因为 $best_i = \min(B[A_j])(j < i)$ 中，$j<i$ 这样就保证了 A_j 在第一次竞赛中名次一定比 A_i 前；如果 $best_i$<B[A_i]，这样就保证了 A_j 在第二次竞赛中名次比 X 前；总之则必然有 A_j 比 A_i better。

〖改进四〗能否把改进三扩展到三次竞赛上面去呢？为了方便说明，我们还需要设第三次竞赛中 A_i 选手名次为 C[A_i]。我们假设这样一个过程：

(1)当 X=A_i，设 $j<i$，则可以保证 A_j 在第一次竞赛中名次比 A_i 前，若不存在这样的 j，则 A_i 是 excellent；

(2)在符合(1)条件的所有 A_j 中找出能满足 $B[A_j]<B[A_i]$的选手，保证 A_j 在第二次竞赛中名次比 A_i 前，若不存在这样的 j，则 A_i 是 excellent；

(3)在符合(1)(2)条件的所有 A_j 中找出 $\min(C[A_j])$，比较 $\min(C[A_j])$与 $C[A_i]$，若 $\min(C[A_j])<C[A_i]$，则保证 A_j 在第三次竞赛中名次比 A_i 前，则必有 A_j 比 A_i better，A_i 必不是 excellent；反之 A_i 是 excellent。

以上三个条件即为：“A_i 是 excellent”等价于“不存在这样的 $j \in \{j \mid j < i, B[A_j] < B[A_i]\}$，使 $\min(C[A_j]) < C[A_i]$”。

〖改进五〗显然，如果我们仍然用枚举的方法来实现第(2)步，则最终的算法复杂度仍然是 $O(N^2)$。为什么我们却说这是改进过后的算法呢？因为我们实际上利用问题的特殊性，改进二是纯粹的枚举，进行的第(1)步以后要第二第三次竞赛同时比较，实现起来是很难优化的。但是改进四则体现了第二第三次竞赛分步选取的思路，利用这一点能否在实现中改进算法的复杂度呢？

我们可以比较，改进二和改进四是不同的，在进行了第(1)步以后，改进二需要同时考虑第二第三次竞赛成绩，两次成绩同时考虑并不是严格有序的，有时候我们不能比较二元组$(B[A_i],C[A_i])$与$(B[A_j],C[A_j])$的大小（例如当 $B[A_i]<B[A_j]$，但 $C[A_i]>C[A_j]$时）。

而改进四的第(2)步可以先考虑第二次竞赛的成绩，如果我们把第二次竞赛的成绩用有序数列来记录（设这个序列为 D_k，即把 $B[A_j](j<i)$从小到大排序），我们能轻易找到符合条件的 A_j。

〖改进六〗由改进五可以知道，我们单单从改进第(2)步，时间复杂度仍然不理想的，能否切实把第(2)步和第(3)步结合起来？经过了改进五的改进，我们把查找到的符合 $B[A_j]<B[A_i](j<i)$ 的 A_j 在一个有序数列中记录下来，这样，符合条件的 A_j 必然在数列中是连续的，设它是从 D_1 到 D_K。为了让第(3)步与第(2)步结合，我们再设另一个数列 E_k，使它和 D_k 一一对应，即 D_k 与 E_k 分别代表同一选手在第二第三次竞赛的名次。这样，第(3)步又可以简化为在一段数列 E_k 中查找最小值，即找 $\min(E_k)(k \le K)$

能否在最短时间内查找出这个最小值呢？我们需要一种优化的数据结构来记录 D_k 和 E_k。该数据结构能很快处理 D_k 和 E_k，并且需要支持两种操作。

插入：加入一对数(key,value)到数据结构中。

查询：查询 key 小于 X 的最小的 value

显然，这里的 key 代表 Dk，value 代表 Ek。

因为我们需要的操作与检索树擅长的操作很类似，所以对数据结构有一定理解的选手，在这里都能够想到我们可以构造一个二叉检索树。

【改进六的实现：优化数据结构】我们构造检索树的目的是查询某一区间的最小 value，这样，我们可以定义每个节点代表一个区间，叶子部分代表一个 key，我们依次把每两个叶子指向同一个父亲节点。例如 1 和 2 的父亲是区间[1,2]；[1,2]和[3,4]的父亲是[1,4]，如此类推。我们定义每个节点表示的区间是 key 的区间，而节点值则记录该区间中的最小 value。

插入和查询具体做法如下：

插入操作：当我们接到一个请求：把(a,b)加入到检索树中。那么我们只需要依次从 key=a 的叶子检索到根节点，比较 b 和途中每个节点的值大小，如果 b 小于该节点的值，就用 b 更新它。

查询操作：我们要让检索树能够查询 key 小于 a 的最小 value。显然我们查询[1,a]的最小 value 时，由检索树的性质，我们可以把它分成小区间并分别查询它。例如当 a=14 时，需要访问到区间[1,8]，[9,12]，[13,13]。对于这题，可以有简单的实现方法：先令 min-value 为 0，依次从表示[a,a]的节点检索到根节点，如果该节点有父亲并且是父亲的右儿子，就比较它的左兄弟和 min-value 的值，如果左兄弟节点的值小于 min-value，就用左兄弟节点的值更新它。到根节点的时候即可以返回 min-value。

这种数据结构能使查询和插入操作都能在 O(LogN)的复杂度下完成，

〖最终算法〗这样我们就能很清晰地得出优化的解法：以第一次竞赛的名次从高到低枚举 X，以第二次竞赛名次为 key，第三次竞赛名次为 value。对于每个 X，只要查询区间[1,key]的最小值 min-value，若 min-value<value，则有选手比 X better，即 X 不是 excellent。反之，若 min-value>value，说明 X 是 excellent，并把(key,value)加入检索树。

由于枚举 X 循环需要 O(N)时间，查询和插入都只需要 O(LogN)，总的时间复杂度是 O(NLogN)。对于所有数据都应该在很短时间内出解。

【小结】在这个部分，我们分析了 Team 一题，过程中总结了很多算法，有的是基本算法，有的是改进不完全的算法。不过很多时候我们思考这一题并不需要这么多的改进，例如我们很容易就能跳过改进一而直接得到改进二的结论。

这一题改进三得出来以后，并没有而且不能直接得出改进四，而需要很重要的一步：分析改进三的本质，从而扩展改进三。

笔者之所以选取了 Team 这题作重点分析，不仅是因为 Team 一题需要灵敏的头脑，全面地观察，还因为它需要对数据结构有一定的认识，笔者当初做这道题的时候就在这里栽了跟头。

四、总结

这篇论文给大家解释了笔者关于规模维数增加的一类问题的看法。由于笔者水平有限，有些道理仍然未能参透，文中的算法还能有改进的地方。但本文旨在阐述在解决这类问题的过程中，应该结合算法的本质和问题的特点。

算法改进的思想其实是开阔进取的思想，即不满足于现状，要开发新的算法，新的思路。人类漫长的历史不就是开阔进取的精神在支配着，没有开阔进取，又何来我们今天的发达的生产力？在 OI 中很多问题我们不能够说随便解决了就 OK，我们还要有开阔进取的精神，不断改进算法和数据结构，才能真正意义上解决问题。

算法改进的思想其实也是创新的思想，即“要创造”的思想。很多时候，信息学奥林匹克的问题并不能直接套用经典算法，而需要加入很多创新的思维，和具体的实践。事实上，任何经典的算法都是由创新的思想总结出来的。只要我们能大胆创新，说不定某一天也会有我们创造的经典算法哩。

【感谢】

感谢林盛华老师指导和杨溢同学的热情帮助。

【参考文献】

[1]《金牌之路 高中计算机竞赛辅导》/江文哉主编.—西安：陕西师范大学出版社，2000.6

[2]《奥赛经典 信息学奥林匹克教程 提高篇》/吴耀斌，曹利国，向期中，朱全民编著.—长沙：湖南师范大学出版社，2001.6

[3]《算法艺术与信息学竞赛》/刘汝佳，黄亮著.—北京：清华大学出版社，2003

[4] Balkan Olympiad Informatic 2004 官方网站提供的原题和解题报告

[5]广东省奥林匹克竞赛 2001（GDOI2001）题目

[6]全国奥林匹克竞赛 2001（NOI2001）题目

【附录】

A.旅行(广东省奥林匹克竞赛 2001)

➢ 〖问题描述〗

GDOI 队员们到 Z 镇游玩。Z 镇是一个很特别的城镇，它有 m+1 条东西方向和 n+1 条南北方向的道路，划分成 $m \times n$ 个区域。Z 镇的名胜位于这些区域内。从上往下数第 *i* 行，从左往右数第 *j* 列的区域记为 *D*(*i*,*j*)。GDOI 队员们预先对这 $m \times n$ 个区域打分 *V*(*i*,*j*)(分数可正可负)。分数越高表示他们越想到那个地方，越低表示他们越不想去。为了方便集合，队员们只能在某一范围内活动。我们可以用(m_1,n_l)与(m_2,n_2) $(m_1 \le m_2, n_1 \le n_2)$ 表示这样的一个范围：它是这些区域的集合：$\{D(i, j) \mid m_1 \le i \le m_2, n_1 \le j \le n_2\}$。GDOI 队员们希望他们活动范围内的区域的分值总和最大。

当然，有可能队员们一个地方也不去（例如，所有区域的分值都是负数。当然，如果某范串内的分值总和为零的话，他们也会去玩）。也有可能他们游览整个 Z 镇。你的任务是编写一个程序，求出他们的活动范围(m_1,n_l),(m_2,n_2)。

➢ 〖输入格式〗

输入数据存放在当前目录下的文本文件"travel.dat"中。数据有 m+1 行。第一行有两个数 m,n(m,n 定义如上)。其中，$(1 \le m \le 250, 1 \le n \le 250)$。接下来的 m 行，每行 n 个整数，第 *i* 行第 *j* 个数表示分值 *V*(*I*,*j*) $(-128 \le V(i, j) \le 127)$。每两个数之间有一个空格。

➢ 〖输出格式〗

答案输出到当前目录下 "travel.out"中，只有一行，分两种情况:

1． 队员们在范围(m_1,n_1),(m_2,n_2)内活动，输出该范围内的分值。

2． 队员们不想去任何地方，只需输出"NO"。

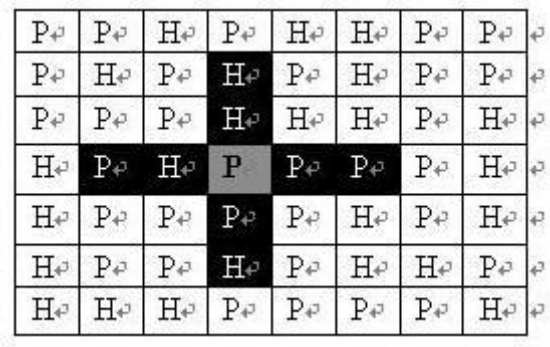
注意：不要有多余空行，行首行尾不要有多余空格。

➢ 〖输入输出举例〗

样例一		样例二	
Travel.dat	travel.out	Travel.dat	travel.out
4 5 1 -2 3 -4 5 6 7 8 9 10 -11 12 13 14 -15 16 17 18 19 20	146	2 3 -1 -2 -1 -4 -3 -6	NO

B.炮兵阵地（NOI2001）

司令部的将军们打算在 $N \times M$ 的网格地图上部署他们的炮兵部队。一个 $N \times M$ 的地图由 N 行 M 列组成，地图的每一格可能是山地（用"H" 表示），也可能是平原（用"P"表示），如下图。在每一格平原地形上最多可以布置一支炮兵部队（山地上不能够部署炮兵部队）；一支炮兵部队在地图上的攻击范围如图中黑色区域所示：



如果在地图中的灰色所标识的平原上部署一支炮兵部队，则图中的黑色的网格表示它能够攻击到的区域：沿横向左右各两格，沿纵向上下各两格。图上其它白色网格均攻击不到。从图上可见炮兵的攻击范围不受地形的影响。

现在，将军们规划如何部署炮兵部队，在防止误伤的前提下（保证任何两支炮兵部队之间不能互相攻击，即任何一支炮兵部队都不在其他支炮兵部队的攻击范围内），在整个地图区域内最多能够摆放多少我军的炮兵部队。

Input

第一行包含两个由空格分割开的正整数，分别表示 N 和 M；

接下来的 N 行，每一行含有连续的 M 个字符('P'或者'H')，中间没有空格。按顺序表示地图中每一行的数据。N <= 100；M <= 10。

Output

仅一行，包含一个整数 K，表示最多能摆放的炮兵部队的数量。

Sample Input
5 4
PHPP
PPHH
PPPP
PHPP
PHHP

Sample Output
6

C.Team Selection (BalkanOI 2004 Day 1 Task 2)

The Interpeninsular Olympiad in Informatics is coming and the leaders of the Balkan Peninsula Team have to choose the best contestants on the Balkans. Fortunately, the leaders could choose the members of the team among N very good contestants, numbered from 1 to N (3 ≤ N ≤ 500000). In order to select the best contestants the leaders organized three competitions.

Each of the N contestants took part in all three competitions and there were no two contestants with equal results on any of the competitions. We say that contestant A is better than another contestant B when A is ranked before B in all of the competitions. A contestant A is said to be excellent if no other contestant is better than A. The leaders of the Balkan Peninsula Team would like to know the number of excellent contestants.

Write a program named TEAM, which for given N and the three competitions results, computes the number of excellent contestants.

The input data are given on the standard input as four lines. The first line contains the number N. The next three lines show the rankings for the three competitions. Each of these lines contains the identification numbers of the contestants, separated by single spaces, in the order of their ranking from first to last place.

The standard output should contain one line with a single number written on it: the number of the excellent.

EXAMPLE 1

Input

3
2 3 1
3 1 2
1 2 3

Output

3

Note: No contestant is better than any other contestant, hence all three are excellent.

EXAMPLE 2

Input

10
2 5 3 8 10 7 1 6 9 4
1 2 3 4 5 6 7 8 9 10
3 8 7 10 5 4 1 2 6 9

Output

4

Note: The excellent contestants are those numbered with 1, 2, 3 and 5.

美，无处不在

——浅谈“黄金分割”和信息学的联系

安徽芜湖一中 杨思雨

摘要

本文主要介绍了“黄金分割”与信息学竞赛的联系及其在一些信息学问题中的应用，全文可以分为四个部分。

第一部分引言，主要介绍了“黄金分割”的由来及其和各领域的联系，进而提出探索黄金分割与信息学竞赛的联系。

第二部分中逐渐深入的分析了一个例题，最终揭示了问题本质上与黄金分割数的联系，说明了黄金分割数在一些数学性较强的题目中有着广泛的应用。

第三部分通过分析另一个例题，介绍了一种优选法——“黄金分割法”。

第四部分指出黄金分割与信息学竞赛联系密切，总结全文。

关键字

黄金分割 信息学 联系

目录

引言

早在古希腊时期，人们就发现了“黄金分割”。两千多年前，数学家欧多克斯提出了这样的问题：如图 1，线段 *AB* 上有一点 *P*，将线段 *AB* 分割为两段——*AP* 和 *PB*，若它们长度之比恰好等于整条线段 *AB* 与较长一段 *AP* 的长度之比，那么 *P* 点应该在线段 *AB* 什么位置呢？

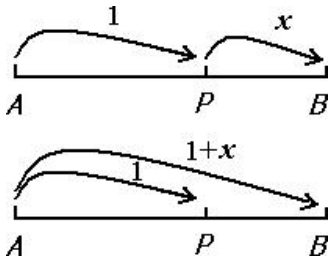


图 1

我们不妨假设线段 *AP* 的长度为 1，设线段 *PB* 的长度为 *x*，那么线段 *AB* 的总长度就是 1+*x*。由 $\frac{AP}{PB}=\frac{AB}{AP}$，得到方程 $\frac{1}{x}=\frac{1+x}{1}$。解出 $x=\frac{\sqrt{5}-1}{2}$，记为*φ*，它的近似值为 0.618。而相应的，线段 *AB* 的长度就是 $1+x=\frac{\sqrt{5}+1}{2}$，记为Φ。

在这之后，人们围绕这个比值开展了许多研究，意大利著名的艺术家、科学家达·芬奇还把*φ*命名为“黄金分割数（Golden Section Number）”。有趣的是，人们发现黄金分割在自然界和其它各个领域中到处可见。例如，人的肚脐是人体总长的黄金分割点，人的膝盖又是肚脐到脚跟的黄金分割点；在有些植

物的茎上，相邻叶柄之间的夹角是 137º28′,这恰好是把圆周分为 1:0.618 的两条半径的夹角。

此外，黄金分割也被看作是美的化身，人们认为符合黄金分割比例的事物都非常协调，富有美感。这就使得它在建筑、绘画、雕塑、摄影和音乐等艺术领域也有着很广泛的应用，例如：在古希腊帕忒农神庙的设计中就存在着许多组黄金分割比；而在荷兰画家梵高的作品《岸边的渔船》中，整幅画的宽和高分别被桅杆和地平线分成两部分，这样的分割也正好符合了黄金分割比例。

其实，在信息学也蕴含着这样的奥妙，本文就将通过两个例题，介绍信息学和黄金分割的一些联系。

例题一

取石子游戏（SHTSC2002,Day2）

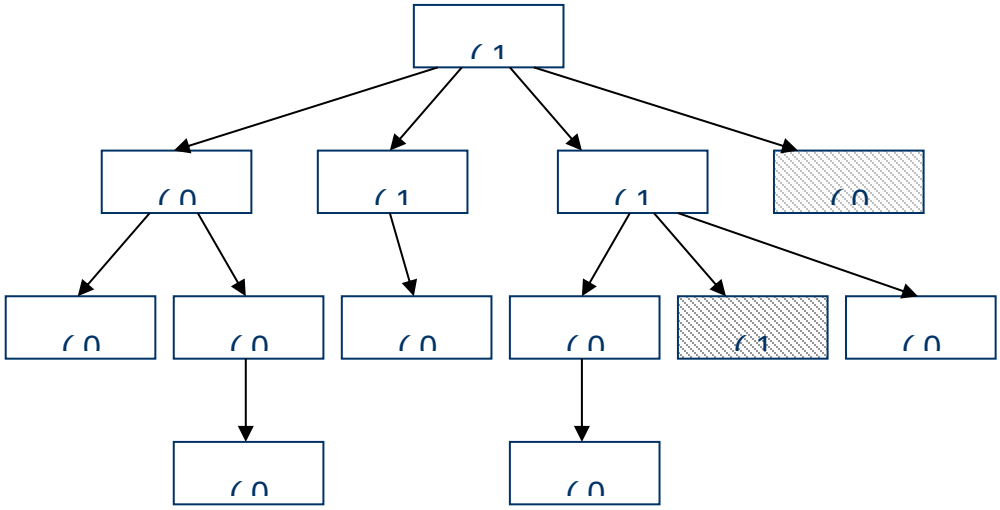
问题描述

有两堆石子，游戏开始后，由两个人轮流取石子，每次有两种取法：一是在任意一堆中取走任意数目的石子；二是可以在两堆中同时取走相同数量的石子。最后把石子全部取完的人是胜者。现在给出初始的两堆石子的数目 a 和 b（$0 \le a,b \le 10^9$），假设双方都采取最好的策略，判断先手是否有必胜策略。

算法一

问题中，两堆石子的个数 a 和 b 决定了最后的胜负。因此，我们用(a,b)作为表示当前局面的“状态”。通过建立博弈树，判断给出的初始状态(a,b)是必胜还是必败。需要注意的是，由于规则中两个石子堆并没有差别，因此状态(a,b)和状态(b,a)实际上是等价的。

来看一个例子，游戏开始时，两堆石子的数目分别是 1 和 2，那么初始状



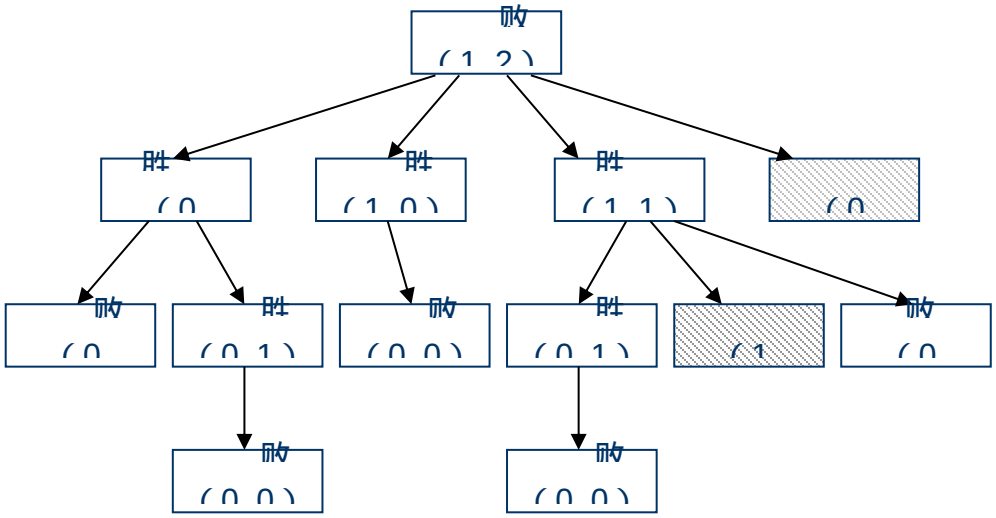
态为(1,2)，自顶向下构造出的博弈树如图 2（其中，重复子状态可以只扩展一个）。

图 2

接下来，就可以分三种情况，判断各个状态的胜负情况：

(1) 如果一个状态没有子状态，根据规则判断胜负；

(2) 如果一个状态至少有一个子状态先手败，那么该状态先手胜；



(3) 如果一个状态的所有子状态都是先手胜，那么改状态先手败。

图 3

这样，我们就可以得出初始状态的胜负情况（如图 3）。具体实现时，可以采用动态规划或记忆化搜索解决。这个算法的空最坏情况下查询的次数约为 2+[log1/ϕ108]

间复杂度为 O(N^2)，时间复杂度为 O(N^3)。附程序 stone_1.pas。

算法二

显然上面的算法并不能有效的解决本题。我们需要进一步挖掘游戏规则的特点。根据两种取石子的方法，可以发现如果状态*(a,b)*是先手败，就能得到下面几种状态必为先手胜：

(1) 状态*(a,i)*（其中，$i \neq b$）或状态*(i,b)*（其中，$i \neq a$）；

(2) 状态*(a+i,b+i)*（其中，$i \neq 0$）。

也就是说，每个正整数在所有先手败状态中将出现且只出现一次，同时任何两个必败状态中两堆石子的差值各不相同。根据这个特点，我们可以构造出所有的先手败状态。首先，最小的先手败状态是(0,0)，根据上面的规律，之后的先手败状态中就不再存在某一堆为 0 个石子的情况，同时也不会出现两堆石子个数相等的情况。因此，第二种先手败的状态中较小的一堆石子个数将是尚未出现的最小的整数 1，而这种状态中两堆石子个数的差也是尚未出现的差值中最小的一个。这样，第二种先手必败的状态就是(1,2)。

由此，我们得到了构造所有先手败状态的方法：开始时只有(0,0)一个先手败状态；第 *i* 次构造时，先找出在已知的先手败状态中没有出现的最小的整数 *a*，则新产生的先手败状态就是*(a,a+i)*。这个算法的时间复杂度和空复杂度均为 O(*N*)。附程序 stone_2.pas。

算法三

虽然算法二的时空复杂度相对算法一有了巨大的进步，但由于本题数据规模巨大，仍然没有办法彻底解决。而这时，我们的分析也陷入如了僵局。那么就让我们先从小规模的情况开始分析。

表 1 中列出了一些构造出来的必败状态（表中小数保留四位精度，下同）。可以发现，后面构造出来的状态中，较小一堆的石子数 *a* 与状态序号 *i*

序号 i	***必败状态(a,b)***	***a / i***	***b / i***
1	(1,2)	1.0000	2.0000
2	(3,5)	1.5000	2.5000
3	(4,7)	1.3333	2.3330
4	(6,10)	1.5000	2.5000
5	(8,13)	1.6000	2.6000
…	…	…	…
997	(1613,2610)	**1.6179**	2.6179
998	(1614,2612)	**1.6172**	2.6172
999	(1616,2615)	**1.6176**	2.6176
1000	(1618,2618)	**1.6180**	2.6180

表 1

序号 i	***Φ·i***	***状态(a,b)***
1	**1**.6180	(**1**,2)
2	**3**.2361	(**3**,5)
3	**4**.8541	(**4**,7)
4	**6**.4721	(**6**,10)
5	**8**.0902	(**8**,13)
…	…	…
997	**1613**.1799	(**1613**,2610)
998	**1614**.7979	(**1614**,2612)
999	**1616**.4160	(**1616**,2615)
1000	**1618**.0340	(**1618**,2618)

表 2

的比值十分接近Φ！这提示我们，也许 *a* 的值与Φ·*i* 有一定的关系。所以，我们反过来观察*Φ·i* 与 *a* 之间的关系。

在表 2 列出的小规模情况中，第 *i* 个必败状态中较小一堆的石子数 *a* 都恰好等于Φ·*i* 的整数部分。于是我们提出了这样的猜想：

第 i 个必败状态是 **([Φ·*i*] , [(Φ+1)·*i*])**[17]。

可以证明，这个猜想是正确的[1]。因此，我们得出结论，对于状态*(a,b)*（其中，*a*≤*b*），如果 $\Phi\cdot\left(\left[\frac{\Phi}{a}\right]+1\right)=a$ 且 $a+\left[\frac{\Phi}{a}\right]+1=b$，则先手必败，否则先手必胜。

这样算法的时空复杂度均为常数级的，问题得到了圆满的解决。附程序 stone_3.pas。

[17] 本文中[*x*]表示 *x* 的整数部分，{*x*}表示 *x* 的小数部分。

解题小结

表 3 将上述三个算法进行了比较。算法一是解决博弈问题的常规方法，同时定义了状态；算法二，深入挖掘了游戏规则，找出了其中的特性；算法三则通过分析一些小规模的数据，找到了常数级的算法。通过这样不断的分析和挖掘，终于找到了问题本质上与黄金分割的联系，发现了这样一个博弈问题中蕴含的美。

	时间复杂度	空间复杂度	基本算法
算法一	$O(N^3)$	$O(N^2)$	根据博弈树进行动归
算法二	$O(N)$	$O(N)$	构造必败状态
算法三	$O(1)$	$O(1)$	直接判断胜负情况

表 3

例题二

登山问题（根据经典问题改编）

问题描述

单峰函数[2]$f(x)$在区间$[0,L]$上有极大值，要求通过一系列查询，找到这个极大值的横坐标 x_0。每次查询中，向交互库[3]提供一个查询点 t，交互库将返回 $f(t)$。

数据范围：$0<L\le10^4$，要求查询次数不超过 40 次，且计算结果与最优点 x_0 误差不得超过 10^{-3}。

算法分析

对于区间内的两个不同的查询点，我们将函数值较大的称为好点，将函数值较小的称为差点。下面，我们将说明，***最优点和好点必在差点的同侧***。

证明 首先需要明确，最优点显然不会与差点重合。

如果最优点与好点重合，那么命题显然成立。

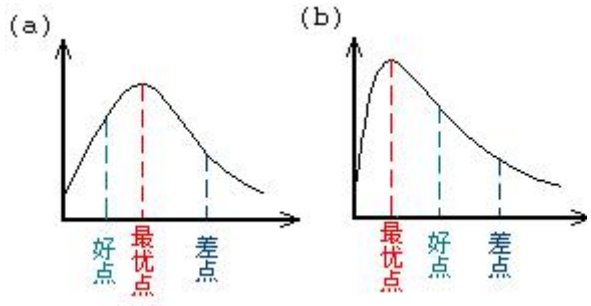


图 4

如果最优点并不在好点的位置，那么将存在下面两种情况：

(1) 如图 4(a)，好点与差点分别位于最优点的两侧，此时命题显然成立；

(2) 如图 4(b)，好点与差点位于最优点同侧；此时，由于 *f(x)*是单峰函数，离最优点较近的查询点必然是好点，因而最优点与好点仍在差点同侧。

证毕。

上面的结论，提示我们，可以通过比较目标区间内的两个查询点位置上的函数值，删去一部分目标区间。这样逐步缩小目标范围，不断逼近最优点，直到达到允许的误差范围内为止。

原题中，允许的误差范围是原区间长度的千万分之一，为了保证精度，我们需要将区间范围缩小为原来的 $1/10^8$。如何尽快的缩小目标范围，就成了我们需要研究的问题。而解决这个问题的关键，就是合理的选择查询点的位置。

下面我们就来分析一下具体应该如何选取查询点的位置。

设当前目标范围为*[a,b]*，两个查询点为 x_1 与 x_2，$x_1<x_2$。

在查询之前，我们无法预先知道两个查询点孰好孰差，也就是说两个查询点成为差点的可能性是相同的。因此我们无法确定究竟会去掉哪一段区间范

围。所以，我们在安排查询点时应该使它们关于当前目标范围的中点对称，即应使 $x_1–a = b-x_2$。这是我们在查询过程中应遵循的第一个原则——**对称原则**。

“对称原则”是十分常用的，例如我们经常在一些查找中使用的“二分法”，就完全符合这个原则。我们也就很容易想到使用二分法来解决本题：每次在当前目标范围中点附近，选择两个非常接近的查询点，并且根据这两个查询点的函数值，确定最优点的范围，缩小目标区间。这样每次可以舍去接近原区间一半长度的部分，为了保证精度，我们需要进行的查询次数就在 $2\log_2 10^8$ 左右，约为 54 次，达不到题目中的要求。

通过分析，我们可以发现造成查询次数较多的原因是，每次为了去掉区间的一半，都需要重新进行两次查询；而实际上，原来的好点仍然处在当前的区间中，二分法却并没有利用到它的函数值信息。由此，我们想到，每次可以只在当前区间中进行一次查询，并与原来的好点进行比较，得出新的好点和差点，进而继续缩小目标范围。

由于我们每次选择的查询点要满足*对称原则*，因此，最开始选择的两个查询点的位置就决定了后面每次可以舍取得区间长度。我们发现，如果像二分法中一样，让 x_1 与 x_2 都尽量靠近，这样一次可以舍去整个因素范围*[a,b]*的将近 50%，但是接下来每次只能舍去很小的一部分了，反而不利于较快地逼近最优点。这个情况又提示我们：最好每次舍去的区间都在全区间中占同样的比例（我们不妨称之为**"成比例的舍去"原则**）。

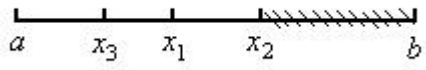


图 5

按照上述两个原则，如图 5 所示，设第一次和第二次查询分别在 x_1 点和 x_2 点，$x_1<x_2$，则在第一次比较结果后，由*对称性*，舍去的区间长度都等于 $b-x_2$。不妨设 x_1 是好点，x_2 是差点，舍去的是(x_2,b)。再设第三次试验安排在 x_3 点，则 x_3 点应在$[a,x_2]$中与 x_1 点对称的位置上。同时 x_3 点应在 x_1 点左侧，否则 x_3 点与 x_1 点比较查询结果后被舍去的将与上次舍去的是同样的长度，而不是同样的比例，违背*“成比例的舍去”原则*。由此可知，x_3 点与 x_1 点比较后，被舍去的区间长度都等于 x_2-x_1。于是按照*“成比例地舍去”原则*，我们得到等式：

$$\frac{b-x_2}{b-a}=\frac{x_2-x_1}{x_2-a}$$

经过变形、化简可得

$$\frac{x_2-a}{b-a}=\frac{x_1-a}{x_2-x_1}$$

这个等式和引言中关于线段分割的方程十分类似，这表明，x_2 正处于区间$[a,b]$的黄金分割点上。也就是说，查询点安排在区间的黄金分割点处较为妥当。因此，我们也将这种逼近最优值的方法称为“黄金分割法”。

效率分析

由于每次查询点都在当前目标区间的黄金分割点处，每次保留下来的区间和原来区间的长度比都是ϕ:1。因此，最坏情况下查询的次数约为 $2+[\log_{1/\phi}10^8]$，恰好在 40 次之内，可以圆满的解决问题。附程序 explorer.pas。

解题小结

在这个问题中，我们摆脱了“二分法”思想的束缚，充分利用每次查询的信息，提出了两个选择查询点时需要遵循的原则。而由于黄金分割所特有的神奇的比例性质，恰好符合了*“成比例的舍去”原则*，为解决本题提供了出路。

总结

上文中两个例题的解决，都与“黄金分割数”有着密切的联系。

例一构造的数对中，蕴藏着黄金分割数的规律。其实信息学竞赛中，常常会出现一些数学性较强的题目。而黄金分割数，恰恰和构造数列、矩阵等密切相关，也就往往会成为解决这类题目的钥匙。

例二中的解决中，黄金分割所特有的比例性质使得它成为解决问题的关键。而在一些类似的查询问题中，黄金分割都有着广泛的应用，理论上可以证明，上面介绍的黄金分割法，查询的次数是各种优选法中最优的[18]。

当然，黄金分割与信息学竞赛的联系远不止这些，也还有很多奥妙需要我们去探索和发现。法国雕塑家罗丹曾说过：“美，是无处不在的，对于我们的眼睛，缺少的不是美，而是发现。”其实，只要勤于思考、勇于探索，每个人都可以拥有一双善于发现美的慧眼。

参考文献

[1]　姜璐主编，《中国少年儿童百科全书——科学·技术》，第 2 版，浙江教育出版社，1994.12.

[2]　<美>Tobias Dantzig 著，苏仲湘译，《数：科学的语言》，第 1 版，上海教育出版社，2000.12

[3]　《数学手册》编写组，《数学手册》，第 1 版，人民教育出版社，1979.05

[18] 具体证明参见“参考文献[6]”。

[4]　张维勇主编，《青少年信息学奥林匹克竞赛试题与解析（安徽省 1994～2004 年）》，第 1 版，合肥工业大学出版社，2004.08

[5]　骆骥，《浅析解“对策问题”的两种思路——从〈取石子〉问题谈起》，2002 年国家集训队论文，2002.02

[6]　华罗庚，《优选学》，第 1 版，科学出版社，1981.04

附录

[1] 例题一，算法三，猜想的证明。

定义 1.1 根据算法二构造列数为 2 的矩阵 A，使得必败状态依次为 $(A_{1,1},A_{1,2})$，$(A_{2,1},A_{2,2})$，$(A_{3,1},A_{3,2})$,……。

定义 1.2 根据猜想构造列数为 2 的矩阵 B，其中，$B_{i,1}=[\Phi\cdot i]$，$B_{i,2}=[(\Phi+1)\cdot i]$。

引理 1.1(Betty 定理) 如果 a 和 b 是两个正无理数，且满足

$$\frac{1}{a}+\frac{1}{b}=1$$

且 $P=\{x \mid x=[at], t\in Z^+\}$，$Q=\{x \mid x=[bt], t\in Z^+\}$，则 P 和 Q 是正整数集 Z^+的一个划分，即

$$P\cup Q=Z^+, P\cap Q=\varnothing$$

证明 对于任意正整数 n，设 P 集合中小于 n 的元素的个数为 x，同样 Q 集合中小于 n 的个数为 y。

那么显然有

$$x<\frac{n}{a}, y<\frac{n}{b}$$

由于 a, b 都是无理数，因此$\frac{n}{a},\frac{n}{b}$都不可能是整数，所以有

$$x=[\frac{n}{a}],y=[\frac{n}{b}]$$

那么考察 $x+y$

$$x+y=[\frac{n}{a}]+[\frac{n}{b}]<[\frac{n}{a}]+\{\frac{n}{a}\}+[\frac{n}{b}]+\{\frac{n}{b}\}=\frac{n}{a}+\frac{n}{b}=n$$

而

$$n-(x+y)=\{\frac{n}{a}\}+\{\frac{n}{b}\}<2$$

综上，有

$$n-2<x+y<n\Rightarrow x+y=n-1$$

那么可以知道，对于任意 n，在 P 和 Q 集合中小于 n 的元素共有(n - 1)个；而小于(n + 1)的元素共有 n 个。

所以，P, Q 两个集合中等于 n 的元素**有且仅有** 1 个！

因此 Z^+中的每一个元素不是在 P 集合中，就是在 Q 集合中，不可能既不在 P 中也不在 Q 中，也不可能既在 P 中又在 Q 中。所以说

$$P\cup Q=Z^+,P\cap Q=\text{Ø}$$

推论 1.1 如果 a 和 b 是两个正无理数，且满足：

$$\frac{1}{a}+\frac{1}{b}=1$$

那么任意正整数 x，可以且只可以表示为[at]和[bt]中的一种形式。（$t\in Z^+$）

推论 1.2 任意正整数 x 都在矩阵 B 中出现且只出现一次。

证明 由于有等式：

$$\frac{1}{\Phi}+\frac{1}{\Phi+1}=1$$

根据*推论 1.1* 和*定义 1.2*，*推论 1.2* 成立。

引理 1.2 对于任意的 i>1，都有 $B_{i,1}$ 是除矩阵 B 的前 i-1 行中数字外最小的正整数。

证明 根据*定义 1.2*，比 $B_{i,1}$ 小的整数都应出现在矩阵的前 *i*-1 行中。

在矩阵第一列中，所有数字都可以表示成为[Φ·*t*]的形式。其中，比[Φ·*i*]小的数字个数为$\left[\frac{[\Phi\cdot i]}{\Phi}\right]$。同理，矩阵 *B* 的第二列中，比[Φ·*i*]小的数共有$\left[\frac{[\Phi\cdot i]}{\Phi+1}\right]$个。

因此，在整个矩阵的前 *i*-1 行中出现的比[Φ·*i*]小的整数个数为

$$\left[\frac{[\Phi\cdot i]}{\Phi}\right]+\left[\frac{[\Phi\cdot i]}{\Phi+1}\right]<\frac{[\Phi\cdot i]}{\Phi}+\frac{[\Phi\cdot i]}{\Phi+1}=[\Phi\cdot i]$$

同时，有

$$\left[\frac{[\Phi\cdot i]}{\Phi}\right]+\left[\frac{[\Phi\cdot i]}{\Phi+1}\right]=\frac{[\Phi\cdot i]}{\Phi}+\frac{[\Phi\cdot i]}{\Phi+1}-\left\{\frac{[\Phi\cdot i]}{\Phi}\right\}-\left\{\frac{[\Phi\cdot i]}{\Phi+1}\right\}>[\Phi\cdot i]-2$$

因此，在整个矩阵的前 *i*-1 行中出现的比[Φ·*i*]小的整数个数为[Φ·*i*]-1，综合*推论 1.2* 可以得出，$B_{i,1}$ 是除矩阵 *B* 中前 *i*-1 行中的数以外最小的正整数。

定理 1 矩阵 *A* 和矩阵 *B* 等价，即对任意的 *i* 均有 $A_{i,1}=B_{i,1}$ 且 $A_{i,2}=B_{i,2}$。

对行数 *i* 进行归纳：当 *i*=1 时，显然有

$A_{1,1}=B_{1,1}$，$A_{1,2}=B_{1,2}$。

假设 *i*<*k* 是均有 $A_{i,1}=B_{i,1}$ 且 $A_{i,2}=B_{i,2}$。根据*定义 1.1*，$A_{k,1}$ 是除掉矩阵 *A* 中前 *k*-1 行中数字外最小的正整数。而根据*引/理 1.2*，也有 $B_{k,1}$ 是除掉矩阵 *B* 中前 *k*-1 行中数字外最小的正整数。根据归纳假设，有

$$A_{k,1}=B_{k,1}$$

根据算法二的构造方法，有 $A_{k,2}=A_{k,1}+k$。而根据*定义 1.2* 有

$$B_{k,2}=[(\Phi+1)\cdot k]=[\Phi\cdot k]+k=B_{k,1}k$$

因此，可以得出

$$A_{k,2}=B_{k,2}$$

综上，定理成立，算法三中关于必败状态的猜想也成立。

[2] 例题二，单峰函数的定义

根据题目，这里只给出存在极大值的单峰函数 $f(x)$定义：

设 x_0 是区间$[a,b]$上的函数最大点，则有

(1) 对于任意的 $x_1<x_2<x_0$，都有 $f(x_1)<f(x_2)<f(x_0)$；

(2) 对于任意的 $x_0<x_1<x_2$，都有 $f(x_0)>f(x_1)>f(x_2)$。

[3] 例题二，交互库

C++程序员使用的交互库：tools_c.h

Pascal 程序员使用的交互库：tools_p.ppu

特别感谢，安徽省芜湖市第一中学的周冬同学为本题编写交互库。

程序描述

[1] Stone_1.pas

```
Program Stone_Algorithm1;

Const
  inf='stone.in';                                    //输入文件
  outf='stone.out';                                  //输出文件
  maxn=100;

Var
  a,b:longint;                                       //石子个数
  ans:array[0..maxn,0..maxn]of longint;              //各个状态的胜负情况

function dfs(a,b:longint):longint;                   //深度搜索
var i,t:longint;
begin
```

```
  if ans[a,b]<>0 then begin                          //状态(a,b)已经计算过
    ans[b,a]:=ans[a,b];
    dfs:=ans[a,b]; exit;
  end;
  if ans[b,a]<>0 then begin                          //状态(b,a)已经计算过
    ans[a,b]:=ans[b,a];
    dfs:=ans[a,b]; exit;
  end;

  for i:=1 to a do                                   //在第一堆中
```

取 i 个石子

```
    begin
      t:=dfs(a-i,b);
      if t=1 then begin                              //有一个子状态先手败
        ans[a,b]:=2; ans[b,a]:=2;                    //先手胜
        dfs:=2;
        exit;
      end;
    end;
  for i:=1 to b do                                   //在第二堆中
```

取 i 个石子

```
    begin
      t:=dfs(a,b-i);
      if t=1 then begin                              //有一个子状态先手败
        ans[a,b]:=2; ans[b,a]:=2;                    //先手胜
        dfs:=2;
        exit;
      end;
    end;
  for i:=1 to a do                                   //在两堆石子
```

中同时取走 i 个

```
    if i<=b then begin
```

```
      t:=dfs(a-i,b-i);
      if t=1 then begin                                  //有一个子状态先手败
        ans[a,b]:=2; ans[b,a]:=2;                        //先手胜
        dfs:=2;
        exit;
      end;
    end
    else break;
  dfs:=1;                                                //所有子状态
都先手胜，该状态先手败
end;

Begin
  assign(input,inf); reset(input);
  assign(output,outf); rewrite(output);
    fillchar(ans,sizeof(ans),0);                         //初始化
    ans[0,0]:=1;                                         //状态
(0,0)，先手败
    while not(seekeof) do                                //多组数据
     begin
       read(a,b);                                        //读入 a,b
       ans[a,b]:=dfs(a,b);                               //进行搜索
       writeln(ans[a,b]-1);                              //输出结果
     end;
  close(input); close(output);
End.
```

[2] Stone_2.pas

```
Program Stone_Algorithm2;

Const
  inf='stone.in';                                        //输入文件
```

```
  outf='stone.out';                                          //输出文件
  maxn=1000000;

Var
  a,b:longint;                                               //石子个数
  f:array[0..maxn]of boolean;                                //各整数是否出现

procedure swap;                                              //交
换 a 和 b
var t:longint;
begin
  t:=a; a:=b; b:=t;
end;

procedure work;                                              //计算过程
var i,t:longint;
begin
  fillchar(f,sizeof(f),0);                                   //初始化
  t:=0;                                                      //构
造出的比败状态的个数
  for i:=0 to a-1 do
    if not f[i] then begin                                   //整数 i 尚未出现
       f[i+t]:=true;
       inc(t);
    end;
  if not(f[a]) and (a+t=b)                                   //判断
    then writeln(0) else writeln(1);
end;

Begin
  assign(input,inf); reset(input);
  assign(output,outf); rewrite(output);
    while not(seekeof) do                                    //多组数据
```

```
    begin
      read(a,b);                                    //读入 a 和 b
      if a>b then swap;                             // a≤b
      work;                                         //计算过程
    end;
  close(input); close(output);
End.
```

[3] Stone_3.pas

```
Program Stone_Algorithm3;

Const
  inf='stone.in';                                   //输入文件
  outf='stone.out';                                 //输出文件
  phi=(sqrt(5)+1)/2;                                //黄金分割数

Var
  a,b:longint;                                      //石子个数

procedure swap;                                     //交
换 a 和 b
var t:longint;
begin
  t:=a; a:=b; b:=t;
end;

procedure main;
var t:longint;
begin
  assign(input,inf); reset(input);
  assign(output,outf); rewrite(output);
    while not(seekeof) do                           //多组数据
    begin
      read(a,b);                                    //读入 a 和 b
```

```
        if a>b then swap;                                    //a≤b
        if a+b=0 then begin                                  //a=b=0，先手败
           writeln(0); continue;
        end;
        t:=trunc(a/phi+1);
        if (a=trunc(t*phi)) and (b=a+t)              //判断胜负情况
        then writeln(0) else writeln(1);
      end;
  close(input); close(output);
end;

Begin
  main;
End.
```

[4] Explorer.pas

```
Uses tools_p;

Const
  zero=1e-4;                                                 //误差
  phi=(sqrt(5)-1)/2;                                         //黄金分割数

Var
  L,h,t:extended;                                            //区间长度L，当前
目标区间[h,t]

procedure work;                                              //计算过程
var lastf,lastx,x,f:extended;
begin
  lastx:=t*phi;                                              //第一个查询点
  lastf:=ask(lastx);                                         //查询函数值
  while abs(h-t)>zero do                                     //直到在误差范围内为止
   begin
     x:=h+t-lastx;                                           //根据对称原则，找到查询点
```

```
        f:=ask(x);                                          //查询函数值
        if f>lastf then begin                               //判别好点差点
                    if x>lastx then h:=lastx                //缩小目标范围
                    else t:=lastx;
           lastx:=x; lastf:=f;                              //更改区间内好点
        end
        else begin
                    if x>lastx then t:=x                    //缩小目标范围
                    else h:=x;
        end;
     end;
end;

Begin                                                       //主
过程
   L:=Start;                                                //初始化
   h:=0; t:=L;                                              //开始目标区间位
[0,L]
   work;                                                    //计算过程
   Answer((h+t)/2);                                         //返回最终结果
End.
```

取石子游戏
(STONE)

有两堆石子，数量任意，可以不同。游戏开始由两个人轮流取石子。游戏规定，每次有两种不同的取法，一是可以在任意的一堆中取走任意多的石子；二是可以在两堆中同时取走相同数量的石子。最后把石子全部取完者为胜者。现在给出初始的两堆石子的数目，如果轮到你先取，假设双方都采取最好的策略，问最后你是胜者还是败者。

输入输出要求

输入文件由若干行，表示若干种石子的初始情况，其中每一行中包含两个非负整数 *a* 和 *b*，表示两堆石子的数目，*a* 和 *b* 均不大于 1,000,000,000。

输出文件对应也是若干行，每行包含一个数字 1 或 0，如果最后你是胜利者，则为 1，反之，则为 0。

样例输入和相应输出

样例一

输入	**输出**
6 10	0

样例二

输入	**输出**
2 1 8 4 4 7	0 1 0

登山问题

(EXPLORER)

【问题描述】

一些 OIer 准备去翻越 OI Mountain，一条雄伟的山脉。但是开始登山之前，OIers 希望先知道这条山脉的最高点的位置。

山脉的横截面都是相同的，而且最高点两侧，山的高度也依次递减。OIer 们可以使用卫星探测系统来测量某一具体位置上山的高度。每次，他们发出指令 Explorer(*x*)，这时卫星就会返回山脉横截面中，坐标为 *x* 的位置上，山的海拔高度。

但是由于和卫星通信总是很慢☹，OIers 希望能够在 40 次内找出那个最高点。你能做到么？

【如何调用交互库】

测试库提供三个函数：Start，Ask，Answer，它们的作用和用法如下：

- Start()必须首先调用，用它来获得实数 *L* 的值。
- Ask(X)的作用是询问，用它来获得函数 *f(x)*的值。
- Answer(Ans)用来告诉测试库你得到的答案。 $Ans \in [0, L]$，表示你的程序判断函数 *f(x)*在 *Ans* 处取到最大值。

你的程序不得自行终止且不得访问任何文件。

【对 Pascal 程序员的提示】

你的程序应当使用下列语句引用测试库：

uses tools_p;

测试库提供的函数/过程原型为：

function Start:double;
function Ask(x:double):double;
procedure Answer(ans:double);

【对 C++程序员的提示】

你程序头加上一行：

#include “tools_c.h”

并且在 *RHIDE* 中设置 *Option->Linker* 为 *tools_c.o*

测试库提供的函数原型为：

double Start();
double Ask(double x);
double Answer(double ans);

【数据说明】

如果问题中区间范围为[0,2]，$f(x)=-(x-1)^2+10$，那么一种可能满分的调用方法如下：

Pascal 选手的调用方法	**C++选手的调用方法**	**说明**
Start;	Start();	返回值 2，表示区间长度
Ask(0);	Ask(0);	返回值 9，表示 $f(0)=9$
Ask(1.5);	Ask(1.5);	返回值 9.75
Ask(1);	Ask(1);	返回值 10
Ask(2);	Ask(2);	返回值 9
Answer(1);	Answer(1);	返回结果，结束程序

注意，该例子只是对库函数的使用说明，并没有算法上的意义。

这里 L 最大为 10^4，要求最后返回的答案与最高点横坐标误差不超过 10^{-3}。山的高度在[0, 8848.13]这个区间内。

【如何测试你的程序】

- 在当前目录下建立文件 *explorer.in*，其中包含一个实数 *L*，表示为函数 *f(x)*的定义域为 0—*L*,我们的交互库将会自动生成 $f(x) \mid x \in [0, L]$的值。
- 运行你的程序
- 运行结束后，交互库将会把整个对话记录在文件 *explorer.log* 中

【如何了解错误信息】

如果对话文件 *explorer.log* 中包行一行或多行以 **Error:**开头的信息，则表示你的程序在运行过程中发生了错误。具体的错误信息如下所示。

Error: Input file explorer.in not found

交互库在当前目录下没有发现输入文件 *explorer.in*。

Error: Start must call first!

你的程序没有初始化交互库。

Error: You have called Start twice

你的程序尝试多次调用 *Start* 函数。

Error: Parameter is out of range

你的程序调用 *Ask* 函数的参数不再$[0, L]$内。

Error: You have called Ask more than 60 times

你的程序过多的调用了 *Ask* 函数（超过 60 次）。

浅谈二分策略的应用

华东师大二附中 杨俊

【摘要】本文着重讨论三种不同类型的二分问题，意在加深大家对二分的认识。它们所考虑的对象从一般有序序列，到退化了的有序序列，最后到无序序列。事实上它们也正代表了二分策略的三种不同应用。

【关键字】二分、序、应用

【正文】

“二分”，相信这个词大家都再熟悉不过了。二分是一种筛选的法则，它源于一个很简单的想法——在最坏情况下排除尽可能多的干扰，以尽可能快地求得目标。

二分算法的高效，源于它对信息的充分利用，尽可能去除冗余，减少不必要的计算，以极大化算法效率。事实上许多二分问题都可以用判定树或其它一些定理来证明，它达到了问题复杂度的下界。

尽管二分思想本身很简单，但它的扩展性之强、应用面之广，或许仍是我们所未预见的。大家也看到，近年来各类竞赛试题中，二分思想的应用不乏令人眼前一亮的例子。下面是作者归纳的二分思想的三种不同类型的应用，希望能让读者有所收获。

类型一：二分查找——应用于一般有序序列

申明：这里所指的有序序列，并不局限于我们通常所指的，按从小到大或是从大到小排好序的序列。它仅包含两层意思：第一，它是一个**序列**，一维

的；第二，该序列是**有序**的，即序列中的任意两个元素都是可以比较的。也就是拥有我们平时所说的全序关系。

虽说二分查找大家都再熟悉不过了，但这里还是先简要地回顾一下二分查找的一般实现过程：

（**1**） 确定待查找元素所在范围

（**2**） 选择一个在该范围内的某元素作为基准

（**3**） 将待查找元素的关键字与基准元素的关键字作比较，并确定待查找元素新的更精确的范围

（**4**） 如果新确定的范围足够精确，输出结果；否则转至（**2**）

让我们看一个经典问题——**顺序统计问题**

[问题描述]

给定一个由 n 个不同的数组成的集合 S，求其中第 i 小的元素。

[分析]

相信大家对这个问题都很熟悉，让我们回顾一下二分查找是如何应用于该问题上的。

（**1**） 确定待查找元素在 S 中

（**2**） 在 n 个元素中随机取出一个记为 x，将 x 作基准

（**3**） 设 S 中比元素 x 小的有 p 个。

如果 i<p，表示我们所要寻找的元素比 x 小，我们就将范围确定为 S 中比 x 小的元素，求该范围内第 i 个元素；

如果 i>p，表示我们所要寻找的元素比 x 大，我们就将范围确定为 S 中比 x 大的元素，求该范围内第 i-p-1 个元素；

如果 i=p，表示第 i 小的元素就是 x。

（4） 如果找出 x，输出；否则转至**（2）**

因为 x 是随机选出的，由简单的概率分析，可得算法的复杂度期望值为 O(n)。

[小结]

举这个例子，是想提醒大家两点：

第一，不要想当然认为二分查找就一定与 logn 有关。算法中的第**（3）**步，即我们通常所说的“分”并不要求每一次都必须在 O(1)时间进行，“分”可以是建立在对序列的有效处理之上（比如上面这个例子中使用了类似于快速排序中的集合分割）。

第二，二分算法的“分”并不要求每次都必须平均（因为有时候可能很难做到这一点），只要不是每次都不平均就已经可以产生高效的算法了，这样也给使用随机化算法带来了契机。

近年来由于交互式试题的出现，也给予二分查找更多活力。相当多的二分查找问题都是以**交互式**试题的形式给出的。比如说，就上面这个例子，摇身一变就成了一道交互式的**中等硬度**问题（IOI2000）。两个题目如出一辙：你问第 i 小的，我问第(N+1)/2 小的；解法当然也就依样画葫芦：你用随机取出的 x，依照与 x 大小关系分成两段，我就随机取出 x,y，依照与 x,y 大小关系分成三段；你的复杂度期望是 O(n)，我的询问次数的期望也是 O(N)。（具体细节这里不再展开，有兴趣的同学可以参考前辈的解题报告[19]）

[19] 2000-2002 集训队论文《中等硬度解题报告》高岳

类型二：二分枚举——应用于退化了的有序序列

二分策略并不总是应用于上述这样显式的有序序列中，它可能借助于问题某种潜在的关联性，用于一些隐含的退化了的有序序列中。与先前介绍的二分查找相比，最大的区别在于这里的二分在判断选择哪一个部分递归调用时没有比较运算。

我们还是先看一个问题——**BTP 职业网球赛**（USACO contest dec02）

[问题描述]

有 N 头奶牛（N 是 2^K），都是网球高手，每头奶牛都有一个 BTP 排名（恰好为 1—N）。下周将要进行一场淘汰赛，N 头奶牛分成 N/2 组，每组两头奶牛比赛，决出 N/2 位胜者；N/2 位胜者继而分成 N/4 组比赛……直至剩下一头牛是冠军。

比赛既要讲求实力，又要考虑到运气。如果两头奶牛的 BTP 排名相差大于给定整数 K，则排名靠前的奶牛总是赢排名靠后的；否则，双方都有可能获胜。现在观众们想知道，哪头奶牛是所有可能成为冠军的牛中排名最后的，并要求你列举出一个可能的比赛安排使该奶牛获胜。（限制 N≤65536）

[分析]

由于 N 很大，我们猜想可以通过贪心方法解决。

我们希望排名靠前的选手总是“爆冷”输给排名靠后的选手。于是我们让 1 输给 K+1、2 输给 K+2……每一轮中每一局总是选择未比赛的排名最前的选手，输给一个排名最靠后的选手（如果有的话）。

但我们很容易找到反例，例如：N=8, K=2，由上述贪心方法我们得到对阵方式结果为 4，见图 BTP-1（其中 X→Y 表示 X 战胜 Y）。

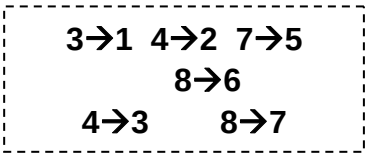


图 BTP-1

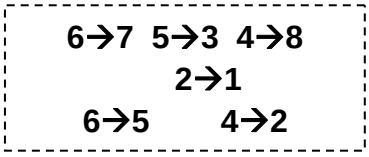


图 BTP-2

但最优解为 6，见图 BTP-2。究其原因，因为我们不知道最优解是多少，所以我们是在盲目地贪心。事实上，最优解的 6 在第一轮就被我们淘汰了，当然就得不到最优解喽！

要想知道最优解？枚举！同时考虑到一个很显然的结论——**如果排名为 X 的选手最终获胜，那么排名在 X 前的选手也可以获胜**。

显然归显然，证明它我们还需要动一点小脑筋。假设排名 X 的选手最终获胜，我们通过对该对战方式的局部修改，构造出一种新的对战方式使任意排名 Y<X 的选手获胜：在 X 最终获胜的对战方式中，假设 Y 是被 Z 击败。如果 Z≠X，由 X>Y，可知 Z 一定也能击败 X，且同一轮及此后 X 所击败的选手都能被 Y 击败，所以我们只需要在此轮让 Z 击败 X，并把之后所有对战中的 X 都改成 Y 即可；如果 Z=X，由 X>Y，我们就让 Y 击败 X，同样把之后对战中的 X 都改成 Y，则最后获胜的也是 Y。

因此，我们就可以用二分枚举最终获胜的 X，大大提高了算法效率。

现在的问题是，如果我们知道最终获胜的是 X，我们能否很快构造出一种对战方式或是证明不存在吗？可以，我们仍旧使用贪心，不过因为我们知道最终谁获胜，所以我们采用倒推。

每一轮，我们都让已有选手去战胜一个排名最靠前的还未出现的选手，由该方法产生的对战方式就如图 BTP-2 所描述那样。至于最靠前的未出现选手如

何找，我们可以采用静态排序二叉树实现，这里不再展开，读者可以自己考虑。

可以证明这样贪心是正确的（证明方法同前面的证明类似，这里也不再重复）。整个问题可以在 $O(N\log^2 N)$时间完成。

[小结]

对于这类需要二分枚举的问题，其实算法的根本是在一个隐含的退化了的有序序列中进行二分查找，只是这个序列仅含有 0 和 1 两种值。上例中当 X<Ans，A[X]=0；当 X≥Ans，A[X]=1。而我们所寻找的就是这两种值的分界点。有了分界点，就有了最优值，也就有了原问题的解。

让我们再看一个问题——**奖章分发**（ACM/ICPC CERC 2004）

[问题描述]

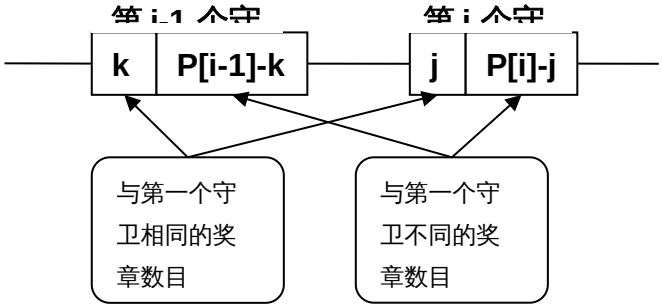
有一个环形的围墙，围墙上有一些塔，每个塔中有一个守卫。现在要发给每个守卫一些奖章，第 i 个守卫需要 P[i]个奖章。每个守卫自己的奖章必须都是不同种类的，而且相邻的两个守卫得到的奖章也不能有任何一个相同。问至少应准备多少种不同的奖章。（限制：2≤n≤10000，1≤P[i]≤100000）

[分析]

假如围墙不是环形的，我们很容易用贪心算法解决：每次发给守卫尽可能多的已准备的奖章，如果不够就再新准备若干种以满足需要。但现在围墙是环形的，最后一个守卫与第一个守卫也不能有重复的奖章，这就是问题的难点。

于是我们尝试将这一难点引入状态，用动态规划求解（动态规划是求解最优性问题的一种常用方法）。

令 a[i,j]表示前 i 个守卫已安排妥当，且第 i 个守卫有 j 个奖章与第一个守卫相同，则除第一个守卫已有的 P[1]种奖章外，至少还需要的奖章种数。



如图，假设第 i-1 个守卫有 k 个奖章与第一个守卫相同，由于这 k 个奖章与第 i 个守卫的 j 个奖章必须互不相同，易知 j+k≤P[1]；又由于第 i-1 个守卫的另外 P[i-1]-k 个奖章必须与第 i 个守卫的另外 P[i]-j 个奖章不同，设需要增加 X 种奖章，则

a[i-1,k]+X ≥ (P[i]-j)+(P[i-1]-k)

得到 **X ≥ (P[i]-j)+(P[i-1]-k) - a[i-1,k]**

于是我们有

$$a[i,j]=\min_{0\le k\le P[i-1],k\le P[1]-j}\{a[i-1,k]+\max\{0,(P[i]-j)+(P[i-1]-k)-a[i-1,k]\}\}$$
$$=\min_{0\le k\le P[i-1],k\le P[1]-j}\{\max\{a[i-1,k],P[i]+P[i-1]-j-k\}\}$$

显然就这样做复杂度高达 O(n*Pmax^2)。即使使用了优化，也很难得到令人满意的结果，我们只得另辟蹊径。

考虑到**如果有 P[1]+X 种奖章，存在一种分发方案满足要求，那么如果我们有 P[1]+X+1 种奖章，也一定存在可行方案**（大不了其中一种我们不用）。这个性质非常重要，因为由此，我们可以就用二分枚举 X，再逐个判断是否有可行方案的方法求得结果。

所以原先问题就转化为——如果我们已经知道共有 P[1]+X 种奖章，我们能否很快判断是否存在满足要求的分发方案呢？答案是肯定的。

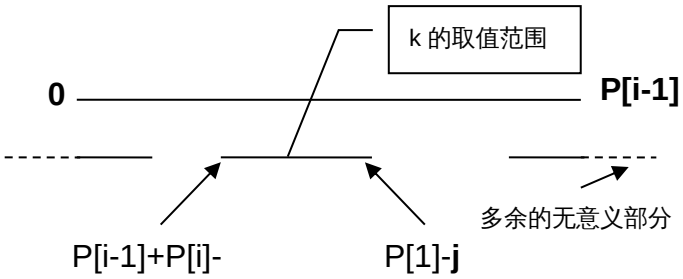
方法仍旧是动态规划[20]，状态表示也类似，a[i,j]表示共有 P[1]+X 种奖章，前 i 个守卫已安排妥当，且第 i 个守卫有 j 个奖章与第一个守卫相同是否可能。

则状态转移变为：

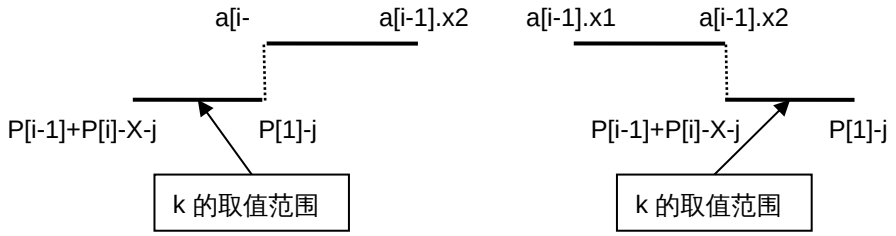
$$a[i,j] = \mathop{\mathbf{or}}_{\substack{0\le k\le P[i-1],\ k+j\le P[1],\\ P[i-1]-k+P[i]-j\le X}} (a[i-1,k])$$

表面上看状态转移仍旧很繁琐，k 的取值有很多限制。但我们仔细观察，第二、三条合起来是 **P[i-1]+P[i]-X-j ≤ k ≤ P[1]-j**。

对于确定的 i，k 的取值范围在数轴上的表示是一条长度确定的线段，且线段的位置由 j 确定。原先第一条限制只不过是再给线段限定一个范围，去除多余无意义的部分（如图）。



初始状态 a[1]中只有 a[1,0]=true，其余均为 false。由归纳法很容易证明对任意 i，a[i,j]中为 true 的 j 一定是连续的一段。由此对每个 i，所有状态 a[i,j]可仅用两个端点表示，即 a[i].x1、a[i].x2。



考虑要使 a[i,j]=true，当且仅当 k 的取值范围与线段**[**a[i-1].x1**,**a[i-1].x2**]**有交集，即满足

P[1]-j ≥ a[i-1].x1 and **P[i-1]+P[i]-X-j ≤ a[i-1].x2**

[20] 准确地说应该是递推

即 **P[i-1]+P[i]-X-a[i-1].x2 ≤ j ≤ P[1]-a[i-1].x2**

由此，状态转移方程变为：

a[i].x1=max(0 , P[i-1]+P[i]-X-a[i-1].x2)
a[i].x2=min(P[i] , P[1]-a[i-1].x1)

初始状态 a[1].x1=a[1].x2=P[1]，状态总数 O(n)，状态转移 O(1)，可以说是高效的。

有了此方法，鉴于先前的分析，我们采用二分枚举 X，并利用高效的测试方法，即可在 O(n*logPmax)的时间解决整个问题。

[小结]

上面这个问题，看似难点仍旧在于动态规划，二分枚举只不过充当了一个附加手段。但实际上事先枚举的 X，极大地简便了动态规划的方程，才使得问题得以解决。应用这类二分枚举思想的最优性问题近来很是热门，而且这类试题很容易诱导选手直接采用动态规划或是贪心算法，而走入死胡同。

所以，今后我们在考虑最优性问题时，应注意问题是否隐含了一个有序的 01 序列，它是否可以用二分枚举的方法将最优性问题转化为可行性问题。切记！“退一步海阔天空”。

类型三：二分搜索——应用于无序序列

如果你认为只有在有序序列中才可以二分，那你就大错特错了，无序序列照样可以进行二分搜索。请看下面这个交互式问题——**推销员的旅行**（JSOI）

[问题描述]

作为一名推销员，Mirko 必须坐飞机访问 N 个城市一次仅一次。已知每两座城市间都有且仅有一条航线，总在整点起飞，途中花费 1 小时。每个航线的

飞机总是不停地往返于两座城市，即如果有一架飞机在 5 点整从 A 市飞往 B 市，则 6 点整到达，且马上又起飞，7 点整回到 A 市……为了保证效率，Mirko 想把 N 个城市排成一个序列 $A_1,A_2,\ldots,A_N$，对于每个 i（1<i<n），Mirko 可以在到达 A_i 市后，做一小时广告，然后立即从 A_i 市出发前往 A_{i+1} 市。

可惜 Mirko 没有飞机的时刻表，所以他不得不打电话问航空公司。每通电话，他可以询问 A、B 两市之间的航线在正午 12 点从 A 飞往 B 还是从 B 飞往 A。由于 Mirko 不想在打电话上花太多钱，请你帮助他，用尽可能少的电话确定一条旅行线路。注意，交互库可能根据你的询问调整线路使你打电话的次数最多。

[分析]

问题比较长，让我们先分析一下它到底要我们做什么。

假设我们在正午 12 点从 A_1 市飞往 A_2 市，1 点整到达，又做了一小时广告，2 点整从 A_2 市出发前往 A_3 市……显然，我们总是在偶数整点时出发从一个城市前往另一个城市。

因为飞机往返时间恰为 2 小时，中间又没有停顿，所以如果正午 12 点的飞机是从 A 飞往 B，则任意偶数整点的飞机也一定是从 A 飞往 B 的。

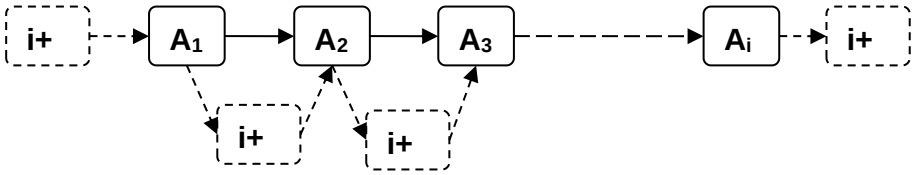
由此，我们可以将先前的问题转化（注意：是转化而不是等价）为在一个竞赛图[21]中寻找一条哈密尔顿路的问题，我们的目标就是尽可能少地进行询问。

为了避免询问的盲目性，我们尝试使用增量法逐步扩展序列。

[21] 即有 N 个顶点，两两之间恰只有一条边的有向图

我们先任意询问两座城市，方便起见就选择城市 1 和城市 2。不妨设 1 到 2 有边，我们就得到一条长度为 1 的线路 1→2。

假设我们已设计了一条含有前 i 座城市、长度为 i-1 的线路 A_1→A_2→…→A_i，我们希望再加入一座城市 i+1，变成长度为 i 的线路。



如图，我们可先询问 i+1 到 A_1 是否有边。如果有，我们就将线路改作 i+1→A_1→A_2→…→A_i；否则，我们再询问 A_i 到 i+1 是否有边，如果有，我们将线路改作 A_1→A_2→…→A_i→i+1。如果还没有，就表示两端不能插入，i+1 只能插在中间。

注意到 i+1 与 A_1 间的边是从 A_1 到 i+1。如果 i+1 有边到 A_2，我们就可以把 i+1 插在 A_1 与 A_2 间；如果 i+1 仍没有边到达 A_2，就表示 A_2 有边到 A_{i+1}，我们可再询问 A_3……

这样的询问会不会问到底也没法插入 i+1？不会，因为 i+1 有边到 A_i。

由此，我们得到了一种询问方法，但最坏情况下总的询问次数可能是 $O(N^2)$ 的。

由于**对每个点 A_k(1≤k≤i)，要么 A_k 有边到 i+1，要么 i+1 有边到 A_k**。且 A_1 有边到 i+1，i+1 有边到 A_i。于是，我们可以用**二分搜索**寻找 A_k，使 i+1 可以插入 A_k 与 A_{k+1} 之间。

假设我们已知 i+1 可以插入 A_p 与 A_r 之间，我们询问 i+1 与 $A_{(p+r)/2}$ 间边的方式。

如果 i+1 有边到 $A_{(p+r)/2}$，则表示 A_p 到 $A_{(p+r)/2}$ 间可以插入 i+1；否则表示 $A_{(p+r)/2}$ 到 A_r 间可以插入 i+1。不断二分，直至 p+1=r。

这样，我们所需要的询问次数仅为 O(nlogn)。

[小结]

对于这类二分搜索问题，其实从根本上讲我们是在一个无序的 01 序列中进行查找，查找的对象是一个特殊的子串‘01’。有了这个子串，再利用构造法，就可以将结果转化为原先问题的一组可行解了。

当我们询问 i+1 到 $A_{(p+r)/2}$ 之间的边，如果 i+1 有边到 $A_{(p+r)/2}$，我们就设法将 i+1 插入 A_p 与 $A_{(p+r)/2}$ 之间。其实，这并不表示 i+1 就一定不能插入到 $A_{(p+r)/2}$ 与 A_r 之间，只是不一定能。而 i+1 一定能插入到 A_p 与 $A_{(p+r)/2}$ 之间。由此可见，这样的二分搜索方法往往应用于那些可行解很多，但需要高效地构造一组可行解的问题。

有了上述思想，我们再看一个例子——**非与门电路**（ACM/ICPC CERC 2001）

[问题描述]

有一种门电路叫做非与门（NAND），每个 NAND 有两个输入端，输出为两个输入端非与运算的结果，即 not(A and B)。给出一个由 N 个 NAND 组成的无环电路（这一点对于一个逻辑电路来说很重要），电路的 M 个输入全部连接到一个相同的输入 x，如图 NAND-1 所示。请把其中一些输入设置为常数，用最少的 x 完成相同功能。图 NAND-2 是一个只用一个 x 输入但可以得到同样结果的电路。

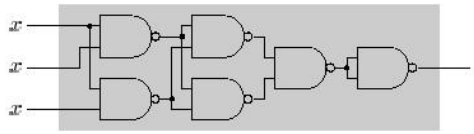


图 NAND-1

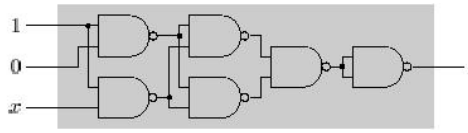


图 NAND-2

[分析]

首先想到的是使用表达式，将各个门电路的输出表示成输入的某个函数运算结果，以此连接整个电路的输入和输出。但由于 NAND 运算不满足结合律，也就无法进行表达式的运算。我们只得另辟蹊径。

事实上，我们很容易算出 x 分别为 0 和 1 时电路的输出结果 ans0、ans1。如果 ans0=ans1，显然我们一个 x 都不需要，将所有输入均设为 0 或 1 即可满足要求；如果 ans0≠ans1 呢？一个都不用是不可能的了，那么只用一个呢？

考虑到，即使只有一个 x，我们能够选择的输入序列仍非常多，只要有一个满足要求即可。因此，我们尽可大胆猜想，最多只要一个未知输入 x，即可满足要求。

只用一个 x，这表示当这个 x=0 时，电路输出恰好为 ans0；当这个 x=1 时，电路输出恰好为 ans1。我们只改变了一个输入端口的值，而且是从 0 改为 1。

由此，我们假想一个全为 0 的输入序列，经过若干次这样的修改后，最终将变为一个全为 1 的输入序列；另一方面，开始时电路输出 ans0，而最后电路

输出 ans1。所以，必定存在某个时刻，当一个输入由 0 变为 1 时，输出恰从 ans0 变为 ans1。

因此这就证明了我们刚才的最多只用一个 x 的猜想是正确的。同时，算法也浮出水面：

假想我们构造了如下 M+1 个不同的输入：

0:　0 0 0 0 0 0 … 0
1:　1 0 0 0 0 0 … 0
2:　1 1 0 0 0 0 … 0
…　…
M:　1 1 1 1 1 1 … 1

要求我们找出相邻两组输入，前一个输出 ans0，后一个输出 ans1。

我们仍旧可以由**二分搜索**高效完成：已知第 p 组输出 ans0、第 r 组输出 ans1，我们计算第(p+r)/2 组输出 ans’。如果 ans’=ans0，取 r=(p+r)/2；否则 p=(p+r)/2，直至 p+1=r。

于是问题在 O(NlogM)时间被很好地解决。

[小结]

可以看出，这类二分搜索问题的难点在于构造。如何将一个完全不相干的问题引入二分搜索的应用范围，利用一个不起眼的 0 到 1 的改变构造出原问题的一个解决方案。构造法没有统一的格式或者规律可循，只能具体问题具体分析。我们所能做的，就是在平时多接触、多思考、多积累。

【总结】

本文这里仅简单地介绍了几个例子，要涵盖二分策略的所有应用甚至是大部分应用都是困难的。我想指出的是二分思想虽然简单，但是它的内容还是非

常丰富的。我们不能让二分思想仅仅停留在一般有序数组上的最基本的二分查找，而应该扩展到更广泛的应用上。

二分策略常与其它一些算法相结合，并借以隐蔽自己，以逃离我们的视线。这就要求我们能有扎实的基本功、丰富的解题经验、大胆合理的猜测以及活跃的创造思维，方可“以不变应万变”。

【参考文献】

《实用算法的分析与程序设计》吴文虎 王建德

《算法艺术与信息学竞赛》刘汝佳 黄亮

让算法的效率“跳起来”！

—— 浅谈“跳跃表”的相关操作及其应用

上海市华东师范大学第二附属中学 魏冉

【目录】

【关键字】

跳跃表 高效 概率 随机化

【摘要】

本文分为三大部分。

首先是概述部分。它会从功能、效率等方面对跳跃表作一个初步的介绍，并给出其图形结构，以便读者对跳跃表有个形象的认识。

第二部分将介绍跳跃表的三种基本操作——查找，插入和删除，并对它们的时空复杂度进行分析。

第三部分是对跳跃表应用的介绍，并通过实际测试效果来对跳跃表以及其它一些相关数据结构进行对比，体现其各自的优缺点。

最后一部分是对跳跃表数据结构的总结。

【概述及结构】

二叉树是我们都非常熟悉的一种数据结构。它支持包括查找、插入、删除等一系列的操作。但它有一个致命的弱点，就是当数据的随机性不够时，会导致其树型结构的不平衡，从而直接影响到算法的效率。

跳跃表（Skip List）是 1987 年才诞生的一种崭新的数据结构，它在进行查找、插入、删除等操作时的期望时间复杂度均为 O(logn),有着近乎替代平衡树

的本领。而且最重要的一点，就是它的编程复杂度较同类的 AVL 树，红黑树等要低得多，这使得其无论是在理解还是在推广性上，都有着十分明显的优势。

首先，我们来看一下跳跃表的结构（如图 1）

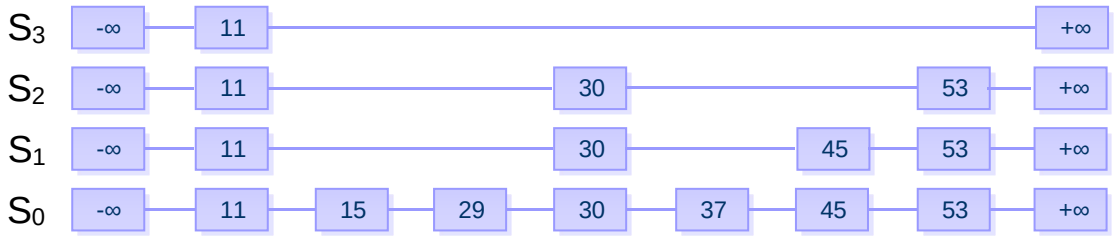


图 1　有 7 个元素的跳跃表

跳跃表由多条链构成（S_0，S_1，S_2 ……，S_h），且满足如下三个条件：

（1）　每条链必须包含两个特殊元素：+∞ 和 -∞

（2）　S_0 包含所有的元素，并且所有链中的元素按照升序排列。

（3）　每条链中的元素集合必须包含于序数较小的链的元素集合，即：

$$S_0 \supseteq S_1 \supseteq S_2 \supseteq \ldots \supseteq S_h$$

【基本操作】

在对跳跃表有一个初步的认识以后，我们来看一下基于它的几个最基本的操作。

一、 查找

目的：在跳跃表中查找一个元素 x

在跳跃表中查找一个元素 x，按照如下几个步骤进行：

i) 从最上层的链（S_h）的开头开始

ii) 假设当前位置为 p，它向右指向的节点为 q（p 与 q 不一定相邻），且 q 的值为 y。将 y 与 x 作比较

(1) x=y 输出***查询成功***及相关信息

(2) x>y 从 p 向右移动到 q 的位置

(3) x<y 从 p 向下移动一格

iii) 如果当前位置在最底层的链中（S_0），且还要往下移动的话，则输出***查询失败***

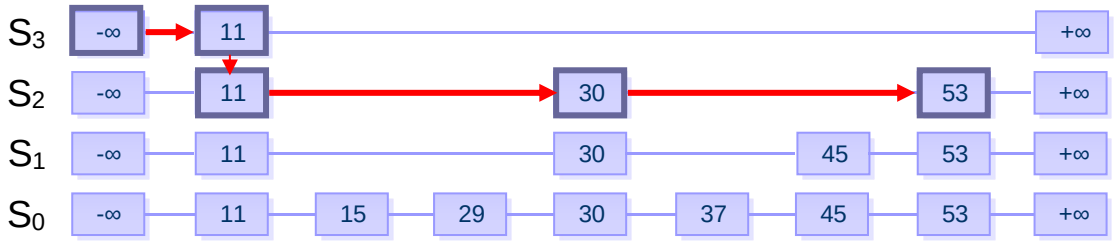


图 2 查询元素 53 的全过程

二、插入

目的：向跳跃表中插入一个元素 x

首先明确，向跳跃表中插入一个元素，相当于在表中插入一列从 S_0 中某一位置出发向上的连续一段元素。有两个参数需要确定，即插入列的位置以及它的“高度”。

关于插入的位置，我们先利用跳跃表的查找功能，找到比 x 小的最大的数 y。根据跳跃表中所有链均是递增序列的原则，x 必然就插在 y 的后面。

而插入列的“高度”较前者来说显得更加重要，也更加难以确定。由于它的不确定性，使得不同的决策可能会导致截然不同的算法效率。为了使插入数据之后，保持该数据结构进行各种操作均为 O(logn)复杂度的性质，我们引入**随机化算法**（Randomized Algorithms）。

我们定义一个**随机决策模块**，它的大致内容如下：

·产生一个 0 到 1 的随机数 r　　r ← random()

·如果 r 小于一个常数 p，则执行方案 A，　　if r<p then do A

否则，执行方案 B

else do B

初始时列高为 1。插入元素时，不停地执行随机决策模块。如果要求执行的是 A 操作，则将列的高度加 1，并且继续反复执行随机决策模块。直到第 i 次，模块要求执行的是 B 操作，我们结束决策，并向跳跃表中插入一个高度为 i 的列。

性质 1： 根据上述决策方法，该列的高度大于等于 k 的概率为 p^{k-1}。

此处有一个地方需要注意，如果得到的 i 比当前跳跃表的高度 h 还要大的话，则需要增加新的链，使得跳跃表仍满足先前所提到的条件。

我们来看一个例子：

假设当前我们要插入元素“40”，且在执行了随机决策模块后得到高度为 4

·步骤一：找到表中比 40 小的最大的数，确定插入位置

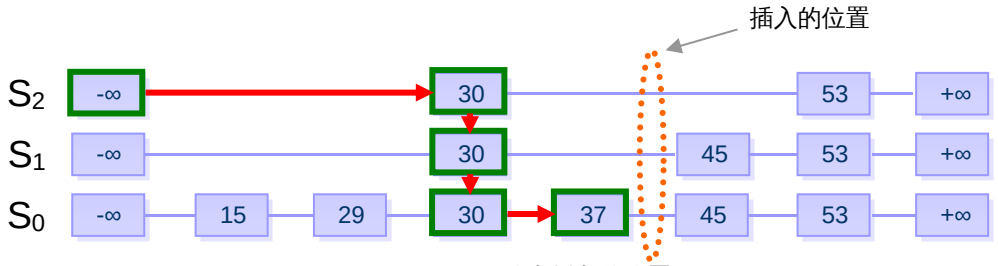


图 3.1 确定插入的位置

·步骤二：插入高度为 4 的列，并维护跳跃表的结构

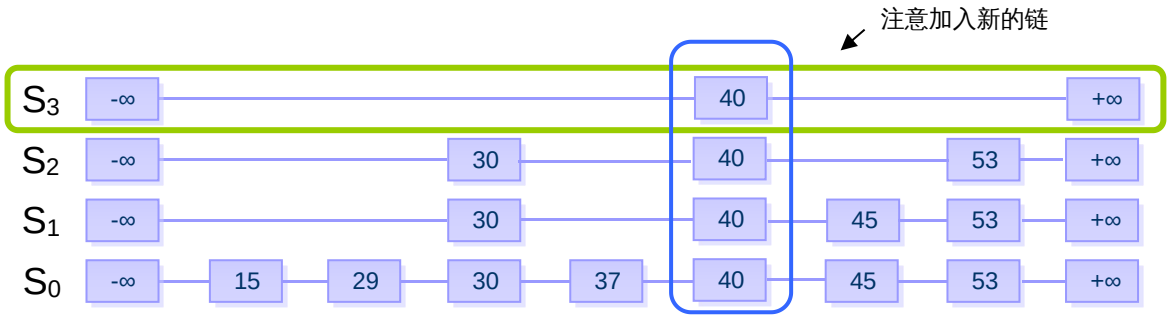


图 3.2 插入高度为 4 的列，并维护跳跃表

三、删除

目的：从跳跃表中删除一个元素 x

删除操作分为以下三个步骤：

（1） 在跳跃表中查找到这个元素的位置，如果未找到，则退出

*

（2） 将该元素所在整列从表中删除

*

（3） 将多余的“空链”删除

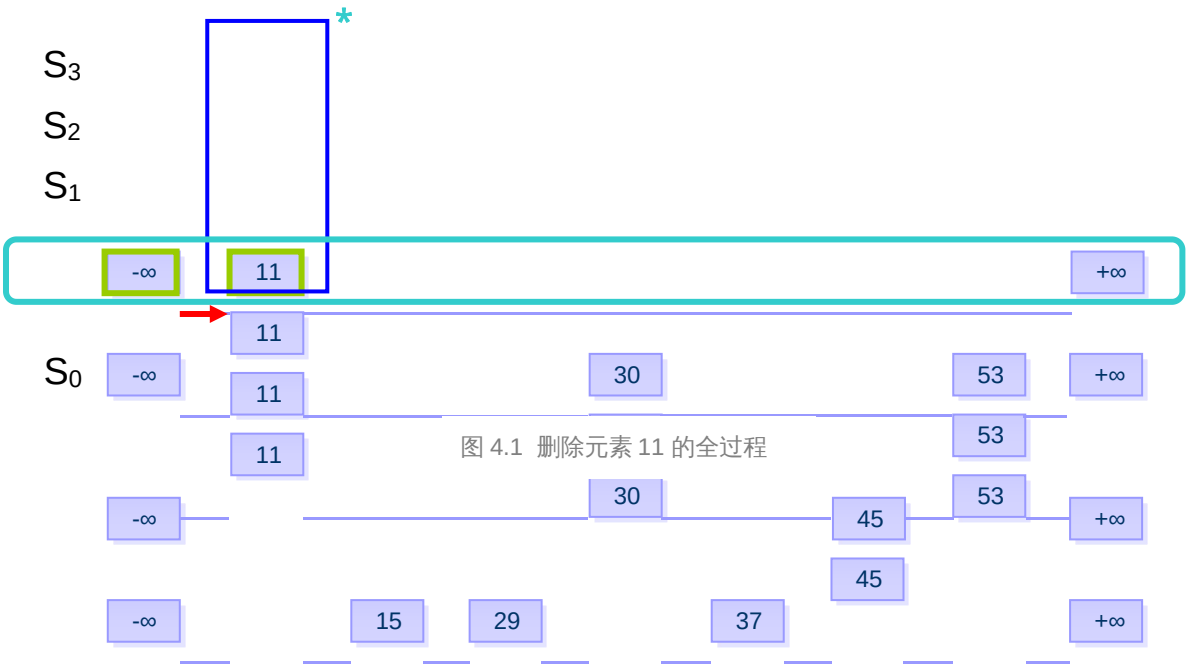


图 4.1 删除元素 11 的全过程

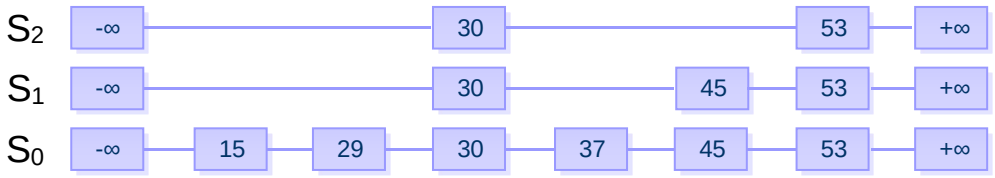


图 4.2 删除以后的结构

四、“记忆化”查找（Search with fingers）

所谓“记忆化”查找，就是在前一次查找的基础上进行进一步的查找。它可以利用前一次查找所得到的信息，取其中可以被当前查找所利用的部分。利用“记忆化”查找可以将一次查找的复杂度变为 O(logk)，**其中 k 为此次与前一次两个被查找元素在跳跃表中位置的距离。**

下面来看一下记忆化搜索的具体实现方法：

假设上一次操作我们查询的元素为 i，此次操作我们欲查询的元素为 j。我们用一个 update 数组来记录在查找 i 时，指针在每一层所“跳”到的最右边的位置。如图 4.1 中橘黄色的元素。（蓝色为路径上的其它元素）

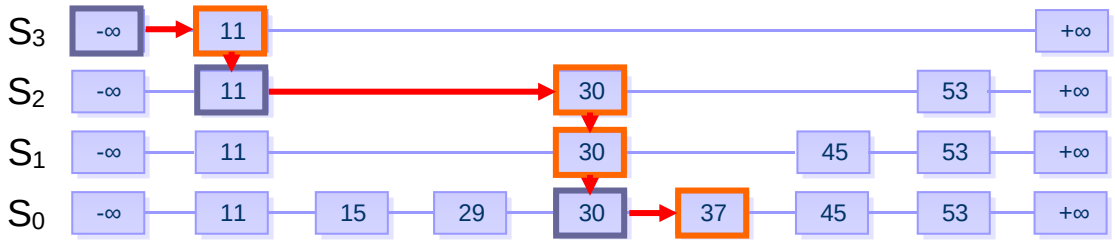


图 4.1 查找元素 37

在插入元素 j 时，分为两种情况：

（1） i<=j

从 S_0 层开始向上遍历 update 数组中的元素，直到找到某个元素，它向右指向的元素大于等于 j，并于此处开始新一轮对 j 的查找（与一般的查找过程相同）

（2） i>j

从 S_0 层开始向上遍历 update 数组中的元素，直到找到某个元素小于等于 j，并于此处开始新一轮对 j 的查找（与一般的查找过程相同）

图 4.2 十分详细地说明了在查找了 i=37 之后，继续查找 j=15 或 53 时的两种不同情况。

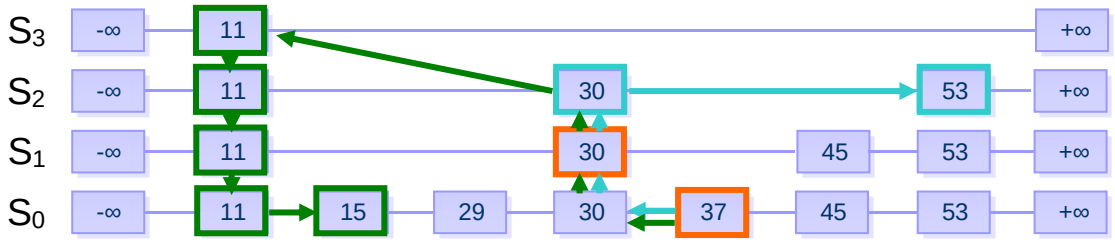


图 4.2 新一轮查找元素为 **15**（**53**）的步骤流程

记忆化查找（Search with fingers）技术对于那些前后相关性较强的数据效率极高，这点可以在后文中的实际测试报告中略见一斑。

【复杂度分析】

一个数据结构的好坏大部分取决于它自身的空间复杂度以及基于它一系列操作的时间复杂度。跳跃表之所以被誉为几乎能够代替平衡树，其复杂度方面自然不会落后。我们来看一下跳跃表的相关复杂度：

空间复杂度： O(n)　　（期望）

跳跃表高度： O(logn)　　（期望）

相关操作的时间复杂度：

查找：　　O(logn)　　（期望）

插入：　　O(logn)　　（期望）

删除：　　O(logn)　　（期望）

之所以在每一项后面都加一个“期望”，是因为跳跃表的复杂度分析是基于概率论的。有可能会产生最坏情况，不过这种概率极其微小。

下面我们来一项一项分析。

一、 空间复杂度分析 O(n)

假设一共有 n 个元素。根据**性质 1**，每个元素插入到第 i 层（S_i）的概率为 p^{i-1}，则在第 i 层插入的期望元素个数为 np^{i-1}，跳跃表的元素期望个数为 $\sum_{i=0}^{h-1} np^i$，当 p 取小于 0.5 的数时，次数总和小于 2n。

所以总的空间复杂度为 O(n)

二、跳跃表高度分析 O(logn)

根据**性质 1**，每个元素插入到第 i 层（S_i）的概率为 p^i，则在第 i 层插入的期望元素个数为 np^{i-1}。

考虑一个特殊的层：第 $1+3\log_{1/p} n$ 层。

$S_{1+3\log_{1/p} n}$ 层的元素期望个数为 $np^{3\log_{1/p} n} = 1/n^2$，当 n 取较大数时，这个式子的值接近 0，故跳跃表的高度为 O(logn)级别的。

三、查找的时间复杂度分析 O(logn)

我们采用逆向分析的方法。假设我们现在在目标节点，想要走到跳跃表最左上方的开始节点。这条路径的长度，即可理解为查找的时间复杂度。

设当前在第 i 层第 j 列那个节点上。

i) 如果第 j 列恰好只有 i 层（对应插入这个元素时第 i 次调用随机化模块时所产生的 B 决策，概率为 1-p），则当前这个位置必然是从左方的某个节点向右跳过来的。

ii) 如果第 j 列的层数大于 i（对应插入这个元素时第 i 次调用随机化模块时所产生的 A 决策，概率为 p），则当前这个位置必然是从上方跳下来的。（不可能从左方来，否则在以前就已经跳到当前节点上方的节点了，不会跳到当前节点左方的节点）

设 C(k)为向上跳 k 层的期望步数（包括横向跳跃）

有：

C(0) = 0

C(k) = (1-p)(1+向左跳跃之后的步数)+p(1+向上跳跃之后的步数)

= (1-p)(1+C(k)) + p(1+C(k-1))

C(k) = 1/p + C(k-1)

C(k) = k/p

而跳跃表的高度又是 logn 级别的，故查找的复杂度也为 logn 级别。

对于记忆化查找（Search with fingers）技术我们可以采用类似的方法分析，很容易得出它的复杂度是 O(logk)的（其中 k 为此次与前一次两个被查找元素在跳跃表中位置的距离）。

四、插入与删除的时间复杂度分析 O(logn)

插入和删除都由查找和更新两部分构成。查找的时间复杂度为 O(logn)，更新部分的复杂度又与跳跃表的高度成正比，即也为 O(logn)。

所以，插入和删除操作的时间复杂度都为 O(logn)

五、实际测试效果

（1）不同的 p 对算法复杂度的影响

P	平均操作时间	平均列高	总结点数	每次查找跳跃次数（平均值）	每次插入跳跃次数（平均值）	每次删除跳跃次数（平均值）
2/3	0.0024690 ms	3.004	91233	39.878	41.604	41.566
1/2	0.0020180 ms	1.995	60683	27.807	29.947	29.072
1/e	0.0019870 ms	1.584	47570	27.332	28.238	28.452
1/4	0.0021720 ms	1.330	40478	28.726	29.472	29.664
1/8	0.0026880 ms	1.144	34420	35.147	35.821	36.007

表 1　进行 10^6 次随机操作后的统计结果

从表 1 中可见，当 p 取 1/2 和 1/e 的时候，时间效率比较高（为什么？）。而如果在实际应用中空间要求很严格的话，那就可以考虑取稍小一些的 p，如 1/4。

（2）运用“记忆化”查找 (Search with fingers) 的效果分析

所谓“记忆化”查找，就是在前一次查找的基础上进行进一步的查找。它可以利用前一次查找所得到的信息，取其中可以被当前查找所利用的部分。利用“记忆化”查找可以将一次查找的复杂度变为 O(logk)，其中 k 为此次与前一次两个被查找元素在跳跃表中位置的距离。

P	数据类型	平均操作时间（不运用记忆化查找）	平均操作时间（运用记忆化查找）	平均每次查找跳跃次数（不运用记忆化查找）	平均每次查找跳跃次数（运用记忆化查找）
0.5	随机（相邻被查找元素键值差的绝对值较大）	0.0020150 ms	0.0020790 ms	23.262	26.509
0.5	前后具备相关性（相邻被查找元素键值差的绝对值较小）	0.0008440 ms	**0.0006880 ms**	26.157	**4.932**

表 1　进行 10^6 次相关操作后的统计结果

从表 2 中可见，当数据相邻被查找元素键值差绝对值较小的时候，我们运用“记忆化”查找的优势是很明显的，不过当数据随机化程度比较高的时候，“记忆化”查找不但不能提高效率，反而会因为跳跃次数过多而成为算法的瓶颈。

合理地利用此项优化，可以在特定的情况下将算法效率提升一个层次。

【跳跃表的应用】

高效率的相关操作和较低的编程复杂度使得跳跃表在实际应用中的范围十分广泛。尤其在那些编程时间特别紧张的情况下，高性价比的跳跃表很可能会成为你的得力助手。

能运用到跳跃表的地方很多，与其去翻陈年老题，不如来个趁热打铁，拿NOI2004 第一试的第一题——郁闷的出纳员(Cashier)来“小试牛刀”吧。

例题一：NOI2004 Day1 郁闷的出纳员(Cashier)

[点击查看附录中的原题]

这道题解法的多样性给了我们一次对比的机会。用不同的算法和数据结构，在效率上会有怎样的差异呢？

首先定义几个变量

R – 工资的范围

N – 员工总数

我们来看一下每一种适用的算法和数据结构的简要描述和理论复杂度：

（1） 线段树

简要描述：以工资为关键字构造线段树，并完成相关操作。

I 命令时间复杂度：O(logR)

A 命令时间复杂度：O(1)

S 命令时间复杂度：O(logR)

F 命令时间复杂度：O(logR)

（2） 伸展树(Splay tree)

简要描述：以工资为关键字构造伸展树，并通过“旋转”完成相关操作。

I 命令时间复杂度：O(logN)

A 命令时间复杂度：O(1)

S 命令时间复杂度：O(logN)

F 命令时间复杂度：O(logN)

（3） 跳跃表(Skip List)

简要描述：运用跳跃表数据结构完成相关操作。

I 命令时间复杂度：O(logN)

A 命令时间复杂度：O(1)

S 命令时间复杂度：O(logN)

F 命令时间复杂度：O(logN)

实际效果评测： （单位：秒）

	Test1	Test2	Test3	Test4	Test5	Test6	Test7	Test8	Test9	Test10
线段树	0.000	0.000	0.000	0.031	0.062	0.094	0.109	0.203	0.265	0.250
伸展树	0.000	0.000	0.016	0.062	0.047	0.125	0.141	0.360	0.453	0.422
跳跃表	0.000	0.000	0.000	0.047	0.062	0.109	0.156	0.368	0.438	0.375

从结果来看，线段树这种经典的数据结构似乎占据着很大的优势。可有一点万万不能忽略，那就是线段树是基于键值构造的，它受到键值范围的约束。在本题中 R 的范围只有 10^5 级别，这在内存较宽裕的情况下还是可以接受的。但是如果问题要求的键值范围较大，或者根本就不是整数时，线段树可就很难适应了。这时候我们就不得不考虑伸展树、跳跃表这类基于元素构造的数据结构。而从实际测试结果看，跳跃表的效率并不比伸展树差。加上编程复杂度上的优势，跳跃表尽显出其简单高效的特点。

参考程序：


cashier_skiplist.txt

例题二：HNOI2004 Day1 宠物收养所(pet)

[点击查看附录中的原题]

此题与《郁闷的出纳员》最大的不同，就在于它的键值范围达到了 2^{31} 级别。这对线段树来说可是一大考验。虽然采取边做边开空间的策略勉强可以缓解内存的压力，但此题对内存的要求很苛刻，元素相对范围来说也比较少，如果插入的元素稍微分散一些，就很有可能使得空间复杂度接近 O（NlogN）！何况如果稍微拓展一下，**插入的元素不是整数而是实数呢？**

而这道题对于跳跃表来说,可真是再适合不过了。几乎对标准的算法不需要做修改，如果熟练的话，从思考到编写完成也就 20 分钟左右的时间，最终的算法效率也很高。更加重要的一点，跳跃表绝不会因键值类型的变化而失效，推广性很强。

参考程序：


pet.txt

【总结】

跳跃表作为一种新兴的数据结构，以相当高的效率和较低的复杂度散发着其独特的光芒。和同样以编程复杂度低而闻名的“伸展树”相比，跳跃表的效率不但不会比它差，甚至优于前者（见附表 1）。

人们在思考一类问题的时候，往往会无意中被局限在一个小范围当中。就拿和平衡树相关的问题来说，人们凭借自己的智慧，创造出了红黑树，AVL 树等一些很复杂的数据结构。可是千变万变，却一直走不出“树”这个范围。过高的编程复杂度使得这些成果很难被人们所接受。而跳跃表的出现，使得人们眼前顿时豁然开朗。原来用与树完全不相关的数据结构也能够实现树的功能！

“跳跃表”这个名字有着其深远的意义。不仅是因为它形象地描述了自身的结构，更有一点，它象征着一种思考方法，一种“跳出定式”的思考方法。在你面临一个困难却山穷水复疑无路的时候，不妨找到问题的原点，“跳”出思维的定式，说不定在另一条全新的路上，你将会看到胜利的曙光。

【参考文献】

[1] William Pugh. Skip Lists: A Probabilistic Alternative to Balanced Trees

[2] William Pugh. A Skip List Cookbook

【附录】

·附表：跳跃表与 AVL 树、2-3 树、伸展树在时间效率上的对比（摘自 *[1]* *William Pugh.* *Skip Lists: A Probabilistic Alternative to Balanced Trees*）

Implementation	Search Time	Insertion Time	Deletion Time
Skip lists	0.051 msec (1.0)	0.065 msec (1.0)	0.059 msec (1.0)
non-recursive AVL trees	0.046 msec (0.91)	0.10 msec (1.55)	0.085 msec (1.46)
recursive 2–3 trees	0.054 msec (1.05)	0.21 msec (3.2)	0.21 msec (3.65)
Self–adjusting trees:			
top-down splaying	0.15 msec (3.0)	0.16 msec (2.5)	0.18 msec (3.1)
bottom-up splaying	0.49 msec (9.6)	0.51 msec (7.8)	0.53 msec (9.0)

Table 2 - Timings of implementations of different algorithms

·程序："跳跃表"的程序(Pascal 语言实现)：

ski pl i st . t xt

附解：为什么 p=1/e 的时候时间效率最高？

解：由复杂度分析中得出，跳跃表的时间效率取决于跳跃的次数，也就是 k/p（k 为跳跃表高度），而 k 是 $\log_{1/p} n$ 级别。故有：

$$f(x) = k/p = \frac{\log_{1/p} n}{p} = \ln n \frac{1/p}{\ln 1/p}$$

令 x = 1/p ，则有：

$$f(x) = \ln n \frac{x}{\ln x}$$

对 $g(x) = \frac{x}{\ln x}$ 求导，有 $g'(x) = \frac{\ln x - 1}{(\ln x)^2}$

当 x=e 时，g'(x)=0，即 f(x)到达极值点。

此时 p = 1/e

[返回"实际测试效果部分"]

附题：NOI2004 Day1 郁闷的出纳员

[返回“跳跃表的应用”部分]

郁闷的出纳员

【问题描述】

OIER 公司是一家大型专业化软件公司，有着数以万计的员工。作为一名出纳员，我的任务之一便是统计每位员工的工资。这本来是一份不错的工作，但是令人郁闷的是，我们的老板反复无常，经常调整员工的工资。如果他心情好，就可能把每位员工的工资加上一个相同的量。反之，如果心情不好，就可能把他们的工资扣除一个相同的量。我真不知道除了调工资他还做什么其它事情。

工资的频繁调整很让员工反感，尤其是集体扣除工资的时候，一旦某位员工发现自己的工资已经低于了合同规定的工资下界，他就会立刻气愤地离开公司，并且再也不会回来了。每位员工的工资下界都是统一规定的。每当一个人离开公司，我就要从电脑中把他的工资档案删去，同样，每当公司招聘了一位新员工，我就得为他新建一个工资档案。

老板经常到我这边来询问工资情况，他并不问具体某位员工的工资情况，而是问现在工资第 k 多的员工拿多少工资。每当这时，我就不得不对数万个员工进行一次漫长的排序，然后告诉他答案。

好了，现在你已经对我的工作了解不少了。正如你猜的那样，我想请你编一个工资统计程序。怎么样，不是很困难吧？

【输入文件】

第一行有两个非负整数 n 和 min。n 表示下面有多少条命令，min 表示工资下界。

接下来的 n 行，每行表示一条命令。命令可以是以下四种之一：

名称	格式	作用
I 命令	I_k	新建一个工资档案，初始工资为 k。如果某员工的初始工资低于工资下界，他将立刻离开公司。
A 命令	A_k	把每位员工的工资加上 k
S 命令	S_k	把每位员工的工资扣除 k
F 命令	F_k	查询第 k 多的工资

_（下划线）表示一个空格，I 命令、A 命令、S 命令中的 k 是一个非负整数，F 命令中的 k 是一个正整数。

在初始时，可以认为公司里一个员工也没有。

【输出文件】

输出文件的行数为 F 命令的条数加一。

对于每条 F 命令，你的程序要输出一行，仅包含一个整数，为当前工资第 k 多的员工所拿的工资数，如果 k 大于目前员工的数目，则输出-1。

输出文件的最后一行包含一个整数，为离开公司的员工的总数。

【样例输入】

9 10

I 60

I 70

S 50

F 2

I 30

S 15

A 5

F 1

F 2

【样例输出】

10

20

-1

2

【约定】

I 命令的条数不超过 100000

A 命令和 S 命令的总条数不超过 100

F 命令的条数不超过 100000

每次工资调整的调整量不超过 1000

新员工的工资不超过 100000

【评分方法】

对于每个测试点，如果你输出文件的行数不正确，或者输出文件中含有非法字符，得分为 0。

否则你的得分按如下方法计算：如果对于所有的 F 命令，你都输出了正确的答案，并且最后输出的离开公司的人数也是正确的，你将得到 10 分；如果你只对所有的 F 命令输出了正确答案，得 6 分；如果只有离开公司的人数是正确的，得 4 分；否则得 0 分。

附题：HNOI2004 Day1 宠物收养所

[返回“跳跃表的应用”部分]

宠物收养所

(pet)

【背景描述】

最近，阿 Q 开了一间宠物收养所。收养所提供两种服务：收养被主人遗弃的宠物和让新的主人领养这些宠物。

每个领养者都希望领养到自己满意的宠物，阿 Q 根据领养者的要求通过他自己发明的一个特殊的公式，得出该领养者希望领养的宠物的特点值 a（a 是

一个正整数，a<2^31），而他也给每个处在收养所的宠物一个特点值。这样他就能够很方便的处理整个领养宠物的过程了，宠物收养所总是会有两种情况发生：被遗弃的宠物过多或者是想要收养宠物的人太多，而宠物太少。

1． 被遗弃的宠物过多时，假若到来一个领养者，这个领养者希望领养的宠物的特点值为 a，那么它将会领养一只目前未被领养的宠物中特点值最接近 a 的一只宠物。（任何两只宠物的特点值都不可能是相同的，任何两个领养者的希望领养宠物的特点值也不可能是一样的）如果有两只满足要求的宠物，即存在两只宠物他们的特点值分别为 a-b 和 a+b，那么领养者将会领养特点值为 a-b 的那只宠物。

2． 收养宠物的人过多，假若到来一只被收养的宠物，那么哪个领养者能够领养它呢？能够领养它的领养者，是那个希望被领养宠物的特点值最接近该宠物特点值的领养者，如果该宠物的特点值为 a，存在两个领养者他们希望领养宠物的特点值分别为 a-b 和 a+b，那么特点值为 a-b 的那个领养者将成功领养该宠物。

一个领养者领养了一个特点值为 a 的宠物，而它本身希望领养的宠物的特点值为 b，那么这个领养者的不满意程度为 abs(a-b)。

【任务描述】

你得到了一年当中，领养者和被收养宠物到来收养所的情况，希望你计算所有收养了宠物的领养者的不满意程度的总和。这一年初始时，收养所里面既没有宠物，也没有领养者。

【输入格式】：(input.txt)

你将从文件 input.txt 当中读入数据。文件的第一行为一个正整数 n，n<=80000，表示一年当中来到收养所的宠物和领养者的总数。接下来的 n 行，按到来时间的先后顺序描述了一年当中来到收养所的宠物和领养者的情况。每行有两个正整数 a, b，其中 a=0 表示宠物，a=1 表示领养者，b 表示宠物的特点值或是领养者希望领养宠物的特点值。（同一时间呆在收养所中的，要么全是宠物，要么全是领养者，这些宠物和领养者的个数不会超过 10000 个）

【输入样例】

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

【输出格式】：(cut.out)

输出文件 output.txt 中仅有一个正整数，表示一年当中所有收养了宠物的领养者的不满意程度的总和 mod 1000000 以后的结果。

【输出样例】

```
3
```

(abs(3-2) + abs(2-4)=3，最后一个领养者没有宠物可以领养)

【运行限制】

运行时限：1 秒钟

【评分方法】

本题目一共有十个测试点，每个测试点的分数为总分数的 10%。对于每个测试点来说，如果你给出的答案正确，那么你将得到该测试点全部的分数，否则得 0 分。

浅析二分图匹配在信息学竞赛中的应用

湖南省长沙市长郡

中学 王俊

[摘要]

本文通过对几道信息学竞赛题目的分析，举例说明了二分图匹配在信息学竞赛中的应用。二分图匹配的应用一般是通过分析某些最优化问题的性质，构造出二分图，再通过求得该二分图的最大匹配，最佳匹配等各种形式的匹配从而解决原问题。

[关键字]

匹配 二分图 最小权 最大权 优化

[正文]

一 引 言

二分图匹配是信息学竞赛中一类经典的图论算法，在近年来信息学竞赛中有广泛应用。如果可以以某一种方式将题目中的对象分成两个互补的集合，而需要求得它们之间满足某种条件的一一对应的关系时，往往可以抽象出对象以及对象之间的关系构造二分图，然后利用匹配算法来解决。这类题目通常需要考察选手对原题进行建模，构造二分图，设计匹配算法，并对其算法进行适当优化等多方面能力。下面就通过两道例题来说明二分图匹配在信息学竞赛中的一些应用。

二 Railway Communication[22]

2.1 问题描述

某国有 *n* 个城镇，*m* 条单向铁路。每条铁路都连接着两个不同的城镇，且该铁路系统中不存在环。现需要确定一些列车运行线，使其满足：

I) 每条铁路最多属于一条列车运行线；

II) 每个城镇最多被一条列车运行线通过（通过包括作为起点或终点）；

III) 每个城镇至少被一条列车运行线通过；

IV) 列车运行线的数量应尽量小。

V) 在满足以上条件下列车运行线的长度和应该尽量小。

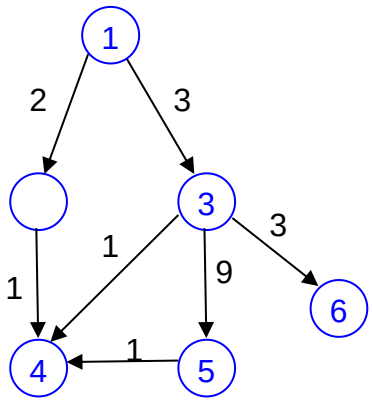


图 1

2.2 问题分析

题目要求列车运行线数最少，又要求在此条件下列车运行线的长度和最小，不便于一起考虑，我们不妨分步研究，先考虑列车运行线数最少的子问题。则该子问题可建立如下数学模型：给定一个有向无环图 $G^0=(N^0, A^0)$，用尽量少的不相交的简单路径覆盖 N^0。

[22] Saratov State University 252. Railway Communication

我们可以给问题建立一个二分图 $G=(N, A)$，如图 2。

a) 建立两个互补的结点集合 X 和 Y，把点 $i(i \in N^0)$拆成 X 结点 i 和 Y 结点 i'。 $N = X \cup Y$ 。

b) 对于图 G^0 中有向边(i, j), $(i, j) \in A^0$, 则在 A 中加入边(i, j')。如果在 G^0 中选定(i, j)作为某条覆盖路径中的边，则在 G 中选定边(i, j')。

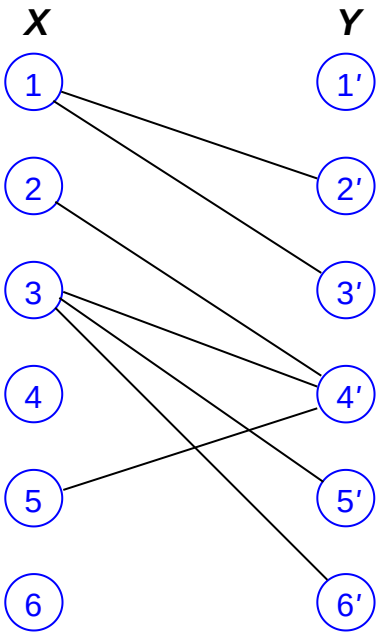


图 2

对于图 G^0 中的任意一个结点 i，可分为三类：

I) 某条覆盖路径的起点，即它没有前驱结点，那么在二分图 G 中点 i' 的邻边均没有选。

II) 某条覆盖路径内部的点，即它有一个前驱结点和一个后继结点，那么在二分图 G 中 i，i' 的邻边各选了 1 条。

III) 某条覆盖路径的终点，即它没有后继结点，那么在二分图 G 中点 i 的邻边均没有选。[23]

[23] 如果某条覆盖路径只有一个结点的话，它显然满足性质 I 和性质 III。

这样问题就转化成在二分图 *G* 中选一些边，且每个点的邻边中至多有一条被选中，显然这是一个二分图匹配的问题。又因为题目要求路径数最少，即路径终点数最少，即尽量多的匹配，所以是求该二分图的最大匹配，可以套用经典的匈牙利算法求解。

再来考虑求列车运行线总长度最小的问题。设原图 G^0 中边(*i*，*j*)的边权为 $W_{i,j}^0$ ，则给图 *G* 的边(*i*，*j*')加入边权 $W_{i,j}$ ，$W_{i,j}=W_{i,j}^0$ (如图 3)。原问题是求图 G^0 中在保证覆盖路径数最少时求覆盖路径总长度最小，即在二分图 *G* 中求保证匹配数最大时匹配边的权值和最小。显然就是求图 *G* 的最小权最大匹配， 由于经典的 *KM* 算法是求最大权最大匹配，那么我们再对图 *G* 进行一定修改，使得 $W_{i,j}=w-W_{i,j}^0 \quad \left((i,j)\in A\right)$，且如果 $(i,j)\notin A$，则添加边(*i*，*j*)，$W_{i,j}=0$。其中 *w* 可以取一个比较大的正整数，但需要满足 $w>\max\left\{W_{i,j}^0\right\}\times n \quad (i,j)\in A^0$。这样用经典的 *KM* 算法求出二分图 *G* 的最大权最大匹配，即可轻易转化得到最小权最大匹配，从而解决原问题。

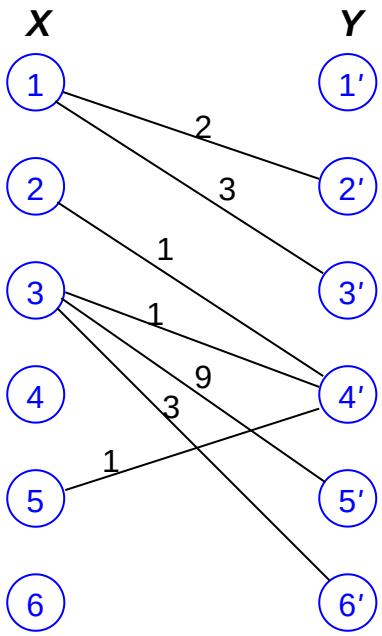


图 2

注：为了使图更简单清晰，省略了边权为无穷大的边。

2.3 小结

这道题目的数学模型很容易建立，就是最小路径覆盖问题的扩展。在分析该问题的时候抓住每个点在一条覆盖路径中至多有一个前驱一个后继这个条件，可以联系到匹配中每个点也至多和一个点匹配，于是顺利转化成匹配的问题。

一一对应是匹配重要的性质。

三 Roads[24]

3.1 问题描述

[24] Saratov State University 206. Roads

一个遥远的王国有 m 条道路连接着 n 个城市。m 条道路中有 n-1 条石头路，m-n+1 条泥土路,任意两个城市之间有且仅有一条完全用石头路连接起来的道路。

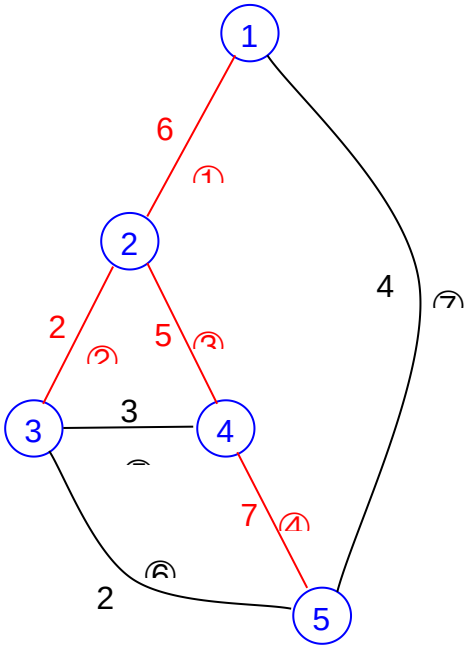
每条道路都有一个唯一确定的编号，其中石头路编号为 1..n-1，泥土路编号为 n..m。每条道路都需要一定的维护费，其中第 i 条道路每年需要 C_i 的费用来维护。最近该国国王准备只维护部分道路以节省费用。但是他还是希望人们可以在任两个城市间互达。

国王需要你提交维护每条道路的费用，以便他能让他的大臣来挑选出需要维护的道路，使得维护这些道路的费用是最少的。

尽管国王不知道走石头路和走泥土路的区别，但是对于人民来说石头路似乎是更好的选择。为了人民的利益，你希望维护的道路是石头路。这样你不得不在提交给国王的报告中伪造维护费用。你需要给道路 i 伪造一个费用 D_i，使得这些石头路能够被挑选。为了不让国王发现，你需要使得真实值与伪造值的差值和 $f=\sum_{i=1}^{m}|D_i-C_i|$ 尽量小。

国王的大臣当然不是白痴，全部由石头路组成的方案如果是一种花费最小的方案，那么他会选择这种方案。

求出真实值与伪造值的差值和的最小值，以及得到该最小值的方案，即每条边的修改后的边权 D_i。



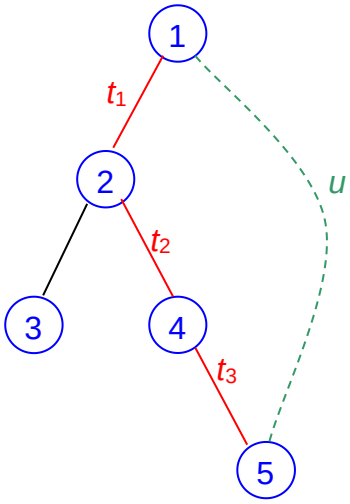
红色边表示树 *T* 中的边

图 4

3.2 问题分析

设原图为 $G^0=(N^0, A^0)$。其中 N^0 表示顶点集合，A^0 表示边集合，即 $N^0=\{1, 2, \ldots\ldots, n\}$，$A^0=\{a_1, a_2, \ldots\ldots, a_m\}$。用 C_i 和 D_i 表示原始及修改后边 a_i 的边权。

由于任意两点都有且仅有一条道路完全是石头路，所以 n-1 条石头路必定是图 G^0 中的一棵生成树，我们设为树 T。而题目则是要求对图中的某些边权进行修改，对于边 a_i，将边权由 C_i 修改成 D_i，使得树 T 成为图中的一棵最小生成树，且 $f=\sum_{i=1}^{m}|D_i-C_i|$ 最小。



实线表示树边

图 5

根据与树 *T* 的关系，我们可以把图 G^0 中的边分为树边和非树边两类。我们先通过对最小生成树性质的研究来挖掘 *D* 所需满足的条件。设 *P*[*e*]表示边 *e* 的两个端点之间树的路径中边的集合。如图 5 中，$P[u]=\{t_1,t_2,t_3\}$，即 $u\notin T$ ，而 $t_1,t_2,t_3\in T$ ，且 *u* 与 t_1, t_2, t_3 构成一个环，所以用非树边 *u* 替换树边 t_1，t_2，t_3 中任意一条都可以得到一棵新的生成树。而如果 *u* 的边权比所替换的边的边权更小的话，则可以得到一棵权值更小的生成树。那么只有满足条件 $D_{t_1}\le D_u, D_{t_2}\le D_u, D_{t_3}\le D_u$ 才能使得原生成树 *T* 是一棵最小生成树。

那么推广到一般情况，如果对于边 *v*，*u*，其中 $v\in P[u], u\notin T$ ，则必须满足 $D_v\le D_u$ ，否则用边 *u* 替换边 *v* 后能得到一棵新的权值更小的生成树 $T-v+u$ 。

我们得到了 *D* 的限制条件，而问题需要求得 $\sum_{i=1}^{m}|D_i-C_i|$ 最小，其中绝对值符号的存在是一个拦路虎。根据以上的分析，要使树 *T* 是一棵最小生成树，得到的不等式 $D_v\le D_u$ 中 *v* 总为树边而 *u* 总为非树边。也就是树边的边权应该尽量小，而非树边的边权则应该尽量大。

设边权的修改量为 Δ ，即 $\Delta_e=|D_e-C_e|$

I) 当 $e\in T$ ，则应该将边权 C_e 减去一个非负的值，即 $\Delta_e=C_e-D_e$

II) 当 $e \notin T$ ，则应该将边权 C_e 加上一个非负的值，即 $\Delta_e = D_e - C_e$

这样成功去掉了绝对值符号，只要求得 Δ 的值，那么 D 值就可以唯一确定，而问题也由求 D 转化成求 Δ。

那么任意满足条件 v，$u(v \in P[u], u \notin T)$的不等式 $D_v \le D_u$ 等价于 $C_v - \Delta_v \le C_u + \Delta_u$ ，即 $\Delta_v + \Delta_u \ge C_v - C_u$。那问题就是求出所有的 $\Delta_i (i \in [1,m])$ 使其满足这个不等式组且 $f = \sum_{i=1}^{m} \Delta_i$ 最小。

由于不等式 $\Delta_v + \Delta_u \ge C_v - C_u$ 右边 $C_v - C_u$ 是一个已知量，你或许会发现这个不等式似曾相识，这就是我们在求二分图的最佳匹配算法时用到的 *KM* 算法中不可或缺的一个不等式。*KM* 算法中，首先给二分图的每个点都设一个可行顶标，*X* 结点 *i* 为 l_i，*Y* 结点 *j* 为 r_i。从初始时可行顶标的设定到中间可行顶标的修改直至最后算法结束，对于边权为 $W_{v,u}$ 的边(*v*，*u*)始终需要满足 $l_v + r_u \ge W_{v,u}$。

于是我们可以构造一个带权的二分图 *G*=(*N*，*A*)，用 *W* 表示边权，如图 6。

a) 构造两个互补的结点结合 *X*，*Y*。把 $a_i(a_i \in T)$作为 X_1 结点 *i*，$a_j(a_j \notin T)$ 作为 Y_1 结点 *j*。

b) 如果图 G^0 中 $a_i \in P[a_j], a_j \notin T$ ，且 $C_i - C_j > 0$ ，则在 X_1 结点 *i* 与 Y_1 结点 *j* 之间添加边(*i*，*j*)，$W_{i,j} = C_i - C_j$ 。

c) 如果 $|X_1| < |Y_1|$ ，则添加 $|Y_1| - |X_1|$ 个 X_2 结点，$Y_2 = \varnothing$ ；如果 $|Y_1| < |X_1|$ ，则添加 $|X_1| - |Y_1|$ 个 Y_2 结点，$X_2 = \varnothing$。

$X = X_1 \cup X_2, Y = Y_1 \cup Y_2, N = X \cup Y$ 。

d) 如果 $i \in X, j \in Y$ 且 $(i, j) \notin A$ ，则添加边(*i*，*j*)，且 $W_{i,j} = 0$

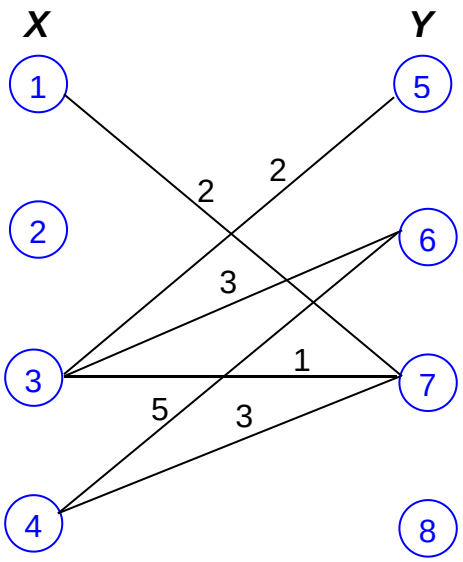


图 6

注：为了使图更简单清晰，省略了边权为 0 的边。

这样图 *G* 是一个二分完全图，设 *Q* 为图 *G* 的一个完备匹配，*X* 结点 *i* 的可行顶标为 l_i，*Y* 结点 *j* 的可行顶标为 r_j，则：

$$l_i + r_j \geq W_{i,j} \quad ((i,j) \in Q)$$

设 *M* 为图 *G* 的最大权匹配，显然 *M* 也是完备匹配，则满足

$$l_i + r_j = W_{i,j} \quad ((i,j) \in M)$$

设完备匹配 *Q* 的匹配边的权值和为 S_Q，显然

$$S_M = \sum_{(i,j)\in M} W_{i,j} = \sum_{i\in X} l_i + \sum_{j\in Y} r_j \quad 。$$

显然此时可行顶标之和取到最小值。

若 $i \in X_2$，且 $(i,j) \in M$，

由 $W_{v,u} = 0 \quad ((v,u) \in A, v \in X_2)$ 和 $M \subseteq A$

可知 $W_{i,j} = 0$，所以 $l_i + r_j = W_{i,j} = 0$

又因为 $l_i \geq 0 \quad (i \in X)$，$X_2 \subseteq X$

所以 $l_i = 0 \quad (i \in X_2)$，同理 $r_j = 0 \quad (j \in Y_2)$

所以 $S_M = \sum_{i\in X} l_i + \sum_{j\in Y} r_j = \sum_{i\in X_1} l_i + \sum_{j\in Y_1} r_j = \sum_{i=1}^{m} \Delta_i = f$

显然 S_M 即是满足 T 是图 G^0 的一棵最小生成树的最小代价，而此时的 D 值则是修改后的一种可行方案，那么问题就转化为求图 G 的最大权完备 M。

至此，问题已得到解决，我们来分析一下该算法的复杂度。预处理的时间复杂度为 $O(|E|)$，而 KM 算法的时间复杂度为 $O(|M||E|)$，所以总的时间复杂度为 $O(|M||E|)$。由于 KM 算法是在完备匹配基础上的，所以 $|V| = 2\max\{n-1, m-n+1\}$，$|M| = \frac{|V|}{2} = O(m)$，又由于图 G 是二分完全图，所以 $|E| = |M|^2 = O(m^2)$，所以总的时间复杂度为 $O(m^3)$。空间复杂度为 $O(m^2)$。

3.3 扩展思考

如果题目只要求出最少的修改量，而不需要求出修改方案，那么是否有更好的算法呢？

3.3.1 算法 1

现在需要求得的是原问题的一个子问题，利用 KM 算法来求出二分图 G 的最大权最大匹配，而 $f_{\min}$ 即为所求，显然可以很好的解决该问题，其时间复杂度为 $O(m^3)$。

3.3.2 算法 2

可以发现，在构造二分图 G 时，其中 *c*)，*d*)两个步骤中构造的都是一些虚结点和虚边，完全是为了符合 KM 算法要求完备匹配的条件，没有太多实际的意义。而利用 KM 算法解此题最大的优势就在于能求出修改方案，而如果题目不要求修改方案，则毫无意义，因此可试探不添加这些虚结点和虚边。

因为 $f=\sum_{i=1}^{m}\Delta_i=\sum_{i\in X}l_i+\sum_{j\in Y}r_j$ ，且 $l_i+r_j=W_{i,j}\quad((i,j)\in M)$ ，所以 $f=\sum_{(i,j)\in M}W_{i,j}$ ，即最小修改量就是最大权最大匹配的匹配边的权值和，所以问题只需求出最大权最大匹配的值。

构造有向图 $G^f=(N^f,A^f)$，W^f 表示边权，U^f 表示容量，R^f 表示流量，如图 7。

a) 构造两个互补的结点集合 X^f，Y^f。把 $a_i(a_i\in T)$作为 X^f 结点 i，$a_j(a_j\notin T)$ 作为 Y^f 结点 j。

b) 如果图 G^0 中 $a_i\in P[a_j],a_j\notin T$ ，且 $C_i-C_j>0$。则在 X^f 结点 i 与 Y^f 结点 j 之间加入有向边(i，j)，$W^f_{i,j}=C_i-C_j$，$U^f_{i,j}=1$ 。

c) 构造源点 s 和汇点 t，$N^f=X^f\cup Y^f\cup\{s,t\}$

d) 添加有向边(s，i) $(i\in X^f)$，$W^f_{s,i}=0$，$U^f_{s,i}=1$；添加有向边(j，t) $(j\in Y^f)$，$W^f_{j,t}=0$，$U^f_{j,t}=1$。

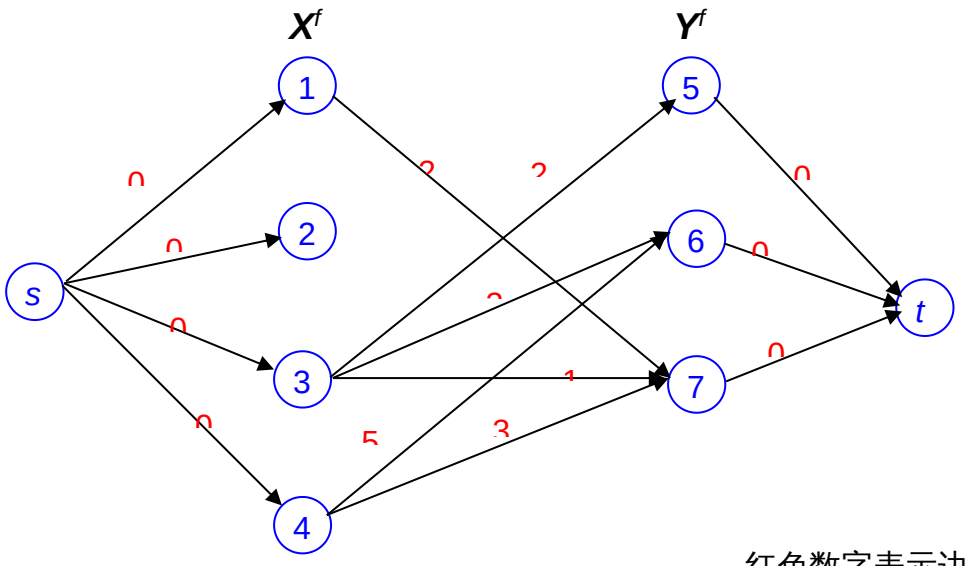


图 7

红色数字表示边的费

[引理一] 设流 *F* 的费用和为 $Cost_F$。图 *G* 上的任意一个完备匹配 *M*，都能在图 G^f 上找到可行流 *F* 与其对应，且 $S_M = Cost_F$。而对于图 G^f 上的任意可行流 *F*，在图 *G* 上的也都能找到一组以 *M* 为代表的完备匹配与其对应，且 $Cost_F = S_M$。

[证明] 对于图 *G* 中任意一条匹配边 $(i, j) \in M$ 且 $W_{i,j} > 0$，都可以找到图 G^f 中一条容量为 1 的流 $s \to i \to j \to t$，其中 $i \in X^f, j \in Y^f$。因为 $W_{s,i}^f = 0,\ \ W_{j,t}^f = 0,\ \ W_{i,j} = W_{i,j}^f$，所以 $W_{i,j} = W_{s,i}^f + W_{i,j}^f + W_{j,t}^f$；而当图 *G* 中 $(i, j) \in M$ 且 $W_{i,j} = 0$，则对 S_M 的值不构成影响。所以当流 *F* 为匹配 *M* 所对应的流的并时，$S_M = \sum_{(i,j)\in M} W_{i,j} = \sum_{(i,j)\in F} W_{i,j}^f = Cost_F$。而反过来对于任意可行流 *F*，同样可以找到完备匹配 *M* 与其对应且 $Cost_F = S_M$。

设图 G^f 的最大费用最大流为 *F*，显然图 G^f 的最大费用最大流则和图 *G* 的最大权最大匹配对应，此时最大费用就等于匹配的权值和，即 $Cost_F = \sum_{(i,j)\in F} W_{i,j}^f \times R_{i,j}^f = \sum_{(i,j)\in M} W_{i,j}$。这样问题就转化为求图 G^f 的最大费用最大流。

如果用 *bellman_ford* 求最大费用路的算法来求最大费用最大流，则复杂度为 $O(|V||E|s)$,其中 *s* 表示流量。$|V| = O(m),\ \ |E| = O(nm),\ \ s = O(n)$，所以复杂度为 $O(n^2m^2)$。

3.3.3 算法 3

由于 $m = O(n^2)$，所以算法二的时间复杂度 $O(n^2m^2)$ 和算法一的 $O(m^3)$ 是一个级别的。观察可发现，用 *bellman_ford* 求最大费用路的复杂度为 $O(|V||E|)$，成为算法 2 的瓶颈，如何减少求最大费用路的代价成为优化算法的关键，但由于残量网络中有负权边，导致类似 *Dijkstra* 等算法没有用武之地。这里介绍一种高效求单源最短路的 *SPFA*（*Shortest Path Faster Algorithm*）算法。

设 *L* 记录从源点到其余各点当前的最短路径值。*SPFA* 算法采用邻接表存储图，方法是动态优先逼近法。算法中设立一个先进先出的队列用来保存待优化的顶点，优化时每次取出队首结点 *p*，并且用 *p* 点的当前路径值 L_p 去优化调整其他顶点的最优路径值 L_j，若有调整，即 L_j 变小了，且 *j* 点不在当前的队列中，就将 *j* 点放入队尾以待进一步优化。就这样反复从队列取出点来优化其他点路径值，直至队列空不需要再优化为止。

由于每次优化都是将某个点 *j* 的最优路径值 L_j 变小，所以算法的执行会使 *L* 越来越小，所以其优化过程是正确的。只要图中不存在负环，则每个顶点都必定有一个最优路径值，所以随着 *L* 值逐渐变小，直到其达到最优路径值时，算法结束，所以算法是收敛的，不会形成无限循环。这样，我们简要地证明了该算法的正确性。

算法中每次取出队首结点 *p*，并访问点 *p* 的所有邻结点的复杂度为 $O(d)$，其中 *d* 为点 *p* 的出度。对于 *n* 个点 *e* 条边的图，结点的平均出度为 $\frac{e}{n}$，所以每处理一个点的复杂度为 $O\left(\frac{e}{n}\right)$。设顶点入队的次数为 *h*，显然 *h* 随图的不同而不同，但它仅与边的权值分布有关，设 $h = kn$，则算法 *SPFA* 的时间复杂度为：$T = O\left(h \times \frac{e}{n}\right) = O\left(\frac{h}{n} \times e\right) = O(ke)$。经过实际运行效果得到，*k* 在一般情况下是比较小的常数（当然也有特殊情况使得 *k* 值较大），所以 *SPFA* 算法在普通情况下的时间复杂度为 $O(e)$。

而此题中需要求的最大费用路显然可将 *SPFA* 算法稍加修改得到。这样我们得到了一个时间复杂度为 $O(|E|s)$，即 $O(n^2 m)$ 的算法。

3.3.4 算法 4

刚才用网络流来求匹配以提高效率，但未对匹配的本质进行深究，现在再回到匹配上来，继续挖掘匹配的性质。前面用 *KM* 算法解此题的时候是构造了一个边上带权的二分图，其实不妨换一种思路，将权值由边上转移到点上，或许会有新的发现。

构造二分图 *G* '=(*N* '，*A* ')，如图 8。

a) 构造两个互补的结点集合 *X* '和 *Y* '，把 a_i($a_i \in T$)作为 *X* '结点 *i*，a_j($a_j \in A'$)作为 *Y*'结点 *j*。

b) 在 *X* '结点 *i* 和 *Y* '结点 *i* 之间添加边(*i*，*i*)。

c) 如果图 G^0 中 $a_i \in P[a_j], a_j \notin T$ ，且 $C_i - C_j > 0$ 。则在 *X* '结点 *i* 与 *Y* '结点 *j* 之间加入有向边(*i*，*j*)。

d) 给 *Y* '结点 *i* 一个权值 C_i，即如果某点被匹配则得到其权值。

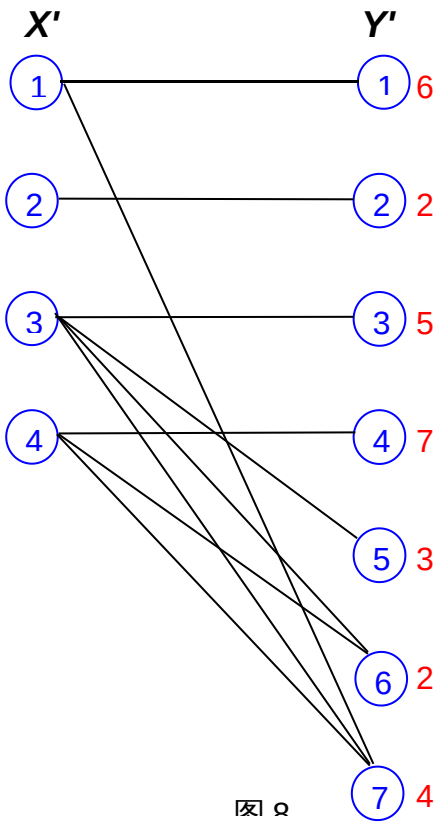


图 8

[引理二] 设 $\mu=\sum_{a_i\in T}C_i$。对于图 *G* 中一个完备匹配 *M*，都可以在图 *G* '中找到一个完备匹配 *M* '与其对应，且 $S_M=\mu-S_{M'}$。而对于图 *G* '的任意一个完备匹配 *M* '，也可以在图 *G* 中找到一组以 *M* 为代表的完备匹配与其对应，且 $S_M=\mu-S_{M'}$。

[证明] 设 Y_1' 表示存在 $i(i\neq j)$ 使得 $(i,j)\in M'$ 的点 *j* 的集合；设 Y_2' 表示存在 i$(i=j)$ 使得 $(i,j)\in M'$ 的点 *j* 的集合。

对于图 *G* 中一条匹配边(*i*，*j*) $(i,j)\in M$ 且 $i\in X_2$，则 $W_{i,j}$=0，对 S_M 的值没有影响。

对于图 *G* 中一条匹配边(*i*，*j*) $(i,j)\in M$ 且 $i\in X_1$，

若 $W_{i,j}$=0，则在对应图 *G*'中可找到一条边 $(i,j)\in M'$ $(i\in X', j\in Y_2'$ 且 $i=j)$ 与其对应；

若 $W_{i,j}>0$，则在对应图 G'中可找到一条边$(i,j)\in M'\ (i\in X',j\in Y_1')$与其对应。

所以

$$\begin{aligned}S_M &= \sum_{(i,j)\in M} W_{i,j}\\ &= \sum_{(i,j)\in M,W_{i,j}>0} W_{i,j}\\ &= \sum_{(i,j)\in M,W_{i,j}>0} C_i - C_j\\ &= \left(\sum_{a_i\in T} C_i - \sum_{j\in Y_2'} C_j\right) - \sum_{j\in Y_1'} C_j\\ &= \sum_{a_i\in T} C_i - \left(\sum_{j\in Y_2'} C_j + \sum_{j\in Y_1'} C_j\right)\\ &= \mu - S_{M'}\end{aligned}$$

同理，图 G'上的匹配 M'也可以在图 G 上找到对应的匹配 M 且 $S_M=\mu-S_{M'}$。

因为 $S_M+S_{M'}=\mu$ 为定值。显然，当 S_M 取到最大值时，$S_{M'}$取到最小值。又因为 M 和 M'均为完备匹配，所以图 G 的最大权最大匹配就对应了图 G'的最小权完备匹配。那么问题转化为求图 G'的最小权完备匹配。

由于图 G' 中的权值都集中在 Y' 结点上，所以 $S_{M'}$只与 Y' 结点中哪些点被匹配有关。那么可以使 Y' 结点中权值小的结点尽量被匹配。算法也渐渐明了，将 Y' 结点按照权值大小非降序排列，然后从前往后一个个找匹配。

用 R 来记录可匹配点，如果 X'结点 $i\in R$ ，则 i 未匹配，或者从某个未匹配的 X'结点有一条可增广路径到达点 i，其路径用 $Path[i]$来表示。设 B_j 表示 j 的邻结点集合，每次查询 Y'结点 j 能否找到匹配时，只需看是否存在点 i，$i\in B_j$且$i\in R$。而每次找到匹配后马上更新 R 和 $Path$。下面给出算法的流程：

Begin

将 *Y'*结点按照权值非降序排列；

$M' \leftarrow 0$ ；

计算 *R* 及 *Path*,并标记点 $i(i \in R)$ 为可匹配点；

For $j \leftarrow 1$ *to m do*

Begin

$q \leftarrow$第 *j* 个 *Y'*结点；

If 存在 *q* 的某个邻结点 *p* 为可匹配点 *then*

Begin

将匹配边(*p*，*q*)加入匹配 *M'*；

更新 *R* 以及 *Path*，并且重新标记点 $i(i \in R)$ 为可匹配点；

End；

End；

M' 是一个最小权最大匹配；

End；

下面来分析一下该算法的复杂度。算法中执行了如下操作：

1） 将所有 *Y'*结点按权值非降序排列；

2） 询问是否存在 *q* 的某个邻结点 *p* 为可匹配点；

3） 更新 *M'*；

4） 更新 *R* 以及 *Path*；

5） 标记点 $i(i \in R)$ 为可匹配点；

操作 1 的时间复杂度为 $O(m\log_2 m) = O(n^2 \log_2 n)$。设 $d_{\max} = \max\{|B_j|\} \quad j \in [1,m]$，操作 2 单次执行的复杂度为 $O(|B_j|)$，最多执行 *m*

次，所以复杂度为$O(md_{\max})=O(n^2d_{\max})$。操作 3 单次执行的复杂度为$O(1)$，最多执行 *n*-1 次，所以复杂度为$O(n)$。操作 5 单次执行的复杂度为$O(n)$，最多执行 *n*-1 次，所以复杂度为$O(n^2)$。

接下来讨论操作 4 的复杂度。我们知道，如果某个点在某次更新中是不可匹配点，那么以后无论怎么更新它都不可能变成可匹配点。又如果某个点为可匹配点，则它的路径必然为$i_0 \to j_1 \to i_1 \to j_2 \to i_2 \to \cdots \to j_k \to i_k \quad (k \ge 0)$，其中 i_0 为未匹配点而且$(j_t,i_t)\ (t\in[1,k])$是当前的匹配边，所以 *Y'*结点中未匹配点是不可能出现在某个 *X'*点 *i* 的 *Path*[*i*]中的。也就是说我们在更新 *R* 和 *Path* 的时候只需要在 *X'*结点中原来的可匹配点以及 *Y'*结点中已匹配点和它们之间的边构成的一个子二分图中进行，显然任意时刻图 *G'*的匹配边数都是不超过 *n*-1 的，所以该子图的点数是$O(n)$的，边数是$O(nd_{\max})$，显然单次执行操作 4 的复杂度即为$O(nd_{\max})$，最多执行 *n* 次，所以其复杂度为$O(n^2d_{\max})$。

算法总的时间复杂度为$O(n^2\log_2 n)+O(n^2d_{\max})+O(n)+O(n^2d_{\max})+O(n^2)$
$=O(n^2(d_{\max}+\log_2 n))$，因为 d_{max} 是$O(n)$级别的，所以该算法的时间复杂度为$O(n^3)$，其空间复杂度为$O(nm)$。

其实本题还有更优秀的算法，但其推导与证明相对比较复杂，这里就不详细介绍了，大家可以参考相关论文。

3.4 小结

该题是一道最优化的问题，尝试发现动态规划，贪心等算法都无从下手。但经过一步步的分析，思路渐渐清晰，在得到了若干重要不等式后，问题豁然开朗，是一道求最佳匹配的问题，可以用经典的 *KM* 算法求解。

而在对不求方案的问题的研究中，不局限于直观的原图，而是从各个方向各个角度入手，构造不同的图，以展现题目各个方面的性质。也将算法复杂度由$O(m^3)$，优化到$O(n^2m)$，再到$O(n^3)$，使效率大大提高。

下表是对上文研究的几个算法的简单比较：

	时间复杂度	空间复杂度	算法核心
算法一	$O(m^3)$	$O(m^2)$	*KM* 算法求最大权最大匹配
算法二	$O(n^2m^2)$	$O(nm)$	用 *bellman_ford* 算法求网络最大费用最大流增广路
算法三	$O(n^2m)$	$O(nm)$	用 *SPFA* 算法求网络最大费用最大流增广路
算法四	$O(n^3)$	$O(nm)$	求结点带权的二分图的最小权匹配

四 总结

通过对以上的例题的分析可见，在信息学竞赛中，二分图匹配算法的应用往往不是显而易见的，而是需要挖掘出问题的本质，从而构造合适的二分图并用匹配算法来求解。而在求匹配的时候往往也不是简单的套用经典算法，而是需要充分利用题目的特有性质，将经典匹配算法加以变形，从而得到更高效的算法。

信息学竞赛中的各种题目，往往都需要通过对题目的仔细**观察**，构造出合适的数学模型，然后通过对题目以及模型的进一步**分析**，挖掘出问题的本质，进行大胆的**猜想**，转化模型，设计合适的算法解决问题。

[感谢]

衷心感谢向期中老师在学习上对我的指导和帮助

衷心感谢任恺、胡伟栋和周源同学对我的大力帮助

衷心感谢肖湘宁和周戈林两位同学对我的论文提出宝贵的意见

[参考文献]

[1] ***Ravindra K. Ahuja & James B. Orlin ，A Faster Algorithm for the Inverse Spanning Tree Problem***

[2] 段凡丁，关于最短路径的*SPFA*快速算法

[附录]

例一原题：

Saratov State University 252. Railway Communication

time limit per test: 1 sec.

memory limit per test: 65536 KB

input: standard

output: standard

There are N towns in the country, connected with M railroads. All railroads are one-way, the railroad system is organized in such a way that there are no cycles. It's necessary to choose the best trains schedule, taking into account some facts.

Train path is the sequence of towns passed by the train. The following conditions must be satisfied.

1) At most one train path can pass along each railroad.

2) At most one train path can pass through each town, because no town can cope with a large amount of transport.

3) At least one train path must pass through each town, or town economics falls into stagnation.

4) The number of paths must be minimal possible.

Moreover, every railroad requires some money for support, i-th railroad requires c[i] coins per year to keep it intact. That is why the president of the country decided to choose such schedule that the sum of costs of maintaining the railroads used in it is minimal possible. Of course, you are to find such schedule.

Input

The first line of input contains two integers N and M (1<=N<=100; 0<=M<=1000). Next M lines describe railroads. Each line contains three integer numbers a[i], b[i] and c[i] - the towns that the railroad connects

(1<=a[i]<=N; 1<=b[i]<=N, a[i]<>b[i]) and the cost of maintaining it (0<=c[i]<=1000). Since the road is one-way, the trains are only allowed to move along it from town a[i] to town b[i]. Any two towns are connected by at most one railroad.

Output

On the first line output K - the number of paths in the best schedule and C - the sum of costs of maintaining the railroads in the best schedule. After that output K lines, for each train path first output L[i] (1<=L[i]<=N) - the number of towns this train path uses, and then L[i] integers identifying the towns on the train path. If there are several optimal solutions output any of them.

Sample test(s)

Input

```
4 4
1 2 1
1 3 2
3 4 2
2 4 2
```

Output

```
2 3
2 1 2
2 3 4
```

例二原题：

Saratov State University 206. Roads

time limit per test: 2 sec.

memory limit per test: 65536 KB

input: standard

output: standard

The kingdom of Farland has N cities connected by M roads. Some roads are paved with stones, others are just country roads. Since paving the road is quite expensive, the roads to be paved were chosen in such a way that for any two cities there is exactly one way to get from one city to another passing only the stoned roads.

The kingdom has a very strong bureaucracy so each road has its own ordinal number ranging from 1 to M: the stoned roads have numbers from 1 to N-1 and other roads have numbers from N to M. Each road requires some money for support, i-th road requires c_i coins per year to keep it intact. Recently the king has decided to save some money and keep financing only some roads. Since he wants his people to be able to

get from any city to any other, he decided to keep supporting some roads in such a way, that there is still a path between any two cities.

It might seem to you that keeping the stoned roads would be the good idea, however the king did not think so. Since he did not like to travel, he did not know the difference between traveling by a stoned road and travelling by a muddy road. Thus he ordered you to bring him the costs of maintaining the roads so that he could order his wizard to choose the roads to keep in such a way that the total cost of maintaining them would be minimal.

Being the minister of communications of Farland, you want to help your people to keep the stoned roads. To do this you want to fake the costs of maintaining the roads in your report to the king. That is, you want to provide for each road the fake cost of its maintaining d_i in such a way, that stoned roads form the set of roads the king would keep. However, to lower the chance of being caught, you want the value of sum(i = 1..M, $|c_i-d_i|$) to be as small as possible.

You know that the king's wizard is not a complete fool, so if there is the way to choose the minimal set of roads to be the set of the stoned roads, he would do it, so ties are allowed.

Input

The first line of the input file contains N and M ($2 \le N \le 60$, $N-1 \le M \le 400$). Next M lines contain three integer numbers a_i, b_i and c_i each — the numbers of the cities the road connects ($1 \le a_i \le N$, $1 \le b_i \le N$, $a_i \ne b_i$) and the cost of maintaining it ($1 \le c_i \le 10\,000$).

Output

Output M lines — for each road output d_i that should be reported to be its maintainance cost so that he king would choose first N-1 roads to be the roads to keep and the specified sum is minimal possible.

Sample test(s)

Input

```
4 5
4 1 7
2 1 5
3 4 4
4 2 5
1 3 1
```

Output

```
4
5
4
```

5
4

参数搜索的应用

摘要

参数搜索法是解最优解问题中的常见的方法，它的应用十分广泛。本文通过几个例子说明了其在实际问题中的应用，并分析了它的优缺点。

关键字

参数搜索 上界 二分

正文：

一．引言

参数搜索是解决最优解问题一种很常见的方法。其本质就是对问题加入参数，先解决有参数的问题，再不断调整参数，最终求得最优解，下面就例举出它在几个不同方面的应用。

二．应用

先来看一个例子，分石子问题：*有N 个石子,每个石子重量 Q_i；按顺序将它们装进K 个筐中；求一种方案，使得最重的筐最轻。*

分析：本题乍一看很容易想到动态规划。事实上的确可以用动态规划解决，稍加分析我们很快得到一个简单的算法。用状态 f(i,k)表示将前 i 个石子

装入 k 个筐最优方案，g(i,j)表示 i-j 中最重的石子，则可以写出状态转移方程：

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

边界条件：$g(i,i)=Q_i, f(1,1)=g(1,1)$

很明显，这个算法的时间复杂度为 $O(N^3)$,空间复杂度为 $O(N^2)$，并不十分理想。经过一些优化可以将复杂度降为 $O(N^2)$，不过这样思维复杂度骤然加大，且算法本身仍不够高效。

现在已经很难在原动态规划模型上做文章了，我们必须换一个思路。按一般的想法，顺序将石子装入筐，即先把石子放入第一筐，放到一定时候再改放第二个筐，第三个筐……但由于筐的重量没有上限，我们无法知道放到什么时候可以停止，转而将石子放入下一个筐。此时，问题的难点已经显露出来，是不是有方法可以化解呢？

我们不妨针对上面的难点，加入一个参数 P，改求一个判定可行解问题：**每个筐石子重量和不能超过 P，是否可以用 K 个筐装下所有石子。**

首先经过分析不难发现，如果当前筐的石子总重量为 T_1，即将处理的石子重量为 T_2，若 $T_1+T_2 \le P$，则我们仍将该石子放入当前框，这是显而易见的。由此可以得出贪心算法，按顺序把石子放进筐，若将石子放入当前筐后，筐的总重量不超过 P，则继续处理下一个石子；若重量和超过 P，则将该石子放入下一个筐，若此时筐的数目超过 K，则问题无解，否则处理完所有石子后就找到了一个可行解。

以上算法时间复杂度 O(N),空间复杂度 O(N)，这都是理论的下界。

现在我们已经解决了可行解问题，再回到原问题。是不是可以利用刚才的简化过的问题呢？答案是肯定的。一个最简单的想法是从小到大枚举 P，不断尝试找最优解为 P 的方案（这就是刚才的可行解问题），直到找到此方案。这就是题目的最优解。

估算一下上面算法的时间复杂度。令 T=∑Qi，则最坏情况下需要枚举 T 次才能找到解，而每次判断的时间复杂度为 O(N)，因此总的时间复杂度为 O(TN)，故需要做进一步优化。

下面考虑答案所在的区间。很明显，若我们可以找到一个总重量不超过 P’的解，则我们一定能找到一个总重量不超过 P’+1 的解，也就是说，可行答案必定可以位于区间[q,+∞]（其中 q 为本题最优解）。因此，我们自然而然的联想到了二分，具体方法为：在区间[1,T]内取中值 m，若可以找到不超过 m 的方案，则尝试区间[1,m-1];若不能，尝试区间[m+1,T]。不断重复以上步骤即可找到问题的最优解。

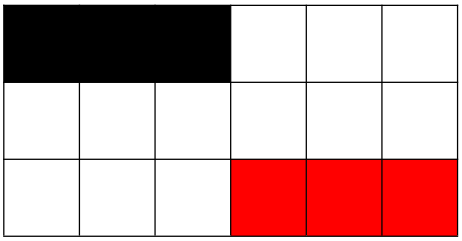
分析一下采用二分法后算法总的时间复杂度：由于每次除去一半的区间，则一共只需判断 $\log_2 N$ 次，而每次判断的时间复杂度为 O(N)，因此这个算法总的时间复杂度为 O(NlgT)。

一般而言，这个复杂度可以令人满意的，并且实际使用中效果非常好。但该复杂度同权值有关，不完全属于多项式算法，我们可以继续求得一个多项式算法（该算法与本文核心内容无关，只作简单探讨）。

首先，此问题的答案必定为某一段连续石子的和，而段数不超过 n^2，因此，我们最多只要判断 $\log_2 N^2=2\log_2 N$ 次即可。现在新的问题是如何找到第 m 大的段。首先，以每个石子开头的所有段，例如：3 2 1 2，以 2 开头

的所有段：2；2 1； 2 1 2， 由于每个石子的重量都大于 0，因此总和一定是递增的。

现在有 n 个递增段，如何快速的找到第 k 大的数呢？设各段长度为 $L_1,L_2,\ldots,L_N$,中点分别为 $M_1,M_2,\ldots,M_N$，我们每次询问各段中点的大小，若 $L_1/2+L_2/2+..+L_N/2>k$，则找到权值最大的中点，删去其后的数（下图中红色部分），否则找到权值最小的中点，删去其前面的部分（下图中黑色部分），可以证明删去的一定不是所求的数。



注：上图中每个各格子代表一个数。

由以上可知每次可以去掉某一段的 1/2，因为有 n 段，故总共需要去 $O(N\log_2 N)$次，每次找最小最大值可以用堆实现，复杂度仅为 O(lgN)，因此总的时间复杂度为 $O(\lg^2 N)$，而由上文我们知道找中值的操作总共有 $\log_2 N$ 次，故这个算法的时间复杂度为 $O(N\lg^3 N)$。

至此我们终于得到了一个多项式级别的算法，当然这个算法实现起来比较麻烦，实际效果也不甚理想，不过具有一定的理论价值，在此不做过多讨论。

回顾本题解题过程，首先我们得到了一个动态规划算法，由于很难再提高其效率，因此我们另辟蹊径，寻求新的算法。在分析过程中我们发现由于筐的重量没有上限，因此很难确定将多少石子放入同一个筐内。为了解决此难

点，我们加入了参数 P，改求可行解问题。参数的加入实际上就是强行给筐定了一个上界。正是由于这无形之中多出的条件，使得问题立刻简单化，于是我们很自然的得到了贪心算法。而最终使用二分法降低了算法的时间复杂度，成功地解决了问题。

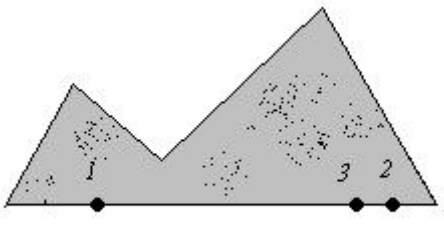
由上面的例子我们得到了此类解法的一般步骤，通常分为两步：

Ⅰ.首先引入参数 P，解决子问题：能否找到一个不优于 P 的方案

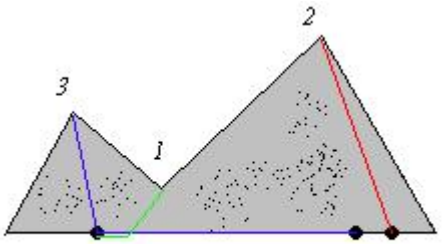
Ⅱ.从小到大枚举 P，判断是否有优于 P 的方案，直到找出原问题的最优解

一般地，参数搜索可以通过二分法或迭代降低时间复杂度，由于迭代法证明比较复杂，所以本文更多的讨论二分法。

这个方法的应用比较广泛，通常情况下和上例一样求最大（小）值尽量小（大）的题目都可以采用此方法，比如下面的例子。神秘的山：*有 n 个队员攀登 n 个山峰，每个队员需要选择一个山峰，队员们攀登的山峰各不相同，要求最后一个登顶的队员用的时间尽量短。*



Mountain T and 3 people



The solution to the example

分析：本问题分为两个部分解决：

1． 求出每个队员攀登到各个山峰所需的时间；

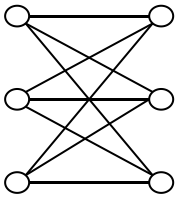
2． 找一个最优分配方案。

第一部分属于几何问题，我们可以枚举每个队员攀登山峰的位置再做简单的判断（题目规定攀登点必须为整点，这就为枚举提供了条件），这样就可求得队员们攀登上各个山峰所需的时间。

下面重点讨论第二部分。首先将我们的数据构图，很明显，我们得到的是一个二部图，假设有 3 个队员 3 个山峰，每个队员攀登的时间如下

	山峰 1	山峰 2	山峰 3
队员 1	1	3	2
队员 2	2	2	2
队员 3	1	3	3

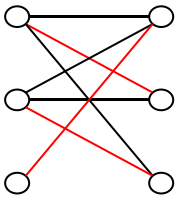
则我们可以构图，每条边附上相应的权值：



现在的任务就是在图中找一个匹配，匹配中权值最大的边尽量小。这个问题是否可以直接套用一些常见的模型呢？比如最大匹配或是最佳匹配？经过分析不难发现在这个问题中它们都是不足以胜任的，因此我们不得不做新的尝试。

上文提到过要求极大（小）值尽量小（大）的题目往往先定出上界比较容易下手，那么这题也采用类似的思路。引入一个参数 T，先求一个判定可行解的子问题：找一个方案，要求最后登山的队员所用时间不超过 T。

改变后的题目有什么不同之处呢？如果队员 i 攀登上山峰 j 所需的时间超过 T，则可以认为他在规定时间内不能攀登上山峰 j，我们就可以把这条边从图中删去。例如在上例中找一个不超过 2 的方案，经过一次“筛选”，保留的图如下。



经过这次过滤保留下来的边都是“合法边”，我们只需要在这个新的二部图中找一个完备匹配即可，一个完备匹配唯一对应着一个可行方案。而找完备匹配可以借用最大匹配的算法，因为如果一个二部图的最大匹配数等于 N，则找到了一个完备匹配，否则该图中将不存在完备匹配。（上图中的红色表示一个完备匹配），那么这一步的时间复杂度为 O(NM)，而在本题中 $M=N^2$，因此我们可以在 $O(N^3)$的时间内判断出是否存在一个方案满足最后等上山顶的队员时间不超过 T。

最后，再用二分法枚举参数 T，找到最优解。由于答案必定为某个队员攀登上其中一个山峰的时间，因此我们开始的时候可以将所有数据排序，这样最终的时间复杂度为 $O(N^3 lgN)$。

引入参数后求可行解的子问题与原问题最大的区别在于定下上界以后，我们很自然的排除了一些不可取条件，从而留下了“合法”条件，使得问题变的简单明了。

.

上面的例子在增加了上界之后，排除了一些无效条件，其实它的作用绝不仅局限于此，有些时候，它能将不确定条件变为确定条件，比如下面的例子，最大比率问题：*有 N 道题目,每道题目有不同的分值和难度,分别为 Ai,Bi；要求从某一题开始, 至少连续选 K 道题目,满足分值和与难度和的比最大。*

分析：最朴素的想法是枚举下标 i’,j’，得到每一个长度不小于 K 的连续段，通过累加 Ai’->Aj’，Bi’->Bj’求出比值，并找出比最大的一段。这样做的时间复杂度很高，由于总共有 N^2 段，每次计算比值的时间是 O(N)，则总的时间复杂度到达 $O(N^3)$，不过计算比值时，可以采用部分和优化，这样能把复杂度降至 $O(N^2)$，但仍然不够理想。

我们需要追求更出色的算法，先考虑一个简单的问题——不考虑难度（B_i），仅仅要求分值和最大（当然此时分值有正有负）。

这个简化后的问题可以直接扫描，开始时为一个长度为 K 的段，设为 $Q_1,Q_2,\ldots Q_k$，每次添加一个新数，若 $Q_1+Q_2+..+Q_{L-K}$ 小于 0，则把它们从数列中删去，不断更新最优解即可。

这样我们就能在 O(N)的时间内找到长度不小于 k 且和最大的连续段。

之所以能成功解决简化后的问题，是由于该问题中每个量对最终结果的影响是确定的，若其小于 0，则对结果产生不好的影响，反之则是有益的影响。那么原问题每个参数对最终结果的影响是不是确定的呢？很遗憾，并不是这样，因为每个题目有两个不同的参数，他们之间存在着某些的联系，而这些联系又具有不确定性，故我们很难知道它们对最终结果是否有帮助。想解决原问题，必须设法消除这个不确定因素！那么有没有办法将这些不确定的因素转化成确定的因素呢？此时，引入参数势在必行！那么我们引入参数 P，求一个新的问题：找一个比值不小于 P 的方案。

这个问题实际就是求两个下标 i',j'，满足下面两个不等式

$j'-i'+1\geq k$ ①

$(Ai'+Ai'+1....+Aj')/(Bi'+Bi'+1\ldots+Bj') \geq p$ ②

由不等式②=>$Ai'+Ai'+1....+Aj' \geq p(Bi'+Bi'+1\ldots+Bj')$

整理得$(Ai'-pBi')+(Ai'+1-pBi'+1)\ldots+(Aj'-pBj') \geq 0$

可以令 Ci=Ai-pBi，则根据上面不等式可知问题实际求一个长度不小于 k 且和大于 0 的连续段。由之前的分析可以知道我们能在 O(N)的时间内求出长度不小于 k 且和最大的连续段，那么如果该段的和大于等于 0 ，则我们找到了一个可行解，如果和小于 0 ，则问题无解。

也就是说，我们已经能在 O(N)的时间内判断出是否存在比值不小于 P 的方案，那么接下来的步骤也就顺理成章了。我们需要通过二分法调整参数 P，不断逼近最优解。

计算一下以上算法的时间复杂度，设答案为 T，则该算法的时间复杂度为 O(NlgT)，虽然这并不是多项式级别的算法，但在实际使用中的效果非常好。

引入参数后，由于增加了一个条件，我们就可以将不确定的量变为确定的量，从而解决了问题。

三．总结

本文主要通过几个例子说明了参数搜索法在信息学中的应用，从上文的例子可以看出加入参数一方面能大大降低思维复杂度，另一方面也能得到效率相当高的算法，这使得我们解最解问题又多了一中有力的武器。当然，任何事物都是具有两面性的。参数搜索在具有多种优点的同时也有着消极的一面。由于需要不断调整参数逼近最优解，其时间复杂度往往略高于最优算法，且通常依赖于某个权值的大小，使得我们得到的有时不是严格意义上的多项式算法。

文章中还总结了使用此方法解题的一般步骤，在实际应用中，我们不能拘泥于形式化的东西，必须灵活应用，大胆创新，这样才能游刃有余的解决问题。

附录

文中例题的原题

问题一

Copying Books

Before the invention of book-printing, it was very hard to make a copy of a book. All the contents had to be re-written by hand by so called *scribers*. The scriber had been given a book and after several months he finished its copy. One of the most famous scribers lived in the 15th century and his name was Xaverius Endricus Remius Ontius Xendrianus (*Xerox*). Anyway, the work was very annoying and boring. And the only way to speed it up was to hire more scribers.

Once upon a time, there was a theater ensemble that wanted to play famous Antique Tragedies. The scripts of these plays were divided into many books and actors needed more copies of them, of course. So they hired many scribers to make copies of these books. Imagine you have m books (numbered $1, 2 \ldots m$) that may have different number of pages ($p_1, p_2 \ldots p_m$) and you want to make one copy of each of them. Your task is to divide these books among k scribes, $k <= m$. Each book can be assigned to a single scriber only, and every scriber must get a continuous sequence of books. That means, there exists an increasing succession of numbers $0 = b_0 < b_1 < b_2, \ldots < b_{k-1} <= b_k = m$ such that i-th scriber gets a sequence of books with numbers between $b_{i-1}+1$ and b_i. The time needed to make a copy of all the books is determined by the scriber who was

assigned the most work. Therefore, our goal is to minimize the maximum number of pages assigned to a single scriber. Your task is to find the optimal assignment.

Input Specification

The input consists of *N* cases. The first line of the input contains only positive integer *N*. Then follow the cases. Each case consists of exactly two lines. At the first line, there are two integers *m* and *k*, *1 <= k <= m <= 500*. At the second line, there are integers $p_1, p_2, \ldots p_m$ separated by spaces. All these values are positive and less than 10000000.

Output Specification

For each case, print exactly one line. The line must contain the input succession $p_1, p_2, \ldots p_m$ divided into exactly *k* parts such that the maximum sum of a single part should be as small as possible. Use the slash character ('/') to separate the parts. There must be exactly one space character between any two successive numbers and between the number and the slash.

If there is more than one solution, print the one that minimizes the work assigned to the first scriber, then to the second scriber etc. But each scriber must be assigned at least one book.

Sample Input

```
2
9 3
100 200 300 400 500 600 700 800 900
5 4
100 100 100 100 100
```

问题二

Mysterious Mountain

A group of M people is chasing a very strange animal. They believe that it will stay on a myterious mountain T, so they decided to climb on it and have a loot.

That is, the outline of the moutain consists of N+1 segments. The endpoints of them are numbered 0..N+1 from left to right. That is to say, x[i] < x[i+1] for all 0<=i<=n. And also, y[0]=y[n+1]=0, 1<=y[i]<=1000 for all 1<=y<=n.

According to their experience, the animal is most likely to stay at one of the N endpoits numbered 1..N . And... funny enough, they soon discover that M=N, so each of them can choose a different endpoint to seek for the animal.

Initially, they are all at the foot of the mountain. (i.e at (si,0)) For every person i, he is planing to go left/right to some place (x,0) (where x is an

integer - they do not want to take time to work out an accurate place) at the speed of wi, then climb directly to the destination along a straight line(obviously, no part of the path that he follows can be OVER the mountain - they can't fly) at the speed of ci. They don't want to miss it this time, so the teamleader wants the latest person to be as early as possible. How fast can this be done?

The input will contain no more than 10 test cases. Each test case begins with a line containing a single integer N(1<=N<=100). In the following N+2 lines, each line contains two integers xi and yi(0<=xi,yi<=1000) indicating the coordinate of the ith endpoints. in the following N lines, each line contains three intergers ci,wi and si describing a person(1<=ci < wi<=100, 0<=si<=1000) - the climbing speed, walking speed and initial position. The test case containing N=0 will terminate the input and should not be regarded as a test case.

For each test case, output a single line containing the least time that these people must take to complete the mission, print the answer with two decimal places.

Sample Input

```
3
0 0
3 4
6 1
12 6
```

16 0
2 4 4
8 10 15
4 25 14
0

Sample Output

1.43

正难则反-浅谈逆向思维在解题中的应用

绍兴市第一中学 唐文斌

【摘要】逆向思维是一种思考问题的方式，它有悖于通常人们的习惯，而正是这一特点，使得许多靠正常思维不能或是难于解决的问题迎刃而解。本文通过几个例子，总结了逆向思维在信息学解题中的应用。

【关键字】

逆向思维 容斥原理 参数搜索

二分 动态规划 记忆化

【正文】

引言

我们先看一个简单的问题：

平面上有四个点，构成一个边长为 1 的正方形。现在进行一种操作，每次可以选择两个点 *A* 和 *B*，把 *A* 关于 *B* 对称到 *C*，然后把 *A* 去掉。

求证：不可能经过有限次操作得到一个边长大于 1 的正方形

操作后的结果是相当复杂的，如果我们从正面着手，很难证明命题。不妨从反面来看问题：观察可以发现，每一步操作都是可逆的。即，我们如果可以把正方形变大，也可以把正方形变小。

证明：

不妨设四个顶点都是整点。

假设我们可以通过有限次操作得到一个边长大于 1 的正方形，那么我们把所有操作反过来对原正方形进行操作，我们可以得到一个边长小于 1 的正方形。

因为四个顶点都是整点，操作之后，点的坐标依然是整数。所以我们得到一个边长小于 1 且四个点都是整点的正方形。这显然不可能。

所以假设不成立。命题得证。

上面的例子说明了逆向思维在数学问题中的应用。山重水复疑无路，应用逆向思维，换个角度看问题，便柳暗花明又一村了。

例一、Dinner Is Ready[1]

题目大意：

妈妈烧了 M 根骨头分给 n 个孩子们，第 i 个孩子有两个参数 Min_i 和 Max_i，分别表示这个孩子至少要得到 Min_i 根骨头，至多得到 Max_i 根骨头。

输入：

第一行包含两个数 $n(0<n\leq 8)$，$M(0<M)$，表示孩子数量和骨头的数量。

接下来 n 行分别输入 Min_i 和 Max_i $(0\leq Min_i\leq Max_i\leq M)$

输出：

输出一个整数，表示妈妈有多少种分配方案(骨头不能浪费，必须都分给孩子们)

[1] Zhejiang University Online Judge 1442(acm.zju.edu.cn)

初步分析：

该题模型很简单，即求如下方程组的整数解的个数：

$$\begin{cases}\sum_{i=1}^{n} X_i = M \\ Min_1 \le X_1 \le Max_1 \\ Min_2 \le X_2 \le Max_2 \\ Min_3 \le X_3 \le Max_3 \\ \cdots\cdots \\ Min_n \le X_n \le Max_n\end{cases}$$

我们知道，方程组简单形式 $\begin{cases}\sum_{i=1}^{n} X_i = M \\ X_i \ge 0\end{cases}$ 的整数解个数为 C_{M+n-1}^{n-1}[1]

设 $Y_i = X_i + Min_i$ ，则原方程组转化为 $\begin{cases}\sum_{i=1}^{n} Y_i = M - \sum_{i=1}^{n} Min_i \\ 0 \le Y_1 \le Max_1 - Min_1 \\ 0 \le Y_2 \le Max_2 - Min_2 \\ 0 \le Y_3 \le Max_3 - Min_3 \\ \cdots\cdots \\ 0 \le Y_n \le Max_n - Min_n\end{cases}$

对于下界限制，我可以通过换元得到简单形式，但是因为有上界的限制，我们似乎还无法直接计算出答案。

应用逆向思维：

设 S 为全集，表示满足 $X_i \ge Min_i$ 的整数解集。

设 S_i 为 S 中满足约束条件 $X_i \le Max_i$ 的整数解的集合，$\overline{S_i}$ 为 S_i 在 S 中的补集，即满足 $X_i > Max_i$。

[1] 这是一个很经典的组合问题，可由隔板原理得出答案。

$|S_i|$无法计算，但是，$|\overline{S_i}|$可解！！！我们希望把$|S_i|$的计算转化到可解的$|\overline{S_i}|$。

于是：

$$Answer = |S_1 \cap S_2 \cap S_3 ... \cap S_n| =$$
$$|S| - (|\overline{S_1}| + |\overline{S_2}| + ... + |\overline{S_n}|) + (|\overline{S_1}| \cap |\overline{S_2}| + |\overline{S_1}| \cap |\overline{S_3}| + ... + |\overline{S_{n-1}}| \cap |\overline{S_n}|) - .... + (-1)^n * |\overline{S_1} \cap \overline{S_1} \cap ... \cap \overline{S_n}|$$

这是一种容斥原理的形式。至此，问题已经解决。我们应用逆向思维，在原集合的模$|S_i|$不可解的情况下，通过可解的$|\overline{S_i}|$得到答案。并给出了一个基于容斥原理的算法。时间复杂度为 $O(2^n * (n+M))$。

例二、Greedy Path[1]

题目大意：

有 n 个城市，被 m 条路连接着。最近成立了一些旅行社，在这些城市之间给旅行者们提供服务。旅行者从城市 i 到城市 j 需要付给旅行社的费用是 $C_{i,j}$，需要的时间为 $T_{i,j}$。很多旅行者希望加入旅行社，但是旅行社只有一辆车。于是旅行社的老板决定组织一次旅行大赚一笔。公司里的专家需要提供一条使得贪心函数 $F(G)$最大的回路 G。$F(G)$ 等于总花费除以总时间。但是没有人找到这样的回路，于是公司的领导请你帮忙。

输入：

第一行包含两个数 n(3≤n≤50),m 分别表示点数和边数。

接下来 m 行每行包含一条路的描述。输入四个数,A,B,$C_{A,B}$,$T_{A,B}$（0≤$C_{A,B}$≤100，0≤$T_{A,B}$≤100）

[1] Saratov State University Online Judge 236(acm.sgu.ru)

输出：

如果不存在这样的路，输出 0。

否则输出回路中包含的城市个数，然后依次输出通过的城市的顺序。如果有很多条这样的路，输出任意一条。

初步分析：

题目要求是求一条回路，但不是边权和最大或者最小，所以我们不能直接使用经典算法。

设 $G = (V , E)$，S 为 G 中所有回路 $C = (V' , E')$组成的集合。

我们的目标是找到集合 S 中的一条回路使得 $F(C)$取到最大值：

$$F(C) = \frac{\sum_{e\in E'} C_e}{\sum_{e\in E'} T_e}$$

应用逆向思维：

如果我们知道 $C^* = (V^*,E^*) \in S$ 是一条最优回路，那么

$$F(C^*) = \frac{\sum_{e\in E^*} C_e}{\sum_{e\in E^*} T_e} \Rightarrow \sum_{e\in E^*} C_e - F(C^*) \times \sum_{e\in E^*} T_e = 0$$

于是我们定义函数 $o(t)$: $o(t) = \max_{C=(V',E')\in S} \sum_{e\in E'} C_e - t\sum_{e\in E'} T_e$

我们做一个猜想：如果有 $o(t^*)=0$，那么存在 $C^* = (V^*,E^*) \in S$ 满足

$$t^* = \frac{\sum_{e\in E^*} C_e}{\sum_{e\in E^*} T_e}$$

我们认为 C^*就是一条最优回路。

证明：假设存在另一条回路 $C_1= (V_1,E_1) \in S$ 更优，则：

$$t1=\frac{\sum_{e\in E_1}C_e}{\sum_{e\in E_1}T_e}>t^*\Leftrightarrow 0<\sum_{e\in E1}C_e-t^*(\sum_{e\in E1}T_e)\le o(t^*)$$

但是这与 $o(t^*)=0$ 矛盾。所以 C*是一条最优回路。

如果 t^*是最优答案，则存在：

$$\begin{cases}o(t)=0\Leftrightarrow t=t^*\\ o(t)<0\Leftrightarrow t>t^*\\ o(t)>0\Leftrightarrow t<t^*\end{cases}$$ （性质一）

于是我们就得到了算法：

我们从一个包含 t^*的区间(t_l , t_h)开始。（例如 tl = $\min\limits_{e\in E}\frac{C_e}{T_e}$ ，th = $\max\limits_{e\in E}\frac{C_e}{T_e}$ ）

每一次，选取 t_l 与 t_h 的中点 t , 计算 $O(t)$。

O(t)的计算方法：

对于边 $e\in E$, 设一个新的参数 $W_e = C_e - t * T_e$

用 *Floyd* 算法计算有向图的最大权值环。该最大权值即 $O(t)$。

根据性质一，更新 t_l 或 t_h , 得到新的上下界，继续二分，直到达到精度要求。

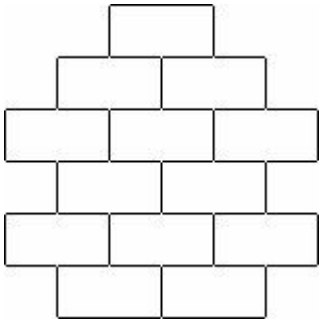
假设二分的次数为 K , 则算法的时间复杂度为 $O(K * n^3)$。这虽然不是一个严格的多项式算法，但是对于题目给定的范围，该算法可以很快地求出答案。

例三、Building Towers[1]

题目大意：

有 N 块积木，我们需要用这些积木造塔。每个塔有 *H* 层，最底层包含 *M* 块积木；对于上面的每一层，包含的积木块数必须比下面一层的多 1 或者少 1。

下图是一个合法的塔(*H*=6，*M*=2，*N*=13)：



你的任务是计算一共可以建造出多少种不同的塔（注意积木不一定要全部用完）。两个塔被认为不同，仅当存在一个 *i*（1≤*i*≤*H*），两个塔的第 *i* 层包含不同数目的积木。

我们用 *H* 个正整数来表示一个塔的结构（从底部到顶部）。你还需要输出字典序第 K 小的建塔方案。

输入：

第一行包含三个整数,*N*,*H*,*M* (*N*≤32767, *H*≤60, *M*≤10)。

接下来每行包含一个正整数 *K*（除了最后一行）。最后一行包含一个-1 表示输入的结束。

输出：

第一行包含可以建造的不同的塔的总数。

[1] Ural Online Judge 1148 (acm.timus.ru)

接下来每一行输出对应的字典序第 *K* 小的塔的结构。

初步分析：

首先，看到题目，思维的惯性让我们下意识的想到了经典、传统的动态规划算法。

用 *F*[*i*,*j*,*k*]表示当前层有 *i* 块砖，还剩下 *j* 层，可用的砖块数量为 *k* 的方案总数。可以写出动态规划方程为：

F[*i*,*j*,*k*] = *F*[*i*+1,*j*+1,*k*-*i*] + *F*[i-1,*j*+1,*k*-*i*]

边界条件: *F*[*M*,*H*,*N*]=1，其他为 0

很容易看出，这个动态规划方程一共涉及了 *N***M***K* 个状态，最大约为 60*70*2400 = 10*M*。如果每个状态用一个 *double* 来保存信息，则至少需要 80*M* 的内存（注意，因为我们需要输出字典序第 *K* 小的塔，所以我们无法使用滚动储存的优化）。不论是时间复杂度还是空间复杂度，都让人难以接受。至此，动态规划算法失败了。

应用逆向思维：

自底向上的动态规划在时空上都难以让人承受。我们不妨在处理时“反个方向”，看看自顶向下的搜索算法。

搜索算法向来是“低效”、“指数”的代名词。然而，如果加入记忆化因子，则正好是一个“逆向动态规划”的过程。

将动态规划的顺序作一个反转，其好处在于：

首先，自顶向下的看问题，有“一览众山小”的开阔视野，可以顺利地为如何搜，先搜什么后搜什么，以及如何剪枝等作合理的布局。在本题中，这一点体现为搜索过程不会搜索到很多荒唐的状态，例如砖块数目明显不

够，奇偶性差异等等。而在一些最优化的问题中，搜索记忆化的方法可以有效地配合最优性剪枝，避免许多明显没有价值的状态，而这正是正向的动态规划所不能企及的。

第二，在类似于本题的计数问题中，可以采用部分记忆化的方法，即只记录那些比较容易被多次搜索到的状态。这样一来，减少了存储的状态量，加快了查询的速度。

实现的时候，我们用三元组（*N*，*H*，*M*）来表示一个状态，即当前层有 *M* 块砖，还剩下 *H* 层，可用的砖块数量为 *N*。*F*(*N*,*H*,*M*)表示该状态下的方案数。用 *Hash* 表存储，并加入以下一些优化：

1） 可以事先计算出当前状态最多和最少还需要的砖块数目。如果 *N* 小于最小的需求量，那么显然答案为 0；如果 *N* 大于最大需求量，那么可以把 *N* 设置为该最大需求量，以避免等同状态的重复存储。

2） 如果 *N* 不小于最大需求量且 $H \le M$，则答案为 2^H。

3） 为了减少存储的状态量，我们设一个存储上界，即仅当 *F*(*N*,*H*,*M*) 不大于该上界时才将其存入 *Hash* 表。为什么这样做是有效的呢？观察可以发现，搜索的状态树是以指数的形式往下展开的。也就是说，在树越深的地方，状态量会越多，进入查询也越频繁，而接近根的地方，状态的分布比较稀疏，当 *F*(*N*,*H*,*M*)较大时，访问它们的频率就会相应的降低。所以我们可以重复搜索这些状态，从而减少存储状态量，加快 *Hash* 表检索速度。

我们不妨来看一下“逆向动态规划”的表现：

N	*H*	*M*	正向动态规划所需要处理的状态数量	逆向动态规划中 *Hash* 表中所存储的状态量	正逆向规划中状态数目的比值
1000	40	10	1200000	3055	392.80
1500	50	10	2625000	5371	488.74
1200	60	10	2880000	49875	57.74
1500	60	10	3600000	38018	94.69

即便是在最坏情况下，逆向动态规划也只需要处理大约 1/50 的状态。

至此，我们已经完美地解决了这个问题。

【总结】

例一，是一种补集转化的思想，我们对原集合无从下手，转而去求原集合的补集，从反面看问题，得到答案。

例二，我们没有去看问题的反面，而是去看一个事实的反面。我们现在不知道答案，如果知道答案又将如何呢？

例三，在经典的动态规划无法解决问题的情况下，我们将规划顺序颠倒，得到了逆向动态规划的方法，从而避免许多无效状态，在时间和空间上都取得了质的飞跃。

上述三个例子，都是逆向思维的一些简单应用。“正难则反”的逆向思维，帮助我们打破思维定势，以退为进，避其锋芒，攻其软肋。让我们在山重水复疑无路时柳暗花明又一村；在踏破铁鞋无觅处时得来全不费功夫；让我们在为伊消得人憔悴后蓦然发现那人却在灯火阑珊处。所谓反弹琵琶成新曲，愿逆向思维助你一臂之力，更上一层楼！

【感谢】

特别感谢安徽芜湖一中周源同学的帮助，感谢所有帮助过我的人。

Acm.sgu.ru
Acm.timus.ru
Acm.zju.edu.cn

【参考文献】

刘汝佳，黄亮《算法艺术与信息学竞赛》 清华大学出版社

Gunnar W.Klau 等《The Fractional Prize-Collecting Steiner Tree Problem On Trees》

任初农《说说逆向思维》

【附件】

例三《Building Tower》一题的源程序 Tower.cpp

【附录】

[例一原题]

Dinner Is Ready

Dinner is ready! WishingBone's family rush to the table. WishingBone's mother has prepared many delicious bones. WishingBone has several brothers, namely WishingTwoBones, WishingThreeBones. WishingBone wants at least one bone, WishingTwoBones two and WishingThreeBones three. :-P But as they are not too greedy (or they want to keep shape), they decide that they want at most twice of their minimal requirement. Now let's suppose mother has prepared 7 bones, one way to distribute them is 2-2-3, the other two ways are 1-3-3 and 1-2-4. Of course mother doesn't want to waste any bones, so if she had prepared 13 bones, she would end up without any feasible distribution.

As curious as WishingBone, mother wants to know how many different ways she can distribute those bones.

Input

The first line of input is a positive integer N <= 100, which is the number of test cases.

The first line of each case contains two integers n and m (0 < n <= 8, 0 < m), where n is the number of children and m is the number of bones mother has prepared.

The next n lines have two integers mini and maxi (0 <= mini <= maxi <= m), which is the minimal and maximal requirement of the ith child.

Output

N integers which are the numbers of ways to distribute the m bones, one per line.

Sample Input

```
2
3 7
1 2
2 4
3 6
3 13
1 2
2 4
3 6
```

Sample Output

```
3
0
```

[例二原题]

Greedy Path

There are N towns and M routes between them. Recently, some new travel agency was founded for servicing tourists in these cities. We know cost which tourist has to pay, when traveling from town i to town j which equals to Cij and time needed for this travel - Tij. There are many tourists who want to use this agency because of very low rates for travels, but agency still has only one bus. Head of agency decided to organize one big travel route to gain maximal possible amount of money. Scientists of the company offer to find such a cyclic path G, when greedy function f(G) will be maximum. Greedy function for some path is calculated as total cost of the path (sum of Cij for all (i,j) - routes used in path) divided by total time of path (similar to Cij). But nobody can find this path, and Head of the company asks you to help him in solving this problem.

Input

There are two integers N and M on the first line of input file (3<=N<=50). Next M lines contain routes information, one route per line. Every route description has format A, B, Cab, Tab, where A is starting town for route, B - ending town for route, Cab - cost of route and Tab - time of route (1<=Cab<=100; 1<=Tab<=100; A<>B). Note that order of towns in route is significant - route (i,j) is not equal to route (j,i). There is at most one route (in one direction) between any two towns.

Output

You must output requested path G in the following format. On the first line of output file you must output K - number of towns in the path (2<=K<=50), on the second line - numbers of these towns in order of passing them. If there are many such ways - output any one of them, if there are no such ways - output "0" (without quotes).

Sample test(s)

Input

```
4 5
1 2 5 1
2 3 3 5
3 4 1 1
4 1 5 2
2 4 1 10
```

Output

```
4
1 2 3 4
```

[例三原题]

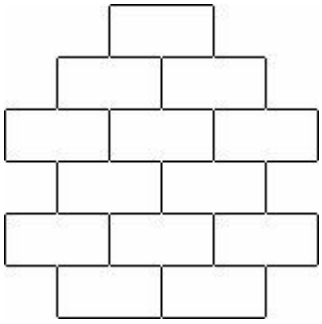
Building Towers

Time Limit: 2 second

Memory Limit: 1000 KB

There are N cubes in a toy box which has 1-unit height, the width is double the height. The teacher organizes a tower-building game. The tower is built by the cubes. The height of the tower is H (h levels). The bottom of the tower contains M cubes; and for all above level, each must contains a number of cubes which is exactly 1 less than or greater than the number of cubes of the level right below it.

Below is an example of a tower of H=6, M=2, N=13.



Your task is to determine how many different towers can be there. Two towers are considered different if there is at least one number i (1< i <=H) so that the i'th level of one tower contains a different number of cubes to the i'th level of the other tower.

Each tower is represented by an array of H positive integers which determines the structure of the tower from the bottom level up to the top level. Those arrays are sorted ascendingly.

Input

1st line contains three positive number N, H and M (N <= 32767, H<=60, M<=10).

Each following line (except the last)contains an integer k. The last line contains number -1.

Output

1st line is the total of different towers that can be founded.

Each following line contains an array that represents the structure of the tower corresponding to the number K from input.

Sample Input

22 5 4
1
10
-1

Sample Output

10
4 3 2 1 2
4 5 4 5 4

Hint

Numbers which are on the same line should be separated by at least one space.

You don't have to use all N cubes.

图论的基本思想及方法

湖南省长沙市长郡中学 任恺

【摘要】

文章着眼于图论基本思想及方法的讨论，不涉及高深的图论算法。

文章主要从两方面阐述图论的基本思想：一是合理选择图论模型；二是如何深入挖掘问题本质，充分利用模型的特性。同时还归纳了一些解决问题的普适性方法。

【关键字】

基本思想、图论模型、问题本质、定义法、分析法、综合法

【正文】

一、引论

图是用点和边来描述事物和事物之间的关系，是对实际问题的一种抽象。之所以用图来解决问题，是因为图能够把纷杂的信息变得有序、直观、清晰。因而图论中最基本的思想就是搭建合适的模型，深刻挖掘问题的本质，分析和利用图论模型各种性质，从而到达解决问题的目的。下面着重从模型的选择和发掘利用图的性质来阐述图的基本思想和运用方法。

二、合理选择图论模型

在解决一道实际问题时，往往先将实际问题抽象成一个数学模型，然后在模型上寻找合适的解决方法，最后再将解决方法还原到实际问题本身。而图论模型就是一种特殊的数学模型。在搭建图论模型时，是通过图中的点和边来体现原问题的特点。搭建的模型务必要真实的、贴切的和透彻的反映出原问题的本质，同时也要做到力求简练、清晰。图论问题往往关系错综复杂，变化多端，因此搭建一个合适的模型实非易事。在选择图论模型时，应该深入分析实际问题的特点，大胆的猜想和验证。下面通过一个具体实例，来揭示选择合适图论模型的重要性和一些方法：

例一：滑雪者（Poland Olympiad of Informatics 2002 Stage III：Skiers）

题目大意：给出一个平面图，图中有 n（$2 \le n \le 5000$）个点，m 条有向边。每个点都有**不同的横坐标和纵坐标**，有一个最高点 v_h 和一个最低点 v_l。每条有向边连接着两个不同的点，方向是从较高点连到较低点。对于图中任意一点 u，都至少存在一条 v_h 到 u 的路径和一条 u 到 v_l 的路径。任务：图中由每个点发出的边都已经**按照结束点的位置从左到右给出**。要求你用若干条从 v_h 到 v_l 的路径覆盖图中所有的边，并且使路径数最少。所谓覆盖，就是指每条边至少在一条路径中出现。选取的路径之间可以有相同的边。（题目中和下面分析中所说的路径都是有向路径，若 a 到 b 存在一条路径，并不表示 b 到 a 一定存在一条路径。）

原题请见附录

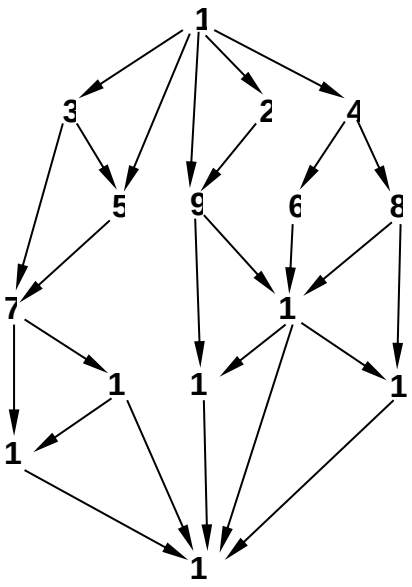


图 2

样例： 图 2-1 中所示的平面图最少需要 8 条路径才能覆盖所有的边。

2.1 以网络流为模型

分析一下样例，很快联想到经典的网络流模型：最高点 v_h 是网络的源点，而最低点 v_l 是网络的汇点。题目中的路径是网络中从源点到汇点的流。要求用路径覆盖图中所有的边，且路径数最少，就是要求网络中每条边的流量大于等于 1，并且从源点流出的总流量最小。

因此解决这个问题只需要建立一个有容量下界的网络，然后求这个网络的最小可行流。大致做法：

首先求出可行流：枚举每条流量为 0 的边，设为(i,j)。找到一条从 s 到 i 的路径和一条从 j 到 t 的路径。对“$s-i-j-t$”路径上的每一条边流量加 1，这样既满足了每个点的流量平衡，又满足了边(i, j)的容量下界。然后在可行流上进行修改，从汇点到源点求一个最大可行反向流，就得到了一个最小可行流。

分析算法二的复杂度：求可行流时，可以先预处理源点和汇点到每个顶点的路径，因此构造可行流的时间复杂度为 $O(|E|+|V|)$。求最大流时，可以用朴素的增广路算法，复杂度为 $O(|E|C)$，C 是进行增广的次数。因为是平面图，根据欧拉公式有 $O(|E|)=O(|V|)$，而反向流的流量最大为 $O(|E|)$，所以总的复杂度为 $O(|V|^2) = O(n^2)$。此算法实际效率很高，能够迅速的通过测试数据。但是，这种模型并没有很好的挖掘原题中平面图的性质，所以改进的余地应该很大。

2.2 以偏序集为模型

题目中强调了每个点都有不同的横纵坐标，图是有向无环平面图。而且图中两点之间的有向边似乎反映着一种元素的比较关系。是否存在更好的模型描述此图呢？为了更好的揭示问题的本质，下面引入偏序集。

2.2.1 偏序集的相关概念

偏序集是一个集合 X 和一个二元关系 R，符合下列特性：

a) 对于 X 中的所有的 x，有 $x\ R\ x$，即 R 是**自反的**。

b) 对于 X 中的所有的 x 和 y，只要有 $x\ R\ y$ 且 $x \neq y$，就有 $y \not R\ x$，即 R 是**反对称的**。

c) 只要有 $x\ R\ y$ 和 $y\ R\ z$，就有 $x\ R\ z$，即 R 是**传递的**。

符合这些特性的关系叫做**偏序**，通常用≤标记 R。$a \le b$ 也可以记作 $b \ge a$。若有 $a \le b$，且 $a \neq b$，那么就记作 $a < b$ 或者 $b > a$。<又叫做**严格偏序**关系。含偏序≤的偏序集 X 用$(X, \le)$表示。

令 R 是集合 X 上的一个偏序，对于属于 X 的两个元素 x、y，若有 $x\ R\ y$ 或 $y\ R\ x$，则 x 和 y 被称作是**可比的**，否则被称为**不可比的**。集合 X 上的一个偏序

关系 R，如果使得 X 中的任意一对元素都是**可比的**，那么该偏序 R 就是一个**全序**。例如，正整数集上的小于等于关系就是一个全序。

令 a 和 b 是偏序集$(X, \leq)$中的两个元素。若有 $a < b$，且 X 中不存在另一个元素 c，使得 $a < c < b$，那么就称 a 被 b **覆盖**（或 b **覆盖** a），记作 $a <_c b$。若 X 是一个有限集，由偏序集的传递性易知，任一个偏序关系都可以用多个覆盖关系表示出来，也就是说可以用覆盖关系有效的表示偏序关系。

有限偏序集$(X, \leq)$的图表示（也就是哈斯图）是用平面上的点描述的。偏序集中的元素用平面上的点来表示。若有 $a < b$，那么 a 在平面上的位置（严格说是坐标平面中的纵坐标）就应当低于 b 在平面上的位置。若 $a <_c b$，那么 a 和 b 之间连一条边。也可以用有向图来表示偏序关系，图中的每个结点对应偏序集中的每个元素。若偏序集中的两个元素有 $a <_c b$，那么对应到图中的两个结点 a 和 b，就有一条从 b 到 a 的有向边(b, a)。因此可以看出原图事实上是一个偏序集的图表示。

链：链是 E 的一个子集 C，在偏序关系≤下，它的每一对元素都是可比的，即 C 是 E 的一个全序子集。

反链（或称杂置）：顾名思义，它和链的定义恰恰相反。反链是 E 的一个子集 A，在偏序关系≤下，它的每一对元素都是不可比的。

链和反链的大小是指集合中元素的个数。

2.2.2 构筑原问题的偏序集模型

有了上文有关偏序集的概念，不难搭建出原问题的偏序集模型：令原图表示的偏序集为$(X, \leq)$，而新构造的偏序集为$(E, \leq)$。则集合 E 满足

$E=\{(u,v)\,|\,u,v\in X$ 且 $u\geq_c v\}$，即 E 中的元素全部是图中的有向边。令 *a*、*b* 为 *E* 中的两个元素，设 $a=(u_a,v_a), b=(u_b,v_b)$。当且仅当 $u_a\leq u_b$ 且 $v_a\leq v_b$ 时，有 *a* ≤ *b*，即存在一条从有向边 *a* 到有向边 *b* 的路径。当且仅当 $v_a = u_b$ 时，有 $a \leq_c b$。

原问题可以重新用偏序集语言表述为：将偏序集（*E*, ≤）划分成最少的链，使得这些链的并集包含所有 *E* 中的元素。直接计算链的个数似乎并不容易，好在有 Dilworth 定理揭示了链与反链的关系，从而使得问题的目标进一步转化。

定理 2.1 : Dilworth 定理 令（*E*, ≤）是一个有限偏序集，并令 *m* 是 *E* 中最大反链的大小，*M* 是将 *E* 划分成最少的链的个数。在 *E* 中，有 *m* = *M*。

证明过程请参见附录。

根据 Dilworth 定理，问题转化成求 *E* 中最长的反链的大小。也就是要求在原图中选尽量多的边，同时保证选出的边是互不可达的（即在(*E*, ≤)中不可比）。如何求解最长的反链呢？事实上，这和原题给出的平面图有很大关系，接下来，返回到原图上继续讨论。

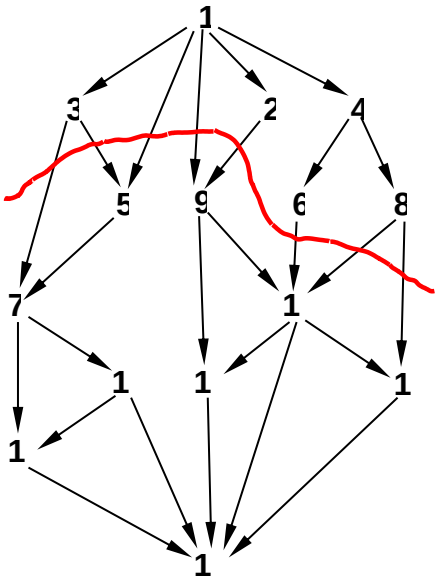
2.2.3 从偏序集模型回归到原题

由于原题给出的图是平面图，而且图中顶点也是从左到右给出的，那么对于反链中的所有边都能按照从左到右的顺序排列好。如果用一条线将最长反链所对应的边从左到右连起来（如图 2-2 所示），那么这条线不会与平面图中的其它边相交。更加准确地说：

定理 2.2：将最长反链所对应的边从左到右排列好，相邻的两条边一定是在同一个域（闭曲面）中。所谓的域，是由从一个点到另一个点（一个是极高

点，一个是极低点）的两条不同路径（两条路径没有公共边）围成的一个曲面，在这个曲面里没有其他的点和边（如图 2-3 所示），记作 *F*。在围成域 *F* 的两条路径中，左边的那条路径定义为 *F* 的左边界，右边的那条路径定义为 *F* 的右边界。

关于定理 2.2 的简略证明请参见附录。



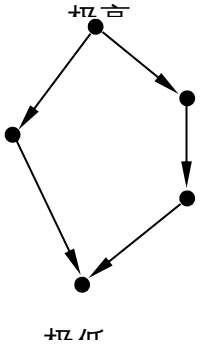


图 2

受定理 2.2 的启发，我们可以用递推的方法求得图中最长反链的长度。设 *f*(*x*)表示在边 *x* 左边的平面区域中以 *x* 结尾的最长反链的长度。设 *x* 在某个域 *F* 的右边界上，有 $f(x)=\max\{f(y)\}+1$（*y* 是 *F* 左边界上的边）。因为根据定理 2.2，若 *x* 在某个最长反链中，那么反链中和 *x* 相邻且在 *x* 左边的边，只有可能在域 *F* 的左边界上。得到这个递推式后，只需要按照从左到右从上到下把每一个域求出进行递推即可。

2.2.4 相应的算法设计

可以用 DFS 深度优先遍历实现平面图中域的寻找。DFS 中需要记录两个信息：结点的颜色和扩展它的父结点。每个结点的颜色用 C[*u*]来记录。C[*u*]有三

种状态：白色表示结点 *u* 尚未被遍历，一开始所有结点的颜色都是白色；灰色表示结点 *u* 已经被遍历，但是它尚未检查完毕，也就是说它还有后继结点没有扩展；黑色表示结点 *u* 不但被遍历且被检查完毕。扩展它的父结点用 pre 记录。

算法的大致框架如下：

DFS（结点 u）

begin

C[u] ← 灰色

while v 是 u 后继结点 **do** （按照从左到右的顺序扩展 u 的后继结点 v）

if C[v] 是白色

then begin

pre[v] ← u

DFS(v)

end else begin

v_l ← v

v_h ← v

while C[v_h] 是黑色 do

v_h ← pre[v_h] (是黑色，说明是域的边界上的结点；灰色就是极高点)

递推求出右边界的 f(*x*) (pre 回溯的边是左边界，$v_h - u - v_l$ 是右边界)

pre[v] ← u

end

C[u] ← 黑色

end

计算上述算法的复杂度：*a*.对每一个点进行 DFS 遍历，复杂度为 $O(|E|)$；*b*.回溯寻找每个域边界上的边，并且进行递推求解。由于是平面图，每条边最多属于两个不同域的边界，因此这一步的复杂度为 $O(|E|+|F|)$。因为原题给出的图是平面图，根据欧拉定理，边数$|E|$和域数(或者说面数)$|F|$都是 $O(n)$级别的。因此总的时间复杂度为 $O(n)$。

2.3 小结

方法一利用网络流模型直接的体现了原题的网络（有向无环）特性，解法具有一般性，但是没有充分的体现出原题的平面图性质。而方法二很好的利用偏序集模型实现了问题目标的转变，从原来的网络流问题回归到问题本身的平面图上，完整的揭示了问题的本质。正是由于回归到问题的本质，后面才能用 DFS 充分挖掘平面图的性质，得到最优复杂度的算法。

从上述两种方法的比较可以看出，两种不同的图论模型导致了两种算法在时间复杂度上的较大差异，可见选择模型的重要性。正所谓“磨刀不误砍柴功”，在设计算法之前，选择一个正确的图论模型往往能够起到事半功倍的效果，不仅能降低算法设计的难度，还使设计出的算法简单高效。

在为实际问题选择合适的图论模型的时候，不能仅仅局限于问题表面所表现的某些性质，只根据自有的经验去解题，否则会落入俗套，得不到效率最高的算法。新题新作，应该创新思路，深入分析问题的各种性质，将这些性质结合在一起，从而寻找到最能体现问题本质的图论模型。

很多时候，最终算法并不是基于所选图论模型来设计的，就如方法二，偏序集并没有出现在 DFS 中，但一旦想到偏序集，问题可以说就解决了一半。图

论模型更多的是思考的一种过渡，使思路变得清晰，就像一座灯塔，指引你到成功的彼岸。

三、充分挖掘和利用模型的性质

上一节讲述了选择合适模型的重要性和搭建图论模型的方法。建立模型仿佛是为算法设计搭建一个平台，接下来的工作是在这个平台上充分挖掘和利用原题的性质，设计一个解决问题的好算法。如何挖掘和利用模型的性质呢？下面同样用一个具体实例来说明。

例二：爱丽丝和鲍勃（ACM *Central European Programming Contest* 2001 Problem A : Alice and Bob）

题目描述：爱丽丝和鲍勃在玩一个游戏。爱丽丝画了一个有 n（$n \le 10000$）个顶点的凸多边形，多边形上的顶点从 1 到 n 按任意顺序标号。然后她在多边形中画了几个不相交的对角线（公共点为顶点不算相交）。她把每条边和对角线的端点标号告诉了鲍勃，但没有告诉他哪条是边，哪条是对角线。鲍比必须猜出顶点顺（逆）时针的标号序列，任意一个符合条件的序列即可。

原题请见附录

例如：$n = 5$，给出的边或对角线是(1,4),(4,2),(1,2),(5,1),(2,5),(5,3),(1,3)。

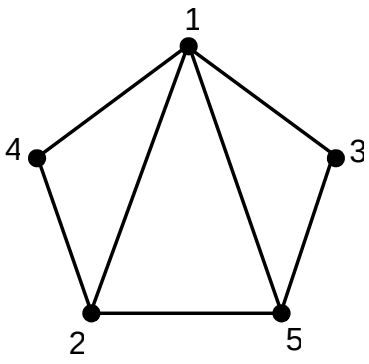


图 2

那么一个可能的顶点标号顺序为(1, 3, 5, 2, 4)。对应的多边形如图 3-1：

3.1 建立模型

根据题目意思，不难得到它的图论模型：凸多边形对应了一个有 n 个顶点的图 G，多边形的边和对角线对应着图 G 中的边。而且十分明显的是，图 G 是一个平面图，根据欧拉公式，图中边的数量级为 $O(n)$。

研究图 G 的平面图性质可以发现**结论 3.1**：**一条对角线将凸多边形分成了两部分，每部分都至少含有一个除对角线外顶点，而且这两部分分居在对角线的两侧**。由此可知，在图 G 中只存在惟一的一条哈密顿回路。因为对角线会将多边形分成不相关连的两部分，所以对角线不可能存于哈密顿回路上。因此哈密顿回路上的边都是由多边形上的边组成，而多边形的边只有 n 条，可知哈密顿回路也就只有一条。不妨设这条哈密顿回路为 $H_1,H_2,H_3……H_n$。问题的目标就是要找到这个哈密顿回路。下面讨论如何利用这些性质来设计算法。

3.2 利用边的连通性

由结论 3.1 可知，如果一条边是对角线，那么将对角线的两个端点从图 G 中删除，图 G 一定会变成两个互不可达的连通分块；而如果一条边是多边形上的边，那么将这条边的两个端点删除，图 G 将仍然是连通的。也就是说能够根据图 G 中边的连通性，来判断一条边是对角线还是边。因此，得到算法一：

算法一：

1．枚举图中的每条边(u, v)，进行如下判断：将 u 和 v 两个顶点从图 G 中删除得到新图 G'。在新图 G'任选一点 a，然后从 a 开始对图 G'进行遍历。如果

a 能够到达图 *G*'中的其它点，那么就说明图 *G*'是连通的，(*u*, *v*)是多边形的边；否则，图 *G*'不连通，(*u*, *v*)是多边形的对角线。

2．将图 *G* 中所有对角线删除，得到新图 *C*。这时的新图 *C* 便是图 *G* 中的哈密顿回路，因此只要对其进行一次遍历就可以得到多边形顶点的标号序列了。

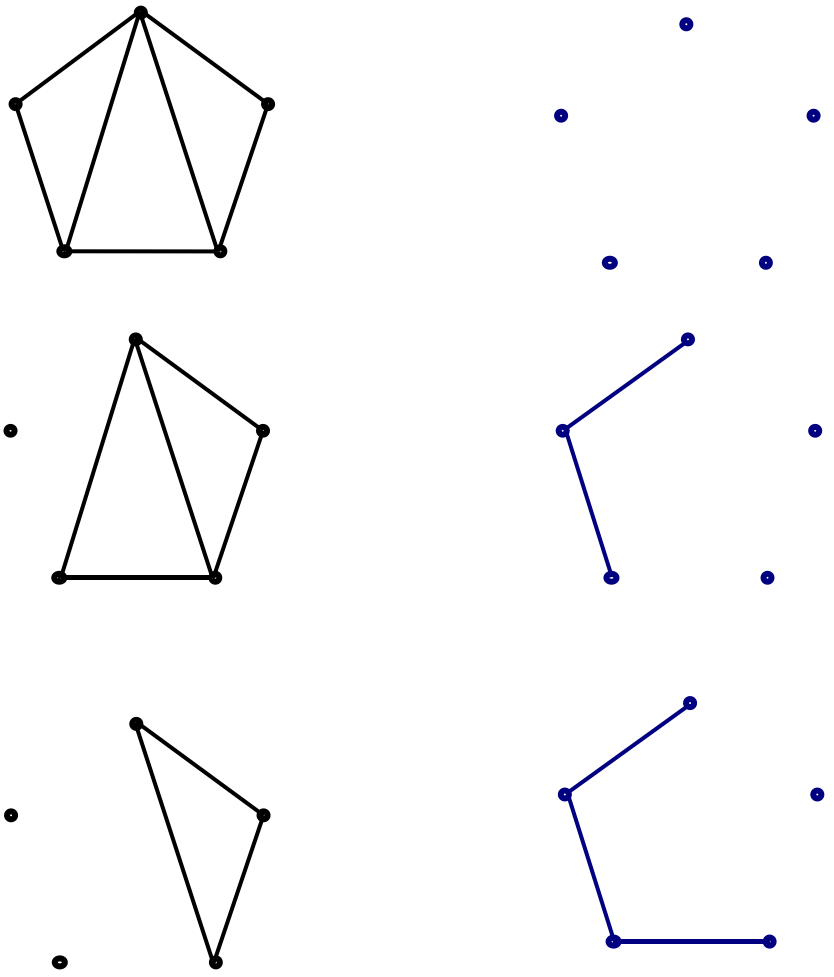
分析一下算法一的复杂度：步骤 1 中，要枚举的边最多为 2*n* 条，每判断一次图的连通性为 $O(n)$，所以复杂度为 $O(n^2)$；步骤 2 中，删除边和遍历新图的复杂度都为 $O(n)$。所以总的复杂度为 $O(n^2)$。n 最大时候达到 10000，因此算法一的复杂度稍稍高了一点，仍要继续优化。

3.3 分而治之

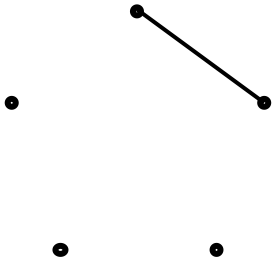
继续挖掘图 *G* 的性质发现，由于图 *G* 中的对角线是不相交的，那么必定存在一个顶点 *u*，它的度数为 2。而且可以肯定的是，和 *u* 相连的两条边一定是多边形的边。将以 *u* 为顶点的三角形从多边形上删除，剩下的图形仍然是一个多边形。是否能用这个性质判断一条边是多边形的边还是对角线呢？答案是肯定的。

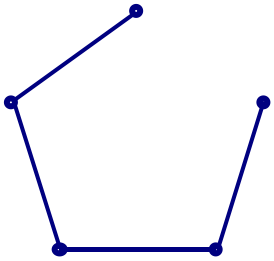
由于图中一定存在一个顶点 *u*，它的度数为 2。设 *u* 是哈密顿回路上的顶点 H_i，其相邻的两个顶点为 H_{i-1} 和 H_{i+1}。而(H_{i-1}, H_i)和(H_i, H_{i+1})两条边都是多边形的边，将这两条边添加到一个附加图 *C* 中（附加图 *C* 一开始只有 *n* 个顶点，对应着多边形的顶点，顶点之间没有边相连）。将以 *u* 为顶点的三角形（即三角形 $H_iH_{i-1}H_{i+1}$）从多边形上砍掉之后，剩下的图形仍然是一个多边形。也就是说，可以把 H_i 从图 *G* 中删除，若 H_{i-1} 和 H_{i+1} 之间没有边相连，就添加一条新边（H_{i-1},H_{i+1}）（虚边），得到新图 *G*'。这时，新图 *G*'中仍唯一存在一条哈密顿

回路……H_{i-2} ,H_{i-1}, H_{i+1}, H_{i+2}……。图 G'和图 G 具有同样的性质，因此也至少存在一个度为 2 的点，令其为 H_j。那么(H_{j-1}, H_j)和(H_j, H_{j+1})这两条边有可能为多边形的边、多边形的对角线或者新添加的虚边，将其中多边形的边添加到图 C 中。然后按照同样的方法把 H_j 从图 G'删除，得到一个新图……。以此类推，不断地从新图中找到度为 2 的点，然后将其相应的两条边中是多边形的边添加入图 C 中，接着从图中删除这个点。如此反复，直到图中不存在度为 2 的点。然后把图中剩下的边中，是多边形的边的添加到图 C 中。最后，图 C 便是要求的原始多边形。



样例的模拟过程如下：





左边是图 C 的变化过 右边是图 C 的变化

图 2

还存在一个问题没有解决：如何判断移除的边是否是原多边形的边呢？首先考虑一下，一条边（H_i, H_j）什么时候才会被移除。一条边（H_i, H_j）当且仅当成为某个多边形的边时，它才有可能被移除。如果（H_i, H_j）是原始多边形的一条对角线，当且仅当哈密顿回路上从 H_{i+1} 到 H_{j-1} 的顶点都已经被删除，（H_i, H_j）才有可能成为新多边形的边。如何判断从 H_{i+1} 到 H_{j-1} 的顶点是否已经被删除呢？这时构造的附加图 C 就起到了作用！若从 H_{i+1} 到 H_{j-1} 的顶点已经被删除，则这些顶点在原多边形上相应的边都已经添加到图 C 中，H_i 到 H_j 在图 C 中一定存在一条路径。因此判断一条移除的边（H_i, H_j）是否是对角线，只需要判断在图 C 中 H_i 和 H_j 是否连通。值得注意的是，当图 C 中已经有了 n -1 条边时，剩下的那条边会被判断成对角线，但此时已经能够确定多边形顶点的标号序列了。

算法的大致轮廓已经清楚了，下面为算法选择合适的数据结构：

a. 图 G 的存储结构：由于图 G 是稀疏图，可以用邻接表存储，用来查找度为 2 的顶点与哪些边相连。还要用一个哈希表存下所有的边，用于查找任意两个点是否相连。设一条边为(u, v)，哈希表可以用 $u \times n + v$ 作为

关键字，哈希函数为 $f(u,v)=(u\times n+v)\bmod P$ ，P 为大质数。这样在保证查找复杂度仍然是 $O(1)$的情况下，存储空间比邻接矩阵小了很多。

b. 枚举度为 2 的结点：很容易想到用最小堆来找出度为 2 的顶点，不过这样的复杂度稍稍高了一点。由于顶点的度数只减少不增加，而且真正有效的度数只有 1 和 2，所以可以建立四个桶来替代堆。将顶点按度数分别放到四个不同的桶中：度数为 0、度数为 1、度数为 2、度数大于 2。在每个桶中用双向链接表存储下不同的结点。当一个顶点的度数被修改的时候，从原桶中删除，放到相应的桶中。在双向链表中的插入和删除结点，时间复杂度都是 $O(1)$。而因为每个结点的度数是不断减少的，所以每个结点最多进出每个桶一次。综上所述，在这些桶中查找和调整一个度为 2 的结点所需的时间复杂度为 $O(1)$。

c. 在图 C 中添加边，判断任意两点的连通性：可以采用并查集来实现边的添加（即将两个连通子图合并）和判断点的连通性（即判断两个点是否在同一个连通子图中）。添加一条边和判断一对点的连通性两种操作的均摊复杂度为 $O(\alpha(n))$。

下面是对算法二的总括：

算法二

1． 根据每个顶点的度数将 n 个顶点放到四个桶中。初始化附加图 C。

2． 如果存在一个度数为 2 的顶点 u，则接下来执行第 3 步，否则转步骤 6。

3． 在邻接表中找到和 u 相连的两个顶点为 v_1 和 v_2。

4． 首先判断(u,v_1)、(u,v_2)是否为虚边。若不是虚边，用并查集检查其是否为对角线。最后将是多边形的边插入到图 C 中。

5． 将顶点 u 标记为不可用。用哈希表检查边(v_1,v_2)是否存在图 G 中。如果边(v_1,v_2)不存在，则添加一条虚边(v_1,v_2)，v_1 和 v_2 的度数不变；否则将 v_1 和 v_2 的度数减一。若 v_1 或 v_2 修改后的度数不符合所在桶的性质，则进行调整。然后转到步骤 2。

6． 找到残图 G 中所有度数为 1 的顶点，一一检查这些顶点相应的边在附加图 C 中的连通性，把非对角线和虚边的边插入到图 C 中。

7． 遍历图 C 得到原始多边形顶点的标号顺序。

在每个顶点删除时，最多添加一条虚边，因此原始边和虚边的总数仍然是 $O(n)$的级别。现在分析一下时间复杂度：步骤 3 中，每个顶点最多查找一次，因此复杂度为边数，即 $O(n)$；步骤 4 和步骤 6 中，用并查集查找和插入的均摊时间复杂度为 $O(n\alpha(n))$；步骤 5 中，用哈希表查找一条边的复杂度为 $O(1)$，边的插入和桶的调整也是 $O(1)$，一共进行顶点数次，所以时间复杂度为 $O(n)$。综上所述，算法二的复杂度为 $O(n\alpha(n))$。

算法二的时间复杂度已经十分低了。但追求无止境的我们不能浅尝辄止，因为算法二仍存在需待改进的地方：编程复杂度实在太高了！算法二虽然已经较为充分的挖掘出问题的各种特性，但在利用这些特性时，是通过不断分解成子问题然后一一击破的方法来实现的。我们不禁提出一个问题：是否能找到一种综合的方法，将所有特性一起利用，使得解决方法既简便又高效呢？

3.3 DFS 树反映的问题特性

回到多边形所在的平面图上。若按深度优先来遍历平面图，当经过一条对角线时，平面图便被分成了两个部分。由于是深度优先遍历，那么在这两部分剩下的点中，有一部分的点会被优先遍历到。因此可以模仿 DFS 寻找块的算法，利用顶点的遍历顺序和 low 函数来判断边的性质。

首先定义一下 DFS 中需要用到的两个函数：

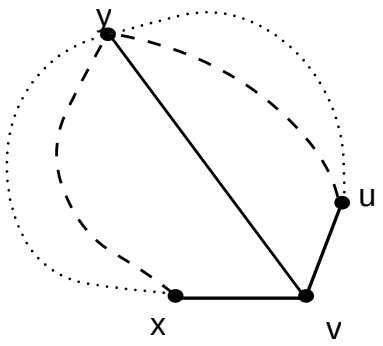
dfn[*v*]表示 *v* 是深搜中第几个被遍历到的。

$\mathrm{low}[v]=\min\{\mathrm{dfn}[v],\mathrm{low}[w_1],\mathrm{dfn}[w_2]\}$ 其中 w_1 表示 *v* 的儿子结点，w_2 表示 DFS 树中异于 *v* 的父亲结点的其他祖先结点。

接下来分情况讨论如何用 dfn 和 low 函数判断一条边的性质：

考虑 DFS 树上的一条边(*u*, *v*)，其中 *u* 是 *v* 的父亲结点。由于图 *G* 是一个块，因此每个点的儿子个数不会超过 2。根据 *v* 的度数分下面四种情况：

1．*v* 的儿子数为 0：意味着(*u*,*v*)是原始多边形的边。否则的话，*v* 所有的祖先结点都在(*u*,*v*)的一侧，而另一侧一定仍有点存在，与儿子数为 0 矛盾。令 *x* 是与 *v* 直接相连的祖先结点中 dfn 值最小的一个结点，那么(*x*,*v*)一定也是原始多边形的边。略证：如图 3-3，另(*x*,*v*)为多边形的边。*v* 不是



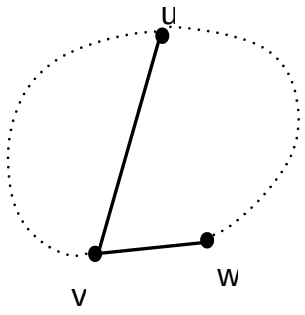


图 2 图 3

x 的父亲结点，说明 *x* 一定先于 *u* 被遍历到，而且 *x* 是 *u* 的祖先结点。也

就是说，x 到 u 存在一条路径，路径上的边都是由 DFS 树上的边组成。令 y 为异于 x 和 u，而与 v 相连的点。由于图 G 是平面图，那么 y 一定在 x 到 u 路径上。因此 y 一定后于 x 被遍历，即 dfn[y]小于 dfn[x]。所以 x 一定是与 v 相连的祖先结点中 dfn 值最小的。

2．v 的儿子数为 1，且 u 为 DFS 树的根：易知，(u, v)一定是多边形的边。

3．v 的儿子数为 1，且 u 不是 DFS 树的根：令 w 为 v 惟一的儿子。

若 low[w]大于等于 dfn[u]，则(u, v)为多边形的对角线。因为(u, v)将平面分成了两部分，w 和其子树结点是遍历不到另一部分的，如图 3-4。令 x 是与 v 直接相连的祖先结点中 dfn 值最小的一个结点，那么(x, v)一定是原始多边形的边（证明同情况 1 中的分析）。而与 v 相连的另一条多边形的边可以在遍历 w 时找到。

若 low[w]小于 dfn[u]，则(u, v)是多边形的边，而与 v 相连的另一条多边形的边可以在遍历 w 时找到。

4．v 的儿子数为 2，则与 v 相连的两条多边形的边可以在遍历儿子结点时找到。

考虑上述四种情况已经能够判断图 G 中所有边的性质了。因此只需要对图 G 进行一次深度优先遍历，就能确定多边形顶点的标号顺序。所以算法三的时间复杂度为线性的——$O(n)$！

3.4 小结

上述三种方法都是建立在同一个模型上的，不同的方法对模型性质挖掘的深度不同也就决定了不同的时间复杂度。算法一运用了定义法，利用对角线将

多边形划分成两部分的性质找到多边形中所有的对角线。而在算法二的设计过程中，发现了每次删除多边形的一个角时仍是一个多边形，也就是找到了子问题的相似性，这样就可以不断地缩小问题的规模。算法二还体现了一种化归的思想，将未知问题分成几个步骤（或者子问题），每个步骤都是我们熟知的，因此可以为每一步选择最好的算法。由于每个子问题都解决了，原问题也就迎刃而解。可以说方法二是一种分析法。算法三体现了一种综合思想，不是单独考虑每一条边的连通性，而是从全局考虑，发现了多边形的边和对角线在 DFS 树中的区别。因此算法三的设计一气呵成，体现了图论中一种“序”的优美性。

三种不同的方法也给了我们不同的启示：定义法告诉我们，当找不到问题思考的方向时，可以考虑从问题最基本的性质入手寻找突破口。分析法的好处是，分解后的子问题一般比原问题要简单，困难的是每一部分都会影响到最终算法的复杂度，因此每个部分的算法设计都要精益求精。综合法的运用需要有全局观，要能够发现最具代表性的模型特性。

四、总结

“模型”一词曾在无数论文中被提及，它是图论基本思想的精华，是解决图论问题的关键。建立模型要求我们熟悉经典模型，同时又要求我们能够勇于探索，大胆创新，敢于跳出经典模型的框框。利用模型的特性需要我们独具慧眼，能够抓住问题的本质，击中问题的要害。

同时文章中还介绍了三种解决问题的方法：定义法、分析法和综合法。这些方法和“模型”一样，具有解决信息学奥赛问题的普适性。它们不仅仅是一种方法，更是一种思维的方式和思考的角度。灵活地运用和掌握它们，有利于我们研究算法、探索科学。

例题二的解析过程，不仅仅体现了图论的基本思想，同时还展现了算法与数据结构的完美结合，以及算法的优化思想。算法与数据结构的结合是优秀程序设计的必然要求，是知识融会贯通的体现。优化是为了进一步的完善程序设计，是一种不断进取的精神。这些同样是今后科学研究的必备素养。

本文或许没有完美的概括图论的基本思想和方法，仅仅是作者对图论的一点理解和想法。但作者希望本文对读者有一定的启发，使读者能够对图论基本算法及方法进行深入思考和总结。所谓万变不离其宗，只有最基本的思想和方法才是解决问题的不二法门。

【参考文献】

[1]《Introductory Combinatorics (3rd Edition)》（美）Richard A. Brualdi 著

[2]《算法艺术与信息学竞赛》 刘汝佳 黄亮 著

[3] Poland Olympiad of Informatics 2002 Stage III：Skiers

[4] ACM Central European Programming Contest 2001 Problems and Solutions

【感谢】

衷心感谢向期中老师在学习上对我的指导和帮助

衷心感谢栗师同学在例题一上给我的启发

衷心感谢胡伟栋和王俊同学对我的大力帮助

由衷感谢周戈林、肖湘宁两位同学对我的论文提出宝贵的意见

【附录】

1．Dilworth 定理的证明：

易知 M 一定大于等于 m，因为最长反链中的元素一定分在不同的链中。因此只需要证明一定存在 M 小于等于 m。接下来，通过对 E 的元素个数 n 的归纳法证明之。

当 n=1 时，定理显然成立。

当 n>1 时，分两种情况讨论：

情况 1：存在一条大小为 m 的反链 $A=\{a_1,a_2,\cdots,a_m\}$，它既不是 P 中的所有极大元的集合，也不是所有极小元的集合。

现定义 $A^-=\{x\in P\mid \exists_i x\le a_i\}$，即 A^-中任一个元素 x 在 A 中能找到一个元素 a_i，使得 $x\le a_i$。类似地，定义 $A^+=\{x\in P\mid \exists_i x\ge a_i\}$。这样直观上的看，$A^-$是所有 A“下方”的元素组成，而 A^+是所有 A“上方”的元素组成。而且还有下列性质成立：

1）由于至少存在一个极大元不在 A 中，所以这个极大元也不在 A^-中。因此 $A^-\neq P$，A^-中的元素个数小于 P 的元素个数 n。

2）由于至少存在一个极小元不在 A 中，所以这个极小元也不在 A^+中。因此 $A^+\neq P$，A^+中的元素个数小于 P 的元素个数 n。

3）$A^+\cap A^-=A$。反设存在一个 x 在 $A^+\cap A^-$中，而不属于 A。根据反设，A 中存在两个元素 a_i 和 a_j，满足 $a_i\le x\le a_j$，这与 A 是一条反链矛盾。因此性质 3 成立。

4）$A^+\cup A^-=P$。反设存在一个 x 在 $A^+\cup A^-$中，而不在 P 中。根据反设，对于 A 中的任一个元素 a_i，既不满足 $a_i\le x$，也不满足 $a_i\ge x$。也就是

说 x 和 A 中的所有元素都是不可比的，那么 $A \cup x$ 是一条比 A 更长的反链，这与 A 是 P 中最长的反链矛盾。因此性质 4 成立。

因为 A^+、A^-中的元素个数小于 n，根据归纳假设，A^+、A^-一定能够划分成 m 条链。设 A^+划分成 $C_1^+, C_2^+ \cdots C_m^+$，且 $a_i \in C_i^+$。类似地，设 A^-划分成 $C_1^-, C_2^- \cdots C_m^-$，且 $a_i \in C_i^-$。显然的有，对于所有 i，a_i是C_i^+中的极小元。因为若 C_i^+中的极小元不是 a_i 而是 x，那么根据 A^+的定义，存在一个 $a_j \le x$，则有 $a_j \le x \le a_i$，这与 A 是反链矛盾，所以 a_i 是极小元。同样地，a_i是C_i^-中的极大元。a_i 是 C_i^+中结尾的那些元素，是 C_i^-开始的那些元素，将 C_i^+和 C_i^-接在一起形成了 m 条链，它们是 P 的一个划分。

情况 2：最多存在两条大小为 m 的反链，即所有的极大元集合和极小元集合中任一个或两个。令 x 是极小元，而 y 是极大元且 x ≤ y(x 可以等于 y)。此时，$P - \{x, y\}$ 的反链的最大的大小为 m – 1。根据归纳假设，$P - \{x, y\}$ 可以被划分成 m – 1 条链。这些链和链{x, y}一起构成了 P 的一个划分。

2．定理 1.2 的证明：

反设最长反链 A 中存在两条相邻的边不再同一个域中，令这两条边为 a 和 b，且 a 在 b 的左边。一条边最多属于两个不同的域。有 3 种情形：

1) a 是某个域 F_1 左边界上的一条边，那么 F_1 右边界上的一条边 c 与 a 和 b 都是不可比的，那么 $A \cup \{c\}$ 是一条更长的反链，矛盾。

2）b 是某个域 F_2 右边界上的一条边，那么 F_2 左边界上的一条边 d 与 a 和 b 都是不可比的，那么 $A \cup \{d\}$ 是一条更长的反链，矛盾。

3）a 是域 F_1 右边的那条路径，b 是域 F_2 左边的那条路径。设 $a=(u_1,v_1)$，$b=(u_2,v_2)$。由于平面图中的最高点 v_h 到 u_1 和 u_2 都至少存在一条路径，因此

这些路径之间至少存在一个公共点既能够到达 u_1 也能够到达 u_2，令这些公共点中纵坐标最低的点为 u_h。类似地，设既能够被 v_1 到达也能被 v_2 到达的公共点中纵坐标最高的点为 u_l。由于 a 和 b 不在同一个域中，因此路径 1（$u_h - u_1 - v_1 - u_l$）和路径 2（$u_h - u_2 - v_2 - u_l$）之间至少存在另一条路径 3，这条路径要么是从路径 1 连到路径 2，要么是从路径 2 连到路径 1。因此路径 3 上一定存在一条边 e，e 和 a、b 都是不可比的，那么 $A \cup \{e\}$ 是一条更长的反链，矛盾。

因此反链中相邻的两条边一定是在同一个域中。

3．滑雪者原题

Skiers

On the south slope of Bytemount there is a number of ski tracks and one ski lift. All the tracks run from the top station of the ski lift to the bottom one. Every morning a group of ski lift workers examines the condition of the tracks. They together take the lift up to the top station, and next each of them skis down along a chosen track to the bottom station. Each worker skis down only once. The tracks of the workers may partially be the same. Each track examined by any of the workers leads always downwards.

The map of ski tracks consists of a network of clearings connected by cuttings in the forest. Each clearing lies on different height. Any two

clearings may be directly connected by at most one cutting. Skiing down from the top to the bottom station one can choose a track to visit any one clearing (although probably not all of them in a single run). Ski tracks may cross only on clearings and do not run through tunnels nor on bridges.

Task

Write a program which:

- reads from the text file nar.in the map of ski tracks,
- computes the minimal number of workers to examine all the cuttings between the clearings,
- writes the result to the text file nar.out.

Input

In the first line of the input file nar.in there is one integer *n* equal to the number of clearings, 2<=*n*<=5 000. The clearings are numbered from 1 to *n*.

Each of the successive *n*-1 lines contains a sequence of integers separated by single spaces. The integers in the (*i*+1)-st line of the file specify which clearings the cuttings from the clearing number *i* lead down to. The first integer *k* specifies the number of those clearings. The

successive *k* integers are their numbers ordered in the direction from west to east, according to the arrangement of the cuttings leading to them. The top station of the ski lift lies on the clearing number 1, and the bottom one on the clearing number *n*.

Output

In the first and only line of the output file nar.out there ought to be exactly one integer - the minimal number of workers that are able to examine all the cuttings in the forest.

4.爱丽丝和鲍勃原题

Problem ：Alice and Bob

this is a puzzle for two persons, let's say Alice and Bob. Alice draws an n-vertex convex polygon and numbers its vertices with integers 1; 2;…; n in an arbitrary way. Then she draws a number of noncrossing diagonals (the vertice- s of the polygon are not considered to be crossing points). She informs Bob about the sides and the diagonals of the polygon but not telling him which are which. Each side and diagonal is specified by its ends. Bob has to guess the order of the vertices on the border of the
polygon. Help him solve the puzzle.

Example

If n = 4 and (1,3), (4,2), (1,2), (4,1), (2,3) are the ends of four sides and one diagonal then the order of the vertices on the border of this polygon is 1, 3, 2, 4 (with the accuracy to shifting and reversing).

Task

Write a program which for each data set:
·reads the description of sides and diagonals given to Bob by Alice,
·computes the order of the vertices on the border of the polygon,
·writes the result.

Input

The first line of the input contains exactly one positive integer d equal to the number of data sets,1 ≤ d ≤ 20. The data sets follow.

Each data set consists of exactly two consecutive lines.

The first of those lines contains exactly two integers n and m separated by a single space, 3 ≤ n ≤10 000, 0 ≤ m ≤ n ≤ 3. Integer n is the number of vertices of a polygon and integer m is the number ofits diagonals, respectively.

The second of those lines contains exactly 2(m+n) integers separated by single spaces. Those are ends of all sides and some diagonals of the polygon. Integers aj; bj on positions 2j -1 and 2j, $1 \le j \le m+n$,$1 \le a_j \le n$, $1 \le b_j \le n$, $a_j \ne b_j$, specify ends of a side or a diagonal. The sides and the diagonals can be given in an arbitrary order. There are no duplicates.

Alice does not cheat, i.e. the puzzle always has a solution.

Output

The output should consist of exactly d lines, one line for each data set.

Line i, 1 ≤ i ≤ d, should contain a sequence of n integers separated by single spaces | a permutation of 1; 2; …; n, i.e. the numbers of subsequent vertices on the border of the polygon from the i-th data set; the sequence should always start from 1 and its second element should be the smaller vertex of the two border neighbours of vertex 1.

遗传算法应用的分析与研究

福州八中 钱自强

【摘要】

随着科技水平的不断发展，人们在生产生活中遇到的问题也日益复杂，这些问题常常需要在庞大的搜索空间内寻找最优解或近似解，应用传统算法求解已经显得相当困难。而近年来，生物学的进化论被广泛地应用于工程技术、人工智能等领域中，形成的一类有效的随机搜索算法——进化算法，有效的解决了诸多生产生活中的难题而显得越来越流行。

本文的首先将介绍进化算法的原理以及历史使大家对进化算法有一个初步的了解，其次将详细介绍应用遗传算法解题的步骤，并提出有效改进和应用建议。紧接着通过一个 NP 难题的优化实例让大家对遗传算法有更深刻的了解，最后通过数据分析证明其方法的有效性。

【关键词】

人工智能；进化算法；遗传算法（GA）；多目标最小生成树

目录

一.进化算法理论

1．1 进化算法概述

从远古时代单细胞开始，历经环境变迁的磨难，生命经历从低级到高级，从简单到复杂的演化历程。生命不断地繁衍生息，产生出具有思维和智能的高级生命体。人类得到生命的最佳结构与形式，它不仅可以被动地适应环境，更重要的是它能够通过学习，模仿与创造，不断提高自己适应环境的能力。

进化算法就是借鉴生物自然选择和遗传机制的随机搜索算法。进化算法通过模拟“优胜劣汰，适者生存”的规律激励好的结构，通过模拟孟德尔的遗传变异理论在迭代过程中保持已有的结构，同时寻找更好的结构。作为随机优化与搜索算法，进化算法具有如下特点：进化算法不是盲目式的乱搜索，也不是穷举式的全面搜索，它根据个体生存环境即目标函数来进行有指导的搜索。进化算法只需利用目标的取值信息而不需要其他信息，因而适用于大规模、高度非线性的不连续、多峰函数的优化，具有很强的通用性；算法的操作对象是一组个体，而非单个个体，具有多条搜索轨迹。

1．2 遗传算法

遗传算法（Genetic Algorithm）是进化算法的一个重要分支。它由 John Holland 提出，最初用于研究自然系统的适应过程和设计具有自适应性能的软件。近来，遗传算法作为问题求解和最优化的有效工具，已被非常成功地应用与解决许多最优化问题并越来越流行。

遗传算法的主要特点是群体搜索策略和群体中个体之间的信息互换,它实际上是模拟由个体组成的群体的整体学习过程,其中每个个体表示问题搜索空间中的一个解点.遗传算法从任一初始的群体出发,通过随机选择，交叉和变异等遗传操作,使群体一代代地进化到搜索空间中越来越好的区域,直至抵达最优解点.

遗传算法和其它的搜索方法相比,其优越性主要表现在以下几个方面：首先,遗传算法在搜索过程中不易陷入局部最优,即使在所定义的适应度函数非连续.不规则也能以极大的概率找到全局最优解,其次,由于遗传算法固有的并行性,使得它非常适合于大规模并行分布处理,此外,遗传算法易于和别的技术(如神经网络.模糊推理.混沌行为和人工生命等)相结合,形成性能更优的问题求解方法.

二.遗传算法

2．1　遗传算法的基本流程

一个串行运算的遗传算法通常按如下过程进行：

(1) 对待解决问题进行编码；t:=0

(2) 随机初始化群体 X(0):=$(x_1, x_2, \cdots, x_n)$；

(3) 对当前群体 X(t)中每个染色体 x_i 计算其适应度 F (x_i)，适应度表示了该个体对环境的适应能力，并决定他们在遗传操作中被抽取到的概率；

(4) 对 X(t)根据预定概率应用各种遗传算子，产生新一代群体 X(t+1)，这些算子的目的在于扩展有限个体的覆盖面，体现全局搜索的思想；

(5) t:=t+1(新生成的一代群体替换上一代群体)；如果没有达到预定终止条件则继续(3)。

2．2 遗传算法中各重要因素分析

▲ 编码理论

遗传算法需要采用某种编码方式将解空间映射到编码空间（可以是位串、实数、有序串）。类似于生物染色体结构，这样容易用生物遗传理论解释，各种遗传操作也易于实现。编码理论是遗传算法效率的重要决定因素之一。二进制编码是最常用的编码方式，算子处理的模式较多也较易于实现。但是，在具体问题中，根据问题的不同，采用适合解空间的形式的方式进行编码，可以有效地直接在解的表现型上进行遗传操作，从而更易于引入相关启发式信息，往往可以取得比二进制编码更高的效率。

▲ 染色体

每个编码串对应问题的一个具体的解，称为染色体或个体。一个染色体可以通过选用的编码理论与问题的一个具体的解相对应，一组固定数量的染色体则构成一代群体。群体中染色体可重复。

▲ 随机初始化

随机或者有规律（如从一个已知离解较近的单点，或者等间隔分布地生成可行解）生成第一代群体。一次遗传算法中有目的采用多次初始化群体会使算法拥有更强的搜索全局最有解的能力

▲ 适应度

一个染色体的适应度是对一个染色体生存能力的评价，它决定了该染色体在抽取操作中被抽取到的概率。估价函数是评价染色体适应度的标准，常见的估价函数有：直接以解的权值（如 01 背包问题以该方案装进背包物品的价值作

为其适应度），累计二进制串中 1/0 的个数（针对以二进制串为编码理论的遗传算法），累加该染色体在各个目标上的得分（针对多目标最优化问题，另外，对于此类问题，本文提出了一种更有效的估价函数）。

▲ 遗传算子

遗传算子作为遗传算法的核心部分，其直接作用于现有的一代群体，以生成下一代群体，因此遗传算子的选择搭配，各个算子所占的比例直接影响遗传算法的效率。一个遗传算法中一般包括多种遗传算子，每种算子都是独立运行，遗传算法本身只指定每种算子在生成下一代过程中作用的比例。算子运行时从当前这代群体中抽取相应数量的染色体，经过加工，得到一个新的染色体进入下一代群体。

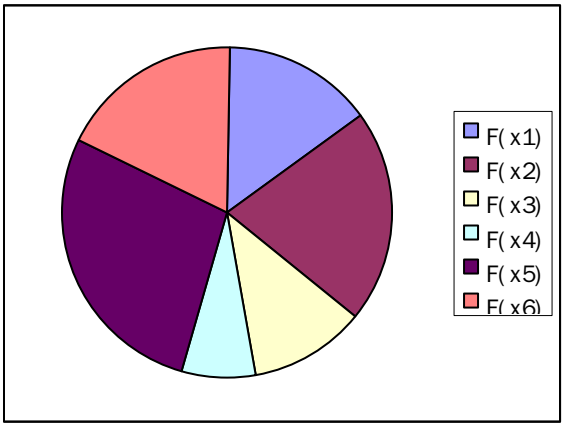
下面列出几种常见的遗传算子：

- 保持算子：抽取 1 个染色体，直接进入下一代。该算子使算法能够收敛。
- 交配算子：抽取 2 个染色体，交换其中的某些片段，选择适应度高的（或 者都选），进入下一代群体。该算子使得遗传算法能够利用现有的解寻求更优的解。
- 变异算子：抽取 1 个染色体，对其进行随机的改变，进入下一代群体。该算子使得算法可以跳出局部最优解，拥有更强搜索全局最有解的能力，防止陷入局部搜索，该算子的概率不可过高，否则将引起解的发散，使得算法无法收敛。

▲ 抽取

抽取操作是遗传算法中一个重要基本操作，作用是按照“优胜劣汰”的原则根据各个染色体的适应度从当前这代群体中挑选用于遗传算子的染色体。通常采用的手段是偏置转盘：

设算法中群体数量为 $population$ ，首先计算当代群体的各染色体适应度之和 $S(t)=\sum_{i=1}^{population}F(x_i)$。将 $1\sim S(t)$ 内的整数划分成 $population$ 个区间段，每个染色体所占的区间段的长度既是它的适应度。这样，随机产生一个 $1\sim S(t)$ 的整数，抽取该点所属区间段相对应的染色体，就可以保证任意一个染色体 x_i 在抽取操作中被抽取到的概率为 $\frac{F(x_i)}{S(t)}$。



▲ 终止条件

遗传算法的终止条件用于防止遗传算法无止境的迭代下去，一般限制条件可以设为达到指定的迭代次数后终止，或当解的收敛速度低于一定值时自动终止。当遗传算法达到终止条件时，遗传算法结束，并按照要求返回中途最优的一个染色体（或所有满足条件的非劣最优解）

2．3 重要参数设置

在应用遗传算法解决问题的时候，往往需要根据实际情况的不同，对不同问题使用不同的遗传参数。在大规模的问题上，一次遗传算法的不同时期也可以设置不同的遗传参数。对遗传算法效率影响较大的参数如下：

群体大小：一代群体中染色体的数量，群体大小越大所能容纳的染色体品种也越多，越有利于搜索全局最有解，但是会下降收敛的速度，所需的时间也更多。

迭代次数：最多更新群体的次数，迭代次数的增加可以使得解收敛更精确但是所需的时间也越多，如果时间允许，采用多次初始化群体的操作要比设置很大的迭代次数来得更高效些。

保持率：保持算子所占的比例，通常不超过 70%

交配率：交配算子所占的比例，通常不超过 50%

变异率：变异算子所占的比例，通常不超过 1%

三.遗传算法在多目标最小生成树问题中的应用

生活中的很多问题，例如道路铺设，电网架设，网络构设等，其实都可以归结到最小生成树模型，经典的 Prim 算法和 Kruskal 算法都可以解决该问题，算法的时间复杂度都是线形的，但是现实生活中的问题往往没有那么简单，一条边上可能不只带一个权，例如一条公路的铺设道路长度还要考虑环境和人文因素，电网架设时除了考虑线路费用还要考虑架设难度，一个网络连接除了考虑网络延时还要考虑传输稳定性和安全性等，于是问题就转化为求解多目标的最小生成树问题的非劣最优解，这个问题是 NP 难的，Prim 算法，Kruskal 算法等常规算法就显得无能为力，搜索算法的复杂度却又过高。

3．1 多目标最小生成树问题

3．1．1　最小生成树

在图论中，一个无回路（圈）的连通子图称为树。设 $T=(N, E_T)$ 是 $|N|\geq 3$ 的一个图，则下列关于树的 6 个定义是等价的：①T 连通且无回路；②T

有|***N***|－1 条边且无回路；③***T*** 连通且有|***N***|－1 条边；④***T*** 连通且每条边都是割边；⑤***T*** 的任两点间都有唯一的路相连；⑥***T*** 无回路，但在任一对不相邻的点之间加连一条边则构成唯一的回路。

设有一无向图$G=(N,E,W)$，其中$W=\sum_{e\in E}w_e$为权函数，若树$T=(N,E_T,W_T)$包含了图 G 的所有顶点，则树 T 为 G 的一个生成树，其中$W_T=\sum_{e\in E_T}w_e$为树 T 的权。图 G 的生成树不唯一，权和最小的生成树称为 G 的最小生成树。

3．1．2 多目标最小生成树

而当图中每个边的权值有多个时，相应的问题称为多目标最小生成树问题。该类问题的目标是找出问题的所有非劣最优生成树，显然该类问题即使不加任何约束条件也属于 NP 问题。

设有一个连通的无向图 *G*=（*V*，*E*），其中 *V*= {v_1,v_2,…,v_n} 是一个有限集合，代表图 *G* 的顶点，$E=\{e_{1,2},e_{1,3},...,e_{i,j},...,e_{n-1,n}\}$,

$e_{i,j}=\begin{cases}1,\ v_i,v_j\text{之间有边}\\0,otherwise\end{cases}$，$(i=1,2,...,n-1;\ j=i+1,...,n)$ 为图 *G* 的边的集合，若边 e_{ij} 存在则该边有 *m* 个值为正数的属性与之对应，用 $\boldsymbol{w}_{i,j}=\{w_{i,j}^1,w_{i,j}^2,...,w_{i,j}^m\}$ 表示，实际问题中 $w_{i,j}^k(k=1,2,...,m)$ 可以是距离、代价等等。

令 $\boldsymbol{x}=(x_{1,2}x_{1,3}...x_{i,j}...x_{n-1,n}),\ x_{i,j}=\begin{cases}1,\ if\ e_{i,j}=1\text{且 被选中}\\0\ ,otherwise\end{cases}$，

$i=1,2,...,n-1;\ \ j=i+1,...,n$ ，表示图 *G* 的一棵生成树。***X*** 为所有 ***x*** 的集合，则多目标最小生成树问题描述如下：

$$\min \quad f_1(\mathbf{x}) = \sum w_{i,j}^1 x_{i,j};$$
$$\min \quad f_2(\mathbf{x}) = \sum w_{i,j}^2 x_{i,j};$$
$$\ldots$$
$$\min \quad f_m(\mathbf{x}) = \sum w_{i,j}^m x_{i,j}; \quad i = 1,2,..n-1, j = i+1,...,n$$

其中 $f_i(\boldsymbol{x})$ 为问题中需要最小化的第 *i* 个目标。

与一般的最小生成树问题相比，多目标最小生成树问题只是目标函数不止一个，然而正是因为目标不止一个而且这多个目标之间是经常是相互冲突的，因此无法使用经典的最小生成树算法通过找出具有最小权值的边逐步生成最小树，而如果将多个目标转换成单个目标再应用经典算法，则只能求出一个解，而不是找到问题的一组非劣最优解，而且多目标到单目标之间的转换对决策者来也是一个难题。

3．2 应用遗传算法解决多目标最小生成树问题

3．2．1 编码设计

Cayley 证明了对于一个完全图 G，连接所有 n 个顶点的树有 n^{n-2} 棵。为此 Prüfer 建立了一个这些树与从 n 个数取 n-2 个数的所有组合之间的一一对应关系，即如果对完全图中所有顶点从 1 到 n 开始编号，则任意一个在从 1 到 n 的 n 个数中取 n-2 个数的组合都与唯一的一棵生成树相对应。

本文对生成树的编码采用基于以上的一一对应建立起来的 Prüfer 数编码机制，把每一棵树与一个长度为 n-2 的数字串对应，而对于任意一个长度为 n-2 的数字串也与唯一的一棵生成树相对应，生成树到数字串的编码与数字串到生成树的解码的详细证明可参考相关文献，本文这里只作简要描述。

★编码过程

- ▲编码串初始为空串
- ▲令 *j* 为树中编号最小的叶节点；
- ▲找到唯一于 *j* 相邻的点 *i*，把 *i* 加入编码串的最右端
- ▲把 *j* 以及连接 *i* 和 *j* 的边从树中删除，这时候树只有 *n*-1 个顶点
- ▲重复以上 3 个步骤直到树中只剩下一条边这时候得到的编码串即为相应树的 Prüfer 编码

★解码过程

- ▲设 *P* 为编码串，$\overline{P}$ 为图的顶点编号不出现在 *P* 中的顶点的集合；
- ▲设 *i* 为 $\overline{P}$ 中编号最小的顶点，*j* 为 *P* 中最左端的顶点，则将连接 *i* 和 *j* 的边加入到树中，然后分别把 *i* 和 *j* 从 *P* 和 $\overline{P}$ 中删除，如果 *P* 中不在出现顶点 *j* 则把 *j* 加入到 $\overline{P}$ 中
- ▲重复以上步骤，直到 *P* 为空；
- ▲当 *P* 为空串时，$\overline{P}$ 中刚好剩下两个顶点，将连接这两个顶点的边加入到树中，最后构成的树即为与最初 *P* 对应的生成树。

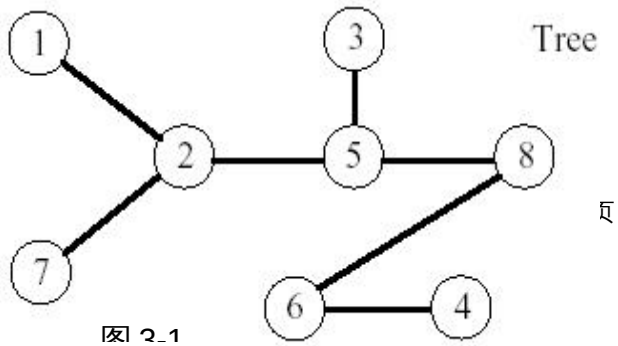


图 3-1

例如，图 3-1 则为一棵生成树以及其相对应的 Prüfer 编

码图 3-2。

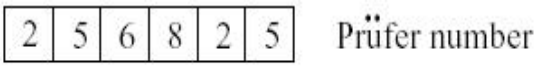


图 3-2　　显然 Prüfer 编码是生成树的一个有效表示方式，应用该编码方式，可以很容易地随机生成一棵生成树，而且 Prüfer 数编码串的每个位置的信息量又相对均匀因此很适合遗传操作。

3．2．2　估价函数设置

定义估价函数 g(x)为 $\left\lfloor \sum_{i=1}^{k} \left(\frac{\min[i]}{f_i(x)} \times 100 \right)^2 \right\rfloor$，　$f_i(x)$ 表示当前的染色体在目标 *i* 的费用情况，min[i]表示截止到上一代为止，产生的所有染色体在目标 *i* 的费用的最小值。

本文提出的这样定义比起常见的直接累加各目标上的权值的好处在于，其不仅很好的体现了一个染色体在各个目标上的优势，与此同时还避免了由于每个目标的取值范围不同或者取值的整体趋势不同而造成的某些个体在某些目标的优势无法被体现，使得算法能够适应现实生活中各类问题。

3．2．3 遗传操作定义

★交配算子

交叉算子我们使用的是小片段等位交叉算子，随机在两个染色体中抽取等位的，长度不超过 2 的片段进行互换，然后选择适应度较高的进入下一代，具体操作方法如下图 3-3 所示：

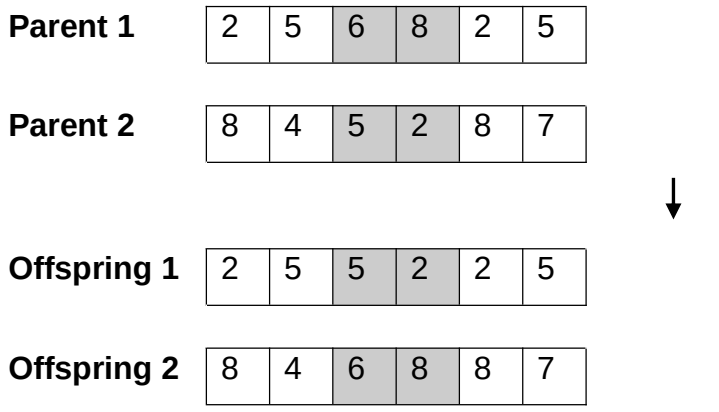


图 3-3 小片段等位交叉算子示意图

★变异算子

变异算子采用常规的单点变异，即随机生成一个 1 到 *n* 之间的数替换 Prüfer 数编码串中的某一位，如图 3-4 所示。

图 3-4 单点变异示意图

可以看出单点变异也可以很大程度上保留了原染色体的性质。

3.3　测试

我们将保持率定为 54%，交配率定为 45%，变异率定为 1%，并且根据数据不同对迭代次数和群体大小进行调整，将其于原始搜索算法进行对比（注：为了能够直观看出遗传算法的近似程度，所有数据采用 2 目标）。

*测试环境：*P4 2.0GhzA 256MDDR WinXp Delphi 7.0

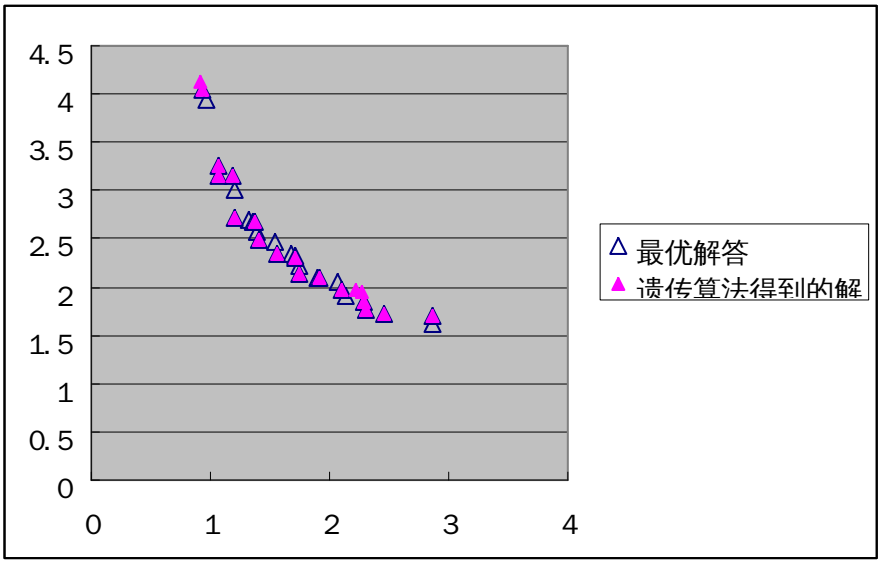
3.3.1 小规模经典测试数据列表

数据文件编号	数据信息	搜索算法表现		遗传算法表现		
		耗时	正确性	参数选择	耗时	正确性
E1	N=3 K=2 Ans=2	0s	完全正确	L=20 P=10	0s	完全正确
E2	N=4 K=2 Ans=6	0s	完全正确	L＝100 P=30	0s	完全正确
E3	N=5 K=2 Ans=12	0s	完全正确	L=1000 P=100	2s	完全正确

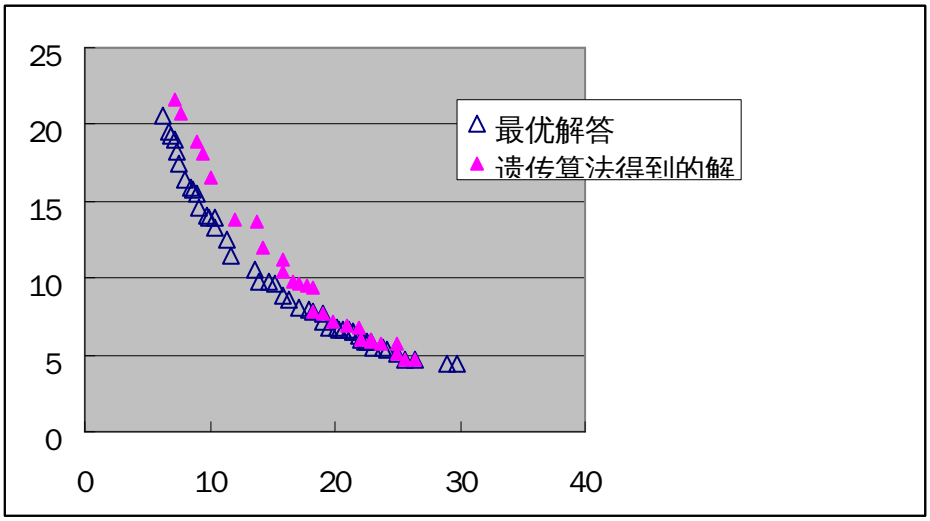
3.3.2 大中规模随机测试数据列表

数据文件编号		H1	H2
数据规模		N=10 K=2	N=11 K=2
搜索算法表现		22 分钟	6 小时 20 分钟
遗传算法	参数选择	L=20000 P=400	L=30000 P=400

表现	耗时	116s	296s
	正确性	算法得到了一组十分近似的解，参见图示	算法得到了一组比较近似的解，参见图示



测试数据 h1 的解答分析



测试数据 h2 的解答分析

3．3．3 特殊测试数据分析

对于 2 目标最优化考虑到现实生活中问题的 2 个目标的费用之间可能具有的一些关系，例如当设备相似时某个网络链接的速度相对较高，那么它的稳定性一定较低，因此我们对正相关和反相关两种特殊情况进行了测试，在这里先给出正相关与反相关的简单定义：

【正相关】

如果对于图中的任意两条边 $e_{i1,j1}, e_{i2,j2}$ 满足如果 $W^1_{i1,j1} \geq W^1_{i2,j2}$ 成立，那么 $W^2_{i1,j1} \geq W^2_{i2,j2}$ 成立，则称这张图的权为正相关的。

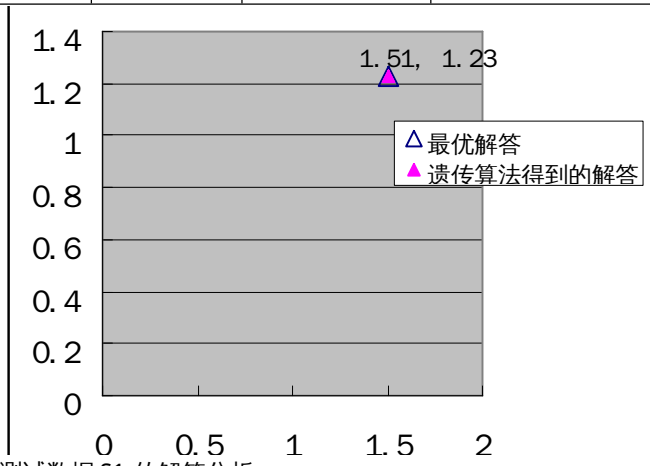
【反相关】

类似的定义，如果对于图中的任意两条边 $e_{i1,j1}, e_{i2,j2}$ 满足如果 $W^1_{i1,j1} \geq W^1_{i2,j2}$ 成立，那么 $W^2_{i1,j1} \leq W^2_{i2,j2}$ 成立，则称这张图的权为反相关的。

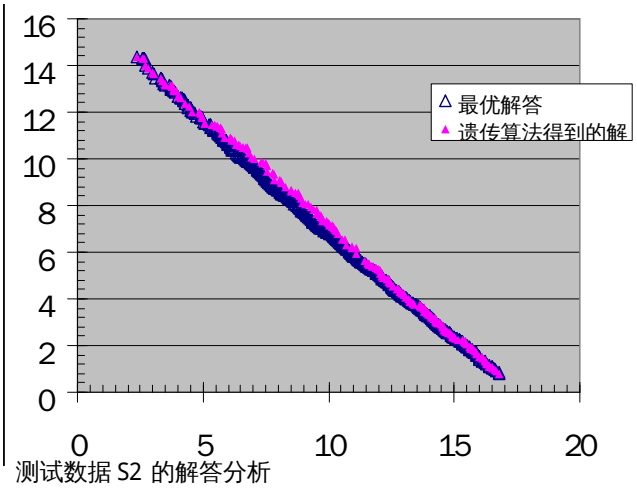
特殊数据测试表

数据文件编号		**S1**	**S2**	**S3**
数据信息	规模	N=10 K=2	N=10 K=2	N=15 K=2
	特殊性	正相关	反相关	正相关
搜索算法表现		15 分钟	28 分钟	600 年（估计）
遗传算法	参数选择	L=2000 P=400	L=20000 P=400	L=20000 P=400

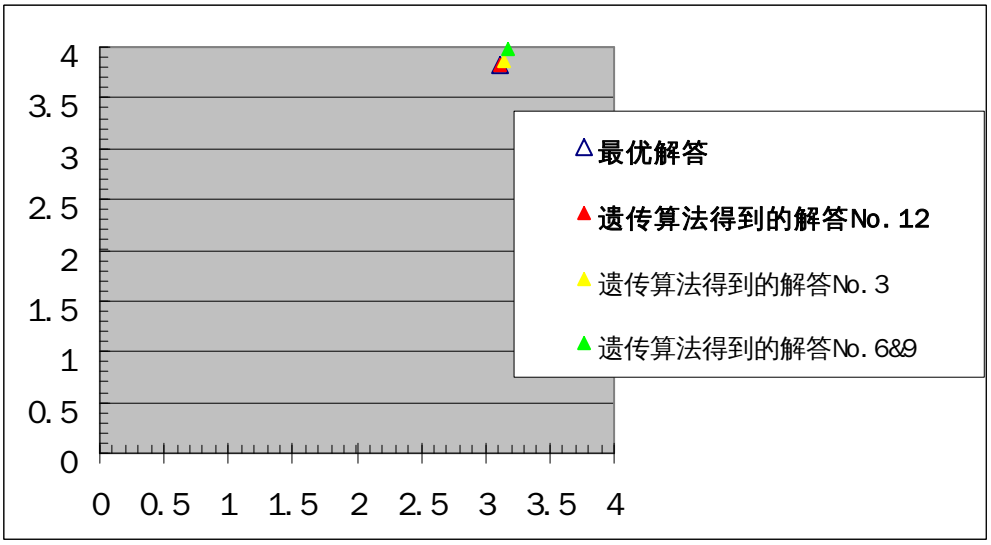
表现	耗时	11 秒	304s	114 秒*15 (15 次运行)
	正确性	该算法在 5 次执行有内 80％（4 次）的概率得到该最优解	算法得到了一组比较相似的解，参见图示	该程序在第 12 次运行的时候得到了我们的最优解(3.11,3.83)，并且值得一提的是在第 3 次运行时就得到一个相当接近的近似解(3.14 3.86)，并且在第 6 次和第 9 次都出现了(3.18,3.99)这个不错的解



测试数据 S1 的解答分析



测试数据 S2 的解答分析



测试数据 S3 的解答分析

四．结束语

本文主要介绍了遗传算法时的一些基本知识和一些使用心得，以及通过测试结果让大家看到了遗传算法在解决组合优化类问题有着和其他算法无法比拟的强大优势。它的特点就是可以在较短的时间内，得到比较令人满意的解，而且算法相对简明。对于现实生活中的大量常规算法无法解决问题，遗传算法都有着良好的应用前景。

遗传算法不仅一种算法，更是一种思想。在搜索中通过灵活的运用进化的思想来解决问题，往往能够收到事半功倍的效果。目前遗传算法在信息学竞赛中还不是那么普遍。本文的目的就是希望越来越多的信息学爱好者了解遗传算法，了解进化算法的思想。

由于作者能力有限，文中难免有所疏漏，欢迎来信指正。

参考文献

陈志平 徐宗本　　《计算机数学》　科学出版社

周 明 孙树栋　《遗传算法原理及应用》 国防工业出版社

邵军力 张 景 魏长华 《人工智能基础》 电子工业出版社

附录：

遗传算法求解多目标最小生成树问题程序源代码：

```
Program Ga;
const
    MaxN         =    50;
    MaxK         =     5;
    PopulationMax =   1000;
    Maxans       =   1000;
       Population    =   40;
      Live         = 2000;
      Change       =    45;
      Suddenly     =     1;
 type code=array[1..MaxN]of integer;
     vector=array[1..Maxk]of real;
     note=record
           v    :vector;
           data :code;
           power:longint;
          end;
     Group=array[1..PopulationMax]of note;
var
  inf,ouf:text;
  d:array[1..MaxK,0..MaxN,0..MaxN]of real;
  ans,ans2:array[1..Maxans]of vector;
  ansb,ansb2:array[1..maxans]of code;
  sn,n,k:longint;maxv:vector;
  population,live,change,suddenly:longint;
   function Genetic_Uncode(s:code):vector;
    var temp:vector;uu:boolean;
        i,j,l,r:longint;
```

```
    s2:code;Se:set of 1..maxn;
  function get:integer;
   var i:longint;
   begin
    for i:=1 to n do
     if i in Se then begin Se:=Se-[i];get:=i;exit;end;
   end;
  begin
   Se:=[1..n];
   FOR I:=1 to k do temp[i]:=0;
   for i:=1 to n-2 do Se:=Se-[s[i]];
   for i:=1 to n-2 do
    begin
     l:=get;
     for j:=1 to k do
      begin
       if (l<=0)or(l>n) or (s[i]<=0 )or(s[i]>n) then
         begin L:=L+1; end;
       temp[j]:=temp[j]+d[j,l,s[i]];
      end;
     uu:=true;
     for j:=i+1 to n-2 do if s[j]=s[i] then begin uu:=false;break;end;
     if uu then Se:=Se+[s[i]];
    end;
   l:=get;r:=get;
   for j:=1 to k do
     temp[j]:=temp[j]+d[j,l,r];
   Genetic_Uncode:=temp;
  end;
procedure Input_Initial;
```

```
 var s,i,j:longint;tempcode:code;
  begin
   readln(inf,n,k);
   for s:=1 to k do
    for i:=1 to n do
     for j:=1 to n do
      read(inf,d[s,i,j]);
   for i:=1 to n-2 do tempcode[i]:=random(n)+1;
   maxv:=Genetic_uncode(tempcode);randomize;
   sn:=1;ans[1]:=maxv;ansb[1]:=tempcode;
    close(inf);
  end;
function find(v1,v2:vector):longint;
 var i:longint;
     check:boolean;
 begin
  check:=true;
  for i:=1 to k do
   if v1[i]<v2[i] then begin check:=false;break;end;
  if check then begin find:=1;exit;end;
  check:=true;
  for i:=1 to k do
   if v1[i]>v2[i] then begin check:=false;break;end;
  if check then begin find:=-1;exit;end;
  find:=0;
 end;
procedure insert(v:vector;vcode:code);
 var i,sn2:longint;
 begin
  sn2:=0;
```

```
  if (v[1]=2) then
   begin end;
  for i:=1 to sn do
    case find(v,ans[i]) of
     0:begin inc(sn2);ans2[sn2]:=ans[i];ansb2[sn2]:=ansb[i];end;
     1:exit;
    end;
  inc(sn2);ans2[sn2]:=v;ansb2[sn2]:=vcode;sn:=sn2;ans:=ans2;ansb:=ansb2;
 end;
procedure Genetic;
  var S,S0:Group;
      i:longint;all:int64;
  procedure Genetic_Initial;
    var i,j:longint;
    begin
      for i:=1 to Population do
       for j:=1 to n-2 do
        s[i].data[j]:=random(n)+1;
    end;
   function Genetic_Compute(v:vector;vcode:code):longint;
   var i,mark:longint;nowv:vector;
   begin
    nowv:=maxv;mark:=0;insert(v,vcode);
    for i:=1 to k do
     begin
      mark:=mark+sqr(round((nowv[i]/v[i])*100));
      if v[i]<maxv[i] then maxv[i]:=v[i];
     end;
    Genetic_Compute:=mark;
   end;
```

```
  procedure Genetic_Start;
   var i:longint;
   begin
    all:=0;
    for i:=1 to Population do
     begin
      s[i].v:=Genetic_Uncode(s[i].data);
      s[i].power:=Genetic_Compute(s[i].v,s[i].data);
      all:=all+s[i].power;
     end;
   end;
  function Genetic_Random:code;
    var now:longint;seed:int64;
     begin
      Seed:=random(all)+1;now:=1;
      while Seed>s[now].power do begin Seed:=Seed-
s[now].power;inc(now);end;
       Genetic_Random:=s[now].data;
     end;
  function Genetic_Suddenly:code;
   var temp:code;
       Seed:longint;
    begin
     temp:=Genetic_Random;Seed:=random(N-2)+1;
     Temp[Seed]:=random(n)+1;
     Genetic_Suddenly:=temp;
    end;
  function Genetic_Change:code;
   var temp1,temp2:code;
       Seed1,Seed2,temp:longint;
```

```
    begin
     temp1:=Genetic_Random;Seed1:=random(N-2)+1;
     temp2:=Genetic_Random;Seed2:=random(N-2)+1;

Temp:=temp1[Seed1];Temp1[Seed1]:=Temp2[Seed2];Temp2[Seed2]:=Temp;
     if
Genetic_Compute(Genetic_Uncode(temp1),temp1)>Genetic_Compute(Geneti
c_Uncode(temp2),temp2) then Genetic_Change:=temp1
                                    else Genetic_Change:=temp2;
    end;
  procedure Genetic_Genarate;
   var Seed,i:longint;
   begin
    Genetic_Start;
    for i:=1 to Population do
     begin
      Seed:=random(100)+1;
      if Seed <= Suddenly then s0[i].data:=Genetic_Suddenly
                  else if seed<= Change+Suddenly then
s0[i].data:=Genetic_Change
                                  else s0[i].data:=Genetic_Random;
     end;
    s:=s0;
   end;
  begin
   Genetic_Initial;
   for i:=1 to Live do
    begin
     Genetic_Genarate;
   end;
```

```
 end;
procedure sortans(l,r:longint);
 var tl,tr:longint;tmp:vector;
 begin
  tl:=l;tr:=r;tmp:=ans[l];
  while tl<tr do
   begin
    while (ans[tr][1]>tmp[1])and(tr>tl) do dec(tr);
    if tr=tl then break;ans[tl]:=ans[tr];inc(tl);
    while (ans[tl][1]<tmp[1])and(tl<tr) do inc(tl);
    if tr=tl then break;ans[tr]:=ans[tl];dec(tr);
   end;
  ans[tl]:=tmp;
  if tl>l+1 then sortans(l,tl-1);
  if tr+1<r then sortans(tr+1,r);
 end;

procedure Output_Result;
 var i,j:longint;
 begin
  sortans(1,sn);
  for i:=1 to sn do
   begin
    write(ouf,'The ',i,'th Answer:');
    for j:=1 to k do
     write(ouf,ans[i][j]:9:2);
     write(ouf,'  Code=');for j:=1 to n-2 do write(ouf,' ',ansb[i][j]);
    writeln(ouf);
   end;
 writeln(ouf);
```

```
  for j:=1 to k do
   begin
    for i:=1 to sn do
     writeln(ouf,ans[i][j]:9:2);
     writeln(ouf,'------');
    end;
  writeln(ouf,'Total=',sn);
  close(ouf);
 end;

procedure start;
begin
  assign(inf,'input.txt');reset(inf);
  assign(ouf,'edit8.text');rewrite(ouf);
  Input_Initial;
  Genetic;
  Output_Result;
 end;
begin
 start;
end.
```

置换群快速幂运算 研究与探讨

江苏省苏州中学 潘震皓

[关键词] 置换 循环 分裂 合并

[摘要]

群是一个古老的数学分支，近几年来在程序设计中置换群得到了一定的应用。本文针对置换群的特点提出了线性时间的幂运算算法，并举例说明了优化后算法的效果。

[正文]

一、引言

置换群是一种优秀的结构，在程序设计中，它的大部分基本操作，时间和空间复杂度都是线性的，甚至有的还是常数的。所以一个问题如果能够抽象归结为一个置换群模型的话，往往能够在程序设计中轻松地解决。但是对于整幂运算来说，似乎只能通过反复做乘法来获得 O(k*乘法)或是 O(logk*乘法)的算法；而对于分数幂运算，则找不到较好的方法实现。

二、置换群的整幂运算

2.1 整幂运算的一个转化

在置换群中有一个定理：设$T^k = e$，（T 为一置换，e 为单位置换（映射函数为$f(x) = x$的置换）），那么 k 的最小正整数解是 T 的拆分的所有循环长度的最小公倍数。

或者有个更一般的结论：设$T^k = e$，（T 为一循环，e 为单位置换），那么 k 的最小正整数解为 T 的长度。

我们知道，单位置换就是若干个只含单个元素的循环的并。也就是说，长度为 l 的循环，l 次的幂，把所有元素都完全分裂了。这是为什么呢？

我们来做一个试验：(下面的置换均以循环的连接表示)

设 n=6，那么$T^6 = (T^2)^3$。任取一 T=(1 3 5 2 4 6)，来做一遍乘法：

$$\begin{aligned}
T^2 &= \begin{pmatrix}1&2&3&4&5&6\\3&4&5&6&2&1\end{pmatrix}\begin{pmatrix}1&2&3&4&5&6\\3&4&5&6&2&1\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5&6\\3&4&5&6&2&1\end{pmatrix}\begin{pmatrix}3&4&5&6&2&1\\5&6&2&1&4&3\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\\
&= (1\ \ 5\ \ 4)(2\ \ 6\ \ 3)
\end{aligned}$$

分裂成了 2 份！而且这 2 份恰好是 T 的奇数项和偶数项！（注意可以写成(1 5 4)(3 2 6)）

再来看看T^3：

$$\begin{aligned}
T^3 &= T^2T\\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}1&2&3&4&5&6\\3&4&5&6&2&1\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}5&6&2&1&4&3\\2&1&4&3&6&5\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5&6\\2&1&4&3&6&5\end{pmatrix}\\
&= (1\ \ 2)(3\ \ 4)(5\ \ 6)
\end{aligned}$$

不出所料，分裂成了 3 份，每份分别是在原来循环中的位置 mod 3=0, 1, 2 的项。

继续看T^4：

$$
\begin{aligned}
T^4 &= T^2T^2 \\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix} \\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}5&6&2&1&4&3\\4&3&6&5&1&2\end{pmatrix} \\
&= \begin{pmatrix}1&2&3&4&5&6\\4&3&6&5&1&2\end{pmatrix} \\
&= (1\ \ 4\ \ 5)(2\ \ 3\ \ 6)
\end{aligned}
$$

与前面不同的是，循环只分裂成了 2 份。并且每一份的循环看起来都是杂乱无章的，只知道是在循环中的奇数项和偶数项。

再来拿T^5做个试验：

$$
\begin{aligned}
T^5 &= T^2T^3 \\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}1&2&3&4&5&6\\2&1&4&3&6&5\end{pmatrix} \\
&= \begin{pmatrix}1&2&3&4&5&6\\5&6&2&1&4&3\end{pmatrix}\begin{pmatrix}5&6&2&1&4&3\\6&5&1&2&3&4\end{pmatrix} \\
&= \begin{pmatrix}1&2&3&4&5&6\\6&5&1&2&3&4\end{pmatrix} \\
&= (1\ \ 6\ \ 4\ \ 2\ \ 5\ \ 3)
\end{aligned}
$$

这次的循环，根本没有分裂，只是顺序改变了一下。

这之间有什么共同点呢？对，那就是 gcd(k,l)。

因为 gcd(6,6)=6，所以循环会完全分裂，而 gcd(2,6)=2, gcd(3,6)=3, gcd(4,6)=2, gcd(5,6)=1 也相对应了上面的每一个试验的结果。

经过多次试验以后，我们得到三个结论：

结论一：一个长度为 l 的循环 T，l 是 k 的倍数，则T^k是 k 个循环的乘积，每个循环分别是循环 T 中下标 i mod k=0,1,2…的元素按顺序的连接。

结论二：一个长度为 l 的循环 T，gcd(l,k)=1，则T^k是一个循环，与循环 T 不一定相同。

结论三：一个长度为 l 的循环 T，T^k是 gcd(l,k)个循环的乘积，每个循环分别是循环 T 中下标 i mod gcd(l,k)=0,1,2…的元素的连接。

可以看出，结论三只不过是把 k 分成 gcd(l,k)*(l/gcd(l,k))，再运用结论一和结论二所得到的。如果这几个结论是正确的话，那显然只需要确定结论二中叙述的T^k，就能够在 O(n)内解决任意循环的任意整幂运算了。

2.2 循环长度与指数互质时的整幂运算

和上面一样，我们来做几个试验。

设 T=(1 2 5 3 4)，则：

$$
\begin{aligned}
T^2 &= \begin{pmatrix}1&2&3&4&5\\2&5&4&1&3\end{pmatrix}\begin{pmatrix}1&2&3&4&5\\2&5&4&1&3\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5\\2&5&4&1&3\end{pmatrix}\begin{pmatrix}2&5&4&1&3\\5&3&1&2&4\end{pmatrix}\\
&= \begin{pmatrix}1&2&3&4&5\\5&3&1&2&4\end{pmatrix}\\
&= \begin{pmatrix}1&5&4&2&3\end{pmatrix}
\end{aligned}
$$

不知大家有没有注意到，如果把循环 T 的奇数项和偶数项取出来，就是 1 5 4 和 2 3，如果两者并在一起，就是刚才求出的 T^2 了。再试一个：

$$
\begin{aligned}
T^3 &= T^2T \\
&= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 1 & 2 & 4 \end{pmatrix}\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 5 & 4 & 1 & 3 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 5 & 3 & 1 & 2 & 4 \end{pmatrix}\begin{pmatrix} 5 & 3 & 1 & 2 & 4 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 2 & 5 & 1 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 3 & 2 & 4 & 5 \end{pmatrix}
\end{aligned}
$$

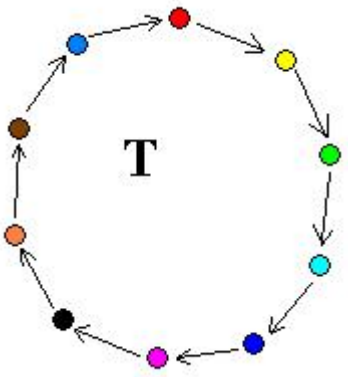
同样地，把 mod 3=1, 2, 0 的项取出来：1 3、2 4、5，连接在一起，就是所求得的新循环了。

把这一试验结果写成一个定理，就是：

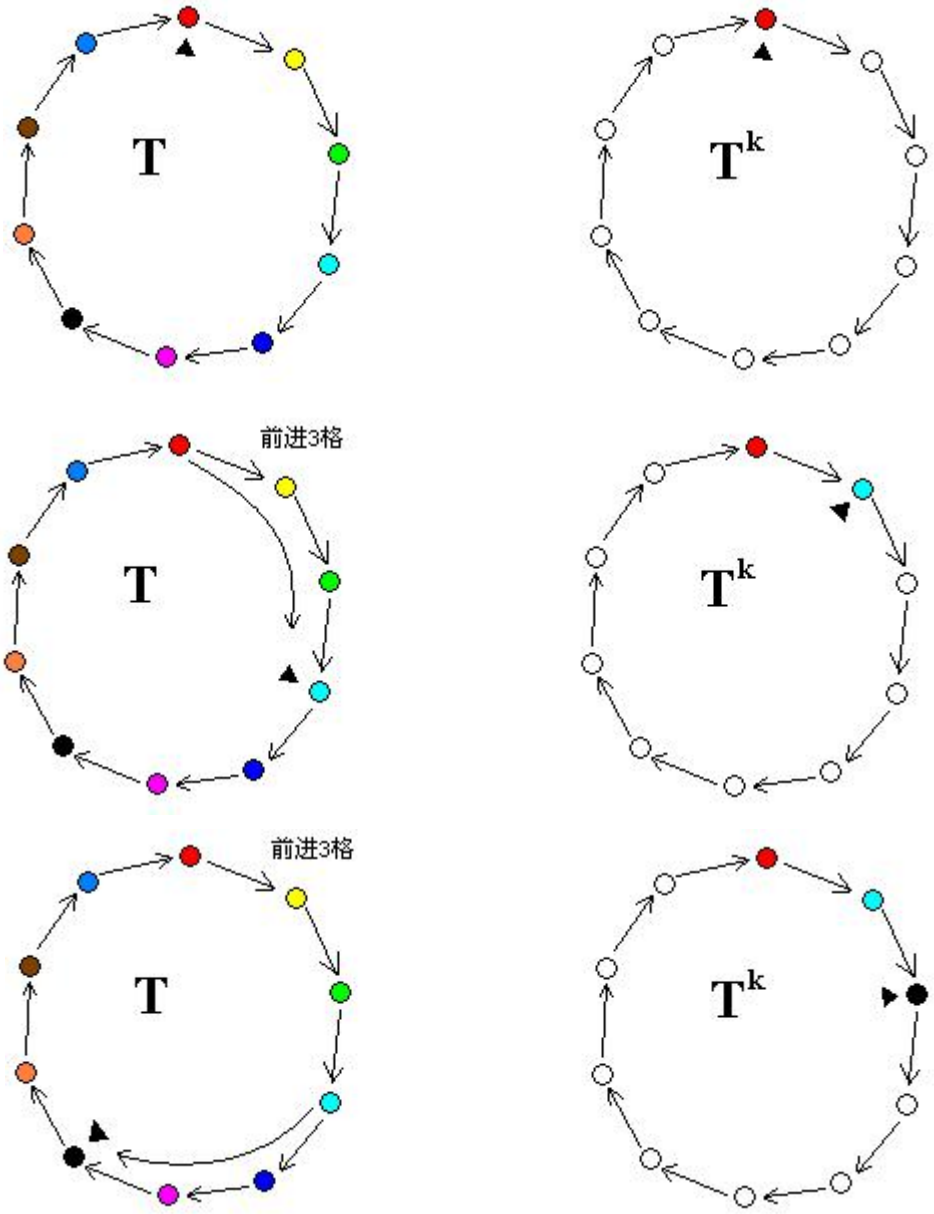
设 $a = T$ ，$a' = T^k$ ，且 gcd(l,k)=1，则 $a'[i] = a[(k+1)i \bmod l]$。

这个定理看起来似乎挺复杂，但如果画张图看，一点也不复杂：

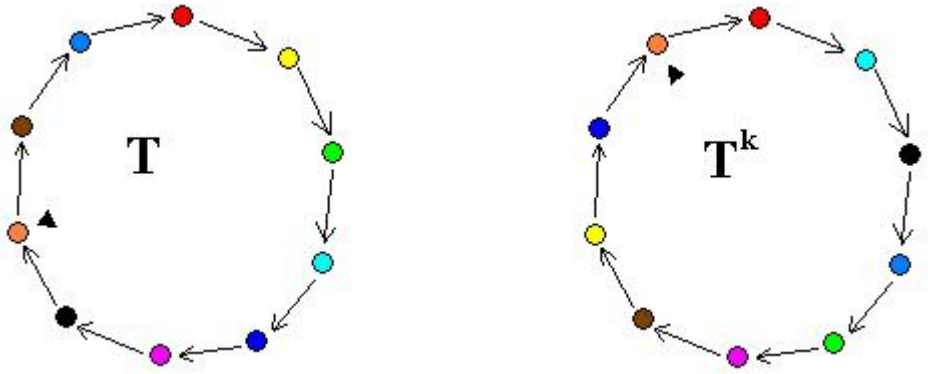
设 l=10, k=3：



我们来一步一步构造 T^k：



像这样反复前进三格，然后涂色...



最后，得到的循环就是所需要求的 T^k 了。

证明：设任意$0 \le j \le n-1$，能唯一地找到$a[t]=j$，$a'[s]=j$。

那么j→T 显然等于$a[(t+1) \bmod n]$，$j(\rightarrow)^p T = a[(t+p) \bmod n]$（$(\rightarrow)^p T$表示连续执行 p 次→T)

由置换的连接的运算法则可知，$j(\rightarrow)^p T = j \rightarrow T^p$。所以$a'[s+1]=a[(t+k) \bmod n]$。由于循环的性质，我们令$a[0]=a'[0]=1$，就得到了上面的公式。

2.3 算法的实现

根据上一节的定理和再上一节的 3 个结论，我们可以很方便地得到求整幂运算的 O(n)算法，但是如果单纯地照着做，常数项是非常大的，有时甚至还不如 O(nlogk)的算法快。针对这一问题，可以使用一个简化的算法：

- For 源置换中每一个循环
  - For 环中每一个未标记元素
    - Do
      - 做上标记
      - 放入结果数组
      - 前进 k 格
    - Until 回到这个元素
    - 将结果数组中的元素取出，得到的环，便是目标置换包含的一个循环

可以分析出，这个算法是符合上一节的定理和再上一节的 3 个结论的，在这里就不再说明了。

循环的储存我们可以单纯地用 2 个数组来实现：一个是 data，把每一个循环按顺序放在里面；一个是 point，保存每个循环在 data 中的起始位置和长度。显然，2 个数组都是 O(n)阶的。

所以，置换群的整幂运算可以用时空复杂度均为 O(n)的算法来实现。

2.4 优化的例子

庆典的日期 （国家集训队原创试题）

[问题描述]

古斯迪尔文明曾在约 10 亿年前在地球上辉煌一时，尤其在历法、数学、天文等方面的发展水平已经超过现代。在古城的众多庙宇中，考古人员都发现了一种奇特的建筑，该建筑包含一排独立的房间。

在每个房间的中央，挂有一个转盘，每个转盘分为 p 个格子，每个格子写着一个 1 到 n 的数字。转盘可以逆时针转动。转盘的红色标记始终指向上方的格子。每个房间的转盘都不相同。

CC 考古工作室近日成功地破译了当时的文字，对进一步研究古斯迪尔文明作出了重要贡献。首先，研究人员翻译了当时的宗教书籍，得知了建筑的用

途。原来每个寺院都要在建成以后每隔若干年举行一次大型的庆典。由于“天机不可泄漏”，寺院方面并不直接说明庆典的日期，而是采用“暗示”的方法。奇特的建筑就是为了确定庆典的日期而专门建造的。

房间从左到右编号为 1，2，3，…n，同时寺院有 n 个祭司也从 1 到 n 编号，这些祭司每年到房间中祈祷一次。建寺那年祭司和自己编号相同的房间祈祷。同时，转盘上红色标记指示的格子的数字就是该祭司第二年祈祷的房间编号。在祭司祈祷完毕以后，将转盘逆时针旋转一格。转盘的设计使得在每年祈祷时，每个房间只有一个祭司。

从建寺以后，当某一年祈祷时，每个祭司的编号都和祈祷房间的编号相同时，就是举行庆典的日期。实际上，每隔若干年，就会有一次庆典。

作为 CC 考古工作室的首席软件顾问，你负责编程求出第一次举行庆典的确切日期。

[输入]

文件第一行是两个整数 n，p，n 表示房间的数目（也就是祭司的数目），p 表示转盘包含的格子的数目。(0 < n, p <= 200)

以下有 n 行，每行 p 个整数，表示每个房间转盘的格子上的数字。每行第一个数表示寺院建立时红色标记指向的数字，以下的数字按照顺时针方向给出。

[输出]

仅一行，表示第一次举行庆典是在建寺以后多少年。如果永远不会出现符合条件的情况或者第一次符合条件的年份超过 10^9（那时古斯迪尔文明已经衰落了），则输出’No one knows.’

[算法分析]

由于每个房间的转盘上的数字都是 p 个，而且每年每个祭司都在不同房间，所以我们可以把这些房间中安置的转盘，转化成 p 个长度为 n 的置换。而每一年祭司本身的位置，也可以组成一个长度 n 的置换。

显然，第 i 年祭司的位置，就是第 i-1 年祭司的位置，与第 i 年转盘上数字代表的置换的连接。现在的问题是求一个最早的年份 y，使得那一年祭司的位置是单位置换。

由于转盘上的置换是以 p 为周期的，所以我们枚举 y mod p=k。那么也就是说：

$$T_1 \cdot T_2 \cdot ... T_p \cdot T_1 \cdot T_2 \cdot .... \cdot T_p \cdot .... \cdot T_{k-1} \cdot T_k = e$$

由于连接运算满足结合律，所以：

$$T_1 \cdot T_2 \cdot .... \cdot T_k \cdot (T_{k+1} \cdot T_{k+2} \cdot .... \cdot T_p \cdot T_1 \cdot T_2 \cdot .... \cdot T_k) \cdot .... \cdot (T_{k+1} \cdot T_{k+2} \cdot .... \cdot T_p \cdot T_1 \cdot T_2 \cdot .... \cdot T_k) = e$$

我们可以预先算出 $T_{Head} = T_1 \cdot T_2 \cdot .... \cdot T_k$ 和 $T_{Step} = T_{k+1} \cdot T_{k+2} \cdot .... \cdot T_p \cdot T_1 \cdot T_2 \cdot .... \cdot T_k$，那么上式就转为：

$$T_{Head} \cdot (T_{Step})^{(y-k)/p} = e$$

$$(T_{Step})^{(y-k)/p} = (T_{Head})^{-1}$$

所以，当我们枚举了 y mod p=k 以后，问题就转化成了：

已知 2 个置换 $T_{Deah} = (T_{Head})^{-1}, T_{Step}$，求一个最小的数 x=(y-k)/p 使 $T_{Step}{}^x = T_{Deah}$。

如果 T_{Step} 和 T_{Deah} 都是一个循环，那根据上面的算法立即可以得到答案的最小值 x，也可以知道，所有可行的答案的形式，都是 x+kn 的形式。那如果有多

个循环的话，显然可以列出一系列形如 $x \bmod n_i=x_i$ 的模线性方程组。解出这个方程组，就可以得到答案了。

这个算法相对于题目作者的算法而言，更优之处在于如何得到某个方程。题目作者的算法是对于每个置换中的元素，向前枚举出每个模线性方程的 x。这样最坏情况下，时间复杂度是 $O(n^2)$，是整个算法的瓶颈。并且由于在一个循环内的元素，所得到的模线性方程是一样的，就会产生大量重复计算。而这个算法求出模线性方程组的复杂度为 O(n)，而使得瓶颈转化到了求解模线性方程组的 O(nlogn)上，整个算法的时间复杂度也就是 O(nlogn)了。

三、置换群的分数幂运算（开方）

3.1 单循环的分数幂运算

a. 循环长度与指数互质的情况

由 2.2 节得到的定理，我们可以类似地得到长度与指数互质时的分数幂运算的过程，不过就是目标循环每次指针向后移 k 位，源循环每次向后移 1 位罢了。

b. 循环长度与指数不互质的情况

如果循环长度与指数不互质，是一定不能开方的。因为：

假设可以开方，分两种情况：

1.目标置换是一个循环

根据 2.1 节的结论，目标置换的 k 次幂，会把自己分裂成 gcd(l,k)份，显然不可能等于源循环。

2.目标置换由多个循环连接而成

目标置换的 k 次幂，只有可能把自己包含的循环分裂，而不可能把包含的置换合并，所以目标置换的 k 次幂也不可能是源循环。

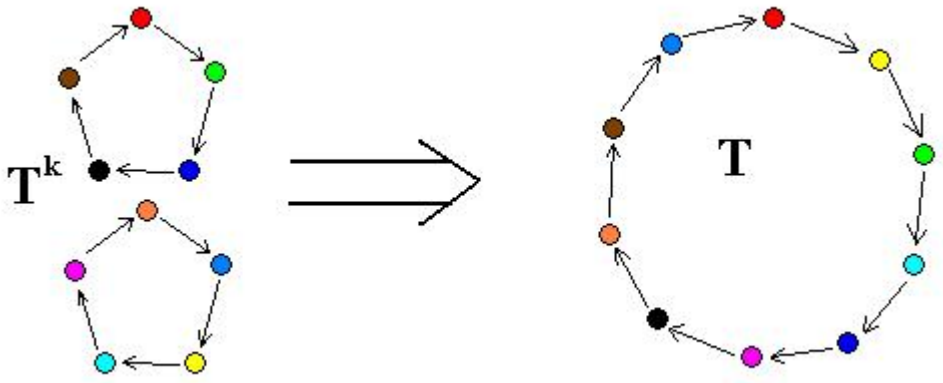
所以，循环长度与指数不互质时，单个循环是不能开方的。

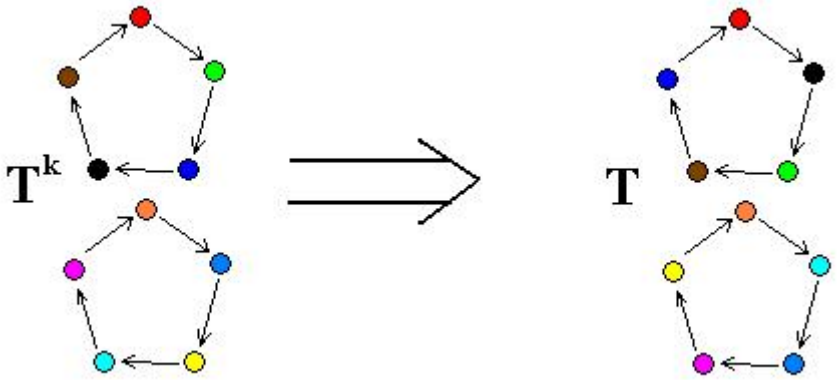
3.2 多个循环的分数幂运算

在 2.1 节的结论中，我们看到当循环长度与指数不互质时，会分裂成 gcd(l,k)份循环。那就可以联想到，如果相似地，将分出来的循环合并，就是它的逆过程了。

事实上，在开方运算中，我们可以将 k 份相同长度 l 的循环依次交错地合并，作为一次开方的过程。因为这样一个长度 k*l 的循环，作 k 次幂运算，势必会分裂成这样的 k 份每份长度 l 的循环，所以这个做法是正确的。

我们来画一张图解释一下：





对于一个由 2 个长度 5 的循环连接而成的置换 T，将它开平方就有 2 种方法：

1. 可以将两个循环交错地合并，得到一个长度 10 的大循环，正如第一张图上显示的那样；
2. 也可以将两个循环分别按照 3.1 节中的做法分别开方，得到的仍然是 2 个小循环，正如第二张图显示的那样。

可以验证，这两种方法都是正确的。

是不是一定要选择 k 个相同长度的循环合并呢？不是。

如果我们选择 m 个相同长度 l 的循环合并而作为一次运算过程的话，只要保证长度 m*l 的循环的 k 次方，会把自己分裂为 m 份，就可以保证这个做法是正确的了。

所以，对于这个 m 的值，我们有这样一个等式：gcd(m*l,k)=m。

可以看出，m 是 k 的因数，并且 gcd(l,k/m)=1。显然，这个式子的充分条件是 m 是 gcd(l,k)的倍数，那么最小的 m 就是 gcd(l,k)了。

回到 3.1 节，如果长度与指数不互质，单个循环就没有办法用 3.1 节的做法来开方。不过，我们可以选择相应 m 个长度相同的循环交错合并来完成开方的过程。可在这种情况下，如果找不到 m 个长度相同的循环，那就一定不能开方。

（因为整幂运算的结果是唯一的，并且我们已经把整幂运算的所有情况都找到了，所以不存在一个置换 T'，它的 k 次方等于这种情况下的源置换 T）

这样我们可以将上面两种情况并在一起了，那就是：

1. m=gcd(l,k)
2. 选择 n*m 份长度为 l 的循环交错合并，n 为正整数，且 gcd(n*m,k/m)=1
   （保证第三步可行咯）
3. 将大循环开 k/m 次方

3.3 算法的实现

较整幂运算来说，分数幂运算（开方）就比较复杂了，需要分好几种情况，解也不是唯一的。在这里，我提供一个能求出一个正确解的 O(n)算法：

- 将源置换中的循环按照环的长度排序（可以使用桶排序咯）
- For 源置换中每一个环
  - m=gcd(l,k)
  - 循环判断，有 m 个相同长度的循环
    - 将 m 个循环交错地输出到目标数组，并保存为一个循环
    - 将得到的大循环仿造 2.3 节的算法开 k/m 次方并保存结果
  - Else
    - 跳出，输出无解

由于排序使用了桶排序，对每个环的操作都是 O(l)的（合并时候可以使用 m 个指针，依次下移并输出，也是 O(m*l)的），所以总的时间复杂度为 O(n)。虽然常数项的确大了点，但相比没有办法来说好多了:P

3.4 优化的例子

洗牌机 (CEOI 1998)

[问题描述]

剴剴和凡凡有 N 张牌（依次标号为 1，2，……，N）和一台洗牌机。假设 N 是奇数。洗牌机的功能是进行如下的操作：对所有位置 I（1≤I≤N），如果位置 I 上的牌是 J，而且位置 J 上的牌是 K，那么通过洗牌机后位置 I 上的牌将是 K。

剴剴首先写下一个 1~N 的排列 a_i，在位置 a_i 处放上数值 a_{i+1} 的牌，得到的顺序 $x_1, x_2, ..., x_N$ 作为初始顺序。他把这种顺序排列的牌放入洗牌机洗牌 S 次，得到牌的顺序为 $p_1, p_2, ..., p_N$。现在，剴剴把牌的最后顺序和洗牌次数告诉凡凡，要凡凡猜出牌的最初顺序 $x_1, x_2, ..., x_N$。

[输入]

第一行为整数 N 和 S。1≤N≤1000，1≤S≤1000。第二行为牌的最终顺序 $p_1, p_2, ..., p_N$。

[输出]

为一行，即牌的最初顺序 $x_1, x_2, ..., x_N$。

[算法分析]

很显然，这题的一副扑克牌就是一个置换，而每一次洗牌就是这个置换的平方运算。由于牌的数量是奇数，并且一开始是一个大循环，所以做平方运算时候不会分裂。所以，在任意时间，牌的顺序所表示的置换一定是一个大循环。

那么根据文章开头提到的定理：设$T^k = e$，（T 为一循环，e 为单位置换），那么 k 的最小正整数解为 T 的长度。

可以知道，这个循环的 n 次方是单位循环，换句话说，如果 k mod n=1，那么这个循环的 k 次方，就是它本身。我们知道，每一次洗牌是一次简单的平方运算，洗 x 次就是原循环的 2^x 次方。

因为 n 是奇数，2^x mod n=1 一定有一个<n 的整数解，假设这个解是 a；那也就是说，一幅牌，洗 a 次，就会回到原来的顺序。使用最终顺序不停地洗，直到回到原始顺序，求出循环节长度 a 以后，再单纯地向前模拟 a-s 次，就可以得到原始顺序了。

上面的算法是出题方给出的标准算法。显然，时间复杂度为 $O(n^2+logs)$。

换一个方向：给定了结果和 s 以后，可以简单地将这个目标置换用 3.1 节的方法开方 s 次得到结果。时间复杂度为 O(n*s)。

或者可以更简单地，算出 2^s，将目标置换直接开 2^s 次方。这里有一个技巧，因为在开方时只需要在循环中前进 2^s 次，所以我们只关心(2^s) mod n，也就免去了大数字的运算。所以，计算 2^s 需要 O(logs)，而开方需要 O(n)。整个时间复杂度为 O(n+logs)。

三、总结

置换群的幂运算这一问题是从最后一个例子洗牌机想到的，这一切都是对问题的深入研究带来的结果；分裂是自然而然的，而合并却是我们自己捏出来的，这一切又都是思想逆转所造成的结果；通过分裂和合并，置换群的幂运算被完美地解决了，这一切又都是多举例子多作猜想而得到的结果。

每当发现问题，探寻问题，解决问题的时候，人就会找到进步的道路。而完成这一切时，人就进步了。

[参考文献]

《算法艺术与信息学竞赛》

刘汝佳

《Graduate Texts in Mathematics》 No.163:

Permutation Groups

John D.Dixon

Brian Mortimer

[附录]

1. 《洗牌机》原题

Alice and Bob have a set of N cards labelled with numbers 1 ... N (so that no two cards have the same label) and a shuffle machine. We assume that N is an odd integer.

The shuffle machine accepts the set of cards arranged in an arbitrary order and performs the following operation of **double shuffle** : for all positions i, $1 \le i \le N$, if the card at the position i is j and the card at the position j is k, then after the completion of the operation of double shuffle, position i will hold the card k.

Alice and Bob play a game. Alice first writes down all the numbers from 1 to N in some random order: a_1, a_2, ..., a_N. Then she arranges the cards so that the position a_i holds the card numbered a_{i+1}, for every $1 \le i \le N-1$, while the position a_N holds the card numbered a_1.

This way, cards are put in some order x_1, x_2, ..., x_N, where x_i is the card at the i_{th} position.

Now she sequentially performs S double shuffles using the shuffle machine described above. After that, the cards are arranged in some final order p_1, p_2, ..., p_N which Alice reveals to Bob, together with the number S. Bob's task is to guess the order x_1, x_2, ..., x_N in which Alice originally put the cards just before giving them to the shuffle machine.

Input data

The first line of the input file **CARDS.IN** contains two integers separated by a single blank character : the odd integer N, $1 \le N \le 1000$, the number of cards, and the integer S, $1 \le S \le 1000$, the number of double shuffle operations.

The following N lines describe the final order of cards after all the double shuffles have been performed such that for each i, $1 \le i \le N$, the $(i+1)_{st}$ line of the input file contains p_i (the card at the position i after all double shuffles).

Output data

The output file **CARDS.OUT** should contain N lines which describe the order of cards just before they were given to the shuffle machine.

For each i, $1 \le i \le N$, the i_{th} line of the output file should contain x_i (the card at the position i before the double shuffles).

序的应用

长沙市雅礼中学 龙凡

【摘要】

信息学竞赛的本质是对数据进行挖掘，而“序”是隐藏在数据之间的一种常见的但却难以发现的关系。如果能够在解题的过程中找出题目中隐藏的序关系，往往题目便可以迎刃而解。本文重点讨论如何在复杂的数据关系中发现和构造适当的序，使得问题获得简化和解决。

【关键字】

数据关系、序、树、DFS 序列、图、拓扑序列

【引言】

信息学中序的应用很广泛，最基本的有基于大小序的二分查找算法、基于拓扑序的有向无环图的动态规划。而为了得到这些序，我们有相应的快速排序算法、和拓扑排序算法。

图、树、线性以及集合等等，这些不同的数据关系，有着与之对应的不同的序。而同一种数据关系，在不同的意义下，有着不同的序。比如有向图在遍历和深度优先的意义下有 DFS 序，而在前后依赖关系的意义下有着拓扑序。序本身并不一定是线性的，拓扑序就不是严格线性的序。

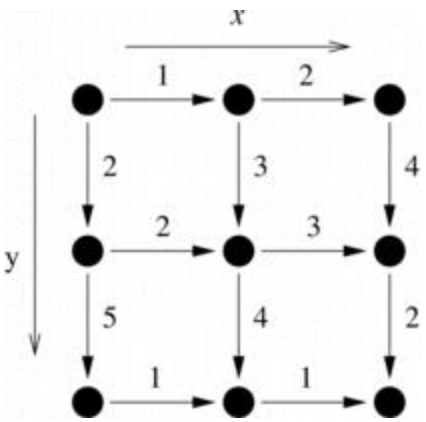
在繁杂的数据中，找到对我们有价值的序，并加以合理的应用，便是我们的课题。

【正文】

一、 通过“序”使得问题解决的例子

很多交互式题目，如果能够找到合适的序，根据序的特性来进行交互，往往能够使得问题得到很好地解决。我们来看这样一个例子：

方格：



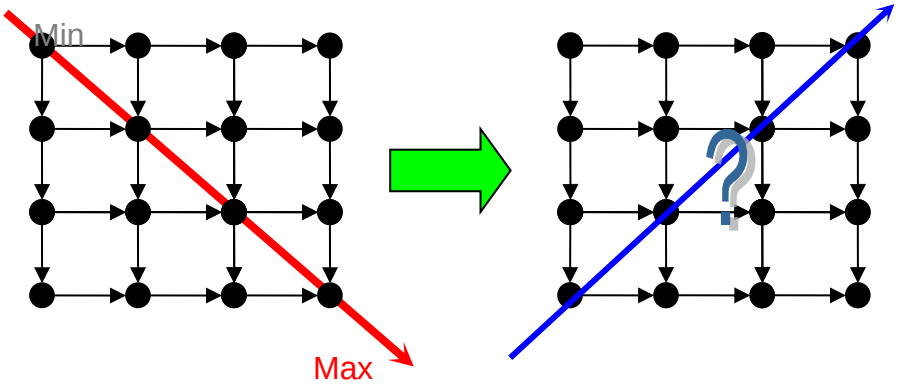
*有一个 N*N（1<=N<=2003）的点阵，相邻点之间会有一条带整数权 w 的有向弧（1<=w<=500000）。并且，从左上角的点（$v_{1,1}$）到某一点的所有路径的长度（途经的所有弧的权之和）都相等。*

*每次可以提问某个点(x,y)到同行最右边的点离他的距离，及同列最下边的点离他的距离。现在要你找到一个点，使得左上点到他的距离为给定的整数 L（*1<=L<=2000000000）*。最多允许提问 6667 次（你最开始仅仅已知 N、L）。*

我们首先可以否定一种最简单的想法，即把每条边的长度都通过询问+解方程计算出来。因为边的总数是 $2(N-1)N$ ，而我们仅仅可以从“*并且，从左上角的点（$v_{1,1}$）到某一点的所有路径的长度（途经的所有弧的权之和）都相等。*”这句话中得到 $(N-1)^2$ 个方程（除了对于每个非左、上边缘之外的点存在一个其左上点到该点的两条路经相等，其他的都可以通过这些方程加减得出）也就是说至少还要询问 $2(N-1)N-(N-1)^2=N^2-1$ 次。而当 N=2003 的时候，6667 仅仅相当于 3N 左右。这是不可能的！

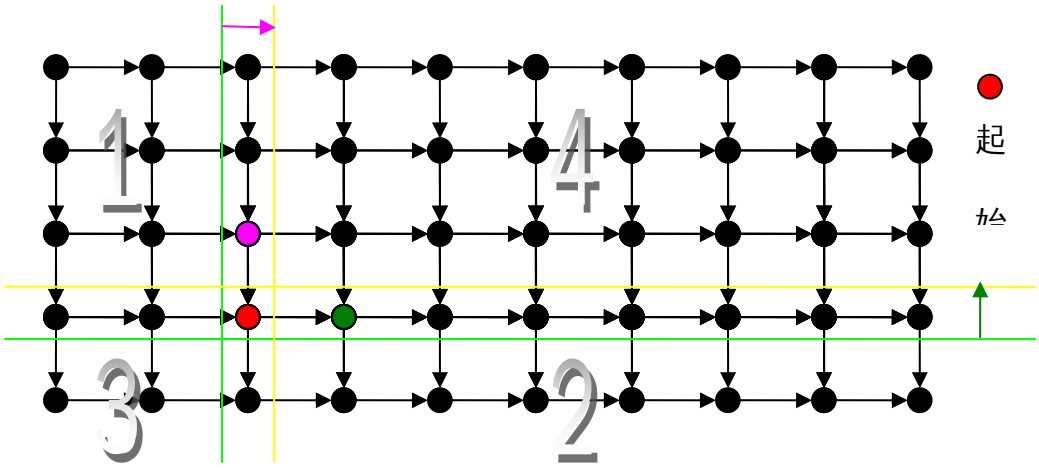
不妨从题目的要求入手：题目仅仅要我们找到一个点，他到左上角点的距离为 L。**也就是说我们并不关心每条边的权值，而只关心每个点的距离。**我们不妨把每个点到左上角点的距离叫做每个点的权值。

有一个很重要的事实即：*每个点的权值，都比处在他右下方位的点的权值小。*



这是本题最基本的序，但是它并不是很好直接利用。但是它让我们不得不思考，既然左上到右下是依次变大，那么左下到右上又有什么性质呢？从直观上感觉权值应该基本*差不多*，那么我们是不是可以先找到一个最左下的权值接近 L 的点，然后依次沿着右上方向找过去呢？

以上是一个非常含糊的思路。最左下的权值接近 L 的点，具体来说：我们先从最左边一列，向下找，找到一个点，他在最左边一列，**且他的权值刚好小于 L**。（即这个点的下方那个点就大于 L 了）然后顺着这个点往右找到最接近 L 的同排点 S，这个点就是我们最初检查的点。我们来考察这个点的性质：



1． 由基本的序，我们可知 1 区域内不存在解。

2． 由我们找的起始点的定义“**且他的权值刚好小于 L**”，可以知道 2,3 区域内不存在解。

也就是说：**解只存在于 4 号区域中。**

那么我们来讨论红色点的权值和 L 的大小关系：

1． 相等：也就是说，这就是解。

2． 大于 L：根据基本的序，我们可以知道所有原来在 4 区域，并且和红色点**同行**的都不可能是解（都在右方）。也就是说我们可以向上移动，转而检查紫色的点。

3． 小于 L：根据基本的序，我们可以知道所有原来在 4 区域，并且和红色点**同列**的都不可能是解（都在上方）。也就是说我们可以向右移动，转而检查绿色的点。

我们始终保持了仅在 4 号区域——也就是检查的点的右上方存在解的性质。并且不断把检查的点向右上方移动，缩小范围，直到找到解。最开始找起始点的时候，使用两次二分法，那么最坏情况下总共需要检查的点的个数是：$2\log_2 n+2n$，当 N 最大时也小于 6667，可以满足题目要求。

通过一个简单、基本的序，并针对它的特点大胆的设计算法的基本思路。使得我们成功地解决的这道题目。这也提示我们不要忽视一些题目内在简单、基本的性质。

二、 通过应用“序”简化问题的例子

先看一道题目：

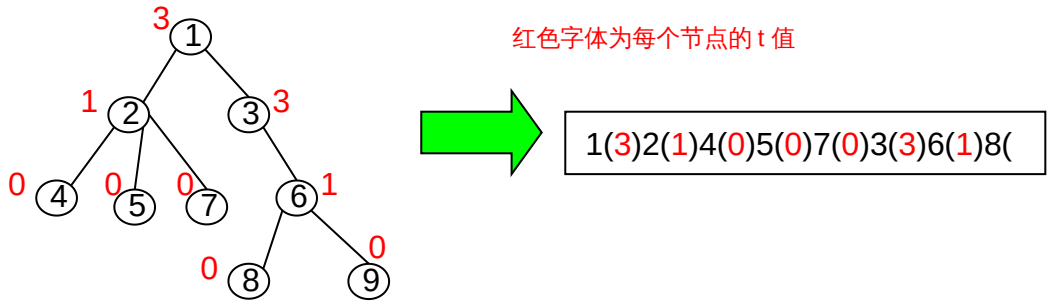
树的构造：

一棵含有 n 个节点的树，所有的节点依次编号为 1, 2, 3, …, n，每个节点 v 有一个权值 s(v)，也分别是 1,2,3,…,n。对于编号为 v 的节点，定义 t(v)为 v 的后代中所有权值小于 v 的节点个数。现在给出这棵树及 t(1),t(2),t(3),…,t(n)，请你求出这棵树每个节点的权值。（多解任意输出一组解）

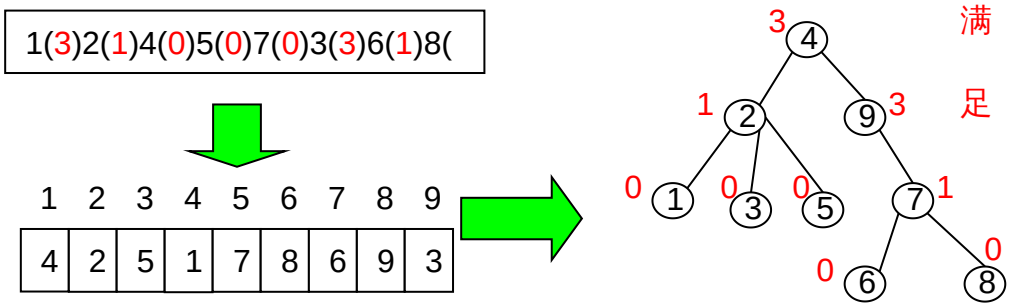
这道题目看似难以下手，但是只要抓住 t(v)的定义，就可以很轻松的解决了。

*定义 t(v)为 v 的**后代**中所有权值小于 v 的节点个数。*

这提示我们利用 DFS 先序遍历来处理这道题，先序遍历的一个重要的特点就是每个节点的后代都紧跟该节点被遍历。下面我们来考察 DFS 先序遍历序列：



假设我们认为从左到右有 N 个格子——权值 1,2,3,…,n 分别对应不同的格子。如果第 J 个格子里填上了 I，我们就认为节点 I 的权值是 J。我们可以根据题目的要求发现：如果依照 DFS 先序遍历的顺序来依次把每个节点填入格子中。那么对于节点 i，我们都至少需要在他的左边留下 t(i)个空位，也就是填在 t(i)+1 个空位。因为它的后代中有 t(i)个比他小，**而他的后代都是紧跟着他出现。**也就是说，在之后的过程中，至少要有 t(i)个数要填在它左边。而事实上，如果每次都留下 t(i)个空位，则恰好可以满足题目的条件。



我们现在来证明这样构造的正确性，我们分两部证明：

(1) 证明这个算法产生的权值，对于一个节点数为 N 的树，既不会重复，也不会超出 1…n 的范围。

我们用数学归纳法证明：一个节点为 N 的树，运行算法，会将自己的所有节点填入前 N 个空位置中。

I. 当 n=1 时，命题成立。因为必然 t(1)=0，所以算法会将节点填入第一个空位置。

II. 当 n<k 时满足条件，现在我们证明 n=k 时满足条件，不妨设树的根节点为 1，而他的所有儿子分别是 $S_1S_2...S_m$ 共 M 个。则根据 DFS 先序遍历的性质，他们在序列中的先后顺序为：

$1+S_1$ 为根的子树+ S_2 为根的子树+…+ S_m 为根的子树

执行算法 1 将填入第 t(1)+1 个空位置。而显然 $S_1S_2S_3…S_m$ 这 M 棵子树的节点总数都小于原树的节点个数 k。根据 n<k 时命题成立，可得 $S_1S_2S_3…S_m$ 会依次被填入。并且和 1 号节点一起，占据前 N 个空位置。

(2) 证明这个算法产生的权值，对于每个节点 i，他之前留下的 t(i)全部填入的是他自己的子孙。

在证明(1)中，我们可以知道根 1 被填入到了第 t(1)+1 个空位置。而它以及它的子孙占据了前 N 个空位置。由于 t(1)<N，所以前 t(1)个空位置，必然是他自己的子孙。命题得证。

我们的问题已经简化成了：已知一串长度为 N 的方格，然后每次接到一个命令，形式是把 A_i 这个数填入第 P_i 个空格，要我们求出最终的状态。

这个问题可以使用树状数组或者线段树解决，我们只要纪录每个方格是否为空格，每次使用 $O(\log_2 N)$的时间复杂度维护和查找。总共有 N 个命令（N 个节点），时间复杂度为 $O(N \log_2 N)$，而 DFS 部分时间复杂度为 O(N)，所以总时间复杂度为 $O(N \log_2 N)$。空间复杂度为 O(N)。

再看一道复杂一点的题目：

士兵排队：

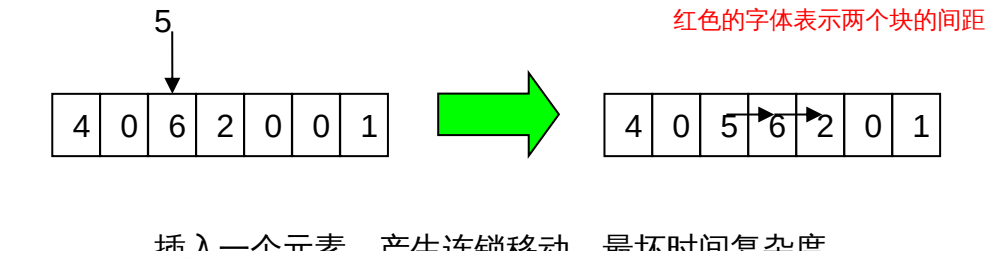
N 个士兵在进行队列训练，从左至右有 M 个位置。每次将军可以下达一个命令，表示为 Goto(L,S)。

a. *若队列 L 位置上为空，那么士兵 S 站在 L 上。*
b. *若队列 L 位置上有士兵 K，那么士兵 S 站在 L 上，执行 Goto（L+1，K）。*

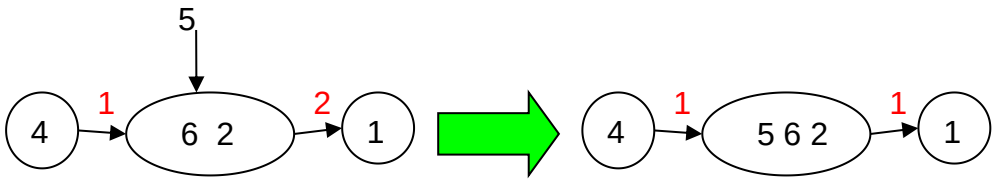
将军依次下达 N 个命令，每个士兵被下达命令一次且仅一次。要你求出最后队列的状态。（有可能在命令执行过程中，士兵站的位置标号超过 M，所以你最后首先要求出最终的队列长度。0 表示空位置）

直接模拟的时间复杂度是 $O(N^2)$，显然并不能让我们满意。通过代码优化（即使用并查集+Move 函数块移动），可以通过这道题目的绝大部分数据，但是时间复杂度归根结底还是没有变化。

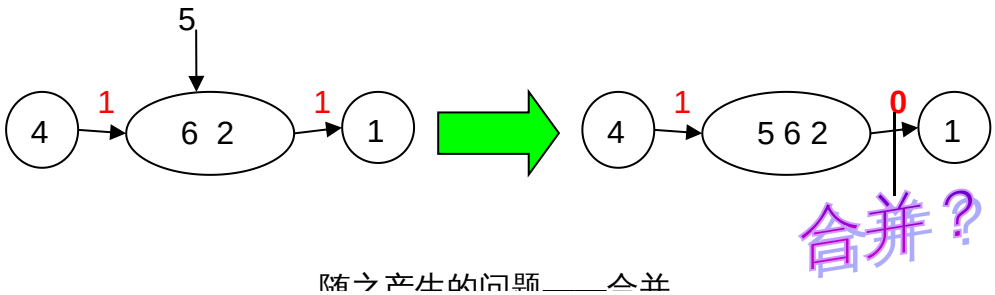
模拟的另外一种方法是，即把整块一起考虑。整块顾名思义，意思是：连在一起的士兵，叫做一整块☺。一个整块对外是一个整体，只是在块内部记录块成员之间的相对位置。这样做的好处：



插入一个元素，产生连锁移动，最坏时间复杂度



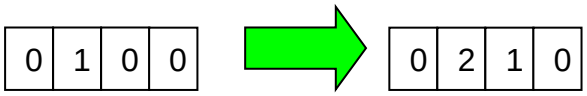
插入一个元素，不产生连锁移动



随之产生的问题——合并

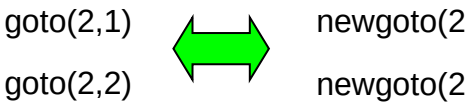
由于因为在块内的定位问题，使得我们涉及到了集合元素的查找，再加上插入，最好的选择无疑是平衡二叉树。平衡二叉树中，我们每个元素的权值就是他在内部的相对位置，这样插入似乎时间复杂度降到了 $O(\log_2 N)$。事实上，出现了新的问题——合并，如上图所示。如果两个块间距变为 0，那么必然要合并。众所周知，平衡二叉树是无法轻松实现合并的，唯一的“野蛮”办法是把小的块中的元素一个一个卸下来往大的块上插。并且由于合并，我们还需要配合并查集来存储块的基础信息。（如长度，开始位置等等）根据分摊时间复杂度分析，总的时间复杂度最坏为 $O(N(\log_2 N)^2)$。而实现起来也是非常之麻烦，需要编写平衡二叉树。一种更好的利用平衡二叉树的方法是，把所有格子按顺序构造一个平衡二叉树，每个节点记录一下空白位置，方便定位，删除等等，这样做时间复杂度是 $O(N \log N)$。

这道题目还有更加简便高效的解决方法。题目的关键在于插入元素会引起连锁移动，我们来看最基本的情况：

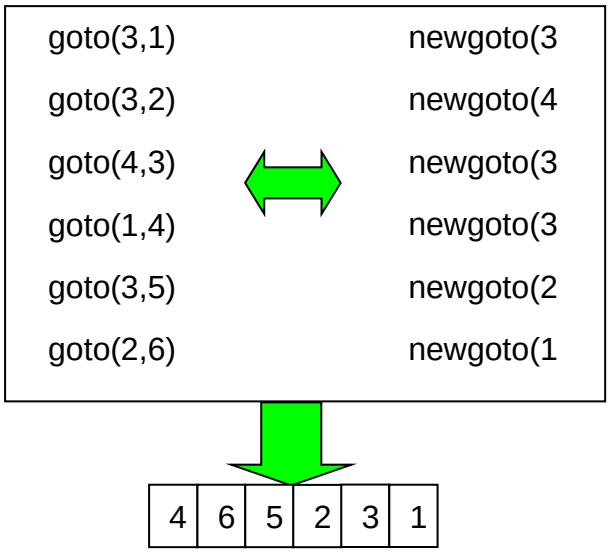


题目有一个特殊条件：***每个士兵被下达命令一次且仅一次。***这就相当于把士兵当作数，位置看作格子的话，这就是一个填数字的问题！上面的例子下达了两个命令 goto(2,1)和 goto(2,2)。而执行 goto(2,2)的时候，无可避免的进行了连锁移动。**试想我们如果先填入 2，再填入 1 的话。我们事实上是将 2 忽略，将 1 填入了第二个空位置中！**这和上一道题目最终转化成的问题如出一辙！

我们定义一个新的插入命令 newgoto(L,S)，意义是将 S 士兵插入在第 L 个空位置（注意是空位置，不是位置）中。那么有：



再看一个复杂一点的例子，有：



我们观察到，只是(L,S)数对的次序变化了而已，左边 goto 命令的效果等价于右边的 newgoto 命令！我们不禁要想，是不是对于任何 goto 插入序列，总存在一个仅仅交换了(L,S)数对次序的 newgoto 插入序列与之对应？

答案是肯定的。先来确认一下，我们需要找的序要满足什么条件：

1. 如果两个互不相干的块 A、B。**A 在 B 之前（即在 B 的左边），则 B 中的所有元素要在 A 的所有元素之前在序列中出现。**这个很容易理解，如果先插入 A 中的元素，会引起 B 中的元素位置后移。
2. 如果两个元素 S、K，由于 S 的插入造成了 K 的连锁移动。则 S 要在 K 之前插入。事实上这就是 newgoto 序列避免连锁插入的原理。

可以知道，如果满足上面的两个条件，则肯定是一个等价的 newgoto 序列。如果我们将 N 个士兵，看成 N 个点。A 士兵如果要在 B 之前插入，则 A 向 B 引一条有向边。我们需要的序就是这个图的**拓扑序**。

我们并不能直接套用拓扑排序算法，有两个很大的困难：

1. 我们并不能方便的构造出这个图，求出每条边的过程事实上就是模拟 goto 命令。

2. 这个图的边是 N^2 级的，直接应用拓扑排序算法的时间复杂度为 $O(N^2)$。

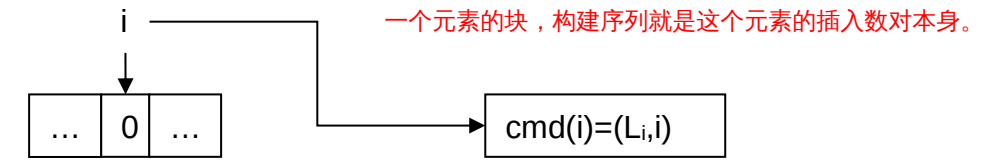
不妨换一种思路，用**不完全模拟**的方法来求出这个拓扑序列。所谓不完全模拟，就是说我们在模拟的时候，不记录块内的元素之间的位置关系。**后面我们会看到元素之间在块内位置关系，和我们要求的序列没有关系。**

由于不要存储块内的位置关系，我们完全可以使用并查集。**并查集对于每个块要记录一个拓扑序，即单独生成这个块的 newgoto 序列。**然后通过不完全模拟，生成整个拓扑序。最开始整个格子都是空，下面给出具体的生成过程，为了方便说明，定义 cmd(A)为块 A 的插入序列。

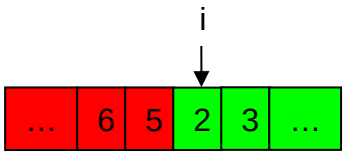
当执行的一个 goto(L_i,i)命令，分两步进行：

a. 插入

如果 i 插入到一个空位置，则我们新建一个块，而这个块的 newgoto 序列即为 goto 命令的数对 **cmd(i)=(L_i,i)**。



如果 i 插入到一个块中（即非空位置），则我们先将 i 放入块中，然后把这个块的插入序列更新，把(L_i,i)放在序列首。因为：

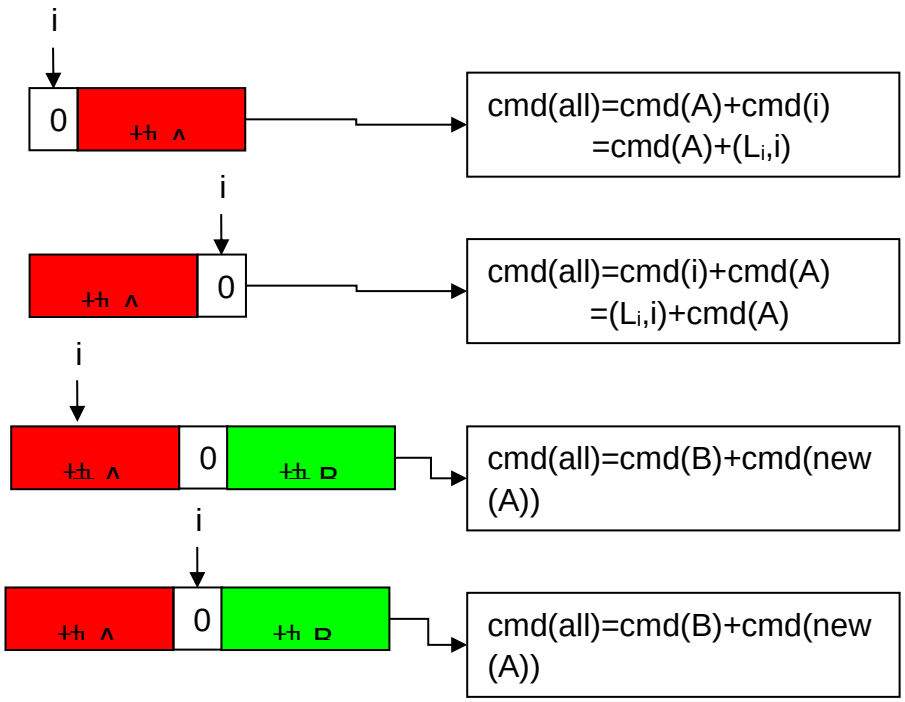


无论是之前还是之后的元素，都必须在 i 之后被插入。

绿色的部分，即因为 i 插入被移动的部分，必须要在 i 之后插入。而红色的部分，i 与它们没有关联，所以必须要先插入 i，若先插入红色部分，则 i 的位置会受到影响。所以只要直接把 i 的插入数对(L_i,i)插到队首即可。即 **$cmd(A+i)=(L_i,i)+cmd(A)$**。

b. 合并

插入有时会引起合并，在并查集上实现合并两个块是很容易的。关键是对于它们的插入序列也要进行合并。我们合并要依据：**A 在 B 之前（即在 B 的左边），则 B 中的所有元素要在 A 的所有元素之前在序列中出现。**即如果两个块 A,B 合并，A 在 B 之前，则合并后的序列 **$cmd(A+B)=cmd(B)+cmd(A)$。**根据这个原理，以下是几种常见合并情况的处理：



根据上面的方法，依次合并。最后再将每个单独的块，按照**从后到前**的顺序将生成序列串到一起，就是我们要找的拓扑序。很容易证明，这个求出来的 newgoto 序列和 goto 序列等价。这个过程的时间复杂度为 O($N\alpha(N)$)。

我们用 O($N\alpha(N)$) 的时间复杂度把 goto 指令序列转化成了 newgoto 指令序列。而模拟 newgoto 指令序列的时间复杂度为 O(Nlog$_2$N)。总时间复杂度为 O(Nlog$_2$N)。

我们通过应用不同的序，把两道看似完全不相干的题目简化成了同一个可以用数状数组或线段树轻松解决的问题。应用合适的序可以帮我们看清题目的本质，让题目得到更好地解决。

【总结】

不同的题目隐藏着不同的可供我们挖掘的序。如果我们能够好好加以利用，那么序就是我们的一双慧眼，让我们把问题看得更加透彻，也更能在眼花缭乱的数据关系中看清问题的实质。

从上面的例子也可以看出，序的运用远远不是那么简单。有些序存在得很隐蔽，甚至需要自己去构造。这就需要平时的积累以及对数据之间的微妙关系的敏感。

【感谢】

感谢雷涛同学原创试题以及栗师同学对其题目的改编

感谢任凯、周源同学对士兵排队问题给予的帮助

【参考文献】

1． CEOI2003 试题

2． <Introduction to Algorithms>

3． 湖南长沙雅礼中学雷涛同学的原创试题

4． 2002 年国家集训队李睿同学的论文

Hash 函数的设计优化

天津南开中学 李羽修

【摘要】

Hash 是一种在信息学竞赛中经常用到的数据结构。一个好的 Hash 函数可以很大程度上提高程序的整体时间效率和空间效率。本文对面向各种不同标本（关键值）的 Hash 函数进行讨论，并对多种常用的 Hash 函数进行了分析和总结。

【关键字】

Hash 函数，字符串，整数，实数，排列组合

【正文】

对于一个 Hash 函数，评价其优劣的标准应为**随机性**，即对任意一组标本，进入 Hash 表每一个单元（cell）之**概率的平均程度**，因为这个概率越平均，数据在表中的分布就越平均，表的空间利用率就越高。由于在竞赛中，标本的性质是无法预知的，因此数学推理将受到很大限制。我们用实验的方法研究这个随机性。

一、整数的 Hash 函数

常用的方法有三种：直接取余法、乘积取整法、平方取中法。下面我们对这三种方法分别进行讨论。以下假定我们的关键字是 k ，Hash 表的容量是 M ，Hash 函数为 $h(k)$ 。

1．直接取余法

我们用关键字 k 除以 M ，取余数作为在 Hash 表中的位置。函数表达式可以写成：

$$h(k) = k \bmod M \text{ 。}$$

例如，表容量 $M = 12$ ，关键值 $k = 100$ ，那么 $h(k) = 4$ 。该方法的好处是实现容易且速度快，是很常用的一种方法。但是如果 M 选择的不好而偏偏标本又很特殊，那么数据在 Hash 中很容易扎堆而影响效率。

对于 M 的选择，在经验上，我们一般选择不太接近 2^n 的一个素数；如果关键字的值域较小，我们一般在此值域 1.1~1.6 倍范围内选择。例如 k 的值域为 $[0,600]$ ，那么 $M = 701$ 即为一个不错的选择。竞赛的时候可以写一个素数生成器或干脆自己写一个“比较像素数”的数。

我用 4000 个数插入一个容量为 701 的 Hash 表，得到的结果是：

测试数据	随机数据	连续数据
最小单元容量：	0	5
最大单元容量：	15	6
期望容量：	5.70613	5.70613
标准差：	2.4165	0.455531

可见对于随机数据，取余法的最大单元容量达到了期望容量的将近 3 倍。经测试，在我的机器（Pentium III 866MHz，128MB RAM）上，该函数的运行时间大约是 39ns，即大约 35 个时钟周期。

2．乘积取整法

我们用关键字 k 乘以一个在 $(0,1)$ 中的实数 A（最好是无理数），得到一个 $(0,k)$ 之间的实数；取出其小数部分，乘以 M，再取整数部分，即得 k 在 Hash 表中的位置。函数表达式可以写成：

$$h(k)=\lfloor M(kA \bmod 1)\rfloor ;$$

其中 $kA \bmod 1$ 表示 kA 的小数部分，即 $kA-\lfloor kA\rfloor$。例如，表容量 $M=12$，种子 $A=\frac{\sqrt{5}-1}{2}$（$A=\frac{\sqrt{5}-1}{2}$ 是一个实际效果很好的选择），关键值 $k=100$，那么 $h(k)=9$。

同样用 4000 个数插入一个容量为 701 的 Hash 表（$A=\frac{\sqrt{5}-1}{2}$），得到的结果是：

测试数据	随机数据	连续数据
最小单元容量：	0	4
最大单元容量：	15	7
期望容量：	5.70613	5.70613
标准差：	2.5069	0.619999

从公式中可以看出，这个方法受 M 的影响是很小的，在 M 的值比较不适合直接取余法的时候这个方法的表现很好。但是从上面的测试来看，其表现并不是非常理想，且由于浮点运算较多，运行速度较慢。经反复优化，在我的机器

上仍需 892ns 才能完成一次计算，即 810 个时钟周期，是直接取余法的 23 倍。

3．平方取中法

我们把关键字 k 平方，然后取中间的 $\lfloor \log_2 M \rfloor$ 位作为 Hash 函数值返回。由于 k 的每一位都会对其平方**中间的**若干位产生影响，因此这个方法的效果也是不错的。但是对于比较小的 k 值效果并不是很理想，实现起来也比较繁琐。为了充分利用 Hash 表的空间，M 最好取 2 的整数次幂。例如，表容量 $M = 2^4 = 16$，关键值 $k = 100$，那么 $h(k) = 8$。

用 4000 个数插入一个容量为 512 的 Hash 表（注意这里没有用 701，是为了利用 Hash 表的空间），得到的结果是：

测试数据	随机数据	连续数据
最小单元容量：	0	1
最大单元容量：	17	17
期望容量：	7.8125	7.8125
标准差：	2.95804	2.64501

效果比我们想象的要差，尤其是对于连续数据。但由于只有乘法和位运算，该函数的速度是最快的。在我的机器上，一次运算只需要 23ns，即 19 个时钟周期，比直接取余法还要快一些。

比较一下这三种方法：

	实现难度	实际效果	运行速度	其他应用
直接取余法	易	好	较快	字符串
乘积取整法	较易	较好	慢	浮点数
平方取中法	中	较好	快	无

从这个表格中我们很容易看出，直接取余法的性价比是最高的，因此也是我们竞赛中用得最多的一种方法。

对于实数的 Hash 函数，我们可以直接利用乘积取整法；而对于标本为其他类型数据的 Hash 函数，我们可以先将其转换为整数，然后再将其插入 Hash 表。下面我们来研究把其他类型数据转换成整数的方法。

二、字符串的 Hash 函数

字符串本身就可以看成一个 256 进制（ANSI 字符串为 128 进制）的大整数，因此我们可以利用直接取余法，在线性时间内直接算出 Hash 函数值。为了保证效果，M 仍然不能选择太接近 2^n 的数；尤其是当我们把字符串看成一个 2^p 进制数的时候，选择 $M = 2^p - 1$ 会使得该字符串的任意一个排列的 Hash 函数值都相同。（想想看，为什么？）

常用的字符串 Hash 函数还有 ELFHash，APHash 等等，都是十分简单有效的方法。这些函数使用位运算使得每一个字符都对最后的函数值产生影响。另外还有以 MD5 和 SHA1 为代表的杂凑函数，这些函数几乎不可能找到碰撞（MD5 前一段时间才刚刚被破解）。

我从 Mark Twain 的一篇小说中分别随机抽取了 1000 个**不同**的单词和 1000 个**不同**的句子，作为短字符串和长字符串的测试数据，然后用不同的 Hash 函数把它们变成整数，再用直接取余法插入一个容量为 1237 的 Hash 表，遇到冲突则用新字符串覆盖旧字符串。通过观察最后“剩下”的字符串的个数，我们可以粗略地得出不同的 Hash 函数实际效果。

	短字符串	长字符串	平均	编码难度

直接取余数	667	676	671.5	易
P. J. Weinberger Hash	683	676	679.5	难
ELF Hash	683	676	679.5	较难
SDBM Hash	694	680	687.0	易
BKDR Hash	665	710	687.5	较易
DJB Hash	694	683	688.5	较易
AP Hash	684	698	691.0	较难
RS Hash	691	693	692.0	较难
JS Hash	684	708	696.0	较易

把 1000 个随机数用直接取余法插入容量为 1237 的 Hash 表，其覆盖单元数也只达到了 694，可见后面的几种方法已经达到了极限，随机性相当优秀。然而我们却很难选择，因为不存在完美的、既简单又实用的解决方案。我一般选择 JS Hash 或 SDBM Hash 作为字符串的 Hash 函数。这两个函数的代码如下：

```
unsigned int JSHash(char *str)
{
	unsigned int hash = 1315423911; // nearly a prime - 1315423911 = 3 * 438474637
	while (*str)
	{
		hash ^= ((hash << 5) + (*str++) + (hash >> 2));
	}
	return (hash & 0x7FFFFFFF);
}

unsigned int SDBMHash(char *str)
{
	unsigned int hash = 0;
	while (*str)
	{
		// equivalent to: hash = 65599*hash + (*str++);
		hash = (*str++) + (hash << 6) + (hash << 16) - hash;
	}
	return (hash & 0x7FFFFFFF);
}
```

JSHash 的运算比较复杂，如果对效果要求不是特别高的话 SDBMHash 是一个很好的选择。

三、排列的 Hash 函数

准确的说，这里我们的研究不再仅仅局限在“Hashing”的工作，而是进化到一个“numerize”的过程，也就是说我们可以在排列和 1 到 A_n^m 的自然数之间建立一一对应的关系。这样我们就可以利用这个关系来直接定址，或者用作 Hash 函数；在基于状态压缩的动态规划算法中也能用上。

1．背景知识

自然数的 p 进制表示法我们已经很熟悉了，即：

$$n=\sum_{k=0}^{m}a_k p^k \quad , \quad 0\le a_k \le p$$

比如 $p=2$ 便是二进制数， $p=10$ 便是十进制数。

引理：$\forall n\in N^*$ ， $n!=1+\sum_{k=1}^{n-1}k\cdot k!$。

证明：对 n 使用数学归纳法。

1) $n=1$时，等式显然成立。

2) 假设 $n=m$ 时等式成立，即 $m!=1+\sum_{k=1}^{m-1}k\cdot k!$。

则 $n=m+1$ 时，

$$n!=(m+1)!=(m+1)m!=m\cdot m!+m!=m\cdot m!+1+\sum_{k=1}^{m-1}k\cdot k!=1+\sum_{k=1}^{m}k\cdot k!$$

即 $n=m+1$ 时等式亦成立。

3) 综上所述， $\forall n\in N^*$ ， $n!=1+\sum_{k=1}^{n-1}k\cdot k!$ 成立。

把这个式子变形一下：

$$n!-1=(n-1)(n-1)!+(n-2)(n-2)!+\cdots+2\cdot 2!+1\cdot 1!$$

上式和 $p^n-1=(p-1)p^{n-1}+(p-1)p^{n-2}+\cdots+(p-1)$ 类似。不难证明，从 0 到 $n!-1$ 的任何自然数 m 可唯一地表示为

$$m=a_{n-1}(n-1)!+a_{n-2}(n-2)!+\cdots+a_1 1!$$

其中 $0\le a_i\le i$ ，$i=1,2,\ldots,n-1$。甚至在式子后面加上一个 $a_0 0!$ 也无妨，在后面我们把这一项忽略掉。所以从 0 到 $n!-1$ 的 $n!$ 个自然数与

$$(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1) \qquad (*)$$

一一对应。另一方面，不难从 m 算出 $a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1$。

我们可以把序列 $(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1)$ 理解为一个“变进制数”，也就是第一位二进制，第二位三进制，……，第 i 位 $i+1$ 进制，……，第 $n-1$ 位 n 进制。这样，我们就可以方便的使用类似“除 p 取余法”的方法从一个自然数 m 算出序列 $(a_{n-1},a_{n-2},a_{n-3},\ldots,a_2,a_1)$。由于这样的序列共有 $n!$ 个，我们很自然的想到把这 $n!$ 个序列和 n 个元素的全排列建立一一对应。

2．全排列与自然数之一一对应

为了方便起见，不妨设 n 个元素为 $1,2,\ldots,n$。对应的规则如下：设序列（*）对应的某一排列 $p=p_1p_2\cdots p_n$ ，其中 a_i 可以看做是排列 p 中数 $i+1$ 所在位置右边比 $i+1$ 小的数的个数。以排列 4213 为例，它是元素 1,2,3,4 的一个排列。4 的右边比 4 小的数的数目为 3，所以 $a_3=3$。3 右边比 3 小的数的数目为 0，即 $a_2=0$。同理 $a_1=1$。所以排列 4213 对应于变进制的 301，也就是十进制的 19；反过来也可以从 19 反推到 301，再反推到排列 4213。

3．更一般性的排列

受到这个思路启发，我们同样可以把更一般性的排列与自然数之间建立一一对应关系。想一想从 n 个元素中选 m 个的排列数 A_n^m 的公式是怎么来的？根据乘法原理，我们有

$$A_n^m = n(n-1)(n-2)\cdots(n-m+1)$$

这是由于在排列的第 1 个位置有 n 种选择，在排列的第 2 个位置有 $n-1$ 种选择，……，在排列的第 m 个位置有 $n-m+1$ 种选择。既然这样，我们可以定义一种“m-n 变进制数”，使其第 1 位是 $n-m+1$ 进制，第 2 位是 $n-m+2$ 进制，……，第 m 位是 n 进制。这样，0 到 A_n^m-1 之间的任意一个自然数 k 都可以唯一地表示成：

$$k = a_m A_{n-1}^{m-1} + a_{m-1} A_{n-2}^{m-2} + \cdots + a_2 A_{n-m+1}^1 + a_1$$

其中 $0 \le a_i \le n-m+i-1$，$1 \le i \le m$。注意到 $A_n^m - 1 = \sum_{k=1}^{m} (n-m+k-1) A_{n-m+k-1}^{k-1}$（证明略，可直接变形结合前面的引理推得），所以从 0 到 A_n^m-1 的 A_n^m 个自然数可以与序列

$$(a_m, a_{m-1}, a_{m-2}, \ldots, a_2, a_1)$$

一一对应。类似地，可以用取余法从自然数 k 算出 $a_m, a_{m-1}, a_{m-2}, \ldots, a_2, a_1$。

我们设 n 个元素为 $1,2,\ldots,n$，从中取出 m 个。对应关系如下：维护一个首元素下标为 0 的线性表 L，初始时 $L=(1,2,\ldots,n)$。对于某一排列 $p = p_1 p_2 \cdots p_m$，我们从 p_1 开始处理。首先在 L 中找到 p_1 的下标记为 a_m，然后删除 $L[a_m]$；接着在 L 中找到 p_2 的下标记为 a_{m-1}，然后删除 $L[a_{m-1}]$……直到 $L[a_1]$ 被删除为止。以在 5 个元素 1,2,3,4,5 中取出 2,4,3 为例，这时 $n=5, m=3$。首先在 L 中取出 2，记下 $a_3=1$，L 变为 1,3,4,5；在 L 中取出 4，记下 $a_2=2$，L 变为 1,3,5；在 L 中取出 3，记下 $a_1=1$，L 变为 1,5。因此排列 243 对应于“3-5 变进制数”

121，即十进制数 19；反过来也可以从十进制数 19 反推到 121，再反推到排列 243。各序列及其对应的排列如下表：

$p_1p_2p_3$	$a_3a_2a_1$	N	$p_1p_2p_3$	$a_3a_2a_1$	N
123	000	0	341	220	30
124	001	1	342	221	31
125	002	2	345	222	32
132	010	3	351	230	33
134	011	4	352	231	34
135	012	5	354	232	35
142	020	6	412	300	36
143	021	7	413	301	37
145	022	8	415	302	38
152	030	9	421	310	39
153	031	10	423	311	40
154	032	11	425	312	41
213	100	12	431	320	42
214	101	13	432	321	43
215	102	14	435	322	44
231	110	15	451	330	45
234	111	16	452	331	46
235	112	17	453	332	47
241	120	18	512	400	48
243	121	19	513	401	49
245	122	20	514	402	50
251	130	21	521	410	51
253	131	22	523	411	52
254	132	23	524	412	53
312	200	24	531	420	54
314	201	25	532	421	55
315	202	26	534	422	56
321	210	27	541	430	57
324	211	28	542	431	58
325	212	29	543	432	59

【总结】

本文对几个常用的 Hash 函数进行了总结性的介绍和分析，并将其延伸到应用更加广泛的“与自然数建立一一对应”的过程。Hash 是一种相当有效的数据结构，充分体现了“空间换时间”的思想。在如今竞赛中内存限制越来越松的情况下，要做到充分利用内存空间来换取宝贵的时间，Hash 能够给我们很大帮

助。我们应当根据题目的特点，选择适合题目的数据结构来优化算法。对于组合与自然数的一一对应关系，我还没有想到好的方法，欢迎大家讨论。

【参考文献】

[1] Thomas H Cormen, Charles E Leiserson, Ronald L Riverst, Clifford Stein. *Introduction to Algorithms.* Second Edition. The MIT Press, 2001

[2] 刘汝佳，黄亮. 《算法艺术与信息学竞赛》. 北京：清华大学出版社，2004

[3] 卢开澄，卢华明. 《组合数学》（第 3 版）. 北京：清华大学出版社，2002

【附录】

常用的字符串 Hash 函数之源代码：

```
// RS Hash Function
unsigned int RSHash(char *str)
{
	unsigned int b = 378551;
	unsigned int a = 63689;
	unsigned int hash = 0;

	while (*str)
	{
		hash = hash * a + (*str++);
		a *= b;
	}

	return (hash & 0x7FFFFFFF);
}

// JS Hash Function
unsigned int JSHash(char *str)
{
	unsigned int hash = 1315423911;

	while (*str)
	{
		hash ^= ((hash << 5) + (*str++) + (hash >> 2));
	}

	return (hash & 0x7FFFFFFF);
}

// P. J. Weinberger Hash Function
unsigned int PJWHash(char *str)
{
```

```
	unsigned int BitsInUnignedInt = (unsigned int)(sizeof(unsigned int) * 8);
	unsigned int ThreeQuarters    = (unsigned int)((BitsInUnignedInt  * 3) /
4);
	unsigned int OneEighth        = (unsigned int)(BitsInUnignedInt / 8);
	unsigned int HighBits         = (unsigned int)(0xFFFFFFFF) <<
(BitsInUnignedInt - OneEighth);
	unsigned int hash             = 0;
	unsigned int test             = 0;

	while (*str)
	{
		hash = (hash << OneEighth) + (*str++);
		if ((test = hash & HighBits) != 0)
		{
			hash = ((hash ^ (test >> ThreeQuarters)) & (~HighBits));
		}
	}

	return (hash & 0x7FFFFFFF);
}

// ELF Hash Function
unsigned int ELFHash(char *str)
{
	unsigned int hash = 0;
	unsigned int x    = 0;

	while (*str)
	{
		hash = (hash << 4) + (*str++);
		if ((x = hash & 0xF0000000L) != 0)
		{
			hash ^= (x >> 24);
			hash &= ~x;
		}
	}

	return (hash & 0x7FFFFFFF);
}

// BKDR Hash Function
unsigned int BKDRHash(char *str)
{
	unsigned int seed = 131; // 31 131 1313 13131 131313 etc..
	unsigned int hash = 0;

	while (*str)
	{
		hash = hash * seed + (*str++);
	}

	return (hash & 0x7FFFFFFF);
}

// SDBM Hash Function
unsigned int SDBMHash(char *str)
{
	unsigned int hash = 0;
```

```
	while (*str)
	{
		hash = (*str++) + (hash << 6) + (hash << 16) - hash;
	}

	return (hash & 0x7FFFFFFF);
}

// DJB Hash Function
unsigned int DJBHash(char *str)
{
	unsigned int hash = 5381;

	while (*str)
	{
		hash += (hash << 5) + (*str++);
	}

	return (hash & 0x7FFFFFFF);
}

// AP Hash Function
unsigned int APHash(char *str)
{
	unsigned int hash = 0;
	int i;

	for (i=0; *str; i++)
	{
		if ((i & 1) == 0)
		{
			hash ^= ((hash << 7) ^ (*str++) ^ (hash >> 3));
		}
		else
		{
			hash ^= (~((hash << 11) ^ (*str++) ^ (hash >> 5)));
		}
	}

	return (hash & 0x7FFFFFFF);
}
```

左偏树的特点及其应用

广东省中山市第一中学 黄源河

【摘要】

本文较详细地介绍了左偏树的特点以及它的各种操作。

第一部分提出可并堆的概念，指出二叉堆的不足，并引出左偏树。第二部分主要介绍了左偏树的定义和性质。第三部分详细地介绍了左偏树的各种操作，并给出时间复杂度分析。第四部分通过一道例题，说明左偏树在当今信息学竞赛中的应用。第五部分对各种可并堆作了一番比较。最后总结出左偏树的特点以及应用前景。

【关键字】 左偏树 可并堆 优先队列

【目录】

【正文】

一、引言

优先队列在信息学竞赛中十分常见，在统计问题、最值问题、模拟问题和贪心问题等等类型的题目中，优先队列都有着广泛的应用。二叉堆是一种常用的优先队列，它编程简单，效率高，但如果问题需要对两个优先队列进行合并，二叉堆的效率就无法令人满意了。本文介绍的左偏树，可以很好地解决这类问题。

二、左偏树的定义和性质

在介绍左偏树之前，我们先来明确一下优先队列和可并堆的概念。

2.1 优先队列，可并堆

2.1.1 优先队列的定义

优先队列(Priority Queue)是一种抽象数据类型(ADT)，它是一种容器，里面有一些元素，这些元素也称为队列中的节点(node)。优先队列的节点至少要包含一种性质：有序性，也就是说任意两个节点可以比较大小。为了具体起见我们假设这些节点中都包含一个键值(key)，节点的大小通过比较它们的键值而定。优先队列有三个基本的操作：插入节点(Insert)，取得最小节点(Minimum)和删除最小节点(Delete-Min)。

2.1.2 可并堆的定义

可并堆(Mergeable Heap)也是一种抽象数据类型，它除了支持优先队列的三个基本操作(Insert, Minimum, Delete-Min)，还支持一个额外的操作——合并操作：

$$H \leftarrow \text{Merge}(H_1, H_2)$$

Merge( ) 构造并返回一个包含 H_1 和 H_2 所有元素的新堆 H。

前面已经说过，如果我们不需要合并操作，则二叉堆是理想的选择。可惜合并二叉堆的时间复杂度为 O(n)，用它来实现可并堆，则合并操作必然成为算法的瓶颈。左偏树(Leftist Tree)、二项堆(Binomial Heap) 和 Fibonacci 堆(Fibonacci Heap) 都是十分优秀的可并堆。本文讨论的是左偏树，在后面我们将看到各种可并堆的比较。

2.2 左偏树的定义

左偏树(Leftist Tree)是一种可并堆的实现。左偏树是一棵二叉树，它的节点除了和二叉树的节点一样具有左右子树指针(left, right) 外，还有两个属性：键值和距离(dist)。键值上面已经说过，是用于比较节点的大小。距离则是如下定义的：

节点 i 称为**外节点(external node)**，当且仅当节点 i 的左子树或右子树为空 (left(i) = NULL 或 right(i) = NULL)；节点 i 的**距离(dist(i))**是节点 i 到它的后代中，最近的外节点所经过的边数。特别的，如果节点 i 本身是外节点，则它的距离为 0；而空节点的距离规定为-1 (dist(NULL) = -1)。在本文中，有时也提到一棵左偏树的距离，这指的是该树根节点的距离。

左偏树满足下面两条基本性质：

[性质 1] 节点的键值小于或等于它的左右子节点的键值。

即 **key(i)≤key(parent(i))** 这条性质又叫**堆性质**。符合该性质的树是**堆有序**的**(Heap-Ordered)**。有了性质 1，我们可以知道左偏树的根节点是整棵树的最小节点，于是我们可以在 O(1) 的时间内完成取最小节点操作。

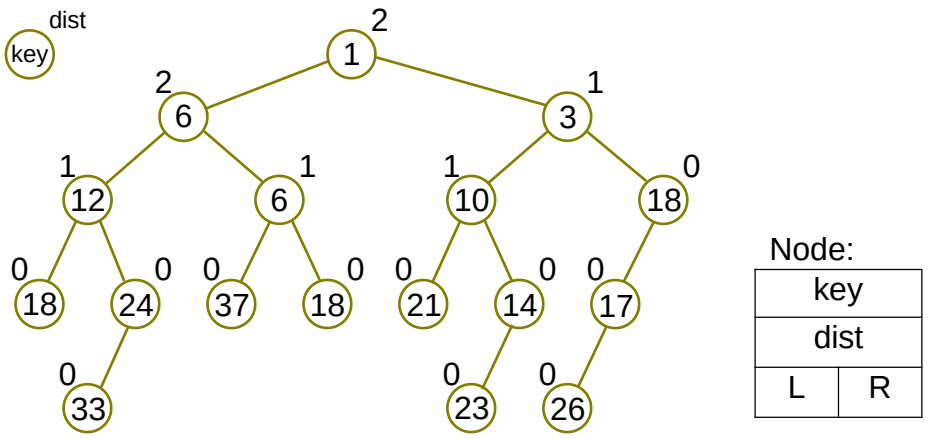
[性质 2] 节点的左子节点的距离不小于右子节点的距离。

即 **dist(left(i))≥dist(right(i))** 这条性质称为**左偏性质**。性质 2 是为了使我们可以以更小的代价在优先队列的其它两个基本操作（插入节点、删除最小节点）进行后维持堆性质。在后面我们就会看到它的作用。

这两条性质是对每一个节点而言的，因此可以简单地从中得出，左偏树的左右子树都是左偏树。

由这两条性质，我们可以得出左偏树的定义：**左偏树是具有左偏性质的堆有序二叉树**。

下图是一棵左偏树：



2.3 左偏树的性质

在前面一节中，本文已经介绍了左偏树的两个基本性质，下面本文将介绍左偏树的另外两个性质。

我们知道，一个节点必须经由它的子节点才能到达外节点。由于性质 2，一个节点的距离实际上就是这个节点一直沿它的右边到达一个外节点所经过的边数，也就是说，我们有

[性质 3] 节点的距离等于它的右子节点的距离加 1。

即 **dist(i) = dist(right(i)) + 1** 外节点的距离为 0，由于性质 2，它的右子节点必为空节点。为了满足性质 3，故前面规定空节点的距离为-1。

我们的印象中，平衡树是具有非常小的深度的，这也意味着到达任何一个节点所经过的边数很少。左偏树并不是为了快速访问所有的节点而设计的，它的目的是快速访问最小节点以及在对树修改后快速的恢复堆性质。从图中我们可以看到它并不平衡，由于性质 2 的缘故，它的结构偏向左侧，不过距离的概

念和树的深度并不同，左偏树并不意味着左子树的节点数或是深度一定大于右子树。

下面我们来讨论左偏树的距离和节点数的关系。

[引理 1] 若左偏树的距离为一定值，则节点数最少的左偏树是完全二叉树。

证明：由性质 2 可知，当且仅当对于一棵左偏树中的每个节点 i，都有 dist(left(i)) = dist(right(i)) 时，该左偏树的节点数最少。显然具有这样性质的二叉树是完全二叉树。

[定理 1] 若一棵左偏树的距离为 k，则这棵左偏树至少有 2^{k+1}-1 个节点。

证明：由引理 1 可知，当这样的左偏树节点数最少的时候，是一棵完全二叉树。距离为 k 的完全二叉树高度也为 k，节点数为 2^{k+1}-1，所以距离为 k 的左偏树至少有 2^{k+1}-1 个节点。

作为定理 1 的推论，我们有：

[性质 4] 一棵 N 个节点的左偏树距离最多为⌊log(N+1)⌋-1。

证明：设一棵 N 个节点的左偏树距离为 k，由定理 1 可知，$N \geq 2^{k+1}-1$，因此 $k \leq \lfloor \log(N+1) \rfloor - 1$。

有了上面的 4 个性质，我们可以开始讨论左偏树的操作了。

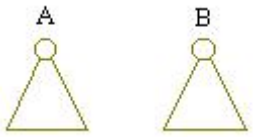
三、左偏树的操作

本章将讨论左偏树的各种操作，包括插入新节点、删除最小节点、合并左偏树、构建左偏树和删除任意节点。由于各种操作都离不开合并操作，因此我们先讨论合并操作。

3.1 左偏树的合并

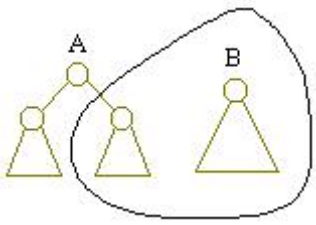
C ← Merge(A,B)

Merge( ) 把 A,B 两棵左偏树合并，返回一棵新的左偏树 C，包含 A 和 B 中的所有元素。在本文中，一棵左偏树用它的根节点的指针表示。



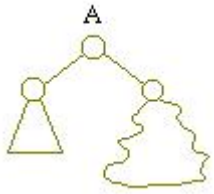
在合并操作中，最简单的情况是其中一棵树为空（也就是，该树根节点指针为 NULL）。这时我们只须要返回另一棵树。

若 A 和 B 都非空，我们假设 A 的根节点小于等于 B 的根节点（否则交换 A,B），把 A 的根节点作为新树 C 的根节点，剩下的事就是合并 A 的右子树 right(A) 和 B 了。



right(A) ← Merge(right(A), B)

合并了 right(A) 和 B 之后，right(A) 的距离可能会变大，当 right(A) 的距离大于 left(A) 的距离时，左偏树的性质 2 会被破坏。在这种情况下，我们只须要交换 left(A) 和 right(A)。



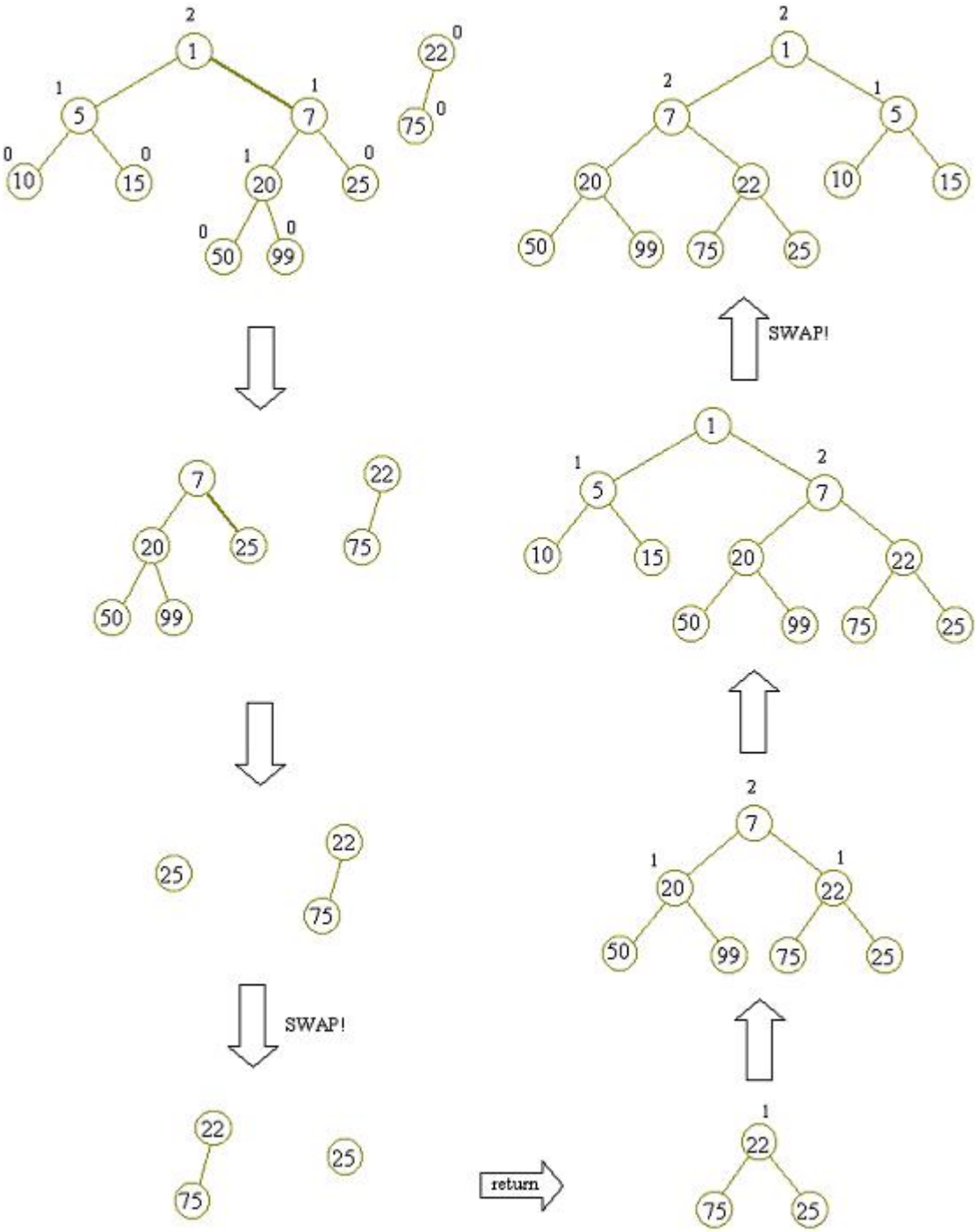
若 dist(left(A)) > dist(right(A))，交换 left(A) 和 right(A)

最后，由于 right(A) 的距离可能发生改变，我们必须更新 A 的距离：

$$dist(A) \leftarrow dist(right(A)) + 1$$

不难验证，经这样合并后的树 C 符合性质 1 和性质 2，因此是一棵左偏树。至此左偏树的合并就完成了。

下图是一个合并过程的示例：



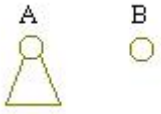
合并流程

我们可以用下面的代码描述左偏树的合并过程：

```
Function Merge(A, B)
        If A = NULL Then return B
        If B = NULL Then return A
        If key(B) < key(A) Then swap(A, B)
        right(A) ← Merge(right(A), B)
        If dist(right(A)) > dist(left(A)) Then
                swap(left(A), right(A))
        If right(A) = NULL Then dist(A) ← 0
        Else dist(A) ← dist(right(A)) + 1
        return A
End Function
```

下面我们来分析合并操作的时间复杂度。从上面的过程可以看出，每一次递归合并的开始，都需要分解其中一棵树，总是把分解出的右子树参加下一步的合并。根据性质 3，一棵树的距离决定于其右子树的距离，而右子树的距离在每次分解中递减，因此每棵树 A 或 B 被分解的次数分别不会超过它们各自的距离。根据性质 4，分解的次数不会超过$\lfloor \log(N_1+1) \rfloor + \lfloor \log(N_2+1) \rfloor -2$，其中 N_1 和 N_2 分别为左偏树 A 和 B 的节点个数。因此合并操作最坏情况下的时间复杂度为 $O(\lfloor \log(N_1+1) \rfloor + \lfloor \log(N_2+1) \rfloor -2) = O(\log N_1 + \log N_2)$。

3.2 插入新节点

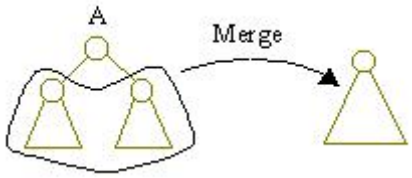


单节点的树一定是左偏树，因此向左偏树插入一个节点可以看作是对两棵左偏树的合并。下面是插入新节点的代码：

```
Procedure Insert(x, A)
        B ← MakeIntoTree(x)
        A ← Merge(A, B)
End Procedure
```

由于合并的其中一棵树只有一个节点，因此插入新节点操作的时间复杂度是 O(log n)。

3.3 删除最小节点



由性质 1，我们知道，左偏树的根节点是最小节点。在删除根节点后，剩下的两棵子树都是左偏树，需要把他们合并。删除最小节点操作的代码也非常简单：

```
Function DeleteMin(A)
        t ← key(root(A))
        A ← Merge(left(A), right(A))
        return t
End Function
```

由于删除最小节点后只需进行一次合并，因此删除最小节点的时间复杂度也为 O(log n)。

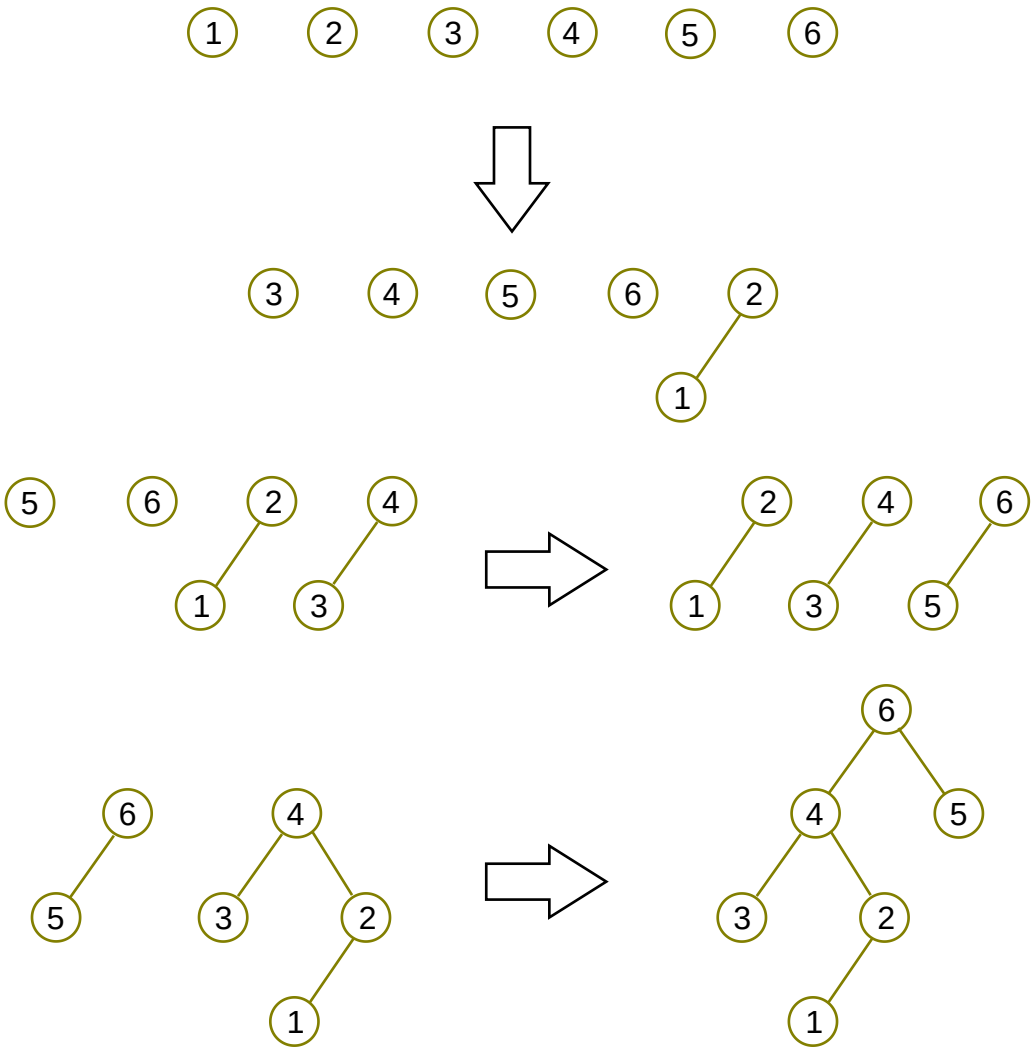
3.4 左偏树的构建

将 n 个节点构建成一棵左偏树，这也是一个常用的操作。

算法一 暴力算法——逐个节点插入，时间复杂度为 O(n log n)。

算法二 仿照二叉堆的构建算法，我们可以得到下面这种算法：

- 将 n 个节点（每个节点作为一棵左偏树）放入先进先出队列。
- 不断地从队首取出两棵左偏树，将它们合并之后加入队尾。
- 当队列中只剩下一棵左偏树时，算法结束。



构建流程

下面分析算法二的时间复杂度。假设 $n=2^k$，则：

前$\frac{n}{2}$次和并的是两棵只有 1 个节点的左偏树。

接下来的$\frac{n}{4}$次合并的是两棵有 2 个节点的左偏树。

接下来的$\frac{n}{8}$次合并的是两棵有 4 个节点的左偏树。

……

接下来的$\frac{n}{2^i}$次合并的是两棵有 2^{i-1} 个节点的左偏树。

合并两棵 2^i 个节点的左偏树时间复杂度为 O(i)，因此算法二的总时间复杂度为：

$$\frac{n}{2}*O(1)+\frac{n}{4}*O(2)+\frac{n}{8}*O(3)+\ldots=O(n*\sum_{i=1}^{k}\frac{i}{2^i})=O(n*(2-\frac{k+2}{2^k}))=O(n)。$$

3.5 删除任意已知节点

接下来是关于删除任意已知节点的操作。之所以强调“已知”，是因为这里所说的任意节点并不是根据它的键值找出来的，左偏树本身除了可以迅速找到最小节点外，不能有效的搜索指定键值的节点。故此，我们不能要求：请删除所有键值为 100 的节点。

前面说过，优先队列是一种容器。对于通常的容器来说，一旦节点被放进去以后，容器就完全拥有了这个节点，每个容器中的节点具有唯一的对象掌握它的拥有权（ownership）。对于这种容器的应用，优先队列只能删除最小节点，因为你根本无从知道它的其它节点是什么。

但是优先队列除了作为一种容器外还有另一个作用，就是可以找到最小节点。很多应用是针对这个功能的，它们并没有将拥有权完全转移给优先队列，而是把优先队列作为一个最小节点的选择器，从一堆节点中依次将它们选出

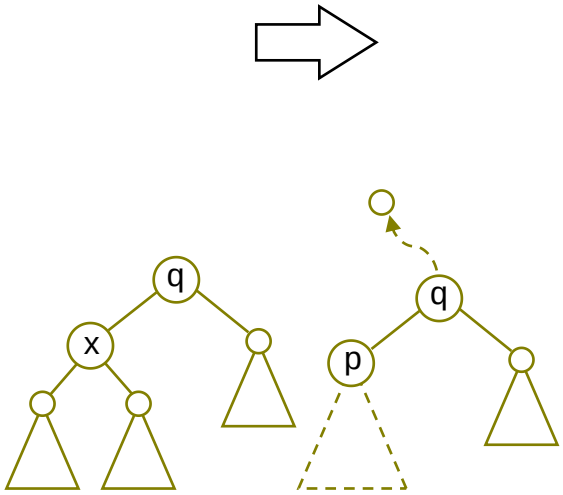
来。这样一来节点的拥有权就可能同时被其它对象掌握。也就是说某个节点虽不是最小节点，不能从优先队列那里“已知”，但却可以从其它的拥有者那里“已知”。

这种优先队列的应用也是很常见的。设想我们有一个闹钟，它可以记录很多个响铃时间，不过由于时间是线性的，铃只能一个个按先后次序响，优先队列就很适合用来作这样的挑选。另一方面使用者应该可以随时取消一个“已知”的响铃时间，这就需要进行任意已知节点的删除操作了。

我们的这种删除操作需要指定被删除的节点，这和原来的删除根节点的操作是兼容的，因为根节点肯定是已知的。上面已经提过，在删除一个节点以后，将会剩下它的两棵子树，它们都是左偏树，我们先把它们合并成一棵新的左偏树。

p ← Merge(left(x), right(x))

现在 p 指向了这颗新的左偏树，如果我们删除的是根节点，此时任务已经完成了。不过，如果被删除节点 x 不是根节点就有点麻烦了。这时 p 指向的新树的距离有可能比原来 x 的距离要大或小，这势必有可能影响原来 x 的父节点 q 的距离，因为 q 现在成为新树 p 的父节点了。于是就要仿照合并操作里面的做法，对 q 的左右子树作出调整，并更新 q 的距离。这一过程引起了连锁反应，我们要顺着 q 的父节点链一直往上进行调整。



新树 p 的距离为 dist(p)，如果 dist(p)+1 等于 q 的原有距离 dist(q)，那么不管 p 是 q 的左子树还是右子树，我们都不需要对 q 进行任何调整，此时删除操作也就完成了。

如果 dist(p)+1 小于 q 的原有距离 dist(q)，那么 q 的距离必须调整为 dist(p)+1，而且如果 p 是左子树的话，说明 q 的左子树距离比右子树小，必须交换子树。由于 q 的距离减少了，所以 q 的父节点也要做出同样的处理。

剩下就是另外一种情况了，那就是 p 的距离增大了，使得 dist(p)+1 大于 q 的原有距离 dist(q)。在这种情况下，如果 p 是左子树，那么 q 的距离不会改变，此时删除操作也可以结束了。如果 p 是右子树，这时有两种可能：一种是 p 的距离仍小于等于 q 的左子树距离，这时我们直接调整 q 的距离就行了；另一种是 p 的距离大于 q 的左子树距离，这时我们需要交换 q 的左右子树并调整 q 的距离，交换完了以后 q 的右子树是原来的左子树，它的距离加 1 只能等于

或大于 q 的原有距离，如果等于成立，删除操作可以结束了，否则 q 的距离将增大，我们还要对 q 的父节点做出相同的处理。

删除任意已知节点操作的代码如下：

```
Procedure Delete(x)
        q ← parent(x)
        p ← Merge(left(x), right(x))
        parent(p) ← q
        If q ≠ NULL and left(q) = x Then
                left(q) ← p
        If q ≠ NULL and right(q) = x Then
                right(q) ← p
        While q ≠ NULL Do
                If dist(left(q)) < dist(right(q))
Then
                        swap(left(q), right(q))
                If dist(right(q))+1 = dist(q) Then
                        Exit Procedure
                dist(q) ← dist(right(q))+1
                p ← q
                q ← parent(q)
        End While
```

下面分两种情况讨论删除操作的时间复杂度。

情况 1：p 的距离减小了。在这种情况下，由于 q 的距离只能缩小，当循环结束时，要么根节点处理完了，q 为空；要么 p 是 q 的右子树并且 dist(p)+1=dist(q)；如果 dist(p)+1>dist(q)，那么 p 一定是 q 的左子树，否则会出现 q 的右子树距离缩小了，但是加 1 以后却大于 q 的距离的情况，不符合左偏树的性质 3。不论哪种情况，删除操作都可以结束了。注意到，每一次循环，p 的距离都会加 1，而在循环体内，dist(p)+1 最终将成为某个节点的距

离。根据性质 4，任何的距离都不会超过 log n，所以循环体的执行次数不会超过 log n。

情况 2：p 的距离增大了。在这种情况下，我们将必然一直从右子树向上调整，直至 q 为空或 p 是 q 的左子树时停止。一直从右子树升上来这个事实说明了循环的次数不会超过 log n（性质 4）。

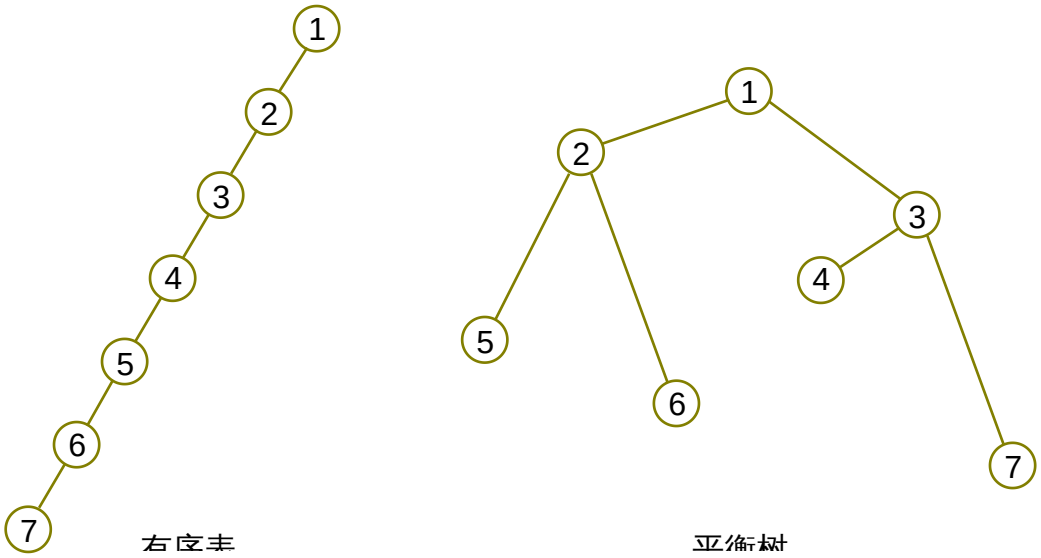
最后我们看到这样一个事实，就是这两种情况只会发生其中一个。如果某种情况的调整结束后，我们已经知道要么 q 为空，要么 dist(p)+1 = dist(q)，要么 p 是 q 的左子树。这三种情况都不会导致另一情况发生。直观上来讲，如果合并后的新子树导致了父节点的一系列距离调整的话，要么就一直是往小调整，要么是一直往大调整，不会出现交替的情况。

我们已经知道合并出新子树 p 的复杂度是 O(log n)，向上调整距离的复杂度也是 O(log n)，故删除操作的最坏情况的时间复杂度是 O(log n)。如果左偏树非常倾斜，实际应用情况下要比这个快得多。

3.6 小结

本章介绍了左偏树的各种操作，我们可以看到，左偏树作为可并堆的实现，它的各种操作性能都十分优秀，且编程复杂度比较低，可以说是一个“性价比”十分高的数据结构。左偏树之所以是很好的可并堆实现，是因为它能够捕捉到具有堆性质的二叉树里面的一些其它有用信息，没有将这些信息浪费掉。根据堆性质，我们知道，从根节点向下到任何一个外节点的路径都是有序的。存在越长的路径，说明树的整体有序性越强，与平衡树不同（平衡树根本不允许

有很长的路径），左偏树尽大约一半的可能保留了这个长度，并将它甩向左侧，利用它来缩短节点的距离以提高性能。这里我们不进行严格的讨论，左偏树作为一个例子大致告诉我们：放弃已有的信息意味着算法性能上的牺牲。下面是最好的左偏树：有序表（插入操作是按逆序发生的，自然的有序性被保留了）和最坏的左偏树：平衡树（插入操作是按正序发生的，自然的有序性完全被放弃了）。



有序表　　平衡树

四、左偏树的应用

4.1 例——数字序列（Baltic 2004）

[问题描述][一]

给定一个整数序列 $a_1, a_2, \ldots, a_n$，求一个不下降序列 $b_1 \le b_2 \le \ldots \le b_n$，使得数列$\{a_i\}$和$\{b_i\}$的各项之差的绝对值之和 $|a_1 - b_1| + |a_2 - b_2| + \ldots + |a_n - b_n|$ 最小。

[数据规模] $1 \le n \le 10^6$, $0 \le a_i \le 2*10^9$

[初步分析]

我们先来看看两个最特殊的情况：

1． a[1]≤a[2]≤…≤a[n]，在这种情况下，显然最优解为 b[i]=a[i]；

2． a[1]≥a[2]≥…≥a[n]，这时，最优解为 b[i]=x，其中 x 是数列 a 的中位数[二]。

于是我们可以初步建立起这样一个思路：

把 1…n 划分成 m 个区间：[q[1],q[2]-1]，[q[2],q[3]-1]，……，[q[m],q[m+1]-1][三]。每个区间对应一个解，b[q[i]] = b[q[i]+1] = … = b[q[i+1]-1] = w[i]，其中 w[i] 为 a[q[i]], a[q[i]+1], ... , a[q[i+1]-1] 的中位数。

显然，在上面第一种情况下 m=n，q[i]=i；在第二种情况下 m=1，q[1]=1。

这样的想法究竟对不对呢？应该怎样实现？

[一] 题目来源：Baltic OI 2004 Day 1, Sequence 本文对原题略微做了改动

[二] 为了方便讨论和程序实现，本文中提到的中位数，都是指数列中第⌊n/2⌋大的数

[三] 这里我们认为 q[m+1] = n+1

若某序列前半部分 a[1], a[2], …，a[n] 的最优解为 $(u,u,\ldots,u)$，后半部分 a[n+1], a[n+2], ...，a[m] 的最优解为 $(v,v,\ldots,v)$，那么整个序列的最优解是什么呢？若 $u\le v$，显然整个序列的最优解为 $(u,u,\ldots,u,v,v,\ldots,v)$。否则，设整个序列的最优解为 $(b[1],b[2],\ldots,b[m])$，则显然 $b[n]\le u$（否则我们把前半部分的解 $(b[1],b[2],\ldots,b[n])$ 改为 $(u,u,\ldots,u)$，由题设知整个序列的解不会变坏），同理 $b[n+1]\ge v$。接下来，我们将看到下面这个事实：

对于任意一个序列 $a[1],a[2],\ldots,a[n]$，如果最优解为 $(u,u,\ldots,u)$，那么在满足 $u\le u'\le b[1]$ 或 $b[n]\le u'\le u$ 的情况下，$(b[1],b[2],\ldots,b[n])$ 不会比 $(u',u',\ldots,u')$ 更优。

我们用归纳法证明 $u\le u'\le b[1]$ 的情况，$b[n]\le u'\le u$ 的情况可以类似证明。

当 n=1 时，u=a[1]，命题显然成立。

当 n>1 时，假设对于任意长度小于 n 的序列命题都成立，现在证明对于长度为 n 的序列命题也成立。首先把 (b[1], b[2], … b[n]) 改为 (b[1], b[1], … b[1])，这一改动将不会导致解变坏，因为如果解变坏了，由归纳假设可知 a[2],a[3],…,a[n]的中位数 w>u，这样的话，最优解就应该为 $(u,u,\ldots,u,w,w,\ldots,w)$，矛盾。然后我们再把$(b[1],b[1],\ldots,b[1])$改为 $(u',u',\ldots,u')$，由于 | a[1] - x | + | a[2] - x | + … + | a[n] - x | 的几何意义为数轴上点 x 到点 a[1], a[2], … a[n] 的距离之和，且 $u\le u'\le b[1]$，显然点 u′ 到各点的距离之和不会比点 b[1]到各点的距离之和大，也就是说，$(b[1],b[1],\ldots,b[n])$ 不会比 $(v,v,\ldots,v)$ 更优。（证毕）

再回到之前的论述，由于 b[n]≤u，作为上述事实的结论，我们可以得知，将(b[1],b[2],…,b[n])改为(b[n],b[n],…,b[n])，再将(b[n+1],b[n+2],…,b[m])改为(b[n+1],b[n+1],…,b[n+1])，并不会使解变坏。也就是说，整个序列的最优解为(b[n],b[n],…,b[n],b[n+1],b[n+1],…,b[n+1])。再考虑一下该解的几何意义，设整个序列的中位数为 w，则显然令 b[n]=b[n+1]=w 将得到整个序列的最优解，即最优解为 (w,w,…,w)。

分析到这里，我们一开始的想法已经有了理论依据，算法也不难构思了。

[算法描述]

延续我们一开始的思路，假设我们已经找到前 k 个数 a[1], a[2], …，a[k] (k<n) 的最优解，得到 m 个区间组成的队列，对应的解为 (w[1],w[2],…,w[m])，现在要加入 a[k+1]，并求出前 k+1 个数的最优解。首先我们把 a[k+1]作为一个新区间直接加入队尾，令 w[m+1]=a[k+1]，然后不断检查队尾两个区间的解 w[m] 和 w[m+1]，如果 w[m]＞w[m+1]，我们需要将最后两个区间合并，并找出新区间的最优解（也就是序列 a 中，下标在这个新区间内的各项的中位数）。重复这个合并过程，直至 w[1]≤w[2]≤…≤w[m]时结束，然后继续处理下一个数。

这个算法的正确性前面已经论证过了，现在我们需要考虑一下数据结构的选取。算法中涉及到以下两种操作：合并两个有序集以及查询某个有序集内的中位数。能较高效地支持这两种操作的数据结构有不少，一个比较明显的例子是二叉检索树(BST)，它的询问操作复杂度是 O(log n)，但合并操作不甚理想，采用启发式合并，总时间复杂度为 $O(n\log^2 n)$。

有没有更好的选择呢？通过进一步分析，我们发现，只有当某一区间内的中位数比后一区间内的中位数大时，合并操作才会发生，也就是说，任一区间与后面的区间合并后，该区间内的中位数不会变大。于是我们可以用最大堆来维护每个区间内的中位数，当堆中的元素大于该区间内元素的一半时，删除堆顶元素，这样堆中的元素始终为区间内较小的一半元素，堆顶元素即为该区间内的中位数。考虑到我们必须高效地完成合并操作，左偏树是一个理想的选择[一]。左偏树的询问操作时间复杂度为 O(1)，删除和合并操作时间复杂度都是 O(log n)，而询问操作和合并操作少于 n 次，删除操作不超过 n/2 次（因为删除操作只会在合并两个元素个数为奇数的堆时发生），因此用左偏树实现，可以把算法的时间复杂度降为 O(n log n)。

[小结]

这道题的解题过程对我们颇有启示。在应用左偏树解题时，我们往往会觉得题目无从下手，甚至与左偏树毫无关系，但只要我们对题目深入分析，加以适当的转化，问题终究会迎刃而解。这需要我们具有敏捷的思维以及良好的题感。

用左偏树解本题，相比较于前面 BST 的解法，时间复杂度和编程复杂度更低，这使我们不得不感叹于左偏树的神奇威力。这不是说左偏树就一定是最好的解法，就本题来说，解法有很多种，光是可并堆的解法，就可以用多种数据结构来实现，但左偏树相对于它们，还是有一定的优势的，这将在下一章详细讨论。

[一] 前面介绍的左偏树是最小堆，但在本题中，显然只需把左偏树的性质稍做修改，就可以实现最大堆了

五、左偏树与各种可并堆的比较

我们知道，左偏树是一种可并堆的实现。但是为什么我们要用左偏树实现可并堆呢？左偏树相对于其他可并堆有什么优点？本章将就这个问题展开讨论，介绍各种可并堆的特点，并对它们做出比较。

5.1 左偏树的变种——斜堆

这里我们要介绍左偏树的一个变种——斜堆(Skew Heap)。斜堆是一棵堆有序的二叉树，但是它不满足左偏性质，或者说，斜堆根本就没有“距离”这个概念——它不需要记录任何一个节点的距离。从结构上来说，所有的左偏树都是斜堆，但反之不然。

类似于左偏树，斜堆的各种操作也是在合并操作的基础上完成的，因此这里只介绍斜堆的合并操作，其他操作读者都可以仿照左偏树完成。

斜堆合并操作的递归合并过程和左偏树完全一样。假设我们要合并 A 和 B 两个斜堆，且 A 的根节点比 B 的根节点小，我们只需要把 A 的根节点作为合并后新堆的根节点，并将 A 的右子树与 B 合并。由于合并都是沿着最右路径进行的，经过合并之后，新堆的最右路径长度必然增加，这会影响下一次合并的效率。为了解决这一问题，左偏树在进行合并的同时，检查最右路径节点的距离，并通过交换左右子树，使整棵树的最右路径长度非常小。然而斜堆不记录节点的距离，那么应该怎样维护最右路径呢？我们采取的办法是，从下往上，沿着合并的路径，在每个节点处都交换左右子树。

下面是斜堆合并操作的代码：

```
Function Merge(A,B)
        If A = NULL Then return B
        If B = NULL Then return A
        If key(B) < key(A) Then swap(A,B)
        right(A) ← Merge(right(A), B)
        swap(left(A), right(A))
        return A
End Function
```

斜堆的这种维护方法也是行之有效的。通过不断交换左右子树，斜堆把最右路径甩向左边了。可以证明，斜堆合并操作的平摊时间复杂度为 O(log n)。这里略去详细的复杂度分析，感兴趣的读者可以自行参考相关的资料。

前面说过，斜堆的其他各种操作都和左偏树类似，因此斜堆各项操作的平摊时间复杂度都与左偏树相同。在空间上，由于斜堆不用记录节点的距离，因此它比左偏树的空间需求小一点。至于编程复杂度，两者都十分的低，不过斜堆的代码还是要比左偏树稍微简洁一些。至于斜堆的不足，大概可以算是它单次合并操作的时间复杂度可能退化为 O(n)，不过通常这并不影响算法的总时间复杂度。

总的来说，斜堆作为左偏树的变种，与左偏树并无优劣之分。在斜堆能派上用场的地方，左偏树同样能出色地实现算法。斜堆与左偏树之间的关系，可以类比于伸展树与 AVL 树之间的关系，只不过前两者的编程复杂度，并没有多大的差别。

5.2 左偏树与二叉堆的比较

二叉堆大概是我们最常用的一种优先队列了，它具有简洁，高效的特点，应用十分广泛。二叉堆和左偏树相比，除了合并操作外的各种操作，时间复杂度都一样，但在实际测试中，二叉堆往往比左偏树快一些。在空间上，由于二叉堆是完全二叉树，用数组表示法，可以剩下左右子树指针的空间，在这点上左偏树又吃了一亏。

介绍了二叉堆的优点，下面的这两个缺点也是不容忽视的：

1． 在进行插入、删除等操作后，二叉堆中元素的物理位置会发生改变。

2． 二叉堆的合并操作太慢，时间复杂度高达 O(n)。

当元素本身所占空间比较大时，频繁地移动元素物理位置将会影响时间效率。而有些时候，我们需要随时掌握某些元素的物理位置，以便对它进行访问或修改（比如实现删除任意已知节点操作），这时，二叉堆的第一个缺点将给我们造成一定的麻烦，我们不得不通过增加元素指针等手段来解决这个问题。而二叉堆合并效率的低下则是由其本身性质所造成的，采用启发式合并可以作为大多数情况下的一个优化，但效果仍不能令人满意。

至于二叉堆空间上的优势，也不是绝对的。数组表示法并不是在任何场合都适用的，比如当我们需要动态维护多个优先队列的时候，二叉堆不得不使用传统的指针表示法，这时二叉堆和左偏树在空间上就相差无几了。

二叉堆的上述特点，决定了它不适合作为可并堆的实现。客观的说，需要实现可并堆的题目在竞赛中并不多见，不过当我们遇到这类题目或者某些特殊的情况时，左偏树将会是比二叉堆更好的选择。

5.3 左偏树与其他可并堆的比较

前面已经提到过，可并堆可以用多种数据结构实现，左偏树并非唯一的选择。二项堆，Fibonacci 堆都是时间效率十分优秀的可并堆。

二项堆是由若干棵深度不同的二项树组成的森林。深度为 k 的二项树 B_k 由两棵二项树 B_{k-1} 连接根节点而成，有 2^k 个节点，并且满足堆性质。二项树组成二项堆的形式可以看作总节点数 n 的二进制表示形式，两个二项堆的合并可以看作二进制数的加法，这点很有启发意义。与左偏树类似，二项堆的各种操作也是在合并操作的基础上完成的。二项堆的结构和性质决定了它可以很高效地完成合并操作，与左偏树相比，虽然复杂度相同，但一般实际情况下二项堆比较快。但二项树实现取最小节点操作需要检查所有二项树的根节点，因此这项操作时间复杂度比左偏树高[一]。

Fibonacci 堆是一个很复杂的数据结构，与二项堆一样，它也是由一组堆有序的树构成的。所不同的是，Fibonacci 堆的树不一定是二项树，而且这些树是无序的，且树中兄弟节点的联系用双向循环链表来表示。Fibonacci 堆实现插入操作和合并操作只是简单地将两个 Fibonacci 堆的根表连在一起，因此这两个

[一] 这里其实有一个办法可以解决这个问题，就是随时记录最小节点，并只在插入、删除以及合并等操作进行的时候更新该记录，这样二项堆的取最小节点操作时间复杂度可以降为 O(1)

操作比其他可并堆都快。但在删除操作时，我们需要对 Fibonacci 堆进行维护，合并所有度数相同的树，这一步异常复杂，并且是最慢的。

有关二项堆和 Fibonacci 堆的详细介绍，有兴趣的读者可以参考相关资料，本文不再赘述。下表列出了各种可并堆的各项操作的时间复杂度[一]，已及它们的空间需求和编程复杂度。

项目	二叉堆	左偏树	二项堆	Fibonacci 堆
构建	O(n)	O(n)	O(n)	O(n)
插入	O(log n)	O(log n)	O(log n)	O(1)
取最小节点	O(1)	O(1)	O(log n)	O(1)
删除最小节点	O(log n)	O(log n)	O(log n)	O(log n)
删除任意节点	O(log n)	O(log n)	O(log n)	O(log n)
合并	O(n)	O(log n)	O(log n)	O(1)
空间需求	最小	较小	一般	较大
编程复杂度	最低	较低	较高	很高

从表中我们可以看出，Fibonacci 堆的时间复杂度非常低，如果我们不需要进行频繁的删除操作，用 Fibonacci 堆实现可并堆将会降低算法的时间复杂度。但实际上，删除操作往往是很重要的操作，而 Fibonacci 堆的删除操作比表中其它可并堆都慢，至于它的空间需求，也大得使人望而却步。更槽糕的是 Fibonacci 堆的编程复杂度太高了，在竞赛中使用实在是不理智的行为。

至于二项堆，虽然各项操作的时间效率都十分优秀，但空间需求和编程复杂度仍然比不上左偏树。和左偏树相比，二项堆在时间效率上的优势微乎其微，用左偏树代替二项堆，的确会牺牲一些算法性能，但换来的却是简洁的代

[一] 表中 Fibonacci 堆的“删除最小节点”和“删除任意节点”两个操作的时间复杂度均为平摊时间复杂度，而二项堆“插入”操作的平均时间复杂度为 O(1)，表中给出的是最坏时间复杂度

码，便于实现和调试的程序。在时间有限的竞赛中，左偏树无疑是更好的选择。

最后，我们应该认识到，虽然二项堆和 Fibonacci 堆某些操作的时间复杂度比左偏树低，但是在实际应用中，那些时间复杂度较高的操作往往会成为算法的瓶颈。比如前面的例题《数字序列》，改用二项堆或 Fibonacci 堆实现，总时间复杂度仍为 $O(n \log n)$，并不能起到降低算法时间复杂度的作用，实现难度反而增加了不少。

六、总结

至此，我们已经对左偏树有了深刻的认识。左偏树在时间效率上不如二项堆和 Fibonacci 堆，在空间效率上不如二叉堆，这样看来左偏树没有任何独树一帜的地方，似乎是个相当平庸的数据结构，但其实这正是左偏树的优势所在。正所谓“鱼与熊掌不可兼得”，时间复杂度、空间复杂度和编程复杂度，这三者之间很多时候是矛盾的。Fibonacci 堆时间复杂度最低，但编程复杂度让人无法接受；二叉堆的空间复杂度和编程复杂度都很低，但时间复杂度却是它的致命弱点。左偏树很好地协调了三者之间的矛盾，并且在存储性质上，没有二叉堆那样的缺陷，因此左偏树的适用范围十分广。左偏树不但可以高效方便地实现可并堆，更可以作为二叉堆的替代品，应用于各种优先队列，很多时候甚至比二叉堆更方便。

【感谢】

感谢余江伟同学及阮志远老师的热心帮助和修改意见。

【参考文献】

[1] 傅清祥，王晓东 算法与数据结构（第二版） 电子工业出版社

[2] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein Introduction to Algorithms (Second Edition) The MIT Press

[3] Mark Allen Weiss Data Structures and Algorithm Analysis in C (Second Edition) Pearson Education

【附录】

附录 I ：例题《数字序列》的原题

SEQUENCE

Short formulation. The number sequence is given. Your task is to construct the increasing sequence that approximates the given one in the best way. The best approximating sequence is the sequence with the least total deviation from the given sequence.

More precisely. Let $t_1, t_2, \ldots, t_N$ is the given number sequence. Your task is to construct the increasing number sequence $z_1 < z_2 < \ldots < z_N$.

The sum $|t_1 - z_1| + |t_2 - z_2| + \ldots + |t_N - z_N|$ should be a minimal feasible.

Input

There is the integer N ($1 \le N \le 1000000$) in the first line of input file seq.in. Each of the next N lines contains single integer – the given sequence element. There is t_K in the (K+1)-th line. Any element is satisfying to relation $0 \le t_K \le 2000000000$.

Output

The first line of output file seq.out must contain the single integer – the minimal possible total deviation. Each of the next N lines must contain single integer – the recurrent element of the best approximating sequence.

If there are several solutions, your program must output any one sequence with a least total deviation.

Example

seq.in	seq.out
7 9 4 8 20 14 15 18	13 6 7 8 13 14 15 18

附录 II ：例题《数字序列》左偏树解法的程序

```
PROGRAM Seq_LeftistTree;
TyPe
  node=
  record
    dist:byte;
    key,left,right:longint;
  end;
VaR
  nd:array[0..1000000]of node;
  stk,q:array[0..1000000]of longint;
  n,cl,i:longint;
Function merge(a,b:longint):longint;
  var
    c:longint;
  begin
    if (a=0)or(b<>0)and(nd[a].key<nd[b].key) then
    begin
      c:=a;
      a:=b;
      b:=c;
    end;
    merge:=a;
    if b=0 then exit;
    nd[a].right:=merge(nd[a].right,b);
    if nd[nd[a].left].dist<nd[nd[a].right].dist then
    begin
      c:=nd[a].left;
      nd[a].left:=nd[a].right;
      nd[a].right:=c;
    end;
    nd[a].dist:=nd[nd[a].right].dist+1;
  end;
Procedure print;
  var
    tot,i,j:longint;
  begin
    tot:=0;
    for i:=1 to cl do
      for j:=q[i-1]+1 to q[i] do
        inc(tot,abs(nd[j].key-nd[stk[i]].key));
    writeln(tot);
```

```
  end;
BeGiN
  assign(input,'seq.in');
   reset(input);
  assign(output,'seq.out');
   rewrite(output);
  readln(n);
  fillchar(nd,sizeof(nd),0);
  cl:=0;
  q[0]:=0;
  for i:=1 to n do
  begin
    readln(nd[i].key);
    dec(nd[i].key,i);
    inc(cl);
    stk[cl]:=i;
    q[cl]:=i;
    while (cl>1)and(nd[stk[cl]].key<nd[stk[cl-1]].key) do
    begin
      dec(cl);
      stk[cl]:=merge(stk[cl],stk[cl+1]);
      if odd(q[cl+1]-q[cl]) and odd(q[cl]-q[cl-1]) then
        stk[cl]:=merge(nd[stk[cl]].left,nd[stk[cl]].right);
      q[cl]:=q[cl+1];
    end;
  end;
  print;
  close(output);
EnD.
```

浅析非完美算法在信息学竞赛中的应用

湖南省长沙市长郡中学 胡伟栋

【目录】

【摘要】

非完美算法就是用算法正确性的少量损失来换取时间、空间效率以及编程复杂度的算法。本文介绍了非完美算法的几种重要方法及非完美算法的优缺点，从中，可以看出：并不是完全正确的算法就一定好过非完美算法，有可能因为非完美算法的不完全性，反而使不正确的算法在很多方面比正确算法表现得更好。

【关键字】

非完美算法 随机化贪心法 抽样测试法 部分忽略法

【正文】

一、引言

在平时，我们研究的算法基本都是完全正确的算法，我们所追求的，也是尽量使我们的算法正确。但在实际应用中，有很多情况都不会对数据进行天衣无缝的正确处理。如：图片、音频、视频的压缩存储，很多压缩率比较高的方法都是有损压缩；很多密码验证的方法都是采用多对一的运算方法，通过验证的不代表密码真正正确；搜索引擎所提供的搜索，并不能将所有满足条件的都找到……显然，我们不会因为它们的不完美而不使用它们，它们的存在，有着它们的实际意义：图片、音频、视频的压缩存储，如果不损失一些，不可能将文件压缩得很小；密码验证虽是多对一，但找到两个所对的是同一个何尝容易；搜索引擎虽不能搜索到100%的结果，但其方便、快捷是无以伦比的……

那么，在信息学竞赛中是否也可以用非完美算法解题呢？

本文试图介绍一些比较有用的常见非完美算法，并希望读者能从此得到一些启发。

在开始此文前，希望读者能认同一点：在信息学乃至整个计算机科学领域，不一定绝对正确的算法就是最好的算法，有可能一个在绝大多数情况下正确的算法(非完美算法)比一个完全正确的算法更有前途。或者，由于它没有完全正确的算法考虑得全面，而使得它的空间或时间使用得较少；或者，由于它没有正确的算法那么严谨，使得编程实现时较容易；或者，由于所用的知识没有正确算法那么深奥，使得它更容易被接受。

二、非完美算法的一些基本方法

1.1 随机化贪心[①]法

随机化贪心是目前用得较广泛的一种非完美算法。特别是对求较优解的题目，这种方法可以说是首选。

随机贪心，就是用随机与贪心结合，在算法的每一步，都尽量使决策取得优，但不一定是最优决策。通过多次运行，使得算法能取得一个较优的解。有时，由于决策的优劣判断较复杂，直接用多次随机也能得到较优的结果。

例：传染病控制[②]

- 题目大意

[①] 随机化贪心在周咏基前辈的《论随机化算法的原理与设计》中有写过，为了此文的完整性，这里将通过一个实例简要的说明。

[②] 题目来源：NOIP2003，原题请见附录。

给出一棵传染病传播树，其中根结点是被感染结点。每个时刻，被感染结点都会将传染病传播到它的子结点。但是，如果某时刻 *t* 切段 *A* 与其父结点 *B* 之间的边，则时刻 *t* 以后，传染病不会从 *B* 传染给 *A*(即 *A* 不会患病)。

每个时刻，只能切断一条传播途径。问每个时刻怎样切段传播途径，使最少的人被感染。

➢ 说明

本题的标准算法是搜索。关于如何用搜索解决此题，请参见附件《传染病控制解题报告》，此解题报告由周戈林同学提供。

➢ 分析

显然，如果某个结点已被感染，则以后没有必要切断它与它的父结点之间的传播途径，因为这样和不切断没有区别(即切断已被感染的结点与其父结点之间的传播途径是无效的)；如果在一个时刻，切断 *A* 与 *A* 的父结点之间的传播途径是有效的，切断 *B* 与 *B* 的父结点之间的传播途径也是有效的，同时，*B* 是 *A* 的子孙结点，则切断 *A* 与 *A* 的父结点的传播途径优于切断 *B* 与 *B* 的父结点的传播途径。由于第 *t* 时刻被感染的显然只可能是前 *t* 层的。因此可得到，第 *t* 时刻切断的必然是第 *t* 层与第 *t*+1 层之间的传播途径。

本题可以用随机贪心来做。

根据经验，每次将一个结点数最多的子树从原树中切开，结果会比较好。但这样并不能适应所有情况，有时，选结点数第二多的或第三多的反而能使以后的最终结果要好一些。所以，可以多次贪心，每次都随机的选一个能切断的

结点数较多的方案切(或者说每次使能切断的结点数多的方案数被切的几率大一些)，这样，虽然大多数情况下其结果都比贪心要差，但只要随机到一定的数量，这中间出现正确答案的概率就很大了。

上面加概率的随机贪心有一点难处理的就是怎样设置每种情况被选择的概率。其实，对于此题，只要将每种情况被选到的概率都设成同样大(即被选中的概率与其子结点数无关)就可以了。这样，这种方法就变成了纯粹的随机法，这种方法也能使此题得到满分。

小结

随机贪心是信息学竞赛中的一个重要工具，由于一般情况下它都是基于简单的贪心，所以往往编程复杂性会比较低，同时运行的时间复杂度也会比较低。随机多次是随机贪心的一个重要手段，通过多次运行，往往能使程序得到非常优的结果。

1.2 抽样测试法

抽样，即从统计总体中，任意抽出一部分单位作为样本，并以其结果推算总体的相应指标。

在某些题目中，题设会给出一系列需要测试的测试元(总体)S，若存在某一个测试元满足某个条件 A，则说 S 满足性质 P，若所有测试元都不满足条件 A，则说 S 不满足性质 P。

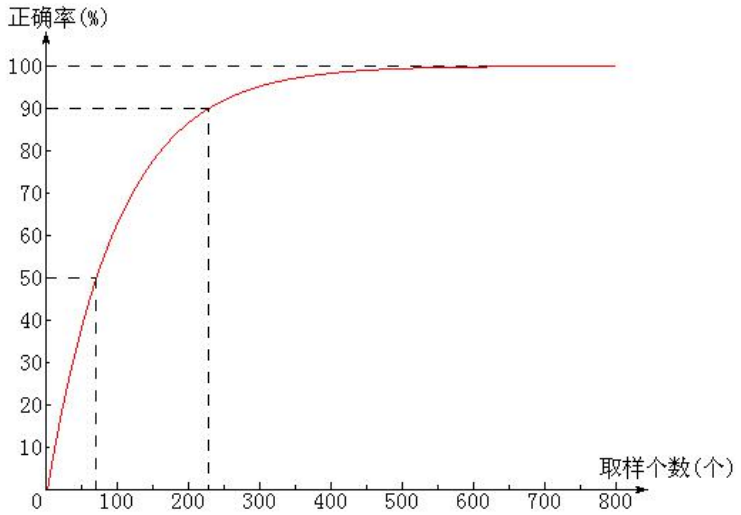
对于这个总体 S，判断它是否满足性质 P，通常要将 S 中所有的测试元全部列举出来才能知道。如果不列举出全部(哪怕只有一个没有列举到)且当前已

测试的测试元都不满足条件 *A*，则不敢保证剩下的测试元不满足条件，也就不能断定 *S* 是否满足性质 *P*。

但是，若总体具有下面的性质：它或者不含满足条件的测试元，或者满足条件的测试元的个数比较多。这时，只要选取一些具有代表性的很少一部分测试元进行测试，就是保证结果基本正确。所谓有代表性，就是指取一些很特殊的测试元，同时取一些不特殊的测试元——就像问卷调查一样，从各种工作岗位、各种年龄阶段的人群中取得的问卷调查结果往往比在同一个工作岗位且年龄相仿的人群中取得的调查结果更让人信服。

下面，我们来看一个例子：一个总体 *S* 中含有 10000 个测试元，其中有 100 个测试元满足条件 *A*，现在，若从 *S* 中取出 100 个测试元进行测试，则检查出 *S* 满足性质 *P* 的概率约为：63.6%，若取 200 个测试元，则检查出的概率为 86.9%，若取 300 个，则检查出的概率可达 95.3%。

下图中给出了取的个数与正确率之间的关系(注意：这里只画了取 0 至 800 个时的图像，当取 800 个以上时，由于其正确率太接近 100%，其图像近似一条水平直线而没有再画出的意义)：



从图中还可以看出，只要抽样大约 70 个，就有 50%的正确率，抽样大约 230 个，就有 90%的正确率。

再看一个例子：如果总体 *S* 是{'**A'**,'**B'**,…,'**Z'**}的所有子集，而条件 *A* 是同时含有'**A'**~'**G'**的子集。按照上面，根据上面一例，可以想到，如果取一定量的子集测试，效果也会比较好，但好，对于总体 *S*，它存在一些特殊的子集——如空集∅、全集{'**A'**,'**B'**,…,'**Z'**}、只含一个元素的集合{'**A'**},{'**B'**}……这些都是我们认为比较特殊的集合，如果从这些特殊的集合中先取出几个测试，则可能在这些集合中就能找到满足条件 *A* 的，如此例中取全集显然就会满足条件。

下面看一个具体的实例：

例：Intuitionistic Logic(直觉主义逻辑)[③]

- 题目大意

给定一个有向无环图 *G*=(*V*,*E*)，在顶点 *V* 之间建立一些偏序关系：对于 *x*,*y*∈*V*，定义 *x*≤*y* 当且仅当 *x* 到 *y* 之间存在路径(可能长度为 0)。对于一个顶点集合 *S*，定义 **Max**(*S*)为 *S* 中所有最大元素的集合(即 **Max**(*S*)中的任意一个顶点都不可能通过一条或多条 *E* 中的边到达 *S* 中的其他顶点)，写成表达式的形式为：

Max(*S*)={*x*∈*S*|不存在 *y*∈*S*,*x*≠*y*,*x*≤*y*}

令*β*为所有满足 Max(*S*)=*S* 的集合所组成的集合，即：

β={*S*|**Max**(*S*)=*S*}

令 **0**=**Max**(*V*)，**1**=∅，显然 **0** 和 **1** 都属于*β*

定义*β*中元素的一些操作：

③ 题目来源：ACM ICPC 2002-2003, Northeastern European Region, Northern Subregion，原题请见附件。

$P=>Q=\{x\in Q|$不存在 $y\in P,x\le y\}$,

$P\wedge Q=$**Max**$(P\cup Q)$,

$P\vee Q=$**Max**$(\{x\in V|$存在 $y\in P,z\in Q,x\le y,x\le z\})$,

$\neg P=(P=>$**0**$)$,

$P\equiv Q=((P=>Q)\wedge(P=>Q))$

问题：对于一个含有变量的表达式，问是否无论变量如何在β中取值，其运算结果都是 **1**？

数据范围：其中$|V|\le 100$；$|E|\le 5000$；对于同一个图，要处理 k 个表示式，$k\le 20$；每个表达式长度不超过 254 个字符；$|\beta|\le 100$；$\sum_{1\le j\le k} H^{v_j}\le 10^6$ (v_j是在第 j 个表达式中用到的变量个数)

- 分析

本题的最后一个数据范围 $\sum_{1\le j\le k} H^{v_j}\le 10^6$ 其实暗示了我们，此题可以用枚举来做，即枚举所有变量的所有可能取值。显然，如果要求完全正确，变量取值的总方案数是不可能减少的，即可能达到 10^6，这时，就得设法减少计算的复杂度。显然，上面的基本运算都是 $O(|V|)$或 $O(|E|)$或更高的，而计算一个表达式可能要使用上百次基本运算，这样，尽管此题有 5 秒的时限，这样的枚举也是很难出解的。

由$|\beta|\le 100$，可以想到，将β中的每一个元素先用一个整数代替，并计算出对这些数进行上面的基本运算所得的值(当然也用整数表示)。对这些值列一个表，以后计算时就只要从表中查找结果了。这样，基本运算的复杂度就可以降到 $O(1)$，总的复杂度就可以降为 $O(\sum_{1\le j\le k} H^{v_j}*|B|)$。

此题还有一种解决方法，那就是抽样。将变量取*β*中一些比较特殊的值(如**0**，**1**，……)看作特殊样品，其它的看作一般样品。抽取一些特殊样品和一些一般样品，对它们进行测试，如果对于一个式子，所有的样品都满足条件，则说明式子满足条件，否则说式子满足条件。实践证明，只要抽取不到 1000 个样品即可得到比较高的正确率。

当然，对于这类多测试数据的题目，要基本保证一个测试点对，每组测试数据的正确概率必须更高一些。如：对于有 20 个组测试数据的测试点，如果保证每组数据的正确概率为 95%，则整个测试点的正确概率只有 0.95^{20}，还不到 36%；如果保证每组数据的正确概率为 99%，整个测试点的正确概率才 82%；如果每组数据正确概率达到 99.9%，则整个测试点的正确概率就会有 98%。

这里特别要说明的是：一般情况下，抽样的数目应该尽量多(在保证不超时的前提下)，这样才能保证正确概率较大。

抽样法的实际运用：

在测试质数时，抽样法是一个非常有用的工具。下面给出一种质数判定方法：

对于待判定的整数 *n*。设 $n-1=d*2^s$(*d* 是奇数)。对于给定的基底 *a*，若

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

或存在 $0 \le r < s$ 使

$$a^{d*2^r} \equiv -1 \pmod{n}$$

则称 n 为以 a 为底的强伪质数(Strong Pseudoprime)。利用二分法，可以在 $O(\mathbf{log}_2 n)$的时间内判定 n 是否为以 a 为底的强伪质数。

对于合数 c，以小于 c 的数为底，c 至多有 1/4 的机会为强伪质数(**Monier** 1980, **Rabin** 1980)。

根据这条，判断一个数 n 是否为质数，可以随机地抽取小于 n 的 k 个数为底。这样，正确的概率不小于 $1-4^{-k}$。显然，这已经非常优秀了，通常只要取几十组就可以保证基本正确。

如果不是随机抽样，而是抽样特殊情况——最小的几个质数，则：

④如果只用 2 一个数进行测试，最小的强伪质数(反例)是 2047(所以一个数显然不够)；

如果用 2 和 3 两个数进行测试，最小的强伪质数大于 10^6；

如果用 2,3,5 进行测试，最小的强伪质数大于 2×10^7；

如果用 2,3,5,7 进行测试，最小的强伪质数大于 3×10^9，已经比 32 位带符号整数的最大值(Pascal 中的 Maxlongint)还大了。

可见，通常只要抽取 2,3,5,7 这几个固定的数进行测试就能保证测试的正确性了。

我们知道，最朴素的质数测试方法是用 2 至 $\sqrt{n}$ 的数对 n 进行试除。这样，测试一个数的复杂度为 $O(n^{0.5})$，用上面的方法，每次测试只要 $O(\mathbf{log}_2 n)$的复杂度，由于只要测试很少的几次，所以，其总复杂度仍是 $O(\mathbf{log}_2 n)$的。可见，用抽样的方法对质数进行测试的算法明显优于朴素的质数测试法。

④ 具体的最小强伪质数列表请见附录。

小结

从上面的例子可以看出，由于抽样法只对总体中的一部分更行测试，所以它要测试的次数非常少，从而使得算法需要的时间比完全枚举少很多，算法的效率也会高很多。

1.3 部分忽略法

在信息学中，可能会遇到这样情况：一个题的分类非常复杂，且某些情况的处理非常复杂，但这些情况往往不是主要情况(即出现的概率很小或不处理这些情况对答案不会造成很大的影响)。有时，忽略这些复杂却影响不大的情况会使程序达到令人满意的结果。

例：Polygon[⑤]

- 题目大意

在此题中，所有的多边形均指凸多边形。

给定两个多边形 *A* 和 *B*，*A* 和 *B* 的 **Minkowski 和**是指包含所有形如(x_1+x_2, y_1+y_2) 的点的多边形，其中(x_1,y_1) 是 *A* 中的点，(x_2,y_2)是 *B* 中的点。

我们考虑 **Minkowski 和**的一种逆运算。给定一个多边形 *P*, 我们寻找两个多边形 *A* 和 *B*，满足：

- *P* 是 *A* 和 *B* 的 **Minkowski 和**；
- *A* 有 2 到 4 个不同的顶点(线段、三角形或四边形)；
- *A* 必须包含尽可能多的顶点。

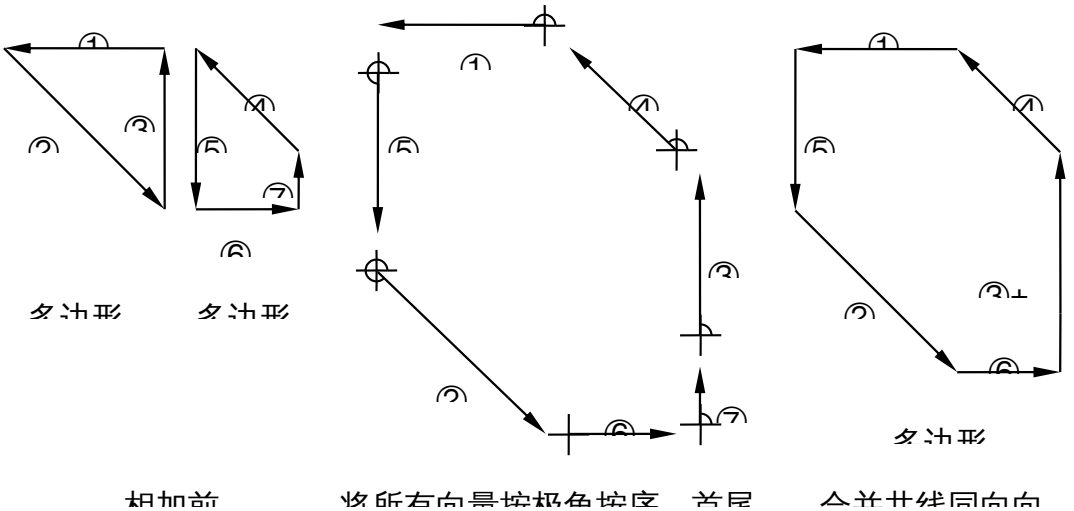
⑤ 题目来源：IOI2004，中文原题请见附录，英文原题在附件中《polygon-1.2》。

说明：此题附官方解答，其复杂度为 $O(N^2m^2)$，其中 N 为多边形 P 的顶点数，m 为 P 的边上的整点个数的最大值

➢ 分析

由于本题中，位置并不是很难处理，所以此题将不考虑多边形的位置问题，只考虑其形状和大小。

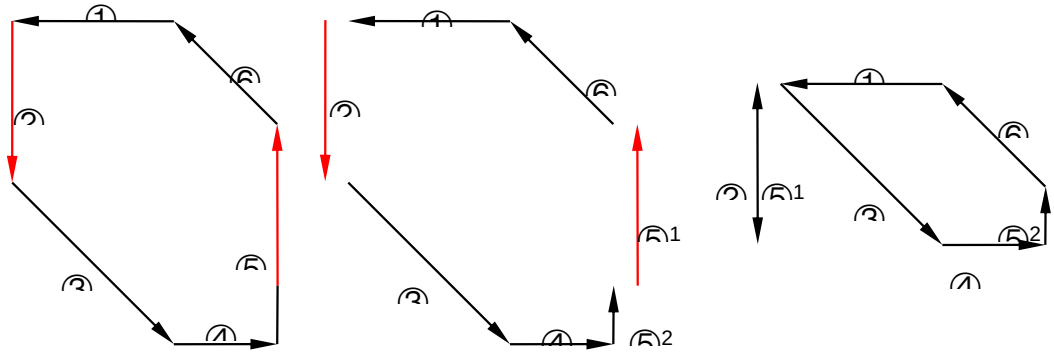
根据定义可以知道，对两个多边形 A、B 求 **Minkowski 和**，就是将它们的所有的边看作有顺序的向量，对所有向量按其级角排序，然后将所有的向量顺次连接，最后将共线同向的向量合并的过程。



由上面加法的过程不难想到，原来的多边形 A、B 的每条边在 P 中仍存在一条边与原边共线同向且模不小于原边，因此：

将一个多边形 P 拆成边数为 K 的多边形 A 与不定边数的多边形 B 相加的形式，就是要从 P 中找出 K 条边，从每边取出一段(或整条边)，使这 K 段正好能围成一个多边形，这个多边形就是 A，剩下的按照向量首尾相连可得到 B。

如：从上图的多边形 P 中取一个边数为 2 的多边形：



边②和边⑤的一部　　取出这两部　　两部分和剩下的部

从 *P* 中取出一个三角形、四边形，也是同样的方法。

这个算法看起来很简单，但实现起来并没有想象中那么轻松。

找一个二边形，只要找到一对共线向量即可。

找一个三角形，可以枚举三条边，然后判断，其复杂度是 $O(N^3m^3)$。在提交答案类题看来，已经很不错了。

找一个四边形，如果枚举四条边，然后判断，其复杂度是 $O(N^4m^4)$，因其中的 *m* 是未知的，所以不敢轻易使用。这时可以枚举三条边，另一条边通过对边的二分查找得到。由于第四条边可能是多边形 *P* 中某条边的一段，而 *P* 中一共可能有 *N***m* 段，因 *m* 未知，使得空间分配也不好处理；同时，由于存储时不仅要存储每段所对应的原来的边的序号，还要存储每段是占原来的几分之几……此时遇到的问题可能还远不只这些，这时，就很难保证程序不出错。

这种，不妨做一个大胆的假设：假设第四条边是 *P* 中的一条完整的边。这样，只要存储 *P* 条边即可，且这 *P* 条边都是原来多边形中的完整的边，所以编程中要考虑的问题就少多了，同时出错的可能性也小多了。

这样做，对于官方的数据，有 9 个测试点可以出解，所出的解全部是最优解。另一个测试点可以用手做。

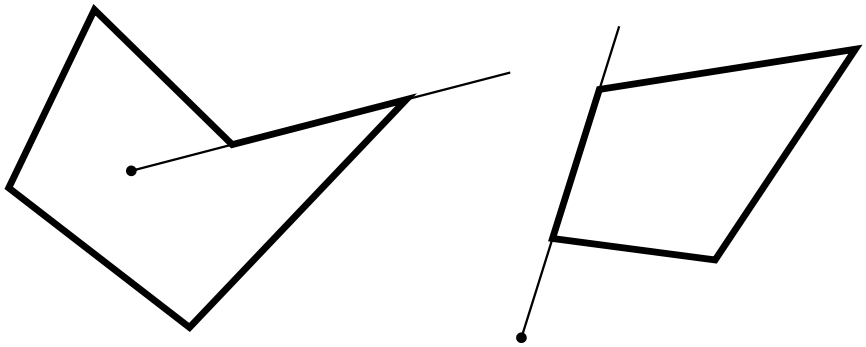
部分忽略法的实际应用

下面看一下判断一个点 *P* 是否在一个简单多边形内的算法：

假设已经判断了 *P* 在多边形边上这种特殊情况，剩下的只有点在多边形内和外的问题。

传统的判断算法是：以 *P* 为端点作一条射线，然后根据射线与多边形的交点个数是奇数个还是偶数个来判断其是否在多边形内。

但是，在计算交点个数的时候，有一种特殊情况：所作的射线经过多边形的某个顶点(有可能更特殊的与多边形的一条边重合)。如下图：



(*a*) (*b*)

在(*a*)中，处理的结果应为射线与多边形有奇数个交点，而(*b*)中，处理的结果应为射线与多边形有偶数个交点，如果不加特殊处理，则两种情况所得到的结果是相同的。

如果采用部分忽略法似乎会出现错误。但是，如果取的射线是一条很“一般”的射线：如在平面内随机取一个异于 *P* 的点 *Q*，且保证 *Q* 的坐标有若干位小数，从 *P* 作一条射线，使射线经过 *Q*，则，可以说，要出错的概率是非常小的。如果多取几个点，分别对这几个点进行判断，然后取这些结果中出现得最多的结果，则要出错就更不可能了。

小结

通过上面的例题可以看出，能过忽略部分复杂情况，能使算法的思考复杂性和编程复杂性降低。仅管它可能造成算法在一定程度上的错误，但这种错误通常是很小的。所以，有时采用部分忽略法仍是非常好的选择。

三、非完美算法的依据——RP 类问题与 Monte-Carlo 算法

前面所讲的一些方法，似乎都是靠的“运气”成分。但是，这些算法有它的依据：

如果一个问题存在一个随机算法，使得它有 50%以上的概率得到期望的结果，那么这个问题属于 RP 类问题。该算法称为 Monte-Carlo 算法。

如果一个问题是 RP 类问题，可以通过多次运行它的一个 Monte-Carlo 算法而得到“几乎每次都是正确”的算法。

由于 RP 类问题与 Monte-Carlo 算法不是本文研究的重点，这里不再多做介绍，有兴趣的读者可以参阅相关的资料。

四、非完美算法的共性

非完美算法有很多种，但是，它们并不是完全不同的，它们之间存在一个共同的性质，那就是不完全性。上面的非完美算法都是由于其不完全性而使得它们具有一些完全正确的算法所不能具备的优势，同时，其不完全性也导致了算法不是完全正确的。

五、非完美算法的优点与缺点

4.1 优点

从上面的分析和举例，相信读者对非完美算法的优点已经有所了解了，现整理如下：

①时空消耗低：这是非完美算法的一个最突出的优点，也是大多数情况下使用它的原因。

②编程复杂度低：因为非完美算法会可能绕过纷繁的处理或会忽略掉非常难处理的一些情况，其编程复杂度可以得到一定的降低。在比赛中，低的编程复杂度往往比低的算法时间复杂更容易得到令人满意的结果。

③思维复杂度低：同样也是因为非完美算法可能忽略掉非常难处理的一些情况。

④能减少编程错误：通过前面几点，这点也就显然了。也许，一个非常优秀的算法，因为它的一点点小错误，就导致其前功尽弃，而一个非完美的算法，因为它较容易实现，减少或避免了编程错误，反而能得到意想不到的好结果。

4.2 缺点

非正确性：这是不言而喻的，非完美算法，值得利用的就是它的非正确性。但非正确性使得算法不仅依赖算法的好坏、代码的好坏，还依赖于数据。如果数据较弱，非完美算法可能得到较高的分数，如果数据较强，其结果不一定很理想。

【总结】

本文主要介绍了非完美算法的几种重要方法。从上面的算法可以看到，并不是正确的算法就一定好过不完全正确的算法，因为非完美算法的不完全性，反而使非完美算法在很多方面比正确算法表现得更好。因此，合理的使用非完美算法能使我们得到令人满意结果。

【感谢】

衷心感谢向期中老师在我写这篇论文时对我的指导和帮助。

衷心感谢刘汝佳教练对我的指导和启发。

衷心感谢王俊、任恺同学对我的支持和帮助。

衷心感谢肖湘宁、周戈林同学在临近期末考试时还能为我看论文，并向我提很多宝贵的意见，特别感谢周戈林同学提供NOIP2003《传染病控制》的解题报告。

【参考文献】

《论随机化算法的原理与设计》——周咏基,1999

《The CRC Concise Encyclopedia of Mathematics》——Eric W. Weisstein,CRC Press

《计算几何——算法分析与设计》——周培德,清华大学出版社,广西科学技术出版社

《算法艺术与信息学竞赛》——刘汝佳、黄亮，清华大学出版社

《IOI'04: Solution of Polygon》——Ioannis Emiris and Elias Tsigaridas,2004

【附录】

最小强伪质数表

使用的最小质数个数	所使用的质数	最小强伪质数
1	2	2047
2	2,3	1373653
3	2,3,5	25326001
4	2,3,5,7	3215031751
5	2,3,5,7,11	2152302898747
6	2,3,5,7,11,13	3474749660383
7	2,3,5,7,11,13,17	341550071728321
8	2,3,5,7,11,13,17,19	341550071728321
9	2,3,5,7,11,13,17,19,23	≤41234316135705689041
10	2,3,5,7,11,13,17,19,23,29	≤155336056607314320554100 2401
11	2,3,5,7,11,13,17,19,23,29,31	≤56897193526942024370326972321

传染病控制原题(NOIP2003)

【问题背景】

近来，一种新的传染病肆虐全球。蓬莱国也发现了零星感染者，为防止该病在蓬莱国

大范围流行，该国政府决定不惜一切代价控制传染病的蔓延。不幸的是，由于人们尚未完

全认识这种传染病，难以准确判别病毒携带者，更没有研制出疫苗以保护易感人群。于是，

蓬莱国的疾病控制中心决定采取切断传播途径的方法控制疾病传播。经过

WHO（世界卫

生组织）以及全球各国科研部门的努力，这种新兴传染病的传播途径和控制方法已经研究
消楚，剩下的任务就是由你协助蓬莱国疾控中心制定一个有效的控制办法。

【问题描述】

研究表明，这种传染病的传播具有两种很特殊的性质；

第一是它的传播途径是树型的，一个人 X 只可能被某个特定的人 Y 感染，只要 Y 不
得病，或者是 XY 之间的传播途径被切断，则 X 就不会得病。

第二是，这种疾病的传播有周期性，在一个疾病传播周期之内，传染病将只会感染一
代患者，而不会再传播给下一代。

这些性质大大减轻了蓬莱国疾病防控的压力，并且他们已经得到了国内部分易感人群
的潜在传播途径图（一棵树）。但是，麻烦还没有结束。由于蓬莱国疾控中心人手不够，同
时也缺乏强大的技术，以致他们在一个疾病传播周期内，只能设法切断一条传播途径，而
没有被控制的传播途径就会引起更多的易感人群被感染（也就是与当前已经被感染的人有
传播途径相连，且连接途径没有被切断的人群）。当不可能有健康人被感染时，疾病就中止

传播。所以，蓬莱国疾控中心要制定出一个切断传播途径的顺序，以使尽量少的人被感染。

你的程序要针对给定的树，找出合适的切断顺序。

【输入格式】

输入格式的第一行是两个整数 n（1≤n≤300）和 p。接下来 p 行，每一行有两个整数 i

和 j，表示节点 i 和 j 间有边相连（意即，第 i 人和第 j 人之间有传播途径相连）。其中节点

1 是已经被感染的患者。

【输出格式】

只有一行，输出总共被感染的人数。

【输入样例】

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

【输出样例】

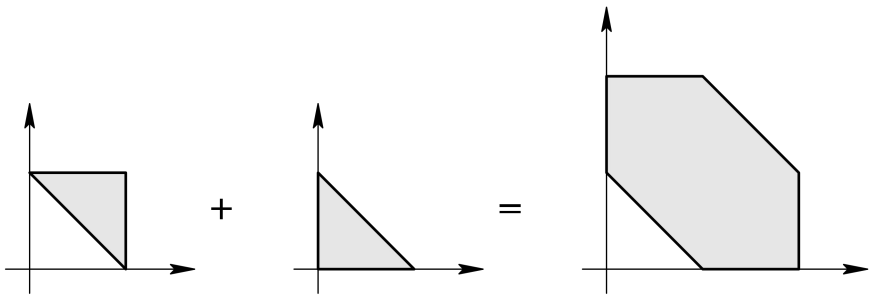
```
3
```

Polygon 中文原题

【问题描述】

凸多边形的定义如下：多边形内任意两点 *X* 和 *Y* 的连线上的所有点都在多边形内部。在本题中的所有多边形都是拥有至少两个顶点的凸多边形并且所有顶点互不相同，且具有整数坐标。多边形的任意三个顶点不共线。在下文中的“多边形” 一词都是指这样的多边形。

给定两个多边形 *A* 和 *B*， *A* 和 *B* 的 *Minkowski 和* 包含所有形如(x_1+x_2, y_1+y_2) 的点，其中 (x_1, y_1) 是 *A* 中的点，(x_2, y_2) 是 *B* 中的点。多边形的 *Minkowski* 和也是一个多边形。下图是一个例子：两个三角形和它们的 *Minkowski 和*。



我们考虑 *Minkowski 和* 的一种逆运算。给定一个多边形 *P*, 我们寻找两个多边形 *A* 和 *B* ，满足:

- *P* 是 *A* 和 *B* 的 *Minkowski 和*,
- *A* 有 2 到 4 个不同的顶点，例如，它是一条线段（2 个顶点），一个三角形（3 个顶点），或一个四边形（4 个顶点），
- *A* 必须包含尽可能多的顶点，例如：
    - *A* 如果可能的话，必须是一个四边形，
    - 如果 *A* 不可能是一个四边形而有可能是一个三角形，则它必须是一个三角形，
    - 否则它是一条线段。

很显然， *A* 和 *B* 都不能等于 *P* ，因为如果其中一个等于 *P*， 则另一个只能是一个点，而点不是一个合法的多边形。

给定多个输入文件，每个输入文件描述一个多边形 *P*。 对于每一个输入文件，你必须找出满足上述条件的 *A* 和 *B*， 并且创建一个文件来描述 *A* 和 *B* 。 对于所有给定的输入文件， A 和 B 一定存在。如果存在多组解，只需要输出其中的任意一组。不要提交源程序，只需要提交输出文件。

【输入格式】

共有十组输入文件从 polygon1.in 到 polygon10.in, 在 polygon 后的数字是输入文件序号。每个输入文件的构成如下：第一行包含一个整数 *N*： 多边形 *P* 的顶点个数。 接下来的 *N* 行以逆时针方向描述了 *N* 个顶点的坐标，每个顶点一行。第 *I*+1 (*I* = 1, 2, ..., *N*) 行包含两个整数 X_I 和 Y_I， 以一个空格隔开，表示第 *I* 个顶点。所有坐标为非负整数。

【输出格式】

针对给定的输入文件，你需要给出相应的 10 个输出文件用以描述相应的 *A* 和 *B*。输出文件的第一行应包含：

#FILE polygon I

这里整数 I 表示相应的输入文件序号。

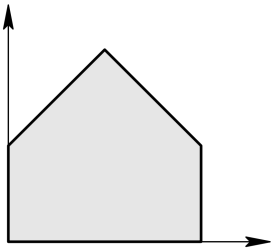
输出文件的格式与输入文件类似。第二行包含一个整数 N_A：A 中的顶点数 ($2 \le N_A \le 4$)。 接下来的 N_A 行以逆时针方向描述 *A* 的 N_A 个顶点，每个顶点一行。第 *I*+2 (*I* = 1, 2, ..., N_A) 行包含两个整数 *X* 和 *Y*, 以一个空格隔开：表示 *A* 的第 *I* 个顶点。

第 N_A +3 行包含一个整数 N_B： 表示 B 中的顶点个数 ($2 \le N_B$)。 接下来的 N_B 行以逆时针方向描述 B 的 N_B 个顶点，每个顶点一行。第 N_A +J+3 (J = 1, 2, ..., N_B) 行包含两个整数 X 和 Y， 以一个空格隔开：表示 B 的第 J 个顶点。

【输入输出样例】

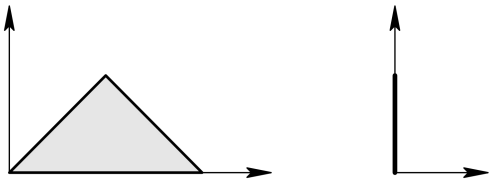
polygon0.in

```
5
0 1
0 0
2 0
2 1
1 2
```

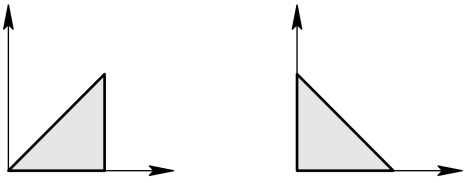


对于上面的输入，下面两种输出都是正确的，因为在每种输出中 A 都是一个三角形，而 A 不可能是一个四边形。

```
#FILE polygon 0
3
0 0
2 0
1 1
2
0 1
0 0
```



```
#FILE polygon 0
3
0 0
1 0
1 1
3
0 1
0 0
1 0
```



传染病控制解题报告

题意简述：

给定一棵树，其根节点 1 是已经被感染的患者。在一个疾病传播周期内，只能切断树的一条边，然后所有被感染的节点的孩子节点也会被感染。要求一个最佳的切断边的方案，使被感染的节点数最小。

解法分析：

将被感染节点中深度最大的节点集合称之为最新传染源，设为 S 。当经过 X 个疾病传播周期，传染病传播的最新传染源的深度是一定的，即为 X （根节点 1 的深度为 0）。因此可以得到以下动态规划算法：设 $f(X,S)$ 表示经过 X 个疾病传播周期且最新传染源集合为 S 被感染节点的最小值，那么由 $f(X,S)$ 可以推得 $f(X+1,son(S)-v)$ ，其中 $son(S)$ 表示 S 集合中所有的孩子节点， v 是被切断边的一个深度较大的节点。似乎问题已经得到了很好的解决，但是考察一下算法的空间复杂度，达到了 $O(N\times 2^N)$ ，这是完全不可能实现的。

鉴于空间上的问题，我们只好放弃动态规划。尝试先从简单的角度思考问题：搜索的效果会怎么样呢？分析一下问题的状态空间：题目给定的是一棵树。把题目中的树称为原树，搜索树的层数就是原树的深度，而搜索树的分叉数多少与原树的宽度有关。综合起来看，原树深度较大，宽度会较小；而原树宽度较大，深度又会较小，所以搜索树的规模没有想象中那么大。

证明了搜索是“可能”出解的，剩下的就是怎么搜的问题：在 X 个传播周期内疾病传播达到的深度是一定的，为 X 。如果在深度小于 X 的位置切断边，由

于已经通过它进行了传播，切断是没有意义的；如果在深度大于 X 的位置切断边，效果肯定不如把它的祖先切断。因此最优的切断方法必须是切断与 S 相连的边中的一条。这样搜索已经可以通过大部分数据，但是对于极限大数据还是无法出解，需要优化。

优化一：初看这道题目的时候会想到贪心。也就是每次选节点数目最大的子树删除，虽然很容易找到反例，但是我们仍然可以认为选节点数目最大的子树删除更“可能”得到最优解，因此可以按照子树节点数目从大到小删除。

优化二：最优性剪枝。设当前已经搜到了第 X 层，有 T 个节点已被感染，那么如果有

$T+\left|son(S)\right|-1\geq best$ 即可剪枝。这是因为在一个疾病传播周期内，最多只能切断树的一条边，所以在下一周期至少还有 $T+\left|son(S)\right|-1$ 个新感染的节点。

优化三：搜索最后一两层时，由于子树的结构都相同，随便选择一个删除就可以退出了。

加入这三个优化后，程序的效率大大提高。最大官方数据在 0.2s 内解出。

参考程序：

```
{$R-,Q-,S-,I-}
{$M 65521,0,655360}
{Author:Zhou Gelin}
{Date:2005.1.11}
{QQ:379688236}
{MSN:zhougelin@hotmail.com}
{Test Report}
{P4 1.7G}
{Windows Xp SP1}
{Turbo Pascal 7.0}
{
EPIDEMIC.in1 = 10.0 (0.06s)
EPIDEMIC.in2 = 10.0 (0.08s)
EPIDEMIC.in3 = 10.0 (0.05s)
```

```
EPIDEMIC.in4 = 10.0 (0.03s)
EPIDEMIC.in5 = 10.0 (0.03s)
EPIDEMIC.in6 = 10.0 (0.03s)
EPIDEMIC.in7 = 10.0 (0.06s)
EPIDEMIC.in8 = 10.0 (0.03s)
EPIDEMIC.in9 = 10.0 (0.20s)
EPIDEMIC.in0 = 10.0 (0.11s)
}
program epidemic;
const inputfilename='epidemic.in';
      outputfilename='epidemic.out';
      maxn=300;
type pnode=^node;
     node=record
       data:integer;
       next:pnode;
         end;
     list=array[1..maxn]of integer;
var n,p,maxdep,ans:integer;
    g:array[1..maxn]of pnode;
    father:array[1..maxn]of integer;
    dep:array[1..maxn]of integer;
    sum:array[1..maxn]of integer;
    deg:array[1..maxn]of integer;
    son:array[1..maxn]of ^list;
    a:array[1..maxn]of boolean;
procedure insert(a,b:integer);
var q:pnode;
begin
      new(q);
      q^.data:=b;
      q^.next:=g[a];
      g[a]:=q;
end;
procedure read_data;
var i,a,b:integer;
begin
      fillchar(deg,sizeof(deg),0);
      assign(input,inputfilename);
      reset(input);
      readln(n,p);
      for i:=1 to p do
      begin
        readln(a,b);
```

```
        insert(a,b);
        insert(b,a);
        deg[a]:=deg[a]+1;
        deg[b]:=deg[b]+1;
      end;
      close(input);
end;
procedure dfs(v:integer);
var q:pnode;
   i,j,k:integer;
begin
      if father[v]>0 then
      begin
        deg[v]:=deg[v]-1;
        dep[v]:=dep[father[v]]+1;
        if dep[v]>maxdep then maxdep:=dep[v];
      end
      else dep[v]:=1;
      getmem(son[v],2*deg[v]);
      sum[v]:=1;q:=g[v];i:=0;
      while q<>nil do
      begin
        if q^.data<>father[v] then
        begin
          i:=i+1;
          son[v]^[i]:=q^.data;
          father[q^.data]:=v;
          dfs(q^.data);
          sum[v]:=sum[v]+sum[q^.data];
        end;
        q:=q^.next;
      end;
      for i:=1 to deg[v]-1 do
      begin
         k:=i;
         for j:=i+1 to deg[v] do
            if sum[son[v]^[j]]>sum[son[v]^[k]] then k:=j;
        j:=son[v]^[i];
        son[v]^[i]:=son[v]^[k];
        son[v]^[k]:=j;
      end;
end;
procedure search(deep,sick:integer);
var i,j,tmp:integer;
```

```
begin
    if deep=maxdep then
    begin
      if ans>sick then ans:=sick;
      exit;
    end;
    tmp:=0;
    for i:=1 to n do
      if a[i] and (dep[i]=deep) then
        tmp:=tmp+deg[i];
    if tmp=0 then begin search(maxdep,sick); exit; end;
    if sick+tmp-1>=ans then exit;
    for i:=1 to n do
      if a[i] and (dep[i]=deep) then
        for j:=1 to deg[i] do
          a[son[i]^[j]]:=true;
    for i:=1 to n do
      if a[i] and (dep[i]=deep) then
        for j:=1 to deg[i] do
        begin
          a[son[i]^[j]]:=false;
          search(deep+1,sick+tmp-1);
          a[son[i]^[j]]:=true;
          if deep=maxdep-1 then break;
        end;
    for i:=1 to n do
      if a[i] and (dep[i]=deep) then
        for j:=1 to deg[i] do
          a[son[i]^[j]]:=false;
end;
procedure answer;
begin
    assign(output,outputfilename);
    rewrite(output);
    writeln(ans);
    close(output);
end;
var time:longint;
begin
    time:=meml[64:108];
    read_data;
    dfs(1);
    ans:=1000;
    fillchar(a,sizeof(a),0);
```

```
    a[1]:=true;
    search(1,1);
    time:=meml[64:108]-time;
    writeln(time);
    answer;
end.
```

数据关系的简化

长沙雅礼中学 何林

I. 摘要

数据之间的关系有着和数据本身同等的重要性。最常见的数据关系有线性关系、树关系和图关系。信息学竞赛的本质就是对数据的充分挖掘。然后有时候挖掘太过充分，反而会把问题复杂化。本来将讨论的就是如何通过适当的简化数据关系，实现数据的合理挖掘。

II. 关键字

树，图，序列，数据关系

III. 引言

我们经常面对大量的数据，但是他们之间不是杂乱无章的。一些用道路连接的城市表现出来的数据关系是图；一些行政部门的上司下属关系表现出来的是树关系；学校的成绩排名表现出来的数据关系是线性关系。

图、树和线性关系是我们在生活中、也是在信息学竞赛中遇到的三种最常见的关系。虽然信息学竞赛强调对输入信息的充分挖掘和应用，但有时候关系过于复杂反而让人眼花缭乱。本文就是要介绍一种重要的思想：简化数据结构。具体的说可以把图简化成树、把树简化成线性结构。这看起来不可思议，因为在简化的过程中必然会丢失一些信息，但是通过本文接下来的分析和举证，你又会发现这是切实可行的一种思想。

IV. 简化图关系

先来看一个题目：

坐船问题

雅礼中学有 n 个学生去公园划船。一条船最多可以坐两个人。如果某两个学生同姓或者同名就可以坐在一条船上。

学校希望每个同学都坐上船，但是小船的租用费用很高，学校想要租用最少的船。请问：学校至少要租多少船？

我们可以把每个学生看作一个顶点。如果两个学生同姓，就在两者之间连一条红边；如果他们同名，就在两者之间连一条蓝边。

这样我们就把问题的全部信息都**毫无遗漏**的包含在这个图里面了。剩下的问题就是求一个最小边覆盖，实际上也就是求最大匹配。

这个图并不一定是二分图，所以求最大匹配涉及到带花树，十分复杂。有没有更好的方法呢？

首先假设这个图是连通的。

下面我们用**一棵树来表示这个图**。这是一颗二叉树。每个节点的左儿子（如果存在）和它同姓，右儿子（如果存在）和它同名。

构树算法如下：

假设树中已经含有 k 个点了。如果 k=n，那么构造算法结束。否则因为整个图是连通的，剩下的点中至少有一个和当前的树连通。不妨设剩下的某个点 P 和树中的一个点 T 之间有边。

第一种情况是 P 和 T 同姓。沿着 T 的左儿子不断的“往左”走，直到到达某一个点 X 无法继续前进（也就是 X 没有左儿子）。因为 X 和 T 同姓，所以它也必然和 P 同姓；同时 X 没有左儿子，令 P 为 X 的左儿子即可。这样树的规模就增大了 1。

第二种情况是 P 和 T 同名。和第一种情况类似。

于是我们总是可以把树的规模增大 1，直到把所有的点都包含到树中。

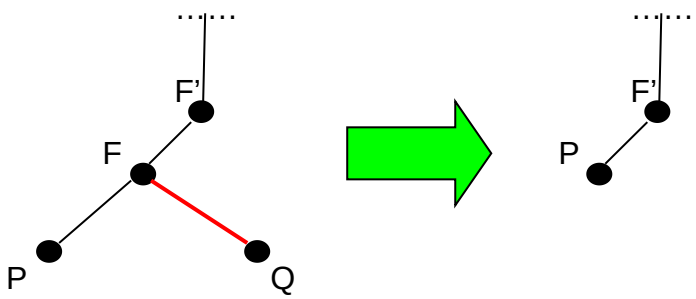
构造出这样一棵树有什么好处呢？下面考虑任意一个叶子节点 P，设它的父亲是 F。

如果 P 是 F 的唯一孩子，那么令 P 和 F 坐一条船。树的规模就减小了 2。否则 F 有两个孩子，设另一个孩子是 Q。为了方便讨论，不妨设 P 是 F 的左儿子，Q 是 F 的右儿子。

第一种情况：F 是根节点。

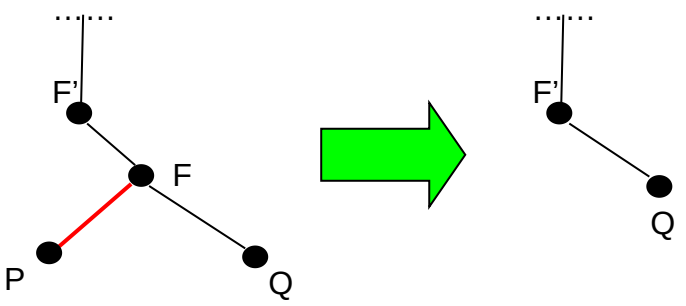
这时候总共有三个学生还没坐船。至少要两条船。令 F 和 P 坐一条船，Q 坐一条船即可。

第二种情况：F 是它父亲的左儿子。设 F 的父亲是 F’。



令 F 和 Q 坐一条船。然后把 F 和 Q 从树中删除。此时 P 的左儿子 P 落了单。因为 F 和 F’同姓，F 又和 P 同姓，所以 F’和 P 同姓。令 P 为 F’的左儿子即可。这就保证了树的连通性。

第三种情况：F 是它父亲的右儿子。



和第二种情况类似。见上图。

通过以上算法，可以不断的把树的规模减小。如果树有 n 个节点，那么最后的结果就是需要$\left\lceil \frac{n}{2} \right\rceil$条船。毫无疑问这是最优解。

如果图不连通，那么把每个连通图按照上述算法分别处理，最后把船数相加即可。

整个算法的复杂度是 O(n)，不仅效率上大大优于匹配算法，而且两者的编程复杂度更是不能同日而语。

这个题目的数据是学生，数据关系是同姓或者同名。毫无疑问，用图来表示学生之间的关系是很自然的想法，而且图也的确可以毫无遗漏的把所有信息包含进去。但是正因为图的包罗万象，使得我们束手无策。虽然存在匹配算法，但是它不仅思维、编程复杂度高，而且效率也不尽人意。

在思维的路上转一个弯，我们用一棵二叉树来表现数据关系。尽管有些人之间的信息被忽略了，比如根节点和某个叶子节点可能同姓或者同名、而在树中他们两者之间也许没有直接关联的边；我们反而能够更加有效率的应用这些简化后的数据关系解题。

可见数据和数据关系的利用也不是越充分越好，有时候合理的运用更有效。

下面我们再来看一个例子。

仙人掌图的判定

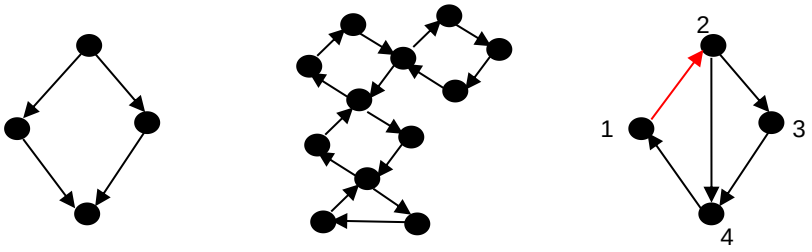
如果一个有向图：

1. *它是一个强连通图。*
2. *它的任意一条边都属于且仅属于一个环。*

这个图就称为仙人掌图。

输入一个 n 个节点，m 条边的图（1<=n,m<=10^5），请判断它是不是仙人掌图。

为了对仙人掌图有一个感性认识，我们先看下面三个图。



a)非强连通　　b)仙人掌图　　c)红色的弧同时属于两个圈

其中 a)和 c)都不是仙人掌图，因为 a)不是强连通的、c)中的红弧同时属于两个圈 1→2→3→4→1 和 1→2→4→1。

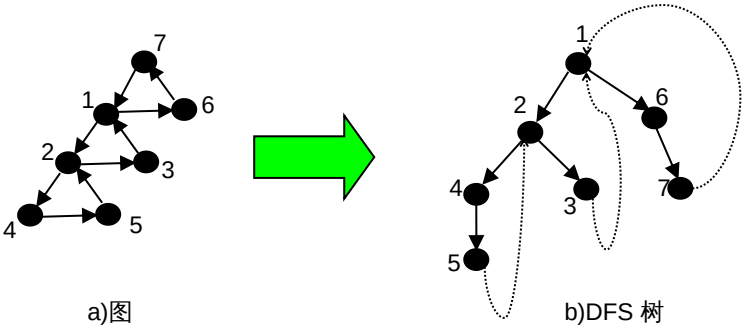
b)就是一个仙人掌。直观的说，仙人掌图就是一个一个的圈直接“粘”在一起的图，圈之间没有公共边。

于是我们很容易得到这样的一个算法：每次找一个圈，如果圈中不相邻的点之间有边，那么该图就不是仙人掌；否则把圈缩成点，然后把圈、点、边的关系进行适当的调整，继续缩圈。

权且不说具体该如何调整圈、点、边的关系，光是缩点这一项就要大费周章。该算法的思维和编程复杂度将非常高。

本文要介绍的另一个种算法是 DFS。对该图进行 DFS 遍历，建立 DFS 树。

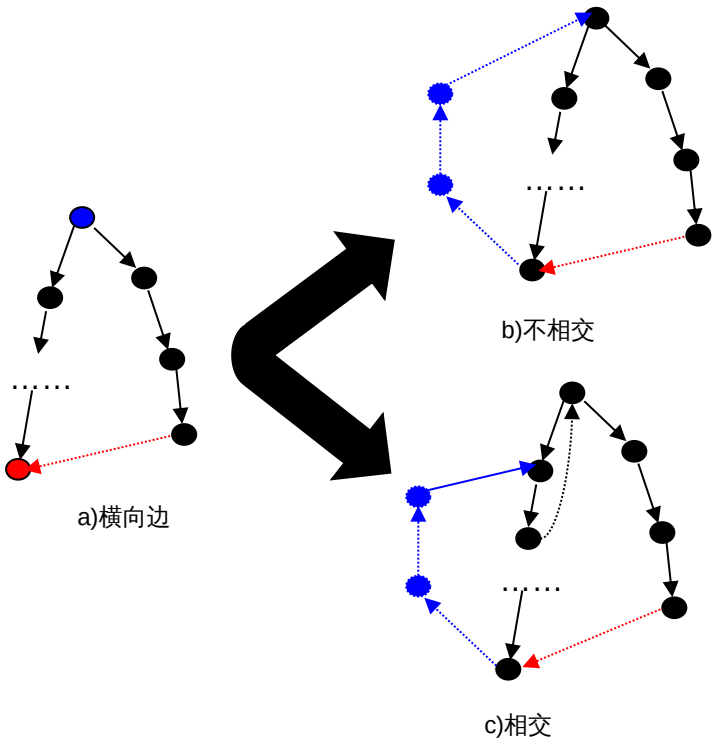
譬如下图：



a)图　　b)DFS 树

如果一个图是仙人掌的话，它的 DFS 生成树有什么特点呢？分析 DFS 树的一般方法是从逆向边和横向边入手。

首先考虑它的横向边。



a)横向边

b)不相交

c)相交

上面 a)图中的红边是横向边。蓝点是红点的祖先，称从蓝点遍历到红点的这条路为红点的“祖先路径”，记作 P。因为仙人掌图强连通，所以必须存在一条从红点到蓝点的路。

如果这条路不和 P 相交，则如 b)图所示；如果这条路和 P 相交，则如 c)图所示。不论是 b)图还是 c)图，蓝色的点和边都同时隶属两个圈。也就是说，只要存在横向边，这棵 DFS 树的原图就肯定不是仙人掌图。

所以：

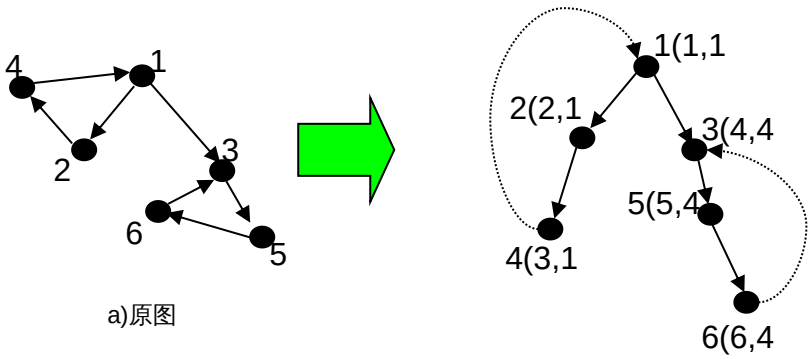
性质 1 仙人掌图的 DFS 树没有横向边。

下面我们进一步考虑逆向边。

对于每个节点 v，定义两个函数：

1. DFS(v)表示 v 在 DFS 树中是第几个被遍历到的点。
2. Low(v)表示通过从 v 以及 v 的所有后代直接指出去的边，可以访问到的 DFS 值最小的点的 DFS 值。

   通过下图读者可以对 DFS 和 Low 的定义获得一个更感性的认识。



a)原图

b)DFS 树

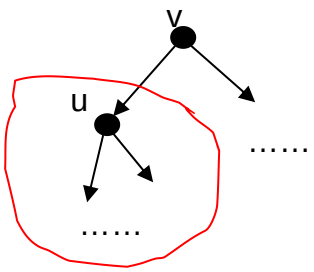
括号中第一个数是 DFS 值, 第二个是 LOW 值

DFS 函数的求解可以通过设立一个计数器，在深度优先遍历的时候每碰到一个新的点就累加 1 来实现。

Low 函数的计算如下：

Low(v)=min{DFS(v), Low(u)} (其中 u 是 v 的儿子)

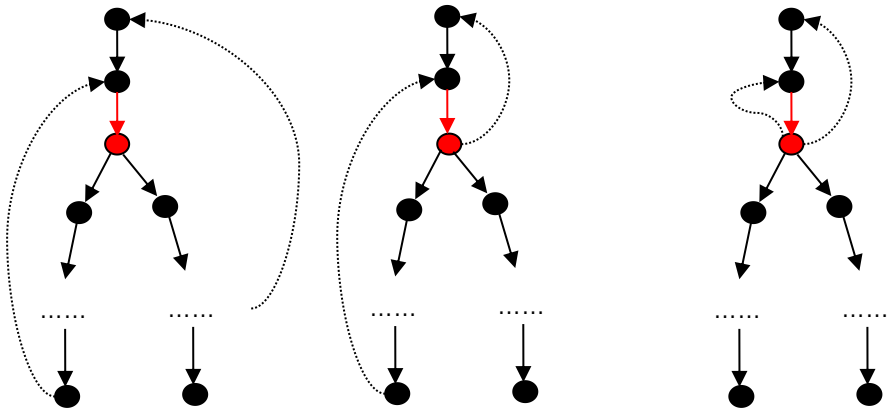
下面考虑某一个点 v。如果它存在一个儿子 u 满足 Low(u)>DFS(v)，那么这个 DFS 树的原图肯定不是仙人掌。



如上图所示，因为 Low(u)>DFS(v)，所以 u 以及 u 的后代都被限制在红色的线圈范围之内，没有指出去的逆向边。这样<v, u>就成了桥。我们知道在一个强连通图中是不可能有桥的，所以：

性质 2 Low(u)<=DFS(v) (u 是 v 的儿子)

然后看下面三个图。



a)图　　b)图　　c)图

上图中的红点都是 v。

在 a)图中，v 有两个儿子的 Low 值小于 DFS(v)，这时红边就同时属于两个圈了。

在 b)图中，v 有一个儿子的 Low 值小于 DFS(v)，同时 v 自己也有一条逆向边。这时红边也同时属于两个圈。

在 c)图中，v 有两条逆向边。这时红边同样属于两个圈。

以上三种情况下，原图都不是仙人掌。归纳起来就是：

性质 3 设某个点 v 有 a(v)个儿子的 Low 值小于 DFS(v)，同时 v 自己有 b(v)条逆向边。那么 a(v)+b(v)<2。

至此我们已经获得了仙人掌图 DFS 生成树的三条性质：

性质 1 仙人掌图的 DFS 树没有横向边。

性质 2 Low(u)<=DFS(v) (u 是 v 的儿子)

性质 3 设某个点 v 有 a(v)个儿子的 Low 值小于 DFS(v)，同时 v 自己有 b(v)条逆向边。那么 a(v)+b(v)<2。

与以上任意性质相悖的图肯定不是仙人掌；反之如果一个图的 DFS 生成树满足以上所有性质，那么它也肯定是仙人掌图（这个可以通过对生成树边和逆向边的分析证明，在此略）。

至此我们就得到了一个仙人掌图判定的 DFS 算法。时间复杂度是 O(n+e)。与“缩圈”算法比较起来，DFS 算法最吸引人的地方还在于其编程复杂度小。

我们把原图用 DFS 树的形式重新描述，这是解题过程中最本质的突破。利用仙人掌图的特殊性，我们可以把 DFS 树中的逆向边、横向边各个击破。同时因为树有祖先和后代之分，具有明显的层次感，为我们设计算法，比如 Low 函

数，提供了灵感；反观原图，既没有明显的拓扑关系，仙人掌图的特殊性也无法有效的用图的性质来体现。

严格的说，仙人掌图的 DFS 判定算法并没有“简化”数据关系，它只是把数据关系用一种更有次序、“原始”的方式表现了出来。形式的简化将数据之间的关系更加清晰的浮上了水面。或许通过对图的分析也能最终解决问题，但是其过程无疑要复杂得多。

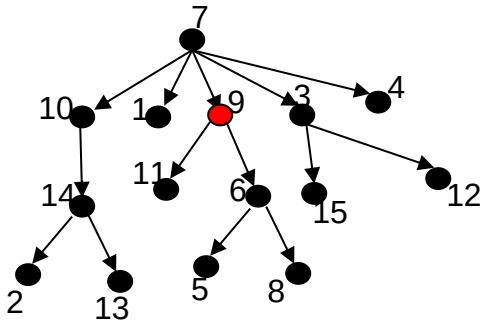
V. 简化树关系

先看一道题目：

树的统计

一棵含有 n 个节点的树，所有的节点依次编号为 1, 2, 3, …, n。对于编号为 v 的节点，定义 t(v)为 v 的后代中所有编号小于 v 的节点个数。输入这棵树，请输出 t(1), t(2), t(3), …, t(n)。

我们当然可以毫不费力的得出一个 $O(n^2)$的算法，但时间复杂度太高。通过对树本身的分析，我们也可以设计出一个 O(nlogn)级别的算法，但涉及到树的拆分等很复杂的操作(对此算法有兴趣的同学可以联系我)。下面我们来看一种别出心裁的算法（注：此题的命题者和该算法提供者是雅礼中学的雷涛同学）。



我们深度优先遍历该树，然后按照访问到的先后顺序把节点依次写下来：

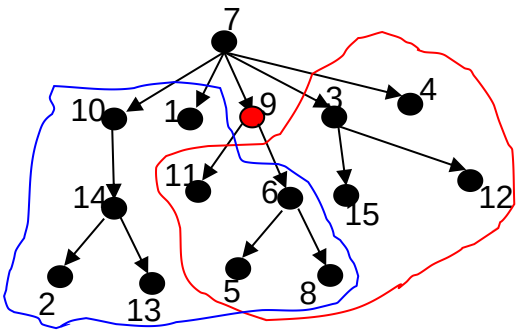
7 10 14 2 13 1 **9** 11 6 5 8 3 15 12 4（该序列称为“DFS 序列”）

我们发现数字 9 后面的部分正是下图中用红线框出来的区域。

然后我们把每个节点的儿子先后顺序倒过来，重新遍历，得到的序列如下：

7 4 3 12 15 **9** 6 8 5 11 1 10 14 13 2（该序列称为“逆 DFS 序列”）

这个序列中数字 9 后的部分正是下图中用蓝线框出来的区域。



很容易发现红蓝两个线圈有一个重叠区域，而这个区域正是“9 的所有后代”。这就提示我们用容斥原理。

图中除了红线和蓝线囊括的区域外，还有一个未被触及的盲区：那就是 9 本身和它的直系祖先。

定义 f(v, S)表示在 S 所描述的集合或者区域中，有多少个节点是小于 v 的。

我们可以得到如下公式：

f(9,红)+f(9,蓝)+f(9,9 的直系祖先)-f(9,整棵树)=f(9,9 的后代)

推广一下就是：

f(v,DFS 序列中 v 之后的部分)+f(v,逆 DFS 序列中 v 之后的部分)+f(v,v 的直系祖先)-f(v,整棵树)=f(v,v 的后代)

很容易观察到 f(v,整棵树)=v-1，而 f(v,v 的后代)就是 t(v)，所以：

t(v)=f(v,DFS 序列中 v 之后的部分)+f(v,逆 DFS 序列中 v 之后的部分)+f(v,v 的直系祖先)-(v-1)

通过一次 DFS 遍历联合线段树就能以 O(nlogn)的时间复杂度求出所有节点的直系祖先中有多少个比它本身小。对于 DFS 序列和逆 DFS 序列，可以采用线段树或者树状数组求出序列每个元素后面有多少个比它本身小；时间复杂度 O(nlogn)。

综上，O(nlogn)的时间复杂度即可解决此题。

这个算法最巧妙的地方是把树对应到了两个序列：DFS 序列和逆 DFS 序列。本来在树中，某个节点和它的后代之间存在具有层次性的拓扑关系；通过变换成序列后，这个关系简化成了一个线性序列中的先后关系。正是这个“简单”的先后关系使我们能够用线段树或者树状数组来解决问题。

下面我们来看一个更加经典的问题：

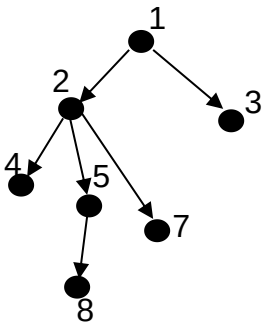
最近公共祖先

已知一棵 n 个节点的树，你会不断收到这类询问：“p 和 q 的最近公共祖先是什么？”。询问总数可能很大。请你设计算法回答所有询问。

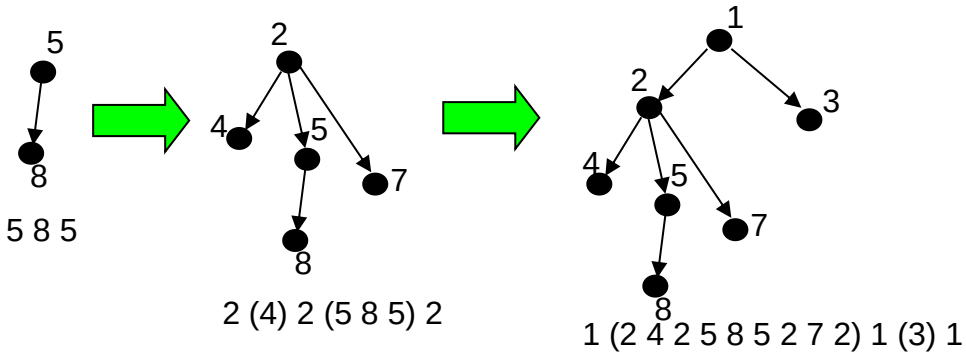
我们考虑把树用一个序列表示出来。

这是一个递归的过程。**如果整棵树就是一个节点，对应的序列就是(Root)；否则把根节点的所有子树分别表示成序列 $L_1, L_2, \ldots, L_t$，原树的序列就是$(Root, L_1, Root, L_2, Root, L_3, \ldots, L_{t-1}, Root, L_t, Root)$。**

譬如下图：



我们可以这样把它变成一个序列：



我们将节点的深度写在旁边：

1(1) 2(2) 4(3) 2(2) 5(3) **8(4) 5(3) 2(2) 7(3)** 2(2) 1(1) 3(2) 1(1)

如果要求两个节点，比如 8 和 7 的最近公共祖先。考察序列中 8 和 7 之间的部分，如上红色部分标记的，其中深度最小的点是 2。所以 8 和 7 的最近公共祖先就是 2。

又比如求 5 和 3 的最近公共祖先：

1(1) 2(2) 4(3) 2(2) **5(3)** 8(4) **5(3)** 2(2) 7(3) 2(2) 1(1) 3(2) 1(1)

注意到序列中有两个 5，应该选哪一个呢？实际上任意选一个都没有影响，比如选如下蓝色部分：

1(1) 2(2) 4(3) 2(2) **5(3) 8(4) 5(3) 2(2) 7(3) 2(2) 1(1) 3(2)** 1(1)

其中深度最小的点是 1，所以 5 和 3 最近公共祖先就是 1。

一般的说：

在序列中任意找一个 p，任意找一个 q。在序列中 p 和 q 之间的部分，找一个深度最小的点，就是 p 和 q 的最近公共祖先了。

下面证明算法的正确性。回头看看我们的序列构造方法：

这是一个递归的过程。如果整棵树就是一个节点，对应的序列就是(Root)；否则把根节点的所有子树分别表示成序列 $L_1, L_2, \ldots, L_t$，原树的序列就是(Root, L_1, Root, L_2, Root, $L_3, \ldots, L_{t-1}$, Root, L_t, Root)。

正确性的证明也是一个递归的过程。假设算法对于 $L_1, L_2, \ldots, L_t$ 这些序列都是正确的。如果两个点 p 和 q 属于同一个 L_i，那么该算法正确。否则 p 属于 L_i，q 属于 $L_j(i \neq j)$，那么 p 和 q 的最近公共祖先肯定是 Root；因为构造序列的时候在任意两个相邻的 L 序列之间都插入了一个 Root，而 Root 的深度又是最小的，因此在这种情况下算法也是正确的。

至此问题基本解决。具体如何求序列中某一连续段中的最小值，可以用线段树(回答一次询问的时间复杂度 O(logn))，也可以用经典的 01RMQ 算法(回答一次询问的时间复杂度 O(1))。

求最近公共祖先是一个很重要的问题，也是一个树转化到序列的经典例子。

VI. 总结

本文的主题是数据关系的简化。两种重要的简化是图→树以及树→序列。本文通过列举四个例子，希望让读者获得对“数据关系简化”这一思想的感性认识。

从严格的意义上来说，所谓简化只是一种形式的变化。用更简单、原始的数据结构表达本来用复杂数据结构表现的关系。这种“简单化”带来的往往是更加清晰的数据关系，为窥见问题根本矛盾创造条件。

具体的说，利用 DFS 生成树和 DFS 序列来解决与图、树有关的问题，是一种很实用有效的思想。

除了 DFS 当然还有其他的数据关系简化的应用。比如图论中的“弦图判定”定义了“完美消除序列”，就是一个很好的例子。

参考资料

1. 湖南省赛试题
2. 2000 年信息学国家集训队队员肖州的论文
3. <Introduction to Algorithms>
4. 湖南长沙雅礼中学雷涛同学的原创试题