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。

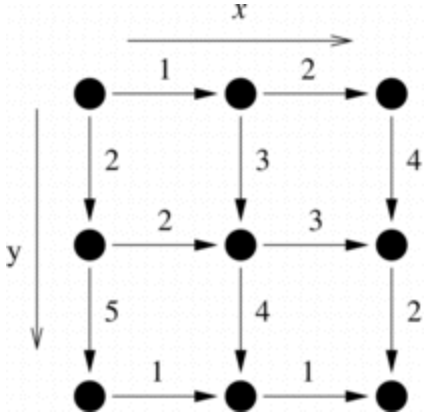
【正文】

1、 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。



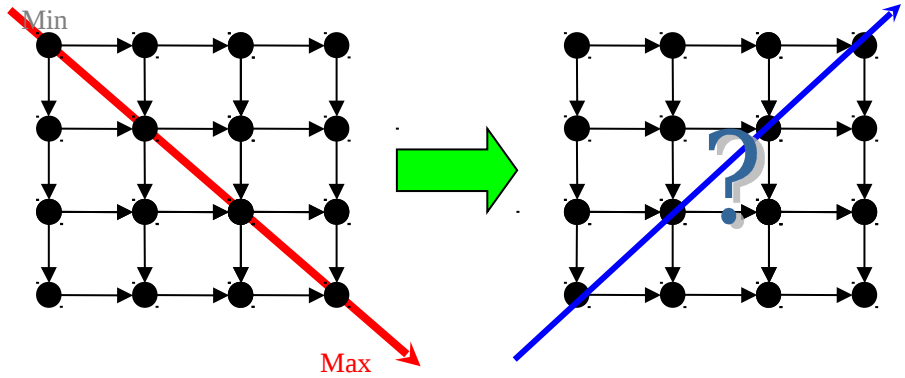
每次可以提问某个点(x,y)到同行最右边的点离他的距离，及同列最下边的点离他的距离。现在要你找到一个点，使得左上点到他的距离为给定的整数 L（1<=L<=2000000000）。最多允许提问 6667 次（你最开始仅仅已知 N、L）。

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！

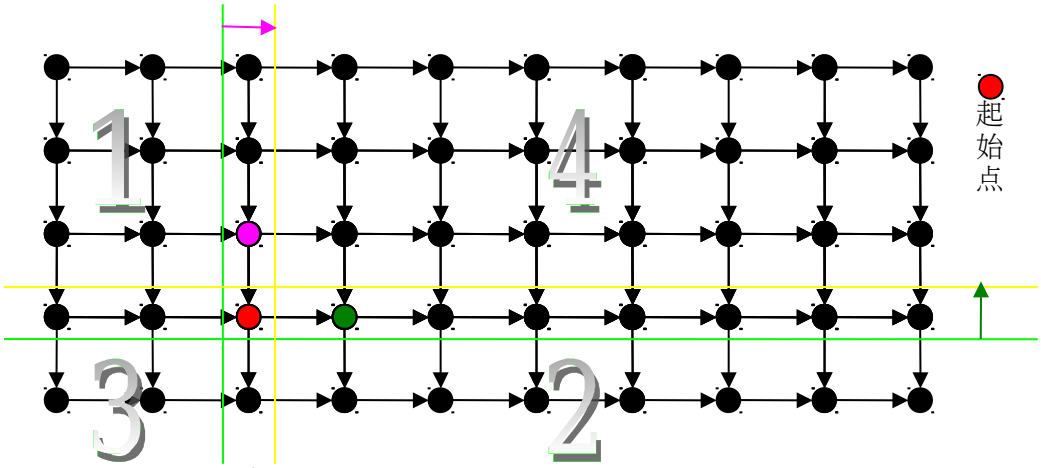
不妨从题目的要求入手：题目仅仅要我们找到一个点，他到左上角点的距离为 L。也就是说我们并不关心每条边的权值，而只关心每个点的距离。我们不妨把每个点到左上角点的距离叫做每个点的权值。

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。

2、 通过应用“序”简化问题的例子

先看一道题目：

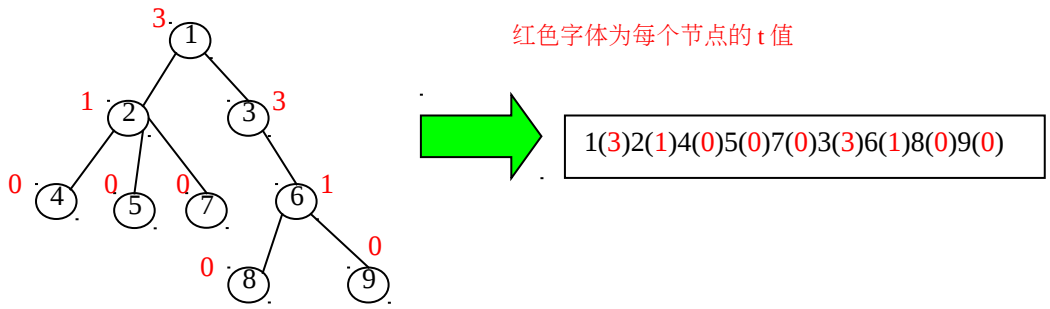
树的构造：

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。

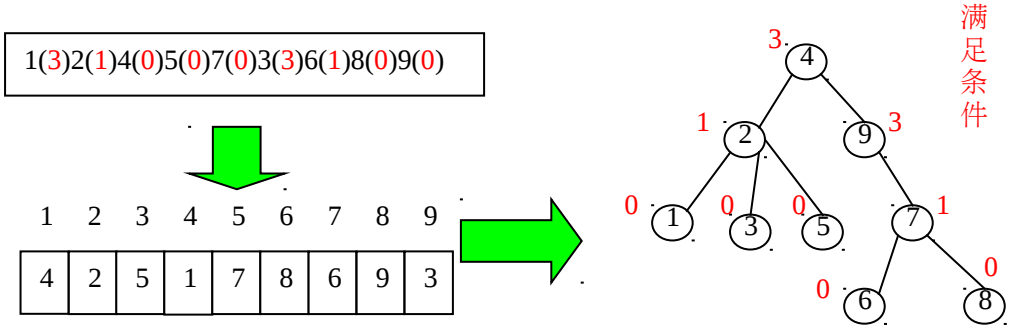
定义 *t(v)*为 *v* 的后代中所有权值小于 *v* 的节点个数。

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)*。

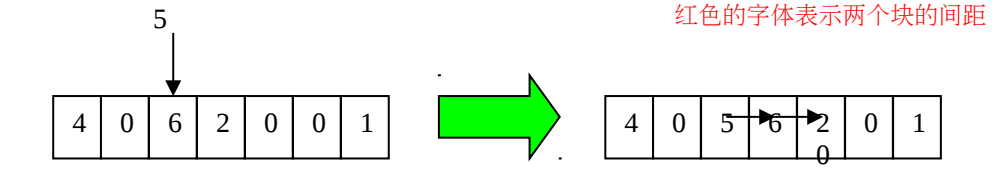
a. 若队列 *L* 位置上为空，那么士兵 *S* 站在 *L* 上。

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*）。

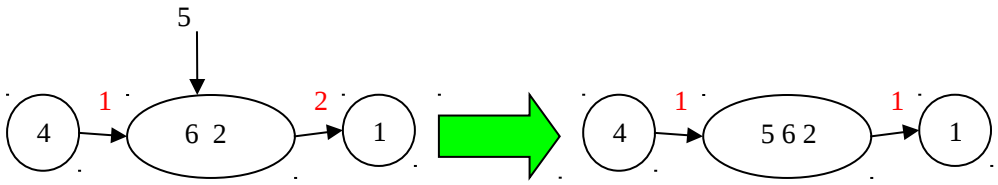
将军依次下达 *N* 个命令，每个士兵被下达命令一次且仅一次。要你求出最后队列的状态。（有可能在命令执行过程中，士兵站的位置标号超过 *M*，所以你最后首先要求出最终的队列长度。*0* 表示空位置）

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。

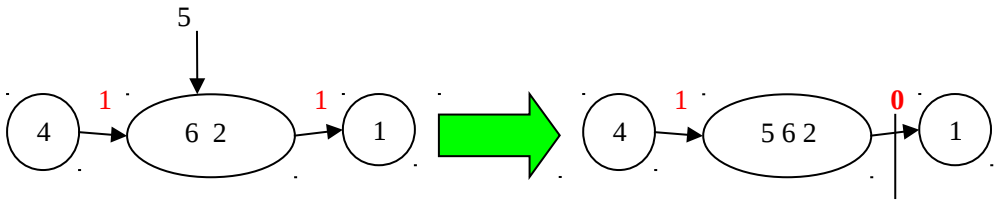
模拟的另外一种方法是，即把整块一起考虑。整块顾名思义，意思是：连在一起的士兵，叫做一整块。一个整块对外是一个整体，只是在块内部记录块成员之间的相对位置。这样做的好处：



插入一个元素，产生连锁移动。最坏时间复杂度O(N)



插入一个元素，不产生连锁移动。

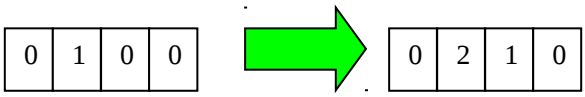


随之产生的问题——合并

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

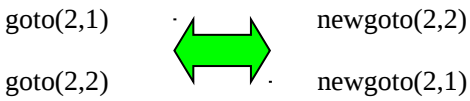
二叉树。一种更好的利用平衡二叉树的方法是，把所有格子按顺序构造一个平衡二叉树，每个节点记录一下空白位置，方便定位，删除等等，这样做时间复杂度是 O(N log N)。

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：

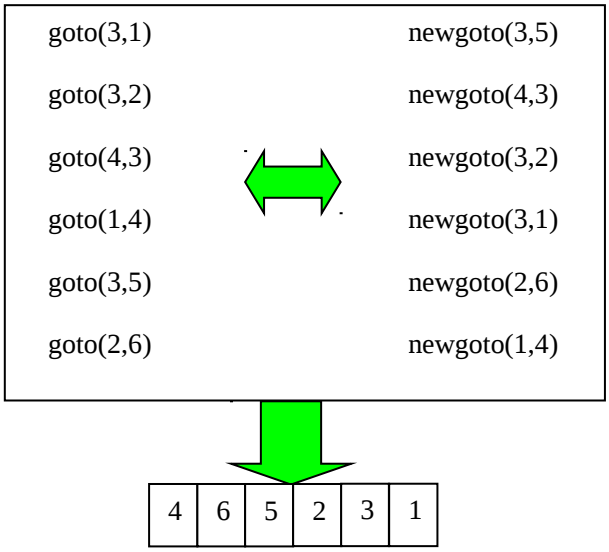


题目有一个特殊条件：每个士兵被下达命令一次且仅一次。这就相当于把士兵当作数，位置看作格子的话，这就是一个填数字的问题！上面的例子下达了两个命令 goto(2,1)和 goto(2,2)。而执行 goto(2,2)的时候，无可避免的进行了连锁移动。试想我们如果先填入 **2**，再填入 **1** 的话。我们事实上是将 **2** 忽略，将 **1** 填入了第二个空位置中！这和上一道题目最终转化成的问题如出一辙！

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 $O(N^2)$。

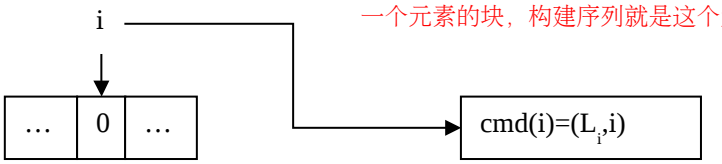
不妨换一种思路，用不完全模拟的方法来求出这个拓扑序列。所谓不完全模拟，就是说我们在模拟的时候，不记录块内的元素之间的位置关系。后面我们会看到元素之间在块内位置关系，和我们要求的序列没有关系。

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。

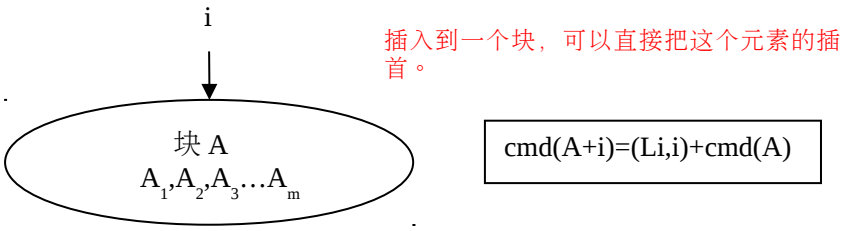
当执行的一个 **goto(L_i,i)**命令，分两步进行：

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)**。

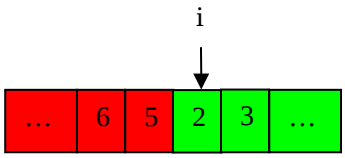


一个元素的块，构建序列就是这个元素的插入数对本身。



插入到一个块，可以直接把这个元素的插入序列放到序列首。

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：

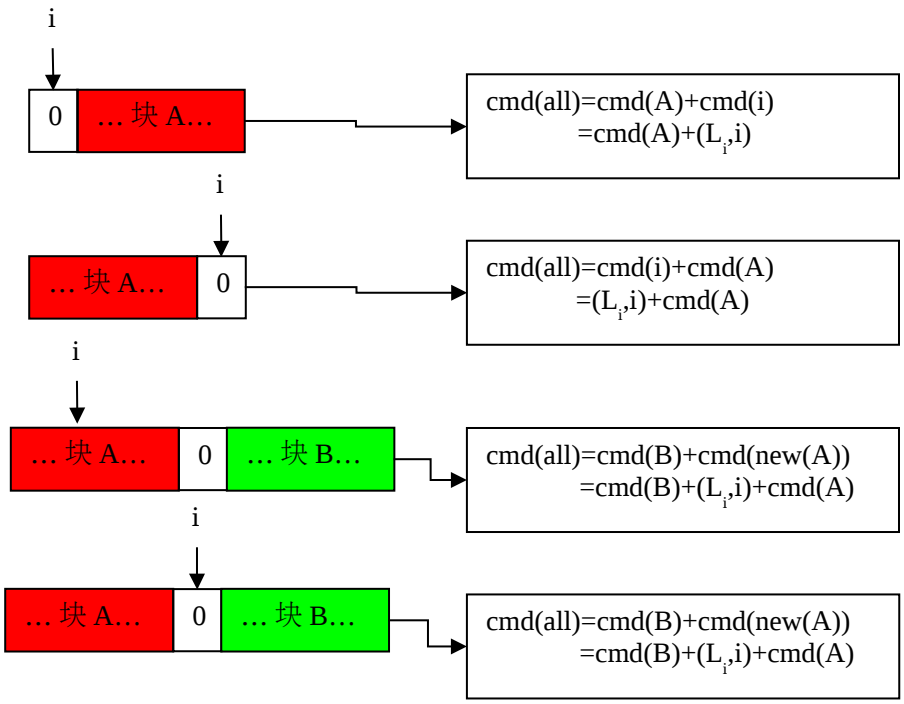


无论是之前还是之后的元素，都必须在 i 之后被插入。

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 $O(N\log_2 N)$。总时间复杂度为 $O(N\log_2 N)$。

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