信息学中的守恒法

【摘要】本文提出和总结了“守恒法”，以及它在信息学竞赛中的一些应用。守恒的本质是寻找变化中的不变量。守恒法能帮助我们跳过、避开纷繁复杂的细节，直接看透问题的本质。

【关键字】守恒法 不变量

【正文】

1、引言

现实生活和实际问题是纷繁复杂的。

问题 1 *两个质量相等的小球，速度分别为 5m/s, 4m/s，他们相向运动，完全弹性碰撞之后速度分别变成多少？*

问题 2 *10g C 和 10g O_2 在密闭容器中反应一个小时。最后的总质量是多少？*

问题 1 我们大概耳熟能详：动量守恒、动能守恒，两个方程就能解出速度。实际上小球碰撞的过程是复杂的，究竟两对力如何互相作用、互相影响、如何做功，思考起来是非常的复杂。如果忽略它们变化的具体过程，我们很容易发现“动量”和“动能”这两个变化中的不变量，抓住不变量，就能跳过繁琐的细节，直达目标。

问题 2 也是类似的题目。C 和 O_2 的反应同样是复杂的。在不同的局部，条件不同，可能产生 CO，也可能产生 CO_2；CO_2 和 C 还可能重新转化为 CO……事实上不可能有人列出三个化学方程去分析——在一个密闭容器中，无论怎么变，总质量必然不变——也就是质量守恒。抓住这一点，我们在 1 秒钟内就能说出答案：20g。

以上两个例子生动的说明守恒的作用。

现实生活和实际问题如此纷繁复杂，条件和变化如此之多，以至于我们考虑稍不周密就可能全盘皆错；抑或限于问题本身的复杂性，根本无法分析。

但是如果能找到一两个守恒量——也就是变化中的不变量，那么问题就能大大的简化了。忽略细节，抓住主要矛盾，这就是守恒法。

2、一个简单的例子

例题 1 有一个数列 $a_1, a_2, a_3, \ldots\ldots, a_n$。每次可以从中任意选 3 个相邻的数 a_{i-1},

a_i, a_{i+1}，进行如下操作（此操作称为“对 a_i 进行操作”）

$(a_{i-1}, a_i, a_{i+1}) \rightarrow (a_{i-1}+a_i, -a_i, a_i+a_{i+1})$

给定初始和目标序列。问：能不能通过以上操作，将初始序列转换到目标序列。

比如初始(1 6 9 4 2 0),目标(7 -6 19 2 -6 6)。可以经过如下操作：

(1 6 9 **4** 2 0)→(1 6 13 -4 **6** 0)→

(1 **6** 13 2 -6 6)→(7 -6 19 2 -6 6)

(加粗的是每次被操作的 a_i)

如果初始序列是(1 2 3)，目标是(1 3 2)。那么无论如何都不能通过操作从初始序列转换到目标序列。

Input.txt	Output.txt
1 6 9 4 2 0 7 -6 19 2 -6 6	Yes
Input.txt	Output.txt
1 2 3 1 3 2	No

我们分析一下这个题目。

操作是有先后顺序之分的。比如先对 a_2 操作，再操作 a_3；先对 a_3 操作，再操作 a_2，结果就有天壤之别。

观察 Sample：

(1 6 9 **4** 2 0)→(1 6 13 -4 **6** 0)→

(1 **6** 13 2 -6 6)→(7 -6 19 2 -6 6)

数字杂乱无章没有规律。仔细观察一下操作规则：

$(a_{i-1}, a_i, a_{i+1}) \rightarrow (a_{i-1}+a_i, -a_i, a_i+a_{i+1})$

直观的看，相当于把中间的数分别加到两边去，然后取反。容易发现，操作前后的总和是不变的！

我们可能很激动的猜想：只要两个序列和相等，他们就能通过操作互达。

但是第二个 sample 很快否定了这个想法：(1 2 3), (1 3 2)是不可达的。因为(1 2 3)能进行的操作仅仅是：(3 -2 5)，再进行一次操作回到(1 2 3)。所以永远不能变成(1 3 2)。

总和虽然不行，我们可以试着考察局部和。

(a_{i-1}, a_i, a_{i+1})	$(a_{i-1}+a_i, -a_i, a_i+a_{i+1})$
$S_1=a_{i-1}$	$S_1'=a_{i-1}+a_i$
$S_2=a_{i-1}+a_i$	$S_2'=a_{i-1}$
$S_3=a_{i-1}+a_i+a_{i+1}$	$S_3'=a_{i-1}+a_i+a_{i+1}$

很容易看出 $S_3=S_3'$，$(S_1,S_2)=(S_2',S_1')$。也就是说把 (S_1,S_2,S_3) 中的 S_1 和 S_2 交换位置就能得到 (S_1',S_2',S_3')。

稍微推广一下：设 $S_i=a_1+a_2+\ldots+a_i$，对(a_{i-1}, a_i, a_{i+1})操作本质上和交换 S_{i-1}, S_i 是等价的。

比如第一个 Sample：

(1 6 9 **4** 2 0)□(1 6 13 -4 **6** 0)□

(1 **6** 13 2 -6 6)□(7 -6 19 2 -6 6)

转化成和之后：

(1 7 **16** **20** 22 22)□(1 7 20 **16** **22** 22)□

(**1** **7** 20 22 16 22)□(7 1 20 22 16 22)

(加粗的是交换的两个数)

比如第二个 Sample:

(1 2 3)□S(1 3 6)

(1 3 2)□S(1 4 6)

对(1 2 3)的操作实质上是不断的交换 S(1 3 6)中的 1, 3。无论如何也不可能变成(1 4 6)，因为 4 根本没在 S(1 3 6)中出现过！

另外还有一点，参与交换的只有 S_1~S_{n-1}，S_n 是雷打不动的。所以我们算法出来了：

1、判断 S_n 是否相等。

2、判断 $\{S_1, S_2, \ldots, S_{n-1}\}$ 是否相等。

0(nlogn)的时间复杂度（排序复杂度）。

一个看似繁琐的题目被很轻松的解决了。

如上文所言，操作的变化过程是纷繁复杂的，可以说没任何规律。如果从每一次操作对总体的贡献入手研究，会碰得头破血流。

但是我们把原来的数列进行了一个小小的转化：求和。正是这个求和，使得操作的本质浮出水面，操作由令人头晕目眩的(a_{i-1}, a_i, a_{i+1})□ $(a_{i-1}+a_i, -a_i, a_i+a_{i+1})$、变成了简单的“交换 S_{i-1} 和 S_i”。

这是一个应用守恒的经典例子。其中的守恒量是$\{S_1, S_2, \ldots, S_{n-1}\}$，这个集合始终保持不变，不会有新的元素生成、也不会无缘无故有元素消失；有的只是顺序上的交换。

抛开琐碎的细节，我们抓住了本质。

为什么会想到求和呢？

因为(a_{i-1}, a_i, a_{i+1})□ $(a_{i-1}+a_i, -a_i, a_i+a_{i+1})$的结构很容易让人发现它的总和守恒；进一步联想到局部和。

问题本身的结构是守恒量选择的主要依据。

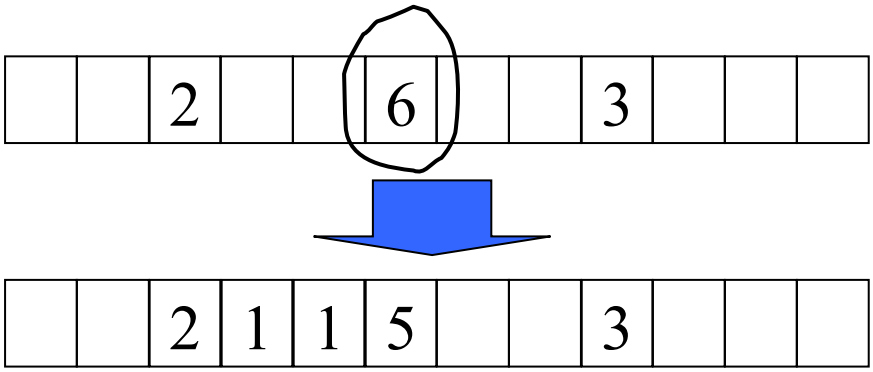
联想和化归是构造守恒量的关键。

此题特点明显，我们再来看一个难一点的试题。进一步领略守恒法的魅力。

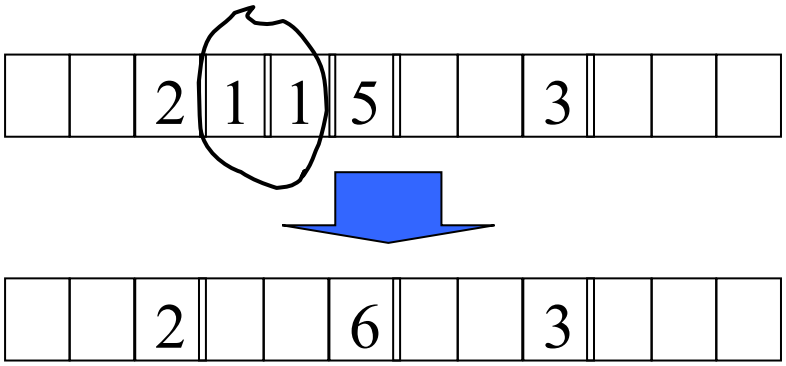
3、一个隐蔽的例子

例 2 跳棋(Jump, POI IV Stage I Problem 2)

有一列无限长的格子里面，某些格子里面放了棋子。如果某个格子里面有棋子，就可以拿走这一颗，并且在这个格子的左边两个格子里面各放一颗。



如果连续两个格子里面都有棋子，可以分别在两个格子里面各拿走一颗，并且在它们右边的格子里面放一颗。



现在的任务是：

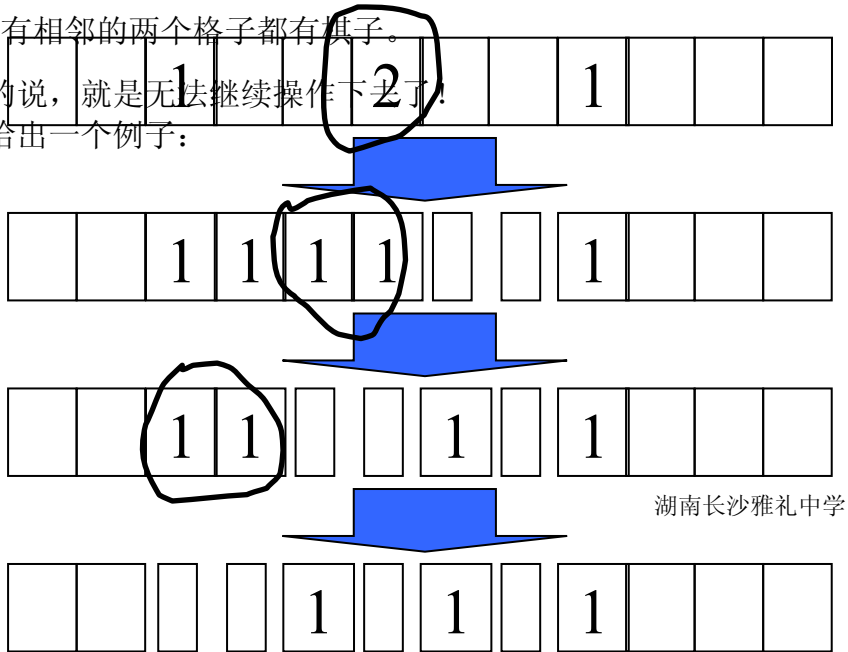
给定初始状态，要求使用以上操作，使得：

1、每个格子至多只有 1 颗棋子

2、没有相邻的两个格子都有棋子。

简单的说，就是无法继续操作下去了！

下面给出一个例子：

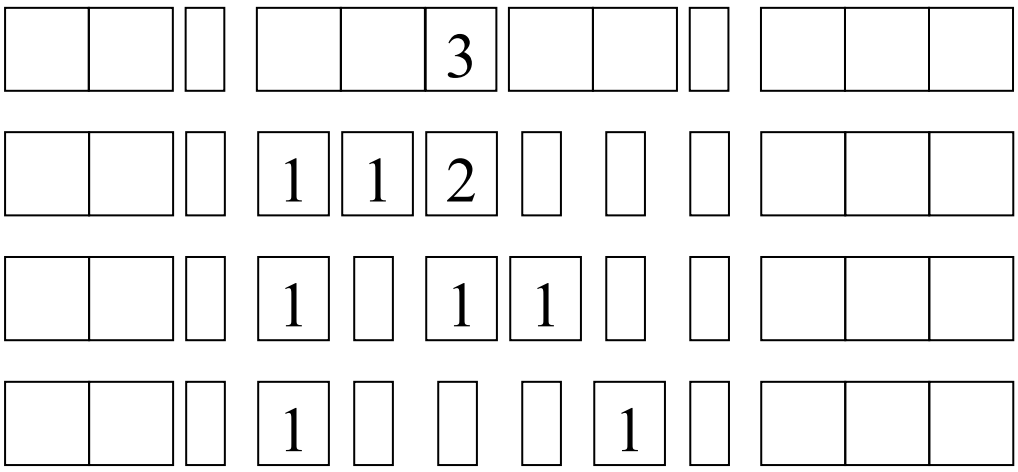


Input.txt　　Output.txt

0 0 1 0 0 2 0 0 1　　0 0 0 0 1 0 1 0 1 0 0 0

下面我们来分析一下这个问题。

这又是一个操作问题。要求不存在格子有>1 的棋子，现在考虑如果一个孤立的格子里面有 3 颗棋子该怎么办：



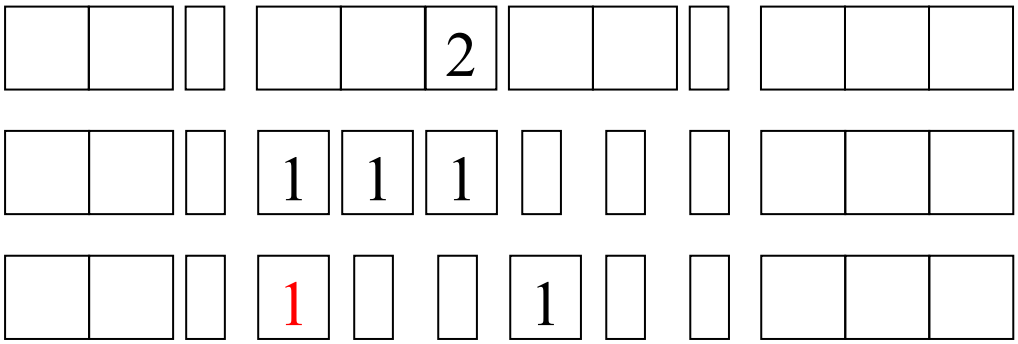
通过这样的转化就能把棋子数较少一个。并且新产生的两枚棋子中间隔开了 3 个格子。

我们可以不断的进行这样的操作：

1、　如果有相邻的格子都有棋子，则用基本操作 2，将他们拿掉，并且放一颗到右边的格子。

2、　如果某格子棋子>=3，则进行上述操作。

按照这两条规则操作到不能操作为止。此时有棋子的格子，必然是一些互不相邻的或包含 1 颗棋子、或包含 2 颗棋子的格子。如果有 2 颗棋子，可以这样做：



每次找最右边的 2 操作。

如果造成新的“相邻有棋格子”或者“>=3 颗棋子的格子”，我们就能设法让棋子数至少减 1。下面考虑没有造成“相邻有棋格子”或者“>=3 颗棋子的格子”。

如果红色的 1 所在的格子本来就是空的，那么我们就能减少一个含 2 的格子。

不然红色 1 所在的格子本来必然是 1，加了一个红 1 后，变成 2 了。

这样我们就把最右边的 2，往左边移了一点。对新生成的 2，继续这样的“移动”。因为不可能有无穷多个 2，所以移到左边的一定位置一定有一个尽头。最后我们就能把 2 的个数减少一个了。

综合上面的说法，我们要不能把棋子数减一，要不能把 2 的个数减一。因为只有有限的棋子和有限的 2，总能把棋盘变化到满足题目最终要求的样子。

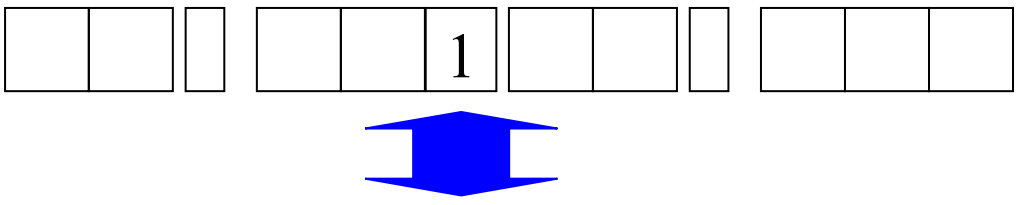
上面我们证明了解的存在性。与此同时，也给出了一种可行解决方法。利用对 3 和 2 的两种基本操作规则不断运用可以求得最后的答案。

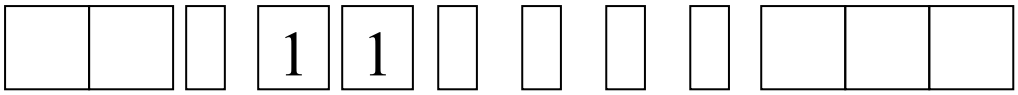
实践证明这也是一种不错的方法，程序运行速度很快、编程复杂度也不太高。

但是解的唯一性呢？是不是只有一组解？另外这种方法有很大的偶然性，时间复杂度不好估计、无法保障。

另外解答给人一种很凌乱的感觉，这促使我寻找更好的方法。

容易发现，题目给出的两个操作是可逆的。





观察一下上面的图，我们会有这样的感觉：一颗第 4 格子的棋子和一颗第 5 个格子棋子合起来，等价一颗第 6 个格子的棋子。

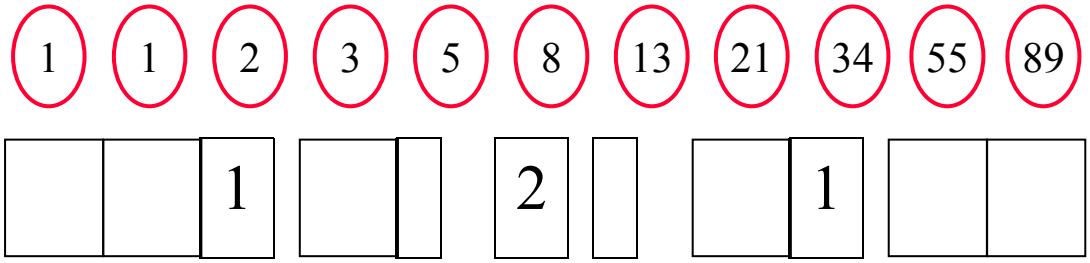
之所以说“等价”，是因为两者可以互相转化的。通过题目给的第一种操作可以实现 6□ 4+5；通过第二种，可以实现 4+5□ 6。

我于是产生一种奇妙的想法，何不给每个格子一个量化的值呢？比如设第 i 个格子的价值是 Fi，那么 Fi 必须满足：$F_{i+1}=F_i+F_{i-1}$。

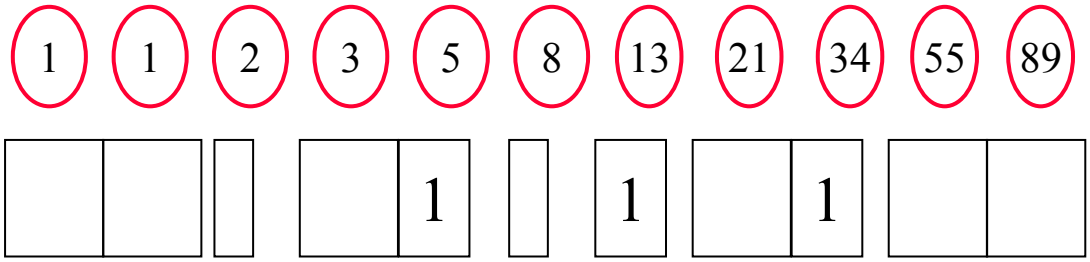
这不是 Fibonacci 数列吗？

如果真能如此，那么在不断变化过程中，所有棋子的价值总和将保持不变。

事实上正是如此。我们可以上面的 Sample 来验证一下。



总价值是 2*1+8*2+34*1=52



总价值是 5+13+34=52

两者的总价值是相等的！

现在的问题是，我们如何利用这一点呢？比如初始状态算出来总价值是 52，我们又怎么知道目标状态的这个 52 是如何分布的呢？

因为每个格子只多只能有一个棋子，并且不能存在相邻格子都有棋子；所

以最后结果必然是**若干不相邻**的 Fibonacci 数的和。

把一个数分成若干不邻的 Fib 数的和，称之为 Fibonacci 分解。

定理 任意自然数 n 的 Fibonacci 分解唯一。(F_1=1, F_2=2)

证明：设不大于 n 最大 Fib 数是 F_p，那么 F_p 必然要包含在分解中。否则：

当 p 是偶数

$F_{p-1}+F_{p-3}+\ldots+F_5+F_3+F_1$
$=F_{p-1}+F_{p-3}+\ldots+F_5+F_3+F_2-1$
$= F_{p-1}+F_{p-3}+\ldots+F_5+F_4-1$
……
$=F_p-1<Fp<=n$

当 p 是奇数

$F_{p-1}+F_{p-3}+\ldots+F_6+F_4+F_2$
$=F_{p-1}+F_{p-3}+\ldots+F_6+F_4+F_2+F_1-1$
$=F_{p-1}+F_{p-3}+\ldots+F_6+F_4+F_3-1$
……
$=F_p-1<F_p<=n$

因此 F_p 必然包含在分解中。令 n□ n-Fp，然后找不大于 n 的最大 F_q。必有 $q\neq p-1$。否则 $F_p+F_q<=n$，$F_p+F_{p-1}=F_{p+1}<=n$，这和“不大于 n 的最大 Fib 数是 F_p”矛盾。

如此进行下去。最后必能顺利分解完毕。

证毕。

分解方法是唯一的；同时之前提出的模拟算法证明了解的存在性质，所以该分解方法必然是一组合法的解！

编程要用到高精度，先算出输入数据的总价值和，然后进行 Fib 分解，输出解即可。

解的存在性和唯一性都得到证明。并且提出了两个可行的算法：模拟法和 Fibonacci 选择法。

模拟法时间复杂度不太好估计，而且算法凌乱；但是他的优点是编程不难，速度也比较快。

Fibonacci 选择法理解上难度大一点，很巧妙，能给人以数学美的享受。但是他要使用高精度，编程上和模拟法不相上下。

最后还有一个问题：Fibonacci 给格子标记权的时候，必须从某一个格子开始标 1，2，3，5……到底从什么地方开始才能保证最后答案的正确性了？

换句话说，最后的结果中，最左边的棋子会偏到什么位置？

（理论上可以从最左边的无穷远处开始标，足以证明唯一性。但是涉及实际编程的时候，必须考虑具体位置，越靠右越好。）

实际上在模拟法中，我们可以找到答案。

根据对 3 和 2 的操作方案，可以知道：（设 M 是棋子总数）

1、对 3 的操作至多把棋子向左边推移 2*Log(3, M)位。

2、当只剩下 1，2 的时候，对 2 操作后，大多可以生成新的“3”，归结到对 3 的操作。

3、如果对 2 操作后，没有形成新的“3”或者“相邻有棋格子”，就只能最右边的 2 不断操作，使得最右边的 2 不断的往左边移。这样最后只会把最左边的棋子推动 2 位。此操作最后对左移棋子的贡献是常数级别的。

也就是说，假设输入数据中最左边的棋子在 a 位置，那么从大概 a-2*Log(3,M)的位置开始标号就可以了。

为了保险，可以多取一点，比如 4*Log(3,M)。这里 M 的范围不超过 10^9。

此题的解答令人耳目为之一亮。从“模拟棋子”到 Fibonacci 数列，看似毫不相干的东西居然能联系起来。

最了不起的是，Fib 的方法完全舍弃了具体的细节。棋子是怎么移动的，采用第一条规则还是第二条规则都不用管，我们需要的是最后的结果！而结果满足一个条件：总价值守恒！

利用这个变化中的不变量，我们很轻松的解决了问题。

“总价值”是题目中没有的概念。进一步说，“价值”也没有出现在题目中。虽然都是格子、虽然都是棋子，但是规则却赋予他们不同的地位。我们正是在观察规则的基础上，总结特点，才将思路不可思议的转到了“价值”这个概念上，并且进一步踏上了 Fibonacci 数列的正轨。

4、上楼梯问题

例 3 Conqueror’s battalion(CEOI2002,day1,problem1) 题目大意如下：

总共有 n 阶台阶，一些台阶上有你的若干士兵。你要把所有的士兵分成两组，然后敌人就会告诉你哪一组士兵留下，哪一组士兵被消灭。那些留下的士兵上一层楼。

然后你重新把剩下的士兵分组，敌人再次选择一组留下；留下的士兵又上一层楼。

……

如是反复。如果最后有一个士兵登顶，也就是上了第 n 层台阶，你就赢了；如果士兵全部被消灭完，敌人赢了。

现在输入 n 和每一层台阶上有多少你的士兵。你的任务是判断你能否获胜；如果能获胜，还必须和库文件交互，由库文件充当敌人，你要通过适当的分组，最终获得胜利。

下面分析这个题目。

设身处地的想，如果你是敌人，你怎么办？对方把士兵分成两组，你会杀掉哪一组？是不是人数最多的那一组？不是。

比如第 n-1 层上站了 1 个人，他单独作为一组；第 1 层上占了 100 个人，他们是一组。你会杀谁？

当然是第 n-1 层上的那个人。因为如果不杀，下一轮，他就上到第 n 层、对方就获胜了！

可见，站在第 n-1 层的人比站在第 1 层的人，更具有威胁性——或者说，短期内，能造成更大的危害。

一般的，站得越高，危险性越大。这里提示我们士兵的地位是不同的，不能以人数的多少来作为判定的依据。

回到我们自己，看看如何才能赢得胜利。

设想只有 2 个人站在第 i 层台阶。

你绝不可能把他们分到一组；否则敌人下一轮就能让他们两个全部玩完。所以肯定是两个人分别在一组。

敌人让某个人死亡后，剩下的那个就能上到第 i+1 层台阶了。

也就是说：站在第 i 层上的 2 个人，其作用相当于站在第 i+1 层上的 1 个人。

类似的模仿例 2，我们令 Wi 表示站在第 i 层上的人的价值，那么 $2W_i=W_{i+1}$。令 $W_0=1$，则 $W_i=2^i$。

最后的目标是要让一个人站在第 n 阶上，也就是目标状态的价值和至少是 2^n。

设想初始状态的总价值也是 2^n。我们分组的时候就设法分成两组，使得每一组的总价值和都是 2^{n-1}，这样无论敌人去掉哪一组，剩下的那组价值和始终是 2^{n-1}。

由于剩下的士兵要上爬一层，相当于每个人*2，总价值和变成了 2^n。

所以在博弈的过程中，总价值和是守恒的：始终保持为 2^n！由于人数不断减少，最后只剩下一个人的时候，必然已经登顶了。

反之如果总价值$<2^n$，分成两组后，必然有一组$<2^{n-1}$，敌人可以把该组保留。该组士兵上爬一层，总价值$<2^n$。变化过程中总价值始终$<2^n$，也就是说无论如何也无法登顶。

如果总价值$>2^n$，总能分成两组，使得都$>=2^{n-1}$。这样变化过程中总价值就始终$>=2^n$了。人数不断减少，最后总有一个人能登顶。

具体怎么分，可以采用从大到小的贪心，这里不详细说了。因为和本文主题无关。这里只想要重点介绍这种标权值的方法。

判断能不能取胜，转化成了判断总价值和是否$>=2^n$。如果$>=2^n$，我们就能想方设法的分组，使得总价值一直保持在 2^n之上；随着人数的减少最后取胜。

“总价值始终保持在 2^n之上”这就是本题的守恒。严格的说他已经不是“等于”关系了，而是一种大于关系。算是广义的守恒吧。

但他的基本思想还是找变化中的不变量。用二进制标记个层阶梯是一个很好的尝试，这种尝试也是建立在仔细观察和联想的基础上的。**结合题目的特点，找变化中的不变量！**

【总结】

以上的几个题目，本身都有一定隐蔽性，看不出什么噱头。只有通过一定的转化，具体地说比如求和、标权值，才能发现其本质。

例 1 为什么求和？例 2 为什么是 Fibonacci 权？例 3 怎么又是二进制？再一次强调：

问题本身的结构是守恒量选择的主要依据。

联想和化归是构造守恒量的关键。

观察和联想是最关键的。观察是观察数据的规律、规则的特点等等；联想是在观察到的特点基础上，与已有的知识储备进行联系，或者自行创造。

知识的积累是很重要的。否则即便是正确的归纳到了特点，也无法有效的类比创造。

另外试图用“守恒法”解题的话，创造性的思维是很重要的。有很多守恒量并不能在题目中找到，必须自己创造。有一个经典的例子是 Fibonacci Heap。

Fib Heap 本身实现没有用到守恒法，但是分析复杂度的时候用到了“势能法”。该方法通过构造一个函数，证明了 Fib heap 删除操作的均摊复杂度是 O(logn)。这也算是守恒思想的一个了不起的应用。

其实“守恒”随处可见。你不经意的列一个方程就是守恒。本文说的守恒是说在题目中挑大梁、演主角的一种思维方法和解决问题的算法。

大概任何人早就有意无意的多次用到了这种思想，本文只是想将这种方法的主要特点进行归纳，并且着重提出，引起注意。以后在遇到问题的时候，可以有意识的往这方面联想，或许能有意向不到的收获。

【参考文献】

CEOI 试题——例 3
POI 试题——例 2
雅礼内部测试题——例 1

【感谢】

感谢长沙长郡中学金恺在例 3 中上对我的帮助。

感谢长沙雅礼中学姚金宇同学提供例 1。

【附录】

例 2 的原题：

POI IV Stage I Problem 2

Jumps

"Jumps" is a board game played on an infinite tape of squares. The board has neither left nor right limit. On the squares there is finitely many pieces (more than one piece on a square is allowed). We assume that the most left, non-empty square is numbered 0. Squares on the right are denoted by consecutive natural numbers 1, 2, 3, ..., squares on the left - by negative numbers: -1, -2, -3, ... The configuration of pieces on the board can be described in the following way: for every square occupied by at least one piece we give the number of the square and the number of pieces standing on this square. There are two kinds of moves that can change the configuration: jump right and jump left. Jump right consists of removing two pieces from the board: one from a square p and the other from the square p+1, and placing one piece on the square p+2. Jump left consists of removing a piece from a square p+2 and placing two pieces on the board: one on the square p+1 and the other on the square p. We say that a configuration is final if each pair of neighbouring squares contains at most one piece. For every configuration there is exactly one final configuration that can be reached after finite number of moves (jumps).

Task

Write a program that:

- reads a description of an initial configuration from the text file SKO.IN,
- finds the final configuration that can be reached from the given initial configuration and writes the result to the text file SKO.OUT.

Input

In the first line of the file SKO.IN there is one positive integer n, 1 <= n <= 10000, which equals the number of non-empty squares of the given initial configuration. In each of the following n lines there is a description of one non-empty square of the initial configuration. Such a description consists of two integers. First is the number of the square, second - the number of pieces standing on this square. The descriptions are given in increasing order with respect to numbers of squares. The biggest number of a square is not greater than 10000 and the number of pieces on a single square is not greater than 100000000.

Output

In the first line of the file SKO.OUT there should be written the final configuration that can be reached from the given initial configuration. More precisely the line should contain the numbers of non-empty squares (in increasing order) of the final configuration. The numbers should be separated by single spaces.

Example

For the text file SKO.IN

2
0 5
3 3
the correct solution is the file SKO.OUT
-4 -1 1 3 5

例 3 的原题：

Conquer

试题

有两个人在玩一种叫“征服”游戏，其中征服者一方在最初拥有最多 1 000 000 000 名士兵。这些士兵站在 N 高地(2<=N<=2000)，每个高地上有若干士兵。征服者可以按任意方式将高地上的士兵分成两队，被征服的一方可以叫任何一队离开，留下的士兵将顺次占领上一级高地。如果征服者一方最终至少有一名士兵到达最高点，征服者就获胜，否则他应该继续这个游戏，直到没有士兵时就失败了。

这是一个交互式的问题。测试者将提供给你一个初始库，假设你为征服者，你将依靠这个库进行游戏，每一轮游戏，你需要提供给库一个分组方案。库将根据你的提供，返回 1 或 2 (1 表示留下你描述给库的那队士兵, 2 表示留下其余的士兵).如果你获胜或者没有士兵进行游戏时，则游戏结束，库将自动终止你的程序，你不能用任何方式进行终止。

库接口

库 libconq 提供两个子程序:

- start – 返回 N 和一个数组 stairs 的数值，表示每个高地的士兵数目 (即 stairs[i] 为高地 i 的士兵数，其中 stairs[1]为最高点)

- step – 提供给它一个入口数组参数 subset (有 N 个元素),描述了一次士兵分队的方案。它将返回 1 或 2（含义见上）;

如果你提供了一个无效的分组方式，库将自动终止你的程序，并得 0 分。

下面是 FreePascal 和 C/C++对子程序的描述:

```
procedure start(var N: longint; var stairs:array of longint);
function step(subset:array of longint): longint;
void start(int *N, int *stairs);1
int step(int *subset);
```

下面给出了一个 FreePascal 和 C/C++关于库使用的样例; 程序包含了两个部分：开始游戏，随机选择某一队士兵。你的程序将可能包含每一轮游戏，循环进行。你可以样例的方式定义数组。(注意在返回时 FreePascal 可以按定义的方式，而 C/C++只能是从 1 到 N)

FreePascal example

```
uses libconq;
var
   stairs: array[1..2000] of longint;
  subset: array[1..2000] of longint;
  i ,N,result: longint;
...
start(N,stairs);
...
for i:=1 to N do
   if random(2)=0 then
      subset[i]:=0
   else
      subset[i]:=stairs[i];
result:=step(subset);
...
```

C/C++ example

```
#include "libconq.h"
int stairs[2001];
int subset[2001];
int i,N,result;
```

```
...
start(&N, stairs);
...
for (i=1;i<=N;i++)
   if (rand()%2==0)
      subset[i]=0;
   else
      subset[i]=stairs[i];
result=step(subset);
...
```

你必须在你的程序使用 uses libconq 进行库的连接（FreePascal）而在 C/C++中使用#include "libconq.h" 形式。这时，程序将对加入的 libconq.c 中的参数一起进行编译.

样例

You	Library
start(N,stairs)	N=8, stairs=(0,1,1,0,3,3,4,0)
Step((0,1,0,0,1,0,1,0))	returns 2
Step((0,1,0,0,0,1,0,0))	returns 2
Step((0,0,0,3,2,0,0,0))	returns 1
Step((0,0,2,0,0,0,0,0))	returns 2
Step((0,1,0,0,0,0,0,0))	returns 2
Step((0,1,0,0,0,0,0,0))	no return: you won

资源

你可以先建立一个 libconq.dat 文件, 该文件包含两行，第一行为 N ,表示高地数，接下来的一行包含 N 个整数，分别表示高地 1..N 中的士兵数目，当然最开始高地 1 不会有士兵.

libconq.dat

```
8
0 1 1 0 3 3 4 0
```