骗　分　导　论

Introduction to Score-cheating in Informatics

第一版 2009-02-21

李 博 杰

石家庄二中 2007 级 19 班　　博杰学习网[1] http://boj.pp.ru/

摘要

本文以竞赛心态的调整为开端，以常数时间优化为基础，以数学分析与猜想为指导思想，以非完美算法为主要策略，以搜索为最后的万能策略，讲述**信息学竞赛**[2]中“**骗分**[3]”的若干策略，再进行实战演习，说明“骗分”的强大功力。

关键字

骗分 信息学竞赛 复杂度 数学 非完美算法 调试

目录

[1] 博杰学习网，官方顶级域名为 http://boj.pp.ru，是本文作者的个人网站，主要发表各学科竞赛（主要是数学、信息学）的信息动态、竞赛试题解答、学习资料、个人研究成果，趣味科学，个人随笔文章，班级事务，是石家庄二中首个学科竞赛资料库、河北省信息学代表队资料库，欢迎访问。
旗下网站有数之理论坛 http://boj.5d6d.com/，属于数学、信息学趣题、文章、资料交流的场所，其中大量语言精辟、分析透彻的教程文章和趣味十足、深入本质的趣题值得读者一看。

[2] 什么是信息学竞赛？笔者认为，信息学竞赛就是考察在有电脑的情况下“如何使用计算机代替人脑的工作，使得人的工作效率更高”，就是考察应用计算机这个高效工具解决实际问题的能力。

[3] 本文所说的“骗分”不是作弊，而是尽可能多得分的应试技巧，请不要误解。

0 前言

本文是《学科竞赛骗分导论》系列文章之二——《骗分导论·信息学竞赛》，同时是NOI2009 河北省代表队论文、博杰学习网 2009 年研究发展规划重要成果。笔者在《骗分导论·数学竞赛》第三版（81 页）受到广泛好评之后，又受到“我是智障”大牛的《骗分导论》启发，决定再写一篇有关信息学竞赛尽可能多得分的方案。

取同样的题目《骗分导论》，并无抄袭之意，只是为了更加具体的阐明“骗分”的方案，为同名文章增添理论色彩、方法色彩，更加具体的阐述“骗分”理论，使“骗分”真正成为信息学竞赛中的有效策略。

笔者在 NOIP2008 中取得较好的发挥，以 330 分进入河北省信息学代表队，又参加了 WC2009 冬令营，取得了 101 分的较好成绩，但真正实力，尤其是高级算法的实力并不强，在欣喜之余，应该总结一些有关“骗分”的技巧，为 OI 事业的长足发展提供动力。

在我学习信息学竞赛的过程中，曾经多方寻找相关资料，而每份资料在介绍时都不能面面俱到，甚至出现两种资料所述观点相反的情况，给我们的学习带来很大不便。据我所知，本文是首篇具体、全面总结信息学竞赛技巧的文章，可以称得上是“第一个吃螃蟹的人”。希望本文的发表能为更多具有总结性质的信息学竞赛资料的出现起到抛砖引玉的作用。

正是因为笔者写作本文的动机是为广大信息学竞赛选手整理、提供一份较为全面的信息学竞赛得分技巧，笔者决定开放本文的版权，任何人都可以修改、添加本文的部分并发布，也可以在注明出处的要求下任意引用本文的语段、程序。

正是因为本文的总结性质，笔者不能保证内容的原创性。如果把《骗分导论》比作是一束美丽的鲜花，那么其中的朵朵鲜花都是各位大牛辛勤努力的成果，本文仅是做了一个筛选鲜花、制作花瓶并将花插入花瓶的工作。

本文的长度有目共睹，为了普及、易懂，拉近与读者的距离，况且不是正式出版物，上面也许有一些废话，或十分简单的试题，请各位大牛原谅，并可以选择性阅读。本文参考了众多大牛的论文、高级教练的书籍，并摘录了一些语段和程序，在此表示衷心感谢。

事实上，笔者在写这篇文章时，感到比《骗分导论·数学竞赛》要轻松一些。因为数学竞赛是要推理过程的，严格的证明始终是困难的；而信息学竞赛只需要结果，而且有“部分分”，这些“黑箱测试”的特点都给“骗分”提供了极大的方便。

本文以竞赛心态的调整为开端，以常数时间优化为基础，以数学分析与猜想为指导思想，以非完美算法为主要策略，以搜索为最后的万能策略，讲述信息学竞赛中“骗分”的若干策略，再进行实战演习，说明“骗分”的强大功力。

值得注意的是，本文将不介绍“我是智障”《骗分导论》中“直接输出结果”的方法，因为在近几年的 OI 竞赛中，这种方法很难得分，也不能体现信息学竞赛的考察本质。

“会做才是硬道理”，本文的多数手段事实上不是“骗分”，而是如何采用稍微差一些的算法得到较高的分数。通过阅读本文，相信读者不仅能学到“骗分”的有关技巧，更能得到“做题”的一些启示。

如果说“**装备精良兵器，精通诸子兵法，部署优势兵力**[1]”是战争取胜的三要素，那么在信息学竞赛中，熟练的编程就是精良的兵器，高效的算法就是诸子兵法，巧妙的策略就是优势兵力。

本文采用了“注释”的写作风格，将大量用于定义、解释、补充说明的语句放在每页下方的注释中，并且对后文要提到的内容在注释中先给出提要，方便理解。这样可以使文章逻辑更加严谨，中心更加突出。需要注意的是，“注释”不等于“不重要”，很多文章中“无地可容”的思想和一些细节方面的看法往往包含在注释中。希望读者习惯这种方式。

本文中出现了很多代码，原因主要有二：首先，本文不是正式出版物，篇幅长短并无大碍，希望读者不要为了节省宝贵的磁盘空间而在意 PDF 的大小；更重要的是，本文是以介绍竞赛策略为核心的信息学文章，需要对编写代码、调试代码的细节进行讲解，而不是像其他文章一样只给出解题的思路和算法。读者如果对这些代码感到厌烦，可以略过它们。

为了充分利用网络资源，《骗分导论》依托数之理论坛开通了官方网站[2]，发布本文的最新版本，有关试题、源程序、测试数据，配套学习资料等，也方便广大读者朋友对本文提出意见建议，与作者对话，讨论本文中涉及的试题，交流“骗分”的思想方法。

由于“骗分”是一个本来不正规的方法，也是多种思想综合的结果，本文强行将各种方法分成章节，未必符合人类的思维特点，也势必造成大量重复，包括事例重复、论证重复，请读者多多包涵。

由于笔者水平有限，错误、疏漏在所难免，希望读者指正，并为《骗分导论》的再版提出意见建议。我们对您的支持表示感谢。

NOIP 取消保送的传言前一段时间沸沸扬扬。竞赛回归自然，确实是 OI 应有的取向。学习 OI，不仅是为了得奖、保送，更是为了学习知识，掌握技能，服务于其他学科的学习和今后的发展。事实证明，学习信息学竞赛的学生在其他学科，尤其是数学上往往有逻辑思维强、表达严谨、知识面广等优点。

目前国产软件的普及仍然远不如国外市场的垄断，这也对我国的信息安全构成了巨大威胁。为了实现民族独立自主发展，就必须有自己的信息产业，有自己的独立知识产权。国内教育界对信息学的重视程度也远远不够，这些难题都需要我们这一代 OIER 做开路先锋。

本文使用永中集成 Office 编辑、制作，并在红旗 Redflag Linux 环境下发表，也是为了支持民族软件，振兴民族产业。我们看到，国产软件无法获得普及的重要原因不是技术低、水平差，而是众多用户对民族软件不了解，还没有使用就认为低人一等。只要我们改变对民族软件的不公平认识，共同反对外国公司的垄断，我们民族软件产业的振兴，就为期不远了。

学习 OI，为民族信息学产业的发展，甚至人类信息学事业的进步贡献力量。我们希望这不是一句空话，而能够让每个人身体力行。

如果这个宏伟目标能够成为现实，或者本文能为一批 OIER 走上民族产业振兴之路做出一点积极的影响，笔者为撰写《骗分导论》花费的巨大精力，也就在所不惜、有所充实了。

衷心祝愿各位读者在《骗分导论》的引导下，在 OI 之路上越走越远，越走越成功。

1 骗分——与复杂度的较量

从本质上说，“程序优化”就是**降低时间复杂度**[3]的过程。而“骗分”，则是**降低编程复**

[1] 引自沈文选《平面几何证明方法全书》序言。
[2] http://boj.pp.ru/olympic/comp/pianfen/
[3] 时间复杂度，不是指程序的运行时间，而是当数据规模扩大时运行时间的增长速度。在信息学竞赛中，

杂度[1]的过程，更准确的说，是**取得得分与编程时间的最优比值**。

有人会问，如果这道题根本不会做，连算法都没有，谈何“降低时间复杂度”？其实不然。对于任何问题，只要它是一个**可解问题**[2]，就一定可以通过枚举法来解决。即使是**NPC问题**[3]，只要将所有可能的情况一一列举，都能在有限时间（不论它有多长）中解决。所以从理论上说，可解问题都有最低时间复杂度。

但在实际编写程序时，我们不能让程序运行 1 年甚至更长，于是需要更为高效的算法。我们信息学竞赛学习如此众多的算法、优化技巧，说到底，就是在降低程序运行的时间复杂度，使得程序在规定时间内出解。正是基于这样的观点，本文的各种算法都关注了时间复杂度的衡量，而且不同于理论著作，更加注重实际运行时间的比较，因为理论复杂度不能说明一切问题，应试中的时限才是硬道理。

要纠正一种错误的观念，“**时间复杂度越低越好**[4]”。有时为了降低一道题目的复杂度，耽误了其他题目解决的时间，得不偿失；这时不如放弃这道题目的进一步优化，争取得到 50-60 分，再去解决其他题目。这种策略尤其对于难题、高档次考试有效。另外，复杂度记号中的常数因子也是不可忽略的。例如搜索中剪枝本身的代价超过被剪掉的搜索代价，就不如不剪。这也提醒我们，养成良好的编程习惯，降低常数复杂度，也是“骗分”的重要手段。

在信息学竞赛中，“**用空间换时间**[5]”是常用的策略。这里我们再提出一个观点：**用运行时间换编程时间，用尽时限**[6]中的每个 CPU 指令。

考虑一个问题：有 **O(nlogn)**[7]的 A 算法但需要 200 行代码，而有 $O(n^2)$的 B 算法只需要 70 行代码，而对于 60%的数据，n<1000；对于 100%的数据，n<100000. 我们需要做出一个决策。采用 A 算法，至少得 60 分，可能用掉 40 分钟；采用 B 算法，100 分到手，可能用掉 100 分钟。对于**不同类型的比赛**，我们应有不同的决策。对于 NOIP 这种其他题目较为简单、这样的试题是压轴题的情况，可以先做其他简单试题，最后视剩下时间长短选择算法；而对于 NOI 这种“难题集萃”类型的试题，如果其他试题没有思路，可以用较长的时间完全解决该题；否则两道 60 分的题胜过一道 100 分的题。

当然，算法的选择还与选手的**竞赛目标**[8]有关。期望拿到 NOIP 一等奖或 NOI 银牌的选手可能希望尽量多得分，实力也不是很强，能得一分算一分，这样应选择 60 分的算法，以“保险”为基础；期望进入省队或国家集训队的选手必须发挥出色，编程水平较高，这样应选择 100 分的算法，拉大与其他选手的差距。

总而言之，算法的选择与“得失”的衡量有关，这样本文所述的“用运行时间换编程时间”就是正确的。例如 NOIP1999 第四题“拦截导弹”，要求输出最长不升子序列。这道题有 $O(n^2)$的动态规划算法，也有 O(nlogn)的基于二分搜索的算法。但原题 n<=1000，就没有必要使用较为复杂且调试困难、容易出错的 log 级算法。所以，盲目优化时间复杂度不是可取的行为。运用这些时间，可以解决其他问题，得到更高的分数。我们的目的，就是找到“时

时间复杂度表示式中隐藏的常数因子不可忽略，尤其是对于搜索类题目。

[1] 编程复杂度，就是一个程序编写所需的时间，而不仅仅是程序的长度。例如一个构思精巧的 100 行程序，可能比一个直接敲上去的 200 行平衡树代码更费时间。

[2] 可解问题，与不可解问题相对。例如著名的停机问题(The Halting Problem) 就属于不可解问题，它们不可能找到一种能在有限时间中得出结论的算法。本文只讨论可解问题。

[3] NPC 问题，指没有多项式时间算法的问题，例如著名的“旅行商问题”(TSP)，只能通过枚举搜索来解决。注意与 NP、P 问题搞清关系。但良好的可行性剪枝、最优性剪枝可以大大提高运行效率。

[4] 这种精益求精的观念在软件编写中确实重要，例如我们常用的记事本、Explorer 软件、各种操作系统的内核，都是经过精心优化的。但在信息学竞赛中，从应试角度出发，应该适度进行复杂度优化。

[5] 用空间换时间，从本质上说，就是采用记忆化的方式，将已经做过的事记录下来，尽量少进行重复计算。

[6] 注意不要真的卡在 990ms 上，为了保险起见，而要留出一定的空余时间（不少于 0.1s），因为实际运行时间与机器的运行状态有关。空间复杂度的处理也要注意留出余地。

[7] 本文中的 log 都略去了角标 2，未加说明时默认为以 2 为底的对数。

[8] 当然，对于高手也要追求“保险”，尽量提高成绩的稳定性，避免无谓的丢分。

间复杂度”与“编程复杂度”的**平衡点**[1]。

2 参赛准备

2.1 心态——成就考试的前提[2]

综观近几年学生的高考等应试情况，有这么一种说法：“考试成绩发挥好不好，就看其心理素质好不好”。这种说法虽有失偏颇，但也不无道理。王极盛教授通过对考入北京大学的 51 个高考状元调查得出结果：在影响高考成绩的 20 个因素中，学习方法的重要性居第 3 位，学习基础的重要性居第 4 位，而考场心态的重要性居第 1 位，考前心态的重要性居第 2 位。可见，学生在高考前和考试中的心态居首要位置，学习策略、技巧和知识基础紧随其后。郑日昌教授认为，影响学生考试发挥的因素主要有知识因素、应试技巧和考试焦虑(心态)。叶平枝等人研究发现，**高考落榜生和高考佼佼者的最大区别不是智力，而是心理素质**。考试心理素质主要涉及考生的信心、情绪状态、焦虑水平和心态。我们认为，决定高考、竞赛成绩的因素为在勤奋和认知结构良好的条件下，提高成绩的关键在于考前和考试中的心理状态及学习策略和考试技巧水平。后几种因素水平高，能使考生提高 50-100 分，反之会降低 50-100 分，乃至考试失败。

根据学生参加考试的实际情况，考生心理状态存在以下四种：第一种，考前盲目自信状态。表现为外表看起来很兴奋，有时也好像很沉着，但对考试的困难和复杂性估计不足，相信能轻易取得成绩。第二种，考前过分紧张状态。表现为对考试兴奋过度，焦急不安等。第三种，考前信心不足状态。表现为情绪低落，反应迟钝，缺乏信心。第四种，战斗竞技状态。表现为有良好的考试态度，并力争出好成绩。

1、考前心理准备

(1)动机适中，目标适宜，强化自信。

(2)克服考试焦虑，优化情绪状态。

(3)讲究学习策略，构建良好的认知结构。

2、积累应试经验，提高应试技巧

※仔细观察，迅速把握试卷全貌，认真审题

※善于试题类化，准确提取知识

※答题简明扼要，切忌画蛇添足

※不仅求快，更要求准

※利用第一印象，卷面可以得分

※能走就走一步，争取多得一分

※只要不倒扣分，不妨尝试回答

※重视复查环节，把好最后一关

※充分利用时间，不必提前交卷

※一旦走出考场，迅速转移注意

事实上，不仅是高考需要良好的心理状态，竞赛同样如此。笔者所在班级有很多同学有过类似的经历，学习好的反而考砸，学习差的反而考好。读者应该也有类似的感受。事实证

[1] 有兴趣的读者可以试着使用博弈论中的“纳什均衡理论”解释一下。

[2] 本节内容自教育部阳光高考平台《阳光高考电子读本》改编。

明，心理因素在竞赛中的影响不容忽视。培养良好的心态，是“骗分”的心理基础。

对于信息学竞赛的特殊性，我们还可以提出，遇到难题不要慌，要沉着冷静，先使用**数学分析**[1]手段，若不可行则采用骗分手段，能得几分算几分，哪怕使用**朴素算法**[2]或**错误的贪心算法**[3]，也不要让 100 分白白流走。

良好的心态，加上合理的应试技巧，就能提高“得分能力”，达到超常发挥的目的。

2.2 知己知彼，百战不殆[4]

俗话说，知己知彼，百战不殆。那么，我们要在信息学竞赛中取得良好的成绩，就要首先了解“出题人的意图”，也就是命题人是如何命制一套信息学竞赛试题的。

2.2.1 单题的命制

首先，命题人需要命制一道“单题”，也就是在不考虑整体结构的情况下的单独试题。衡量一道试题，通常有以下几个指标：创新性、趣味性、正确性、严密性、区分度、合理性。

在这六大指标中，趣味性是一般不需达到的非重要要求，试题中一般会设置一个有趣的情景，这不仅是为了有趣，更是为了考察选手将实际问题抽象成数学模型的能力、读题审题的能力。在模拟竞赛中，往往设计的背景为游戏、科幻等，更为有趣，同时也带来了理解上的困难[5]，因此趣味性应该适度。

创新性是大型比赛中一般具有的特点。众所周知，正规竞赛不能出陈题、旧题，就是对创新性的要求。可惜这一点在模拟竞赛中往往没有做到，需要进一步努力。

正确性是任何一道试题必不可少的条件。最低要求是，这个问题存在正确答案，就是问题是“可解问题”，并且标准答案唯一[6]，被大家公认。进一步要求，正确性是可以在有限时间内验证的。例如“长生不老药”，就是无法在有限时间内验证的[7]。在信息学竞赛中，往往还要求验证能在多项式时间内完成，即问题为一个 NP 问题。例如要求**输出**[8]一个程序，只利用 if 语句完成 n 个数的排序，这道题严格上就不是很好的。要证明程序的正确性，必须对每种可能的 n 个数的大小关系都进行测试，需要 2^n 的时间，这是评测时间所不允许的。[9]况且，这道题有规模庞大的输出，I/O 操作所占的时间占据了程序运行时间的大部分，这不符合命题的主要目的。

正确性的**底线**包括，题目描述（文字、图表、举例）正确无歧义；输入数据符合要求；

---

[1] 信息学竞赛中的数学分析通常不需要严格的证明，只要观察出“看起来正确”又举不出反例的特点即可。主要应用于贪心算法，也应用于动态规划、搜索的优化。

[2] 朴素算法，相对于标准算法而言，就是较为简单的、时间效率较低的算法，通常包括模拟、枚举、搜索、局部贪心、局部动态规划等，但都是正确的算法。

[3] 错误的贪心算法，就是有反例、不保证正确的算法，但应用时必须保证较高的正确率，只有少数特殊的反例才可以。这样的算法包括“传染病传播”一题中每次选择度最大的点、多重背包问题的非单调队列 $O(n^2)$算法。若对于某种普遍发生的情况均未考虑，则得分一般不超过 30 分，例如 NOIP2008 第四题。

[4] 本节参考了刘汝佳《纵论命题与参赛》。模拟赛、正式比赛的命题人也可参考这些建议。

[5] 很多模拟试题被称为“语文题”，题目逻辑不严密，语言晦涩难懂，甚至出现语病，都是不应该的。

[6] 要求给出一组解并按解的优劣给出部分分数的试题除外。

[7] 如果试验者死亡，证明药无效；但没有死亡，就始终不能判断药是否有效。也就是说，只能判断出假的长生不老药，永远不能确定真的长生不老药。

[8] 注意不是“编写”，而是编写一个“能编写程序的程序”。

[9] 同样的，我们编写的程序 AC 不能表示程序正确，因为尚未通过所有可能性的检测。但在通常条件下，对所有可能性的检测是不现实的，因此只能得到“相对正确”的程序。

输出数据和评分程序（Special Judge）正确；如有交互库，库的行为符合题目规定；下发的题目附件（数据文件、辅助工具）正确。事实上还有标程和时限，但这些要求往往不能实现。例如 NOI 的试题，往往不能提供一个**正确的[1]标程**，模拟赛更是如此。

严密性，一是指题目本身描述严密、逻辑清晰，不会引起歧义，不会发生“题目没有说明这种情况，该如何处理”的争议。尽量要求题目清晰易懂，而不是“玩语言游戏”。另外，在交互题中，交互库所占的资源；在输入输出庞大的题目中，I/O 操作所占的时间，都应该有所控制，正如上节中的“输出排序程序”，以及“输入与运算复杂度同级[2]”的题目，都应该尽量避免。在正规的比赛中，**防止作弊**[3]也是重要的要求，尤其是交互库，更要严防作弊。

区分度，就是指一个试题在区分不同选手时的作用。区分度可以有多种指向，例如**算法、程序实现、其他努力**等。例如 A 程序编写了 200 行的高级算法但效率偏低，B 程序编写了 100 行的数学方法而效率很高。显然，B 程序得分不少于 A 程序的得分。那么，作为命题人，是应该尽量拉大还是尽量缩小 AB 两程序的得分呢？笔者认为，尽管 B 的方法更好，但高级算法同样是一个优秀选手应该掌握的，况且数学算法的构造带有一定偶然性，所以不应过分贬低。对于命题人来说，应当事先考虑到一道试题可能具有的若干不同算法，并决定给每种算法不同的分值，再根据这个分类去设计测试数据，考察选手的不同方面能力。

显然，区分度应该尽量细化，例如 IOI 就有多达 40 组测试数据，而国内竞赛却只有 10 组数据。笔者认为，**增加测试数据的组数**，可以更好地区分不同层次的选手，但这同时给命题人带来了困难。命题人需要考虑各种不同情况，并更加精心的生成每一组数据，而不是随机生成或全是极限数据，否则测试数据组数的增加就没有了意义。除了增加测试数据的组数，**多个小问部分分、优劣程度部分分、超时程度部分分**都是增强区分度的有效方式。

多个小问部分分，就是像数学、物理的解答题一样，一道题有多个任务，完成每个任务给一定的分数。各个任务之间可以互相独立，也可以有逻辑关系，应该分开难度梯度。这样某个选手在一个小问题上卡住，可能得到其他已解决部分的分数，使得“黑箱测试”更加合理，更加开放，更能反映选手的真实水平。WC2009 中第二题就是一个不错的例子。

优劣程度部分分，就是要求输出一组解，根据解的优劣评判得分。这样可以使信息学竞赛编程更加接近真实应用的情况：不是要求“最优”，而是要求“尽可能优”。尤其是对于难题，这样的方式更能体现算法的优劣。例如 WC2009 的提交答案试题[4]采用了这种方式。

超时程度部分分，就是按程序运行所需的时间评分。例如，运行时间超过 10s 的不得分，5s-10s 的得 2 分，2s-5s 的得 4 分，1s-2s 的得 7 分，1s 以内的得 10 分。这样的方式适合搜索类试题，或有多种不同复杂度算法的试题。目前这种方式尚未被正式比赛采用，但这样的方式能够使选手更加珍惜程序的运行时间，在程序的常数优化方面下更多的功夫，也就使信息学竞赛更贴近实际应用的“优化，优化，再优化”要求。

最后，就是有关合理性的问题。合理性，就是试题要“厚道”，不要引人误入歧途。样例数据就是重要的一种方式。笔者建议，每道试题至少三组样例测试数据，就像 NOIP2008 一样，并对其中有代表性的一组进行解释。三组样例数据尽量包括：特殊情况、一般小范围情况、容易忽略的情况。当然，如果命题人的目的就是考察选手的认真细致能力，例如字符串处理、复杂的模拟、分类讨论试题，可以另当别论。一般来说，样例数据可以启发选手的思路[5]，可以排除低级错误，也可以误导选手。究竟采用哪一种方法，笔者认为对于主要考

[1] 严格的说，正确性不能针对几组给定的测试数据，而要对所有满足题目要求的测试数据在规定时限内出解，包括极限数据。但一些难题，尤其是提交答案试题的标程通常做不到这一点。所以经常出现标程无法解出选手给出的测试数据的情况。

[2] 例如输入 n 个整数，求其中的最大数。输入输出复杂度与运算复杂度接近。

[3] NOI 采用 Linux 操作系统，可能就是出于稳定性、安全性的角度考虑。

[4] 题目本身很简单：给出含有一些布尔变量的逻辑表达式，求出一组变量的取值，满足尽可能多的表达式。

[5] 如构造类问题。例如排比赛日程表问题，8*8 的样例足以发现其中的分治对称规律。

察高级算法的题目，可以减少低级错误；而主要考察细心、编程能力的试题，可以留给选手处理，甚至做适当的误导。

2.2.2 测试数据的命制

如果说一道试题是一款电脑游戏，那么试题的原型就是游戏的内核控制系统、基本设计理念，而测试数据则是闯关系统、游戏中的地图与角色。显然，游戏的内核编写完毕后，可以拿出来玩一玩 demo 版，但是不能面向社会发布。很多人认为这完成了游戏设计的前 90%，于是称为“前 90%”。但是，一旦深入游戏的角色情节设计、闯关系统，要保持游戏者的积极性，带有一定的趣味性，就要认真完成这项艰巨的任务。通常来说，后一部分的开发时间要占整个游戏的 90%，于是称为“后 90%”。前后两个 90%加起来并不等于 100%，这是因为前 90%是心理上的，后 90%才是实际上的。

命题何尝不是如此。好的测试数据，往往需要比命原题更多的时间。首先需要编一个尽可能优化的标准程序，并用各种极限数据测试。然后制作数据生成器，构造具有代表性的数据。最后用标准程序计算，做出正确的标准输出。其中的第二步是最费时间、精力的一步，也正是很多命题人容易偷懒忽略的一步。

笔者认为，一个正规的试题，应该具有一个题目包，包含以下部分：

①试题正文和相关文件（如 LaTeX 源码）；

②选手能拿到的文件（包括辅助工具、提交答案题的输入数据，交互库、传统题目的样例数据）；

③评测所用的文件，包括测试数据、交互库、评测插件、辅助工具；

④模拟测试用的代码（参考程序、非完美程序、测试评测插件鲁棒性的程序、交叉验证程序）；

⑤其他文件，包括随机数据生成器、数据分析/可视化工具、所有程序源代码、命题说明等。建议制作手工数据生成器（尤其是几何题），并在参考程序中验证数据合法性。

在现今的竞赛考试中，通常具有、对外公布的是①②③，但④⑤容易在命题过程中忽略。为了保证试题的严密性，避免出现纠纷，就应该保证题目包的完整性。

命制测试数据，需要关注区分度的问题，已经在上节中讨论；对于试题得分的分布，最好服从“**正态分布**[1]”，这一点也是其他学科竞赛中容易满足的；唯独信息学竞赛，难题普遍分数高甚至满分，简单题普遍分数低甚至零分，中档题满分、零分人数都多，但共同特点是分数居中的很少。这种“逆正态分布”的情况正说明了目前命制测试数据的不合理性。

我们注意到，形成正态分布的条件是“**很多独立随机事件**”，要求事件多、相互独立。在其他学科竞赛中，例如数学竞赛，试题的数量较多，例如现在的高中数学联赛有 6 道选择题、6 道填空题、3 道难度中等解答题、3 道难度较高解答题，其中解答题还有分步给分的规定。这使得事件的数量增多，得分接近正态分布。但仔细观察数学竞赛的得分曲线，仍然出现了低分过度聚集、高分明显偏少的现象，这是 3 道高分值难题作用的结果，也是数学竞赛选拔高水平人才的需要，不能盲目追求正态分布。但我们再来观察高考的“一分一档统计表”，发现几乎就是正态分布。在高考中，科目多，题量大，分值零碎，所以分布更加正常。那么信息学竞赛中偏离正态分布，甚至出现“逆正态分布”的原因，也就不难解释了：**题目少**，NOIP 只有 4 道，NOI 只有 6 道，WC 只有 3 道；**分值大**，不是指一个测试点分值大，而是一道题如果做出来可能满分，没有做出来可能零分，几乎没有中间档次；**关联强**，信息

[1] 正态分布，就是很多独立随机事件中发生次数的分布曲线，呈钟形，中间大两边小，指数级变化迅速。

学竞赛中的算法不多，一个部分不会可能导致满盘皆输。这些不符合“事件多，相互独立”的要求，当然不能形成正态分布。

针对信息学竞赛的特点，我们要尽量改变现状，就要**增大测试数据的颗粒度**，增加测试数据组数，考察更多的方面，一是针对更多的特殊情况、边界情况展开测试，减少“漏网之鱼”；二是增强测试数据的层次性，各种数据规模的数据都要有，并合理安排比例，考虑到各种可能的算法。**降低试题之间的关联度**，一是每道试题有针对性的考察特定的算法，尽量使知识点相对分散，除了最后要求综合实力的难题；二是采用上节中“部分分”的做法增大不同程度选手的得分概率，使得各种算法均有其用武之地，避免“一棒子打死”。

例如 WC2009 提交答案试题的测试数据，笔者认为命制得较为科学。该题作为提交答案试题，考察了选手观察数据特点，并有针对性的运用多种算法解决不同数据的能力。可以称为是测试数据命制科学化的一个典范。

测试数据的命制，是一门很有艺术的科学，绝不是简单的“打开随机数据生成器，做几个输入输出”，需要各位命题人仔细揣摩。

2.2.3 套题的命制

一套优质的信息学竞赛试题，不应该只是几道单题的堆积，而要使得各题目之间搭配合理，达到选拔人才的目的。

假设有一道题 X，当 X 和两道简单题搭配时，设平均分为 A；当 X 和两道中档题搭配时，设平均分为 B；当 X 和两道难题搭配时，设平均分为 C。讨论：A、B 和 C 的大小关系？

为了解决这个问题，我们需要考虑到考试时间、心理状态、考场决策对试题得分的影响。如果 X 是一道简单题，显然得分率都会很高，因为都会选择简单题先做，这也是各大竞赛中水题满分成堆的原因了。

如果 X 是一道中等题，当与两道简单题搭配时，由于简单题很快做完，得分率较高；若与难题难题搭配，由于相对简单，得分率也较高；相对而言，与中档题搭配时得分率较低，由于可能因为前两道题而耽误了时间。

如果 X 是一道难题，得分率总是较低的；但相对而言，与简单题搭配时由于剩余时间充裕，得分较高；与中档题搭配，剩余时间很少，几乎没有人做，得分最低；与难题搭配，得分率一般，也是存在解题的先后问题。

以上的讨论中，都忽略了心态对于考试的影响[1]。如果第一题是难题，就会严重挫败考生的信心，还会浪费大量的时间。正规的竞赛中，试题由易至难排列，就是尊重了考生的解题习惯，先保证较为简单的题得分，再花费心思解决难题，以获得最高的“分数时间比”。

当然，正规的竞赛中，一般来说简单题、中档题、难题会各有一道，例如 NOIP2008 就有一道简单题、两道中档题、一道难题，分布较为合理。这也使得 NOIP2008 的分数避免了高分稀少、低分扎堆的情况，出现了近似的正态分布曲线，是竞赛改革的有力步伐。

选拔性的竞赛，要考察选手对于多方面知识、能力、素质的掌握情况。包括：

抽象思维、形象思维。适合几何类问题。

大胆猜想、找规律。适合数学类问题。

建立和求解数学模型。适合实际应用类问题。

运用常见分析手段、实现经典算法及其变形。适合算法考察类试题。

探索陌生领域、深入分析问题。适合贪心、搜索、提交答案类问题。

[1] 例如 WC2009 中的多位大牛，考出了偏低的成绩，就是因为长时间卡在第一题上，打乱了解题步伐。

编写、调试复杂的程序。适合模拟、字符串处理、逻辑推理问题。

小心处理各种特殊情况。适合模拟、字符串处理、逻辑推理问题。

评估、管理风险和收益的能力[1]。

显然，一套好的竞赛试题，应该对多个方面进行考察，而不是“只出图论题”“只出数学题”。那么如何搭配这些方面，是针对某一方面单独考察，还是考察选手综合运用知识的能力，甚至考察选手现场分析问题、创造算法的能力，都是命题人需要考虑的问题。

在近几年的竞赛中，考察选手“创新意识”的试题越来越多，即一道试题不能使用一个经典的现成算法来解决，而需要选手根据问题的需要，构造合适的数据结构，创造合适的算法解决问题。这种试题能有效的避免选手“押题猜宝”、考前“临阵磨枪”背诵算法代码，或者做几道经典题“以不变应万变”；能选拔出真正具有分析、思考、编程能力的人才。

在实际命制套题的过程中，还需要考虑“非智力因素”的影响。

心理：题目顺序和题面的复杂程度是否会影响选手发挥？应如何安排题目顺序，以及调整题目的复杂程度？笔者认为，从“不给选手制造麻烦”的角度考虑，应该做到有较明显的难易梯度，并从易到难排列。

总体策略：第一试和第二试。指导思想和风格（保守、激进）。命题者是否应当把选手的这些心理活动作为搭配的依据？笔者认为，应该尽量稳定选手的情绪，减少因心态引起的失误。

不确定性：是否是一种合理的风险来源？例如，不给数据规模的具体分布，将造成得分风险。这样的风险是合理的么？笔者认为，在普及类型的竞赛（如 NOIP）中，应该尽量提高试题的确定性，以保证多数人能得分；在选拔类型的竞赛（如 NOI、CTSC）中，应该使试题具有一定的不确定性，考察选手分析、处理风险的能力，也更接近实际应用中的情况。

我们相信，随着竞赛改革的深入，套题的命制、试题的分布会越来越合理。

2.3 应该学习的内容[2]

1、 NOIP 之前的知识
2、 高等图论
    |->网络流
        |->最大流
        |->最大流最小割切定理
        |->最小费用最大流
        |->容量有上下界的最大流最小流
        |->容量有上下界的最小费用最大流*
    |->二分图
    |->连通图、最大独立集、最大支配集
    |->
3、 树
    |->平衡树
        |->SBT
        |->AVL

[1] 将在本文的后面章节讨论。
[2] 引用自 2009 河北省代表队训练计划。

　　|->红黑树
　|->线段树（树状数组）
　|->字母树 Trie
　|->RMQ
　|->堆
　|->区间树
　|->
4、其它
　|->跳表
　|->后缀数组
　|->并查集
　|->矩形分割
　|->离散化
　|->拟阵*
5、动态规划
　|->状态压缩
　|->树型 DP
　|->四边形不等式优化
　|->单调队列，双端队列优化
　|->多做题
6、数学相关
　|->线性规划、线性代数
　|->递推
　|->数值数论
　|->组合数学
　|->离散数学
　|->几何
　|->概率学
　|->数学建模、抽象化思维
　|->数学归纳法
　|->矩阵、行列式
　|->高斯消元
7、搜索
　|->双向、反向
　|->A*搜索，IDA*搜索
　|->简单博弈问题
　|->减枝
　|->分支定界、差分约束
8、计算几何
　|->基本东西
　|->凸包
　|->旋转卡壳
　|->线段相交、扫除法
　|->随机增量

|->求多边形面积

2.4 找准位置，水滴石穿[1]

考试时的方法多数不是“灵机一动”现场创造出来的，而是平时刻苦训练中积累出来的。俗话说得好，“水滴石穿”，但“水滴未必石穿”，如果水根本没有滴到石头上，或者恰好滴到了最坚硬的位置，即使费劲九牛二虎之力，也未必能够“穿石”。所以，只有“找准位置”，再加上时间的积累，才能达到“水滴石穿”的效果。

信息学竞赛平时的训练也是如此，如何在有限的训练时间内达到更好的效果，就是本文与读者共同探索的方向。不仅是在考试时要讲究应试策略，平时的训练也要讲究方法，讲究技巧，才能“事半功倍”，收到较好的效果。

书籍推荐：绿书+蓝书（青少年信息学奥林匹克竞赛培训教材 系列）、算法导论（也是黑书）、粉书（全国青少年信息学奥林匹克联赛培训教材）、黑书（算法艺术与信息学竞赛 建议水平高的人看）、新编实用算法（建议水平高的人看）

建议参加 NOIP 的选手将绿书、蓝书、粉书全部看完，再看一些算法导论中的基础内容。参加 NOI 的选手应将算法导论看完，算法艺术中不是特别难的题目看完。

题库推荐：PKU（顶级推荐，题目质量高，题库影响力大，不仅学信息学还能学英语）、Usaco（强烈推荐，很经典的题库，Nocow 有翻译）、Vijos（一个非常不错的中文题库）、Ural（水平有所提高了可以去做做，也有很多经典题）、Rqnoj（题目质量不是太好，不过也是一个题库）

建议参加 NOIP 的选手以做中文题库为主，由于当前命题趋势越来越简单，所以 Vijos、Rqnoj 都是不错的选择。笔者 NOIP 前，在 RQNOJ 上通过了 286 题，位居第一名[2]，最终在 NOIP2008 中取得了 330 这个不错的成绩。练好简单题，算法一眼看出，程序直接敲上，保证一次 AC，也是一种难以达到的水平，代表着选手对编程语言的熟悉程度，在今天的 NOIP 中有着得天独厚的优势。尤其提醒以“拿一等奖”为目的的选手，不要盲目钻牛角尖、死抠难题，这样浪费了时间，因为 NOIP 并不考这样难的题目。

建议参加 NOI 的选手做一些难题，如 PKU 就是不错的选择。以 NOI 为努力目标的 NOIP 选手，在备战时诚然应该做一些难题，为将来打基础，但更重要的是练好基础，保证进入 NOI。笔者所在的竞赛小组就有几位难题实力很强的选手因为考试时紧张，出现了低级错误，未能进入省队，令人惋惜。NOIP 结束，才可以抛开水题，全力攻坚。因为信息学竞赛不同于其他学科竞赛，省赛、全国赛之间有很大的时间差，这时猛攻难题是完全来得及的。

知识来源：40%来自于书本、40%来自于做题、20%来自于网上其它的资料。

Cai0715 的 OI 技巧：

1、**学会总结**，我基本上每学完一个章节的东西就会拿多一段时间来进行总结。总结的时候，先列出一个目录，把这一节学到的知识点写进去。然后在下面分别拓展出这个知识点的原理、用途、编程流程、关键代码、优化、和其它同类算法的比较、复杂度估计、模块化代码、相应习题等等。这样，可以很容易的把一个一个的知识点串在一起记住它。（附录里有我的总结样例）

[1] 本节主要参考了保定二中 Cai0715 《我的 OI 技巧，送给广大信息学初学者》。
查看原文，请访问 NOI2009 河北省队训练基地 http://oier.5d6d.com/

[2] http://www.rqnoj.cn/RankList.asp

2、**要拿出一定的时间看书**，书是人类进步的阶梯。相信很多 OIer 都很喜欢做题，忽略了看书这个重要的环节。其实只有阅览群书，才能学到更多的知识。例如，某个算法你会 N^3 的算法，但是某本书上用 N^2，甚至 N 的复杂度就给解决了。如果不看书，如果考试正好出这个知识点，或许你就只能拿部分分了。

3、**合理安排时间**（特指放假或集训的时候）。清晨 7 点到 9 点这段时间，是人们头脑最清醒的时间，这个时间段内，尽量不要去调程序，可以去做一些其它的事情，比如看书、总结等等。而 9 点之后这段时间，建议去做题，这个时候一般花一个小时就可以编出在其它时间要花一个半小时才能编出的程序。到了中午，吃完饭一定要睡觉，否则下午会很没精神，效率会很低的。下午呢，一般就比较综合了，比较随意了，可以自己安排。晚上，我一般都是继续调程序，因为想不出晚上干什么会高效一些。

4、**学会适当的休息**，不要长时间干同一件事情。当你编程序进入了一个死角的时候，或许思维会很混乱，总想找出错误或优化这个算法，但是一片空白，很盲目。这个时候，你真的需要休息。去外面小小的溜达一圈，看一看窗外的风景，这样就可以换一个心情，换一个思维。在你休息完之后，你会发现，刚才你之所以找不出错误，是因为身在此山中。

5、**养成编代码的良好习惯**，这个各位可以参考其它大牛的程序。我的程序一般都是用过程堆起来的，每个程序必有的 Init 和 Main 过程，可能 Init 里只有个 read(n)，但是为了保持程序美观，完整个人的习惯，还是单写了一个过程。但是不要大量的调用过程，比如在某个 3 重循环里调用过程，因为调过程也是需要一定时间的。除了这个，我还在每个过程中间添加了一个分割线，以便阅读。还有，各位还需要注意换行、空格等问题，养成良好的习惯，尽量使其美观，方便阅读（附录里有代码样例）。

6、**学会心理暗示**。这个也是很重要的，当你做不出某个题的时候，一定不要乱，心理默默的暗示自己，既然自己不会，别人做起来一定也不会舒服。当你做出某个题，一定不要盲目的高兴，要把自己的思维控制住，这样才能用形象的思维去做下一个题，所以我们一定要暗示自己，这个题自己会做，别人做起来也会很容易的，不能骄傲。

7、**学会抗干扰**，干扰有很多类，大概就是人为、自然因素。人为因素，当别人早早的做完题或者别人在说话、讨论的时候，一定要控制住自己，不要慌乱，否则你可能会编的程序最后得个 0 分，所以一定要在众多次干扰中，积累抗干扰的经验。自然因素，当太阳直射你的时候、当寒风呼啸你的时候、当键盘生硬难敲的时候、当屏幕反光的时候，你一定要学会去适应，因为很多时候在考场上会出现这样或者那样的问题，给你的只有 3 个小时，没有多余的时间去考虑这些无关的问题，只能适应。综上所述，抗干扰很重要，即使没有干扰，我们也可以为自己去制造干扰。

8、**写程序的流程要合理安排**，这个很重要，也是非常重要的。我就把自己是怎样做的写出来吧，可能不是太好，只是一个借鉴。1）我会用大概 5 分钟左右去完整的阅读题目，因为多一点时间阅读题目，总会有意想不到的发现。2）用 10-15 分钟的时间去设计算法，要尽量躲避第一印象思路，因为这个思路往往是错的，设计算法不仅要证明这个算法的正确性，还要从时间、空间等因素来考虑是否，千万不要很草率的结束这个过程，因为当编完程序再来改正错误的算法，往往会浪费更多的时间，例如说高一的我，看见题就想做，大概理出了思路就去编程，但是反过头来发现，其实是错的，结果浪费了时间不说，心情还很不好。3）利用 5 分钟的时间写出程序的框架，第一步该干什么，第二步又该干什么，一步一步的写出来，再对每一步进行一些拓展，写出关键的伪代码等等。这样，才能让自己在编程的时候条理性清晰，才能降低出错的几率。4）编程 10-20 分钟，前面的工作都做好了，这个过程应该是非常容易的，注意不要犯打错变量等低级错误就行了。5）查错 10-20 分钟，往往第一次编出的程序都是错的，具体的查错技巧下面会写出来。这样，一个程序就算写完了，我这个算法流程只是提供一个参考，具体每个流程的时间大家可以看情况去安排，一般简单

程序 30 分钟敲完，中等点的 50 分钟，难点的 1 个小时左右。这样才可以更上 NOIP3 个小时的节奏。

9、**静态查错**。这个是很重要的，也可以说是非常重要的。何谓静态查错，就是编完代码之后，不去干其它事情，只是安静的从头到尾的把自己的代码阅读一遍，比如说普通的编译错误、变量是不是打错了、数组开的够不够大、程序的逻辑性是不是还存在问题等等。这个时候，一般是很容易发现错误的，并且还会有一种成就感。但是如果你编完之后去测样例，可能样例是过了（因为样例是很弱的数据），但是其实程序仍漏洞百出，或者测样例都错了，这个时候会严重影响你的心情，再去查错的话，事倍功半。

10、**出测试数据**是个大学问。测试数据一般分为，小数据、大数据、极限数据等等。所以我们一定要从这几个方面，各出几组测试数据。小数据可以手算，很容易出结果，相信是 OIer 最喜欢的。极限数据也是指那些边缘数据，比如说某个数据导致你数组越界、被 0 除等等，一般很多题目都存在一两组这样的数据。大数据的话，一般是去检验程序是否超时间和超空间，因为结果是否正确，真的很难手算出来，除非很离谱的错误。

11、**检验程序的正确性**，这个除了设计算法时的证明外，如果有时间允许，我们还可以写一个效率低但是绝对正确的算法来和原程序进行对比。这样，我们利用上面出测试数据的学问，加上这个手段，一般可以 80%的判断出你写的这个程序是否正确，进而不断完善。（还有一个小技巧，就是利用.bat 文件，判断两个输出是否等价。）

12、**善于交流**，这个就不知道怎么说了，就是多和其它人交流。

其它：1）培养对 OI 的兴趣。2）胜不骄，败不馁。3）不抛弃，才有希望；不放弃，才能成功。4）天才是 99%的汗水+1%的灵感，对学 OI 尤其适用。5）学完某个算法时，尽量把模块化代码保存下来。6）如果 NOIP 一等奖名额只有 1 个，那么相信自己就是那一个。7）制定好目标，分为远大的目标（上哪个大学），中期的目标（拿一等、冲省队、夺金牌），近期的目标（学会哪个算法、模拟赛要达到多少分）。8）赛前一周多复习，多做简单题，不要再去钻高难度的题，这是 NOIP 不是 NOI。9）专心致志，学习时就不上 QQ，不上论坛。10）细心，细心，再细心。

以上是 Cai0715 的 OI 技巧，笔者也十分认同。正像一句歌词“想唱就唱，要唱得漂亮，总有一天能见到挥舞的荧光棒”，我们相信只要找准方向，努力奋斗，总有辉煌的一天。

2.5 临阵磨枪，不快也光

在赛前两周的准备时间中，老师往往教导我们，不要再做难题，要做一些模拟题，“信心题”，并注意练熟基本算法，不要出低级错误。

在正式比赛之前，应该熟悉比赛所用的操作系统、编译器，尤其是编译环境，如果没有在考前进行充分的练习，竞赛时就会慌神，不知道该如何使用这些工具，甚至不知道一些有用的功能，还会造成一种陌生感，影响考试心情。

很多选手认为比赛所用的编程环境并不好用，这是由于我们经常不用而不太习惯。在 Linux 的标准配置下，都有 gedit 源代码编辑工具、gcc 编译程序，这样使用记事本和命令行完全可以解决问题，根本没有必要使用那些编译器、调试器。所以在平时练习时，尤其是赛前突击训练时，应该尽量使用比赛规定的编程环境。

分类	软件	版本	说明
系统软件	NOI Linux	内核 - 2.6.24 NOILinux – 1.2	操作系统
编译器	Gcc	4.1.2	C 编译器

	G++	4.1.2	C++编译器
	Freepascal	2.0.4	Pascal 编译器
调试器	Gdb	6.6	命令行启动
	Ddd	3.3.11	命令行启动
集成开发环境	GUIDE	1.0.0	单文件程序 IDE（C/C++/Pascal）
	Anjuta	1.2.4	C/C++ IDE
	Lazarus	0.9.22	Pascal IDE

2.6 玩自己的养成游戏

不知读者是否玩过称为“养成游戏”的一类计算机游戏，养成游戏就是给出一个人或动物，要求玩家利用现有的金钱、道具等，将这个生物培养成希望达到的目标。例如，让他“进行锻炼”，就能增长体力值；让他“上一小时辅导班”，就能提高学习成绩，等等。当然，现实生活中远不止这么简单，“努力未必成功，成功必须努力”，我们不能完全仿照虚拟世界中的简单做法处理实际问题，但“玩自己的养成游戏”，可以成为人生进取的观念。

我们学习信息学竞赛，事实上是在养成一种学习方法，养成一种思考问题的习惯，使我们在解决其他方面的问题时有章可循。

3 复杂度常数优化[1]

本文已经指出，“骗分”就是与“复杂度”的较量，而在信息学竞赛中，时间复杂度又是最重要的，所以我们先从时间复杂度入手，领略“骗分”聚沙成塔的威力。

3.1 时间复杂度常数优化的意义

在科学研究意义上，时间复杂度的常数优化并不是十分重要的[2]。但在信息学竞赛中，同样的复杂度为 $O(n^2)$的程序，对于一组 n=5000 的数据，有的可能常数为 20，需要运行 1000ms，有的可能常数为 5，需要运行 500ms。这样，两个看似相同的算法，一个超时错误，一个正确得分。所以，很多同学有“常数优化不重要，关键在算法[3]”的看法，是不正确的。众所周知，好算法的设计不是人人都能完成，而常数优化的技巧可以在平时积累，注意训练，考试时就能习惯的写出高效的代码。

在信息学竞赛中，常遇到程序运行超时的情况。然而，同一个程序设计思想，用不同算法，会有不同的运行效率；而即使是同样的算法，由于在代码的细节方面设计有所不同，

[1] 本节前几个有关“运算速度”的标题改编自笔者 NOIP2008 前夕的文章《算法效率与程序优化》。
[2] 例如陈启峰的 SBT 并未引起学术界重视，是由于时空复杂度并未根本改变，常数优化意义不大。编程复杂度的降低适用于信息学竞赛，可以加快编程速度。
[3] 在骆可强《论程序底层优化的一些方法与技巧》发表后，某些参加培训同学表示不仅从未考虑过常数优化问题，还对其演讲不屑一顾，认为“常数优化太复杂，不实用”“好的算法胜过一切常数优化”。

执行起来效率也会有所不同。当遇到所需时间较长的问题时，一个常数级优化可能是 AC 的关键所在。下面通过**实际测试**[1]和**理论分析**[2]，探索**常数时间优化**[3]的方法策略。

3.2 基本运算的速度[4]

基本运行时间[5]，是指在准备计算的运算复杂度之外，只包括循环控制变量的加减与比较所消耗的时间。要从实际运行时间中减去基本运行时间，才是这种运算真正的运行时间，称为**净运行时间**。

```
#include<stdio.h>
main()
{ int i,j;
  double a,b,sum=0;
  for(j=0;j<20;j++)
  { //此处添加随机数产生
    a=clock();
    for(i=0;i<100000000;i++);//此处添加运算
    b=clock();
    printf("%lf\n",b-a);
    sum+=b-a;
  }
  printf("ans = %lf",sum/20.0);
  getch();
}
```

运行平均时间是：202.3ms。[6]

（1）赋值运算

净运行时间 0.8ms，与基本运行时间 202.3ms 相比，可忽略不计，故以后将赋值运算作为基本运行时间的一部分，不予考虑。

（2）加法运算

产生随机数

```
x=rand();
y=rand();
```

循环内加法：

```
t=x+y;
```

[1] 本试验所采用的环境是： CPU Celeron 3.06GHz，内存 248M，操作系统 Windows XP SP2，程序语言 C。编译环境 Dev-c++ 4.9.9.2。配置略好于 NOIP 标准测试机 CPU 2.0GHz。

[2] 这里的分析十分粗略，没有使用汇编语言和 CPU 内部指令分析，仅在高级语言的层面上分析。

[3] 当然，没有必要使用“汇编语言”进行优化，这样未免过于复杂。本文的讨论限于高级语言内部。

[4] 为了增强算法效率的计算准确性，我们采用重复试验 20 次取平均值的做法。每次试验运行 100000000 次。

[5] 这些定义都是本文作者自行给出的。

[6] 本文中的程序，都使用 C 语言。在比较效率时，只写出了改变的程序段。有时省略了变量声明、输入输出，完整的程序请读者自行补全。运行时间是实测结果，可能与理论值有出入。

下面的各种简单运算只是更改运算符即可，不再写出代码。

净运行时间 41.45ms，即：在 1s 的时限中，共可进行 (1000-202.3)/ 41.45 *10^8=1.9*10^9 次加法运算，即：只通过循环、加法和赋值的运算次数不超过 1.9*10^9.。

而应用+=运算，与普通加法时间几乎相同，所以+=只是一种方便书写的方法，没有实质效果。同样的，各种自运算并不能提高效率。

（3）减法运算

净运行时间 42.95ms，与加法运算基本相同。可见，在计算机内部实现中，把减法变成加法的求补码过程是较快的，而按位相加的过程占据了较多的时间，借用化学中的一句术语，可以称为整个运算的“**速控步**[1]”。当然，这个“速控步”的运行速度受计算机本身制约，我们无法优化。在下面的算法设计中，还会遇到整个算法的“速控步”，针对这类情况，我们要对占用时间最多的步骤进行细心优化，减少常数级运算次数。

（4）乘法运算

净运行时间 58.25ms，明显比加减法要慢，但不像某些人想象的那样慢，至少速度大于加减法的 1/2。所以在实际编程时，没有必要把三个或更多的加法变成乘法，其实不如元素乘常数来得快。不必“谈乘色变”，实际乘法作为 CPU 的一种基本运算，速度还是很快的。

以上四种运算速度都很快，比循环所需时间少很多，在普通的算法中，每个最内层循环约含有 4-5 个加、减、乘运算，故整个算法的运行时间约为循环本身所需时间的 2 倍。

（5）除法运算

净运行时间 1210.2ms，是四种常规运算中最慢的一种，耗时是加法的 28 倍，是乘法的 21.5 倍，确实如常人所说“慢几十倍”，一秒的时限内只能运行 8.26*10^7 次，不足一亿次，远大于循环时间。所以，在计算时应尽量避免除法，尤其是在对时间要求较高的内层循环，尽量不安排除法，如果整个循环中值不变，可以使用在循环外预处理并用一个变量记录，循环内再调用该变量的方法，可以大大提高程序运行效率。

（6）取模运算

净运行时间 1178.15ms，与除法运算速度几乎相当，都非常慢。所以，取模运算也要尽量减少。在大数运算而只要求求得 MOD N 的值的题目中，应尽量利用数据类型的最大允许范围，在保证不超过 MAXINT 的前提下，尽量少执行 MOD 运算。例如在循环数组、循环队列的应用中，取模运算必不可少，这时优化运算便十分重要。可利用计数足够一定次数后再统一 MOD，循环完后再 MOD，使用中间变量记录 MOD 结果等方法减少次数。

在**高精度计算**中，许多人喜欢边运算边整理移位，从而在内层循环中有除、模运算各一次，效率极低。应该利用 int 的数据范围，先将计算结果保存至当前位，在各位的运算结束后再统一整理。

以下是用**统一整理法**编写的高精度乘法函数[2]。

```
int a[10000]={0},b[10000]={0},c[10000]={0};
void mul()
{ int i,j,t;
  for(i=0;i<10000;i++)
   for(j=0;j<10000-i;j++)
    c[i+j]+=a[i]*b[j];
  for(i=0;i<9999;i++)
```

[1] 在一系列的连续反应中，若其中有一步的速率最慢，它控制了总反应的速率，使总反应的速率基本等于这最慢步的速率，则这最慢的一步反应称为速控步(rate controlling step)或决速步(rate determing step)。
在信息学竞赛中，“速控步”通常称为程序效率的“瓶颈”。为了体现个性，本文仍采用“速控步”的说法。
“速控步”是本文的重要概念，也是效率优化时抓住重点、有的放矢的重要理念，请读者仔细体会。

[2] 数据规模为 10000。

```
  { c[i+1]+=c[i]/10;
    c[i]%=10;
  }
}
```

以上函数运行后，平均用时 276.55ms。

以下是边运算边整理的程序。

```
void mul()
{ int i,j,t;
  for(i=0;i<10000;i++)
   for(j=0;j<10000-i;j++)
   { c[i+j+1]+=(c[i+j]+a[i]*b[j])/10;
     c[i+j]=(c[i+j]+a[i]*b[j])%10;
   }
}
```

以上函数运行后，平均用时 882.80ms。统一整理与边整理边移位相比，快了 3.2 倍，有明显优势。故尽量减少除法、取模运算的次数，是从常数级别降低时间复杂度的方法。

（7）大小比较

```
if(x>y) x=y;
```

净运行时间 39.1ms，与加法运算速度相当。故比较运算也属于较快的基本运算。

3.3 位运算的速度

（1） 左移、右移

```
x<<1; x>>1;
```

净运行时间无法测出，证明位运算速度极快。而使用自乘计算需要 64.1ms，自除运算需要 164.85ms，所以尽可能使用位运算代替乘除。

（2）逻辑运算

```
t=x|y; t=x^y; t=x&y;
```

净运行时间约 30ms，比加法运算（约 40ms）快较多，是因为全是按二进制位计算。但加减与位运算关系并不大，所以利用位运算主要是利用左右移位的高速度。

位运算的优化[1]

在计算机内部，一切数据都是以二进制进行储存，其支持的最基本的元操作应该是布尔逻辑运算，例如逻辑与(and)、逻辑或(or)、逻辑非(not)、逻辑亦或(xor)等等，这些运算的实现，几乎可以说是直接与电路的基本元件逻辑门相对应起来，是计算机天生所最擅长的操作，其执行速度无疑也是最快的。至于整数的四则运算等操作，是用众多逻辑门所搭建起来的，加减法电路的逻辑门数量比较少，同样可以在较短的时间内进行运算，但如果像除法一般逻辑复杂，电路也相应庞大，速度就很难以接受了。再到更复杂的浮点运算，速度也就更

[1] 引自骆可强《论程序底层优化的一些方法与技巧》。需要注意的是，此文一些基于汇编语言的优化技巧并不适用于信息学竞赛，我们只需要掌握高级语言中的优化技巧。

慢。简而言之，在众多的 CPU 指令中，凡是在 bit 级别进行操作的，基本上都能够在一个 **clock cycle**[1]中完成，例如位移(bit shift)、位扫描(bit scan)、逻辑运算等等，再加上加减法、取补码等等简单的整数运算，组成了一个非常高速的指令集合。而巧妙地组合这些运算，往往能取代一些原本缓慢的操作，获得速度上的极大提升。这里列出一些我（骆可强）所收集或者原创的 C 代码位运算技巧[2]：

一、位压缩的技巧[3]

在竞赛中，为了节省内存、加快运算速度或者进行状态压缩 DP，我们时常需要将一组布尔变量压缩到一个打包的 32bit 变量中。要读写其中的单个 bit，使用位运算是最合适不过的了。

```
读取第 k 位： a>>k&1
读取第 k 位并取反： ~a>>k&1
将第 k 位清 0： a&=~(1<<k)
将第 k 位置 1： a|=1<<k
将第 k 位取反： a^=1<<k
将第 k1~k2 位反转： a^=((1<<(k2-k1+1))-1)<<k2
是否恰好只有一个 true： !(x&(x-1))&&x
判断是否有两个相邻的 true： x>>1&x
是否有三个相邻的 true： x>>1&x>>2&x
```

二、打包位统计

延续上面的话题，在 bit 被打包以后，我们时常想要快速知道一些关于整个一个包的信息，例如里面有多少个 true 等，固然可以一个 bit 一个 bit 地处理，不过这样就使打包的优势荡然无存了，想要 O(1)又没有特别好的办法，一个比较常见的处理方式是先预处理一张比较小的表，例如 216，然后将要查询的包分段查询再汇总。这里要介绍的方法技巧性比较强，纯粹使用位运算在对数时间进行处理。

(1) 统计 true 的个数的奇偶性：

```
x^=x>>1;x^=x>>2;
x^=x>>4;x^=x>>8;
x^=x>>16;
```

运算结果的第 i 位表示在原始数据中从第 i 位到最高位 true 数目的奇偶性，有了这个结果，我们就可以很方便地得到任意一段的奇偶性：如果想要得到 k1~k2 位中 true 个数的奇偶性，直接计算`(x>>k1^x>>(k2+1))&1`即可。

(2)统计 true 的数目：

```
int count(unsigned int x)
{
  x=(x&0x55555555)+(x>>1&0x55555555);
  x=(x&0x33333333)+(x>>2&0x33333333);
  x=(x&0x0F0F0F0F)+(x>>4&0x0F0F0F0F);
```

[1] 时钟周期，指 CPU 完成一个基本指令所需的最小时间单位。例如 CPU2.0GHz，就是指 1s 内能完成约 20 亿次基本运算。理论上说，位运算一般需要 1 个时钟周期，加减法需要 2 个时钟周期，乘法需要 3 或 4 个时钟周期，除法（取模）需要超过 20 个时钟周期。浮点运算要在相应的运算基础上增加计算位数的时间。

[2] 同样，本文不在汇编级别研究这个问题，虽然可能在汇编级别思路更为广阔、优化效果也会更好。

[3] 下面的代码都假设已经声明了一个 32bit 变量 a，它的 32 个 bit 是表示 32 个布尔标志。

```
  x=(x&0x00FF00FF)+(x>>8&0x00FF00FF);
  x=(x&0x0000FFFF)+(x>>16&0x0000FFFF);
  return x;
}
```

又是一个基于二分思想的算法，需要特别注意，传入的参数 x 的类型是 unsigned int。在位运算中，我们所期望的类型都应该是**无符号整数**[1]。

(3)反转位的顺序：

```
unsigned int rev(unsigned int x)
{
  x=(x&0x55555555)<<1|(x>>1&0x55555555);
  x=(x&0x33333333)<<2|(x>>2&0x33333333);
  x=(x&0x0F0F0F0F)<<4|(x>>4&0x0F0F0F0F);
  x=(x&0x00FF00FF)<<8|(x>>8&0x00FF00FF);
  x=(x&0x0000FFFF)<<16|(x>>16&0x0000FFFF);
  return x;
}
```

原理和上一个程序基本一样。

三、消除分支

(1) 计算绝对值：

```
int abs(int x)
{
  int y=x>>31;
  return (x+y)^y;
}
```

(2)求较大值：

```
int max(int x,int y)
{ int m=(x-y)>>31;
   return y&m|x&~m;
}
```

(3)x 与 a,b 两个变量中的一个相等，现在要切换到另一个：

```
x^=a^b;
```

四、其他

（1）不使用额外空间交换两个变量：

[1] 因为对于有符号的类型，在进行右位移时将采用算术位移(arithmetic shift)的方式，对应于指令集中的 sar 指令，它和 shr 指令的行为是有差异的，如果这里出了错，往往会导致艰苦的调试。有符号数据常出的另一个问题是，在进行类型扩展时，不是填充 0 而是扩展原来的最高位，这里也常常导致非常难以发现的错误。总之，在进行位运算的子程序中，推荐全部使用无符号数据类型。

```
void swap(int& x,int& y)
{x^=y;y^=x;x^=y;};
```

（2）计算两个整数的平均数，不会溢出：

```
int ave(int x,int y)
{return (x&y)+((x^y)>>1);};
```

3.4 数组运算的速度

（1） 直接应用数组

```
for(i=0;i<10000;i++)
   for(k=0;k<10000;k++)
     t=q[k];
```

净运行时间 39.85ms。这里计算了内层循环的时间。若改为

```
for(i=0;i<100000000;i++)
     t=q[0];
```

则净运行时间为 1.50ms，很快，与 202.3ms 的循环时间相比，可以忽略。故应用数组，速度很快，不必担心数组寻址耗时。同时我们发现，循环耗时在各种运算中是很大的，仅次于乘除，故我们要尽量减少循环次数，能在一个循环中解决的问题不放在两个循环中，减少循环变量次数。

（2）二维数组

```
for(i=0;i<5000;i++)
  for(k=0;k<5000;k++)
    t=z[i][k];
```

实际运行时间为 80.45ms，若规模扩至 10000 则 10s 内无法出解，由于频繁访问虚拟内存。可以试想，若物理内存足够大，则运行时间约为 320ms，仅为 202.3ms 的基准运行时间的 3/2，差距似乎并不是很大；由此推得其净运行时间约为 120ms。但相较加、减等简单操作，速度仍为 3 倍，尤其与几乎不需时间的一维数组相比差距巨大。尤其是在计算中，二维数组元素经常调用，时间效率不可忽视。所以，对于已知数目不多的同样大小的数组，可用几个变量名不同的一维数组表示，如 x、y 方向，两种不同参数，而不要滥用二维数组。在滚动数组中，可用两个数组交替计算的方式，用二维数组同样较慢。

由汇编语言的有关讨论[1]，由于 imul 是一种比例时间较大的指令，因此如果能在运算中消去这一指令，便能够产生较大幅度的速度的优化。所以，一种很巧妙的优化方式就产生了：如果操作的变址方法固定（比如像宽度优先搜索，变址操作为+1,-1,+N,-N），那么用 type*指针加减操作以及辅助记录要更快一些（省去了乘法操作）。

对指针的滑动操作可以用一个常量数组定义或者用多段代码的方式实现。这种操作被我称为指针的“行走”操作。使用这个优化有个条件，就是指针变化方式固定。

[1] 引自周以苏《论反汇编在时间常数优化中的应用》。

减少高维数组寻址[1]

例如在常见的动态规划程序中可以看到类似这样一句话：

```
if(a[x-1][yy][z-3]<a[x][y][z])
  a[x][y][z]=a[x-1][yy][z-3];
```

重复对高维数组进行寻址的代价将是不可忽视的，这样写效率就会高上许多：

```
int &mn=a[x][y][z],v=a[x-1][yy][z-3];
 if(v<mn)mn=v;
```

我们对一个多维数组的读写时常并不是随机顺序的，在许多情况下，我们访问了一个多维数组中的元素，接下来要访问的元素就在它的附近。例如当我们遍历一个高维数组时，访问了 a[x][y][z]之后就将访问 a[x][y][z+1]，而其实 a[x][y][z+1]的地址可以直接通过 a[x][y][z]的地址递增得到。又比如，在动态规划程序中，我们可能总是会从 a[x-1][y-2]去更新 a[x][y]，在这种情况下，**预先算出一个偏移量**，然后就可以通过简单的指针加减运算来访问两个内存位置了。 在编写和数组相关的代码时，强烈建议**尽量使用指针**来进行所有操作，并且尽量不去依赖于编译器实现的多维到一维的映射。就算是进行简单的数组遍历，也能取得很可观的优化效果。

我们可以看出[2]，尽管使用 2 的方幂数据在许多方面都会使程序更加高效，但在做多维数组的尺寸时恰恰相反，**在定义多维数组时，我们不应该让任意一维的尺寸是 2 的方幂**，最好所有的尺寸都是奇数。在编写基于状态压缩的动态规划程序时我们常常会违背这个原则，因为 dp 的状态中有一维总会是 2 的方幂，在这种时候我们可以适当把数组开大一点点，虽然不保证一定会取得优化，但养成良好的习惯终会受益。

3.5 实数运算的速度

测试方法与“基本运算”类似。（次数：100000000，单位：ms）

运算符	=	+	-	*	/	%
long int	43.05	31.3	-74.75	69.55	299.65	360.5
int64	41.45	14.95	7.9	566.4	1243.45	1858.85
double	46.15	10.25	12.6	33.65	1753.55	--

由上表可见，涉及乘除、取模时 int64 很慢，要慎用；int 显然最快，但对大数据要小心越界。若一组变量中既有超出 int 的，又有不超过 int 的，则要分类处理，不要直接都定义成 int64，尤其在乘除模较多的高精度过程中。

以上讨论了主要基本运算的速度问题。概括起来说，除、模最慢，二维数组较慢，加减乘、逻辑位运算、比较大小较快，左右移位、一维数组、赋值几乎不需要时间。而循环 for 或 while 语句十分特殊，它的运算速度大于判断大小、自加控制变量所用时间之和，无

[1] 引自骆可强《论程序底层优化的一些方法与技巧》。

[2] 骆克强原文中讨论了 CPU Cache 的运算机制，从地址冲突的角度得到这样的结论。这样的讨论超出了本文的范畴。实验步骤是：对一个 2048×2048 的二维数组进行遍历并赋值，外层再循环 100 次来放大效果，简单地使用 linux 的 time 命令来测量时间，此程序运行的 user time 为：1.760s。 现在我交换 x 与 y 循环的顺序，即改为按照列顺序来访问，按理来说，这个改动在 C 语言级别，甚至在汇编语言级别都应该是无伤大雅的，我们再次运行程序并测量时间。令人大跌眼镜地是，改动后的程序运行时间为 18.757s，慢了整整十倍还要多。 现在，我将数组的尺寸扩大 1，即变为 2049×2049，现在需要处理的数据更多，想象起来程序应该继续变慢，但是测量的结果是：4.644s，比上一个程序却快了 4 倍。

论采用内部 if 判断退出，还是在入口处判断，都回用去约 200ms 的时间。所以尽量减少循环次数，是优化算法的关键。对于双层或多层的循环，应把循环次数少的放在最外层，最大的循环位居最内部，以减少内层循环的执行次数。

3.6 程序书写习惯

有时为了程序效率的提高，需要牺牲简洁性、直观性，但在信息学竞赛中是必要的。

对一个需要反复使用且值不变的表达式，可以将其赋值为一个变量，需要时直接引用。例如字符串的长度，可以先用`l=strlen(a);`表示出来，以后直接应用 l 即可。

在多重循环的判断中，在外重循环及时剪掉无用的“枝条”。在程序的注释中蕴含着程序常数优化的思想。例如经典的 **24 点游戏**[1]：

程序一：（Accepted, 92ms）[2]

```
#include<stdio.h>
double fun(double a1,double a2,int b)
{ switch(b)
  { case 0:return (a1+a2);
    case 1:return (a1-a2);
    case 2:return (a1*a2);
    case 3:return (a1/a2);
  }
}
main()
{ int i,j,k,l,n,m,r,save[4];
  double num[4]={1,1,1,1},tem1,tem2,tem3,abc=1111;
  char sign[5]="+-*/",t[4][2]; //像这种a[n][2]的数组定义是不合理的，应当定义成a1[n],a2[n]，或应用结构体struct，避免高维数组的寻址。[3]
  for(i=0;i<4;i++)
  { scanf("%s",t[i]);
    if(t[i][0]>='1'&&t[i][0]<='9')
     num[i]=t[i][0]-'0';
    if(t[i][0]=='A')  num[i]=1;//这种逐一判断的方法既麻烦，又容易出错，不妨用一个数组建立起一一映射，再用 for 循环查找。
    if(t[i][1]=='0')  num[i]=10;
    if(t[i][0]=='J')  num[i]=11;
    if(t[i][0]=='Q')  num[i]=12;
    if(t[i][0]=='K')  num[i]=13;
  }
  if(num[0]==3&&num[1]==3&&num[2]==8&&num[3]==8)
  { printf("yes");
```

[1] 24 点的规则很简单，就是给你 4 张扑克（从 1 至 13，用 A 代替 1，J 代替 11，Q 代替 12，K 代替 13）通过加减乘除来求得 24。提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=74

[2] 笔者编写，http://www.rqnoj.cn/Status_Show.asp?SID=23443

```
    return 0;
  }
  if(num[0]==5&&num[1]==5&&num[2]==5&&num[3]==1)
  { printf("yes");
    return 0;
  }
    for(i=0;i<4;i++)
     for(j=0;j<4;j++)
      if(j!=i) //判断剪枝
      { for(k=0;k<4;k++)
         if(k!=i&&k!=j) //判断剪枝
         { for(l=0;l<4;l++)
            if(l!=i&&l!=j&&l!=k) //判断剪枝
            { for(n=0;n<4;n++)
               for(m=0;m<4;m++)
                for(r=0;r<4;r++)
                { tem1=fun(num[i],num[j],n);
                  tem2=fun(tem1,num[k],m);
                  tem3=fun(tem2,num[l],r);
                  if(tem3==24.0)//这些 if 语句可以合成一句
                   goto loop;
                  else if(tem3==-24.0)
                   goto loop;
                  else if(tem3==1.0/24.0)
                   goto loop;
                  else if(tem3==-1.0/24.0)
                   goto loop;
                  else
                  { tem1=fun(num[i],num[j],n);
                    tem2=fun(num[k],num[l],r);
                    tem3=fun(tem1,tem2,m);
                    if(tem3==24.0)
                     goto loop;
                  }
                 }
            }
         }
      }
      printf("no");
      return 0;
      loop:printf("yes");
}
```

[0] 事实上，对于一个元素多种性质的问题，应用结构体，而不是二维数组或多个一维数组，可以提高效率，且逻辑清晰。这样的问题在动态规划、搜索中常常出现。

程序二：(Accepted, 517ms）[1]

```
#include<iostream>
#include<string>
using namespace std;
string card="!A234567890JQK";
double pos[4]; bool used[4];
bool dfs_try(int depth,double amount){
if (depth==4){
if (amount>24-0.00001&&amount<24+0.00001) return true;
//对于实数的比较，一定要使用误差机制
else return false;
}
else{
for (int i=0; i<4; i++)
if (!used[i]){
used[i]=true;
if (dfs_try(depth+1,amount+pos[i])) return true;
if (dfs_try(depth+1,amount-pos[i])) return true;
if (dfs_try(depth+1,pos[i]-amount)) return true;
if (amount!=0&&dfs_try(depth+1,amount*pos[i])) return true;
//这里 amount!=0 的判断十分重要，边界情况注意处理
if (amount!=0&&dfs_try(depth+1,amount/pos[i])) return true;
if (amount!=0&&dfs_try(depth+1,pos[i]/amount)) return true;
used[i]=false;
}
return false;
}
}
int main (void){
string c;
for (int i=0; i<4; i++){
cin>>c; if (c=="10")c="0";
//在 C++语言中，cin 要比 scanf 慢很多，在读入大数据（如矩阵）时尤其应该注意。
if(c=="1") pos[i]=1;
else pos[i]=card.find(c);
used[i]=false;
}
if (dfs_try(0,0)==true) cout<<"yes";
else cout<<"no";
return 0;
}
```

[1] 飞雪天涯编写，http://www.rqnoj.cn/Status_Show.asp?SID=115404

显然，第二个程序比第一个程序简练很多，但时间效率也慢了不少（92ms，517ms）。两个程序的设计机理都是由 4 个选择两个经运算变为 3 个，再变为 2 个，最后得到结果。但由于判重和优化上的区别，时间效率、编程复杂度都有所不同。

再举一个例子《**凯撒密码**[1]》：将一个正整数分成尽可能少的平方数之和。题目本身不难，但体现了本文中的多个理念。首先，懂得一个重要的数学定理“**拉格朗日四平方和定理**[2]”，即每个正整数均可表示为不超过四个正整数的平方和，是解题的关键。其他值得注意的细节标注在源代码中。看似繁杂，实则是一种编程习惯，养成后受益匪浅。

```
#include<stdio.h>
#include<string.h>
#include<math.h>
char c[100000]; //尽量避免“用多少开多少”，要留出少量空闲，以免数组越界
int x[300]={0}; //注意变量赋初值，除非循环变量、临时变量，尽量赋为零
void print(int p,int n) //单独的输出函数，逻辑性更强[3]
{ int i,j;
  for(i=0;i<n;i++)
   for(j=0;j<n;j++)
    printf("%c",c[p+n*j+i]);
}
int main()
{ int n,l,t,i,j,k,p,a;
  gets(c);
  l=strlen(c); //将多次用到的表达式、函数存在变量中，避免重复计算
  t=sqrt(l);
  if(t*t==l) //首先解决“平凡问题[4]”，也是易出错的问题，尽量特殊处理
  { print(0,t);
    return 0; //尽量避免使用 goto 语句
  }
  for(i=1;i<=t;i++)  //预处理[5]平方数
   x[i]=i*i;
  for(i=t;i>=0;i--)
   { a=sqrt(l-x[i]);  //又用到了事先计算保存，因为 sqrt 的效率很低
     if(x[a]+x[i]==l) //这里的技巧是判断一个数是否平方数，不用再枚举
     { print(0,i);
       print(x[i],a);
       return 0;
     }
   }
  for(i=t;i>=0;i--)
   for(j=i;j>=0;j--)//这句可优化为 a=sqrt(l-x[i]),j<=a,避免无效枚举
```

[1] 提交地址：http://www.vijos.cn/Problem_Show.asp?id=1480

[2] 笔者模拟赛时不知道这个定理，是通过计算机枚举试验得到这个规律的，考后才找到证明。这体现了“骗分”思想中的“数学分析与猜想”。

```
        if(x[i]+x[j]<l)  //在外层循环过程中判断可能性
         { a=sqrt(l-x[i]-x[j]);
           if(x[i]+x[j]+x[a]==l)
           { print(0,i);
             print(x[i],j);
             print(x[i]+x[j],a);
             return 0;
           }
         }
      for(i=t;i>=0;i--)
       for(j=i;j>=0;j--)
        if(x[i]+x[j]<l)
          for(k=j;k>=0;k--)
           if(x[i]+x[j]+x[k]<l)
            { a=sqrt(l-x[i]-x[j]-x[k]);
              if(x[i]+x[j]+x[k]+x[a]==l)
              { print(0,i);
                print(x[i],j);
                print(x[i]+x[j],k);
                print(x[i]+x[j]+x[k],a);
                return 0;
              }
            }
      return 0;
}
```

以上的优化总结起来，主要包括：预处理、记忆化、判断可能性剪枝、减少函数调用。在编程的习惯方面，主要有：数组大小、结构体、指针数组、特殊情况处理、数组越界、输入输出、变量初值、边界处理。

程序的优化是无止境的。我们通常说，复杂度为 $O(n^2)$的算法 n 最大为 5000，共有 $2.5*10^7$ 次操作。而注意到 CPU 的主频是 2.0GHz，就是 1s 内完成 $2*10^9$ 次基本操作。难道每次操作真的要占用 100 个时钟周期吗？事实上我们循环体内的运算也不是十分复杂。那么，必然有大量的时间被“浪费”掉了。不合理的编程习惯，如反复调用函数、无效状态的继续处理、频繁计算同一个表达式、直接使用高维数组[1]，都会导致程序效率大大降低。

由此可见，我们的程序还有很大的优化空间。一个编写精致的代码和一个算法大概正确的代码，执行起来效率将相差一倍以上，直接影响极限数据的通过与否。对于信息学竞赛中时限很紧的试题，复杂度常数不能不引起我们的重视。我们相信，只要重视常数复杂度优化，

[-2] 笔者一般不主张独立出只使用一次的非递归函数，由于这样会增加函数调用的时间。尤其是类似 max {a,b}, abs(a)之类的函数，完全可以用 a>b?a:b, a>0?a:0 代替。能够连续完成的任务，完全可以放在一个函数中完成，可以用注释、空行来说明模块。对于内部重复多次使用的模块，更要编写逻辑紧凑，精心优化，减少冗余计算。模块化思想与程序效率也是一对矛盾，在竞赛中我们宁可损失可读性也要追求效率。

[-1] 平凡问题，就是“一眼看上去就十分显然”的特殊化问题。平凡解，就是一般问题中为零、为 1、相等等特殊、易于观察的取值。在信息学竞赛中，往往这些看似简单的特殊情况要单独处理。

[0] 预处理，一般是对于频繁用到的数量事先计算，如素数、平方数、平方根、2 的次幂，是与输入无关的常量。最大子矩阵问题中的预处理出从左上角开始的子矩阵之和，是更高级的预处理。

[1] 会导致程序使用多次乘法为元素定址，不如指针直接加减运算迅速。

养成良好的编程习惯，就能用同样的算法 AC 更多的数据。

3.7 小算法 大优化[1]

一些十分经典的算法，看起来效率很高，没有什么优化的余地了；事实上，如果从小处着眼，一些经典算法仍有不少可优化的空间。

首先看**快速排序**。

```
#define MAX 10000000
int a[MAX];
int p(int l,int r)
{ int x=a[l],i=l-1,j=r+1,t;
  while(1)
  { do{--j;
    }while(a[j]<x);
    do{++i;
    }while(a[i]>x);
    if(i<j)
    { t=a[i];a[i]=a[j];a[j]=t;
    }
    else return j;
  }
}
void q(int l,int r)
{ int n;
  if(l<r)
  { n=p(l,r);
    q(l,n);
    q(n+1,r);
  }
}
```

运行时间[2]：2948ms。另外，快速排序不是**稳定的排序**[3]，需要保持原顺序的不能用此法。

```
void p(int l,int r)
{ int x=a[l],i=l-1,j=r+1,t;
  if(l<r){
  while(1)
  { do{--j;
    }while(a[j]<x);
    do{++i;
```

[1] 本节改编自笔者《算法效率与程序优化》。

[2] 测试环境为：CPU Intel Core 1.73GHz*2，内存 512M，操作系统 Windows 7 Ultimate Beta，程序语言 C，编译环境 Dev-c++ 4.9.9.2。并在运行较多程序时测试，不具有理论意义，只具有相对比较意义。

[3] 稳定的排序包括冒泡排序、插入排序、归并排序、基数排序等，关键字相同的排序后顺序不变。

```
    }while(a[i]>x);
    if(i<j)
    { t=a[i];a[i]=a[j];a[j]=t;
    }
    else break;
  }
  p(l,j);
  p(j+1,r);
  }
}
```

若程序改为以上形式，则运行时间为 2917ms，稍快了一些，是因为减少了函数调用次数。对于函数调用，我们进行这样的测试。

```
s=clock();
for(i=0;i<100000000;i++)
   test();
t=clock();
int test()
{ int n;
  n++;
}
```

净运行时间 200ms

```
int test()
{ int n;}
```

净运行时间 84ms

```
int test()
```

净运行时间 10ms

由此可见，调用函数本身并不浪费时间，仅相当于循环本身时间 400ms 的 1/40，相当于加法 80ms 的 1/8，是很快的运算。但由于在函数内部需要进行现场保护，调用系统堆栈，所以用时大幅增加，定义变量后只是一个自增运算就用去 120ms，相当于主程序中加法运算时间的 3/2 倍。故函数中的运算比主程序中要慢，尤其是反复调用函数，会增加不必要的时间开销。所以，一些简单的功能尽量在一个函数或主程序内完成，不要使用过多的函数；涉及全局的变量不要在函数调用时由接口给出，再返回值，尽量使用全局变量。这些方法可能使程序的可读性降低，不利于调试，但有利于提高时效，正如汇编语言程序比高级语言快一样。

（1）用插入排序优化

由于递归调用浪费大量时间，本算法的思想是，当首尾间距小于 min 时，改用效率较高的插入排序，减少反复递归。这个想法是好的，但运行效果并不如人意。当 min=4 时，程序运行时间降为 2901ms，优化幅度不大，且增加了编程复杂度，故不宜采用。其原因是递归调用、插入排序内部循环所用时间过长。

（2）用小数据判断优化

```
  if(l+1<r){
  while(1){}//与普通快速排序相同
  else if(l+1==r&&a[l]<a[r])
    { t=a[i];a[i]=a[j];a[j]=t;}
```

仅就区间长度为 2 的情况进行判断优化，减少一次递归调用，就能使运行时间缩短为 2699ms，降低了 200ms 以上。而区间长度不小于 3 的直接判断有困难。

快速排序运行时间（单位：ms）

数据规模	10000	100000	1000000	10000000
原始快速排序	1.50	25.20	258.55	2716.45
优化快速排序	2.30	23.40	245.40	2564.80

我们再看一个优化 Kruskal 算法的例子。笔者有意不加注释，请读者自行比较、思考。

```
int father(int x)
{ if(set[x]!=x)
   set[x]=father(set[x]);
  return set[x];
}
void kruskal()
{ int i,j,start,end,value,cost=0;
  for(i=0;i<n;i++)
   set[i]=i;
  for(k=1;k<n;k++)
  {  value=MAXINT;
     for(i=0;i<n;i++)
      for(j=0;j<n;j++)
       if(i!=j&&father(i)!=father(j))
        if(data[i][j]<value)
        { start=i;
          end=j;
          value=data[i][j];
        }
      cost+=value;
      set[father(start)]=father(end);
   }
   printf("%d ",cost);
}
```

当 max=100000 时，即共 100000 个点，共 1000000 条边时，平均运行时间为 3563ms。

下面给出一个较为优化的算法。

```
int find(int x)
{ if(f[x]!=x)
   f[x]=find(f[x]);
  return f[x];
}
void kruskal()
{ int i,j,a,b,tot=0,num=0;
  for(i=0;i<n;i++)
   f[i]=i;
  for(i=0;i<n;i++)
```

```
  { a=find(s[i]);
    b=find(e[i]);
    if(a!=b)
    { num++;
      tot+=w[i];
      f[a]=b;
      if(num==max)
       break;
    }
  }
  printf("%d ",tot);
}
```

此处 sort 函数是快速排序函数，采用了本文上述“优化的快速排序算法”。

当 max=100000 时，即共 100000 个点，共 1000000 条边时，平均运行时间为 409.4ms。当 max=10000 时，平均运行时间为 43.7ms。

从 3563ms 到 409ms，接近 9 倍的时间差别，不能不让我们对常数优化的威力叹为观止。

由上面的讨论可以看出，即使是经典算法，也有少量的优化空间。积少成多，聚沙成塔，尤其是对“速控步”的优化，程序的效率会有一个飞跃。

在信息学竞赛中，提高程序运行速度，是我们不懈的追求。本文中给出的算法一定不尽完善，也有很多值得玩味、优化的算法没有在这里列出。从细节入手，对症下药，是本文的主旨，也是我们希望众多 OIers 做到的一点。我们相信，只要认真对待程序的每一个语句，在平时养成优化代码的好习惯，就能在竞赛中的常数级时间上占优势，达到“同样的算法，不同的分数”的目的。

3.8 空间复杂度优化

在信息学竞赛中，空间复杂度往往是选手容易忽略的方面。我们常说“用空间换时间”，于是很多选手不注重空间复杂度优化，盲目开大数组、高维数组，程序运行起来动辄 20M，是完全没有必要的。尤其是盲目开高维数组，还会造成时间复杂度的升高。

一个好端端的程序，却在内存占用上被卡住，出现了 Memory Limit Exceed，就得不偿失了。所以我们进行空间复杂度优化，目的就是要“恰好”利用题目中所给的内存，不要造成空间浪费。

下面给出常用变量的空间占用情况，供参考。

（1）INT（long）类型占 4 字节

（2）Long long int 或 int64 类型占 8 字节

（3）一个 int a[1000000]的数组占 4M，通常空间限制为 64M 时，最多开 15 个这样的数组。若要开规模为 10000000 的 int 数组，占用达到 40M，一定只允许一个。

（4）一个 int a[1000][1000]的二维数组同样占 4M。一个 int a[2000][2000]的二维数组占 16M，最多开 3 个。二维数组最多允许 3000*3000 的规模，且只有一个。

（5）在使用空间时，一定要留出 10M 左右的空闲空间，不要将允许空间挤满，在大内存程序调试时可以使用工具查看内存实际占用情况，不要铤而走险打“擦边球”。不可能只有一个数组占用内存，要综合考虑整个程序。

（6）如果是允许 256M 内存，一维数组最多到 5*10^7，即 int a[50000000]，如果两个

最多到 3*10^7。二维数组最多开到 int a[7500][7500]，或两个 5000*5000 的数组。这里的计算都考虑了余量，不能紧逼上限，以免发生意外。

下面是笔者一些空间复杂度优化的经验：

（1）将二维带参变量的数组用结构体一维数组代替

（2）使用滚动数组，将无用的空间及时释放

（3）高精度运算尽量在原数基础上运算

（4）搜索时在原来状态基础上修改，少引入中间状态、临时状态

（5）队列使用循环队列，采用“MOD 队列长”的办法解决，不要只有 begin, end 两个指针，造成使用过的空间浪费。尤其是广度优先搜索中注意。

（6）少使用链表等非线性结构，减少定位域的个数

以上这些降低空间复杂度的经验，由于减少了重复计算和数据复制，还可以提高时间效率，这在搜索类问题中尤其见效。

但我们仍然强调，除非题目的最大内存使用数不确定，尽量在初始化时一次性申请好所有内存，而不是使用 malloc 和 free 去动态申请内存。因为这样增加了编程复杂度，况且动态申请内存也会出现 MLE 的情况，还不如直接估计好最大可能达到的，或最大能够承受的内存，利用静态内存进行运算。

在避免数组越界方面，尽量遵守这些规则：

（1）数组规模比测试数据的最大规模略大，如 n=10000 的试题，数组可以开到 10005

（2）尤其是 C 语言，减少 0 位的使用（高精度运算除外），从 1 使用到 n，符合自然思考习惯，便于输入输出，不易出现数组元素正负 1 的问题，并以 0 位置为“哨兵”，放上 0 或者特殊标记，既不会在状态转移时出现数组越界，又避免了初始位置的特殊讨论，何乐而不为？

（3）在定义数组时为今后的编程提供方便，减少特殊情况数，而不是到具体处理时再解决边界问题、特殊处理问题，这样既浪费时间又容易出错。

当然，某些题目确实状态空间庞大，无法使用“小修小补”的常数空间复杂度优化解决问题，那么就要考虑“**状态压缩**”，就是**将无用的空间及时释放**。从这个意义上说，动态规划中的滚动数组、广度优先搜索中的循环队列，都是状态压缩的具体算法。

3.9 降低编程复杂度

竞赛时间是有限的，为了解决尽可能多的试题，需要降低编程复杂度，用尽可能短的代码达到较好的目的。

降低编程复杂度，这里给出笔者的几条经验：

（1）减少使用指针，尽量使用数组

（2）保持程序逻辑清楚，变量名、函数名明确

（3）定义常量，将最大规模、MAXINT、数学常数等用#define 或 const 预定义。

（4）注意缩进，层次清晰

（5）对于复杂的分类问题，将各种类别表示在数组中，使用循环进行统一判断，而不是分别判断处理。例如走棋盘类问题中四个方向的处理，表达式处理中的不同运算符，搜索问题中的相似选择支，模拟类问题中的数据读入、不同操作符数学模型的转化，字符串处理中的功能相似的符号，都应该在数组中进行归类处理，这样程序的可移植性增强，可以快速添加、删除、修改类别，便于调试时对不同类

别同时修改，还不容易出错。[1]

降低编程复杂度，很重要的便是**模块化思想**。模块化思想的最大优点就是，**逻辑清晰、便于调试**。在本文“调试程序”一节中会体现模块化的巨大优势。

（1）将程序的各大功能板块，如输入、运算、输出分开编写。

（2）每一个部分解决一个问题，不要“眉毛胡子一把抓”，尽量保持各个部分之间的独立性，尤其是变量不要反复引用，平行的中间变量尽量不要重叠，在程序段的开始时注意初始化（尤其是最大最小值之类）。

（3）将一些成型的、经典的算法在考前练习熟练，考试时直接用函数写出来，不需要再多考虑、调试。包括排序、并查集、基本数学算法等，在本文“应该学习的内容”中有所介绍。

（4）这里要指出，模块化不是将一些较小的、反复用到的函数单独拿出来，因为这样虽然逻辑清晰，但损失了时间复杂度，可以使用 Ctrl+C 的办法解决，反正源代码的长度是没有限制的。

看一个例子：PKU1657《Distance on Chessboard》[2]，在国际象棋盘上，给定起始位置和目标位置，计算王、后、车、象从起始位置走到目标位置所需的最少步数。先看一位 AC 者的代码[3]：

```
#include<stdio.h>
#include<math.h>

int max(int a,int b){
              return a>b?a:b;
};
int main()
{
int n,flag,i;
char str[100],str1[100];
int n1,n2,n3,n4;
int dist1,dist2;

scanf("%d",&n);
for(i=1;i<=n;i++){
  flag=0;
scanf("%s %s",&str,&str1);
dist1=abs(str[0]-str1[0]);
dist2=abs(str[1]-str1[1]);

if(dist1==0&&dist2==0)
              n1=n2=n3=n4=0;
else if(dist1==dist2){
```

[1] 笔者在搜索问题中深受“Ctrl+C”的危害，有时修改这处忘了那处，有时本来不同的两种变换在复制代码时忘记修改，调试十分困难。使用数组法后，不仅代码量减少，编程所需时间也少了。

[2] 提交地址：http://acm.pku.edu.cn/JudgeOnline/problem?id=1657

[3] http://acm.pku.edu.cn/JudgeOnline/showmessage?message_id=99375，已修正其中的错误。

```
                n1=dist1;
                n2=1;
                n3=2;
                n4=1;
    }
    else if(dist1==0&&dist2%2!=0){
                n1=dist2;
                n2=1;
                n3=1;
                flag=1;
    }
    else if(dist1==0&&dist2%2==0){
                n1=dist2;
                n2=1;
                n3=1;
                n4=2;;
    }
    else if(dist2==0&&dist1%2!=0){
                n1=dist1;//这句原来错写为 n1=dist2;
                n2=1;
                n3=1;
                flag=1;
    }
    else if(dist2==0&&dist1==0){
                n1=dist2;
                n2=1;
                n3=1;
                n4=2;
    }
    else
if((dist1%2==0&&dist2%2==0)||(dist1%2!=0&&dist2%2!=0)){
                n1=max(dist1,dist2);
                n2=2;
                n3=2;
                n4=2;
    }

    if(flag)
                printf("%d %d %d Inf\n",n1,n2,n3);
    else
                printf("%d %d %d %d\n",n1,n2,n3,n4);
    }
    return 0;
}
```

看到这个代码，读者一定心生厌烦，为了对每种情况分别计算出 4 个棋子的步数，运用了复杂的特殊情况处理和同余分析。为什么不能将如此众多的相同情况合并起来考虑？

下面这段代码是一个 WA 的[1]：

```
for(i=0;i<t;i++){
              scanf("%s %s",qidian,zhongdian);
              wang=WANG(qidian,zhongdian);
              hou=HOU(qidian,zhongdian);
              ju=JU(qidian,zhongdian);
              xiang=XIANG(qidian,zhongdian);
if(xiang!=-1)printf("%d %d %d %d\n",wang,hou,ju,xiang);
              else printf("%d %d %d Inf\n",wang,hou,ju);
}
```

整个程序十分冗长，单看变量名、函数名就让人头疼，怎能顺利解出题呢？代码的逻辑性一定要强，变量名、函数名要言简意赅，过长会浪费“打字时间”。

最后看我的代码[2]：

```
#include<stdio.h>
int main()
{ int t,i,l1,l2,s1,s2,s3;//变量名简单清楚，易于编程
  char a[5],b[5];
  scanf("%d",&t);
  for(i=0;i<t;i++)//多组测试数据的问题注意初值，可以在循环内定义变量
  { scanf("%s%s",&a,&b);
    l1=a[0]>b[0]?a[0]-b[0]:b[0]-a[0];//采用问号表达式，避免函数调用
    l2=a[1]>b[1]?a[1]-b[1]:b[1]-a[1];//预处理一些常用的取值
    s1=l1>l2?l1:l2;//抓住特殊性
    if(l1==0&&l2==0)//先进行特殊情况处理，避免边界错误
    { printf("0 0 0 0\n");continue;}
    else if(l1==0||l2==0)//合理归类，按逻辑分析
      s2=s3=1;
    else
    { s2=1;s3=2;}
    printf("%d %d %d ",s1,s2,s3);//已经确定的值立即输出，避免被修改
    if(l1==l2)
      printf("1\n");
    else if(l1%2==l2%2)
      printf("2\n");
    else printf("Inf\n");//抓住奇偶性特征，分类输出
  }
```

[1] http://acm.pku.edu.cn/JudgeOnline/showmessage?message_id=40218，这个程序的特点是非要将各种棋子的分析分开来，与上个程序分别走入了两个极端。

[2] http://acm.pku.edu.cn/JudgeOnline/showmessage?message_id=113424

```
    }
```

比较上述三个代码，我们发现很大的不同。同样的目的，得到的代码无论是长度还是逻辑性都大相径庭。短小精悍的代码，说明了编程者对问题分析深入、透彻，全面把握了问题内在的逻辑关系。对于情况变化多端的问题，抓住内在的逻辑关系，合理分类，“合并同类项”，就可以减少判断，降低编程复杂度。但不能忘记了特殊情况、边界情况，否则容易出错。这在一些模拟、字符串处理等情况众多的问题中尤其应该注意。

再看一个**大合数分解问题**[1]，要求递增输出质数的质因子。

我的程序[2]是：（AC，93ms）

```
#include<stdio.h>
main()
{ __int64 n,flag=1,i=2;
  scanf("%I64d",&n);
  while(n!=1)
  { while(n%i==0)
    { if(!flag)
       printf(" ");
      printf("%d",i);
      flag=0;
      n/=i;
    }
    i++;
  }
}
```

这采用了最简单的试除分解算法，没有使用任何优化。

另一个自称 100ms 的 AC 程序[3]：

```
#include <iostream>
using namespace std;
#ifndef _WIN32
typedef unsigned long long longlong_t; //定义跨平台的 64 位机器数类型
#else
typedef unsigned __int64 longlong_t;
#endif
bool IsLikePrime(longlong_t n, longlong_t base)
{
longlong_t power = n-1;
longlong_t result = 1;
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=114
[2] http://www.rqnoj.cn/Status_Show.asp?SID=19037
[3] http://www.rqnoj.cn/Solution_Show.asp?DID=4038

```
    longlong_t x = result;
    longlong_t bits = 0;
    longlong_t power1 = power;
    while (power1 > 0) //统计二进制位数
    {
    power1 >>= 1;
    bits++;
    }
    while(bits > 0) //从高位到低位依次处理 power 的二进制位
    {
    bits--;
    result = (x*x)%n;
    //二次检测
    if (result == 1 && x != 1 && x != n-1) return false;
    if    ((power&((longlong_t)1<<bits))    !=    0)    result    =
(result*base)%n;
    x = result;
    }
    //费尔马小定理逆命题判定
    return result == 1;
    }
    const int primes[]={2,3,5,7,11};  //前 5 个素数
    int primes_six[sizeof(primes)/sizeof(primes[0])];  //前 5 个素数的
6 次方，由后面的 init 对象初始化
    class CInit //静态初始化类
    {
    public:
    CInit()
    {
    int num = sizeof(primes)/sizeof(primes[0]);
    for (int i = 0;  i < num;  i++)
    {
    primes_six[i] = primes[i]*primes[i]*primes[i];
    primes_six[i] *= primes_six[i];
    }
    }
    }init;
    bool JudgePrime(longlong_t n) //王浩素数判定函数
    {
    if(n<2)return false;
    if(n == 2) return true;
    int num=sizeof(primes)/sizeof(int);
    bool bIsLarge = (n >= primes_six[3]);
    for (int i = 0;  i < num;  i++)
```

```
    {
    if (bIsLarge)
    {
    if (primes[i] == 3) continue; //当n >= 7^6时，不进行上界判断，固定地用{2,5,7,11}做判定。
    }
    else
    {
    if (primes[i] != 2 && n < primes_six[i]) break; //当n < 7^6时，进行上界判断，但是2例外。
    }
    //做一次子判定
    if (!IsLikePrime(n, primes[i]))
    return false;
    }
    //所有子判定通过，则n必为素数！
    return true;
    }
    int main()
    {
    longlong_t n;
    cin>>n;
    bool flag=false;
    for(int i=2; i<=n; i++)
    if(JudgePrime(i))
    while(n%i==0)
    {
    if(flag)cout<<' ';
    flag=true;
    cout<<i;
    n/=i;
    }
    //end
    return 0;
}
```

上述程序使用了较为先进的素数判定算法[1]，当问题规模扩大时，第二种算法的优势将十分明显，这也是我们值得学习的，所以笔者用大量篇幅将整个代码放入论文，也当作是介绍了一种大合数分解、素数判定的高效算法[2]。但注意到题目中的问题规模，我们就应该考虑花费这么多时间编写高效算法是否划算[3]。

[1] 未经测试，不知这个算法与 Miller-rabbin 算法（将在数学方法一节中给出）的优劣。
[2] 当然，这个算法的效率不是最高的，它至少没有应用高等数论；也远不能用于破解 RSA 密码之类。
[3] 刘汝佳教练曾经与我们讨论，是否应该在题目中给出数据规模。笔者坚定地认为应该给出，因为算法的选择要以数据规模为依据，这样才能达到时间复杂度与编程复杂度的平衡。同时，本文后面的“非完美算法”也要以数据规模的分布（对 40%的数据，n<=1000 之类）为基础，考虑低效算法的得分情况。

读了上述两个程序，第一个短短 15 行，第二个长达 86 行，运行时间均为约 100ms，颇具讽刺意味。由此可见，“用运行时间换编程时间”，在时限不紧的情况下还是很实用的。[1]

在解决信息学竞赛试题时，不要看到题目，发现是某个“经典算法”，就动手写程序，而要想一想这个问题是否是这种通用算法的特殊情况？有没有优化的可能？

例如，经典的网络流问题可以转化为线性规划问题，而线性规划问题有单纯形法可以解决，相信没有选手会把一般的最大流问题转化成线性规划来做吧。由此可见，有些问题的特殊情况，有编程复杂度更低、时间复杂度更低的特殊算法，这时应该思考一下优化问题。

例如“恐怖分子”一题：[2]

一个国家由 n 个城市组成，任意两个城市间都有一条双向的道路。如今，恐怖分子想要炸掉其中的一些道路，使得这 n 个城市变得不再连通，也就是说，存在至少两个城市，它们之间不可通过道路互相到达。要炸毁一条道路需要一定的代价。请你计算一下要使恐怖分子达到它们的目的，所需要的最小总代价是多少。

读者可能看到这个问题，就马上想到了最大流，于是开始应用最高标号算法或预流推进算法求最大流。但显然代码长度不少于 150 行，且需要 O(n^4)的复杂度。

不过看清本质之后，这一题本质就是求无向图的全局最小割，这其实是有专门算法的，叫 Stoer-Wagner 算法，复杂度不超过 o(N^3)，而且代码行数很短。[3]

首先我们定义对于不存在的边 (u,v)，w[u, v] = 0。

之后介绍经典的 Contract(a, b) 操作：

加入新点 u，对于 a、b 之外的每个点 v，加入无向边 (u, v) 并令 w[u, v] = w[a, v] + w[b, v]

完成之后删除 a、b 两点及相关联的所有边

不难看出，每做一次 Contract 操作，图的节点数减少 1

Stoer-Wagner 算法的正确性基于如下定理：

一个无向图的全局最小割，等于这个图的 s-t 最小割，或对这个图进行 Contract(s, t) 得到的图的全局最小割。

定理是显然的，下面我们说明如何求无向图的 s-t 最小割：

首先定义对于点集 A 和点 u，w[A, u] 表示 A 中所有点与 u 连边的权值之和。再次说明，不存在的边视为 0 处理。

1. 初始化点集 A={}
2. 任意取一点 a 加入 A 中
3. 每次取 w[A, u] 最大的点 u，将其加入 A，直到 A=V
4. 可取 3 中倒数第二次加入的点为 s，倒数第一次加入的点为 t（无向图顺序无关紧要），则对应的 s-t 最小割为 w[U-{t}, t]

不难看出，上述 s-t 最小割算法的朴素实现是 O(N^2) 的（第三步显然还可以用最大堆优化，达到更好的时间界），而对于一个图，Contract 操作最多进行 n-1 次（此时图中只剩 1 个点了）。所以算法的复杂度不超过 O(N^3)。

[1] 这里涉及本文的中心思想：Keep it simple and stupid. 就是著名的 KISS 原则。后面章节将具体讨论。
[2] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=77
[3] 引自 http://www.rqnoj.cn/Solution_Show.asp?DID=4234

显然，上述算法比 HLPP 要短得多，容易调试，时间复杂度也降低了。

本节开头提出的“用运行时间换编程时间”，就是运用一些并非最优的算法，使程序更易编写。但前面的“常数时间优化”又要求“用编程时间换运行时间”。权衡利弊，如果题目的运行时间卡的不紧，有较多的空余时间（多于 0.5s），或者简单而略微低效的思路比提高一点效率要省很多代码，可以“用运行换编程”；如果运行时限非常紧，如最大的数据需要 0.8s，或者增加、改变很少的几句代码就可以对“速控步”进行优化，可以“用编程换运行”。总之，不让评测机的 CPU 闲置，减少敲打键盘所用的时间，用最少的时间获得最大的收益，是我们“骗分”的追求。[1]

4 数学分析与猜想

4.1 数学与信息学[2]

数学是在信息学竞赛中，许多问题更是与数学息息相关。在组合数学书籍中，一般都有这样的语句“随着计算机科学的发展，组合数学也逐渐得到发展进步”。可见数学与信息学是紧密相关的。事实上，学习信息学竞赛，需要的数学基础，尤其是高等数学，并不比数学竞赛少。

事实上，信息学竞赛对抽象思维的要求，甚至比数学还高。因为数学竞赛往往面对的是具体问题、具体数字，只需要对这一种情况进行考虑，遇到特殊情况临时解决即可。而信息学竞赛是编程题目，计算机面对的是可能性众多的数据，而程序是选手事先写好的，不可能让计算机“随机应变”，故选手必须提前考虑好所有可能，找到解决此类问题的通用解法。所以，信息学竞赛无论是题目描述，还是解答、程序，都远比数学题长。

在大千世界中，我们所面对的事物形形色色，扑朔迷离。它们都是由许多信息构成的。这些现实世界中对客观问题表面的自然语言描述，称为信息原型。当然，在信息学竞赛中我们所面临的问题也是信息原型。信息原型本身是由扑朔迷离的信息构成，掩盖了其重要的属性。我们因而无法直接从信息原型入手找到问题的答案。为此，我们就需要一种方法来“改造”信息原型，使之既具有原来的重要属性，也具有可研究性。

于是，我们试图将信息原型的属性一起转移到一个模型中。模型即是对客观问题属性的模拟。显然，这个对应出的模型可以说是信息原型的代表。我们就可以对这个模型进行研究。

用什么方法将信息原型对应到模型上去呢？我们期望运用数学方法。这样对应出的模型即具有原问题的属性又具有数学的可研究性。我们称之为数学模型。数学模型：运用数学语言对信息原型通过抽象加以翻译归纳的产物叫做数学模型。

信息原型是现实的问题，对应到的数学模型又是理论上的模型，对该模型进行研究使我们得出了现实问题的解。这就是信息学竞赛中解题的简单过程：现实——理论——现实。

数学模型是人们解决现实问题的有力武器。人们把现实问题经过科学地抽象、提炼得到数学模型，再用数学方法去解决。数学模型可分为离散和连续两种。连续数学模型需要大量

[1] 两者其实并不矛盾，本文所述的常数时间优化，是在不影响程序逻辑性的前提下养成一些良好的编程习惯；而运行时间换编程时间，则是增强程序的逻辑性，使程序更加紧凑、易于调试。殊途同归，两种策略最终在时间复杂度低、编程复杂度低、逻辑性强获得交汇，唯一损失的就是程序的可读性。

[2] 本节摘录自笔者《骗分导论·数学竞赛》。参考、引用了多篇信息学集训队论文，一并感谢。

的高等数学知识，中学生很少接触。在信息学竞赛经常出现的则是离散数学模型。

对于同一个现实问题，可能可以建立不同的数学模型。这种现象十分普遍，也就是平时所说的一题多解。既然如此，这里有必要讨论一下数学模型的选择问题，其实也就是评判一个模型好坏标准的问题。一个好的数学模型不仅要能够准确地反映出现实模型（可靠性），所建立的模型还必须能够被有效地解决（可解性）。这里“有效”指的是解决它的算法所需的空间与时间都在可以承受的范围之内。通常在解一些要求最优解或要求准确计数的一类具有唯一正确解的试题时，我们一般在保证可靠性的前提下，尽量提高模型的可解性。若几个模型都具有可靠性，则当然可解性越强的模型越好。

一个模型可能同时适用于多个现实问题。这也就是我们要研究数学模型的主要原因之一。我们解决一个数学模型就相当于解决了一类问题。比如说，最短路径问题，可谓在现实生活中无处不在，例如在城市交通网中，求两点间的最短路径；网络流的模型也有着很高的实用价值，例如城市的供水网络能够有多大的流量。这些数学模型的解决使得许多实际问题迎刃而解。然而，数学模型的解决只是解决一个现实问题的一半，另一半就是如何将现实问题转化为一个已经解决的数学模型，即如何建模。建模在现实与抽象之间起着桥梁的作用。尤其在竞赛中，后者有时显得更为重要。

要能够建立数学模型，首先必须熟悉一些经典的数学模型及其算法。这是建模的基础，没有这个基础则根本谈不上建立什么数学模型。竞赛中许多问题最终都可以转化为经典的数学模型，因此必须掌握这些常见的模型。其次建立数学模型需要选手有把实际问题相互联系，类比的能力。数学模型之间必然存在着一些相同或相似。相互联系、类比是发掘这些信息的有效手段。

下面我们举几个例子，说明数学模型的建立与应用。

例 1 对 n*n 的方阵，每个小格可涂 m 种颜色，求在旋转操作下本质不同的图像个数。

分析 我们可以在方阵中分出互不重叠的长为 $[\frac{n+1}{2}]$，宽为 $[\frac{n}{2}]$ 的四个矩阵。当 n 为偶数时，恰好分完；当 n 为奇数时，剩下中心的一个格子，它在所有的旋转下都不动，所以它涂任何颜色都对其它格子没有影响。令 m 种颜色为 0~m-1，我们把矩阵中的每格的颜色所代表的数字顺次(左上角从左到右，从上到下；右上角从上到下，从右到左；……) 排成 m 进制数，然后就可以表示为一个十进制数，其取值范围为 $[0, m^{[\frac{n^2}{4}]}-1]$. (因为 $[\frac{n}{2}]*[\frac{n+1}{2}]=[\frac{n^2}{4}]$) 这样，我们就把一个方阵简化为 4 个整数。我们只要找到每一个等价类中左上角的数最大的那个方案(如果左上角相同，就顺时针方向顺次比较) .

进一步考虑，当左上角数为 i 时,令 $R=m^{[\frac{n^2}{4}]}$ $0 \le i \le R-1$，可分为下列的 4 类：其它三个整数均小于 i，共 i^3 个。右上角为 i，其它两个整数均小于 i，共 i^2 个。右上角、右下角为 i，左下角不大于 i，共 i+1 个。右下角为 i，其它两个整数均小于 i，且右上角的数不小于左下角的，共 i(i+1)/2 个。此处的讨论中，要注意“最大性原则”，并考虑边界情况。

因此，

$$L=\sum_{i=0}^{R-1}(i^3+i^2+i+1+\frac{1}{2}i(i+1))=\sum_{i=0}^{R-1}(i^3+\frac{3}{2}i^2+\frac{3}{2}i+1)$$

$$=\sum_{i=1}^{R}((i-1)^3+\frac{3}{2}(i-1)^2+\frac{3}{2}(i-1)+1)=\sum_{i=1}^{R}(i^3-\frac{3}{2}i^2+\frac{3}{2}i)$$

$$=\frac{1}{4}R^2(R+1)^2-\frac{3}{2}\times\frac{1}{6}R(R+1)(2R+1)+\frac{3}{2}\times\frac{1}{2}R(R+1)$$

$$=\frac{1}{4}(R^4+R^2+2R)$$

当 n 为奇数时，还要乘一个 m .

由此我们就巧妙地得到了一个公式。但是，我们应该看到要想到这个公式需要很高的智能和付出不少的时间。另一方面，这种方法只能对这道题有用而不能广泛地应用于一类试题，具有很大的不定性因素。因此，如果能掌握一种适用面广的原理，就会对解这一类题有很大的帮助。

下面我们就采用 Pólya 定理。我们可以分三步来解决这个问题。

1．确定置换群。在这里很明显只有 4 个置换：转 0 °、转 90 °、转 180 °、转 270 °。所以，置换群 G={转 0 °、转 90 °、转 180 °、转 270 °}。

2．计算循环节个数。首先，给每个格子顺次编号（1~n^2），再开一个二维数组记录置换后的状态。最后通过搜索计算每个置换下的循环节个数，效率为一次方级。

3．利用 Pólya 定理得到最后结果。 $L=\frac{1}{|G|}(m^{c(g_1)}+m^{c(g_2)}+...+m^{c(g_n)})$.

由此可见，作为组合数学理论一部分的 Pólya 定理在信息学竞赛中得到应用。我们还可以看到，数学竞赛题与信息学竞赛题解题思维的不同。数学竞赛要求得出最终结果，所有计算过程都由人完成；而信息学竞赛只要求提供一种具体的、解决通用性问题的方法，具体计算工作则由计算机完成。这样，利用计算机的高速计算能力，可以简化问题的考虑步骤，完成很多人类无法完成的工作。

例 2 方格取数问题。在 n*n 的方格纸中，每个格子中都有一个正整数。从中取出若干数，使得任意两个取出的数所在方格没有公共边，且取出的数的总和最大。

分析 本题可以抽象成一个图论问题。将每个格子看成一个顶点，填入的正整数为其权值；两个相邻格子之间连一条边。目标是求出一个顶点集合，使得其中任意两顶点没有边相连，且顶点的权值之和最大。这是图论中的著名问题，请读者思考。

例 3 有 n 根柱子，现在有任意正整数编号的球各一个，请你把尽量多的球放入这 n 根柱子中，满足：①放入球的顺序必须是 1,2,3…，且每次只能在某根柱子的最上面放球；②同一根柱子中，相邻两个球的编号和为完全平方数。请问，在 n 根柱子上最多能放多少个球？

分析 经简单的试验，可以发现一些规律。如果借助计算机程序，可以更快。令 $f(n)=[\frac{n^2}{2}+n-\frac{1}{2}]$，我们猜想 f(n)为正确答案。根据 n 的奇偶性分类讨论，若 n 根柱子放了 f(n)+1 个球，考虑最大的 n+1 个球中最小两球数字和 min、最大两球数字和 max，则 min、max 夹在两个相邻完全平方数之间，故 n+1 个球的任意两个不能在同一柱子上，矛盾。具体地说，当 n=2k+1 时，$f(n)=2k^2+4k+1$, $\min=4k^2+6k+3>(2k+1)^2$,$\max=4k^2+8k+1<(2k+2)^2$；当 n=2k 时，$f(n)=2k^2+2k-1$,$\min=4k^2+2k-1>(2k)^2$,$\max=4k^2+4k-3<(2k+1)^2$. 而使用贪心方法，即尽量往左边的柱子上放，依次寻找，能放则放，则利用数学归纳法，最后可放入 f(n)个球。

例 4 三维数码难题：给出 n*n*n 的立方体，恰好有一个方块为空，其他每个方块上写着 $[1,n^3-1]$ 的不同数字。每次可以把一个与空位相邻的方块移动到空位中。给出初始状态、终止状态，问是否可以实现。

分析 我们先考虑二维数码难题。我们知道，如果只交换两个数字的位置，则肯定无法

达到。对于此类变换问题，通常从奇偶性的角度考虑。

例 5 给定 n,m(n≥m)，从正 n 边形的顶点中选出 m 个，能组成多少个不同的 m 边形？若一个多边形可由另一个通过旋转、翻转、平移得到，则认为两个多边形相同。

分析 把每种旋转、翻转视作一种置换，看做若干个互不相交的循环，它的 V 值设为从置换中取出 m 个元素并使每个循环中的元素或同被取出，或都未被取出的方案数。最终答案即为所有置换的 V 值的平均数。

在顺时针旋转 a 格的置换中，循环的个数为(n,a), 每个循环的长度为 n/(n,a). 对于翻转，当 n 为奇数时，只有一种情况，即[n/2]个循环长为 2，一个长为 1，由组合数公式，$V=C_{[n/2]}^{[m/2]}$.当 n 为偶数时，有 n/2 个置换使得 n/2 个循环长度均为 2，此时若 m 为偶数，$V=C_{n/2}^{m/2}$,否则 V=0；有 n/2 个置换使得 n/2-1 个循环长为 2，两个循环长为 1，此时若 m 为偶数，$V=C_{n/2-1}^{m/2}+C_{n/2-1}^{m/2-1}$；否则 $V=2C_{n/2-1}^{(m-1)/2}$。故当 n 为偶数，m 为奇数时，$V=nC_{[n/2]-1}^{[m/2]}$；当 n 为奇数或 n、m 为偶数时，$V=nC_{[n/2]}^{[m/2]}$.所以旋转的所有置换的 V 值之和为 $\sum_{L|(m,n)} C_{n|L}^{m|L}\varphi(L)$. 此处φ表示欧拉函数，(a,b)表示 a、b 的最大公约数。

类似的，利用群论思想，还可以解决诸如化学中的同分异构体计数等需要判定并消除重复的组合计数问题。至于其中的原理，请参考有关组合数学、群论书籍。

吴文虎教授说，信息学提供给我们的是一种“计算思维”，就是在有计算机时如何运用程序加速我们的思考与工作。笔者在思考和研究数学问题时，经常想到编写程序解决问题，例如对于一类数列、集合的性质，用笔算试验可能过于麻烦，就可以使用计算机程序先对小规模数据进行试探，再从中寻找规律。实践证明，这种方法比直接从数学角度推导更节省时间。这也就是本文“试探法”的思想。

例 6 若四元集 E={a,b,c,d}中的四个数 a,b,c,d 能够分成和相等的两组，则称 E 为“平衡集”。试求集 M={1,2,...100}的平衡子集的个数。

分析 本题若用官方解答的方法，十分复杂。但若从计算机的角度考虑，则较为简单。事实上，本题就是要找到 1~100 中的 4 个不同数 a<b<c<d，使得 a+d=b+c，或 a+b+c=d. 若采用计算机枚举，将首先枚举第一个数，再枚举第二个数，再枚举第三个数，最后判断据此算出的第四个数是否符合题意。对两数之和相等的情况，只需找到可能的不同两数和 1+2=3～99+100=199，再考虑每个和共有多少种由不同数的组成方式 f_i，则 $C_{f_i}^2$ 即为从其中选出两组的方案数，最后对 f_i 求和。这样，两数和问题变成了对组合数的求和。对三数之和等于第四数，实际上等价于找三个不同的正整数，和不超过 100. 这样，分别对和为 1+2+3=6～100 进行讨论，当和为 s 时，枚举最大数[s/3]+1～s-3, 对（s-最大数）计算有多少种两数组成方式。这样，三数和问题变成了两次累加，先解决两数和固定，再解决三数和固定，最后解决三数和小于等于 100 问题. 最后将两种情况相加即可。

当然，对于这道题，还可以有更深入的探究。可以推导出在前 n 个正整数中找到 m 个互不相同的数，使其和为 s 的方法数。对于允许重复的情况，也有类似的讨论。这是组合数学中的一类重要计数问题。就本题而言，也可以使用减去重复的办法，先考虑总数，再减去重复情况数×重复次数。由此可见，学习信息学竞赛，能够启发数学竞赛的思路。

很多信息学竞赛题，事实上是数学竞赛题的推广。例如本文中提到的与互补数列有关的

双人取棋子游戏，就是全国信息学竞赛试题，要求判断是否能够达到。另外，求用 3^k 组成的数。有多道 IMO 试题后来被改编为信息学竞赛试题。例如第 34 届 IMO 第 3 题：在无限的方格棋盘上，有 n*n 个格放有棋子，每次移动必须跳过一个棋子进入空格，并“吃掉”被跳过的棋子，问最后是否有可能只剩一个棋子。在信息学竞赛中，该问题被推广为：对于 m*n（行列数可能不等）的棋子，最少剩下多少个棋子？当然，本题在信息学中不能算是很难的题目，由于编写程序只需得出结论而无需证明。但这足以说明信息学竞赛对数学竞赛的促进作用。有兴趣的同学可以参看信息学竞赛国家集训队论文，比较数学竞赛与信息学竞赛中数学问题，尤其是组合问题的难度高低与考察方面的不同。

4.2 计算数学基础

4.2.1 素数判断算法[1]

任务：生成 1-n 所有素数。

1.朴素算法（试除法）

```
int prime(int x)
{ int i,t=sqrt(x);
  for(i=2;i<=t;i++)
   if(x%i==0)
    return 0;
  return 1;
}
```

运行时间：当 n=100000 时，用时 63ms；当 n=1000000 时，用时 1328ms。

2.优化的试除法（记录素数）

```
int p[n]={2},num=1;
int prime(int x)
{ int i,t=sqrt(x);
  for(i=0;i<num;i++)
  { if(x%p[i]==0)
     return 0;
    if(p[i]>t)
     break;
  }
  p[num++]=x;
  return 1;
}
```

运行时间：当 n=100000 时，用时 40ms；当 n=1000000 时，用时 562ms。显然，与朴素的试除相比，带记忆化的试除可以减少对众多合数的检验，提高运行效率，尤其在大数据时优势明显。

[1] 本节引用自笔者《算法效率与程序优化》。

3.筛法

```
void prime()
{ int i,j;
  for(i=2;i<=n;i++)
   if(!p[i])
    for(j=i+i;j<=n;j+=i)
     p[j]=1;
}
```

运行时间：当 n=100000 时，用时 15ms；当 n=1000000 时，用时 265ms。显然，与上述两种试除算法相比，速度又提高了不少。缺点是：空间复杂度为 O(n)，当 n 过大时内存无法承受。之所以简单而经典的筛法能取得如此高的效率，是因为避免了缓慢的除法和根号。同时提醒我们，除法和取模的速度极慢，会导致常数因子较大，与加减乘不可同日而语，应当尽量减少。

4.Miller-rabbin 算法[1]

```
int test(int a,int b,int c)
{ int x=a,y=1;
  while(b!=0)
  { if(b&1)
     y=(y*x)%c;
    b=b>>1;
    x=(x*x)%c;
  }
  return y;
}
int prime(int a,int flag)
{ int i,r,d=a-1,t;
  while(!(d&1))
   d=d>>1;
  t=test(flag,d,a);
  while(d!=a-1&&t!=1&&t!=a-1)
  { d=d<<1;
    t=(t*t)%a;
  }
  return (t==a-1)||(d&1);
}
```

主程序：

```
s=clock();
    for(i=3;i<=n;i+=2)
     if(prime(i,2)&&prime(i,7)&&prime(i,61));
    t=clock();
```

注意：在 longint 范围内，只需测试 2、7、61，这也是信息学竞赛中最常用的。在 int64

[1] 改编自 Matrix67 博客 http://www.matrix67.com/blog

（1E16）范围内，只需测试 2、3、7、61、24251，并排除 46 856 248 255 981 这个强伪素数。

当 n=100000 时，运行时间：46ms；当 n=1000000 时，运行时间：500ms。当 n=10000000 时，运行时间：5063ms。从表面上看，这种高效算法似乎比朴素的筛法慢，其实筛法的最大缺点是占用内存过多以及不能判断单个素数。记忆化的试除法似乎差不多，但内存用量同样较大，不能判断单个素数。普通的试除则效率过低。所以，miller-rabbin 算法在只需要判断少数离散的大素数时，是十分高效的算法。

4.2.2 欧几里德算法[1]

1.最大公约数与最小公倍数

```
int gcd(int a,int b)
{ int t;
  if(a<b)
  { t=a;
    a=b;
    b=t;
  }
  while(b!=0)
  { t=b;
    b=a%b;
    a=t;
  }
  return a;
}
```

主程序：

最大公约数 `gcd(a,b)`

最小公倍数 `a*b/gcd(a,b)`

2.扩展欧几里德算法[2]

这个算法可以由上面的“欧几里德辗转相除法”推广而来。两个算法的时间复杂度均为 O(logn). 目的是求整数 x,y 使得 ax+by=d. 这里用了迭代法的思想。

```
int extendgcd(int a,int b)
{ int t;
  if(b==0)
  { x=1;y=0;
    return a;
  }
  t=x;x=y;
  y=t-(a/b)*y;
```

[1] 本部分主要参考了刘汝佳、黄亮《算法艺术与信息学竞赛》。从本文“常数优化”的角度考虑，还可以定义变量，省去一些不必要的除法、取模运算。由于复杂度仅为 O(logn)，必定不是程序的速控步，且优化的效果不明显，这里没有进一步优化。

[2] 算法中的 x,y 表示待求 ax+by=d 的取值，是全局变量。a,b 是输入条件。

```
  return extendgcd(b,a%b);
}
```

3.解线性同余方程[1]

这个算法是“扩展欧几里德算法”的应用[2]。

```
int modequation(int a,int b,int n)
{ int d;
  d=extendgcd(a,n,x,y);
  if(b%d!=0)
   return -1;
  e=x*(b/d)%n;
  for(i=0;i<d;i++)
   ans[i]=(e+i*n/d)%n;
}
```

下面来看一道有关最小公倍数的例题：找出整数 1,2,...n 的最小公倍数，给出 mod M 的结果即可。[3]

【算法一】

最朴素的想法，使用求 n 个数的最小公倍数的经典方法，先求出 1、2 的最小公倍数 2，再求出 2、3 的最小公倍数 6，再求出 6、4 的最小公倍数 12，继续求 12、5 的公倍数，以此类推，直到 n。这样的方法显然需要使用高精度运算具体求出每个数，对于 n 较小时可行，但对于原题 100000 的数据范围，显然是小巫见大巫了。

【算法二】

追求 n 个数最小公倍数的本质：对于 1,2,...n 的质因子分解式，最小公倍数即为每个质因子次幂数的最大值。那么只需要对 1,2,...n 进行质因数分解，求出每个质因数的最高幂次，使用快速幂的方法计算 MOD M 的值，再将不同的余数取模相乘，得到结果。

但进一步考虑，从本题 100000 的规模出发，有必要采用快速幂吗？只有不超过 10 个质因子相乘，使用二分算法显然是“大材小用”。

【源代码】

```
#include<stdio.h>
#include<math.h>
#define M 987654321
int n,tot=0,p[100000]={0,0,1},q[100000]={0},a[100000]={0},ans=1;
void prime()
{ int i,j,t=(int)sqrt(n)+1;
  for(i=2;i<=n;i++)
   p[i]=1;
  for(i=2;i<t;i++)
   if(p[i])
   { q[tot++]=i;
     j=i+i;
```

[1] 算法中的 a,b,n 表示 ax≡b (mod n)，ans 数组存放的是所有可能的 x 值。

[2] 事实上等价于整数方程 ax-ny=b。本算法求出了所有解，如果不需要可以只计算第一组。

[3] 来源：2008 年第 2 期《NOI 专刊》竞赛跑道栏目。

```
    while(j<=n)
    { p[j]=0;
      j+=i;
    }
  }
  for(i=t;i<=n;i++)
   if(p[i])
    q[tot++]=i;
}
int main()
{ int i,j,t,st,tmp;
  scanf("%d",&n);
  prime();
  for(i=2;i<=n;i++)
  { t=i;
    for(j=0;t>1;j++)
    { tmp=0;
      while(t%q[j]==0)
      { tmp++;
        t/=q[j];
      }
      if(tmp>a[j])
       a[j]=tmp;
    }
  }
  for(i=0;i<tot;i++)
   for(j=0;j<a[i];j++)
   { ans*=q[i];
     ans%=M;
   }
  printf("%d\n",ans);
  getch();
}
```

4.2.3 函数类算法[1]

1.快速求平方根

```
double InvSqrt(double x)
{
double xhalf = 0.5f*x;
int i = *(int*)&x;          // 求浮点位
```

[1] 本节改编自笔者《算法效率与程序优化》，并参考了相关数论书籍。

```
i = 0x5f375a86- (i>>1);    // 得到 y0 的初始预测值
x = *(float*)&i;           // 变回浮点类型
x = x*(1.5f-xhalf*x*x);    //牛顿迭代法，减少误差
return x;
}
```

2.方程求根

```
#include<stdio.h>
int n,tot=0;
double a[100]={0},delta=0.000001;
double calc(double x)
{ double ans=0,pow=1;
  int i;
  for(i=0;i<=n;i++)
  { ans+=pow*a[i];
    pow*=x;
  }
  return ans;
}
void root(double from,double to)
{ double mid=(from+to)>>1;
  double f=calc(from),t=calc(to),m=calc(mid);
  if(m>-delta&&m<delta)
  { printf("Root %d = %lf\n",++tot,mid);
    return;
  }
  if(m>0)
  { if(f<0)
     root(from,mid);
    if(t<0)
     root(mid,to);
  }
  else
  { if(f>0)
     root(from,mid);
    if(t>0)
     root(mid,to);
  }
}
main()
{ int i;
  printf("方程求根。\n 输入最高次方数\n");
  scanf("%d",&n);
  printf("输入从最高次项开始各项系数。\n");
```

```
  for(i=n;i>=0;i--)
   scanf("%lf",&a[i]);
  root(-2147483647,2147483647);
  if(tot==0)
   printf("此方程无解！\n");
  else printf("此方程在(-2^31,2^31)内共 %d 个根。\n",tot);
  getch();
}
```

3.Hash 排序、查找

```
#define MOD 999997  //As a big prime
#define MAX 10000   //As the max length of the string
int ELFhash(char *key)
{ unsigned long h=0;
  while(*key)
  { h=(h<<4)+*key++;
    unsigned long g=h&0Xf0000000L;
    if(g) h^=g>>24;
    h&=~g;
  }
  return h%MOD;
}
```

Hash 排序，就是先对读入的字符串求 Hash 值。第一种方案是，再对 Hash 值进行快速排序，最后应用二分搜索查找。此时无需 MOD 大质数。第二种方案是，将 Hash 值 MOD 后，放入 Hash 表中，并用开散列等方法处理冲突。显然，第二种方案更优，但编程复杂度也较高。

4.3 组合计数[1]

组合计数问题，作为信息学中的一类重要问题，在信息学竞赛中有着重要的应用。

组合计数，顾名思义，就是简化运算步骤，减少处理次数，不必要逐一求出所有可行解，只要求输出最终的结果数，甚至是 MOD M 的结果。

首先看一道利用组合计数避免枚举的例题：单色三角形问题[2]

空间里有 n 个点，任意三点不共线。每两点之间都用一条红色或黑色线段连接。如果一个三角形的三条边同色，则称这个三角形是单色三角形。对于给定的红色线段的列表，找出单色三角形的个数。输入点数 n、红色边数 m 以及这 m 条红色的边所连接的顶点标号，输出单色三角形个数 R。3<=n<=1000，0<=m<=250000。

【初步分析】

很自然地，我们想到了如下算法：用一个 1000*1000 的数组记录每两个点之间边的颜色，

[1] 参考了符文杰《Polya 原理及其应用》。

[2] 引自许智磊《浅谈补集转化思想在统计问题中的应用》。

然后枚举所有的三角形（这是通过枚举三个顶点实现的），判断它的三条边是否同色，如果同色则累加到总数 R 中（当然，初始时 R 为 0）。

这个算法怎么样呢？姑且不论它非常奢侈地需要一个 1M 的大数组，它的时间复杂度已经高达 O(C(n,3))，也就是 $O(n^3)$的级别。对于 n 最大达到 1000 的本题来说，实在不能说是个好算法。

那些常规的技巧能不能用到本题上呢？离散化和极大化看起来跟本题毫不沾边。由于本题的限制条件非常少，也不是求最值型的计数问题，所以二分和事件表看来也用不上。

看来，想要循规蹈矩地解决这题是不可能了，我们需要全新的思路！

【深入思考】

稍稍进一步分析就会发现，本题中单色三角形的个数将是非常多的，所以一切需要枚举出每一个单色三角形的方法都是不可能高效的。

单纯的枚举不可以，那么组合计数是否可行呢？从总体上进行组合计数很难想到，那么我们尝试枚举每一个点，设法找到一个组合公式来计算以这个点为顶点的单色三角形的个数。

这样似乎已经触及到问题的本质了，因为利用组合公式进行计算是非常高效的。但是仔细分析后可以发现，这个组合公式是很难找到的，因为对于枚举确定的点 A，以 A 为一个顶点的单色三角形 ABC 不仅要满足边 AB 和边 AC 同色，而且边 BC 也要和 AB、AC 边同色，于是不可能仅仅通过枚举一个顶点 A 就可以确定单色三角形。

经过上面的分析，我们得出枚举＋组合计数有可能是正确的解法，但是在组合公式的构造上我们遇到了障碍。这个障碍的本质是：

从一个顶点 A 出发的两条同色的边 AB、AC 并不能确定一个单色三角形 ABC，因为 BC 边有可能不同色。也就是说，我们无法在从同一个顶点出发的某两条边与所有的单色三角形之间建立一种确定的对应关系。

【补集转化】

让我们换一个角度，从反面来看问题：因为每两点都有边连接，所以每三个点都可以组成一个三角形（单色或非单色的），那么所有的三角形数 S=C(n,3)=n*(n-1)*(n-2)/6。又因为单色三角形数 R 加上非单色三角形数 T 就等于 S，所以如果我们可以求出 T，那么显然，R=S－T。于是原问题就等价于：怎样高效地求出 T。

纯枚举的算法想都不用想就被排除，那么在上面分析中夭折的枚举＋组合计数的算法又怎样呢？这个算法原先的障碍是无法在“某两条边”与“单色三角形”之间建立确定的对应关系。那么有公共顶点的某两条边与非单色三角形之间是否有着确定的关系呢？对了！这种关系是明显的：

非单色三角形的三条边，共有红黑两种颜色，也就是说，只能是两条边同色，另一条边异色。假设同色的两条边顶点为 A，另外两个顶点为 B 和 C，则从 B 点一定引出两条不同色的边 BA 和 BC，同样，从 C 点引出两条不同色的边 CA 和 CB。

这样，一个非单色三角形对应着两对“有公共顶点的异色边”。

另一方面，如果从一个顶点 B 引出两条异色的边 BA、BC，则无论 AC 边是何种颜色，三角形 ABC 都只能是一个非单色三角形。也就是说，一对“有公共顶点的异色边”对应着一个非单色三角形。

很明显，我们要求的非单色三角形数 T 就等于所有“有公共顶点的异色边”的总对数 Q 的一半。而这个总对数是很好求的：每个顶点有 n-1 条边，根据输入的信息可以知道每个顶点 i 的红边数 E[i]，那么其黑边数就是 n-1-E[i]。枚举顶点 A，则根据乘法原理，以 A 为公

共顶点的异色边的对数就是 E[i]*(n-1-E[i])。所以 $Q=\sum_{i=1}^{n}E[i]*(n-1-E[i])$

求出 Q 之后，答案 R=S－T=n*(n-1)*(n-2)/6-Q/2。这个算法的时间复杂度仅为 O(m+n)，空间复杂度是 O(n)，非常优秀。

【小结】

在这个例子中，我们经历了如下的思维过程：

（1）直观的枚举算法无法实现，常用的技巧没有用武之地

（2）考虑枚举＋组合计数

（3）正面分析，组合公式很难构造

（4）从反面思考（补集转化思想），得到突破

补集转化思想在其中起着至关重要的作用：通过补集转化，我们才能够在原来无法联系起来的“边”和“三角形”之间建立起确定的关系，并以此构造出组合计数的公式。这样，就由单纯的枚举算法改为了枚举+组合计数的算法，大大降低了时间和空间复杂度。在这里，补集转化思想的作用体现在为找到一个本质上不同的算法创造了条件。

【评注】

以上部分的讨论，是许智磊《浅谈补集转化思想在统计问题中的应用》中的原文。事实上，如果学习过数学竞赛的有关知识，探索的历程不需要如此繁杂。

在数学竞赛中，这样的方法被称为利用中间量“算两次”。算两次主要针对下面的量：

（1）**角** 当估计的量与同色三角形有关时，可计算同色角、异色角，这是一种“减元”策略，本题应用的正是这样的方法。

（2）**子集** 若各子集满足两两之交<=r,则计算 r+1 元子集，此时对不同的 ai,aj，r+1 元子集两两互异，否则交集>r.

（3）**对子** 将具有特殊关系的 2 个元素配对，用两种途径计算对子总数。

（4）**次数** 某种元素出现的总次数。

事实上，早在第 36 届 IMO 预选题中，就出现了本题的雏形：平面上有 18 个点，任意三点不共线，每两点用红或蓝色线段连接，已知某点 A 引出的红色线段为奇数条，且其余的 17 点引出的红色线段数互不相等。求图中红色三角形的个数、恰有两边为红色的三角形个数。

作为一道数学试题，直接数是不现实的，因此采用了本题所述的方法。解得红色三角形 204 个，两边红色三角形 240 个，读者可以一试。

由此可见，数学竞赛中的某些思想，在信息学竞赛中还是大有用场的。

下面讨论组合计数中 Polya 定理的应用。

先看一道例题：n×n 的方阵，每个小格可涂 m 种颜色，求在旋转操作下本质不同的解的总数。

Polya 定理 设 G 是 p 个对象的一个置换群，用 m 种颜色涂染 p 个对象，则不同染色方案为：$L=\frac{1}{|G|}(m^{c(g_1)}+m^{c(g_2)}+\ldots+m^{c(g_s)})$.

我们从简单的角度理解这个问题。n 个元素 1,2,…,n 之间的一个置换 $\begin{pmatrix}1 & 2 & \cdots & n\\ a_1 & a_2 & \cdots & a_n\end{pmatrix}$ 表示 1 被 1 到 n 中的某个数 a1 取代，2 被 1 到 n 中的某个数 a2 取代，

直到 n 被 1 到 n 中的某个数 an 取代，且 a1,a2,…,an 互不相同。所谓循环，就是一组数 $(a_1a_2\cdots a_n)=\begin{pmatrix}a_1 & a_2 & \cdots & a_{n-1} & a_n\\ a_2 & a_3 & \cdots & a_n & a_1\end{pmatrix}$，每个置换都可以表示成若干个互不相交的循环。c(gi)就是互不相交的循环个数，叫做循环节数。例如 $\begin{pmatrix}1&2&3&4&5\\3&5&1&4&2\end{pmatrix}=(13)(25)(4)$，则其循环节数为 3.

对于每个置换，由于各个循环之间互相独立，每个循环都能随意染一种颜色，而确定一个置换需要同时确定 c(g)的循环，是分步事件，共有 $m^{c(g)}$ 种染法。而循环内部由于组内数的关系确定，当确定第一个元素所染的颜色后，只有一种可能。故对每个置换，染色方案数为 $m^{c(g)}$。

而对于不同的置换之间，由于互相独立，而每个置换都对应一种情况，因此是分类时间，使用加法原理，共有 $m^{c(g_1)}+m^{c(g_2)}+...+m^{c(g_s)}$ 种不同方法。

最后考虑到置换的重复问题，由于不同的置换之间互为等价类，每种情况都计算了|G|次，应该除以置换的个数，得到 $L=\frac{1}{|G|}(m^{c(g_1)}+m^{c(g_2)}+...+m^{c(g_s)})$.

具体编程时，首先确定置换群。在本题中，只有 4 个置换：转 0,90,180,270 度。分类时要注意不重不漏，并注意等价性。

然后计算循环节数，将每个格子顺次编号（可以使用二维数组的指针），通过搜索计算每个循环节下的置换个数。

最后代入公式，求得结果。

根据 Polya 定理编写的程序：

```
#include<stdio.h>
#define maxn 100
int a[maxn][maxn]={0},b[maxn][maxn]={0},m,n;
void turn()//转过 90 度
{ int i,j;
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    a[j][n+1-i]=b[i][j];
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    b[i][j]=a[i][j];//使用临时数组 b 辅助矩阵旋转
}
int calc()//计算当前状态下个数，类似 floodfill 求连通区域数
{ int i,j,i1,j1,k=0,p,pow=1;
  for(i=1;i<=n;i++)//枚举当前格子
   for(j=1;j<=n;j++)
    if(a[i][j]>0)
```

```
    { k++;  //k 为循环节数，即互不相交的循环个数
      i1=(a[i][j]-1)/n;
      j1=(a[i][j]-1)%n+1;
      a[i][j]=0; //下面把当前循环的格子全部标记，一个循环找到
      while(a[i1][j1]>0)//如果没有访问过，就在下一步访问它对应的格子
      { p=a[i1][j1];
        a[i1][j1]=0;
        i1=(p-1)/n+1;
        j1=(p-1)%n+1;
      }
    } //事实上，上述函数可以更紧凑，首先赋值 i1,j1,然后用一个 while
  for(i=1;i<=k;i++)//计算次幂数 m^c(g)
   pow*=m;
  return pow;
}
int main()
{ int i,j,ans=0;
  scanf("%d%d",&m,&n);
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    b[i][j]=a[i][j]=(i-1)*n+j;//b 是转动时的临时存储数组
  ans+=calc();//计算转 0、90、180、270 度的情况
  turn();
  ans+=calc();
  turn();
  ans+=calc();
  turn();
  ans+=calc();
  printf("%d\n",ans/4);//最后除以循环节数
  return 0;
}
```

再看一道图论中的计数问题[1]：

竞赛图中一些边给定，另一些边需要定向。目标：定向后长度为 3 的环最多。

这个问题描述起来很简单，但仔细考虑，就会发现绝非易事。我们发现“已经给定的一些边”是个很麻烦的事，不利于我们考虑问题。因此

4.4 计算几何算法

计算几何，顾名思义就是使用计算机解决几何问题。但这会遇到很大的麻烦：平面几何

[1] Winter Camp 2007 Problem 3

中边角关系错综复杂，需要强烈的几何直观，这是计算机难以做到的；解析几何是代数化的，但浮点误差过大，不适用于计算机处理。那么“计算几何”就是一种折中的办法——即保留了解析几何的代数化处理技巧，又将复杂的方程求解模糊化，像平面几何一样只追求位置关系的判断和处理。就像数学竞赛中的三角法由于广收平面、解析两家之长而广泛用于几何问题的求解，计算几何在信息学竞赛中有关几何问题的处理中，往往起到举足轻重的作用。

在信息学竞赛中，历年来出现的几何问题并不多，所以计算几何学没有必要研究过深。本文仅就信息学竞赛中最常用的两个经典问题：多边形求交或并的面积、求点集的凸包[1]进行讨论，并启发读者养成计算几何的思维方式。

做计算几何试题，就像做解析几何试题，要遵循“**平面几何优先**”的原则。例如“**线段**[2]”一题：

给定两个圆各自的圆心坐标和半径长。过其中一个交点作直线，该直线与圆的另外两个交点分别为 A、B。线段 AB 最长是多少？

如果盲目采用解析几何办法，先求出 AB 长度的表达式，再求出该函数的最值，不要说计算机，人类完成都很困难。我们如果仔细观察，进行数学推导，就会发现一个奇妙的性质：AB 长度的最大值恰好等于圆心距的 2 倍，此时 AB 与圆心连线平行。（证明略）

```
#include<stdio.h>
#include<math.h>
main()
{ double x1,y1,r1,x2,y2,r2;
  scanf("%lf%lf%lf%lf%lf%lf",&x1,&y1,&r1,&x2,&y2,&r2);
  printf("%.6lf",sqrt((x1-x2)*(x1-x2)+(y1-y2)*(y1-y2))*2);
}
```

由此可见，遇到几何问题，不要盲目下手，而要首先观察平面几何特征。

再看一道题“**轰炸**[3]”：给出平面上的一些整点，求一条穿过尽可能多的整点的直线。

首先考虑，枚举每两个整点，计算它们连线的斜率；再枚举不同的第三个点，考察三点是否共线。满足三点共线的点的最多数目即为所求。

在处理时，考虑到浮点运算效率低，可以用乘法代替除法。实测结果表明，用浮点除法+误差估计的方法只能通过 7 个测试点。

```
#include<stdio.h>
main()
{ int n,i,j,k,a[1000],b[1000],bij,aij,max=0,now;
  scanf("%d",&n);
  for(i=0;i<n;i++)
   scanf("%d%d",&a[i],&b[i]);
  for(i=0;i<n;i++)
   for(j=i+1;j<n;j++)
   { now=2;
     aij=a[i]-a[j];
     bij=b[i]-b[j];
     for(k=0;k<n;k++)
```

[1] 这两个问题主要参考了刘汝佳、黄亮《算法艺术与信息学竞赛》。
[2] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=358
[3] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=150

```
      if(i!=k&&j!=k&&bij*(a[i]-a[k])==(b[i]-b[k])*aij)
       now++;
     if(now>max)
      max=now;
   }
  printf("%d",max);
}
```

测试结果，上述程序通过 9 个测试数据，用时 1454ms。通过的最慢一个点 453ms。

然后考虑，乘法所占的时间较多，可以利用常数优化技巧，将变量设出，减少重复计算。这道题同时提供了判断三点共线较为高效的算法。

```
#include<stdio.h>
main()
{ int n,i,j,k,x[1000],y[1000],a,b,c,max=0,now;
  scanf("%d",&n);
  for(i=0;i<n;i++)
   scanf("%d%d",&x[i],&y[i]);
  for(i=0;i<n;i++)
   for(j=i+1;j<n;j++)
   { now=0;
     a=y[j]-y[i]; //a,b,c 首先计算出循环中的不变量
     b=x[i]-x[j];
     c=b*y[i]+a*x[i];
     for(k=0;k<n;k++)
      if(a*x[k]+b*y[k]==c)//判断三点共直线
       now++;
     if(now>max)
      max=now;
   }
  printf("%d",max);
}
```

测试结果，上述程序用时 1782ms，通过所有 10 个测试数据。其中第一个程序未通过的测试点用了 688ms。上个程序用 453ms 的测试点，本程序用了 297ms。

由此可见，算法的常数优化在程序设计中有重要应用，即使 n=700，$O(n^3)$的算法还是有可能 AC 的。事实上，本题有 $O(n^2\log n)$的算法，但编程复杂度显然较高。

下面我们来解决一类重要的问题：线性规划。

现有 n 种原料，每种原料分别有 $s_1, s_2, \ldots s_n$ 吨，要生产 m 种产品，生产一吨产品 i 需要分别消耗 1,2,...n 种原料 $a_{i1}, a_{i2}, a_{i3}, \ldots a_{in}$ 吨，其收益为 w_i，问如何安排生产才能使收益最大？

用数学化的语言描述，就是：

$$\begin{cases} a_{11}x_1+a_{12}x_2+\ldots+a_{1m}x_m \le s_1 \\ a_{21}x_1+a_{22}x_2+\ldots+a_{2m}x_m \le s_2 \\ \ldots \\ a_{n1}x_1+a_{n2}x_2+\ldots+a_{nm}x_m \le s_n \end{cases}$$

最大化 $f(x)=w_1x_1+w_2x_2+\ldots+w_mx_m$.

我们推测，最大值应该使至少一个等号成立，因为如果没有等号成立，可以适当增大某个 x_i，使得至少一个等号成立。由于变量的非负性，f(x)必然增大。

进一步推测，当 n<=m 时，最大值应满足所有等号；当 n>m 时，最大值应满足至少 m 个等号。这是由于，假设成立了 k 个等号，固定 k 个变量不变，让 m-k 个变量调整，可以保证这 k 个等号成立，同时成立一些新的等号，使某些 x_i 增大，使得 f(x)更大。

由此我们得到一个朴素[1]的算法：如果 n<=m，直接解出这个方程组，采用高斯消元法。如果 n>m，枚举 n 个方程组中的 m 个，这可以用字典序的办法用递归实现。再对每 m 个方程组求解，得到 C_n^m 个不同的最大值，取其最大者即可。

当 n 很大时，这个算法显然太慢了。下面的代码称为“**单纯形法**[2]”，就是沿着 n 维空间的“边”不断向最优的方向“爬”，直到最优值的“定点”，类似图论中的“松弛”操作。

```
pivot(l, e){//转轴操作，就是“爬”的过程
    tb[e] = b[l]/A[l][e];
    tA[e][l] = 1/A[l][e];
    for（i 属于 N 且 i 不等于 e）
        ta[e][i] = A[l][i]/A[l][e];
    //上述部分得到了我们需要的以 xe 为左侧变量的等式
    for(i 属于 B 且 i 不等于 l)
        tb[i] = b[i]-A[i][e]*tb[e];
        For(j 属于 N 且 j 不等于 e)
            tA[i][j] = A[i][j]-A[i][e]*tA[e][j];
    //上述部分将等式带入了其它的等式当中
    tv = v+tb[e]*c[e];
    tc[l] = -tA[e][l]*c[e];
    for(i 属于 N 且 i 不等于 e)
        tc[i] = c[i]-c[e]*tA[e][i];
    //上述部分将等式带入了目标函数
    v = tv;
    N = N 除去 e 加上 l;
    B = B 除去 l 加上 e;
    A = tA;
    b = tb;
    c = tc;
}
```

[1] 其实并不“朴素”，很多人想不到任何有效的解决方案。

[2] 伪代码来自李宇骞《浅谈信息学竞赛中的线性规划》。

```
opt(){//主函数
    while (1) {
        if (所有的 i 属于 N 都有 c[i] <= 0)
            return 已经找到了解;
        else 选取一个 e 使得 c[e] > 0;
        delta = 无穷大;
        for(i 属于 B)
            if (A[i][e] > 0 && delta > b[i]/A[i][e])
                    delta = b[i]/A[i][e];
                l = i;
        if (delta == 无穷大) return 最优值无穷大;
        else pivot(l, e);
  }
}
init(){//预处理
    找到 b 中最小的元素 b[l];
    if (b[l] >= 0) return;//原问题已经满足 b 都大于等于 0
    origC = c;//记录下原来的目标函数，以便还原
    N 加上 0。
    for(i 属于 B)
        A[i][0] = -1;
    for(i 属于 N)
        c[i] = 0;
    c[0] = -1;
    //辅助问题已经构造完毕
    pivot(l, 0);//辅助问题的 b 现在都为非负数了
    opt;//把辅助问题解出来
    if (v < 0) return 无解; //辅助问题的最优值小于 0
    把 0 除去; //注意 0 在 B 中的情况
    c = origC;
    for(i 满足 c[i] > 0 且 i 属于 B)
        v = v+c[i]*b[i];
        for(j 属于 N)
            c[j] = c[j]-A[i][j]*c[i];
        c[i] = 0;
    //c 已经被还原
    return 已经成功初始化;
}
```

以上程序从理论上分析，是非多项式复杂度的；但实际运行效果，比多项式复杂度的“椭球算法”更优。这是由于在“爬”的过程中始终向较优的方向逐步调整，实际经过的节点数远少于总可能数。这可以成为本文“逐步调整法”的范例。这再次告诉我们：算法的理论时间复杂度不能说明一切，O 中隐藏的常数才是王道。

上面的朴素算法中提到了高斯消元法，本文不妨粘贴一段代码[1]，以比较各种高斯消元法的异同。事实上，最朴素的高斯消元法就是“代入消元法”，即第 i 步求出 x_i 的关于后面变量的表达式，再将求出的变量代入后面的所有方程，这样便消去了一个变量；最后只剩下一个变量和一个常数，解出 x_n，然后再一一回代，逐步求出 n 个变量。代码见 guss1 函数。

无论何种方法，都要注意无解、无穷多组解的情况。能求出精确解的方法，复杂度均为 $O(n^3)$。迭代法每次复杂度为 $O(n^2)$，但迭代次数可由精度要求决定。

关于迭代法与高斯消元法的比较，主要在于：高斯消元法是理论上精确的算法，但在实际编程中，由于浮点误差积累，已经到了不可忽略的程度。迭代法是理论上模糊的算法，但当迭代次数足够多时，可以达到较高的精度，由于不存在误差积累问题。这里的“迭代法”是本文“逐步调整法”的范例。

实际选择算法时，如果问题的规模较大，例如超过了 100 个方程、100 个未知数的大型方程组，采用迭代法以提高精度，迭代次数由问题的精度要求决定，可以进行试验；而较小型的方程组，采用高斯消元法以提高速度，降低编程复杂度。

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>
void guss1(float **a,int n) //有回代的高斯消元法
{ int i,j,k;float c;
for(k=0;k<n-1;k++) {
c=a[k][k];
for(j=k;j<=n;j++) a[k][j]/=c;
for(i=k+1;i<n;i++) {
c=a[i][k];
for(j=k;j<=n;j++) a[i][j]-=c*a[k][j];
}
}
a[n-1][n]/=a[n-1][n-1];
for(k=n-2;k>=0;k--)
for(j=k+1;j<n;j++) a[k][n]-=a[k][j]*a[j][n];
}
void guss2(float **a,int n) //无回代的高斯消元法
{ int i,j,k;float c;
for(k=0;k<n;k++) {
c=a[k][k];
for(j=k;j<=n;j++) a[k][j]/=c;
for(i=0;i<n;i++) {
if(i==k) continue;
c=a[i][k];
for(j=k;j<=n;j++) a[i][j]-=c*a[k][j];
}
}
a[n-1][n]/=a[n-1][n-1];
for(k=n-2;k>=0;k--)
```

[1] http://zhidao.baidu.com/question/59869140.html

```
for(j=k+1;j<n;j++) a[k][n]-=a[k][j]*a[j][n];
}

void guss3(float **a,int n) //全主元高斯消元法
{
}
void diedai1(float **a,int n)
{
float *x1,*x2,c; int i,j,t,k;
x1=(float *)malloc(n*sizeof(float));
x2=(float *)malloc(n*sizeof(float));
for(i=0;i<n;i++) {
c=a[i][i];
for(j=0;j<n;j++) a[i][j]/=-c;
a[i][n]/=c;
a[i][i]=0;
}
for(i=0;i<n;i++)x2[i]=0;
do {
for(i=0;i<n;i++) x1[i]=x2[i];
for(i=0;i<n;i++) {
x2[i]=a[i][n];
for(j=0;j<n;j++) x2[i]+=a[i][j]*x1[j];
}
c=0;
for(i=0;i<n;i++) if(fabs(x2[i]-x1[i])>c) c=fabs(x2[i]-x1[i]);
} while(c>=0.0001);
for(i=0;i<n;i++) a[i][n]=x2[i];

}

void diedai2 (float **a,int n)
{
}
void main()
{ int i,j,k;
float
a[4][5]={{8,2,1,2.5,1.5},{1,8,-0.5,2,-3},{1.5,2,8,-1,-4.5},{1,0.5
,0.7,8,3.2}};
float aa[4][5],*b[4]={aa[0],aa[1],aa[2],aa[3]};
while(1) {
printf("**********************************************************
*****\n");
printf("* 1. 有回代高斯消元法 2. 无回代高斯消元法 *\n");
```

```
printf("* 3. 全主元高斯消元法 4. 简单迭代法 *\n");
printf("* 5. 高斯-赛得尔迭代法 6. 退 出 *\n");
printf("**********************************************************
*****\n");
scanf("%d",&k); if(k==6) break;
for(i=0;i<4;i++)
for(j=0;j<5;j++) aa[i][j]=a[i][j];
switch(k) {
case 1: guss1(b,4); break;
case 2: guss2(b,4); break;
//case 3: guss3(b,4); break;
case 4: diedai1(b,4); break;
//case 5: diedai2(b,4); break;
}
for(i=0;i<4;i++) printf("%.4f\t",aa[i][4]);
printf("\n");
}
}
```

4.5 博弈论中的数学

4.5.1 纳什均衡[1]

信息学竞赛题中的“博弈论”，作为数学的重要分支，在经济、生活中有重要应用。

有人可能会想，为什么社会总是有不平等存在？而在不平等之后，为什么又总是趋向和谐与稳定？纳什均衡理论，作为博弈论的重要思想，将引领我们获得社会发展的新启示。

我们先来看一个简单的博弈游戏。两个囚徒被关在监狱中，并彼此分开，无法接触。检察官告诉他们，如果两人都坦白事实，就都判刑 10 年；若只有一人坦白，则坦白者无罪释放，不坦白者重判 15 年；若两人都不坦白，都判刑 1 年。若两个囚犯都足够聪明，则会采取什么样的策略，才能使自己（不考虑另一个囚徒）尽可能判刑减少？

我曾经用这个问题考验过很多人，他们的想法不尽相同，您也应该先分析一下。这里我们给出一种简单的考虑方法。若对方坦白，则自己坦白判刑 10 年，比不坦白的 15 年好；若对方不坦白，则自己坦白判刑 0 年，比不坦白的 1 年好。无论如何，坦白都比不坦白更占优势。于是两人都会选择坦白，最终都被判刑 10 年。但从整体的角度考虑，显然，两人选择不坦白是较优的。这就是著名的“囚徒困境”问题。

这种矛盾在社会生活中是常见的。这集中体现了“个人利益”与“集体利益”的冲突。也就是说，个人善以待人，遇到另一个善良的人，合作的效率较高；两个斤斤计较的人，斗争导致效率较低。若好人遇上坏人，则坏人受益极大，而好人也会损失极大。对比以上囚徒困境

[1] 摘自笔者《从博弈游戏到社会和谐》，http://boj.pp.ru/message/nash.htm

问题，你会发现这个模型在众多社会问题中的普遍存在。于是，根据我们的选择，大家都会做“坏人”，这样社会风气就无法保证，而且整体效率也会降低。

为了更加深入的分析上述问题，我们从博弈论的角度，引入“纳什均衡”的概念。纳什均衡，指的是在一个博弈的过程中，存在这样一种策略，使得在外界条件、他人决策变化的情况下，任何人都没有理由打破这种均衡的状态。这个说法可能较为抽象，从囚徒困境问题上说，两人都选择坦白，就是该游戏的“纳什均衡点”，由于选择“不坦白”总比选择坦白要差。

还有一种博弈游戏，叫做“智猪博弈”。两只猪被关在一个猪圈中，猪圈的一侧有一个踏板，另一侧是食槽。任何一只猪去踩踏板，都会立即在食槽中喂食。若大猪踩了踏板，小猪会先开始吃，等到大猪赶回来时，还有一半食料留给大猪；若小猪踩踏板，大猪在小猪回来时刚好将食吃完。试问，两只猪那个占优势？

多数人会不假思索的说，是大猪。由于大猪跑的快，每次总比小猪占优势。但这样想就错了。我们先来看小猪，它即使踩了踏板也得不到食，还不如不踩，这样省力。大猪知道小猪不会去踩踏板，于是为了吃到食，就被迫去踩踏板，为了那一半食物。这样，博弈的结果是，大猪费力跑过去踩踏板，得到一半食物；小猪等在食槽旁边，也得到同样多的食物。显然，大猪费了力，还得不到更多的回报，是占有劣势的。

体力占优势的大猪，反而在这场博弈中成了弱势群体。这种奇怪的现象，是为什么呢？问题的关键在于游戏规则的设置。如果更改投料的多少，就会有意想不到的效果。如果把投料量增加一倍，若小猪踩踏板，大猪与小猪平分食料；若大猪踩踏板，小猪占四分之一，大猪占四分之三。这样，两只猪都会争着去踩踏板，由于踩踏板总比不踩得到的食多。这就是所谓的“多劳多得”。相反，如果把投料量减少到一半，小猪、大猪踩踏板都会被对方抢完食，自己踩踏板没有任何意义，所以两只猪只好饿着。这就是所谓“三个和尚没水吃”。

由此可见，如果是对一个企业而言，员工之间的博弈便类似“智猪博弈”。作为企业的管理者，应该选定恰当的奖惩方案，使得多劳动者多收益，这样才能提高员工的效率。事实上，这就是修改“纳什均衡点”的过程。原来大猪的纳什均衡点是踩踏板，小猪是不踩踏板，大猪自然怨气冲天，拒绝参加博弈，在现实生活中这样的公司是留不住能人的；增加一倍食料后，两只猪的纳什均衡点都是踩踏板，于是效率极大提高。但如果将食料增加至 10 倍，踩不踩踏板差别不大，有时不如费力的损失多，另外 10 倍的食料对猪来说可能已经吃饱了，猪踩踏板的积极性又降低了。于是，在现实生活中，过高的待遇或福利也会导致积极性的降低。

因此，我们要提高一个群体的工作效率，就要修改“纳什均衡点”，使个人利益与集体利益实现统一，就是使“纳什均衡点”重合。纳什均衡理论，还有很多值得研究的地方，也是一个数学化的理论，由于篇幅限制和便于阅读理解，本文不再从数学推导的角度给出严格解释。

我们回到“囚徒困境”问题。初看这个问题，似乎类似心理问题，但经过简单分析之后，就会茅塞顿开。我们对这个游戏进行扩展，考虑重复的双人对弈游戏。如果囚徒困境的结果是减分 0，-1，-10，-15，然后让两个人重复博弈 100 次，会选择怎样的策略？

我们采用倒推法思考这个问题。首先，最后一次博弈后面没有其他博弈，选择什么都不会影响，于是与单次博弈相同，选择坦白；然后考虑第 99 次博弈，这时已知第 100 次不可能合作，这样也没有必要维持信任，于是还是选择坦白。利用数学归纳法，我们可以知道，这 100 次博弈都会选择坦白，与一次的情况相同。这与我们认为的，长时间的竞争，会建立合作互信关系，是矛盾的。原因在于，重复博弈的次数是固定的。

如果将“博弈 100 次”改为“博弈永不停止”，就不能运用倒推法了，由于最后一次博弈根本不存在，我们将必须考虑自己的选择对后面的影响。并且，我们希望达到双方持久的合作，就是都不坦白。这样，我们该采取怎样的策略呢？

曾有科学家对这个问题进行深入的研究，他们选择了数十种不同的策略，利用计算机进行模拟博弈，采用单循环赛的方式，重复若干次，得分最多者为优胜。最终，一种名叫

“Tit-for-tat”的策略以优异的表现取胜。这个策略很简单：首次选择合作；以后把对方上次的策略当作自己的策略。很容易看出，这就是所谓的“以牙还牙”，英文名字 TFT 也就是这个意思。

有趣的是，若游戏双方都选择“以牙还牙”策略，则整个游戏过程都是合作状态；即使与较优的策略进行对抗，“以牙还牙”策略也会渐渐使对方减少背叛，增加合作，双方得分都有所提高。这种简单而巧妙的策略，在现实生活中也许会有极大的作用。

社会不平等的存在，是由于很多博弈都类似“囚徒困境”，所以对单次博弈会采取互相背叛。而社会又趋向于公平，是因为很多博弈是要进行很多次的，而且实际中次数也是不确定的。这样高明的人都会选择“以牙还牙”策略，在增加自己受益的同时，有利于增加合作，减少背叛，从而实现社会的和谐与稳定。我们应当相信，无论中间有多少不公与挫折，社会的进步终究是历史的必然。

另外，修改政策，更改“纳什均衡点”，也是促使合作尽快建立的重要手段。例如国家的法律，企业的奖惩制度，都是让个人利益与集体利益、国家利益尽可能取得平衡、统一，从而在不损害国家利益的前提下实现个人利益最大化。这样，社会和谐就为期不远了。

纳什均衡理论，作为博弈论的重要思想，在经济分析、社会分析等领域有着重要作用。学习纳什均衡，修改纳什均衡，利用纳什均衡，我们的世界必将更加美好！

4.5.2 数学分析的应用

首先我们需要明确“组合游戏”的概念。[1]

组合游戏有两个人参与，二者轮流做出决策。当达到某种事先确定的局面时判定胜负。游戏可以在有限步内结束，且游戏不会有平局出现。任意一个游戏者在某一确定状态可以作出的决策集合只与当前的状态有关，而与游戏者无关。游戏的状态是完全信息公开的。

利用数学归纳法，可以证明所有组合游戏必有一方（先手或后手）有必胜策略。这里研究的问题需要增加一个条件，就是游戏的每一步都只有有限种选择[2]。

假设这个游戏至多在 n 步后终止，对 n 使用数学归纳法：

当 n=1 时，游戏至多进行一步，必胜策略是显然的。

设 n=k 时命题成立。当 n=k+1 时，假设这个至多 k+1 步的游戏由 A 开始，A 有 a 种选择，剩下的游戏变成了至多 k 步的游戏。由归纳假设，这种游戏不是 A 有必胜策略，就是 B 有必胜策略。如果有一种选择产生的新游戏中 A 有必胜策略，A 就执行这种选择，于是在原游戏中 A 有必胜策略；如果所有选择产生的新游戏中，B 都有必胜策略，那么 A 在原游戏中必败，即 B 有必胜策略。

综上，组合游戏中必有一方有必胜策略。[3]

---

[1] 贾志豪《组合游戏略述》中这样描述组合游戏：游戏有两个人参与，二者轮流做出决策。且这两个人的决策都对自己最有利。当有一人无法做出决策时游戏结束，无法做出决策的人输。无论二者如何做出决策，游戏可以在有限步内结束。游戏中的同一个状态不可能多次抵达。且游戏不会有平局出现。任意一个游戏者在某一确定状态可以作出的决策集合只与当前的状态有关，而与游戏者无关。
相对而言，曹钦翔《从 k 倍动态减法出发探究一类组合游戏》的定义较为严谨。
笔者认为，“同一个状态不可能多次抵达”是不必要的条件，只要能在有限步内结束即可；
“两个人决策对自己最有利”是不必要的，只是在分析组合游戏时使用；
“无法做出决策时游戏结束”有些含糊其辞，实际上游戏规则规定的是一种结束游戏的确定状态；
需要强调“信息公开原则”，例如扑克牌、麻将等具有随机性的、状态不完全公开的游戏不属于组合游戏，这是组合游戏与其他游戏容易混淆的一点。

[2] 例如“任意选择一个整数”“选择一个 0 到 1 的实数”都不满足“有限种选择”的要求。

[3] 得到的结果包括，围棋、跳棋中总有一方有必胜策略；象棋、五子棋中，总有一方有不败策略。但由于这些组合游戏可能性太多，无法枚举出搜索树，故必胜策略目前只是理论存在，无法求出。

以上的归纳步骤也给我们用搜索方法解决组合游戏问题提供了算法。例如“**质数取石子**[1]”问题：

桌上有若干石子，DD 先取，轮流取，每次必须取质数个。如果某一时刻某一方无法从桌上的石子中取质数个，比如说剩下 0 个或 1 个石子，那么他/她就输了。当 DD 确定会取得胜利时，他会说：“不管 MM 选择怎样的取石子策略，我都能保证至多 X 步以后就能取得胜利。”那么，最小的满足要求的 X 是多少呢？

对于这个问题，由于暂时没有考虑到更好的数学办法，又涉及“质数”这个很麻烦的事情，只能使用枚举出整个搜索树的办法。

```
#include<stdio.h>
#include<math.h>
#define MAX 20005
int x[MAX]={0},p[MAX]={0};
//x 的绝对值表示当前胜方的最小步数，符号表示必胜策略的归属
int calc(int n,int f)
{ int i,t,min=f*MAX,max=0;
  if(x[n]!=0)
   return f*x[n]; //f 有+1,-1 两种取值，代表当前交手人选
  for(i=2;i<=n;i++) //枚举当前所取的质数。这里可以优化，构造质数表时直接记录所有质数，而不是枚举每个数再判断是否质数。
   if(p[i])
   { t=calc(n-i,-f); //递归搜索，注意对手要调换
     if(t*f>0)
     { if(abs(t+f)<abs(min))
        min=t+f; //正负号表示对手还是自己，是处理双人的办法
     }
     else
     { if(abs(t-f)>abs(max))
        max=t-f;
     }
   }
  if(min!=f*MAX) //计算最小步数，更新最优值
  { x[n]=min;
    return min;
  }
  x[n]=max;
  return max;
}
main()
{ int n,i,j,a;
  for(i=2;i<MAX;i++)
   p[i]=1;
  for(i=2;i<150;i++)//先生成质数表，避免每次都计算
   if(p[i])
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=116

```
   { j=i+i;
     while(j<MAX)
     { p[j]=0;
       j+=i;
     }
   }
  x[0]=x[1]=-1;
  scanf("%d",&n);
  for(i=0;i<n;i++)
  { scanf("%d",&a);
    for(j=0;j<=a;j++)
     calc(j,1);
    if(x[a]<0)
     printf("-1\n");
    else printf("%d\n",x[a]-1);
  }
}
```

当然，解决组合游戏问题，不能只靠枚举，更重要的是数学分析。下面介绍一个著名的“取石子游戏”。

例[1] 甲、乙两人轮流从两堆棋子中取棋子，满足下列要求：或者从一堆中取出任意多枚（至少一枚）棋子，或只从两堆中取出同样数目（至少一枚）的棋子，将两堆取完并取到最后一枚棋子者获胜。问：在什么情况下，甲（先取者）有必胜策略？

有趣的是，构造数列 $a_n=[\frac{\sqrt{5}-1}{2}n]$, $b_n=[\frac{\sqrt{5}+1}{2}n]$ 这两个数列恰好补充不漏的覆盖整个正整数集；性质：对任意 $n\in N^+, a_{n+1}=a_n+\begin{cases}1 & n\notin\{a_i\}, i=1,2,...n\\ 2 & n\in\{a_i\}, i=1,2,...n\end{cases}$. 这是两道 IMO 试题的命题思想来源。

这是由于，一个著名的“贝蒂定理”：设 a、b 是正无理数且 1/a +1/b =1。记 P={ [na] | n 为任意的正整数}，Q={ [nb] | n 为任意的正整数}，则 P 与 Q 是 Z+的一个划分，即 P∩Q 为空集且 P∪Q 为正整数集合

证明：因为 a、b 为正且 1/a +1/b=1，则 a、b>1，所以对于不同的整数 n，[na]各不相同，类似对 b 有相同的结果。因此任一个整数至多在集合 P 或 Q 中出现一次.

下证 P∩Q 为空集；(反证法)假设 k 为 P∩Q 的一个整数，则存在正整数 m、n 使得[ma] =[nb]=k.即 k < ma，nb<k+1，等价地改写不等式为 m/(k+1)< 1/a < m/k 及 n/(k+1)< 1/b < n/k. 相加起来得 (m+n)/(k+1) < 1 < (m+n)/k，即 k < m+n < k+1.这与 m、n 为整数有矛盾，所以 P ∩Q 为空集.

下证 Z+=P∪Q；已知 P∪Q 是 Z+的子集，剩下来只要证明 Z+是 P∪Q 的子集. (反证法)假设 Z+\(P∪Q)有一个元素 k，则存在正整数 m、n 使得[ma]< k <[(m+1)a]、[nb]< k < [(n+1)b]. 由此得 ma < k <=[(m+1)a]-1<(m+1)a -1，类似地有 nb < k <=[(n+1)b]-1<(n+1)b -1. 等价地改写为 m/k < 1/a < (m+1)/(k+1)及 n/k < 1/b < (n+1)/(k+1)。两式加起来，得(m+n)/k < 1

[1] 本例解答摘录自《骗分导论·数学竞赛》。

< (m+n+2)/(k+1)，即 m+n < k < k+1 < m+n+2.这与 m, n, k 皆为正整数矛盾.

所以 Z+=P∪Q.

我们看到，令 $a=\frac{\sqrt{5}+1}{2}, b=\frac{\sqrt{5}+3}{2}$,恰好满足“贝蒂定理”的条件，于是构造出两个“互补”的数列. 至于第二个性质，是由于在将 1,2,...依次排入两个数列时，不属于 a 数列的自动进入 b 数列，占据了一个数，而 b 数列中没有两数连续，故此时 a 数列相邻两项差为 2. 其余时 a 数列各项为连续正整数. 事实上，对每个 $a_i, a_{a_i+1}-a_{a_i}=2$,此时 $b_i=a_{a_i}+1$.更巧妙的是，$b_n=a_n+n$. 但若取 a,b 为其他无理数，则难以找到如此巧妙的性质。

有一种取石子游戏：两堆石子分别有 m、n 个，两人轮流取石子，或从一堆中取若干个，或从两堆中取同样多个。最后取完所有石子者获胜。问先取者是否有必胜策略？

我们惊奇的看到，当 m、n 分别是上述数列的 a_i, b_i 时，后取者有必胜策略；否则先取者有必胜策略。原理十分简单。我们称一个状态 (a_i, b_i) 为胜状态，其他(a,b)为败状态。对任意状态(a,b)，若为胜状态，则经过一次操作，由于两个数列无重复项，从一堆中取显然不能到另一个胜状态；从两堆中取，没有两个胜状态的石子数之差相等，也不可能到胜状态。而从败状态出发，总有一种取法使得成为胜状态，由于两个数列递增，若状态中的一个数存在于对应数列 a、b 中，则取走另一堆中多余的棋子；若状态中的两个数都不在对应数列中，则取走相同数目，让较少的一堆棋子数成为 a 数列中的元素，则此时另一堆棋子数必在 b 数列中。由此，必胜策略的原理得证。

只要发现了数学方法，本题极为简单。但如果没有发现，就难以得分。朴素的搜索只能通过极少数测试数据。

当寻找该题的策略时，可以首先编写一个搜索程序，利用数学归纳法，求出较小情况时两堆棋子数的规律，如果数学眼光敏锐，就可以发现“黄金分割”的关系。

令 f[i][j]表示棋子数为 i,j 而当前轮到先取者是否有必胜策略,1 表示是,0 表示不是。

显然，f[0][j]=f[i][0]=f[i][i]=1;由于可以直接取完所有棋子。

对于一般的 f[i][j],如果对所有的 f[k][j],0<=k<i、所有的 f[i][k],0<=k<j、所有的 f[i-k][j-k],0<k<min{i,j}，都为必胜状态(1)，则 f[i][j]=0(必败状态)；否则为必胜状态。

原理是，只要有一种下一步状态可以导致其取得胜利，则他会立即抓住机会，使对方成为必败；若所有下一步状态都会失败，则他必败，对手必胜。

这样枚举整棵博弈树，递归得到答案。

关于博弈问题，我们给出另外一个题目“**课间十分钟**[1]”，以体现数学思想的应用。

在游戏开始前，他们先约定一个正整数 n，同时令 m=1。游戏过程中，每个人都可以将 m 的值扩大 2 到 9 中的任意倍数。第一个使 m≥n 的人就是最后的赢家。

在本题的求解中，注意到两者的最优策略是可以计算出来的：对于先取者，应取 9m-8;对于后取者，应取 2m-1. 这样反复迭代，就可以得到结果。这也是数学分析在博弈论中的应用。

事实上这个方法可以称作是贪心：先取者希望尽可能快的达到目标以取胜，后取者希望尽可能慢的增长以拖住先取者的步伐。

[1] http://www.rqnoj.cn/Problem_Show.asp?PID=185

```
#include<stdio.h>
main()
{ __int64 now=2,n;
  scanf("%I64d",&n);
  if(n==1)
  { printf("181818181818");
    return;
  }
  while(1)
  { now=now*9-8;
    if(now>n)
    { printf("181818181818");
      break;
    }
    now=now*2-1;
    if(now>n)
    { printf("ZBT");
      break;
    }
  }
}
```

上面的程序不禁让人拍案叫绝。由此可见，要想成功“骗分”，扎实的数学功底是必要的。尤其是归纳、猜想的能力，更是每个 OIer 应该重视的。

4.5.3 SG 定理与组合游戏[1]

对于一个组合游戏，一般可以用 f：X.P(x)来描述游戏规则。其中 X 表示状态集，P(x)表示 X 的子集簇。f(x)就表示玩家可以从 x 出发一步操作后到达的状态的集合。我们称任一个)(xfy.，y 是 x 的一个后继。也就是说，本文讨论的组合游戏可以被理解成一个有向无环图 G=(V,E)，图的点集 V=X，边集 E=(xy| y . f (x))。

很多有关论文将 SG 定理描述的过于数学化、过于深奥，本节将从易于理解的角度出发，简单介绍 SG 定理。

所谓的 SG 函数简单的说就是一个布尔函数，如果一个状态是先手必胜，则取 1；如果一个状态是后手必胜，则取 0. [2]显然，根据上节中的归纳算法，容易得到每个状态的 SG 值。但这样太慢了。

先给出几条性质：对于任意的局面，如果它的 SG 值为 0，那么它的任何一个后继局面的 SG 值为 1. 这是由于，一旦某个局面 A 不能胜，则后面游戏再进行下去，始终是 B 必胜；反过来，如果有一个后继局面 A 必胜，则 A 可以按此方法选择，导致 SG=1，矛盾。

对于任意的局面，如果它的 SG 值为 1，那么它至少有一个后继局面的 SG 值为 1. 这是由于，如果所有后继局面均不可胜，则根据归纳法，SG=0，矛盾。

[1] 参考了贾志豪《组合游戏略述》。

[2] 这是不严格的说法，并不是只能取 0、1，但实际应用中我们只关心必胜策略的归属。

考虑著名的“取石子游戏”，即 NIM 模型：桌子上有 N 堆石子，游戏者轮流取石子。每次只能从一堆中取出任意数目的石子，但不能不取。取走最后一个石子者胜。

我们考虑，如果只有一堆石子，即 n=1，则 SG=1，先取者直接取走所有石子。

如果有两堆石子，即 n=2，如果两堆石子个数相同，显然无论先取者在某一堆中取多少个棋子，后取者可以再另一堆中取若干个棋子使得两堆棋子个数又相同，这样继续下去，后取者必胜。如果开始时两堆石子个数不同，先取者可以将其变为个数相同，这样先取者胜。

我们注意到，这与石子个数用二进制表示的异或值有关。将 N 堆石子的个数按位相加，并不进位，最后得到的结果如果为 0，则先手必败；否则先手必胜。实际上就是取异或值的过程。

根据定义，证明一种判断状态的性质的方法的正确性，只需证明三个命题： 1、这个判断将所有负状态判为 SG=0；2、根据这个判断被判为胜状态的局面一定可以移动到某个负状态；3、根据这个判断被判为负状态的局面无法移动到某个胜状态。

第一个命题显然，平凡状态只有一个，就是全 0，异或仍然是 0。

第二个命题，对于某个局面(a1,a2,...,an)，若 a1^a2^...^an!=0，一定存在某个合法的移动，将 ai 改变成 ai'后满足 a1^a2^...^ai'^...^an=0。不妨设 a1^a2^...^an=k，则一定存在某个 ai，它的二进制表示在 k 的最高位上是 1（否则 k 的最高位那个 1 无法得到）。这时 ai^k<ai 一定成立。则我们可以将 ai 改变成 ai'=ai^k，此时 a1^a2^...^ai'^...^an=a1^a2^...^an^k=0。

第三个命题，对于某个局面(a1,a2,...,an)，若 a1^a2^...^an=0，一定不存在某个合法的移动，将 ai 改变成 ai'后满足 a1^a2^...^ai'^...^an=0。因为异或运算满足消去律，由 a1^a2^...^an=a1^a2^...^ai'^...^an 可以得到 ai=ai'。（事实上，单独改变一个数，必然改变各个数的异或值）所以将 ai 改变成 ai'不是一个合法的移动（没有取走棋子）。证毕。

对于一切简单组合游戏，都可以划归到 NIM 取石子游戏。SG 定理的适用范围是，**可以分为若干个独立的“小游戏”，每次操作必定在其中之一进行**，这样只须计算出每个小游戏的 SG 值，再取异或，就得到整个游戏的 SG 值。

我们在解题时，只要记住**对每个单一游戏取异或值**这一点，就可以避免对多个独立事件所有可能性的探讨。进一步的讨论超出了本文的范围。

4.6 概率

概率，就是描述一件事发生的可能性。很多同学在数学课上，就对“概率”这件事理解的不深刻。尤其是“等可能事件”以及涉及“无穷”的概率问题，容易搞混。

例如这样一个问题[1]：现有 N 个投票器，每人随机投一票，当 N 个票数相等时停止投票，问最后能够停止的概率。

某些同学可能简单的认为，把 M 次投票后所有可能出现的票数情况列出来，共有 N^M 种情况，而票数相等的最多有一种，因此 $M\to\infty$ 时概率为 1，即必定不能停止。

这样的分析显然是荒谬的，因为开始 N 次如果每个投了一次票，就会停止，停止的概率必定不是 0. 那么，这样的分析错误在哪里？

首先，忽略了“相等时停止”这个重要条件，因此把条件概率问题转化为无条件的概率问题，显然是错误的。

其次，不同的票数情况是不等可能的。考虑最简单的情形，2 个投票器 A、B，投 2 次，则 P（A 得 2 票）=P（B 得 2 票）=1/4；P（A、B 各得一票）=½。显然不等可能。问题出在哪

[1] http://boj.pp.ru/olympic/math/combine/suijitoupiao.htm

里？因为这里的“随机投票”是针对每一次投票，而不是总体的投票过程。因此情况的划分应该针对每一次投票，而不是每个投票器。

所以，讨论“等可能事件”时一定要从“基本事件”入手，保证每组基本时间是题目中保证是等可能的，再考虑基本事件的重复试验问题。像这道题，当 2 个投票器时，不考虑停止问题，两个投票器的票数就符合“二项分布”。当 m→∞时，两个投票器的票数就服从正态分布。

讨论：当 n=2 时，可以化为“猫捉老鼠”问题：有一只猫，在数轴原点处，一只老鼠在 x=1 处，猫每步随机的向正半轴或负半轴走一个单位，老鼠不动。现在求猫捉到老鼠的概率。把 A 投一票看成猫向右走一步，把 B 投一票看成猫向左走一步。不同的是，这次猫和老鼠开始相同，只需第一步将它们错开一个单位即可。

解法一：设从 0 出发，走到 1 的概率为 x。则由于走无穷多步，从-1 出发走到 0 的概率也为 x。而从 0 出发捉到老鼠，或者向右走，有 1/2 的概率；或者向左走，有 1/2 的概率，此时要回到 1，必须向右走两步，独立事件相乘，概率为 x*x。于是有方程 x=1/2+1/2x*x. 解得 x=1.这表明无穷多步后猫一定捉到老鼠。

解法二：对有限步的情况，考虑猫最后一个折返点（最后一个向右走的起点）。从坐标为-n+1 的点走到 1 的概率为 2^(-n)，由于每步 2 种情况，互相独立，共 2^n 种，而只有向右走符合要求。这样，概率为从 0,-1,...-n+1 开始向右走达到 1 的概率之和，为 1/2+1/4+1/8+...+2^(-n+1)=1-2^(-n+1).当走无穷多步后，n→∞时，概率的极限为 1.

原题：不能采用“猫和老鼠”的二维情况（有 1 个自由度），由于有 3 个变量，只能划归为 2 个相互独立的变量（即有 2 个自由度）。讨论较为困难，尚不清楚。

这个问题绝不是简单问题。笔者认为，本题概率为 0，即投票在无穷多步后必然结束。[1]但没有找到合适的证明。

先来考虑一个类似问题。A、B、C、D 四个人玩一个残酷的游戏，游戏规则如下： 四个人分别拥有一个幸运值 a、b、c、d. 游戏的每一回合，其中一个人的幸运值将加 1. 其中，A 的幸运值加 1 的概率是 a/(a+b+c+d)， B 的幸运值加 1 的概率是 b/(a+b+c+d)， C 的幸运值加 1 的概率是 c/(a+b+c+d)， D 的幸运值加 1 的概率是 d/(a+b+c+d). 每到下一回合都将以新的 a,b,c,d 来计算. 这意味着强者更强，弱者更弱，如果弱者在最初没有机会翻身，以后将永远没有出头之日了. 已知现在 a=1,b=2,c=3,d=4. 求无数个回合（N 个回合，N→∞）之后 A、B、C、D 的幸运值是最大的的概率分别是多少. [2]

我们还是先来考虑两个人玩的情况。设 a=100,b=200.

简单的考虑，设游戏进行了 299 步，则 A 要赶上 B，则 A 可能赢 299，298...200 步，B 可能赢 0,1,...99 步，否则 A-B<100, 由于 a=100,b=200，A 没有赶上 B。因此 A 要赢，必须从 299 局中赢 0,1,...99 局，根据组合式可以算出；而每步实际上有两种可能，共有 2^{299} 种情况，故根据古典概型：A 胜的概率为 $\frac{C_{299}^{0}+C_{299}^{1}+...+C_{299}^{99}}{2^{299}}$，B 胜的概率为 1-P(A). 计算结果，P(A)=0.0000000026465545, P(B)=0.9999999973534455, 也就是说 A 几乎必胜。[3]

对于原题，P(A)=56763039/2579890176 ≈2.2002%

P(B)=254244950/2579890176 ≈9.8549%

P（C）=662921693/2579890176 ≈ 25.6957%

P(D)= 1605960494/2579890176 ≈ 62.2492%

[1] 注意，在状态空间为无穷的概率问题中，概率为 1 不等于必然事件，概率为 0 不等于不可能事件。例如向一个圆内投针，针恰好扎到圆心的概率为 0，由于圆心的面积为 0. 但针扎到圆心却是可能发生的。

[2] http://tieba.baidu.com/f?kz=266867465

[3] 以上的讨论没有使用“无穷”的思想，只是一个粗略的讨论。准确的讨论需要使用积分知识。

是否出乎了您的意料？

再来看一道简单问题：将(0,1)区间上随机找两个点，分成三条线段，问这三条线段组成三角形的概率。

要构成三角形，也就是最长的一条<1/2. 我们可以从反面来求解这个问题，就是求最长的线段>=1/2 的概率.

分 3 个部分：

（1）从左向右第一条线段>=1/2 的概率,即为两点均在后半段的概率 P1=1/2*1/2=1/4

（2）从左向右第三条线段>=1/2 的概率,即为两点均在前半段的概率 P2=1/4

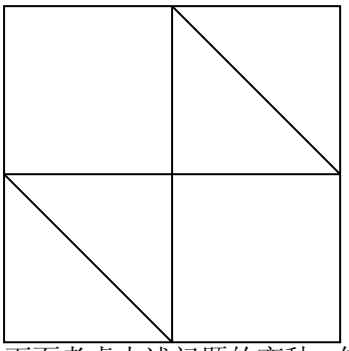
(3) 中间一条线段>=1/2 的概率:

若第一个点（假设两点按时间先后投放，不影响结果）在（0，1/2）上的 x 处，则其在 x 附近 dx 长度上概率为 dx,此时第二个点在其右边>=1/2 处的概率为 1-(x+1/2)=1/2-x，将以上 2 个概率相乘并在（0，1/2）区间上积分，得到概率为 1/8

对应地，第一个点在(1/2,1)上，并且第二个点在其左边>=1/2 处的概率同样是 1/8，因此 P3=1/8+1/8=1/4;

综上，构成三角形的概率为 1-1/4-1/4-1/4=1/4.

以上的讨论使用了分类讨论，还使用了积分的思想。简单起见，我们还可以使用几何方法解释。令 x 轴表示分点 A 的坐标，y 轴表示分点 B 的坐标。还是分上述情况讨论，但可以使用图形，其中左下角的三角形、右上角的三角形表示两点都在左、右两侧，左上角、右下角的正方形表示两点间距大于 1/2，只剩下 1/4 的可能性。由此可见，使用几何概率模型，能够避免复杂的积分，更加直观，易于思考。



下面考虑上述问题的变种：任取三条线段（注意这次三条线段互相独立，而上题中只有两条长度独立），问形成三角形的概率。

三条线段必有一条最长，设其长为 z，令两条先段长分别为 x,y,可知 0<x,y<z，坐标 x0y 上为一正方形． 又因 x,y 满足 x+y>z，刚好就占正方形的 1 / 2． 所以概率为 1 / 2 。

通过以上问题的求解，我们发现选择适当的“等可能事件”作为“基本事件”对于解决概率问题是至关重要的。不同的概率模型、思考角度，会带来不同难度的解答。

在概率问题的求解中，我们应当发现，不论问题中所说的物品是相同还是不同，我们都可以将它们看做不同的，根据古典概型，总可能数要使用排列数计算，否则每种情况是不等可能的[1]；不论随机取物的先后顺序，概率都相同，故可以人为指定一个顺序。因此在具体求解“取球”“摸奖”类问题时，可以将所有可能发生的事件编号，当做不同的处理；而顺序是可以打乱的，例如“同时摸出”在具体处理时可以变成“依次摸出”。

[1] 这里的等可能性是由于分步解决的问题，每步的若干个事件等可能，则由每步中选出一个事件组成一个过程，这样的过程之间是等可能的。

4.7 组合构造

上文介绍了种种数学方法，却没有介绍在实际应用中如何“想到”这些方法。这个“想”的过程，就是“组合构造”。

例如这样一道题：现有 n*m 颗糖果，要求分到尽可能少的袋子里，使得若有 n 个小朋友来时，可以给每个小朋友一些袋子，使他得到 m 个糖果；若有 m 个小朋友来，可以给每个小朋友一些袋子，使他得到 n 个糖果。输入 n，m，输出最少袋子个数和每个袋子中的糖果数。

一眼看去，就知道这是一道数学问题，因为输入很少。采用简单的搜索，显然是不可行的。首先我们将这个问题转化为数学问题：一个有 m*n 个元素的集合，求一个最小覆盖的子集族，使得这些子集族可以构成 n 个有 m 个元素的集合，也可以构成 m 个有 n 个元素的集合。

我们从较小的情况入手，看一下 n=7,m=11 的情况[1]。由于需要构成 7 个大小为 11 的集合和 11 个大小为 7 的集合，那么袋子的大小不能超过 7. 为了使袋子数尽可能少，我们需要使袋子大小为 7 的尽可能多。进一步，由于只需要 7 个大小为 11 的集合，多于 7 个大小为 7 的集合将无法放入 11 的集合。因此只能构成 7 个大小为 7 的集合。此时剩下 28 个元素，7 个大小为 11 的集合各剩下 4 个元素，11 个大小为 7 的集合还剩下 4 个集合。

我们惊喜的发现，现在问题的规模被缩小了，变成了 4*7 的问题。观察 4 产生的原因，正是 11-7=4. 因此我们产生了类似“辗转相除法”的思想，再构造 4 个大小为 4 的袋子，剩下 4*3 的规模；再构造 3 个大小为 3 的集合，变成 1*3；最后，三个大小为 1 的集合，解决问题。总共 17 个集合。由于辗转相除的性质，容易发现 17=7+11-1。

进一步猜想，是否对所有情况，都有 n+m-1 呢？显然不是。当 n=2，m=4 时，只需要 4 个大小为 2 的集合，4=4+2-2. 对其他情况考虑，加上“辗转相除法”的启发，我们猜想：最小集合数为 n+m-(n,m)，这里(n,m)表示 n 与 m 的最大公约数。

为了证明这个命题，我们构造一个二分图，将 m 个有 n 个元素的集合视作 A 类顶点，将 n 个含有 m 个元素的集合视作 B 类顶点。如果 A 类顶点 a 与 B 类顶点 b 的交集非空，则将 a、b 之间连一条边。那么每一条边对应的便是我们的一个“袋子”。下面证明的，就是图中的边数不小于 n+m-(n,m).

设整个图被分为 k 个连通子图，即子图内部互相可达，而与外部没有边相连。每个子图中 A 类集合与 B 类集合所包含的糖果是相同的。不妨设 A 类集合包含的糖果数为 pn，B 类包含的糖果数为 qm，p，q 为正整数。则有 pn=qm。由数论知识，(n,m)|p, (n,m)|q，故有[n,m]|pn, [n,m]|qm, 即糖果个数不小于[n,m]=n*m/(n,m).

假设 k>(n,m)，则每个子图中的个数>=n*m/(n,m)，总糖果数>n*m，矛盾。因此 k<=(n,m). 对于每个连通子图，由于连通性，至少需要构成一棵树，即边数>=糖果数-1. 对 k 个子图求和，总边数>=n*m-(n,m). 命题得证。

而这个下界是可以达到的。若 n=m, 直接构造 m 个每个含 m 个元素的袋子，问题结束；若 n!=m,不妨设 n>m，仍然构造 m 个含 m 个元素的袋子，将问题的规模缩减为 n,n-m，再递归解决。

由于最大公约数的性质，最后 n=m 一定在(n,m)时取到。因此可以现将 n/=(n,m), m/=(n,m),变为互质的 n,m，逐次相减，其和恰好为 n+m-1，再同时乘上(n,m)，即可说明下界是可达的。

在具体编程实现时，可以采用类似辗转相除法的 while 循环，边计算边输出。

通过以上实例，我们体味了一道数学问题解决的“心路历程”。在类似的需要找到公式的组合构造类问题时，不要盲目编写搜索程序，而要从小处着眼，发现规律，验证规律，利

[1] 如果选择 n=2,m=3 之类过小的数据，不能发现规律。

用数学知识辅助解题。

递推方法，在信息学竞赛中的数学类试题往往有重要应用。递推方法，就是类似“数学归纳法”的思想，从最简单的情形出发，寻求两个相邻状态之间的关系，进而逐步求出最终结果。

我们来看一道运用递推方法解决的类似搜索问题：跳马的哈密尔顿回路问题。[1]

某种在 n*n 的棋盘中玩的新型“国际象棋”中的马，仍然是从 2*3 的矩形一个角跳到另外一个角上。问是否能够从某个点出发，补充不漏的跳遍整个棋盘，最后回到出发点？

首先观察跳马时的一些规律性的变化。按照国际象棋的要求，将棋盘黑白相间染色，发现马无论怎么走，都必须按黑格－白格－黑格－白格，如此循环。由于要回到起点（起点与终点同色），途经两种颜色的格子数必相等，可知 n 为奇数时无解。

首先对小数据进行手工试验，由于马“伸不开腿”，显然 n<6 时无解。

当 n>=6 且为偶数时，用递推法构造：假设(n-4)*(n-4)的棋盘已找到哈氏圈。

1） n 除以 4 余 2 时，

在内矩形四个角（A、E、I、M）上分别开口。

	C							O	
			D			P			
		A					M		
				B	N				
				F	J				
		E					I		
			H			L			
	G							K	

1	16	19	26	7	4
20	25	2	5	18	27
15	26	17	8	3	6
24	21	32	11	28	9
35	14	23	30	33	12
22	31	34	13	10	29

1 将 C 与 D 所在的外回路与“内矩形”的回路在 A、B 上对接，变成 A-C-. . . -D-B。

2 将 G 与 H 所在的外回路与“内矩形”的回路在 E、F 上对接，变成 E-G-. . . -H-F。

3 将 K 与 L 所在的外回路与“内矩形”的回路在 I、J 上对接，变成 I-K-. . . -L-J。

4 将 O 与 P 所在的外回路与“内矩形”的回路在 M、N 上对接，变成 M-O-. . . -P-N。

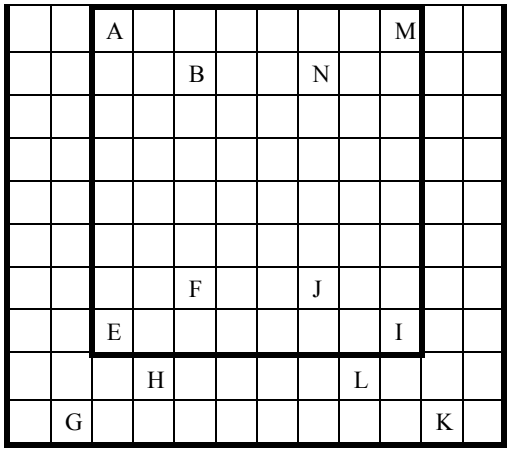
在这里，要注意一个问题，就是作为基础矩形的“内矩形”的回路，首先要满足：A 的下一步到 B，E 的下一步到 F，I 的下一步到 J ，M 的下一步到 N。只有这样，构造成的新矩形才能继续作为“内矩形”按上述规则向外扩展。现给出满足要求的基础矩形的一组解（N=6）

2） n 除以 4 余 0 时

在内矩形四个角（A、E、I、M）上分别开口。

			C					O			
					D	P					

[1] 引自方奇《染色法和构造法在棋盘上的应用》。



1	54	47	38	49	52	31	26
46	39	2	53	32	27	22	51
55	64	37	48	3	50	25	30
40	45	56	33	28	23	4	21
63	36	61	44	57	20	29	24
60	41	34	15	12	5	8	19
35	62	43	58	17	10	13	6
42	59	16	11	14	7	18	9

1 将 C 与 D 所在的外回路与“内矩形”的回路在 A、B 上对接，变成 A-C-. . . -D-B。
2 将 G 与 H 所在的外回路与“内矩形”的回路在 E、F 上对接，变成 E-G-. . . -H-F。
3 将 K 与 L 所在的外回路与“内矩形”的回路在 I、J 上对接，变成 I-K-. . . -L-J。
4 将 O 与 P 所在的外回路与“内矩形”的回路在 M、N 上对接，变成 M-O-. . . -P-N。

以上便是采用“逐增构造”解决马的哈密尔顿回路问题的方法。而马的哈密尔顿环问题，则显得简单很多，因为不需要回到原来的出发点。也许出乎读者的意料，一种贪心的方法即可解决此问题：每次跳向尚未到达过的、与最少的可达点相连的格子。说得更清楚一些，就是将每个格子看做一个顶点，两个格子之间跳马可达则连一条边，每次寻找相邻的度最小的顶点即可。

每次跳向可达点最少的格子，事实上就是先往“边角”处跳，避免中间的跳过之后“边角”被堵死。对于每个位于边角位置的格子，至少有两条路径与外界相通；当第一次到达其中一个相邻点时，由于度数的最小性，会前往边角处，此时还可以从另一个相邻点出去。如此类推，就不会出现“进入死胡同”的情况。只要 n>3，即可达到目的。

读者可以思考这样的问题：如果棋盘是 n*m 的矩形，又会怎样？如果马的跳动规则不是 2*3，而是 p*q，又会怎样？

笔者给出一种猜想：当 p、q 不互质时，必定不存在哈密尔顿链，由于不是(p,q)倍数的点不可达到；当 p、q 互质时，只要矩形的较短边长不小于 p+q-1，就可以形成哈密尔顿回路。更大胆的猜想，对于这样的棋盘和跳动方式，仍然可以采用上文所述的贪心策略：每次寻找度数最小的顶点跳动。希望读者思考这个问题。

Matrix67 的博客[1]上有这样一道题：定义 f(n)的值为将 n 拆分成若干个 2 的幂的和，且其中每个数字出现的次数不会超过两次的方案数。规定 f(0)=1。例如，有 5 种合法的方案可以拆分数字 10：1+1+8, 1+1+4+4, 1+1+2+2+4, 2+4+4 和 2+8。因此，f(10)=5。请用一句最简单的话来描述集合{ f(n)/f(n-1) }。证明你的结论。

答案：数列{ f(n)/f(n-1) }以最简形式包含了所有的正有理数。

如果 n 是奇数（等于 2m+1），那么数字 1（即 2^0）必须出现且只能出现一次。现在的问题就是，2m 的拆分方案中有多少个方案不含数字 1 呢？稍作思考你会立即发现，它就等于 f(m)，因为 m 的所有拆分方案的所有数都乘以 2 后正好与不含 1 的 2m 拆分方案一一对应。因此，f(2m+1) = f(m)

[1] http://boj.5d6d.com/thread-471-1-1.html

如果 n 是偶数（等于 2m），那么数字 1 要么没有出现，要么恰好出现两次。对于前一种情况，我们有 f(m)种可能的方案；第二种情况则有 f(m-1)种方案。因此，f(2m) = f(m) + f(m-1)

另外，显然 f(k)都是正数。于是，f(2k-1) = f(k-1) < f(k-1)+f(k) = f(2k)

这样，我们可以得到以下三个结论：

结论 1：gcd(f(n),f(n-1)) = 1

证明：对 n 进行数学归纳。显然 gcd(f(1),f(0)) = gcd(1,1) = 1

假设对于所有小于 n 的数结论都成立。根据 n 的奇偶性，下面两式中必有一个成立：

gcd(f(n),f(n-1)) = gcd(f(2m+1),f(2m)) = gcd(f(m), f(m)+f(m-1)) = gcd(f(m),f(m-1)) = 1

gcd(f(n),f(n-1)) = gcd(f(2m),f(2m-1)) = gcd(f(m)+f(m-1), f(m-1)) = gcd(f(m),f(m-1)) = 1

结论 2：如果 f(n+1)/f(n) = f(n'+1)/f(n')，那么 n=n'

证明：还是数学归纳法。当 max(n,n')=0 时结论显然成立，因为此时 n=n'=0。

假如对于所有小于 n 的数结论都成立。由于 f(2k-1)<f(2k)，那么要想 f(n)/f(n-1) = f(n')/f(n'-1)，n 与 n'的奇偶性必须相同，于是可以推出 f(m)/f(m-1) = f(m')/f(m'-1)，根据归纳我们有 m=m'，这就告诉我们 n=n'。

结论 3：对于任何一个有理数 r，总存在一个正整数 n 使得 r=f(n)/f(n-1)。

证明：把 r 写成两个互素的数 p 和 q 的比。我们对 max(p,q)施归纳。

显然，当 p=q=1 时结论成立，此时 n=1。

不妨设 p<q，那么定义 r'为 p/(q-p)。根据归纳假设，总存在一个数 m 满足 r'=f(m)/f(m-1)。于是 r=f(2m+1)/f(2m)。当 p>q 时同理可证明。

以上问题的答案，我们开始时很不容易猜到，所以才是一道妙题。这道题使用了数学归纳法，主要是应用了数学中“一一对应”的原理。

例如这样一道信息学竞赛难题[1]：

如果一个字符串 v 在另一个字符串 u 中出现，则称 v 是 u 的子串。如‘cde’是‘abcdef’的一个子串。如果 vv 是 u 的子串，则称 vv 是 u 的重复子串，这里 vv 表示两个字符串 v 和自己的连接，如当 v=‘abc’时，vv=‘abcabc’是 u=‘cabcabcd’的子串，因此‘abcabc’是‘cabcabcd’的一个重复子串。

不妨来数一下一个字符串中不同重复子串的个数，如 u=‘abcabcbcbc’中，u 的所有重复子串在下表中列出：

abcabcbcbc　abcabcbcbc　abcabcbcbc　abcabcbcbc

其中不同的重复子串有‘abcabc’、‘bcbc’和‘cbcb’，共 3 个。

我们称仅由字符 0 和 1 构成的字符串为 01 串。

给定字符串的长度，希望你能找到一个这样的 01 串，包含尽量少的不同的重复子串。

首先，一些选手会想到一个 log n 的想法：观察这样一个序列

1 2 1 3 1 2 1 4 1 2 1 3 1 2 1 5 …

[1] Winter Camp 2007 Problem 1

没有重复子串，但字母表比较大：O(logN)

转化：k -> 0 ...0 (k 个 0) 1 0 1 ... 1 (k 个 1)，重复基本出现在 0（k 次）和 1（k 次），但还有一些交叉情况。

效果是：

N	answer	score
3	0	10
7	1	10
18	2	10
65	5	8
206	7	7
739	9	7
1691	9	8
5000	11	8
10000	13	8
20000	13	9

得分已经不错，但仍然不理想。那么，标准解法又会是怎样的？

当 n=3 时，010，answer = 0

当 n=7 时，0001000，answer = 1

当 n=18 时，010011000111001101，answer = 2

当 n>18 时，answer=3.

1. 构造数列$\{a_n\}$: $a_{2n} = a_n$;　$a_{2n+1} = 1 - a_n$; $a_0 = 0$。得到串 u = 0100110100101100110…，使用这个串得分约 50~70.
2. 下面由 u 进行编码，每一位和后一位生成一个编码，01->0, 10->1, 00,10 -> 2 ，得到一个 012-串 T。
3. 在 T 串中相邻的 12 或 21 间插入 3，得到一个 0123-串：Q。
4. 用下列方法替换 Q 中的字符得到 S

   0　011 000 111 001

   1　011 100 011 001

   2　011 001 110 001

   3　011 0001 0111 001

可以证明 S 中只有 3 个不同的重复子串：00, 11, 0101。

如此构造，即可得到最终结果。笔者尚不清楚其最优性证明，即为什么 n>18 时重复子串至少有 3 个。这个构造也是十分巧妙的，可以认为是把“组合构造”发挥到了极致。

5 非完美算法

“我是智障”《骗分导论》中说，大牛是少见的，能够每道题 AC 的大牛更是少见的。让 100 分白白流去，实在是太可惜了。要“精彩的骗”，就要使用“非完美算法”，尽可能得分。

所谓“非完美算法”，就是不能得到“最优解”的算法，但所得到的解或者有很大可能是正确解，或者距离正确解十分接近。

在信息学应用技术中，很多实际问题都是 NPC 问题，无法在多项式时间内找到问题的最优解。如旅行商问题，实际的许多路线设计、物资调配问题与之类似。再如人工智能问题，

就更不能把所有情况一一枚举，而要利用小范围的搜索和“估价函数”，甚至事先输入的“经验”，得到相对较优的解，象棋人机对弈就是如此。

甚至一些问题是不可解问题，无法找到问题的精确解，而只能得到“概率意义下正确”的答案，如天气预报。但实际应用中用如此多的时间换取理论意义上的“最优性”“精确性”是没有必要的，我们只需要一个“较优”的解，使得人们能够接受，又不会花费太多的时间。在当今算法研究领域，一方面是降低算法的时间复杂度，另一方面就是使解更优、更精确。

在信息学竞赛中，有一些问题是有“部分分”的，即凭借解的优劣给分；或者某种“贪心”算法能在大部分情况下正确，非完美算法就是不错的选择。笔者也建议今后的比赛，尤其是高层次比赛，多出现使用非完美算法的题目，因为没有现成算法可循的试题才能反映选手建立模型、数学分析、创造算法、优化程序的能力，才能体现选手驾驭计算机语言，用计算机解决未曾遇到的实际问题的能力。

5.1 贪心法

《现代汉语词典》对“贪心”的解释是：贪得无厌，不知足。我们信息学竞赛中的“贪心法”，就是取自“不知足”之意。贪心的人，每次都选择“当前看起来最好”的选择，而不顾及后面事件的发展。当然，在现实生活中，由于问题的复杂性，这种只顾眼前、不顾长远的做法显然不是好的策略。但在信息学中，一些较为简单的问题，满足“局部最优”的特点，就能运用贪心法解决。

类比实际生活，我们已经得到“贪心法”的适用范围：局部最优性、无后效性。简单的说，就是每步都选择当前一步看起来最优的，不考虑其他影响。如果需要考虑前面已经做出的决策，那就是动态规划；如果还与后面未作出的决策有关，就要使用其他方法解决，或使用变通的方法。例如 IOI“餐巾”一题，由于具有后效性，可以使用转换的方法，假设所有餐巾都是第一天买，并将洗餐巾的动作尽量靠后，即不必花钱的时候少花钱，需要时在用。这样，一个看似只能用搜索解决的问题，便有了高效的贪心算法。

首先我们看一个错误的贪心（动态规划）：[1]

十一、动态规划的算法比较

1.0/1 背包问题

```
for(i=0;i<n;i++)
    { v[i]=rand()%100+1;
      w[i]=rand();
    }
    s=clock();
    for(j=0;j<n;j++)
      for(i=max;i>=v[j];i--)
        if(f[i-v[j]]+w[j]>f[i])
          f[i]=f[i-v[j]]+w[j];
    e=clock();
```

当 n=1000，max=10000 时，即共有 1000 个物品，每个物品体积为 1-100，最大容量为 10000 时，运行时间：78ms。当 n=3000，max=30000 时，运行时间：718ms。

2.完全背包问题

[1] 笔者《算法效率与程序优化》

```
for(j=0;j<n;j++)
 for(i=v[j];i<=max;i++)
  if(f[i-v[j]]+w[j]>f[i])
   f[i]=f[i-v[j]]+w[j];
```

当 n=1000，max=10000 时，运行时间：66.25ms。当 n=3000，max=30000 时，运行时间：593ms。注意到上述两个程序仅是内层循环枚举容量的方向不同，导致一个是 0/1 背包，一个是完全背包（允许重复的）。

3.限制个数的背包问题

```
for(j=0;j<n;j++)
 { int x[max]={0};
  for(i=v[j];i<=max;i++)
   if(x[i-v[j]]<k&&f[i-v[j]]+w[j]>f[i])
   { f[i]=f[i-v[j]]+w[j];
     x[i]=x[i-v[j]]+1;
   }
 }
```

当 n=1000，max=10000，k=10 时，运行时间：127.9ms。当 n=3000，max=30000，k=10 时，运行时间：1156.75ms。基本上是完全背包问题时间的 2 倍，并不是想象中的 k 倍，由于只增加了对 x 数组的判断与赋值，仍然是 O(n*m)的算法。注意到上述程序有只是增加了一个记录当前使用该物品个数的数组，从而实现了减少重复枚举。

NOIP2008 结束后，笔者才意识到，上述关于“限制个数背包问题”（一般称“多重背包问题”）的讨论是错误的。其中错误之处在于，对于多重背包问题的 k 件物品之间，不一定满足“最优子结构”性质，即后面取的物品不一定要从前面价值最高的那个加一个物品继承得来，而可能是从前面价值第二高的那种上面加一个得来。

如果使用正确的方法，应该使用“单调队列”。

队列里面存放决策，队列维护一个连续时间范围内的最优决策集合，“过时”的决策被弹出，队列元素的“时间戳”单调递增，则队列元素的“优劣程度”递减。

令 f[i] = max/min { f[j] + w(i, j) }，其中 w(i, j)是凸的。

令 g(a, b) 表示 min{i | i > b > a }，决策 a 没有决策 b 优。

决策队列满足 $g(i_1, i_2) < g(i_2, i_3) < \ldots g(i_k, i_{k+1})$ （一个决策一旦被另一个决策追上就永远不会优了）

每到达一个时刻(即 i++) 从队头开始把被追上的决策清理出去，然后队首的元素既是求 f[i]的最优决策，同时将 f[i]加入决策队列的队尾。

```
qh = qt = 0
f[qh] = 0
for (int i=1;i<=n;i++) {
while (qh < qt && catch[qh+1] <= i) qh ++;
f[i] = F(i, q[qh]);
while (qh < qt && findcatchup(q[qt], i) <= catch[qt])
qt --;
catch[qt+1] = findcatchup(q[qt], i);
q[++qt] = i;
}
```

这是一个 $O(n^2)$的方法，但显然编程复杂度较高。有趣的是，上面的错误的贪心方法反而正确率很高，用这个算法在 RQNOJ 上提交三道题目，一道 AC，一道 90 分，一道 70 分。而使用 $O(n^2k)$的算法，都不超过 60 分。对于不会使用单调队列的 NOIP 选手，这种贪心方法又何尝不是一种“骗分”的好选择？

类似的，很多“错误的贪心”也能取得不错的分数。这些情况的出现，正是因为“反例”难于出现，正如快速排序中线性数据较少，所以退化概率很低一样，我们才能够使用这些“看起来正确”的贪心方法。

但是，多数“错误的贪心”却要比“搜索+剪枝”要得到更少的分数，因此笔者发现的贪心只是一种巧合。其他情况中，例如 0/1 背包问题非要用“每次取最大”这种贪心策略，就得不到多少分数了。

再看一道经典贪心试题：NOIP2004“合并果子[1]”。

有 N 堆果子，每次选取两堆进行合并，合并消耗的体力为两堆果子的数目之和，求将所有果子合并成一堆消耗的最小体力。

我们首先会想，每次选取当前最小的两堆进行合并。那么，问题的关键在于，如何选出最小的两堆果子。或者说，合并后的果子应该放在哪里。

我们有几种算法。

一、堆排序。维护一个小根堆，每次取出堆顶的两个元素，求和后再插入堆中。需要编写堆中取出堆顶元素、插入元素的函数，还要特别注意元素相同的情况。编程复杂度较高。

二、一个线性队列。首先对堆的数目进行快速排序，每次取出队列头的两个元素，求和后二分查找，插入到指定位置。但这又涉及到元素的移动问题，效率较低。[2]

三、两个线性队列。初始时所有元素快速排序，进入队列 A。队列 B 为空。每次从 A、B 队列的头部选择较小的（如果有元素的话），重复两次，将选出的两个最小值相加后放进 B 队列尾。主要是排序的时间复杂度，线性队列的 O(n)是很优秀的。

正确性：A、B 两个队列均为递增队列，从 A、B 两个队列头依次选出两个元素，这两个元素是 A、B 队列中的最小元素；加和后由于比原来产生的和相同或更大（否则与选择元素的最小性矛盾），所以放在 B 队列尾，不破坏 B 的单调性。

以上一、三算法，时间复杂度均为 O(nlogn)，而两个线性队列的方法明显编程简单。这就是贪心方法的应用。

```
#include<stdio.h>
int a[10001]={0};
int p(int l,int r)//快速排序过程
{ int x=a[l],i=l-1,j=r+1,t;
  while(1)
  { do{--j;
    }while(a[j]<x);
    do{++i;
    }while(a[i]>x);
    if(i<j)
    { t=a[i];a[i]=a[j];a[j]=t;
    }
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=25

[2] 用“跳跃表”可以解决查找与插入时间的矛盾，可以称为是数组、链表的结合，但编程复杂度较高。

```
    else
     return j;
  }
}
void q(int l,int r)
{ int n;
  if(l<r)
  { n=p(l,r);
    q(l,n);
    q(n+1,r);
  }
}//本文中的快速排序，两个函数可以合成一个函数，这样看起来更紧凑，同时减少了函数调用，能提高程序运行效率。这是笔者 AC 的程序，故没有修改。
main()
{ int i,n,b[20001]={0},t,as,ae,bs,be,a1,a2,tot=0;
  scanf("%d",&n);
  for(i=0;i<n;i++)
   scanf("%d",&a[i]);
  q(0,n);
  for(i=0;i<=n/2;i++)//调转顺序，没有必要，可以在快排中改变大小比较
  { t=a[i];a[i]=a[n-i-1];a[n-i-1]=t;}
  bs=0;be=-1;as=0;ae=n-1;
  while(ae-as+be-bs>=0)//队列不为空
  { if(bs>be||as<=ae&&a[as]<b[bs])
     a1=a[as++];
    else a1=b[bs++];
    if(bs>be||as<=ae&&a[as]<b[bs])
     a2=a[as++];
    else a2=b[bs++];//取出两个元素
    b[++be]=a1+a2;
    tot+=b[be];
  }
  printf("%d",tot);
}
```

再看一道经典贪心问题“婚礼上的袭击[1]”，实质就是“田忌赛马”问题。

田忌有 n 匹马，力量分别为 a[1],...a[n]；齐王有 n 匹马，力量分别为 b[1],...b[n].田忌选择马与齐王的马一一对决，每匹马恰好出场一次，如果田忌马的力量大于齐王的，则田忌胜一局。要求恰当安排顺序，使得田忌胜尽可能多局。

算法可以用 DP，或者给每匹马连线赋权变为二分图最佳匹配，还有就是贪心了。[2]

1. 当田忌最慢的马比齐王最慢的马快，先赢一场

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=303

[2] 本题的算法分析源自飞雪天涯 http://www.rqnoj.cn/Discuss_Show.asp?DID=3513

2.当田忌最慢的马比齐王最慢的马慢，和齐王最快的马比，输一场

3.当田忌最快的马比齐王最快的马快时，先赢一场

4.当田忌最快的马比齐王最快的马慢时，拿最慢的马和齐王最快的马比，输一场。

5.当田忌最快的马和齐王最快的马相等时，拿最慢的马来和齐王最快的马比.

田忌赛马贪心的正确性证明。

先说简单状况下的证明：

1.当田忌最慢的马比齐王最慢的马快，赢一场先。因为始终要赢齐王最慢的马，不如用最没用的马来赢它。

2.当田忌最慢的马比齐王最慢的马慢，和齐王最快的马比，输一场。因为田忌最慢的马始终要输的，不如用它来消耗齐王最有用的马。

3.当田忌最慢的和齐王最慢的马慢相等时，分 4 和 5 讨论。

4.当田忌最快的马比齐王最快的马快时，赢一场先。因为最快的马的用途就是来赢别人快的马，别人慢的马什么马都能赢。

5.当田忌最快的马比齐王最快的马慢时，拿最慢的马和齐王最快的马比，输一场，因为反正要输一场，不如拿最没用的马输。

6.当田忌最快的马和齐王最快的马相等时，这就要展开讨论了,贪心方法是，拿最慢的马来和齐王最快的马比。

下面用数学归纳法证明:田忌最快的马和齐王最快的马相等时拿最慢的马来和齐王最快的马比有最优解。

1）假设他们有 n 匹马，看 n=2 的时候.　a1 a2　b1 b2　因为田忌最快的马和齐王最快的马相等 所以 a1=b1,a2=b2 所以这种情况有 2 种比赛方式,易得这两种方式得分相等。

2）当数列 a 和数列 b 全部相等时(a1=b1,a2=b2...an=bn)，显然最慢的马来和齐王最快的马比有最优解,可以赢 n-1 场，输 1 场，找不到更好的方法了。

3）当数列 a 和数列 b 元素全部相等时(a1=b1=a2=b2...=an=bn)，无法赢也不输。

现在假设 n 匹马时拿最慢的马来和齐王最快的马比有最优解，证明有 n+1 匹马时拿最慢的马来和齐王最快的马比也有最优解。

数列

a1 a2 a3 a4...an an+1

b1 b2 b3 b4...bn bn+1

其中 ai>=ai-1,bi>=bi-1

数列 a 和数列 b 不全部相等时，拿最慢的马来和齐王最快的马比数列得到数列

(a1) a2 a3 a4...an an+1

b1 b2 b3 b4...bn (bn+1)

分 4 种情况讨论：

（1）b1=b2,an=an+1

则有

a2 a3 a4...an

b2 b3 b4...bn

其中 a2>=a1,a1=b1,b1=b2,得 a2>=b2(此后这种大小关系不再论述）,an>=bn.

此时若 a2=b1,根据归纳假设，有最优解，否则 a2>b1，根据前面“公理”论证有最优解。

当且仅当 a 数列,b 数列元素全部相等时有 an+1=b1,已证得，所以 an+1>b1，赢回最慢的马来和齐王最快的马比输的那一场。

（2）b1<=b2,an=an+1

交换 b1,b2 的位置，

数列

(a1) a2 a3 a4...an an+1

b2 b1 b3 b4...bn (bn+1)

此时 a2>=a1,an>=bn,

对于子表

a2 a3 a4...an

b1 b3 b4...bn

根据前面“公理”或归纳假设，有最优解。

an+1>=b2,当且仅当 b2=b3=b4=..=bn+1 时有 an+1=b2，这种情况，a 中其它元素<=b1,b2,b3,b4..bn,对于这部分来说，能赢 x 盘(x<=n)，假如不拿最慢的马来和齐王最快的马比则拿最快的马来和齐王最快的马比，此时平一盘，能赢 x-1 盘，而拿最慢的马来和齐王最快的马比，输一盘能赢 x 盘，总的来说，还是 X 这个数，没有亏。

（3）b1=b2,an<=an+1

（4）b1<=b2,an<=an+1 证明方法类似，不再重复。

已证得当有 n+1 匹马的时候，田忌和齐王最快最慢的马速度相等时，拿最慢的马来和齐王最快的马比有最优解，已知当 n=2 时成立，所以对于 n>2 且为整数（废话，马的只数当然是整数）时也成立。

当 n=1 时，似乎不用讨论。由数学归纳法原理，可以得到贪心方法。

程序如下：

```
#include<stdio.h>
int a[100001],b[100001];
void qa(int l,int r);//a 数组的快速排序代码，略去
void qb(int l,int r);//b 数组的快速排序代码，略去
main()
{ int n,i,sa,ta,sb,tb,tot;
  while(1)
  { scanf("%d",&n);
    if(n==0)
     return 0;
    sa=sb=0;ta=tb=n-1; tot=0;
    for(i=0;i<n;i++)
     scanf("%d",&a[i]);
    for(i=0;i<n;i++)
     scanf("%d",&b[i]);
    qa(0,n-1);
    qb(0,n-1); //事实上两个排序可以用传递数组指针的方法合成一个
    while(sa<=ta&&sb<=tb) //赛马的选择过程
    { if(a[ta]>b[tb])
      { tot++;//tot 记录赢的局数
        ta--; //ta 记录田忌的队列头，sa 记录田忌的队列尾
        tb--; //b 记录齐王的情况。由于贪心讨论，两个队列都是从头尾取元素。
      }
      else if(a[ta]<b[tb])
```

```
      { if(a[ta]>b[sb])
         tot++;
        if(a[ta]<b[sb])
         tot--;
        ta--;
        sb++;
      }
      else if(a[sa]>b[sb])
      { tot++;
        sa++;
        sb++;
      }
      else
      { if(a[ta]>b[sb])
         tot++;
        if(a[ta]<b[sb])
         tot--;
        ta--;
        sb++;
      }
    }
    printf("%d\n",200*tot);
  }
}
```

5.2 随机法

所谓随机化算法，就是在程序中使用了随机化函数，使得程序运行的结果或过程受到随机函数取值的影响的算法。

并不是所有应用了随机函数 rand 的算法都是随机化算法。例如，将 rand 函数的返回值赋给一个没有在运算中应用的变量，这个 rand 的取值不影响程序的运行结果、过程，所以不是随机化算法。另一方面，一个 rand 函数不影响结果的算法不一定不是随机化算法。例如，采用随机化算法求出搜索题的一个可行解，不同的算法找到解的期望时间不同，此时 rand 函数影响了程序运行的过程，仍然属于随机化算法。

随机化算法对于程序影响的程度，一般体现在三个层次上：

1. 影响程序运行的时间。例如只需要求出一个可行解的程序，随机性选择搜索分支，“运气”好的话，很快就能搜索到正确解；反之，可能整个搜索树都快遍历完了，才找到可行解。
2. 影响程序解的优劣。例如按照解的优劣程度给部分分的题目，一个好的随机算法，能够在更大的程度上逼近最优解，从而得到更多的分数。也就是说，同样的时间，同样的枚举（随机）次数，由于随机化方向的不同，以及一些人为的剪枝、判断措施，使得解的优劣程度不同。

3. 影响解的正确性。例如，使用随机算法找到一个图的最大割，其正确性是有一定概率的，因此这样的算法影响了解的正确性。对于这样的算法，我们必须在时间允许的前提下，尽可能多次地运行这个算法，使得得到正确解的概率尽可能大。

衡量随机化算法的优劣，一般来说从以下几个方面来考虑。

1. 解的正确性。如果一个随机化算法以很小的概率出正确解，那么也就没有必要使用了。
2. 时间复杂度。如果一个随机化算法比搜索+剪枝要用更多的时间，就没有必要去“冒险”；同样，在竞赛中，我们还要衡量“编程复杂度”与“得分”之间的“性价比”，追求一个平衡点。也许一种搜索或者动态规划算法比随机化要慢很多，但题目的数据范围比较弱，完全可以通过，那么就没有必要用随机化。
3. 稳定性。这往往是衡量对于同一个问题的不同随机化算法的重要指标。稳定性一方面指时间复杂度的稳定性，即是否会因为特别的输入数据而出现退化；一方面指输出结果的稳定性，即对于不同的测试数据，这个程序有多大的概率出正确解，或者最优解；是否存在一种特殊的输入，使得这个算法根本无法出解。

在信息学竞赛中使用随机化算法，首先要注意随机化算法的稳定性，不能指望着“运气好”，而要确实有较大的概率得分；其次，对于较难的题目，我们可以采取“按照题目规模分块处理”的策略，对于规模较小、直接搜索动规不会超时的数据，可以使用确定性算法；对于规模较大，可能超时的数据，应该采用随机化算法，尽可能多得分。

上面所述的“按规模处理”策略并不局限于随机化算法，而是一种普遍适用的策略。对于规模小的数据，采用保证正确但效率较低的算法；对于规模大的数据，采用不保证正确但效率较高的算法。这也是“分类”方法的体现，尤其在 NOI 及更高级别的比赛中常用。

首先我们看一个经典的“Miller-Rabbin 素数判定算法”，这种算法不保证所判定的素数是正确的，但可以以很高的效率、很高的准确性完成判定，尤其是在不很大的数中几乎没有反例，因此在信息学竞赛中有着广泛的应用。

```
int test(int a,int b,int c)
{ int x=a,y=1;
  while(b!=0)
  { if(b&1)
     y=(y*x)%c;
    b=b>>1;
    x=(x*x)%c;
  }
  return y;
}
int prime(int a,int flag)
{ int i,r,d=a-1,t;
  while(!(d&1))
   d=d>>1;
  t=test(flag,d,a);
  while(d!=a-1&&t!=1&&t!=a-1)
  { d=d<<1;
    t=(t*t)%a;
  }
```

```
  return (t==a-1)||(d&1);
}
```

主程序：

```
s=clock();
    for(i=3;i<=n;i+=2)
     if(prime(i,2)&&prime(i,7)&&prime(i,61));
    t=clock();
```

注意：在 longint 范围内，只需测试 2、7、61。在 int64（1E16）范围内，只需测试 2、3、7、61、24251，并排除 46 856 248 255 981 这个强伪素数。

当 n=100000 时，运行时间：46ms；当 n=1000000 时，运行时间：500ms。当 n=10000000 时，运行时间：5063ms。从表面上看，这种高效算法似乎比朴素的筛法慢，其实筛法的最大缺点是占用内存过多以及不能判断单个素数。记忆化的试除法似乎差不多，但内存用量同样较大，不能判断单个素数。普通的试除则效率过低。所以，miller-rabbin 算法在只需要判断少数离散的大素数时，是十分高效的算法。

这个算法判定出的数称为“伪素数”，因其不一定是素数。通过单次判断素数得到的结果中，有很多都是伪素数，因此应用价值不大；但从结果上看，多次判断素数就几乎一定是正确的，这是因为不同的相互独立事件概率相乘，使得一个较大的错误概率变得很小，可以忽略。这也就是不保证结果正确的随机化算法的核心思想：多次随机，每次采取不同的随机种子或随机方向，甚至采用多个不同的随机化算法，使得正确的概率尽可能大。

下面这个“判定无向图的三染色”算法[1]，也许更能体现“多次随机”的本质。

将一个可三染色图分成两个不含三角形的子图：若一个无向图可以三染色，则我们可以将其顶点集合 V 分成三个不相交的子集：V = X∪Y∪Z，而每一个子集在原图中都是一个独立集(即原图中没有一条边的两个顶点同时存在于一个子集中)。由于判定图是否可以三染色的问题是 NP-Complete，因此没有有效算法可以将可三染色图分成三个不相交的独立集。

那么换一个简单一点的问题：是否可以将一个可三染色图的顶点集合分成两个不相交的子集：V = A∪B，使得没有原图中的三角形的三个顶点同时存在于一个子集中。

算法：首先随机的将 V 分成两个子集 A 和 B。对于一个不合法的三角形（三个顶点同时存在于 A 或 B）中，随机选择其中一个顶点，放到对面的子集中。这样不断的循环下去，直到没有不合法的三角形。

时间复杂度分析：原图可三染色：存在 V 的划分 V = X [Y [Z，使得原图中的任何三角形，三个顶点分别落在 X, Y 和 Z 中。

² 令 c = jXj + jY j，则 c · n。

² 假设算法初始时随机划分为 V = A [B，令 N(A;B) = jA \ Xj + jB \ Y j，显然 0 · N(A;B) · c。

² 若 N(A;B) = 0 或 N(A;B) = c，则(A;B)是一个合法的划分：没有原图中的三角形的三个顶点同时

在 A 或 B 中。

² 在算法的循环中 0 < N(A;B) < c。

² 对于任何一个不合法的三角形，算法在三个顶点 v1; v2; v3 中选择一个随机的点 v 放到对面的子集中：从

划分(A;B)到(A0;B0)，考虑 N(A0;B0) ¡ N(A;B)：

N(A0;B0) ¡ N(A;B) =

[1] 引用自周源《随机化算法简介》，2009 年冬令营讲义。

8>><

>>:

+1 当 v 2 A \ Y 或 v 2 B \ X

¡1 当 v 2 A \ X 或 v 2 B \ Y

0 当 v 2 Z

² 由于 v1; v2; v3 分别落在 X; Y;Z 中，于是 v 以相同的概率（各 1=3）落在 X; Y;Z 中，以上三项的概率都

是 1=3。

² 算法的执行相当于在[0; c]的一条线段上进行随机行走(random work)：从一个整数点出发，结束点

在 0 和 c，在行走中的每一步，各以 1=3 的概率向左走一步(-1)，向右走一步(+1)和原地不动(0)。

引理 6.1 在上述随机行走中，期望行走的步数不超过 3c2=8。

证

还有一类随机化算法，是针对时间复杂度的随机，也就是说结果是确定的，只是运行时间长短的问题。下面我们通过对“随机化的快速排序”的分析，加深对随机化算法的理解。

众所周知，快速排序的时间复杂度是 O(n logn)，我们也知道当待排序的序列恰好是最坏情况时，快速排序就变成了类似冒泡排序的东西，复杂度 O(n^2)，显然是很不稳定的[1]。

有人说，这种最坏情况出现的概率太小了。但是，可能对某个问题来说，我们遇到的输入大部分都是最坏情况或次坏情况。一种解决的办法是不用 x=A[lo]划分 A[lo..hi]，而用 x=A[hi]或 x=A[(lo+hi) div 2]或其它的 A[lo..hi]中的数来划分 A[lo..hi]，这要看具体情况而定。但这并没有解决问题，因为我们可能遇到的这样的输入：有三类，每一类出现的概率为 1/3，且每一类分别对于 x=A[lo]，x=A[hi]，x=A[(lo+hi) div 2]为它们的最坏情况，这时快速排序就会十分低效。[2]

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

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

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

[1] 这里的“稳定”指的是时间复杂度受输入数据影响很小甚至没有，不是“稳定的排序”之意。

[2] 随机化快速排序算法及分析摘录自周咏基《论随机化算法的原理与设计》，IOI2009 集训队论文。

分析随机化快速排序算法

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

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

下面是原来的快速排序和随机化后的快速排序的性能对照表。

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

由此可见，随机化后的算法虽然在随机输入中比原算法略逊一筹，但对于极端输入，优势就十分明显。在信息学竞赛中应用，快速排序很小的一点常数并不是程序的瓶颈，解决了最坏输入问题才是重要的，因为命题人可能有意设计这种极端数据，或者问题本身的特点可能出现最坏情况。因此我们推荐使用随机化的快速排序代替普通快速排序。

5.3 枚举法

5.4 调整法

当直接求出问题的解不太可能，或者只要求一个“较优”的解时，我们可以采用“逐步调整”的策略。逐步调整，就是从一个随意决定的初始状态出发，每次改变现有解的若干参量，使得目标函数比上次更大，迭代若干次后，就能得到一个较优的解。

调整法适用于目标函数为“凸函数”的问题，即从任意一个状态出发，每次进行任意方向的修改，使得解取得改进，经过无穷多次的调整后，总能到达最优值的位置。如果函数为一对一的，图像应该是凸多边形或曲线；如果函数为二对一的，图像应该是凸多面体。至于多对一的情况，可以认为是高维空间内的凸多面体。

只有保证了函数的“凸性”，才能保证“调整法”不会走进“死胡同”，即钻入多面体的一个并非最优的“角”内出不来。调整法与启发函数搜索的最大区别就在于此，调整法沿任何一个调整方向都能直通最优解，而启发函数搜索则只能得到局部最优解，要得到全局最优解还需要将多个局部合并起来。从编程的角度来说，调整法是线性结构，而启发函数搜索是有分支的树形结构。

调整法在不同的场合有不同的名称，如模拟退火算法、迭代法，但本质都是“逐步逼近”的思想。本文已经介绍的开平方根的牛顿迭代法、高斯消元法的迭代法、线性规划的单纯形法，以及最短路算法中的松弛操作、网络流中的增广路算法，都用到了“调整法”的思想。

5.5 模拟法

模拟法，

6 搜索算法

本文之所以把“搜索算法”独立一节，与上节“非完美算法”中的贪心、随机、试验、调整、模拟等方法分开来，是因为搜索算法的地位与整个非完美算法同级，换句话说，就是在信息学竞赛中，搜索算法是“无计可施”时的“最后一击”。

“我是智障”《骗分导论》中针对的主要是“非完美算法”，也就是一些巧妙的得分方法。这些方法听起来巧妙，用起来不易。所讲述的内容“止增笑耳”，难以真正在考试中起到大幅度提高分数的作用。从那篇文章最后的“实战演练”中看到，无论如何采用上述方法，得到的分数都不会很高，最后也只能是“二等奖”。但事实上，即使从功利的角度考虑，二等奖也是没有大用的，我们追求的是更高更远的目标；况且考虑信息学竞赛的目的，只要这些“小伎俩”对水平的提高无益，所以还要正统的、有实效的算法。

6.1 可行性剪枝

本文开头就提到，所有可解问题都能使用“搜索”来解决，而搜索的效率则是我们不懈的追求。一个确定的搜索方向，如何在不改变搜索整体结构的基础上优化时间效率，就需要“剪枝”——将不可能的、非最优的枝条“剪去”。

笔者的朋友肖世康编写了如下代码，并询问是否有优化的措施：

```
Private Sub Form_click()
 Dim a, b, c, d, e, f, g, h, i, j
 For a = 1 To 9
  For b = 0 To 9
   For c = 0 To 9
    For d = 1 To 9
     For e = 1 To 9
```

```
      For f = 0 To 9
       For g = 0 To 9
        For h = 0 To 9
         For i = 1 To 9
          For j = 0 To 9
           If 1101 * a + 10 * b + c + d + 100 * f + g + 1000 * e +
10 * h - 10000 * i - 1000 * j = 0
           And a <> b And a <> c And a <> d And a <> e And a <> f
And a <> g And a <> h And a <> i And a <> j _
           And b <> c And b <> d And b <> e And b <> f And b <> g
And b <> h And b <> i _
           And c <> d And c <> e And c <> f And c <> g And c <> h
And c <> i _
           And d <> e And d <> f And d <> g And d <> h And d <> i _
           And e <> f And e <> g And e <> h And e <> i _
           And f <> g And f <> h And f <> i _
           And g <> h And g <> i And h <> i Then
           Print a & b & c & d & f & g & h & i & j
           End If
          Next j
         Next i
        Next h
       Next g
      Next f
      Next e
      Next d
      Next c
      Next b
      Next a
  Text1.Text = "kkkkkk"
End Sub
```

我在邮件中这样回复：

这个程序需要实现的功能是解数字谜。针对

```
  a a b a
  e f h g
        c
+       d
---------
i j 0 0 0
```

要求 a,b...j 这十个数字恰好是 0,1,2,...9，所以进行枚举。注意部分数字只能取 1-9.

在枚举的过程中，注意通过数学方法优化。由于十个数字不能重复，应该在枚举完一个数字后进行判断，避免重复发生。本程序中采用的是 use 数组，a[i]记录数字 i 是否被使用过。在枚举到一个元素时，首先判断是否重复，不重复则进行标记，枚举完成后再将标记去除。这样保证重复的情况被尽快排除。

在计算过程中，注意到每位相加的结果末位均为 0，所以只需枚举一部分，最后一个可以直接算出。注意进位的问题。

运行结果是：

```
4 2 9 7 8 5 0 6 1 3
4 6 9 7 8 5 0 2 1 3
4 2 7 9 8 5 0 6 1 3
4 6 7 9 8 5 0 2 1 3
4 2 0 9 8 5 7 6 1 3
4 6 0 9 8 5 7 2 1 3
Problem solved.
```

源代码：

```
#include<stdio.h>
int main()
{ int a,b,c,d,e,f,g,h,i,j,use[10]={0},jin1;
 for(a=1;a<=9;a++)
 { use[a]=1;
  for(g=0;g<=9;g++)
  if(use[g]==0)
  { use[g]=1;
   for(d=1;d<=9;d++)
   if(use[d]==0)
   { use[d]=1;
    if(a+g+d<=10)
    { c=10-a-g-d;jin1=1;}
    else if(a+g+d<=20)
    { c=20-a-g-d;jin1=2;}
    else continue;
    if(use[c]==0)
    {
    use[c]=1;
    for(b=0;b<=9;b++)
    if(use[b]==0)
    { use[b]=1;
     if(b+jin1<=10)
     { h=10-b-jin1;
      if(use[h]==0)
      { use[h]=1;
       f=10-a-1;
       if(use[f]==0)
       { use[f]=1;
        for(e=1;e<=9;e++)
        if(use[e]==0)
        { use[e]=1;
         if(a+e+1<=10)
```

```
          { j=a+e+1; i=0;}
          else
          { j=a+e+1-10; i=1;}
          if(i!=j&&use[i]==0&&use[j]==0)

printf("%d %d %d %d %d %d %d %d %d %d\n",a,b,c,d,e,f,g,h,i,j);
           use[e]=0;
          }
         use[f]=0;
        }
        use[h]=0;
       }
      }
      use[b]=0;
     }
     use[c]=0;
     }
     use[d]=0;
    }
    use[g]=0;
   }
   use[a]=0;
 }
 printf("Problem solved.");
 getch();
}
```

上述程序对于多数 OIer 来说，也许十分简单。但这样的简单问题，体现着不简单的思想。

（1）观察题目的特征，注意到看似杂乱无章的代码，蕴含着“数字谜”的本质。这样的“追本溯源”思想在解决初赛的“读程序写结果”和“程序完形填空”时有重要作用。盲目模拟程序的运行，显然太慢；所以我们要找到问题对应的数学模型，进而对症下药，进行优化。

（2）在上述代码中，已经将数学的用处尽可能发挥，排除多种不可能的情况，这体现了我们“可行性剪枝”的思想。这里我们提出，**多余的枝条尽量早剪**，剪得越早，做的无用功越少，程序效率越高。

（3）当然，判重也应该有个“度”，如果判重的时间代价太高，就得不尝失了。因此我们一般采用“记录数组”的办法，使得在 O(1)的时间内实现重复判断。但需要注意记录数组的初始化和还原。即搜索开始前，记录为空；对于该元素的搜索结束后，该元素的记录清零。这两点虽然反复强调，在编写搜索程序时仍然是容易出错的地方，应当加倍重视。

进一步思考这个“数字谜”问题，我们不难发现本题事实上就是 NOIP2004《虫食算》[1] 一题。上面的讨论是针对单个例子，而这里则给出了一种解决此类问题的通用程序。

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=27

这道题的主要算法是搜索，但当然还有解方程等其他方法，这里主要介绍搜索的方法[1]。

深度搜索每个字母的值，如果盲目暴搜，显然O(n!)的复杂度会超时，那就必然要剪枝。

对于一个测试数据

5

ABCED

BDACE

EBBAA

一、剪枝：

如最后一列的D,E,A。

如果D，E，A的值都已经搜出来一种方案了，那么A =(D+E) mod n 即D+E除以n的余数是A，因为D+E有可能〉n并且进位，所以要mod n，详细一点即，对于搜索出来的D，E，A如果上一位进位则 A=（D+E+1）mod n，不进位则A=（D+E）mod n,不知道是否进位则(A=（D+E+1）mod n)or(A=（D+E）mod n)，如果满足这些条件则继续，否则退出。

如果只知道D，E则（D+E）mod n（如进位则是（D+E+1）mod n）这个数没被赋值到其他的字母上去就可以继续搜，同样只知道E，A和D，A也可以这样剪枝。

E，A：[A-E mod n]（进位则是 (A-E-1) mod n)没被赋给除D外的其它字母

D，A：[A-D mod n]（进位则是 (A-D-1) mod n)没被赋给除E外的其它字母

二、优化：

还有一个优化就是从搜索顺序的优化：搜索顺序的改变也成为大大提高程序效率的关键：从右往左，按照字母出现顺序搜索，有很大程度上提高了先剪掉废枝的情况。

三、注意：

进位情况要特别注意：

（1）如果D，E，A都知道，那么（D+E）DIV n<>0则进位否则不进（上一位进位则（D+E+1）DIV n<>0 ）

（2）如果知道D，E那么（D+E）DIV n<>0则进位否则不进（上一位进位则（D+E+1）DIV n<>0 ）

特殊情况：如果上一位不确定是否进位那么又要分情况讨论：①如果进位，不进位两种情况中只有一种情况合法（即所确定的数符合剪枝条件）,那么就按这种情况确定是不是进位。②如果两种情况都合法则，如果两种情况的进位是否都相同，那么可以确定这一位是否进位，不同的话则进位状态赋值为待定。

（3）知道A，E那么 A<E则进位否则不进（上一位进位则（A-1）<E ）

特殊情况：同（２）。

（4）知道A，D那么 D<E则进位否则不进（上一位进位则（A-1）<D ）

特殊情况：同（２）。

（5）只知道D,E,A中一个的值，则进位状态待定。

程序源代码：[2]

```
#include <iostream>
#include <string>
using namespace std;
bool finish, hash[256], used[27];
int n, stk[27];
string a, b, c;
```

[1] 本题的解题报告摘自 http://www.rqnoj.cn/Solution_Show.asp?DID=2888，原作者为 Withflying。

[2] 由于很多选手尚未通过此题，并且虫食算也是一个优化搜索的经典问题，所以贴出源代码。

```
string word;
void init() {
cin >> n >> a >> b >> c;
finish = false;
}
void outsol() {
int i, ans[27];
for (i = 0; i < n; i ++)
ans[word[i] - 65] = stk[i];
printf("%d",ans[0]);
for (i = 1; i < n; i ++)
 printf(" %d",ans[i]);
finish = true;
}
void addup(char ch) {
if (!hash[ch]) {
hash[ch] = true;
word = word + ch;
}
}
string change(string str, char x, char y) {
for (int i = 0; i < n; i ++)
if (str[i] == x)
str[i] = y;
return str;
}
void pre_doing() {
word = "";
memset(hash, 0, sizeof(hash));
for (int i = n - 1; i >= 0; i --) {
addup(a[i]); addup(b[i]); addup(c[i]);
}
memset(used, 0, sizeof(used));
}

bool bad() {
int p, g = 0;
for (int i = n - 1; i >= 0; i --) {
if (a[i] >= n || b[i] >= n || c[i] >= n) return false;
p = a[i] + b[i] + g;
if (p % n != c[i]) return true;
g = p / n;
p %= n;
}
```

```
return false;
}

bool modcheck() {
int i, p, p1, p2, g = 0;
//a + b = c, all know
for (i = n - 1; i >= 0; i --) {
if (a[i] >= n || b[i] >= n || c[i] >= n) continue;
p = (a[i] + b[i]) % n;
if (!(p == c[i] || (p + 1) % n == c[i])) return true;
}

//a + ? = c
for (i = n - 1; i >= 0; i --) {
if (!(a[i] < n && c[i] < n && b[i] >= n)) continue;
p1 = (c[i] - a[i] + n) % n;
p2 = (p1 - 1) % n;
if (used[p1] && used[p2]) return true;
}

//? + b = c
for (i = n - 1; i >= 0; i --) {
if (!(a[i] >= n && c[i] < n && b[i] < n)) continue;
p1 = (c[i] - b[i] + n) % n;
p2 = (p1 - 1) % n;
if (used[p1] && used[p2]) return true;
}

//a + b = ?
for (i = n - 1; i >= 0; i --) {
if (!(a[i] < n && b[i] < n && c[i] >= n)) continue;
p1 = (a[i] + b[i]) % n;
p2 = (p1 + 1) % n;
if (used[p1] && used[p2]) return true;
}

return false;
}

void dfs(int l) {
int i;
string A, B, C;

if (finish) return;
```

```
if (bad()) return;
if (modcheck()) return;

if (l == n) {
outsol();
return;
}

for (i = n - 1; i >= 0; i --)
if (!used[i]) {
used[i] = true; A = a; B = b; C = c;
a = change(A, word[l], i);
b = change(B, word[l], i);
c = change(C, word[l], i);
stk[l] = i;
dfs(l + 1);
used[i] = false; a = A; b = B; c = C;
}
}

int main() {
init();
pre_doing();
dfs(0);
return 0;
}
```

下面再来看一个著名的“造币厂问题”：[1]

n 个造币厂生产同一种硬币，但其中某些厂由于材料问题造出了非标准的硬币.设标准的硬币重 c 克(已知),非标准的硬币只有一种重量,重 c(1+e)克,其中 e 是个不为 0 的未知数,可以取正数或负数.为了查出哪些厂生产的硬币是非标准的,从各厂中抽出一些样品(个数不限)放在一起进行称重,只能称 2 次.设 ai,bi(i=1,...,n)分别为第一次,第二次称重从第 i 个厂中取得的样品个数.每个厂提供的样品或者都是标准的,或者都是非标准的.设 pn=Σmax(ai,bi)为某一称重方案的样品总数,求一种使 pn 最小的称重方案.

这个问题目前尚未得到完整的数学化解决，因此只能通过搜索的方法处理。

（1）采用格雷码生成 2n 的不同的 n 元二进制串，从而将要在每个串上进行的 n 个除法、2n 个乘法和 2n 个加法减少为至多 n 个比较与 4 个加法。

（2）搜索中，当 a[i],a[i-1],b[i-1]选定且 a[i]=a[i-1]时，要搜索 b[i]，只对大于 b[i-1]的值进行搜索，从而在很大程度上减少了重复搜索。

（3）利用当 i≠j 时，（a[i]/b[i]）≠（a[j]/b[j]）可以减少一些多余的搜索。

（4）限定 a[1]最大，b[1]=0，对于 a[1]，按升序搜索，对于 a[i]，（i>1）采用降序搜索，可尽快找到所需要的解。

[1] 算法来自李学武《造币厂问题》

6.2 最优性剪枝

在很多要求“最优解”的试题中，“可行解”的总数是异常庞大的，求出所有可行解，再判断其最优性，显然是不可能完成的任务。所以在搜索过程中，应当特别注意解的筛选，即将“非最优”的枝条剪去，只留下“最优解”。

必要时，还可以实现使用贪心算法构造一个“较优”的解，作为约束起点，在其基础上进行调整，使得解更优；或者另起炉灶，但有一个解的最优性限制。

我们先来看原创的一道数学试题。

将正整数 $1,2,...2009^2$ 填入 2009×2009 的方格表中, 使得每个数恰好出现一次. 称任意两个有公共边的方格中所填数的较大者与较小者之比为“牛比”. 显然, 共有 2×2009×2008 个大于 1 的牛比. 这些牛比中的最小值称为“最小牛比”. 试求所有符合条件的填数方案的“最小牛比”的最大值.(不要化成最简分数,只要令分子分母为方格内的两个数即可)

笔者命制此题时未多考虑，就给出了结果：$\frac{4035076}{2018040}$

一种可行的填数方案是：从左上角开始，按行自上而下，按列从左至右，对奇数行填奇数列的方格，对偶数行填偶数列的方格，依次填入 1,2,3...。形成类似国际象棋盘的格局。然后对剩下的数，按行自上而下，按列从左至右，依次填入未填数的方格。这样，最后一行倒数第二列与倒数第二行倒数第二列两方格数之比即为所求。证明留给读者。

后来笔者希望找到一个数学化的证明，于是编写了下面的搜索程序。[1]

```
#include<stdio.h>
#define MAX 10000
int n,f[10][10]={0},fenzi,fenmu,ans[10][10]={0},use[100]={0};
double max=1;
double check()
{ int i,j;
  double t,min=MAX;
  for(i=0;i<n-1;i++)
   for(j=0;j<n;j++)
                                                                    {
t=ans[i][j]>ans[i+1][j]?ans[i][j]*1.0/ans[i+1][j]:ans[i+1][j]*1.0
/ans[i][j];
     if(t<min)
      min=t;
   }
  for(i=0;i<n;i++)
   for(j=0;j<n-1;j++)
                                                                    {
t=ans[i][j]>ans[i][j+1]?ans[i][j]*1.0/ans[i][j+1]:ans[i][j+1]*1.0
/ans[i][j];
```

[1] 为了保持原状，未去掉调试时的输出语句、主程序中的一些冗余运算，程序代码事实上可以再精简。
程序说明：输入为方格大小 n，开始改变的参数 x，y（如果全部搜索设为 0），输出为最新找到的较优的解。

```
    if(t<min)
     min=t;
   }
  return min;
}
void solve(int x,int y,double min)
{ int i,j;
  double t,mint;
  if(x==n)
  { if(min>max)
    { max=min;
      for(i=0;i<n;i++)
       for(j=0;j<n;j++)
        ans[i][j]=f[i][j];
         printf("%lf %lf\n",max,check());
       for(i=0;i<n;i++)
       { for(j=0;j<n;j++)
          printf("%3d",ans[i][j]);
         printf("\n");
       }
    }
    return;
  }
  for(i=1;i<=n*n;i++)
   if(!use[i])
   { mint=min;
     if(x>0)
     { t=(i>f[x-1][y]?i*1.0/f[x-1][y]:f[x-1][y]*1.0/i);
       if(t<mint)
        mint=t;
       if(mint<max)
        continue;
     }
     if(y>0)
     { t=(i>f[x][y-1]?i*1.0/f[x][y-1]:f[x][y-1]*1.0/i);
       if(t<mint)
        mint=t;
       if(mint<max)
        continue;
     }
     use[i]=1;
     f[x][y]=i;
     if(y==n-1)
      solve(x+1,0,mint);
```

```
    else solve(x,y+1,mint);
    use[i]=0;
   }
}
int main()
{ int i,j,startx,starty;
  scanf("%d %d %d",&n,&startx,&starty);
  for(i=0;i<n;i++)
   for(j=i%2;j<n;j+=2)
    f[i][j]=ans[i][j]=n*n/2+1+(n*i+j)/2;
  for(i=0;i<n;i++)
   for(j=(i+1)%2;j<n;j+=2)
    f[i][j]=ans[i][j]=(n*i+j)/2+1;
  if(ans[n-1][n-2]>ans[n-2][n-2])
  { fenzi=ans[n-1][n-2];  fenmu=ans[n-2][n-2];}
  else
  { fenmu=ans[n-1][n-2];  fenzi=ans[n-2][n-2];}
  max=fenzi*1.0/fenmu;
    printf("%lf %lf\n",max,check());
  for(i=0;i<n;i++)
  { for(j=0;j<n;j++)
     printf("%3d",ans[i][j]);
    printf("\n");
  }
  for(i=0;i<startx;i++)
   for(j=0;j<n;j++)
    use[f[i][j]]=1;
  for(j=0;j<starty;j++)
   use[f[startx][j]]=1;
  solve(startx,starty,MAX);
  printf("answer\n%lf %lf\n",max,check());
  for(i=0;i<n;i++)
  { for(j=0;j<n;j++)
     printf("%3d",ans[i][j]);
    printf("\n");
  }
  getch();
}
```

通过以上程序，我们能够发现多处优化。

（1）使用 t 表示一个较长的式子，避免重复计算

（2）首先给出一种“较优”的构造，在此基础上搜索，减少“较差”构造的机会

（3）及时舍去不必要的“枝条”，对当前不是最优的立即舍去

事实上，这个优化并不是最好的。还可以考虑

（1）在现有较好的构造基础上进行数字的对调，而不是“另起炉灶”重新枚举

（2）构造启发函数

但以上两种方法都有很大难度，笔者尚未想到。

在编写程序时，为了保证正确性，笔者采用了 check()函数，使用牛比的定义，检测最后输出的解是否正确，于是输出了两个相同的比值。这看似没有必要，但对于复杂的题目，可以检查程序在计算目标函数时是否存在问题。

下面是对于一些 n 的搜索结果。

当 n=2 时，比值为 1.5（3/2）

3	1
2	4

当 n=3 时，比值为 1.75（7/4）输入 3 1 0

5	1	6
2	7	3
8	4	9

当 n=4 时，比值为 1.857143（13/7）输入 4 2 0

1	9	2	10
11	3	12	4
5	13	6	15
14	7	16	8

当 n=5 时，比值为 1.916667（23/12）输入 5 2 0（未验证全部）

13	1	14	2	15
3	16	4	17	5
18	6	19	7	22
8	20	9	23	11
21	10	24	12	25

当 n=6 时，比值为 1.941176（33/17）输入 6 3 0（未验证全部）

19	1	20	2	21	3
4	22	5	23	6	24
25	7	26	8	27	9
10	32	11	28	12	29
35	16	33	13	30	14
18	36	17	34	15	31

当 n=7 时，比值为 1.958333（47/24）输入 7 4 0（未验证全部）

25	1	26	2	27	3	28
4	29	5	30	6	31	7
32	8	33	9	34	10	35
11	36	12	37	13	38	14
39	15	40	16	41	17	46
18	42	19	44	20	47	23
43	21	45	22	48	24	49

我们发现，事实上“交叉排”的填数方案并不是最优的。真是“意料之外，情理之中”。

可以证明[1]，比值无论如何也会小于 2，由于分子分母都是 $1-n^2$ 的数，最大的比值就是 $\frac{2[(n^2-1)/2]-1}{[(n^2-1)/2]}$．这里对于偶数的情况，容易证明是“最大可能的”；但对于奇数的情况，最大可能的要更大一些。例如 n=4 时，本应该是 9/5，实际却显然是 7/4；n=5 时，理论最优值为 25/13=1.923，但实际搜索结果却只有 23/12，所以总结出上述表达式，是有依据的。为什么奇数时这个“理论值”达不到呢？但对于较大规模的数据，并未搜索出确实的证据，更谈不上证明了。[2]

但从搜索结果上看，我们的表格又与“交叉填”十分相似，仅是个别数据发生了交换。从枚举范围可见，最大值也只是对后几行（上面的结果表明是后一半）的同类格子进行内部交换后得到，没有出现“前一半”与“后一半”发生交换的现象。这也很好理解：最优解的要求“充分接近 2”是相当苛刻的，一个数字的更换会导致牛比大幅下降。这就给我们进一步优化搜索提供了方案：

（1）将整个棋盘划分成“交叉”的国际象棋棋盘模样，分别对黑色区取较大的一半数，白色区取较小的一半数。

（2）对每个区域进行内部枚举，时间复杂度降为 $O(\sqrt{A})$，其中 A 为原来的复杂度。

（3）可以采用“交换”的方法进行优化，还是因为较为复杂而没有采用

（4）仍然提供部分修改的功能，对后面几行进行修改，而不变动前面。因为结果显示，在最优棋盘中，前面几行基本是保持不动的，只是后面几行的少数元素发生了位置交换。

读者可以尝试进一步优化这个程序，或者让程序运行更长的时间，以获得更优的结果。看似简单的“牛比”问题，事实上绝非易事。这也提示我们，程序的优化是无止境的。一个搜索算法，可以“百嚼不厌”，时时有更新。

6.3 局部贪心、动态规划

6.4 启发式搜索

7 骗分攻略

7.1 数学题——观察

数学题的特点，就是规律性明显，而难以使用直接搜索的方法求解。这时，找到数字背后蕴藏的规律，便成为我们解题的重要手段。[3]

[1] 这是很有意思的一道组合证明题，参加数学竞赛的读者可以尝试。
[2] 这里将这道原创问题给予读者讨论，希望您有新的发现，并与我交流。
[3] 数学方法的内容在本文前面已经讲过，上文所述是数学方法，而这里主要是找规律的手段。

例如胡伟栋教练讲的一道试题：在一个给定的无向连通图 G 中，A 每次向一个节点处扔一颗炸弹，B 每次从一个节点移动到相邻的节点。A 是不能看到 B 的，所以需要事先确定扔炸弹的序列。问是否存在一定能炸死 B 的轰炸序列。

读者也许第一反应是：先对小数据搜索一下再说。但仔细想一想，这个问题显然不是搜索问题，因为 A 的决策有很多种，B 的对策又有很多种，况且本题要求的是“存在”性问题，这无穷多种情况是枚举不完的。所以我们要先进行数学分析。

在进行分析之前，我们首先需要列出几个“显而易见”的事实：

（1）先移动和先扔炸弹是相同的，所以约定先扔炸弹，后移动。

（2）如果某次轰炸后，仍然有可能有人的位置集合与之前的某个状态相同，那么期间进行的轰炸无效，是不成功的。

（3）如果一次轰炸前某位置的旁边有人，且没有被炸死，那么在下一次移动中有可能到达此位置，标记为有人。

首先考虑只有一个点、一条线段的“平凡”情况，显然能够炸死。对于 3 个点连成一条线的情况，不妨从左至右设为 A、B、C，则先炸 A，A 中一定没有人；下一步移动，A、B、C 又都有可能有人，因为 B、C 仍然是不定向移动的，回到原来状态，轰炸失败。先炸 C 的道理相同。所以如果能够炸死，一定先炸 B。第二次轰炸前，A、C 的人一定到达 B，再对 B 进行轰炸，一定炸死。

三个点的连接方式除了一条线，还有“环”的可能。这时轰炸 A、B、C 是等效的，不妨设为 A。轰炸完后，剩下两侧的点 B、C 可能补充过来，导致 A 又有人，轰炸无效。所以三元环是不可能炸死的。类似的，n 元环也不可能炸死。不论炸哪个位置，由于环的连通性，两侧的点都可以补充过来，也可以由另一侧的点移动过来，所以不成功。我们得到一个重要的结论：只要图中存在环，就不可能炸死。下面只需要考虑“树”的情况。

从简单入手，考虑 4 个点 A、B、C、D 成为一条线的情况。先炸 B，可能到 B、C、D；再炸 C，可能到 A、C、D；炸 D，可能到 B、D；再炸 D，可能到 A、C；炸 C，可能到 B；炸 B，轰炸成功。因此轰炸序列 BCDDCB 是可行的。我们发现一个有趣的事实：从 2 号点依次向右轰炸，再从右侧端点依次向左轰炸，最后一定炸死。

仔细思考，发现是有依据的。对 n 个点排成一条线的情况，用数学归纳法：假设现在已经正向轰炸到了第 k 个点，那么前面 k-1 个点中只有一半的点是有人的，它们相间排列，形成 k-2, k-4, k-6... 1 或 2 的序列。轰炸 k 个点之后，由于前面的点必须移动一步，于是改变了奇偶性，形成 k-1, k-3, ...2 或 1 的序列。第 k 个点刚被轰炸，显然没有人；后面的点不受影响。于是，我们的奇偶相间序列向前推了一位，到达 k+1 个点处。如此归纳，直到轰炸完第 n 个点，这 n 个点一定成奇偶相间排列，亦即只有一半的点有人。

下面的思考就简单了。既然一次轰炸能使有人的点控制在一半，且与奇偶有关，那么换一个奇偶性，再轰炸一次，就能使有人的点在另一半，两者取交集，轰炸成功。为了避免奇偶讨论，我们采用了“反向轰炸”的方法，即从 n 开始，逆着原来的步骤轰炸。我们惊喜的发现，由于奇偶性相反，反向轰炸点右侧都不再有人，左侧的人正在被逐一消灭。

这不禁使我们想到物理学中波的传播与干涉：一列波随着时间的推移，波峰与波谷作周期性变化；波的传播需要时间，波前之前的按振动规律振动，波前之后的尚未开始振动。如果从反向引入一个频率相同、相位相反的波，两列波的叠加会发生干涉现象，导致反相波传到的位置振动停止。

于是，我们解决了一个基础的，也是至关重要的问题：一列点的情况，可以从头开始，逐个轰炸到尾部，再从尾部开始逐个轰炸到头部，就能保证轰炸成功。

对于其他较为复杂的树形结构，我们想，利用化学上的思想，找到“最长链”恐怕十分重要。从直观上想，如果整棵树错综复杂，“旁生枝节”很长很多，就很难找到可以堵住的

地方。于是我们从有三个长度为 2 的链的简单情况入手。

1—2—3—4—5
　　|
　　6
　　|
　　7

(1) Bomb 2, Save 2,3,4,5,6,7
As we mentioned before, bombing the first point is useless. For simple, we start with the second. Since the tree structure is similar to a straight line, it is reasonable to try again.

(2) Bomb 3, Save 1,3,4,5,6,7
It's possible to follow the pattern of line sturcture. So have a try. In case of this "go right and come left" method also works, we will succeed.

(3) Bomb 6, Save 2,3,4,5,6
It's not clever to leave the subtree out. If we only work on the main chain, how can the tree differ from the line? So go downstairs.

(4) Bomb 3, Save 1,3,4,5,7
Not too deep and fail to come up! It is time to get back to the main chain. In this way, we can not only keep the past odd-even situation from right cone to left again, but also make the subtree the way we want.
Mention: Subtree 6-7 has been separated by odd and even, it is a prediction of success.

(5) Bomb 4, Save 2,6,5
Poor man! Now he can only survive in half of the left part, and there is no doubt that he will be killed when bombing left in the opposite direction of odd or even.

(6) Bomb 5, Save 1,3,7
The first round of bombing is finished successfully. Our wishes come true, and the Death is coming to the poor man.

(7) Bomb 5 for another time, saving 2,4,6, waiting for the odd and even changes.

(8) Bomb 4, Save 1,3,7

(9) Bomb 3, Save 2,6

(10)Bomb 6, Save 1

(11)Bomb 3, Save 2

(12)Bomb 2, the bombing action is successful without any doubt.[1]

由上述的描述，读者可能已经发现，只要支链深度不超过 2，就可以用上述方法解决。因此矛盾的焦点指向了支链长度不小于 3 的情况。

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

显然，不同于上个例子，如果首次轰炸支链上的点，旁边的点会立即补充过来，导致轰炸失败。所以首次轰炸一定要选择中间节点 4. 我们要证明，不存在这样的轰炸序列，亦即

[1] 这里的英文是笔者为了调动读者的积极性特意写的。笔者尚未知此题的提交地址。

要找到一种对策，不论如何轰炸都能逃脱。

(1) First bomb 4 is centain. Goto 4 after bombed.

(2) If bomb 1 or 2, it's useless and back to (1).
So must bomb one of 3,5,8. Take 3 into consideration.
Then we have to hide in 5 or 8.

(3) If not bomb 4 again, go back to 4, situation back to (1).
If bomb 4, we have to go to 6 or 9.

(4) If not bomb 5 or 8, we go back to it immediately. Remenber: Any reappear of situation will lead to a infinite circulation.
So must bomb 5 or 8. Take 5 as an example.
Then we go to 7,8,10, avoiding the bomb.

(5) If bomb 4, go back to 6 or 9 to make a circle.
So must bomb a non-4 point.
But 8 is near to 4, and we can go there!
The situation (1) reappears, and the bomber has no way out.

因此，只要存在形如上例这样支链深度不小于 3 的情况，就一定不存在这样的轰炸序列。

总结上面的讨论，我们可以得出存在轰炸序列的充要条件：

（1）图中不存在环

（2）树中支链深度不超过 2

下面就是具体的程序实现。

首先执行的一定是“判断不存在环”的操作，使用拓扑排序即可解决。

下面就需要判断树中支链的深度不超过 2。有几种方案可以选择。

1.利用树的结构，首先进行一次深度优先搜索，计算每个节点子树的最大深度。对于一个至少有两棵子树的节点，如果它的父枝、两个子枝均深度大于 3，则退出。

```
Caculate the depth of each vertex
For each vertex I in tree
 If it has not less than 2 subtrees
 { For each subtree
    If depth >= 3  //Here depth includes itself
     Number++;
  If number >= 3
    Return 0;
  If number >= 2
   { if(fa[fa[fa[I]]]!=fa[fa[I]]) //third father exists
       Return 0;
     If(fa[fa[I]!=fa[I]) //second father exists
      If(depth[fa[I]] >= 2)
         Return 0;
     If(fa[I] != fa[I]) //not root
      If(depth[fa[I]] >= 3)
       Return 0;
   }
 }
Return 1;
```

2.模仿有机化学中的方法，首先找到主链（最长的链），这一步可以使用 NOIP“树网的核”一题的算法解决。然后从主链上的每个分支点出发，对侧链进行深度优先搜索，一旦深度达到 3 则失败退出。

3.考虑到以上算法中找到主链并没有起到关键的作用，所以简单的对每个节点进行深度优先搜索，一旦深度达到 3 则失败退出。

4.考虑到“深度为 3”的特殊情况，忽略掉树的“上下级”性质，而只利用“无环”性质，可以直接寻找三条链，每条链长度不小于 3，则失败退出。

```
For each vertex I in the tree
 If nextnumber[I] >= 3
 { number=0;
   For each next[I]
    If next[next[I]] not empty
     For each next[next[I]]
      If next[next[next[I]] not empty
       Number++;  //next[I] is a chain of 3 vertexs
   If number >= 3
    Return 0; //I has 3 subtrees with depth of 3
 }
Return 1;
```

以上几种方案，显然都是可行的；但哪种方案更加容易编程实现，也是显而易见的。因此我们在思考问题时，不能仅仅停留在表面上，不能想到一个“可行”的算法就急于动手编程，而要思考现在的算法是否是最优的，是否抓住了问题的本质，是否对这个问题思考得透彻了。如果每次解题都多思考一步，就能得到更加本质、更加简单自然的算法。

纵观全局，枚举思考的过程是由浅入深的。在每个特例进行尝试构造解决之后，通过观察特例代表的类型及处理问题的方法，归纳出这一类结构的处理通法。

（1）长为 3 的链，熟悉问题，了解基本事实

（2）长为 3 的环，进而否定所有环，限定到树形

（3）长为 4 的链，发现“折返”的轰炸方式，进而推广到所有链形

（4）3-3-3 的支链结构，在链形的基础上，找到“深入子树一层”的轰炸方式，进而推广到所有支链深度不超过 2 的情况

（5）4-4-4 的支链结构，通过构造对策，否定支链深度不小于 3 的情况

以上问题的解决，是一个较为复杂的探究过程。其中数学思想的应用占据主导地位。尤其是联想到有机化学中“主链”的思想，从侧链长度出发提出猜想，更是常人不易想到的，也使这道题的难度提高了一个档次。

但从特例出发，选取有代表性的小数据通过手工计算，寻找、发现规律，进而进行验证的思想，在有数学味的信息学问题的解决中，有着重要作用，是我们应该掌握的。

下面我们再来看一个通过观察得到递推关系，从而解决问题的例子：纸牌定位[1]。

一叠纸牌共有 n 张，先把第一张丢开，把第二张放到底层，然后把第三张丢开，把第四张放在底层，如此进行下去，直至只剩最后一张牌，问这张牌是原来的第几张牌？

我们首先对较小的情况进行试验。

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=477

7.2 小数据——交表

交表这种大家在 OJ 上惯用的“骗分”伎俩，不用再下定义了；就是针对

7.3 交互式试题——贪心

有这样一道试题：随机生成 N 个数，每次向交互库发出询问，每次可以询问 M 个数的大小关系。例如，询问 a1, a2, a4，若 a2>a1>a4, 交互库会返回 a2, a1, a4. 程序的任务是，尽可能快的将 M 个数排序。程序的得分由询问的次数决定。

我们考虑 M 较小的情况，例如 M=2. 显然，这就是我们常用的排序算法：每次比较两个数的大小。采用快速排序，至少要用 O(nlogn)次比较。

当 M=64，N=8 时，就是清华大学 2009 年自主招生数学试题。原题的要求是通过至多 50 次比较，我们可以给出 49 次的一种方法：

（1）对 a1-a8，a9-a16,...,a56-a64 分别排序，共用 8 次。

（2）利用归并的思想，将 a1-a8 和 a9-a16 合并成一个大组。不妨设 a、b 均为不减的。首先取出 a1-a4, a9-a12，排序后前四大的数一定是 a1-a16 中最大的四个。同理，取出 a5-a8，a13-a16，排序后后四小的数一定是 a1-a16 中最小的四个。这样，剩下中间 8 个的顺序尚未确定，再执行一次排序即可。

如此共用 3 次，将 a1-a16 排序，类似的合并 a17-a20,a21-a24，共用 3*4=12 次。

（3）再次对 a1-a16,a17-a32 进行归并。仍然采用上一步的策略，每次确定 4 个最大元素的顺序。具体步骤：取出 a1-a4,a17-a20 排序，得到 b1-b4；除去已经放入 b 的 4 个元素，再分别取出 a1-a16,a17-a32 中的 4 个未除去的最大元素，得到 b5-b8。以此类推，经过 32/4-2=6 次，只剩下两组中最小的 8 个元素。此时对它们进行排序，共用 7 次即可。

由此可见，对于 4n 个数的两组归并排序，需要 n-1 次即可完成。此步共用 14 次。

（4）对 a1-a32，a33-a64 进行归并，使用 64/4-1=15 次，完成排序工作。

共用 8+12+14+15=49 次。

从上例的解答过程中，我们不难发现一些规律：

7.4 提交答案题——枚举

对于提交答案类试题，测试数据中的输入部分已经给出。那么针对给定的输入，五个小时的机器时间一定不能闲着，要让计算机“动”起来，这就是本节要介绍的“枚举”法。

对于搜索类问题，我们可以采用限制深度、加贪心条件、加估价函数的方法，在一定范围内搜索，得到一个“较优”的解。

对于毫无头绪的问题，搜索工作量太大，我们只能采用“枚举”的办法。例如 WC2009 第三题，就是一个难以用简单搜索算法解决的问题。但可喜的是，本题部分分很多，又有判断程序正确个数的 checker 工具，于是“随机化”成为当时多数选手的算法。好的随机算法，甚至能得到 60-80 分，但难度远低于针对不同测试点分别设计的标准算法。[1]

下面的一段程序，就是 C 语言中用于批处理的程序段。这个程序的作用是，每次生成一

[1] 笔者由于当时对随机化了解不充分，也没有接触过提交答案题目，于是简单的比较、输出了几组全 0、全 1、01 相间，只得了 21 分，甚至只有 5 个变量的第一个点都没有做出来。

个随机数，并添加到字符串中，利用 system 函数调用子程序，给子程序每次随机的输入。

```
#include<stdio.h>
int main()
{ int i,t;
  char a[12]={"check 10000"};
  randomize();
  while(1)
  { t=rand();
    for(i=0;i<5;i++)
    { a[6+i]='0'+t%10;
      t/=10;
    }
    system(a);
  }
}
```

下面一段程序，就是我们编写的随机化计算程序。这个程序的主体部分就是我们平时编写的随机化算法，使用随机数仍然是 rand()，但主函数的接口处有一些变通，目的是接受上面程序段的随机数，指导 srand 随机数初始化。

```
#include<stdio.h>
#include<stdlib.h>
int main(int argc,char *argv[])
{
srand(argv[1][0]*10000+argv[1][1]*1000+argv[1][2]*100+argv[1][3]*
10+argv[1][4]);
  printf("%d\n",rand());
}
```

有人可能会说，这样写真是“脱了裤子放屁”，只要每次调用函数，并在开始时进行 rand 初始化，不就行了吗？但我们需要了解 randomize 的运行机理，是使用系统时间来初始化函数的。在同一秒中运行这个程序，只会得到同样的结果。读者可以将 while(1)中的内容改成直接调用，函数初始化使用 randomize，就会发现屏幕输出是每秒钟变化一次的。这就意味着每秒钟很多次的随机化只有一次有效，太浪费了。于是采用了含参数调用函数的方法。

在 Pascal 中，含参数调用函数较为复杂，所以可以采用“文件辅助”的变通方案。C 语言也可以参考。具体做法是，主程序将生成的随机数写进文件，再调用子程序，子程序从文件中读入随机数，实现初始化。

进行调用的主程序：[1]

```
#include<stdio.h>
int main()
{ FILE *fp;
  randomize();
  while(1)
  { fp=fopen("rand.txt","w");
```

[1] 由于读者对 C 比较熟悉，仍使用 C 语言编写，读者可以自行改成 Pascal 语言。

```
    fprintf(fp,"%d",rand());
    fclose(fp);
    system("check");
  }
}
```

实现随机化功能的 check 程序：

```
#include<stdio.h>
int main()
{ int n;
  FILE *fp;
  fp=fopen("rand.txt","r");
  fscanf(fp,"%d",&n);
  srand(n);
  printf("%d\n",rand());
  fclose(fp);
}
```

在实际运行时，发现硬盘灯频繁闪烁，证明在频繁读写文件。要注意，主程序中必须在每次调用之前关闭文件，下次修改时再重新打开。如果写成只打开一次文件的形式，刚修改的内容仍在硬盘缓存中，就调用了读文件函数，容易导致文件读写紊乱；并且删除刚写的一个数，也是困难的事情。

我们相信，**骗分不是不可能，只是暂时没有找到方法**。对症下药，量体裁衣，各个击破，针对不同的问题采用合适的“骗分”手段，就能收到意想不到的效果。

7.5 全面地考虑问题[1]

在编程序时常常会遇到这样的问题:一道很简单的题目,编出的程序却错了很多测试点。这其中的主要原因是由于考虑问题不全面，只想到了一些普通的情况，而遗漏了很多特殊的地方。

下面通过几个例子来进行讨论。

1.项链(IOI'93 第一题)

由 n(n≤100)个珠子组成一个项链，珠子有红、蓝、白三种颜色，各种颜色的珠子的安排顺序由输入文件任意给定。

图 1.1 可看作由字符 b(代表蓝色珠子)和字符 r(代表红色珠子)所组成的字符串。假定从项链的某处将其剪断，把它摆成一直线，从一端收集同种颜色珠子(直到遇到另一种颜色的珠子时停止)。然后再从另一端重复上述过程(请注意，这一端珠子的颜色不一定和另一端珠子的颜色相同)。

brbrrrbbbrrrrbrrbbrbbbbrrrrb

图 1.1

请选择项链被剪断的位置，目标是使两端各自颜色相同的珠子数目之和最大。例如，对

[1] 本文摘自《1993-1996 美国计算机程序竞赛试题与解析》。
清华大学出版社 吴文虎 主编 赵鹏 编著
本文作者 赵鹏 是原信息学奥林匹克竞赛中国队的金牌得主
转载自 http://218.22.18.86/info/Data_Structures_and_Algorithms/contest/index.html

于上图(只有红蓝两种颜色)，最大值 M 是 8，断点位置在珠子 9 和珠子 10 之间，或珠子 24 和珠子 25 之间。

项链中可以有三种颜色用 b(蓝)、r(红)和 w(白)表示。白色既可看成是红色，又可看成蓝色。

(1)一个 ASCII 文件 NECKLACE.DAT 中的内容:该文件中每一行代表某个项链中各种颜色珠子的配置。把输出内容写入 ASCII 输出文件 NECKLACE.SOL 中。

作为举例，输入文件的内容可以是:

brbrrrrbbbbrrrrrbbrbbbbrrrrb

bbwbrrrwbrbrrrrb

(2)对于给定的每个项链的配置，求出收集到的珠子数的最大值 M 及相应的断点位置(注意可能存在多个位置)。

(3)在输出文件 NECKLACE.SOL 中写入收集到的珠子数的最大值 M 及断点位置。

例如:

brbrrrbbbrrrrrbbrbbbbrrrrb

8 between 9 and 10

bbwbrrrwbrbrrrrrb

10 between 16 and 17

作为竞赛的第一题，这道题目显然是比较简单的题目。它只包含两个步骤:剪断项链和收集同颜色的珠子。例如下面的一条项链(a)从 N=3 处断开变为项链(b)。这个操作只需要将前 N 个珠子移到后边即可。

```
    brb | rrwb ------> rrwbbrb
   (a)                (b)
```

现在只剩下收集同颜色的珠子这一步，根据上面的例子我们很容易写出下面的程序。

用变量 c 来记录最左边珠子的颜色;

```
Left:=0;
FOR i:=1 TO 项链长度 DO
IF 左数第 i 个珠子的颜色与 c 相同
THEN Inc(Left)
ELSE Break;
```

这样变量 Left 中存放的就是从左边收集到的珠子的数目，同理可求得从右边收集到的珠子的数目 Right，则所求的值为 Lett+Right。这个程序显然能通过上面的例子，由于这是一道简单的题目，谁也不想在它上面多费时间，往往做到此为止。可是如果仔细想想， 再举几个例子，就会发现错误。上面的那条项链断开后左有两个珠子为红色和蓝色，在题目中这两种颜色的珠子都比较"普通"，只有白色的珠子比较"特殊"。所以应举一个断开后左右两端有白色珠子的例子。还是上面那条项链入 N=6 处断开。

brbrrw| b------->bbrbrrw

正确答案应是收集到 5 个珠子:左边 2 个，有边 3 个。而上面的程序得到的结果却是 3 个:左边 2 个，右边 1 个。错误就在于没有考虑到左右两端有白色珠子的情况。一种较容易的解决方案是先将左有两端的白色珠子均取下，记其数目为 Other，再用上面的程序来求，结果为 Left 十 Right 十 Other。我们解决了左右两端出现白色珠子的情况，还有没有别的特殊情况呢?一个真正"特殊"的项链不应包含所有颜色的珠子，最好只包含一种颜色。 如下面的项链是由 10 个红色的珠子组成。

rrrrrrrrrr

用我们的程序得出的结果是 20 个，显然是不对的。因为题目中要求是收集珠子而不是

数珠子，所以最后得到的总数不应超过珠子的总数。这虽然只是一个字眼的问题，却使当年中国队的选手失了不少分。一个简单的改正措施是判断最后的结果是否大于珠子总数，如果是则输出珠子的总数即可。

虽然项链这道题比较简单，却很难"简单"地得到满分，最容易出的错误就是考虑的不全面。

2.多项式加法

由文件输入两个多项式的各项系数和指数，编程求出它们的和，并以手写的习惯输出此多项式。

要求:

(1)多项式的每一项 axb 用 axb 的格式输出。

(2)两个多项式在文件中各占一行，每行有 2m 个数，依次为第一项的系数，第一项的指数，第二项的系数，第二项的指数……

例如输入文件为:

1 2 3 0

-1 1

输出:

x2-x+3

此题是一道大学生竞赛的题目，很多人只用了很短的时间就编出程序。但最后测试的结果却令他们很惊讶:通过的数据还不到一半!他们犯的错误归根结底就是考虑得不够全面。

此题对于多项式相加的过程很简单，只要找出幂次相同的项相加即可。关键在于题目中要求用符合手写的习惯输出结果。何为手写的习惯呢?例如多项式 3x2-x 中就有很多手写的习惯。我们不会将其写成 3x2 一 1x1+0。因为首先当某项系数为 1 时，我们习惯于不写系数;其次对于一次项我们也要省略指数;还有我们从来不写出系数为 0 的项。一个简单的多项式就有这么多的手写习惯，我们已经感觉到了要把这题全面地做出很不容易。虽然我们平时总在写多项式,但是谁也不会留心我们写多项式时的习惯。我们写多项式的习惯究竟有哪些呢?

(1)首先我们考虑对于多项式中的任一项 axb 它有多少手写习惯:

当 a=0 时，此项省去不写;

当 a=1 时，省去 a;

当 a=-1 时，系数只写一个负号'-';

当 b=0 时，省去 x 和 b;

当 b=1 时，省去 b;

当 a<一 1 时，省去此项前面的加号(首项除外)。

我们一口气写了这么多条规则，每一条看起来都很正确，但合在一起是否还正确呢?当 a=1 或-1 时要省去其中的数字 1，这是针对一般情况而言。如果 b=0，则数字 1 就不应当省去。所以我们不仅要单独考虑 a 和 b，而且要将其和起来考虑。

(2)其次对于整个多项式有哪些规则呢?

多项式的首项系数前不应有加号'+';

如果一个多项式为零多项式，则应写出数字'0'。

现在看起来这道题并不是一道很容易的题目。它需要一个人在很短的时间内能全面地总结出上述那么多规则。这对一个人的全面考虑问题的能力是一个很好的检验。

3.求最长的公共子串(NOI'93 第一题)

求 N 个字符串的最长公共子串，N<20，字符串长度不超过 255。例如 N=3，由键盘 依次

输入 3 个字符串为

What is local bus ?

Name some local buses.

local bus is a high speed I/O bus close to the processor.

则最长公共子串为"local bus"。

此题也是作为第一题出现，同样有很多入在此题上失分。我们都做过求 n 个数最大公 约数的问题，在那道题中求 3 个数的最大公约数时，可以先求两个数的最大公约数，再将此数与第三个数求一次最大公约数。有人从那道题中得到"启发"，设 s(p,q)为字符串 p 和 q 的最长公共子串，则 p、q、r 的最长公共子串为 s(s(p,q),r)。这样只需编写一个求两个字符串的最长公共子串的过程即可。但这种方法对不对呢?看看下面的例子。

三个字符串分别为'abc'、'cab'、'c'，则 s(p,q)='ab',s(s(p,q).r)=''。事实上这三个字符串有公共子串'c'。显然上面的算法是错误的，原因在于没有考虑到本题与求最大公约数那道题在性质上的不同之处。最大公约数可以由局部解得到全局解，而本题却不能。正确的做法是列举出其中一个字符串的所有子串，找出其中最长的而且是公共的子串。

```
FOR i:=1 TO 第一个字符串的长度 DO
FOR j:=i TO 第一个字符串的长度 DO
IF (第 i 个字符到第 j 个字符的子串为公共子串)AND(j-i+1>当前找到的最长公共子串的长度 max)
THEN
BEGIN
max:=j-i+1;
最长公共子串:=此子串;
END;
```

为了提高效率，我们可以将最短的字符串作为第一个字符串。此题需要考虑的并不像多项式加法那道题那么多，但是它提醒我们在不清楚题目的性质之前，不能滥用以前的方法。

4.可重复排列(NOI'94 第一题)

键盘输入一个仅由小写字母组成的字符串，输出以该串中任取 M 个字母的所有排列及排列总数(输入数据均不需判错)。

此题是由全排列问题转变而来，不同之处在于:一个字符串中可能有相同的字符，导致可能出现重复的排列。例如从字符串'aab'中取 2 个字符的排列只有三种:'aa'、'ab'、'ba'。如何去掉那些可能重复的排列呢?一种想法就是每产生一种不同的排列就记录下来，以便让以后产生的排列进行比较判重。这种想法显然没有考虑到随着字符串长度的增加，排列将会多得无法记录，而且这种判重方法在效率上也会很低。最好有一种方法能在产生排列的过程中就能将重复的去掉。先看一看全排列的递归过程

```
PROCEDURE Work(k);
BEGIN
 IF k=m+1 THEN 打印结果
      ELSE FOR i:=1 TO 字符串长度 n DO
        IF (i<>e[1],e[2],e[k-1]) THEN
         BEGIN
          e[k]:=i;
          Work(k+1);
         END;
```

```
END;
```

让我们来分析产生重复的原因。考虑从字符串。'aab'中取 2 个字符的排列。当 e[1]从 1 变到 2 时，字符串中的字符却没有变，都是'a'。这样我们只要在改变 e[k]时，判断其对应的字符是否也改变。即在上面的过程的循环中加一句判断(设字符串为 s):IF s[i]<> s[e[k]] THEN …; 这当然只是一个粗略的想法，我们仅仅用上面的例子就能发现问题: 程序在对 e[k]的每一次赋值之前都要进行一次判断，而不是我们预想的在改变 e[k]时才进行判重。我们用一个布尔型的局部变量 First 来记录是否是对 e[k]进行第一次赋值。修改后的程序如下:

```
PROCEDURE Work(k);
BEGIN
First:=True;
IF k=m+1 THEN 打印结果

ELSE FOR i:=1 TO 字符串长度 n DO
IF (i<>e[1],e[2],e[k-1]) THEN
BEGIN
First:=false;
e[k]:=i;
Work(k+1);
END;
END;
```

很多选手的程序到此就为止了，可是它还有一个致命的错误:我们在判重时假定这个字符串中的字符已经排好顺序，即相同的字符已经连在一起。事实上并不是这样，输入的字符串中的字符排列是任意的,需要我们在递归之前作一次排序的初始化才能保证程序运行得正确。可见虽然本题的难点是在递归过程中，但是那些简单的初始化却是程序最容易忽略，最容易出错的部分。

5·删除多余括号(IOI'94 国家队选拔赛第一题)

键盘输入一个含有括号的四则运算表达式，可能含有多余的括号，编程整理该表达式，去掉所有多余的括号，原表达式中所有变量和运算符相对位置保持不变，并保持与原表达式等价。

例:输入表达式

a+(b+c)

(a*b)+c/d

a+b/(c-d)

应输出表达式

a+b+c

a*b+c/d

a+b/(c — d)

注意输入 a+b 时不能输出 b+a。

表达式以字符串输入，长度不超过 255。输入不要判错。

所有变量为单个小写字母。只是要求去掉所有多余括号，不要求对表达式化简。

同多项式加法一样，此题要求的也是一个我们平时很习惯但却没有注意的规则:去掉多余的括号。哪些括号是多余的呢?这要根据紧挨着括号前后的运算符号和括号中最后一个运

算符号来决定。例如表达式 a+(b*c+d)-e 中，括号前的符号为"+"，括号后的符号为"-"，括号中最后一个运算符号为"+"，所以括号是多余的。总结括号中最后一个运算符号为"+"、"-"、"*"，"/"时，括号前、后为何运算符号时括号是多余的，我们很容易得出表 1.1。

表 1.1 多余的括号满足的条件

括号前的符号	括号中的符号	括号后的符号
+	+	+、-
+	-	+、-
+、-、*	*	+、-、*、/
+、-、*	/	+、-、*、/

上表只是对一般情况的分析，显然是很不全面的。首先括号前，括号中和括号后都可能无运算符号，例如表达式((a))+b；其次变量前还可能带有负号，例如表达式(-a)+ (-b)中的括号一个是多余的，一个不是多余的。还有一些其它的情况如表达式只有括号无任何变量等需要考虑。此题留给大家自己去完成。注意多用一些特殊的数据来测试自己的程序。大家也可以试着用表达式树的方法来做此题。

6.合并表格(NOI'93 第二题)

在两个文本文件中各存有一个西文制表符制成的未填入任何表项的表结构，分别称之为表 1 和表 2，要求编程将表 1 和表 2 按下述规则合并成表 3。

规则:表 1 在上表 2 在下，表 1 与表 2 在左边框对齐，将表 1 的最底行与表 2 的最顶行合并。

例如，3 张表见图 1.2。

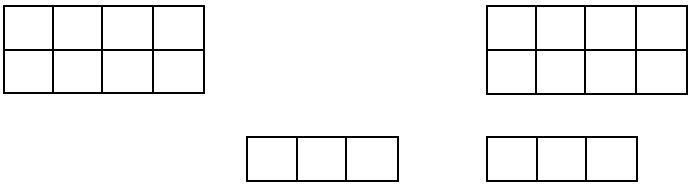


表 1　　表 2　　表 3

图 1.2

编程要求:

(1)程序应能从给定的文本文件中读入两个源表并显示。

(2)若源表有错，应能指出其错(错误只出在表 1 的最底行和表 2 的第一行)。

(3)将表 1 和表 2 按规则合并成表 3，并显示之。

(4)所有制表符的 ASCII 码应由选手自己从给出的示例文件中截取。

我们把此题的分析留给读者去独立完成。

"千里之堤，毁于蚁穴"。一些程序往往不是错在算法上，而是错在考虑得不全面上。希望通过以上几个例子，能使大家对此引起重视，在以后的编程中多加注意，避免不应有的遗憾。

7.6 设计获胜策略[1]

一个好的取胜之道是制定在竞赛中指导你行动的策略。无论是在好的情况下还是在坏的情况下，它将帮助你决定你的行动。用这种方法你可以在竞赛中将时间花费在解决编程问题上而不是试图决定下一步该干什么……这有点像预先计算好你面对各种情况的反应。

心理上的准备也很重要。

竞赛中的策略

首先通读**所有的**题目；草拟出算法，复杂度，数量，数据结构，微妙的细节，……

- 集体讨论所有可能的算法 —— 然后选择最“笨”但却可行的算法。（注：请注意这一点，对参赛选手来说获奖就是唯一目的）
- 进行计算！（空间和时间复杂度，并且加上实际期望和最坏情况下的数量）
- 试图证明该算法错误——使用特殊的（退化的）测试数据。
- 将问题排序：根据你所需付出的努力，将能够最快解决的问题排在前面。（答题的次序为：以前做过的，容易的，不熟悉的，难的）

编写程序解决一个问题 —— 对每一道题而言，一次一道题

- 确定算法
- 构造特殊情况的测试数据
- 写出数据结构
- 编写并测试输入子程序（编写额外的子程序来显示数据输入的正确性）
- 编写并测试输出子程序
- 逐步细化：通过写注释来刻划程序的逻辑轮廓
- 一个部分一个部分地填充并调试代码
- 完成代码使其正常运转，并验证代码的正确性（使用一般情况的测试数据）
- 试图证明代码错误——使用特殊情况的测试数据来验证代码正确性
- 逐渐优化——但足够了即可，并且保存所有的版本（使用最坏情况的（即运行时间长的）测试数据来计算出实际运行时间）

时间安排策略和“故障控制”方案

制定一个计划决定在各种（可预测的）故障发生时的行动；想象你可能遇到的问题并计算出你所希望做出的反应。核心问题是：“你何时花费更多的时间在调试程序上，你何时放弃并继续做下一题？”。考虑以下问题：

- 你已经花费了多长时间来调试它？
- 你可能有什么样的 BUG？
- 你的算法有错吗？
- 你的数据结构需要改变吗？

[1] Receipted from http://ace.delos.com/usacogate/
Translated by Starfish
转载自 http://218.22.18.86/info/Data_Structures_and_Algorithms/contest/index.html

- 你是否对什么地方可能会出错有一些头绪？
- 花费较短的时间（20 分钟）在调试上比切换去做其他别的事要好；但是你或许能够在 45 分钟内解决另一个问题（A short amount (20 mins) of debugging is better than switching to anything else; but you might be ableto solve another from scratch in 45 mins.）
- 你何时返回到一个你先前放弃的问题？
- 你何时花费较多的时间优化一个程序，你何时放弃当前优化工作而切换去作其他事？
- 从这点考虑出去（Consider from here out）——忘记先前的努力，着眼于将来：你如何才能就你目前所有的抓住下一个小时。

在你上交你的答案之前列出一个校验表：

- 在竞赛结束前五分钟结束编写代码（Code freeze five minutes before end of contest）。
- 将所有的声明关闭。
- 将调试输出关闭。
- 确认输入输出文件名正确。
- 确认输入输出格式正确。
- 重新编译并再测试一次。
- 将文件以正确的文件名复制到正确的位置（软盘）。

提示和技巧

- 如果可以就用暴力法（即穷举法）解决 （注：居然将这条作为技巧，可见竞赛的目的就是获奖，为此要“不择手段”。）
- KISS(=Keep It Simple, Stupid)：简单就好！(KISS: Simple is smart!)
- 提示：注意限制（在问题陈述中指明）
- 如果可以给你带来方便的话就浪费内存（假如你能侥幸逃脱规则处罚的话）
- 不要删除你额外的调试输出，将它注释起来
- 逐渐地优化，足够了即可
- 保留所有的工作版本
- 编码到调试：
    - 注意留有空白（whitespace is good）
    - 使用有意义的变量名
    - 不要重复使用变量
    - 逐步细化
    - **在写代码之前先写注释**
- 有可能的话尽量避免使用指针
- 避免使用麻烦的动态内存：静态地分配所有的东西。
- 尽量不要使用浮点数；如果你不得不使用，在所有使用的地方设置允许的误差（绝对不要测试两个浮点数相等）
- 如何写注释：

  - 不要写得太长，简洁的注解就可以了
  - 解释复杂的功能：++i; /* increase the value of i by 1*/ 这样的注释是毫无意义的。
  - 解释代码中的技巧
  - 将功能模块划定界限并且归档（Delimit & document functional sections）
  - 注释要好像是写给某个了解该问题但并不了解程序代码的聪明人看的
  - 对任何你不得不考虑的东西加以注释
  - 在任何你看到了以后会问“他到底干什么用”的地方加注释（Anything you looked at even once saying, "now what does that do again?"）
  - 总是注释数组的索引次序

- 记录你每一次竞赛的情况：成功之处、犯的错误，以及何处你可以做得更好；利用这些记录来改进你的策略。

复杂度

基础和阶符号

复杂度分析的基本原理围绕着符号大“O”，例如：O(*N*).这意味着算法的执行速度或内存占用将会随着问题规模的增倍而增倍。一个有着O(N^2)的算法在问题的规模增倍时其运行时间将会减慢4倍（或者消耗4倍的空间）。常数时间或空间消耗的算法用O(1)表示。这个概念同时适用于时间和空间；这里我们将集中讨论时间。

一种推算一个程序的O( ) 运行时间的方法是检查它的循环。嵌套最深的(因而也是最慢的)循环支配着运行时间，同时它也是在讨论O( ) 符号时唯一考虑的循环。有一个单重循环和一个单层嵌套循环（假设每个循环每次执行N次）的程序的复杂度的阶是O(N^2)，尽管程序中同时有一个O(*N*)循环。

当然，递归也像循环一样计算，并且递归程序可以有像O(b N)， O(N!)， 甚至O(N N)的阶。

经验法则

- 在分析一个算法以计算出对于一个给定的数据集它可能要运行多长时间的时候，第一条经验法则是：现代（1999）计算机每秒可以进行10M次操作。对于一个有五秒钟时间限制的程序，大约可以处理50M次操作。真正优化的好的程序或许可以处理2倍甚至4倍于这个数目的操作。复杂的算法或许只能处理这个数目的一半。
- 640K确实是苛刻的内存限制。幸运的是，1999-2000赛季将是这个限制的最后一次起作用。
- 2^{10}约等于10^3
- 如果有k重嵌套的循环，每重大约循环N次，该程序的复杂度为O(N^k)。
- 如果你的程序有 *l* 层递归，每层递归有 *b* 个递归调用，该程序复杂度为O(b^l)。
- 当你在处理有关排列组合之类的算法时，记住N个元素的排列有N!个，N个元素的组合或N个元素组成的集合的幂集的有2^n个。
- 对N个元素排序的最少时间是O(*N*log*N*)。
- **进行数学计算！**将所有的数据加起来。(Plug in the numbers.)

例子：

一个简单的重复 N 次的循环复杂度为 O(*N*)：

```
1 sum=0
2 fori=1ton
3 sum=sum+i
```

一个双重嵌套循环的复杂度通常为 $O(N^2)$：

```
#fill array a with N elements
1 fori=1ton-1
2 forj=i+1ton
3 if(a[i]>a[j])
swap(a[i],a[j])
```

注意，虽然这个循环执行了 $N\times(N+1)/2$ 次 if 语句，但他的复杂度仍然是 $O(N^2)$，因为 N 加倍后执行时间增加了四倍。

解决方案的范例

产生器 vs. 过滤器

产生大量可能的答案然后选择其中正确的（比如 8 皇后问题的解答），这样的程序叫做过滤器。那些只产生正确答案而不产生任何错误节点的叫做产生器。一般来说，过滤器较容易（较快）编程实现但是运行较慢。通过数学计算来判断一个过滤器是否足够好或者是否你需要尝试制作一个产生器。

预先计算

有时生成表格或其他数据结构以便快速查找结果是很有用的。这种方法叫做预先计算（在这里用空间换取时间）。你可以将需要预先计算的数据和程序一起编译，在程序开始时计算；也可以干脆记住预先计算出的结果。比如说，一个程序需要将大写字母转化为小写字母，可以不需任何条件地利用一个表格进行快速查找来实现。竞赛题经常要用到素数——生成一长串素数在程序中某处使用通常是很实用的。

分解（编程竞赛中最困难的事）

虽然在竞赛中经常使用的基本算法不超过 20 种，但是某些需要将两种算法结合才能解决的组合型问题却是很复杂的。尽量将问题不同部分的线索分离开来以便你可以将一个算法和一个循环或其他算法结合起来以独立地解决问题的不同部分。注意，有时你可以对你的数据的不同（独立）部分重复使用相同的算法以有效地改进程序的运行时间。

对称

许多问题中存在着对称（例如，无论你按哪一个方向，一对点之间的距离通常是相同的）。对称可以是 2 路的（？？ 原文是 2-way），4 路的，8 路的或是更多的。尽量利用对称以减少

运行时间。

例如，对于 4 路对称，你只需解决问题的四分之一，就可以写下 4 个结果，这四个结果和你所解决的一个结果是对称的（注意自对称的解答，他当然只应该被输出一次或两次）。

正向 vs. 逆向

令人惊讶的是，许多竞赛题用逆向法解决比正面突破要好得多。以逆序处理数据或构造一种基于某种非明显的方式或顺序检索数据的突破策略时，要特别小心。

简化

某些问题可以被改述为一个有点不同的其他问题，这样你解决了新问题，就已经有了原始问题的答案或者容易找出原始问题的答案；当然，你只需解决两者之中较容易的那个。另外，像归纳法一样，你可以对一个较小的问题的解答作一些小小的改变以得到原问题的完整答案。

8 调试程序

调试程序，往往是编程中最耗时的阶段，程序越复杂，特点越明显。例如一个 Windows 系统的开发过程中，要经过数百万次，乃至上千万次调试，才能保证尽可能少的出现错误，尤其是明显的、严重的错误。即使如此，Windows 还是漏洞连连、补丁不断，这充分说明了调试程序的困难性。

在信息学竞赛中，如果读者仍然认为编写程序代码用去了大部分解题时间，那么您的编程水平就有待于提高了。对于简单的程序，确实不太需要调试；但对于较为复杂的程序，调试的困难性则是众所周知的。一般来说，50 行的代码几乎不用调试，100 行的代码调试时间与编程时间相仿，而 200 行的代码调试时间远大于编程时间，这也是“程序越长，错误率越高”的原因所在。

8.1 从静态查错开始

所有有关“程序调试”的书籍、资料中都无一例外的提到了“静态查错”这个重要的方法。很多选手习惯于编写完程序后就设计测试数据，一看输出结果与正确答案相去万里，就忙于了解程序运行的内部原因，或编写输出语句，或利用编译器的断点功能，往往苦思冥想不得其果，浪费了大量时间，还严重影响心情。事实上，问题往往在于很小的一处失误。

在程序刚刚写完时，应该做的第一件事，就是通读整个程序，发现一些“低级错误”并及时修正。这就是所谓的“静态查错”。

针对程序设计中出现的种种错误，笔者将其归结为下列几类，并“对症下药”，结合具体实例，给出预防错误、排查错误的一些技巧。

8.1.1 编译错误

编译错误，就是指不能通过编译的问题。这种错误是最容易排除的。通常由于变量未定义、忘记分号、括号不匹配等问题，只要遵循编译时的错误提示，对相应的语句进行修改即可。

但修改编译错误是，务必注意**不能改变程序的逻辑结构**。对于括号不匹配问题，要注意将括号加在哪个位置才符合算法逻辑。如果发现错误，则有可能存在更大的问题——程序逻辑设计错误。尤其要注意 break，continue 语句的使用是否恰当。

对于变量未定义问题，不能盲目的直接“缺什么补什么”，而要看看所缺的变量是否是必需的，是否是由于笔误导致，并注意变量冲突问题。尤其是修改后的程序出现这类问题，更要引起重视。

编译错误看似容易排除，但如果方法得当，就能发现更大的问题，节省调试时间。

但更重要的是预防错误，平时编写程序时应该养成习惯，先定义变量再使用变量，不要出现错误后再去补。编程竞赛不是打字竞赛，不要忙着向下写程序，写每一行时都要注意括号匹配、句末分号等“语法”问题。对于程序框架的问题，例如 Pascal 中的 begin 和 end，应该采用合理的缩进，保持逻辑结构清晰，还可以使用编译器的高亮匹配功能，避免错误。尤其是一段代码复制粘贴到其他处时，应特别注意“相容性”的问题，不仅是括号，变量问题也要注意。

8.1.2 运行错误

运行错误，就是程序运行过程中错误退出，包括陷入死循环、栈溢出、数组溢出、被零除等“硬性错误”，程序没有输出便终止了。

对于存在递归函数的程序，“闪一下”往往是栈溢出的标志。栈溢出，原因一般是递归层数过多。下面的小程序可以测试堆栈深度，在 DOS 命令行（Windows）或终端（Linux）下运行，程序输出的最后一个最大的数，即为递归的最大深度。

在 Windows XP SP2，Dev-cpp 下测试，深度约为 130000；在 Redflag Linux 7.0，gcc 下测试，深度约为 780000。如此大的差距，证明了 Linux 在程序设计中有着得天独厚的优势。

```
#include<stdio.h>
void stack(int n)
{ printf("%d\n",n+1);
  stack(n+1);
}
int main()
{ int n;
  stack(0);
}
```

从上述测试可以看出，当正规比赛用 Linux 评测时，为了保险起见，最好控制堆栈深度在 500000 以内。事实上，这样大的深度完全够用了，一般情况下的深度优先搜索，不会超过 10000 层。而对于图的连通块之类的问题，建议使用 Floodfill，即广度优先搜索，这样

效率更高。于是堆栈深度应该不是我们需要担心的问题。

但有时很简单的递归程序出现了堆栈溢出，原因可能是

（1）忘记递归终止条件，或者条件不充分，导致在边界处停留或超出边界

（2）循环递归，例如快速排序中间点同时被两个区间包含的错误

这时检查递归函数，往往能发现细节上的失误。

陷入死循环，则往往是程序算法设计不完善的问题。

（1）变量重复，如内层、外层循环变量相同，导致内层修改了外层的循环控制变量，外层无法跳出。往往是笔误造成，仔细观察，静态查错就能发现问题。

（2）对于 while 型循环，忘记循环控制变量自增，多出口时忘记一个出口的改变，是易于出错的地方。查看每个出口处的循环控制变量自增情况。

（3）对于广度优先搜索，忘记标记判重，不能保证仅出入队一次，导致循环出入队。查看 visit 数组的变化情况、是否有入口判断、是否有标记语句。

（4）continue 和 break 用错，检查逻辑结构，想清楚到底是要退出到哪一层。

（5）没有加上循环结束的标志，或者结束标志错误，常常出现在 while 语句中根本不可能达到的语句作为终止条件，例如 0/1 的区别，a[n]与 a[n-1]的区别。

数组溢出，往往是数组空间定义不够导致的。我们建议使用#define MAX 或 const 预定义出最大规模，在程序中直接引用，便于修改，也不容易出错。并且要预留出少量空余空间，例如规模为 10000 的程序最好开 10005 的数组。还有其他方面的技巧，本文中已经介绍，不再赘述。如果仍然出现数组溢出问题，可能是忘记循环控制变量的退出标志，对于数组来说会导致溢出，假设空间无限大，则会导致死循环。排查错误方式类似。

被零除，往往是在计算中由于未考虑特殊情况导致的错误。

（1）在解析几何或计算几何问题中，忽略了直线斜率不存在的情况，特殊处理即可。这时可以使用一个事先定义的很大的 MAXINT 代替无穷大。

（2）在高斯消元法解方程中，忽略了无穷多组解、无解的情形，特殊处理即可。这时视情况讨论，忽略此方程，判定无解，或其他决策。

（3）在处理有关实数的问题中，一个很小的数被当作“机器零”，使用误差判断即可。关于实数的比较，一般都要采用“允许误差”的方式，不能直接处理。

（4）表达式求值时中间步骤计算错误。

被零除，意味着程序在其他步骤出现错误，或者没有对“零”这种特殊情况进行特殊处理。一般来说，如果程序其他地方无误，对“零”进行特殊判断、特殊处理之后，就可以解决问题。

8.1.3 变量冲突

变量冲突，就是外层程序的变量在内层中不经意的修改了，导致程序出错。这也是编写较复杂的程序时容易出错的一点。

下面针对不同类型的冲突提出解决方案。

（1） 循环变量冲突

例如这样一段简单的程序：

```
for(i=0;i<n;i++)
```

```
 for(j=0;j<n;i++)
```

或程序段

```
for(i=0;i<n;i++)
 for(j=0;i<n;j++)
```

都会导致死循环。但这样的错误在快速编写程序时会经常发生。

```
for(i=0;i<n;i++)
{ for(j=0;j<n;j++)
  { ... }
  for(i=0;i<n;i++)
  { ... }
}
```

上述程序会导致程序在执行完一次外部循环后结束，由于内层循环控制变量与外层发生重复。在分情况处理的问题中，这类错误是常见的。

例如在分解质因数的问题中：

```
  for(i=2;i<=n;i++)
  { for(j=0;t>1;j++)
    { tmp=0;
      while(i%p[j]==0)
      { tmp++;
        i/=p[j];
      }
      if(tmp>a[j])
       a[j]=tmp;
    }
  }
```

事实上应该将 t 赋值为 i，对 t 进行运算，而不能在 i 的基础上直接操作。

（2） 临时变量冲突

用于临时记录的变量发生重复，常发生在内外层之间。

```
while(max!=flag)
{ flag=max;
  while(...)
  { flag=0;
    for(...)
     if(...)
      flag=1;
    if(flag)
     break;
  }
  max=flag;
}
```

上面的程序问题很显然，在于 flag 记录值在运算过程中被改变，导致不能起到临时记录 max 值的作用。解决类似的问题，方法很简单，就是保证程序中各处使用的临时变量各不

相同。例如，t 专门用作 swap(a,b)，flag 专门用于标记某处是否被访问，不要为了节省变量而重复利用，因为单个变量占用的空间很少，一旦重复非常麻烦。

（3） 全局变量冲突

```
int n;
int solve(int n)
{ int n;}
int main()
{ int n;}
```

在上段程序的四个位置全局变量、主函数、其他函数、函数调用的参数中，同时出现了同样功能的变量 n，这显然会导致错误。事实上四个位置中，只有两个函数中可以分别出现而互不干扰。

如果 n 仅在主程序中应用，例如输入输出，就把它定义成主函数的内部变量；如果 n 在整个程序中反复应用，就把它定义成全局变量。当然，Pascal 语言对这两种是没有区分的。

如果 n 是一个不随程序运行而变化的常量，最好定义成全局变量，不要作为函数的参数反复传递，这样容易出错。类似的，递归函数产生的最大值最好也使用全局变量进行更新，尽量少使用函数返回值。

使用全局变量：

```
int max=0,n;
void solve(int pos,int now)
{ int i;
  if(pos==n)
  { if(now>max)
     max=now;
    return;
  }
  for(...)
   solve(pos+1,now+...);
}
main()
{ solve(0,0);
  printf("%d",max);
}
```

使用返回值传递：

```
int n;
int solve(int pos,int now)
{ int i,t,max=0;
  if(pos==n)
   return now;
  for(...)
  { t=solve(pos+1,now+...);
    if(t>max)
     max=t;
```

```
  }
  return max;
}
main()
{ printf("%d",solve(0,0));
}
```

从代码长度、实现复杂度上看，两种算法似乎没有本质区别。但事实上，使用返回值传递不仅增加了程序反复递归传递返回值的时间，更重要的是对于判断最大值更新的操作，全局变量算法只是在达到目标状态时进行更新，而返回值算法需要对每种可能的情况自底向上进行归纳更新，显然多了一些操作次数。所以从常数时间复杂度考虑，使用全局变量是较好的选择。

从算法的可扩展性来说，因为目标状态往往不只需要记录最大值，还要保留最优状态，或进行其他操作，这样全局变量法利于在最后对目标状态进行处理，十分方便。

但使用全局变量，也给我们避免变量冲突提出了更高的要求。全局变量只能在工作函数中达到目标状态时被修改，其他时候是“只读”的，不能对其再进行重复利用，否则会出现记录被覆盖的情况。

“以静代动”，是信息学中的重要思想。例如我们常见的线段树，都是采用自顶向下，逐步剖分的动态申请方法。这种方法有不易浪费空间的优点。可是，如果区间又小又零散，建出来的线段树类似一棵完全二叉树，就得不偿失了。既浪费了指针域的空间，又使得动态申请内存、指针运算繁琐易错。这时我们不妨“从反面考虑问题”，化动为静，采用自下而上逐步汇总的方式，在线性结构的数组中建立一棵完全二叉树。

例如 1-8 为叶子节点，逐次向上归纳，第三层为(1,2),(3,4),(5,6),(7,8)，第二层为(1,2,3,4),(5,6,7,8)，再加上根节点，依次存放在数组中。

这样一来，节点父子之间、兄弟之间只需通过简单的*2,/2,+1,-1 即可完成，还能通过位运算加速，实现很多“奇妙”的功能，将整棵树通过简单的几种操作动态的联系起来，既节省了空间，又方便了编程。

静态线段树的优点远不止于此。对于二维甚至高维线段树，它的优势更加明显。普通二维线段树，如果有 n 层，则插入一条线段需要在第一维、第二维的每个节点插入，如果有 T 条线段，需要 Tn^2 次运算。利用静态线段树，可以将一维看做一个“有层次的”线性表，二维就是一个二维数组（矩阵），首先填入 T 条线段，然后逐级向上归纳。

1,1	2,1	3,1	4,1	1-2,1	3-4,1	1-4,1
1,2	2,2	3,2	4,2	1-2,2	3-4,2	1-4,2
1,3	2,3	3,3	4,3	1-2,3	3-4,3	1-4,3
1,4	2,4	3,4	4,4	1-2,4	3-4,4	1-4,4
1,1-2	2,1-2	3,1-2	4,1-2	1-2,1-2	3-4,1-2	1-4,1-2
1,3-4	2,3-4	3,3-4	4,3-4	1-2,3-4	3-4,3-4	1-4,3-4
1,1-4	2,1-4	3,1-4	4,1-4	1-2,1-4	3-4,1-4	1-4,1-4

如上表所示，整个 7*7 表格被分成了 9 个部分，记录了 4*4 个元素的线段树。白色表示两维均为叶子；绿色表示一维为二级，另一维为叶子；灰色表示两维均为二级；黄色表示一维为根节点，一维为叶子；橘红色表示一维为根节点，一维为二级节点；青色表示两维均为根节点。我们看到不需要其他任何附加信息，十分简洁。

逐层向上归纳的方案是，绿色节点由两个对应的白色节点产生，例如 1,1+2,1->1-2,1.

黄色节点由两个对应的绿色节点产生，黄色由绿色产生。灰色可由上面或左面的绿色产生，结果相同。红色、青色也是如此。实现时可以按行序计算，也可以按列序计算。也就是说，可以先计算每维内部的，再对两个维度之间的公共部分进行计算。这样一来，维与维之间、维内部的元素之间联系更紧密，二维线段树的代码也就更优美了。同时，三维线段树，可以对应成一个“立方体”，即三维数组，具有可推广性。

至于时间复杂度，由于表格的规模仅为原来的 4 倍（这是由于等比数列的性质），平均每次插入仍然是常数复杂度，而不是经典算法的 $O(n^2)$，其中 n 为树的深度。

由此可见，动态内存在算法中并不是“万金油”，有时静态的数组可以比指针更好用。于是笔者在本文中一直提倡“化动为静”“化递归为线性”，使程序的逻辑结构更简单，致力于“Keep it simple and stupid”的“骗分”原则。

事实上，上述“静态”的方法是我在今年寒假解题时受到“完全二叉树”和“堆”的启发结合起来的，因为“堆”就有线性、可直接运算的特点，将其运用到线段树上，岂不更好？新的算法也许并不成熟，但毕竟能够体现我们创新的精神，还是应该积极思考一些的。

还有一类全局变量冲突，也发生在递归函数中：

```
int a[MAX]={0},w[MAX]={0},n,use[MAX]={0};
void solve(int pos,int now)
{ int i;
  RETURNTEST;  //终止条件测试处理
  for(i=0;i<n;i++)  //循环枚举当前变量
   if(!use[i])  //检测是否用过，忘记此句，会导致重复枚举或使用
    if(SOME PROPERTIES OF a[])  //如果可行则开始递归
    { a[pos]=i; //记录 a 当前的状态
      use[i]=1; //标记当前元素已用过，忘记会导致重复使用
      solve(pos+1,now+w[i]); //递归处理
      use[i]=0;  //释放空间，忘记会导致只能达到一次，无法回溯
    }
  a[pos]=0; //标记此处为未处理，忘记此句某些问题会出错
}
```

在处理时，一定要注意 pos+1 是给下一个递归函数的，而不要使用 pos++，这样会使得其他 i 的枚举过程中位置变化，导致错误。

释放空间，枚举结束后复位原来状态，尽量不破坏原始数据，是一个良好的习惯。所以对每个 i 枚举完后对 i 复位，全部枚举完后对当前的 pos 复位。这是最容易忘记的地方，很多令人“百思不得其解”的错误正是源于此处。

例如一道“填数字”类试题，要求每格所填数字与周围不同，当然还有其他约束条件。那么如果忘记 a[pos]=0，在这轮的填数中 pos 被赋上了最后一个枚举到的 i 值，例如是 n-1，函数结束，值保留下来。下一次在另一个递归过程中枚举旁边的方格，发现此处填有一个数，于是这个数由于与相邻方格相同被列入了“黑名单”。事实上这个“填入”的数早已与这次的枚举无关了。

养成“复位”的好习惯，是十分重要的。每次完成对一个变量的操作后，如果后面还要使用这个变量，那么在操作结束时复位这个变量到初始状态，是一种好的习惯。有人可能会说，下次使用时再初始化不就得了，我们就想，如此复杂的程序，万一忘记一个，岂不浪费宝贵的调试时间？况且，对于上文中“填数字”的例子，使用时根本不知道应该复位哪个变量，这样就只能采用“自己清理自己”的方法，在这个函数运行结束时，不要给整个程序留

下“垃圾”。

对于输入数据，要尽量保护好，除非特殊需要，尽量不要改动初始数据，以备不时之需。有些选手为了搜索的方便，直接在读入的数据上“大开杀戒”，到最后验证结果时，读入数据已经“面目全非”，程序不得不重新编写。

笔者认为，对于任何定义的变量、数组，都应该赋初值为零，因为赋初值的操作只占用编译时间，不占用运行时间。这样不会因为一次偶然的“越界”导致访问到“随机数”，出现问题，毕竟 0 对于多数问题相当于“没有”。

函数运行的过程酷似做实验。在做实验前检查器材是否损坏，确保器材完好，就是“赋初值”的过程。做实验的过程中不要破坏原有试剂瓶的药品，即应当将药品取出适量到试管中去反应，而不是直接在试剂瓶里做试验，这类似于“保护原有数据”的做法。做完实验，除了反应的生成物作为实验成果，其他中间产物都要自行清理干净，将器材归位，这就是函数结束时的“复位”，除了要输出的，什么也不留下。诚然，有些学生不按实验规程做实验，也能取得实验的成功；但这样的操作，势必增大了风险系数，是我们不推荐的。

另外，在编写递归程序时，还容易发生“形参实参”的问题。传递给函数的单一变量可以看做一个常量，在“子”函数运行过程中可以修改，但不影响“母”函数。而数组、指针的传递却可以起到真正修改变量的作用。这样的内容早在初赛就应该了解，NOIP2008 初赛也出了一道有关的试题。大致内容如下：

```
Function fun(int a, int b)
{ int t;
  t=a;a=b;b=t;
}
Int main()
{ int a=1,b=2;
  printf("%d %d\n",fun(a,b));
}
```

显然，上述程序应该输出 1,2，由于 fun 根本没有改变 a，b 的值。但很多选手考试时却偏偏没有注意到这一点，将输出写反。初赛尚且如此，到了程序复杂的复赛，很多选手更是误以为函数的形参可以保留真正的值，犯了严重的错误，还百思不得其解。为了保险起见，我们不建议使用指针、数组作为函数的形参，而完全可以使用全局变量代替，这样不容易出问题。这方面的例子主要有搜索中的当前状态数组、最终结果数组、最大值记录变量等。

8.1.4 张冠李戴

张冠李戴，就是将此变量误认为彼变量，错误的赋值、运算。引起“张冠李戴”问题的主要原因是，没有分清各个变量、数组的作用而写下一个不适合的数组元素。

例如原来的 pos 变量表示现在枚举到的数组下标位置，现在编程时误认为是把元素放到的位置，显然会导致错误。解决此类问题，一种是采用没有歧义的变量名，一种是想清楚每个变量代表的意义，必要时可以加上注释。

在程序段复制时常出现上述情况，所以应该尽量避免程序段的复制。确需分类讨论的，复制后一定要仔细观察，对每个有变化的地方仔细修正。据统计，笔者 70%以上的“张冠李戴”错误源自于程序段复制时忘记修改或修改“不彻底”，留下了“死角”。

例如这样一段很简单的动态规划的程序：

```
if(a[j]+wa[i][j]<a[i])
 a[i]=a[j]+wa[i][j];
```

针对 b 数组再次处理，复制后也许变成

```
if(b[j]+wa[i][j]<b[i])
 b[i]=b[j]+wa[i][j];
```

显然 w 数组忘记修改了。当程序段复杂时，内部逻辑关系错综复杂，甚至有两个数组互相交织的情况，如果图省事而轻易放过，极易引发“张冠李戴”的错误。

8.1.5 正反对调

观察下述程序段

```
for(i=0;i<n;i++)
 for(j=0;j<i;j++)
  if(a[i]+w[j]<a[j])
   a[j]=a[i]+w[j];
```

显然是 i，j 写反了，达不到动态规划的目的。

如果说这样迷惑性还不大，可以看看下面的：

```
void GetNext(char* T, int *next)
{
        int k=1,j=0;
        next[1]=0;
        while( k<T[0] ){
                if (j ==0 || T[k] == T[j])
                        next[++j] = ++k;
               else    j= next[j];
        }
}
```

这是一段 KMP 算法中求 next 数组的程序，读者能够一眼看出问题所在吗？[1]如果不能在一分钟之内发现错误，那么您一定是在编程中经常为这类问题而苦恼了。

我们强调“静态调试”，就是要在程序没有实际运行的情况下，在脑中先思考整个算法，查看程序的逻辑结构是否出现问题。要做到静态调试，就要

（1）在编写每个函数前，这个函数的功能是什么？要应用哪些外部数据，输出的结果是什么，存放在哪里？除了要求的输出，对其他数据造成了什么影响？

（2）在编写每个循环前，这个循环枚举的对象是什么？枚举的范围是什么？应用的数据应该初始化成什么样子？最后的结果保存在哪里？

（3）在编写每条语句前，这条语句的功能是什么？是对哪些变量进行怎样的操作，又将结果保存在哪里？是否对一些其他变量，尤其是循环控制变量造成了影响？是将谁赋值给谁，即哪一项应该是被更新的？

（4）在调用每个变量前，这个变量所存放的是什么？对于二维数组，行和列到底哪个应该对应前后？是否确实是这个下标对应的元素？

上述的思考，看似麻烦，实则简单，在编程中就是脑子一闪而过的念头，不必要真的写

[1] 上段程序只是将 next[++k]=next[++j]换成了 next[++j]=next[++k]，导致赋值方向错误。

下来逐一比对。但经过上述思考，程序出错的机会就大大降低了。

8.1.6 就差一点

在程序设计中，尤其是有关字符串处理的题目，+1、-1 是必不可少的。但就在这正负号之间，存在着巨大的隐患。很多错误都是不变、+1、-1 三种情况写错导致调试时出现重大问题，故称“就差一点”。

“就差一点”的错误在信息学竞赛中是常见的。例如 NOIP2000《计算器的改良》[1]

```
#include<stdio.h>
#include<string.h>
main()
{ int l,i=0,x=0,n=0,flag=1,nowflag,now;
  char a[200],ch;
  scanf("%s",a);
  l=strlen(a);
  while(i<l)
  { i++;
    if(a[i]>='a'&&a[i]<='z')
    { ch=a[i];break;}
  }
  i=0; //记录当前字符串处理的位置
  while(i<l)//注意不再需要 i++
  { if(a[i]=='=')
    { flag=-1;//若为等号，整体符号反向，相当于移项
      i++;
    }
    if(a[i]=='-')//若为负号，当前正负相反
     nowflag=1;
    else
    { if(a[i]!='+') //若不为+号，证明是数字，需要退回一个字符
       i--;
      nowflag=-1;
    }
      i++;//考察下一位，即数字或字母
      if(a[i]>='a'&&a[i]<='z')
       now=1;
      else
      { now=0;
        while(a[i]>='0'&&a[i]<='9')//整理，产生数，终止于非数字位
        { now=now*10+a[i]-'0';
          i++;
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=344

```
        }
      }
      if(a[i]>='a'&&a[i]<='z') //未知数，终止于非字母位
      { x+=now*flag*nowflag;
        i++;
      }
      else n-=now*flag*nowflag;//记录变化情况
  }
  printf("%c=%.3f",ch,n*1.0/x);
}
```

有些读者可能会说，这样的试题真是“水题”，一编就出来了。但是，您能做到一次 AC 吗？进一步，您能做到不用调试，就直接 AC 吗？这样“表达式处理”的问题，没有什么思考难度，却要求很多编程的细致。在上述程序中，标记当前字符串位置的 i，不时发生++、--的操作，只要写错一处，程序就前功尽弃。

8.1.7 逻辑混乱

逻辑混乱，此处所指的就是

8.1.8 边界溢出

8.1.9 格式错误

8.1.10 忽略特例

据统计，50%以上的细节错误都与“0”这个特殊数字有关。

8.1.11 分类失误

8.1.12 算法错误

8.2 善用输出语句

程序调试，要牢记这样一点：祸患常积于忽微，千里之堤溃于蚁穴。一个复杂的程序，除非开始时算法错误，往往不会通篇错误，只是在一两个细节处出错，导致整个程序调不出来。这样，找到这为数不多的“错误点”，就是解决问题的关键。为了找出问题所在，只使用“黑箱测试”是不够的，我们要深入程序运行的内核，观察程序运行过程中的变化情况，进而针对运行轨迹进行分析，找出问题所在。

相信很多读者喜欢使用编译器中提供的“查看变量”“断点”“单步进入”等工具进行查错，但笔者自初学开始就不推崇这些工具，而是使用一种更为灵活的方法——输出语句。

使用输出语句方法查错，相信读者并不陌生，就是使用输出函数将希望观察的变量变化情况显示在屏幕上，以进行分析。相比编译器中的工具，输出语句可以加在程序中的任何位置，可以选择性的输出；可以对当前变量取值进行一些运算后输出，便于分析；可以针对某些预定的情况执行重复执行、跳出循环等操作；可以在不同的地方设置不同的输出语句，有的放矢的观察程序在各个位置的运行情况。这些都是编译器中简单的工具望尘莫及的。

8.2.1 字符串处理程序的调试

下面以一道字符串处理试题的调试过程为例，说明“输出语句”的应用。

【猪王争霸】[1]

工商部门查获了有 N 个人正在贩卖注水猪肉,现在要你对这 N 个人的注水猪肉的数量从大到小的排序,并且算出这 N 个人的注水猪肉总和(单位为….斤)…我们姑且称这些贩卖者为”猪王”吧..

【输入格式】

输入的第一行是一个 1 到 1000 的整数 N，表示总共有 N 位猪王参加了争霸赛。以下依次给出每位猪王的描述，一位猪王的描述占据两行，第一行为一个仅由小写字母组成的长度不超过 13 的字符串，代表这个猪王的名字，第二行一个高精度的整数(非负数)，代表这个猪王的注水猪肉总斤数。注意，这个整数的首位没有不必要的 0。所有猪王贩卖的注水猪肉数量的总长度不会超过 2000。

【输出格式】

依次输出按照注水猪肉多少从大到小排好序的各位贩卖者的名字,每个名字占据单独的一行。不能有任何多余的字符。若几个名字的发贴数相同,则按照名字的字典顺序先后排列。(名字长度<=13 且均为大写字母),在 N+1 行输出所有猪王贩卖注水猪肉的数量的总和(输出最后 490 位)

【调试历程】[2]

共提交 7 次，其中第 1、3、4 次为 20 分，2、5、6 次为 0 分，第 7 次 AC。

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=168

[2] http://www.rqnoj.cn/Status_Search.asp?UID=2136&PID=168&SID=&Status=98

关于本题，有众多的讨论。[1] 很多都是因为细节上的失误导致前功尽弃。

【代码 1】20 分[2]

编写思路：

```
#include<stdio.h>
#include<string.h>
char name[1001][15],nums[1001][2001];
int num[1001][2001]={0},l[1001];
int comp(int a,int b)
{ int i;
  if(l[a]<l[b])
   return -1;
  if(l[a]>l[b])
   return 1;
  for(i=0;i<l[a];i++)
  { if(num[a][i]<num[b][i])
     return -1;
    if(num[a][i]>num[b][i])
     return 1;
  }
  return 0;
}
main()
{ int n,i,j,t,x[1001]={0},ans[492]={0};
  scanf("%d",&n);
  for(i=0;i<n;i++)
  { scanf("%s%s",name[i],nums[i]);
    l[i]=strlen(nums[i]);
    for(j=0;j<l[i];j++)
     num[i][l[i]-j-1]=nums[i][j]-'0';
    x[i]=i;
  }
  for(i=0;i<n;i++)
   for(j=0;j<i;j++)
    if(comp(x[i],x[j])>0)
    { t=x[i];x[i]=x[j];x[j]=t;
    }
  for(i=0;i<n;i++)
   for(j=0;j<i;j++)
    if(comp(x[i],x[j])==0&&strcmp(name[x[i]],name[x[j]])<0)
    { t=x[i];x[i]=x[j];x[j]=t;
    }
  for(i=0;i<n;i++)
```

[1] http://www.rqnoj.cn/Discuss.asp?PID=168
[2] http://www.rqnoj.cn/Status_Show.asp?SID=22171

```
   printf("%s\n",name[x[i]]);
  for(i=0;i<n;i++)
   for(j=0;j<490;j++)
    ans[j]+=num[i][j];
  for(i=0;i<490;i++)
  { ans[i+1]+=ans[i]/10;
    ans[i]%=10;
  }
  for(i=489;i>=0;i--)
   printf("%d",ans[i]);
}
```

【代码 2】0 分[1]

```
#include<stdio.h>
#include<string.h>
char name[1001][15],nums[1001][2001];
int num[1001][2001]={0},l[1001];
int comp(int a,int b)
{ int i;
  if(l[a]<l[b])
   return -1;
  if(l[a]>l[b])
   return 1;
  for(i=0;i<l[a];i++)
  { if(num[a][i]<num[b][i])
     return -1;
    if(num[a][i]>num[b][i])
     return 1;
  }
  return 0;
}
main()
{ int n,i,j,t,x[1001]={0},ans[492]={0};
  scanf("%d",&n);
  for(i=0;i<n;i++)
  { scanf("%s%s",name[i],nums[i]);
    printf("%s+%s\n",name[i],nums[i]);
    l[i]=strlen(nums[i]);
    j=0;
    while(j<l[i]&&nums[i][j]=='0')
     j++;
    for(t=j;t<l[i];t++)
     nums[i][t-j]=nums[i][t];
    l[i]-=j;
```

[1] http://www.rqnoj.cn/Status_Show.asp?SID=46699

```
    for(j=0;j<l[i];j++)
     num[i][l[i]-j-1]=nums[i][j]-'0';
    x[i]=i;
  }}
```

【代码 3】100 分[1]

```
#include<stdio.h>
#include<string.h>
char name[1001][15],nums[1001][2001];
int num[1001][2001]={0},l[1001];
int comp(int a,int b)
{ int i;
  if(l[a]<l[b])
   return -1;
  if(l[a]>l[b])
   return 1;
  for(i=l[a]-1;i>=0;i--)
  { if(num[a][i]<num[b][i])
     return -1;
    if(num[a][i]>num[b][i])
     return 1;
  }
  return 0;
}
main()
{ int n,i,j,t,x[1001]={0},ans[492]={0};
  scanf("%d",&n);
  for(i=0;i<n;i++)
  { scanf("%s%s",name[i],nums[i]);
    l[i]=strlen(nums[i]);
    for(j=0;j<l[i];j++)
     num[i][l[i]-j-1]=nums[i][j]-'0';
    x[i]=i;
  }
  for(i=0;i<n;i++)
   for(j=0;j<i;j++)
    if(comp(x[i],x[j])>0)
    { t=x[i];x[i]=x[j];x[j]=t;
    }
  for(i=0;i<n;i++)
   for(j=0;j<i;j++)
    if(comp(x[i],x[j])==0&&strcmp(name[x[i]],name[x[j]])<0)
    { t=x[i];x[i]=x[j];x[j]=t;
    }
```

[1] http://www.rqnoj.cn/Status_Show.asp?SID=46707

```
  for(i=0;i<n;i++)
   printf("%s\n",name[x[i]]);
  for(i=0;i<n;i++)
   for(j=0;j<490;j++)
    ans[j]+=num[i][j];
  for(i=0;i<490;i++)
  { ans[i+1]+=ans[i]/10;
    ans[i]%=10;
  }
  for(i=489;i>=0;i--)
   printf("%d",ans[i]);
  return 0;
}
```

上面的试题调试历经了小错误——大错误——无错误的“循环”过程，期间放弃了原来正确的算法，而导致更大的错误；最后回归到原算法上去，在开始的程序基础上进行修改，成功。

在正式比赛时，经常会出现类似的情况：程序改来改去，越来越乱，变成了“四不像”，调试无法进行下去；想要回到最初的版本重新修改，却没有备份，后悔莫及。因此在较大的程序的修改前，一定要对原来的版本进行备份，不要直接在上面大动干戈。

如果要更换算法，就更要谨慎，因为开始的算法很有可能是正确的，如果一时冲动，换成了一个并不成熟的算法，既浪费了时间，又不能得到问题的解决，会陷入更深的迷茫之中。因此一旦确定算法，就不要随意更改；如果确信原来的算法错误，现在的算法又十分成熟，可以将源程序另存为，只保留输入输出，重新编写算法。万不可贪图一时之便，在原来的程序上改来改去，一来破坏了程序的逻辑结构，这样改出来的程序往往逻辑混乱，不易调试；二来在原来的问题基础上添加问题，只会导致程序中不为人知的细节错误越来越多，达到难以调试的地步。这时，从头编写程序虽然有“前功尽弃”的感觉，但确实是明智的选择。

8.2.2 数学算法的调试

下面我们来看本文 4.2.1 节一道数学类试题的调试过程。求 1,2,3...n 的最小公倍数。

我们的算法是：对 1,2,...n 进行质因数分解，求出每个质因数的最高幂次，再将不同的余数取模相乘，得到结果。

最初得到的代码是：[1]

```
#include<stdio.h>
#include<math.h>
#define M 987654321
int n; //输入数据
int tot=0; //记录素数总数
int p[100000]={2}; //是否素数的标记
int q[100000]={0}; //从小到大的素数
int a[100000]={0}; //每个素数的最高幂次
```

[1] 有些错误是本来不该犯的，这里为了方便讲解故意弄错了一些。

```
int ans=1; //取模后的结果
void prime()//计算素数，使用筛法
{ int i,j,t=(int)sqrt(n)+1;
  for(i=2;i<t;i++)
   if(p[i])
   { q[tot++]=i;
     j=i+i;
     while(j<n)
     { p[j]=0;
       j+=i;
     }
   }
  for(i=t+1;i<=n;i++)
   if(p[i])
    q[tot++]=i;
}
int main()
{ int i,j,t,st,tmp;
  scanf("%d",&n);
  prime();
  for(i=2;i<=n;i++)//计算每个数的分解式
  { t=i;
    for(j=0;t>1;j++)
    { tmp=0;
      while(t%p[j]==0)
      { tmp++;
        t/=p[j];
      }
      if(tmp>a[j])//更新最高幂次
       a[j]=tmp;
    }
  }
  for(i=0;i<tot;i++)//将每个素数的幂乘起来，计算结果
   for(j=0;j<a[i];j++)
   { ans*=p[j];
     ans%=M;
   }
  printf("%d\n",ans);
  getch();
}
```

初看起来，似乎没有问题，但运行一下，就发现是“运行错误”，程序非法退出。下面我们需要找到导致程序运行错误的位置，采用“输出间断点”的方法，类似物理实验中查找电路故障的方法，在程序的每个区块前依次加入输出语句，则程序崩溃前输出的最后一条信息之后的程序块，必定是运行错误的位置。当然，这个步骤也可以运用编译器中的“断点”

功能实现。

对于这个程序，采用从前往后逐一排除的策略，首先检查 prime 函数是否有问题，于是在 prime(); 与 for(i=2;i<=n;i++)之间加入输出语句 printf("-----%d\n",tot); 这样既利用-----检查了错误发生的位置，同时可以通过 tot 变量的数值检查 prime 函数是否求出了正确的素数。

输入：10

输出：-----0 （错误退出）

证明 prime 函数存在问题。深入函数内部，首先在程序头加上 printf("%d\n",t); 输出结果：4 证明 t 的计算正确。

下面考察 for 循环内部，加入下述输出语句：

```
q[tot++]=i;
printf("%d\n",i);
j=i+i;
```

输出结果为空，证明根本没有进入循环。检查 p[i]语句，发现 i=2 时就不可能——原来是循环初始条件弄错，p[]的初值应为{0,0,1}，而不是{2}。这里的错误是变量数组的“张冠李戴”错误，将 p 数组标记每个数是否为素数的功能误认为是 q 数组自小到大记录素数。

修改后，发现产生了 2 这个素数，但后面仍未产生，证明是记录错误。观察

```
while(j<n)
{ p[j]=0;
  j+=i;
}
```

一段程序，发现 p 数组的初值为 0，修改后仍然为 0，怎能起到“筛法”的作用？解决方案有两个：一是修改初值，需要开始时将所有值赋为 1。二是修改标记，将 1、0 的作用对调。如果元素个数较多，赋初值为 1 需要较多的时间，应当选用方法二。这里由于规模只有 100000，就没有必要大动干戈了。

```
int i,j,t=(int)sqrt(n)+1;
for(i=2;i<=n;i++)
 p[i]=1;
printf("%d\n",t);
```

修改上述程序段后，再次运行，输出为 2 3 5 7 10

为什么将最大的非素数 10 输出了？一定是<与<=的问题。哪里涉及了这样的比较？原因在于最大值判断 while(j<n)应该改成 while(j<=n)。这样才能保证最后一个数不被误判。

再次运行，发现 2 3 5 7 程序的输出正确，主函数中 tot 的个数统计也正确。为确保正确性，再输出几组数，如 15、20，确认无误后，删除 prime 函数的调试输出语句，此函数调试正确。

此时仍然提示程序运行错误。加入输出语句

```
for(j=0;t>1;j++)
{ tmp=0;
  printf("%d\n",p[j]);
  while(t%p[j]==0)
```

运行只输出了一个 0. 找到变量定义位置，问题显然：p 数组是一个 bool 标记数组，被 0 除当然会导致错误。又把 p 和 q 弄反了。而且主程序其余位置也有类似错误，一并修改。

终于摆脱了运行错误的困扰，但输出仍为 0. 原因在哪里？首先看分解因式是否有问题。　　if(tmp>a[j])

```
        a[j]=tmp;
        printf("%d %d %d\n",i,q[j],tmp);
      }
```

在这段程序中加上了输出语句，分别输出当前枚举的变量、质因子和质因子个数。

运行结果为 2 2 1　3 2 0　3 3 1　4 2 2　5 2 0　5 3 0　5 5 1...尤其是最后一组 10 2 1　10 3 0　10 5 1，这证明分解质因数是正确的。那么为什么输出了 0?

将原来的输出语句改为 printf("%d %d\n",q[j],a[j]); 发现正常。

进入最后的计算环节，发现严重错误，刚才没有改过来：p 与 q 弄反了。修改后，程序输出为 720，仍然不符合题意。进一步观察乘积函数，发现 i 和 j 弄反了，这里是变量冲突类问题。j 仅仅是控制乘积次数的变量，q[i]才是应该乘上的质因子。

再次检测，输出 2520，就是我们想要的结果。但事与愿违，当输入 8 时，出现了程序崩溃。再次重复上面的检测步骤，发现还是 prime 的问题：输入 7 时只有 2 5 7，3 到哪里了？原因是循环时 for(i=2;i<t;i++)但后面枚举时 for(i=t+1;i<=n;i++)，这样恰好空出了 3 这个点。将后面的一句改为 for(i=t;i<=n;i++)，再次检测，发现一切正常，尤其是对于 1，2 这类的特殊情况，更要引起重视。

最后一定不要忘记把调试输出的语句全部删除。如果题目要求的是文件输入输出，在进行修改即可。

在整个程序的调试过程中，我们走了不少弯路。原因之一是，开始编程时过于马虎，犯了大量低级错误。原因之二是，调试时没有对一个区块调试彻底就开始下一个步骤，导致最后出现问题还要回头重来。

8.2.3 枚举程序的调试

对于一些条件众多、枚举复杂的程序，很容易在编写过程中出现失误。

例如解一道数字谜。

```
    a b c
d a f g g
+e g h a
----------
i j g g f
```

开始时，使用了如下的源代码：

```
#include<stdio.h>
int main()
{ int a,b,c,d,e,f,g,h,i,j,use[10]={0},jin1;
  for(c=0;c<10;c++)
  { use[c]=1;
    for(g=0;g<10;g++)
     if(use[g]==0)
     { use[g]=1;
       for(a=0;a<10;a++)
       { use[a]=1;
         jin1=(c+g+a)/10;
         f=(c+g+a)%10;
```

```
        if(use[f]==0)
        { use[f]=1;
          for(b=0;b<10;b++)
           if(use[b]==0)
           { use[b]=1;
             if(b+jin1>0&&b+jin1<=10)
             { h=10-b-jin1;
               if(use[h]==0)
               { use[h]=1;
                 f=10-1-a;
                 if(use[f]==0)
                 { use[f]=1;
                   for(e=0;e<10;e++)
                    if(use[e]==0)
                    { use[e]=1;
                      j=a+e+1-10;
                      if(j>=0&&use[j]==0)
                      { use[j]=1;
                        for(d=0;d<9;d++)
                         if(use[d]==0&&use[d+1]==0)

printf("    %d %d %d\n%d %d %d %d %d\n+ %d %d %d %d\n---------\n%d
 %d %d %d %d\n\n",a,b,c,d,a,f,g,g,e,g,h,a,i,j,g,g,f);
                      }  }}}}}}}}}
  printf("Problem solved.");
  getch();
}
```

运行结果是，无输出（只输出了 Problem Solved）。

对于如此纷繁复杂的程序，首先应该从较浅层的循环出发，进行输出当前变量操作，以查看是否枚举完毕。像这类问题，往往是由于逻辑问题，而导致枚举过程没有进行完就退出了。

首先检测末位的枚举情况。在

```
f=(c+g+a)%10;
printf("%d %d %d %d\n",c,g,a,f);
if(use[f]==0)
```

加入输出，程序输出为

0 1 0 1
0 1 1 2
0 1 2 3
0 1 3 4
0 1 4 5
0 1 5 6
0 1 6 7
0 1 7 8

```
0 1 8 9
0 1 9 0
Problem solved.
```

我们不由得疑问：为什么 c 的取值只有 0 这个很小的数，而没有更大的枚举？原因一定是在中间时枚举中断了。观察有无控制枚举中断的 return，break，continue，发现不存在，于是断定是 use 数组的问题：忘记归零。

修改后，删除输出语句，发现输出中的某个数很大，一定是输出了地址，或者忘记赋值。发现正是 i 的位置，原因是 d+1 事实上并没有赋值为 i，于是将 i 替换为 d+1.

输出很多，发现了两个问题：大量重复；末位和不正确。

明明是末位和求了出来，为什么输出时被改变了？一定是内层循环中有修改 f 之处。果然，

```
f=10-1-a;
if(use[f]==0)
{ use[f]=1;
```

一句便是对 f 的重新处理。遇到这种情况，不能简单的将后面的语句删除，而要思考出现重复的原因：没有考虑到 f 在式中出现了两次，重复枚举。注意到 a 和 f 都是第一次枚举出来的，于是 a+f=9 成为枚举的一个约束条件。

删除上面的语句和有关恢复语句 use[f]=0;，修改逻辑结构。加入

```
f=(c+g+a)%10;
if(a+f==9)
if(use[f]==0)
```

判断，成为约束条件。

再次运行，不再出现末位和不正确的情况，但仍然出现了重复，例如

```
    7 8 9
5 7 2 6 6
+ 3 6 0 7
---------
6 1 6 6 2
```

一组解中 i 和 g 重复。下面检查没有进行重复剪枝的问题。

```
for(a=0;a<10;a++)
if(use[a]==0)
{ use[a]=1;
```

中，原来没有对 a 的重复性进行判断。

修改后，运行结果为（分三栏）

6 0 2 7 6 3 5 5 + 4 5 9 6 --------- 8 1 5 5 3	6 9 2 7 6 3 5 5 + 4 5 0 6 --------- 8 1 5 5 3	6 0 5 7 6 3 2 2 + 4 2 9 6 --------- 8 1 2 2 3	6 9 5 7 6 3 2 2 + 4 2 0 6 --------- 8 1 2 2 3
7 0 6 4 7 2 9 9 + 3 9 8 7 --------- 5 1 9 9 2	7 8 6 4 7 2 9 9 + 3 9 0 7 --------- 5 1 9 9 2	7 0 9 4 7 2 6 6 + 3 6 8 7 --------- 5 1 6 6 2	7 8 9 4 7 2 6 6 + 3 6 0 7 --------- 5 1 6 6 2

通过这个问题的解决过程，我们体验了一个较为复杂的枚举问题的调试过程。也许读者

看起来这些错误都很“幼稚”，但在我们编程的过程中，正是这种“幼稚”的错误一再降低着我们的 AC 率。因此我们一方面要在写程序时更加小心，另一方面要学会一定的调试技巧，尽可能在短时间内调试成功。

8.3 设计测试数据

我们

8.4 考试有风险，答题需谨慎[1]

在竞赛中，风险是难以避免的。但是，没有风险就没有收益，我们必须处理好“风险”与“收益”的关系，力求取得两者的平衡。

就风险而言，主要表现在单题与整体两个方面。就单题而言，会出现如下情况：

想不出来。对于难题，通常有这样的情况。但自己做不出来的试题可能别人也无法解出，不会造成更大的损失。我们应考虑使用“非完美算法”来骗分。

想出来了，写不出来。对于需要高级数据结构和算法，而自己对这些内容不熟悉的试题，往往出现这种情况。可以考虑换一种思路，或者用较差的算法代替。在竞赛应试中，除非时间充裕，不要试图编写自己不熟悉的高级算法，那样很有可能枉费时间。[2]为了减少这种情形的发生，平时对各种算法一定要非常熟悉，不要抱有侥幸心理，在考试前几天默写一遍代码，可以起到较好的效果。

想出来了，写出来了，调不出来。对于数据结构复杂、情况众多的试题，以及多个算法合在一起的试题，极易出现这样那样的小错误，从而无法通过测试数据。这种问题不仅浪费考试时间，还影响考生心理，容易导致整场发挥失常，应该尽量避免。一般来说，超过 200 行的程序在考试时限内难以调试成功[3]，所以在正式编写程序前，应该对程序的编程复杂度有一个大概的估算，考虑这个程序可能花费多少时间，可能得到多少分数，与其他试题相比较，是否值得这样去做。如果认为不值，可以考虑使用效率较低但容易编写的算法，甚至非完美算法，以取得尽可能高的分数。

想出来了，写出来了，调出来了，得分较低[4]。主要是对于分支较多、逻辑复杂的题目，或者数据规模大而自己的算法不能胜任，会出现这种情况。主要还是审题不仔细、考虑不周全的问题，甚至犯文件名、输入输出、数组越界、循环变量错误等**低级错误**[5]。犯了这种错误，是最令人懊悔的。作为一名优秀的 OIER，不能仅仅会大量的高级算法和逻辑分析能力，更要有扎实的编程基本功，做到**“会编则编对”**。

想出来了，写出来了，调出来了，得满分了，没能获奖。例如一场考试有 A、B 两道试题，A 试题难度较大，需要使用线段树，但朴素的模拟可以得到 30 分；B 试题编程复杂度较大，需要使用复杂的字符串处理，忘记一种情况可能就得 10 分甚至 0 分。假设两道试题完全做出所用时间均为 2 小时，而比赛时间为 3 小时。选手甲平时实力较强，会高级算法，选

[1] 参考了刘汝佳《纵论命题与参赛》。

[2] 当然，也有可能恰好“蒙对”了高级算法，例如 WC2009 第一题，我当时只想到了 dijkstra，但认为效率太低，便想到用队列优化，事后才知道这就是 SPFA。但这是十分偶然的，决不能寄希望于考试时创造算法。

[3] 经常写五六百行程序的大牛除外，本文只针对一般水平的选手。

[4] 刘汝佳原文中“得分为零”的说法似乎不太妥当，因为经过调试，小规模的正常数据已经可以通过，一道合格的试题不应该全是特殊情况、极限数据，所以至少能得到部分分数。

[5] 本文中所指的低级错误，是指一个初级水平的选手，看到题目和原程序后能迅速、明确地指出错误的错误，也就是“显而易见”的错误。其他算法设计、逻辑分析、边界处理方面的错误称为“高级错误”。

择了试题 A，最后 1 小时做试题 B 由于考虑情况不周全只得了 10 分。选手乙平时高级算法积累不够，但编程基本功扎实，选择了试题 B，做完后反过来用朴素算法解决 A，得到 30 分。最后结果，甲得到 110 分，乙得到 130 分，如果获奖分数线恰好在两者之间，实力平平的乙获奖，牛气冲天的甲却惨遭失败。考后，甲说，要是不去碰题 A，先解决 B，得分肯定比现在高；或许还大骂出题人，这样的试题“不能体现实力”。这样的事例不是空想虚构，而是现实竞赛中常常出现的，NOIP2008 中表现尤为明显。由于未安排好比赛时间，未能正确估价风险与收益的关系，导致高手败北、菜鸟得胜的情况比比皆是。[1]

算法不等于程序，程序不等于正确，正确不等于得分，得分不等于获奖。如此多的不等号，反映的都是“风险与收益”的关系。

考虑套题的整体性，可能出现如下的抉择：是集中精力解决一道难题，拿到一个满分；还是将各道难题都用朴素算法或非完美算法解决，拿到若干部分分。在通常情况下，如果不能完整的解决所有试题，多个部分分之和往往大于一个满分。[2]尤其是当某一两个测试数据难以处理时，完全可以放弃这 10-20 分，转而解决其他问题。

但是，如果给出的问题是较为复杂的逻辑推理、表达式处理类问题，就不如专注于一道试题。因为这类试题的很小失误，可能导致整个试题几乎不得分，而且编写一个差不多的程序本身就花了较多的时间，这样“部分分”就不如满分了。无论如何，要把握的原则就是，“降低风险，提高收益”。

可惜的是，我们平时的训练往往注重的是“AC 率”，而不是“得分率”。诚然，这种“追求完美”的训练能够使我们思维缜密，提高处理特殊情况的能力，尽可能优化算法，达到平时打好基本功的目的；但这也给应试带来了隐患：容易盲目追求单个程序的完全正确而忽视了考试时间的限制，导致“为什么不早些停止调试第一题，留出时间来做第二题”的遗憾。

事实上，“风险与收益”可以转化为信息学竞赛中经典的一类问题——动态规划问题。设比赛时间为 T，试题数为 n，每道题 i 使用时间 t[i][j]得到的分数为 w[i][j]，求最大得分。当然，正式考试时不可能先做这样一道“动态规划题”，而要在平时的模拟比赛中给自己设定几个参数，DP 一把，看看事实与我们的想象有多少差距。由于试题数少，经典的贪心[3]往往有反例，这也就使我们正确规划竞赛中的用时分配成为一项耐人寻味的课题。

考试有风险，答题需谨慎。“投资有风险。一般情况下，高收益伴随着高风险，低风险也必然导致低收益”，传统的中档题目应以此为准则进行微调。“拥有正确的投资理念和投资策略，有可能实现低风险下的高收益；投资方法不当，有可能出现高风险下的低收益甚至负收益”，在考虑简单题和难题的搭配时应有意考验选手评估、管理风险和收益的能力。

我们相信，只要正确估价竞赛中的风险与收益，在平时多进行应试模拟练习，多思考应试中时间分配的技巧，就能充分发挥自身的实力，让我们的 OI 之路不留遗憾。

9 实战演习

只说不做，等于没有工作。下面通过 2008-2009 年国内外重大赛事真题、NOIP2008 石家庄二中四套模拟赛试题，实地讲解“骗分”策略的应用。

[1] 不是谦虚，本人就是 NOIP2008 中的“菜鸟得胜”的典型。

[2] 当然，ACM/ICPC 另当别论，它们是必须完全正确才能得分；而中学的竞赛则是计算部分分的，这样盲目追求 AC 并不是一个明智的选择。

[3] 就是每次寻找“性价比”最高的一组，依次加入，直到时间耗尽。

9.1 2008 试题回顾与分析[1]

9.1.1 2008 主要赛事题目与分类

序号	赛事	英文题目/目录名	中文题目	类型	问题/主要方法/基本数据结构	时限 s/内存 M
	国内竞赛					
1	NOI2008	party	假面舞会	传统	最短路径问题、并查集	1/128
2		design	设计线路	传统	动态规划、状态转移优化	2/128
3		employee	志愿者招募	传统	网络流、搜索与剪枝	2/128
4		trans	奥运物流	传统	建模、贪心、动态规划	1/128
5		candy	糖果雨	传统	模拟、线段树或树状数组	2/128
6		match	赛程安排	提交答案	枚举，随机化等综合	–
7	CTSC2008 (选拔赛)	triangle	三角形教学楼	提交答案	计算几何、三角形处理	–
8		river	祭祀	传统	二分图的最大匹配、有向无环图的最大独立集	1/
9		volunteer	奥运抽奖	传统	平衡树、堆、BFS、三维线段树、数论与组合数学	3/
10		totem	图腾	传统	数据结构、线段树	3/
11		network	网络管理	传统	平衡树与线段树,公共祖先路径剖分	5/
12		village	唯美村落	提交答案	图绘制，随机、搜索等	–
13	冬令营	trip	游览计划	传统	树形动态规划、基于连通性的状态压缩	2/
14		password	窥孔密码	提交答案	欧拉回路，贪心、调整	–
15		camp	超能纸带机	传统	图灵机、表达式运算的构造方法	1/
16	NOIP2008（提高组）	word	笨小猴	传统	字母统计、求素数	1/50
17		matches	火柴棒等式	传统	枚举法、预先计算查找	1/50
18		message	传纸条	传统	经典动态规划	1/50
19		twostack	双栈排序	传统	二分图判定，贪心建模	1/50
20	NOIP2008（普及组）	isbn	ISBN 号码	传统	字符转数字、基本编程	1/50
21		seat	排座椅	传统	贪心法、排序	1/50
22		ball	传球游戏	传统	简单动态规划	1/50
23		drawing	立体图	传统	循序用字符打印图形	1/50
	国际竞赛					
1	IOI2008	type printer	打印机	传统	排序、前缀树 Trie + DFS 遍历	1s/64M
2		islands	岛屿	传统	求特殊连通无向图的最长路	2/128

[1] 本节摘自王宏《2008 试题回顾与竞赛特点分析》。

3		fish	鱼	传统	离散数学,线段树	3/64
4		linear garden	线状花园	传统	搜索＋剪枝、动态规划	1.5/64
5		teleporters	传送器	传统	排列图、贪心	1/64
6		pyramid base	金字塔地基	传统	二分、平面扫描、线段树	5/256
7	APIO2008	beads	珠链交换器	交互式	链表、有序数组二分查找	2/256
8		roads	免费道路	传统	生成树、连通性、并查集	1/128
9		dna	DNA	传统	字符串匹配、动态规划	1/128
10	CEOI2008	dominance	步步为赢	传统	几何,离散化	2/32
11		knights	悬崖勒马	传统	数学、搜索	1/32
12		information	情报传送	传统	matroid intersection-拟阵交问题、有向图的遍历	1/32
13		**snake**	魔法蛇	交互式	折半搜索、三叉树搜索	1/32
14		**order**	订单选择	传统	网络流	1/32
15		**fence**	多边形篱笆	传统	求点集凸包、最短路径 DP	1/32
16	BOI2008 (Baltic Olympiad in Informatics)	**elections**	选举	传统	背包问题	-/32
17		**game**	游戏	传统	数学证明、动态规划	-/32
18		**gates**	阀门	传统	2-SAT	-/32
19		**gloves**	手套	传统	枚举、数学、贪心	-/32
20		**grid**	网格	传统	搜索+贪心	-/32
21		**mafia**	黑手党	传统	网络流	-/32
22		**magical stones**	魔石	传统	动态规划	-/32

9.1.2 题目方法与分类比例

从上表简单统计可以得出，2008 年主要竞赛题目中，

涉及图论、网络流方面的题目约占 1/3，所占比重最大；应用动态规划思想的题目占到接近 1/4。

涉及线段树等数据结构的题目约占 1/5（国内的前 3 个主要竞赛的比例更高）。

应用或部分应用贪心策略求解的题目约占 1/6。

几何类或涉及计算几何方面知识的题目也占有相当比重。

由于有些题目可能同时使用多种策略或同属多种类型,所以上述题目分类在统计上会有一些重复，但从中大致可以看出 2008 年信息学竞赛题目的主要类型或方法分类。

关于三种类型题目的分布：

信息学奥赛题目的另一种普遍认可的分类方式，是将题目按照传统型（提交源程序,batch tasks）、提交答案型（output-only tasks）和交互式（reactive tasks）三大类题目划分。从上表统计可以得出，2008 年国内竞赛除 NOIP2008 外，其他几次国内重要赛事的题目都覆盖了两种类型（NOI、CTSC、冬令营共有 4 道提交答案型的题目）。特别值得一提的

是，APIO2008 只有 3 道试题，除 2 道传统型题目外，还包括一道交互式题目。CEOI2008 也包括一道交互式题目。相对而言，IOI2008 年竞赛题目的类型比较单一。

对于一个优秀选手而言，在保持传统型题目解题优势的基础上，应有针对性地进行传统型之外其他两种类型题目的特殊训练，以全面适应各种类型题目的要求。2008 年国内赛事也已证明，在提交答案型题目上选手的得分能力（包括所花费的时间），对最后的竞赛结果将产生重要影响和制约作用。

9.1.3 竞赛规则

IOI2008 和 CEOI2008 的竞赛规则与统计数据有几点值得我们关注：沿用在线提交实时测试反馈的方式：根据竞赛规则和题目中规定，IOI2008 仍然沿用了部分题目（6 题中有 3 题）可以进行在线提交，在部分正式测试数据上进行测评的方式，选手还能及时得到反馈的测试结果与说明。这种在线提交实时测试反馈的方式是在 IOI2007 大赛中首次采用的，对于防止选手因看错题目或题意理解偏差而出现的意外失误具有非常积极的意义，但同时对承办地的硬件设施（在线测试的机器台数）也提出了更高的要求。根据组委会提供的数据，绝大部分选手都能充分利用这次比赛提供的这一有利条件。IOI2008 的测试数据也值得关注：首先是 IOI2008 的测试数据规模庞大，且组数很多，多数情形下每组数据的得分不同。王宏教授与 IOI2008 SC 成员与命题人之一讨论过，究其主要原因是部分题目的最优算法复杂度比较低，而非最优算法的复杂性常数又比较小，以及评测机器的速度较高。有人做过比较后认为，IOI2008 的数据数量并不多于近年的 POI 和 BOI（BOI2008 的数据规模是 26M）。另外，这次数据的主要编制方法是先写（确定）一些答案，然后构造数据使答案得到应有的分数。命题过程中曾专门建立了一些文件记录了大部分数据的测试目的，但主办方没有提供。测试数据规模加大和组数增加，在很大程度上加大了得分的难度，对于那些善长“恶搞”等侥幸得分的选手来说十分不利。

对于国内的竞赛，采用在线提交的方式，可以进行尝试，以减少低级错误导致的失分，这样利于选手尽快发现错误并排除错误，就像在线题库[1]上做题时能及时了解错误原因从而有针对性的修改一样。但这也带来了一些弊端，例如选手可能根据反馈的情况得到有关测试数据的信息，从而实现“骗分”；这一点可以通过“两套测试数据”的办法解决，但这又难以保证反馈信息的准确性。更重要的是，在线提交可能导致选手出现“试验算法”的现象，不需要自行设计测试数据，而让这个工作由测评机完成，这就少考察了选手的一项重要能力——设计测试数据的能力，进而降低了选手对思维严密性的要求。

至于增加测试数据的组数，本文中已经提及，是增强题目区分度的重要方式，建议国内竞赛尽快采用。

回顾 2008，展望 2009，随着国内外竞赛的脚步，让我们与 OI 一起启航。

9.2 NOIP 模拟赛（一）[2]

从本节开始的四节，将分析石家庄二中为备战 NOIP2008 而举行的四届模拟赛试题。这些试题全部由学生命制，必然有一些不足之处，解答也未必尽善尽美，敬请指正。

[1] 这里的在线题库指 RQNOJ、VIJOS、USACO 等提供测试结果的网站，而不是 PKU、ZJU 等只通知是否 AC 的网站。

[2] 标程参见 http://boj.pp.ru/olympic/comp/1.htm

命题人：李博杰

题目一览

题目名称	取棋子游戏	数字识别	小明学算术	拦截导弹
代号	game	num	add	mil
输入文件	game.in	num.in	add.in	mil.in
输出文件	game.out	num.out	add.out	mil.out
时限	1 秒	1 秒	1 秒	1 秒

说明：

文件名（程序名和输入输出文件名）必须是小写。

C/C++中函数 main()的返回值类型必须是 int，程序正常结束时的返回值必须是 0。

统一评测使用的机器配置是 CPU 2.93GHz，内存 256M，Windows XP Professional SP2，CENA 测试器。[1]

输入输出必须严格按照标准，不得有多余的空格与换行。

考试时间为 3 小时，满分 400 分。

只提交编译后的.exe 可执行文件。对未编译或因编译错误而导致的结果不正确，我们将不予重新评测。

1、取棋子游戏[2]

(game.pas/c/cpp)

【问题描述】

甲、乙两人轮流从两堆棋子中取棋子，满足下列要求：或者从一堆中取出任意多枚（至少一枚）棋子，或只从两堆中取出同样数目（至少一枚）的棋子，将两堆取完并取到最后一枚棋子者获胜。问：在什么情况下，甲（先取者）有必胜策略？

【输入文件】

输入文件 game.in 只有一行，包含两个用空格隔开的整数表示两堆棋子的个数（每堆棋子至少一个，至多 10000 个）。

【输出文件】

输出文件 game.out 只有一行，包含一个字符。若先取者有必胜策略，输出“Y”（不含引号）；否则，输出“N”（不含引号）。

【样例输入】

样例输入 1：

1 1

样例输入 2：

5 3

【样例输出】

样例输出 1：

[1] 机器配置由学校硬件决定，不同于正式评测机。

[2] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=256

Y

样例输出 2：

N

【分析】见本文“数学分析”一节。

【骗分攻略】合理安排时间，本题看似简单，实际上想到黄金分割策略是不简单的。将本题作为第一题，考察了选手合理分配时间、稳定心理的能力，不能因为第一题的难度而被吓倒。本题考查数学分析、归纳能力。

下面给出一个 90 分的程序。功亏一篑，对于 10000 10000 这个测试点，测试结果为“崩溃”。原因在于数组过小，当较大者等于 10000 时，used[i+plus]>10000，导致数组越界。将数组规模扩大为 20000，即可 AC。由此可见，在编写程序时，要特别注意数组的越界问题，包括向下越界，常在 0、-1 处发生；向上越界，对数组的元素范围估计不足，尤其是涉及队列、高精度的问题。

这个程序的设计思想是值得学习的。在考试的短暂时间内，没有找出数学化的表达式是正常的，但本程序指出了产生这个“互补数列”的方法，也是通过试探、归纳得到的规律。

```
#include <stdio.h>
#include <stdlib.h>
main()
{
    int used[10004] = {0};//数组越界
    int plus = 1;
    int i;
    int success = 1;
    int a;
    int b;
    int tmp;

    freopen("game.in","r",stdin);
    freopen("game.out","w",stdout);
    scanf("%d%d", &a, &b);
    if(a>b)
    {
        tmp = a;
        a = b;
        b = tmp;
    }

    for(i=1; i<=a; i++)
    {
        while(used[i])
        {
            i++;
        }
        used[i] = 1;
```

```
        used[i + plus] = 1;//越界就发生在这里
        if(i==a && i + plus==b)
        {
            printf("N");
            success = 0;
            break;
        }
        plus ++;
    }
    if(success)
    {
        printf("Y");
    }
    return 0;
}
```

【期望】[1]时间 50 分钟，主要是编写搜索程序、寻找规律时间，得分 100 分。

标程源代码：

```
#include<stdio.h>
#include<math.h>
int main()
{ int i,a,b,t;
  double x=(1+sqrt(5))/2,y=(3+sqrt(5))/2;
  FILE *fp,*fp1;
  fp=fopen("game.in","r");
  fp1=fopen("game.out","w");
  fscanf(fp,"%d%d",&a,&b);
  if(a>b)
  { t=a;a=b;b=t;
  }
  for(i=0;i<10000;i++)
   if((int)(x*i)==a&&(int)(y*i)==b)
    break;
  if(i==10000)
   fprintf(fp1,"Y");
  else fprintf(fp1,"N");
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

[1] “期望”是指一个普通水平的 NOIP 提高组选手比赛时的预计解题时间和得分，仅供参考。

2、数字识别[1]

（num.pas/c/cpp）

【问题描述】

手写数字识别(Handwritten Numeral Recognition)是光学字符识别技术(Optical Character Recognition，简称 OCR)的一个分支，它研究的对象是，如何利用电子计算机自动辨认人手写在纸张上的阿拉伯数字。

在整个 OCR 领域中，最为困难的就是脱机手写字符的识别，到目前为止，尽管人们在脱机手写英文、汉字识别的研究中已取得很多可喜成就，但距实用还有一定距离。而在手写数字识别这个方向上，经过多年研究，研究工作者已经开始把它向各种实际应用推广。

现在你的任务就是编写一个简单的数字识别程序。

为了降低难度，我们约定笔迹都是直线，且互相垂直；为了便于处理，所有点都用 01 矩阵表示。1 表示有笔迹，0 表示空白。

各数字的样式遵从普通计算器屏幕的模式。输入中数字大小、笔画长度不一定相同，但笔画方向必定相同。笔画的宽度都是 1。横向各笔画长度分别相等。保证输入的矩阵所代表数字 0-9，符合规范，没有多余的空格和字符。

【输入文件】

输入文件 num.in 第一行为矩阵的长宽 n,m。(0<n<=10,0<m<=10)

以下 n 行每行 m 个无空格的字符 0 或 1，表示该点有无笔迹。

【输出文件】

输出文件 num.out 只有一个数字，即所识别出的数字(是 0,1,2...9 中的一个)。

【样例输入】

样例 1

```
8 5
00000
11111
00001
11111
10000
10000
10000
11111
```

样例 2

```
1 1
1
```

【样例输出】

样例 1

```
2
```

样例 2

```
1
```

【分析】本题的关键在于分类讨论。一定要做到不重不漏。模拟赛时有多人得到 90 分，

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=206

就是因为少考虑了某种情况。

【骗分攻略】找到每个数字决定的特点：只要每条“边”确定了，整个数字也就确定了。于是首先找到四个“角”，针对每种情况寻找“角”周围的“边”，最后确定整个数字。本题考查分类解决问题的能力。

【期望】时间 30 分钟，100 分

```
#include<stdio.h>
int a[4][2];
char x[11][11];
int num()
{ int i;
  if(a[0][0]==a[1][0]&&a[0][1]==a[1][1])
   return 1;
  if(a[2][0]==a[3][0]&&a[2][1]==a[3][1])
  { if(x[a[0][0]][a[0][1]+1]=='1')
     return 7;
    return 4;
  }
  if(x[a[0][0]+1][a[0][1]]=='0')
  { if(x[a[2][0]-1][a[2][1]]=='1')
     return 2;
    return 3;
  }
  if(x[a[1][0]+1][a[1][1]]=='0')
  { if(x[a[2][0]-1][a[2][1]]=='1')
     return 6;
    return 5;
  }
  if(x[a[2][0]-1][a[2][1]]=='0')
   return 9;
  for(i=a[0][0]+1;i<a[2][0];i++)
   if(x[i][a[0][1]+1]=='1')
    return 8;
  return 0;
}
int main()
{ int n,i,j,m;
  FILE *fp,*fp1;
  fp=fopen("num.in","r");
  fp1=fopen("num.out","w");
  fscanf(fp,"%d%d",&n,&m);
  for(i=0;i<n;i++)
  { fscanf(fp,"\n");
    for(j=0;j<m;j++)
     fscanf(fp,"%c",&x[i][j]);
```

```
}
for(i=0;i<n;i++)
{ for(j=0;j<m;j++)
   if(x[i][j]=='1')
   { a[0][0]=i;
     a[0][1]=j;
     break;
   }
  if(j<m)
   break;
}
for(i=0;i<n;i++)
{ for(j=m-1;j>=0;j--)
   if(x[i][j]=='1')
   { a[1][0]=i;
     a[1][1]=j;
     break;
   }
  if(j>=0)
   break;
}
for(i=n-1;i>=0;i--)
{ for(j=0;j<m;j++)
   if(x[i][j]=='1')
   { a[2][0]=i;
     a[2][1]=j;
     break;
   }
  if(j<m)
   break;
}
for(i=n-1;i>=0;i--)
{ for(j=m-1;j>=0;j--)
   if(x[i][j]=='1')
   { a[3][0]=i;
     a[3][1]=j;
     break;
   }
  if(j>=0)
   break;
}
fprintf(fp1,"%d",num());
fclose(fp);
fclose(fp1);
```

```
  return 0;
}
```

3、小明学算术[1]

(add.pas/c/cpp)

【问题描述】

小明最近接到了一项算数的作业。黑板上初始时只有一个数 1。每次取在黑板上的任意两个数（可以相同）相加，得到另一个数，要求这个数比黑板上已有的任意一个数都大，并把所得的符合要求的和数也写在黑板上。这称为一次操作。当黑板上首次出现指定的整数 n (2<=n<=1000)时，停止操作。

小明的加法学得很不好，算一次加法需要很长时间。他希望学编程的你找到一种方案，用最少的操作次数得出指定的数 N。

【输入文件】

输入文件 add.in 只有一行。包含一个整数 n(2<=n<=1000)，表示指定的整数。

【输出文件】

输出文件 add.out 包含两行。第一行一个整数，表示最少操作次数。第二行若干个空格隔开的整数，表示操作结束时黑板上的所有数，按从小到大的顺序输出。

【样例输入 1】

2

【样例输出 1】

1

1 2

【样例输入 2】

200

【样例输出 2】

9

1 2 4 8 16 32 64 128 192 200

【分析】

本题是一道搜索题。如果盲目选择搜索顺序，则会导致超时。笔者采用的方法是，从大至小搜索，尽量先加较大的数，这样首先可以减少步数，尽量快的逼近最优值；另外可以尽快减去不可能的“枝条”，提高效率。另外，搜索时遵循“每次生成的数最大”的原则，如果生成的数比最大数还小，一定在其他时候已经重复枚举了。

本题有两个剪枝：可行性剪枝、最优性剪枝。可行性剪枝：如果当前获得的数比目标整数 n 还大，则停止搜索；最优性剪枝：若当前已经搜索到的数最快到达 n 需要的步数比当前最优值还大，则停止搜索。这里的估计使用了“乘二的次幂”的方法，由于无论如何变化，速度不可能超过自加。

有人认为，为了减少重复运算，还可以采用记忆化的方式，记录达到每个值所需的最小步数。这样的想法是错误的，由于两个数相加时可能有前面用过的公共的数，不是简单的步数相加；记录前面的数也不可行，由于得到同一个 n 有多种方法。程序具体实现时，要注意常数优化。

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=286

【期望】时间 40 分钟，得分 100

【骗分攻略】这道题某些人可能想到动态规划，但尝试后发现并不可行。因此转而进行搜索。下面给出一个 40 分的程序。由于没有进行最优性剪枝，效率较低。

```
#include<stdio.h>
#include<stdlib.h>
#define N 16
int sl=N,sol[N];
int pub[N];
int des;
void search(int k);
int main()
{
  FILE *fp1,*fp2;
  int i;
  fp1=fopen("add.in","r");
  fp2=fopen("add.out","w");
  fscanf(fp1,"%d",&des);
  if(des==2){fprintf(fp2,"1\n1 2"); goto loop;}
  pub[1]=1; pub[2]=2;
  search(3);
  fprintf(fp2,"%d\n%d",sl-1,sol[1]);
  for(i=2;i<=sl;i++)
   fprintf(fp2," %d",sol[i]);
  loop:
  fclose(fp1);
  fclose(fp2);
  return 0;
}
void search(int k)
{
   int i,j,l;
   if(k>sl) return ;
   for(i=k-1;i>=1;i--)
   {
     if(pub[i]*2<pub[k-1]) return ;
     for(j=i;j>=1;j--)
     {
           if(pub[i]+pub[j]<=pub[k-1]) break;
           pub[k]=pub[i]+pub[j];
           if(pub[k]==des)
           {
                char flag=0;
                if(k<sl) flag=1;
                else if(k==sl)
```

测试点编号	得分	用时	空间	注释
add-01 (add...	10.00	0.02s	0.21M	正确
add-02 (add...	10.00	0.02s	0.21M	正确
add-03 (add...	10.00	0.02s	0.21M	正确
add-04 (add...	10.00	0.11s	0.21M	正确
add-05 (add...	0.00	1.05s	0.21M	超时
add-06 (add...	0.00	1.01s	0.21M	超时
add-07 (add...	0.00	1.03s	0.21M	超时
add-08 (add...	0.00	1.03s	0.21M	超时
add-09 (add...	0.00	1.01s	0.21M	超时
add-10 (add...	0.00	1.01s	0.21M	超时

```
                for(l=k;l>=1;l--)
                {
                    if(sol[k]>pub[k]){flag=1;break;}
                    if(sol[k]<pub[k]){flag=0;break;}
                }
            if(flag)
            {
                sl=k;
                for(l=1;l<=k;l++) sol[l]=pub[l];
            }
            return ;
        }
        if(pub[k]<des)
         search(k+1);
    }
  }
}
```

再给出一个 70 分的程序，本程序效率很高，但细节方面出现失误，所以丢了 30 分。也就是说，这个程序找到的不一定是最优解。

```
#include <iostream>
using namespace std;
int main(){
    freopen("add.in","r",stdin);
    freopen("add.out","w",stdout);
    int max=1;
    int n,ans=0;
    int list[1001];
    for (int i=1;i<=1000;++i) list[i]=0;
    list[0]=1;
    cin>>n;
    while (max*2<=n) {
        max*=2;
        list[++ans]=max;
        }
    n-=max;
    int j=ans;
    while (n>0) {
        while (list[j]>n) --j;
        n-=list[j];
        max+=list[j];
        list[++ans]=max;
        }
    cout<<ans<<endl;
```

测试点编号	得分	用时	空间	注释
add-01 (add...	10.00	0.02s	0.23M	正确
add-02 (add...	10.00	0.02s	0.23M	正确
add-03 (add...	10.00	0.02s	0.23M	正确
add-04 (add...	10.00	0.02s	0.23M	正确
add-05 (add...	0.00	0.02s	0.23M	不匹配：13 :: 11
add-06 (add...	0.00	0.02s	0.23M	不匹配：13 :: 12
add-07 (add...	10.00	0.03s	0.23M	正确
add-08 (add...	10.00	0.03s	0.23M	正确
add-09 (add...	10.00	0.02s	0.23M	正确
add-10 (add...	0.00	0.02s	0.23M	不匹配：14 :: 12

```
    cout<<1;
    for (int i=1;i<=ans;++i) printf(" %d",list[i]);
    }
```

下面是优化后的标准程序。需要指出的是，这个程序对数据范围内的各种情况不能全部解决，对于某些特殊数据仍然会超时，需要进一步优化。

```
#include<stdio.h>
int
n,num=0,x[15]={1},min=14,ans[15]={1},p[15]={1,2,4,8,16,32,64,128,
256,512,1024,2048,4096,8192};//事先计算出 2 的方幂
void solve(int now)
{ int i,j;
  if(now>=min)//可行性剪枝，若大于 n 则退出
   return;
  if(x[now-1]==n)//得到一组解，更新最优值
  { if(now<min)
    { min=now;
      for(i=0;i<min;i++)
       ans[i]=x[i];
    }
    return;
  }
  if(x[now-1]*p[min-now]<n)//最优性剪枝，如果最优值的步数内不可达则退出
   return;
  for(i=now-1;i>=0;i--)//枚举当前要加上的两个整数
   for(j=now-1;j>=i;j--)
    if(x[i]+x[j]<=n&&x[i]+x[j]>x[now-1])//每次枚举的数必须大于前面的
    { x[now]=x[i]+x[j];
      solve(now+1);//递归计算
    }
}
int main()
{ int i;
  FILE *fp,*fp1;
  fp=fopen("add.in","r");
  fp1=fopen("add.out","w");
  fscanf(fp,"%d",&n);
  solve(1);
  fprintf(fp1,"%d\n",min-1);
  for(i=0;i<min-1;i++)
   fprintf(fp1,"%d ",ans[i]);
  fprintf(fp1,"%d",n); //输出文件末尾没有换行不是好习惯，应该有\n
  return 0;
}
```

测试点编号	得分	用时	空间	注释
add-01 (add...	10.00	0.02s	0.21M	正确
add-02 (add...	10.00	0.02s	0.21M	正确
add-03 (add...	10.00	0.02s	0.21M	正确
add-04 (add...	10.00	0.02s	0.21M	正确
add-05 (add...	10.00	0.17s	0.21M	正确
add-06 (add...	10.00	0.56s	0.21M	正确
add-07 (add...	10.00	0.02s	0.21M	正确
add-08 (add...	10.00	0.02s	0.21M	正确
add-09 (add...	10.00	0.08s	0.21M	正确
add-10 (add...	10.00	0.09s	0.21M	正确

4、拦截导弹[1]

（mil.pas/c/cpp）

【问题描述】

某国为了防御敌国的导弹袭击，发展出一种导弹拦截系统。但是这种导弹拦截系统有一个缺陷：虽然它的第一发炮弹能够到达任意的高度，但是以后每一发炮弹都必须严格小于前一发的高度。某天，雷达捕捉到敌国的导弹来袭。由于该系统还在试用阶段，所以只有一套系统，因此有可能不能拦截所有的导弹。

输入导弹依次飞来的高度（雷达给出的高度数据是不大于 100000 的正整数），计算这套系统最多能拦截多少导弹。

【输入文件】

输入文件 mil.in 包括两行，第一行一个整数 n(1<=n<=100000)，表示导弹的个数。第二行 n 个用空格隔开的整数，表示导弹依次飞来的高度（不大于 100000 的正整数）。

【输出文件】

输出文件 mil.out 只有一行，包含一个整数，表示这套系统最多能拦截的导弹数。

【样例输入 1】

1

1

【样例输出 1】

1

【样例输入 2】

8

389 207 155 300 299 170 158 65

【样例输出 2】

6

【数据规模】

对于 40%的数据，1<=n<=1000；

对于 100%的数据，1<=n<=100000。

【分析】

本题源自 NOIP 原题《拦截导弹》，题目描述几乎相同，但将“高度不大于前一发”改为“严格小于前一发”，并将数据范围做了扩大。由此我们应该得到启示，竞赛中遇到曾经做过的类似试题，一定不要忙于动手敲程序，而要认真审题，仔细看题目描述、数据规模、输入输出格式是否发生变化，根据试卷确定算法，而不是凭经验、相当然。

由于数据规模变成了 100000，我们需要寻找 O(nlogn)级别的算法。笔者采用了二分搜索算法，每次通过二分搜索找到当前元素应该插入的位置，并更新最长下降子序列长度。最后队列的长度即为输出结果。

【骗分攻略】如果使用动态规划算法，显然无法 AC。但当不会高级算法时，如此的“朴素”算法也不失为一种得分的计策。在下面的程序中，由于使用了指针，使得程序编程困难、难以调试，故笔者不主张使用过多的指针。

```
#include"stdio.h"
#include"stdlib.h"
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=217

```
struct line
{
   int val;
   int num;
   struct line *next;
   struct line *last;
};

int main()
{
   int n,res=1;
   struct line *head,*tail,*p,*q;
   int i,stav,stan;

   freopen("mil.in","r",stdin);
   head=(struct line*)malloc(sizeof(struct line));
   head->last=NULL;head->next=NULL;
   tail=head;
   scanf("%d",&n);
   for(i=1;i<=n;i++)
   {
      scanf("%d",&(tail->val));
      tail->num=1;
      p=(struct line*)malloc(sizeof(struct line));p->next=NULL;
      p->last=tail;
      tail->next=p;
      tail=p;
   }
   fclose(stdin);
   tail=head->next;
   while(tail->next!=NULL)
   {
      p=tail;
      stan=p->last->num;stav=p->last->val;
      while(p->last!=NULL)
      {
         p=p->last;
         if(p->val<stav&&p->num<stan)
         {
            if(p->last!=NULL)
            {
               p->next->last=p->last;
               p->last->next=p->next;
```

```
            }
            else
            {
               p->next->last=p->last;
               head=p->next;
            }
            q=p;p=p->next;
            free(q);
         }
         else
         {
            if(tail->val<p->val&&tail->num<p->num+1)
               tail->num=p->num+1;
         }
      }
      if(tail->num>res)
         res=tail->num;
      tail=tail->next;
   }
   tail=head;
   freopen("mil.out","w",stdout);
   printf("%d",res);
   fclose(stdout);
   return 0;
}
```

【期望】

二分搜索算法：解题时间 40 分钟，得分 100 分（不熟悉的前提下，主要是思考用时）

动态规划算法：解题时间 30 分钟，得分 40-60 分（依常数复杂度而定）

```
#include <stdio.h>
int a[100001],f[100001],n,i,j,k,size;
int bsearch(int f[],int size,int a) {//查找 a 应插入的位置
    int l=0,r=size-1,m;
    while (l<=r) {
        m=(l+r)/2;
        if (a>f[m-1] && a<=f[m]) return m;
        else if(a<f[m]) r=m-1;
        else l=m+1;
     }
}
int main() {
    FILE *fp,*fp1;
```

```
    char name[100],name1[100];
    fp=fopen("mil.in","r");
    fp1=fopen("mil.out","w");
    fscanf(fp,"%d",&n);
        for (i=0;i<n;i++) fscanf(fp,"%d",&a[n-i-1]);
        f[0]=a[0]; size=1;
        for (i=1;i<n;i++) {//插入元素 a[i]
            if (a[i]<=f[0]) j=0;
            else if (a[i]>f[size-1]) j=size;
            else j=bsearch(f,size,a[i]);
            f[j]=a[i];//更新序列元素
            if (j==size) size++;//更新序列长度
        }
        fprintf(fp1,"%d",size);
    fclose(fp);
    fclose(fp1);
    return 0;
}
```

附动态规划程序，复杂度 $O(n^2)$

```
#include<stdio.h>
main()
{ int i,n,j,a[1000]={0},h,f[1000]={0},max=0,num=0,ans=0;
  scanf("%d",&n);
  for(i=0;i<n;i++)
   scanf("%d",&a[i]);
  for(i=0;i<n;i++)//动态规划，计算当前最优值
   for(j=0;j<i;j++)
    if(a[j]>=a[i]&&f[j]+1>f[i])
     f[i]=f[j]+1;
  for(i=0;i<n;i++)
   if(f[i]>max)
    max=f[i];
  printf("%d ",max+1);
  while(num<n)//统计子序列个数，是原题的要求
  { h=100000;
    ans++;
    for(i=0;i<n;i++)
     if(a[i]<h)//找到较小值更新
      { h=a[i];
        num++;
        a[i]=100000;
      }
  }
  printf("%d",ans);
```

```
  return 0;
}
```

总体而言，此次模拟赛作为第一次石家庄二中模拟赛，命题风格较为自然，也没有太多理解题意的难度。

考察了数学分析、模拟、搜索、动态规划算法，但没有涉及高级数据结构。其中第二题考察了编程复杂度、情况分析，第三题考察了时间复杂度优化。

其中搜索题的标程不能解决所有情况，违反了本文提出的命题原则。分数分布较为合理，出现了正态分布。

9.3 NOIP 模拟赛（二）[1]

By 最弱的羽毛书

题目一览

试题名称	电脑选择	飞翔的羽毛	笔记本们的并联和串连	充分运用盗取的精简版 war 敌军情报
代号	choose	fly	line	war
输入文件	choose.in	fly.in	line.in	war.in
输出文件	choose.out	fly.out	line.out	war.out
时限	1 秒	1 秒	1 秒	1 秒

1. 电脑选择

(choose.pas/c/cpp)

【问题描述】

YuMS 打算开一个 oj，他有一个小电脑，100 美元让贫困学生用上电脑计划的那种，虽然防摔易带，但做服务器很不适合。

YuMS 不想买一个专业服务器，他有一个这样的想法：再买几台强劲的笔记本，配合无线路由组一个无线服务器组。这样既能省钱，又能在枕上、厕上无线上网。

于是 YuMS 携巨款跑到了太和电子城。

太和电子城门口有一个采购机，可以方便买主先查到自己要买的东西，再去相应的地方买所需的东西。YuMS 没见过这么先进的东西，不太会高级搜索，只能输入“笔记本”，然后手动翻页，找自己所需。太和电子城太大了，与笔记本有关的东西那么多！保安看他找个没完，后面排了那么长的队，就与 YuMS 商量将搜索结果通过蓝牙发到了他的无敌手机里。接着他通过洲际蓝牙将数据发给了你，让你帮他挑选一个最合适的电脑。

每台电脑都有自己的性能指数，用一个整数来表示。当然还有价格，也是一个整数(万)。YuMS 非常性急，在你翻看搜索结果的时候他就会不时给你打电话，询问你截止到你翻到的地方他可以买到的最好的电脑一台电脑的性能指数。有时候他还要问截止到你翻到的地方可

[1] http://boj.pp.ru/olympic/comp/2.htm
注意，这套模拟赛的命题人是学 Pascal 语言的，所以上述链接的网页中有 Pascal 标程，但效率低于 C。

以买到的笔记本们的最大性能指数，因为 YuMS 懒，所以有时候他就不提这件事了。

【输入文件】

输入文件共 m 行。[1]第一行为 YuMS 带的钱数 T（正整数）万。第 2-m 行为操作序列，see 5 6 表示你又看到了一个价格为 5 万性能为 6 的笔记本，ask 表示询问当前可以买到的最好的电脑的性能指数，most 表示询问当前可以买到的电脑们的最大性能指数。

【输出文件】

输出文件与输入文件有关，当输入文件出现 ask 或 most 时，输出文件需输出对应的性能指数。每个数一行。

【输入样例 1】

```
2
see 1 10
see 2 5
ask
see 1 20
ask
```

【输出样例 1】

```
10
20
```

【输入样例 2】

```
4
see 3 10
ask
most
see 4 11
ask
most
see 1 2
ask
most
```

【输出样例 2】

```
10
10
11
11
11
12
```

【范围说明】

对于 30%的数据 A=0;

对于 30%的数据 m<=1000;

对于 100%的数据，m<=10000;0<T<=100;价格为正整数(万)

【分析】

这道题的输入数据规模未知，所以需要使用 EOF 来判断终止，类似 PKU 的处理。

对于 max 询问，处理较为简单，只需记录当前最大性能即可。

[1] 原题有误，第一行为“是否询问”，事实上是个冗余参数，还与后面的数据矛盾，故去除。

对于 most 询问，需要使用 0/1 背包问题的最经典算法。

【骗分攻略】这是一道基础题，但一定要细心，不要出错。

【期望】解题时间 30 分钟，得分 100 分。

```
#include<stdio.h>
int p[10001]={0},v[10001]={0},f[10001]={0};
int main()
{ int n,i,t,flag,max=0,most=0;
  char ord[10];
  FILE *fp,*fp1;
  fp=fopen("choose.in","r");
  fp1=fopen("choose.out","w");
  fscanf(fp,"%d%d",&flag,&t);
  n=0;
  while(fscanf(fp,"%s",ord)!=EOF)
  { if(ord[0]=='s')
    { fscanf(fp,"%d%d",&p[n],&v[n]);
      if(v[n]>max)
       max=v[n];
      if(flag)
      { for(i=t;i>=p[n];i--)
         if(f[i-p[n]]+v[n]>f[i])
         { f[i]=f[i-p[n]]+v[n];
           if(f[i]>most)
            most=f[i];
         }
      }
      n++;
    }
    else if(ord[0]=='a')
     fprintf(fp1,"%d\n",max);
    else if(ord[0]=='m')
     fprintf(fp1,"%d\n",most);
  }
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

2. 飞翔的羽毛

(fly.pas/c/cpp)

【问题描述】

这天 YuMS 在厕所半天没有出来，原来是他用他新买的最牛的一台笔记本在厕所编了个

小游戏，游戏是图形界面，其中有一大堆羽毛，有黑色有白色有绿色有红色……游戏任务是让你在这堆羽毛中找到每种颜色组成的面积最大的颜色相同的矩形。

【输入文件】

输入文件第1行有一个整数k，表示矩阵中共出现了k种颜色。第2行包含两个整数m、n分别表示矩阵的行数列数，接下来m行，每行n个整数，每个整数代表一种颜色，表示游戏矩阵。

【输出文件】

输出文件共k行，第i行需输出颜色为i的最大矩阵的面积。

【输入样例】

3
3 5
1 2 2 3 2
2 1 2 3 3
1 2 1 3 3

【输出样例】

1
2
4

【范围说明】

对于30%的数据m,n<=100;

对于100%的数据，m,n<=500;k<=10

【分析】如果直接考虑左上角、右下角，再计算这样的矩形是否同色，复杂度高达O(n^6)，显然不可以。我们考虑：任何一个“最大子矩形”一定是从某行开始的。而从某行指定的i到j位置为第一行的矩形，一定尽可能向下延伸，使其行数最多。

根据这种理念，对于每种颜色，首先枚举左上角的位置，然后枚举此行的另外一个点，作为右上角。这里有一个剪枝，如果现在的点不满足右上角条件，更靠右的点一定不可能，直接跳出。然后向下逐行寻找，直到出现异色或最后一行。最后统计最大值。

【骗分攻略】本题就是动态规划中的一类经典问题：最大子矩形问题。某位同学不假思索就提交了如下的代码：

```
#include<stdio.h>
main()
{ int n,i,j,k,l,a[101][101]={0},x[101][101]={0},max=0;
  scanf("%d",&n);
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    scanf("%d",&a[i][j]);
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
   { x[i][j]=x[i][j-1]+x[i-1][j]-x[i-1][j-1]+a[i][j];
     if(x[i][j]>max)
      max=x[i][j];
   }
  for(i=0;i<=n;i++)
   for(j=0;j<=n;j++)
```

```
    for(k=i+1;k<=n;k++)
     for(l=j+1;l<=n;l++)
      if(x[k][l]+x[i][j]-x[i][l]-x[k][j]>max)
       max=x[k][l]+x[i][j]-x[i][l]-x[k][j];
  printf("%d",max);
}
```

但需要注意的是，首先 n 的范围是 500，O(n^4)的算法不可能通过。其次，这次是要求找最大的“同色”矩形，是要求面积最大，而不是数和最大。这两道题截然不同，因此审题时必须细心，避免“低级错误”。

【期望】时间 40 分钟，100 分

```
#include<stdio.h>
int f[501][501]={0};
int main()
{ int n,i,j,k,l,c,num,m,x,y,max=0,depth;
  FILE *fp,*fp1;
  fp=fopen("fly.in","r");
  fp1=fopen("fly.out","w");
  fscanf(fp,"%d%d%d",&num,&m,&n);
  if(num==1)
  { fprintf(fp1,"%d",m*n);
    goto loop;
  }
  for(i=0;i<m;i++)
   for(j=0;j<n;j++)
    fscanf(fp,"%d",&f[i][j]);
  for(c=1;c<=num;c++)
  {  max=0;
  for(i=0;i<m;i++)
   for(j=0;j<n;j++)
   {depth=n;
    for(k=i;k<m;k++)
    { if(f[k][j]!=c)
       break;
      for(l=j;l<depth;l++)
       if(f[k][l]!=c)
        break;
      depth=l;
      if((k-i+1)*(depth-j)>max)
       max=(k-i+1)*(depth-j);
    }
   }
  fprintf(fp1,"%d\n",max);
  }
  loop:fclose(fp);
```

war-01	(war...	10.00	0.02s	1.07M	正确
war-02	(war...	10.00	0.02s	1.07M	正确
war-03	(war...	0.00	0.02s	1.07M	不匹配：9 :: 4
war-04	(war...	10.00	0.02s	1.07M	正确
war-05	(war...	10.00	0.02s	1.07M	正确
war-06	(war...	10.00	0.02s	1.07M	正确
war-07	(war...	0.00	0.03s	1.07M	不匹配：2 :: 1
war-08	(war...	0.00	0.02s	1.07M	不匹配：10 :: 3
war-09	(war...	0.00	0.02s	1.07M	不匹配：8 :: 7
war-10	(war...	0.00	0.02s	1.07M	不匹配：10 :: 9

```
  fclose(fp1);
  return 0;
}
```

3. 笔记本们的并联和串连[1]

(line.pas/c/cpp)

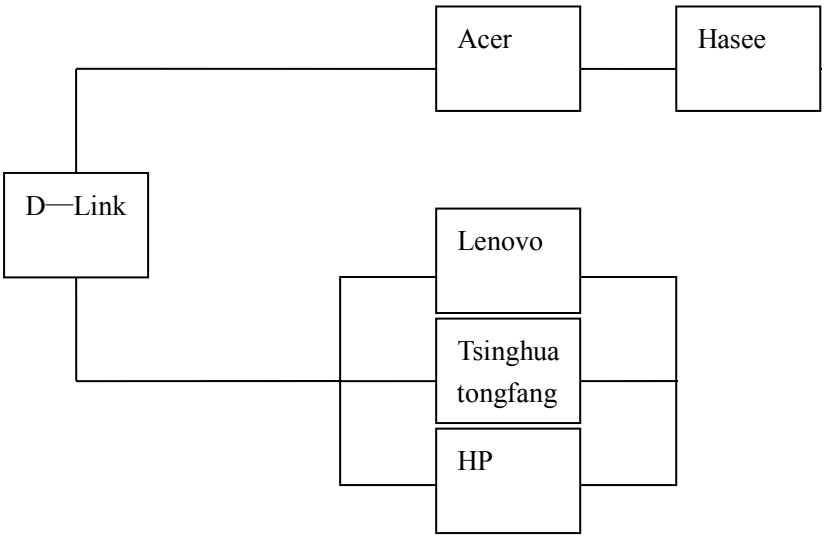
【问题描述】

YuMS 的笔记本群要开始工作了，可是 YuMS 买的路由器巨垃圾，只能连出两条线，一条上面笔记本们串联，另一条上面笔记本们并联。这使他的笔记本只能像电路一样，笔记本与路由器不是星状排布，而是笔记本之间的线状连接。我们设想连接的时候串连的笔记本中每台笔记本们性能都将变为这条支路上性能最低的一台的性能;并联的几台笔记本中每台笔记本的性能变为他们性能的平均数除以 2，并联路要求至少有两台笔记本；并联的电脑们性能不受串联路的影响，也就是说图一中 Acer 机与 Hasee 机相互制约，而与其他笔记本无关；并联的几台笔记本中每台笔记本的性能变为他们性能的平均数除以 2，而与 Acer、Hasee 无关。为了使问题变得简单一些，我们规定类似电路的这个连接中

并联不能与并联串连;（图二）

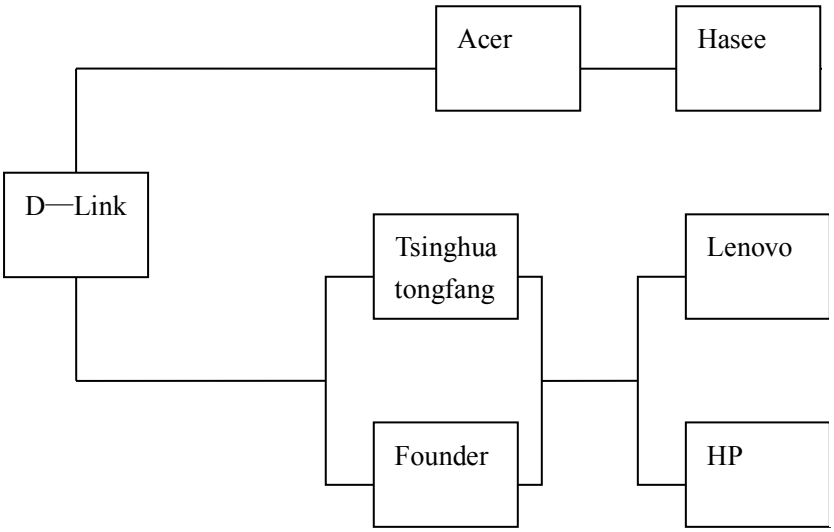
并联中不能出现串连（图三、四）

即如图一所示，而不会出现如图二、三、四所示的连接方式。YuMS 还打算把加上还不如不加的电脑留下玩游戏，即他不一定用掉所有的笔记本。

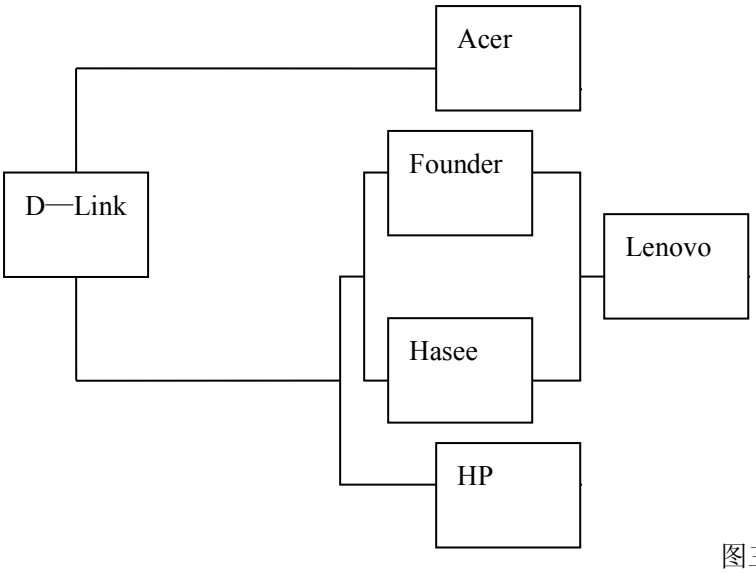


图一

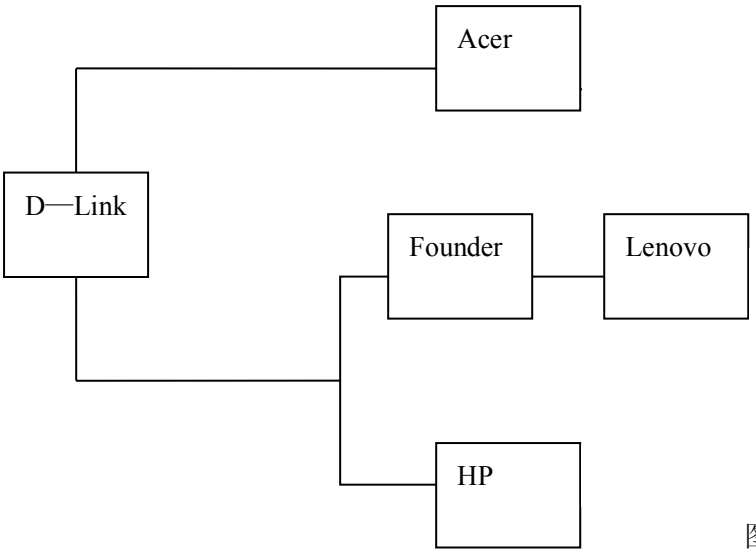
[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=288



图二



图三



图四

YuMS 没有精力再去编这程序了，否则他会累的生病，现在请您帮她作出设计方案，只

要求输出最终笔记本群性能总和的最大值。

【输入文件】

输入文件第 1 行有一个整数 n，表示共买了 n 台笔记本电脑。从第 2 行开始，每行有一个整数代表笔记本的性能。

【输出文件】

输出文件只包含一个实数（保留两位小数），表示你设计的方案中笔记本群的性能总和最大值。

【输入样例】

2
5
9

【输出样例】

10.00　　（5 9 串联）

【范围说明】

对于 30%的数据 n<=5

对于 100%的数据，n<=20

【分析】本题事实上并不是一个难题，关键是题目描述较为复杂，难以理解题意。

事实上，本题就是将 n 个笔记本分成 2 组，一组全部并联，变成平均值的一半；一组全部串联，变成最小值。进一步考虑，并联组中对各个平均值的一半求和，相当于对一半的平均值求和，相当于将价值的一半求和。这样，每个笔记本有两种选择：或者变成价值的一半，或者变成一组中的最小值。

现在，“最小值”的未知性给我们解决问题带来很大麻烦。所以应该首先枚举确定串联组中的最小值。既然涉及“最小”问题，排序是必不可少的，至少这样为解题的有序性提供方便。那么，枚举每个值作为串联支路的最小元素，比最小元素还小的必须属于并联支路，否则与最小性矛盾；而最小元素右边的每个元素，由于其价值与其他元素无关，因此可以贪心选择，如果其一半大于选出的最小元素，则进入并联支路，否则进入串联支路。由于元素的有序性，只需找到第一个并联者，后面的全部并联。

【骗分攻略】

上面的算法时间复杂度为 O(n^2)，对于 n=20 的数据，完全可以胜任。但某些同学一看到 n=20 就想到了枚举、搜索，而不注重数学方法的思考，导致时间效率偏低。更有甚者，看到了“复杂”的连接图形就被吓倒了，不敢认真思考，更不能解决问题。

【期望】时间 40 分钟，100 分

```
#include<stdio.h>
int main()
{ int n,i,j,t,a[101];
  double tot=0,max=0;
  FILE *fp,*fp1;
  fp=fopen("line.in","r");
  fp1=fopen("line.out","w");
  fscanf(fp,"%d",&n);
  for(i=0;i<n;i++)
   fscanf(fp,"%d",&a[i]);
  for(i=0;i<n;i++)
   for(j=0;j<i;j++)
```

```
    if(a[i]<a[j])
    { t=a[i];
      a[i]=a[j];
      a[j]=t;
    }
  max=a[0]*n;
  if(a[1]*(n-1)>max)
   max=a[1]*(n-1);
  tot=a[0]+a[n-1]*1.0/2.0;
  for(j=2;j<n-1;j++)
   if(a[j]*1.0/2.0>a[1])
    tot+=a[j]*1.0/2.0;
   else tot+=a[1];
  if(tot>max)
   max=tot;
  for(i=2;i<=n;i++)
  { tot=0;
    for(j=0;j<i;j++)
     tot+=a[j]*1.0/2.0;
    for(j=i;j<n;j++)
     if(a[j]*1.0/2.0>a[i])
      tot+=a[j]*1.0/2.0;
     else tot+=a[i];
    if(tot>max)
     max=tot;
  }
  fprintf(fp1,"%.2lf",max);
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

4. 充分运用盗取的精简版 war 敌军情报[1]

(war.pas/c/cpp)

【问题描述】

这天 YuMS 又在厕所里玩游戏了，war 超级精简版，其实就不是暴雪做的。这个游戏需要上网玩，具体规则是这样的：两个种族交战，他们站在一起，每人每天可以对军队发出一次指令。UD 种族每一天只能有一个生物活动，其攻击力为无穷大，血量为 A；NE 种族每天所有的生物都可以活动，但是他们只能去打同一个人，他们的攻击力都为 G（攻击力就是每次打的血量），如果所有生物只需其中一部分就可以干掉一个 UD 生物的话，其它生物也不回去打别的 UD 生物，而是闲着（换句话说就是 NE 每天最多杀死 UD 一个生物，而 UD 在序列成

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=258

立的情况下每天一定杀死一个 NE 生物)。虽然看起来 UD 比 NE 强，但 YuMS 还是非常喜欢可爱的 NE，他每次都选择 NE 进行对战。由于游戏比较烂，没有操作界面，作战之前双方需提交战斗序列。约定 UD 队伍有 U0 个生物，分别为 U1-Uu0。NE 队伍有 N0 个生物分别为分别为 N1-NN0

UD 的战斗序列有 n 行，每行有两个整数 i，j 代表命令 Ui 攻击 Nj。若 Ui 或 Nj 已死亡或本来就不存在，则命令无效。

每天 NE 都先发起进攻。

YuMS 这个坏孩子竟然通过向别人电脑发木马的方法搞到了对方的命令序列。然后 YuMS 又犯懒了，他不想自己写命令序列，只想等你写完后把对方杀个大败。你能帮他吗?

显然能，但命令序列不是唯一的，这导致 YuMS 检验你是否在骗他很麻烦。

其实，YuMS 自己可以想出来命令序列，只是想考考你他 n 天过去他可以最少死几个兵（不一定要杀掉别人最多的兵)。

【输入文件】

输入文件第 1 行有 4 个整数 A、G、U0、N0。第 2 行有一个整数 n

下面 n 行是 YuMS 偷到的 UD 的命令序列。每行有两个整数 i，j 代表命令 Ui 攻击 Nj。

【输出文件】

输出文件只包含一个整数，表示 YuMS 最少死的兵数。

【输入样例 1】

```
50 20 10 5
5
1 1
2 1
3 3
4 4
1 5
```

【输出样例 1】

```
0
```

【输入样例 2】

```
101 20 10 5
1 1
2 1
3 3
4 4
1 5
```

【输出样例 2】

```
2
```

【范围说明】

对于 100%的数据 U0,N0 <=10;

A,G<=10000;

n<=100000

【分析】

本题是此次模拟赛中最难的一道试题。这道题目目前没有完全正确的算法，下面的源代码也是一种贪心算法。

方法是：如果能打当前的 UD，那么尽快发动进攻，进攻时针对现存血量最少的，以保

证尽可能快的消灭。换句话说，就是各个击破敌人。

与其等着被消灭，不如早些进攻，于是越先被消灭的生物越先进攻，这样才不白消灭。事实证明，这样的贪心是有反例的，但数据比较弱，就可以 AC 了。

具体的算法请参考源代码。

【骗分攻略】

本题由于较为困难，所以“骗分”手段大显身手。

首先，最简单的情况就是，由于 NE 先发起攻击，如果每次都能消灭一个 UD，那么自己的损失为 0. 这样的情况是 g*n0>=a. 前 2 个点都是这样的情况。

其次，如果恰好是样例数据，直接输出，第 6 个点是这样的。

最后，由于不会做，就想到如果 UD 的攻击力很强，那么 n 个回合后 NE 会全部死光，所以直接输出 NE 的个数。第 3、4、5 个点都是这样的。

根本不用复杂的分析，就能轻松得到 50 分，可见“骗分”的强大功力。

```
#include<stdio.h>
int f[100001],t[100001];
int main()
{ int n,a,g,u0,n0,i,now;
  FILE *fp,*fp1;
  fp=fopen("war.in","r");
  fp1=fopen("war.out","w");
  fscanf(fp,"%d%d%d%d%d",&a,&g,&u0,&n0,&n);
  if(g*n0>=a)
   fprintf(fp1,"0");
  else if(a==101)
   fprintf(fp1,"2");
  else fprintf(fp1,"%d",n0);
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

【源代码】[1]

```
type
  alive=array[1..10]of boolean;
var a,g,u0,n0,n:longint;
    nalive:alive;
    ualive:alive;
    list:array[1..100000,1..2]of integer;
    i:longint;
    nlast,ulast:integer;
    min:integer;
    shown:alive;
procedure
```

[1] 本代码为 Pascal 语言，是命题人羽毛书所写。

```
search(p:longint;nalive,ualive:alive;nlast,ulast:integer);
  var i:longint;
      nalivep,ualivep:alive;
      nlastp,ulastp:integer;
      tmp:longint;
              function       kill(i:longint;var       p:longint;var
nalive,ualive:alive;var nlast,ulast:integer):alive;
    var now:longint;
        fight:longint;
    begin
      now:= a;
      fight:= nlast*g;
      while (now>0)and(p<=n)and(nlast>0) do begin
        dec(now,fight);
        if now<=0 then ualive[i]:= false;
        if (ualive[list[p,1]] and nalive[list[p,2]]) then begin
          dec(nlast);
          nalive[list[p,2]]:=false;
          dec(fight,g);
        end;
        inc(p);
      end;
      kill:=nalive;
    end;
  begin
    if nlast<n0-min then exit;
    if ((nlast=0)or (ulast=0)or(p>n)) then begin
      if min>(n0-nlast) then min:= n0-nlast;
      exit;
    end;
    nalivep:= nalive;ualivep:= ualive;
    nlastp:= nlast;ulastp:= ulast;
    for i:= 1 to u0 do begin
      if ((shown[i])and(ualive[i])) then begin
        tmp:= p;
        kill(i,tmp,nalive,ualive,nlast,ulast);
        ulast:= ulast-1;
        search(tmp,nalive,ualive,nlast,ulast);
        ualive:=ualivep;
        nalive:= nalivep;
        ulast:= ulastp;
        nlast:= nlastp;
      end;
    end;
```

```
  end;


begin
  assign(input,'war.in');reset(input);
  readln(a,g,u0,n0);
  readln(n);
  fillchar(shown,sizeof(shown),0);
  ulast:= 0;
  for i:= 1 to n do begin
    readln(list[i,1],list[i,2]);
    shown[list[i,1]]:= true;
  end;
  for i:= 1 to u0 do begin
    if shown[i] then inc(ulast);
  end;
  min:=maxint;
  nlast:= n0;
  fillchar(nalive,sizeof(alive),1);
  fillchar(ualive,sizeof(ualive),1);
  search(1,nalive,ualive,nlast,ulast);
  close(input);
  assign(output,'war.out');rewrite(output);
  writeln(min);
  close(output);
end.
```

总体来说，本次模拟赛显得成熟了一些，虽然命题的正确性不如第一次（第一题考试时出错），但题目背景的设置更加生动有趣。

考查知识点集中分布在动态规划（背包问题、最大子矩阵）、贪心两大问题上，很多重要知识点没有考到，是最大的缺陷。

后两道题的命制较有新意，没有现成的算法，体现了选手现场分析、解决问题的能力。

9.4 NOIP 模拟赛（三）

Alyosha 佣兵团圣战记模拟赛

[背景]

在 Alyosha 生活的大陆上有两个种族——龙族和凤族。两个种族虽然偶尔有一点小摩擦，但是总体上还是相处得很和睦。但是有一天，战争的阴云笼罩了整片大陆——凤族对龙族不宣而战。没有准备的龙族被凤族打得连连后退，眼看着就要灭亡了。

就在这时，伟大的 Alyosha 和他的伙伴们站了出来，组成了 Alyosha 佣兵团，开始了为整片大陆上种族的共同发展的斗争。

这是一场惨烈的战争。后人回忆时总是敬畏地叫它“圣战”，并用崇敬的语气来评价扭转局势的 Alyosha 佣兵团。下面我们就一起来回顾一下 Alyosha 佣兵团的圣战历程。

[注意事项]
本次比赛时间为三个小时，题量为四道，满分 400 分。
本次比赛采用文件输入输出的方式。各题目名称和输入输出文件名在各题目中给出。

下面我们就开始和伟大的 Alyosha 一起出发吧。

1. 运输[1]

(trans.c/pas/cpp/in/out)

[背景]
由于龙族在战争初期处于守势，所以 Alyosha 决定先修建一些防御工事来迟滞凤族的进攻。而一个很显然的事实是：修建坚固的防御工事需要大量的材料，包括石头、木头和一种特殊材料 Plastica。而 Alyosha 这里正好有很多奇妙的魔法材料，这些魔法材料可以在上述三种材料中任意转化。
不幸的是，Alyosha 放材料的地方和军营相距很远。为了解决运输问题，佣兵团中的机械师 Sire 设计了一种人形运输工具。Sire 会在别的地方对这些运输工具进行指挥。如果我们把放材料的地方当作坐标原点建立坐标系的话，那么指挥的方式如下：
（1）运输工具前进的基本方向为北、南、西、东，分别用 W,S,A,D 表示。同时它还可以向复合方向前进，所谓复合方向就是指东北、西北、东南、西南四个方向，他们的表示由基本方向复合而成，例如东北方向就可以表示为 WD 或 DW，二者是等价的。运输工具只能向这八个方向前进。
（2）运输工具可以在一个时间段内对材料持续施加 10000N 的力。
（3）Sire 会在某个时间点对运输工具发出新的指令。接到新指令后，由于物理定律的存在，运输工具会保持原有的运动状态。并且在两个指令之间，材料的受力和材料种类不发生变化。
（4）由于某些不可预知的原因，Sire 在发出的指令会附加一些让魔法材料转化的效果。
由魔法材料转化来的石头、木头和 Plastica 的质量分别为 50t，5t 和 1t。当然，材料的变化在运输工具的前进之前就已经完成了。
（5）材料在最开始时是静止的。
现在 Sire 想知道在某一个时刻 M 运输工具的坐标。
[输入]
第 1 行：一个正整数 N，表示 Sire 共发出了 N 条指令。
第 2..N+1 行：先是一个整数 W，表示 Sire 在时刻 W 发出了一个指令。接下来给出指令中让运输工具前进的方向和指令中让材料变换成的种类（石头，木头和 Plastica 分别用 S,W,P 表示）。方向和种类用 ’.’ 分隔。
第 N+2 行：一个正整数 M，表示 Sire 想知道运输工具位置的时间。
[输出]
两行，第一行为 时刻 M 时材料位置的横坐标，第二行为其纵坐标，保留 1 位小数，四舍五入。对于实数运算，请采用 double 数据类型。

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=357

[样例输入]
3
1 W.S
3 W.P
4 D.P
4

[样例输出]
5.0
16.2

[数据规模]
对于 100% 的数据， 1<=N<200 ， M<=3000 ，所有的 W<=3000

【分析】

这是一道模拟题，并具有一些物理的味道，需要用到匀变速运动的知识。

（1）进行字符串处理，把运输工具前进的方向、材料的种类分开。

（2）依次处理运输工具的每次移动，根据上次的末速度和本次的加速度（可用受力/质量算得）计算出下一变换时刻的速度（v1=v0+at），并计算位移。

（3）对每次移动，重复（2）。

（4）如果下一时刻大于终止时刻，则视为最后一次移动。将终止时刻视为下一次的开始时刻，进行（2）的操作，得到终点坐标。

（5）处理时注意运动的分解，应该将所有受力、速度、加速度、位移分解在 x、y 两个互相垂直的方向上，便于计算。8 个允许运动方向也正是方便分解而设置的。

这道题有一些违反物理定律的地方：材料变化种类后，质量发生变化，而速度不变，推出动能瞬间改变，即力是无穷大的，这是不科学的。可以将题目改为：材料变化种类后，动能不变，这样需要对上次的速度进行变换。

【骗分攻略】可能出现的问题主要是字符串处理、最后一次移动，要小心细致。

【期望】时间 30 分钟，100 分

【源代码】[1]

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
struct key
{
    int dx;
    int dy;
    int time;
    int mass;
};
int cmp(const void* x, const void* y)
{
    return ((struct key*)x) -> time - ((struct key*)y) -> time;
}
```

[1] 这是标准程序。笔者的代码有些长，3KB 14ms，这里只有 2KB 10ms，故选用了标程。

```
int mass(char x)
{
    switch(x)
    {
        case 'P': return 1;
        case 'W': return 5;
        case 'S': return 50;
    }
}
int main()
{
    freopen("trans.in", "r", stdin);
    freopen("trans.out", "w", stdout);
    struct key dat[3024];
    double x,y,vx = 0,vy = 0,vtx,vty;
    int run_time,time,n,i;
    char direct;
    char sub;
    scanf("%d", &n);
    for(i=0; i<n; i++)
    {
        scanf("%d ", &(dat[i].time));
        scanf("%c", &direct);
        while(direct!='.')
        {
            switch(direct)
            {
                case 'W': dat[i].dy += 10;
                          break;
                case 'S': dat[i].dy -= 10;
                          break;
                case 'A': dat[i].dx -= 10;
                          break;
                case 'D': dat[i].dx += 10;
                          break;
            }
            scanf("%c", &direct);
        }
        scanf("%c", &sub);
        dat[i].mass = mass(sub);
    }
    scanf("%d", &time);
    dat[n].time = time + 1;
    dat[n].dx = 0;
```

```
    dat[n].dy = 0;
    dat[n].mass = 0;
    n++;
    qsort((void*)dat, n, sizeof(dat[0]), cmp);

// for(i=0; i<n; i++)  printf("%d  %d  %d  %d\n",  dat[i].time,
dat[i].dx, dat[i].dy, dat[i].mass);

    for(i=0; i<n - 1; i++)
    {
        run_time = dat[i + 1].time - dat[i].time;
        vtx = vx + 1.0 * dat[i].dx / dat[i].mass * run_time;
        vty = vy + 1.0 * dat[i].dy / dat[i].mass * run_time;
        x += (vtx + vx) / 2 * run_time;
        y += (vty + vy) / 2 * run_time;
        vx = vtx;
        vy = vty;
    }

    printf("%0.1lf\n%0.1lf\n", x, y);
    return 0;
}
```

2. 流星雨[1]

(meteor.c/pas/cpp/in/out)

[背景]

凤族的进攻之所以如此猛烈，是因为他们拥有一位厉害的魔法师阿努比斯。为了打乱凤族的进攻，Alyosha 决定派他的好友，刺客 llxy7 前去刺杀阿努比斯。很不幸的是，llxy7 在靠近阿努比斯的一瞬间被阿努比斯强大的精神力发现了。阿努比斯用瞬间移动躲开了 llxy7 的刺杀，并在远处施放了他的绝招——流星雨。

我们以 llxy7 最开始所在的位置为原点建立坐标系。llxy7 在每一个单位时间内可以向相邻的四个整点（横纵坐标都是整数的点）移动一格，但不能移出第一象限和 x,y 轴正半轴。同样地，流星也只会落在 llxy7 的活动范围之内。每一枚流星会引起它落下的点以及周围四个整点的燃烧，且燃烧不会停止。显然，llxy7 不能处于这些着火的点。

阿努比斯自信地认为没有人可以在他的流星雨下生还，而 llxy7 不这么看，他认为自己还是有机会生还的。实在不行，他还有一件宝物——海洋之心，它可以让使用者重生。但是除非被迫，llxy7 不想使用海洋之心，毕竟这件宝物太珍贵了。逃出来之后，llxy7 会趁着阿努比斯魔法力枯竭而且警惕放松的时候再次进行刺杀，这时的刺杀是一定可以成功的。

而一个最关键的问题是：llxy7 最少需要多少时间可以到达一个不能燃烧的点?

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=369

[输入]

第 1 行：一个正整数 M(1<=M<=50000)，表示阿努比斯的流星雨魔法里包含的流星数目。

第 2..M+1 行：包含三个正整数 X，Y，T（0<=X,Y<=300，0<=T<=1000），分别表示每一颗流星落到地面上的坐标以及这颗流星落到地面的时间。

[输出]

一行，包含一个整数 T，表示 llxy7 逃脱需要的最短时间。如果他不能逃脱，输出-1。

[样例输入]

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

[样例输出]

```
5
```

【分析】

本题是一道宽度优先搜索试题。这是由于到达点的先后与时间有关，而着火的时间又决定了能否到达。其中有一个贪心的思想：每个点都尽可能早的到达。如果早到，可以尽快走到其他点，总是可以到的，不会有害处；如果晚了，就可能来不及逃脱，所以应该选择尽可能早的路线。

使用一个队列维护可达的点集，每次取出时间最早的一个，进行扩展，扩展时注意着火的时间限制，并进行记忆化，避免重复处理。由于每次扩展的都是最早的节点，所以每个点一定是在所有可能路线中最快的一个，这保证了最优性。

如果最后队列为空，则不可能逃脱；若到达一个安全点，则成功输出结果。读入时应该预处理出每个点最早着火的时间（或安全）。

在处理队列的时候，要注意保证队列中所有点时间上的单调性。注意到达时间恰好等于着火时间的情况。

【骗分攻略】笔者的程序得了 80 分，原因是在边界位置没有考虑，导致走出边界，出现错误。这提示我们必须注意处理边界问题。本题编程复杂度较高，要注意细心。

【期望】时间 50 分钟，100 分（如果有疏忽可能得 60-80 分）

【源代码】AC，310ms。本代码函数过多，可以进行优化，就是将非递归函数整理到主程序中。事实上，由于程序是线性运行的，除了快速排序，都可以整合在 main()主程序中。

```
#include<stdio.h>
#include<stdlib.h>
#define N 50010
#define L 310
struct a
{
   int x,y,t;
}a[N];
int r=L,c=L;
int x[5]={-1,0,1,0,0},y[5]={0,1,0,-1,0};
```

```
//两个方向向量，将运动分解，降低编程复杂度
int g[L][L];
int b[L][L]={0};
int steps[L][L]={0};
int crash[L][L]={0};
int n;
int ans=2147483647;

void Qsort(int i,int j)
{
    int i1=i,j1=j;
    struct a tmp=a[i];
    while(i<j)
    {
        while(i<j && a[j].t>=tmp.t) j--;
        if(i<j)
        {
            a[i]=a[j];
            i++;
        }
        while(i<j && a[i].t<=tmp.t) i++;
        if(i<j)
        {
            a[j]=a[i];
            j--;
        }
    }
    a[i]=tmp;
    if(i1<i-1)  Qsort(i1,i-1);
    if(j+1<j1)  Qsort(j+1,j1);
}

void init()
{
    int i;
    scanf("%d",&n);
    for(i=0;i<n;i++)
    {
        scanf("%d%d%d",&a[i].x,&a[i].y,&a[i].t);
    }
    Qsort(0,n-1);
}

int in_range(int row,int col)
```

```
{
    return row>-1 && row<L && col>-1 && col<L;
}

void strike(int t)
{
    int i;
    for(i=0;i<5;i++)
    {
        if(in_range(a[t].x+x[i],a[t].y+y[i]))
        {
            crash[a[t].x+x[i]][a[t].y+y[i]]=1;
        }
    }
}

int judge(int row,int col)
{
    return in_range(row,col) && !b[row][col] && !crash[row][col];
}

void solve()
{
    int p=0;
    int i;
    int q[2*N]={0};//时间单调队列
    int head=0,tail=1;
    int row,col;
    int pre=-1;
    b[0][0]=1;
    while(head<tail)
    {
        row=q[head]/L;
        col=q[head]%L;
        head++;
        if(steps[row][col]>pre)
        {
            pre=steps[row][col];
            while(p<n && a[p].t==pre)
            {
                strike(p);
                p++;
            }
        }
```

```
        if(crash[row][col])
        {
            continue;
        }
        for(i=0;i<4;i++)//按四个方向进行扩展
        {
            if(judge(row+x[i],col+y[i]))
            {
                b[row+x[i]][col+y[i]]=1;
                steps[row+x[i]][col+y[i]]=steps[row][col]+1;
                q[tail]=(row+x[i])*L+col+y[i];
                tail++;
            }
        }
    }
    while(p<n)
    {
        strike(p);
        p++;
    }
}

void print()
{
    int mark=0;
    int i,j;
    for(i=0;i<L;i++)
    {
        for(j=0;j<L;j++)
        {
            if(b[i][j] && !crash[i][j])
            {
                mark=1;
                if(steps[i][j]<ans)
                {
                    ans=steps[i][j];
                }
            }
        }
    }
    if(!mark)
    {
        ans=-1;
    }
```

```
    printf("%d\n",ans);
}

int main()
{
    freopen("meteor.in","r",stdin);
    freopen("meteor.out","w",stdout);
    init();
    solve();
    print();
    return 0;
}
```

3. 列队[1]

(line.c/pas/cpp/in/out)

[背景]

在 llxy7 刺杀了阿努比斯后，凤族的进攻被迟滞了。龙族的军队在 Alyosha 的带领下进行了反攻，并渐渐收回了失地。在若干年的僵持之后，大决战终于到来了。

高瞻远瞩的 Alyosha 早早地预见了这场决战的到来，为此，他早在几年以前就授意军师 JosephWei 开始进行准备。准备的一项内容就是要训练一些绝对的精锐部队。

JosephWei 要训练的兵种有三种，我们将他们编号为 1~3。在第一次列队时，他们的队形很混乱，也就是说，1~3 这三种士兵在队列中是任意排列的。由于这些新兵根本不懂得如何列队，JosephWei 无奈之下只能更改他们的兵种，使这些士兵的编号成为一个不升或不降的序列。比如编号序列 1 3 2 1 2 3 可以被变成 3 2 2 1 1 1 或 1 1 1 2 2 2 或 1 1 1 2 3 3。

现在 JosephWei 想知道他如何改变这些士兵的兵种才能让调整的次数最少。

[输入]

第 1 行：一个正整数 N（1<=N<=30000），表示 JosephWei 要训练的士兵数。

第 2..N+1 行：每行一个 1~3 之间的正整数，表示这些士兵第一次列队时的编号。

[输出]

一行。一个正整数 M，表示 JosephWei 需要调整的最小次数。

[样例输入]

```
5
1
3
2
1
1
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=353

[样例输出]

1

【分析】

【骗分攻略】

笔者不会使用近似为 O(n)的动态规划算法，于是想到了 O(n^2)的朴素动态规划算法。

首先，针对特殊情况处理。如果本身就是有序排列的，就没有必要调整。

注意到只有 3 种兵，而前两种兵的个数一旦确定，最后的也就确定了。在确定兵种的升降顺序后，枚举前两种兵的个数，计算三种兵分别需要改变的次数，就可以求得最小值了。

在计算改变次数时，逐一枚举显然太慢了，我们联想到最大子段和问题中“预处理前 n 个数之和”的办法，事先计算出前 n 个兵中三种兵分别占的个数，这可以边读入边累加。然后从 i 到 j 中 A 种兵的个数，等于 A[j]-A[i-1]. 而需要改变的个数，就等于不同种兵的个数，就等于总个数减去同种兵的个数。这样每次计算的复杂度降为 O(1).

下面是笔者 80 分的程序，有效用时 100ms，其余两个点超时。

```
#include<stdio.h>
int main()
{                                                                    int
n,i,j,a[30005]={0},x[30005][4]={0},min=100000000,cheat=0,cheat1=0
;
  FILE *fp,*fp1;
  fp=fopen("line.in","r");
  fp1=fopen("line.out","w");
  fscanf(fp,"%d",&n);
  for(i=1;i<=n;i++)
  { fscanf(fp,"%d",&a[i]);
    if(a[i]<a[i-1])
     cheat=1;
    if(i!=1&&a[i]>a[i-1])
     cheat1=1;
    for(j=1;j<4;j++)
     x[i][j]=x[i-1][j];
    x[i][a[i]]++;//统计到目前为止每种兵的个数
  }
  if(cheat==0||cheat1==0)//先对特殊情况处理
  { fprintf(fp1,"%d",0);
    fclose(fp);
    fclose(fp1);
    return 0;
  }
  for(i=0;i<=n;i++)//枚举前两种兵的终止位置
   for(j=i;j<=n;j++)
   { if(n+x[i][2]+x[j][3]-x[n][3]-x[i][1]-x[j][2]<min)//升序
      min=n+x[i][2]+x[j][3]-x[n][3]-x[i][1]-x[j][2];
```

```
    if(n+x[i][2]+x[j][1]-x[n][1]-x[i][3]-x[j][2]<min)//降序
     min=n+x[i][2]+x[j][1]-x[n][1]-x[i][3]-x[j][2];
   }
  fprintf(fp1,"%d",min);
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

【期望】O(n^2):时间 30 分钟，80 分

O(n)：时间 50 分钟，100 分

【标程】AC，140ms。

```
#include<stdio.h>
#include<stdlib.h>
#define N 30010
int pos[N];
int dp[N][4]={0};
int n;
int ans;

int min(int a,int b)
{
    return a<b?a:b;
}

void init()
{
    int i;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        scanf("%d",&pos[i]);
        dp[i][1]=dp[i][2]=dp[i][3]=2147483647;
    }
}

void solve()
{
    int i,j,k;
    int min1,min2;
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=3;j++)
        {
```

```
            for(k=1;k<=j;k++)
            {
                if(dp[i-1][k]+(pos[i]!=j)<dp[i][j])
                {
                    dp[i][j]=dp[i-1][k]+(pos[i]!=j);
                }
            }
        }
    }
    min1=min(dp[n][1],min(dp[n][2],dp[n][3]));

    for(i=n;i>=1;i--)
    {
        for(j=1;j<=3;j++)
        {
            dp[i][j]=2147483647;
        }
        for(j=1;j<=3;j++)
        {
            for(k=1;k<=j;k++)
            {
                if(dp[i+1][k]+(pos[i]!=j)<dp[i][j])
                {
                    dp[i][j]=dp[i+1][k]+(pos[i]!=j);
                }
            }
        }
    }
    min2=min(dp[1][1],min(dp[1][2],dp[1][3]));
    ans=min(min1,min2);
}

int main()
{
    freopen("line.in","r",stdin);
    freopen("line.out","w",stdout);
    init();
    solve();
    printf("%d\n",ans);
    return 0;
}
```

4. 魔法阵[1]

(magic.c/pas/cpp/in/out)

[背景]

在 Alyosha 的带领下，龙族赢得了战争。但是战争带来的永远是创伤。

Alyosha 的伟大之处不在于他过人的军事才能，而是在于他在战争后为这片大陆重新带来了生机。

Alyosha 为了实现他的伟大愿望，布置了一个上古时期流传下来的魔法阵。这个魔法阵中有 N 颗魔法石，方便起见，将他们编号 1~N，每块魔法石都有一个法力值，为 1~10 之间（含 1 和 10）的正整数。某些魔法石之间用一些魔纹连接，连接是双向的，魔纹有一定的长度。魔法石之间可以通过其他魔法石和这些魔法石之间的魔纹连通，一个连通的值即是这些魔纹的长度和。两块魔法石之间的距离就是它们之间所有连通的值中最小的那个。每块魔法石最多和 10 块魔法石直接用魔纹连接。当 Alyosha 将自己的魔法力注入阵中后，魔法阵就可以与上古时期的生命之神的法力沟通，给大陆带来生命力。

为了更好地了解魔法阵的结构，我们做出了如下定义：

魔法石 i 对魔法石 j 敏感，当且仅当不存在魔法石 k（k 可能与 i 相等）到 i 的距离比 j 到 i 的距离短且 k 的魔法值比 j 的魔法值高。这时（i，j）叫做一个敏感的魔法石对。

显然，i 对 j 敏感并不表示 j 对 i 敏感。也就是说（i，j）与（j，i）是不同的。魔法石 i 对自己对自己是不敏感的。

现在，给出这个魔法阵的具体情况，请你求出所有敏感的魔法石对的数目。

[输入]

第一行：N（N<2000）和 M，表示魔法石数目和魔纹数目。

第 2..N+1 行：每行一个正整数 rank(i)，表示魔法石 i 的法力值。

第 N+2～N+M+1 行：每行三个正整数 i，j，l(i,j)，表示 i 与 j 之间有一道长度为 l(i,j)的魔纹。

[输出]

一行，表示敏感的魔法石对的数目。数据保证结果的最大值是 3*n。

[样例输入]

```
6 5
4
2
3
2
2
4
1 2 1
2 3 1
3 4 1
4 5 1
5 6 1
```

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=354

[样例输出]

14

【分析】图论试题。

【骗分攻略】使用朴素算法。

首先观察到题中所说“两个魔法石间的距离”即为一个 Floyed 算法求两点间最短路。

直接枚举每对魔法石 i,j，再枚举中间魔法石 k，按要求判断(i,j)是否敏感。

时间复杂度为 O(n^3).

处理时特别注意输入两个权值不同的相同边时，要选择的更新。

```
#include<stdio.h>
int d[2001][2001]={0},w[2001]={0};
int main()
{ int n,m,a,b,t,i,j,k,tot=0;
  FILE *fp,*fp1;
  fp=fopen("magic.in","r");
  fp1=fopen("magic.out","w");
  fscanf(fp,"%d%d",&n,&m);
  for(i=1;i<=n;i++)
   fscanf(fp,"%d",&w[i]);
  for(i=1;i<=n;i++)
   for(j=1;j<=n;j++)
    if(i!=j)
     d[i][j]=100000000;
  for(i=1;i<=m;i++)
  { fscanf(fp,"%d%d%d",&a,&b,&t);
    if(d[a][b]>t)//若当前更优才更新
     d[a][b]=d[b][a]=t;
  }
  for(k=1;k<=n;k++)//Floyed 算法求最短路
   for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
     if(d[i][k]+d[k][j]<d[i][j])
      d[i][j]=d[i][k]+d[k][j];
  for(i=1;i<=n;i++)//按题目条件统计，注意细节
   for(j=1;j<=n;j++)
    if(i!=j)
   { for(k=1;k<=n;k++)
      if(d[k][i]<d[j][i]&&w[k]>w[j])
       break;
     if(k>n)
      tot++;
   }
  fprintf(fp1,"%d",tot);
  fclose(fp);
  fclose(fp1);
```

```
}
```

【期望】时间 30 分钟，得分 50 分[1]

【标程】暂无

总体看来，这套模拟赛是题目质量最高的一届，由于命题人 llxy7, Alyosha 等都是我们信息学竞赛组中的大牛，但题目有抄袭之嫌（四道题目似乎都有来源）。

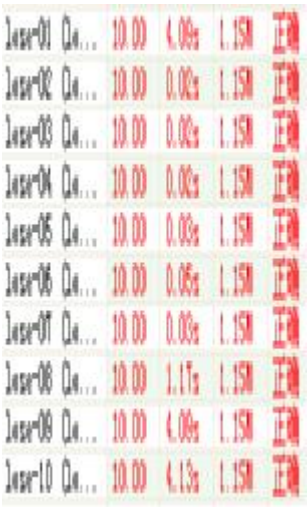
本次模拟赛首次使用了较规范的评测器 Cena，但这里比较选手成绩就不如清澄方便了。

考察的内容很有代表性，分别是模拟、BFS、动态规划、图论。模拟题具有物理味道，较有新意。BFS 试题要求较高的编程复杂度，并对特殊情况的处理提出较高的要求。

动态规划试题区分度较差，优秀的 O(n)算法与较差的 O(n^2)动态规划算法只差 20 分，笔者认为后者至多得 60 分。而后者采用了预处理的思想，又比单纯枚举的算法好。O(n)的算法没有经典模型，而要自己思考。所以这道题较能体现选手的水平。

最后一道试题难度过大，没有一人得分，应该放在更高水平的竞赛中。可以设计成标准算法难度很大，但用一些动态规划、错误的贪心能够得到部分分，例如 NOIP2008 第四题。

由于难度层次较为明显，所以出现了实战中常出现的现象：总分最高的选手，并不是每道题满分的选手，而是均衡分配比赛时间，每道题取得一定分数的选手。这也体现了我们“骗分”的宗旨：不求完美，只求高分。

总分	运输	流星雨	列队	魔法阵
	得分 用时 100 0.14 0 0 20 0.02 100 0.1 0 0 0 0 0 0 0 0 0 0 20 0.02 0 0 0 0	得分 用时 80 0.19 40 0.04 0 0 100 0.31 0 0 60 0.08 0 0 30 0.03 100 0.17 10 0.01 0 0 0 0	得分 用时 80 0.1 100 0.12 0 0 100 0.14 0 0 0 0 0 0 30 0.03 0 0 60 0.1 0 0 0 0	得分 用时 0 0 0 0 0 0 100 2.21 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

从以上的成绩分析表中我们发现，位居得分第一名的笔者只有第一题是满分，剩下两道题都是 80 分。而第二、三题满分的两人总得分都不是很高。有趣的是，较为简单的第一题却几乎没有人得分。笔者在时间安排上也出现问题，最后半个小时不是解决最后一个问题，骗一些分，而是盲目去调试第二、三题的程序，结果第二题发现问题，由 60 分变成了 80 分，但最后一题的 50 分就白丢了。[2]

由此可见，在信息学竞赛中，得分是关键，要形成正确的答题顺序和答题习惯，学会使用“非完美算法”，才能尽可能得分。

[1] 测评记录：http://www.rqnoj.cn/Status_Show.asp?SID=166368，RQNOJ 上测试数据已打乱顺序。

[2] 请各位大牛不要笑话，当时 NOIP 时笔者编程速度还没有上去，撰写本文时临时编写第四题的朴素算法大约只需要 7 分钟。本文针对 NOIP 普通水平的选手，所做的分析适用于他们，请各位大牛见谅。

9.5 NOIP 模拟赛（四）[1]

十七班大同盟模拟赛

1、LBJ 序列

命题人：石武

(abcz.c/pas/cpp/in/out)

【问题描述】

众所周知(?),一个 LBJ 序列是由字母 A,B,C,Z 组成的，这些字母对于 LBJ 有特殊的意义。例如，如果一个 LBJ 序列中包含 ZBC，那么 LBJ 今天可能会见“鬼”；如果包含 BC，LBJ 就可能很不走运。

现在我们有 m 个不好的子串（如，ZBC），我们不想让它们出现在 LBJ 序列中，请你求出长度为 n 的 LBJ 序列个数。

【输入格式】

第一行两个整数 m,n;

第二行到第 m+1 行，每行包含一个不好的子串（长度不超过 10）。

【输出格式】

一个整数，长度为 n 的 LBJ 序列个数。

【数据范围】

对于 30%的数据 0<n<=1000

对于全部数据 0<n<=2000000000，0<m<=10;

【样例输入】

4 3

AB

AC

AZ

AA

【样例输出】

36

【注释】

对于样例

BBA,BBB,BBC,BBZ,BCA,BCB,BCC,BCZ,BZA,BZB,BZC,BZZ

CBA,CBB,CBC,CBZ,CCA,CCB,CCC,CCZ,CZA,CZB,CZC,CZZ

ZBA,ZBB,ZBC,ZBZ,ZCA,ZCB,ZCC,ZCZ,ZZA,ZZB,ZZC,ZZZ

共 36 个 LBJ 序列

【来源】PKU 《DNA Sequence》

【分析】

本题是一个典型的递推题。解答较为复杂，请查看 PKU 题解。现摘录如下：

题目大意是，检测所有可能的 n 位 DNA 串有多少个 DNA 串中不含有指定的病毒片段。合法的 DNA 只能由 ACTG 四个字符构成。题目将给出 10 个以内的病毒片段，每个片段长度不超过 10。数据规模 n<=2 000 000 000。 下面的讲解中我们以 ATC,AAA,GGC,CT 这四个病毒片段为例，说明怎样像上面的题一样通过构图将问题转化为例题 8。我们找出所有病毒片段的

[1] http://boj.pp.ru/olympic/comp/4.htm

前缀， 把 n 位 DNA 分为以下 7 类：以 AT 结尾、以 AA 结尾、以 GG 结尾、以?A 结尾、以?G 结尾、以?C 结尾和以??结尾。其中问号表示“其它情况”，它可以是 任一字母，只要这个字母不会让它所在的串成为某个病毒的前缀。显然，这些分类是全集的一个划分（交集为空，并集为全集）。现在，假如我们已经知道了长度为 n-1 的各类 DNA 中符合要求的 DNA 个数，我们需要求出长度为 n 时各类 DNA 的个数。我们可以根据各类型间的转移构造一个边上带权的有向图。例如，从 AT 不能转移到 AA，从 AT 转移到??有 4 种方法（后面加任一字母），从?A 转移到 AA 有 1 种方案（后面加个 A），从?A 转移到??有 2 种方案（后面加 G 或 C），从 GG 到??有 2 种方案（后面加 C 将构成病毒片段，不合法，只能加 A 和 T）等等。这个图的构造过程类似于用有限状态自动机做串匹配。然后，我们就 把这个图转化成矩阵，让这个矩阵自乘 n 次即可。最后输出的是从??状态到所有其它状态的路径数总和。 题目中的数据规模保证前缀数不超过 100，一次矩阵乘法是三方的，一共要乘 log(n)次。因此这题总的复杂度是 100^3 * log(n)，AC

【矩阵乘法】[1]

题目大意是，检测所有可能的 n 位 DNA 串有多少个 DNA 串中不含有指定的病毒片段。合法的 DNA 只能由 ACTG 四个字符构成。题目将给出 10 个以内的病毒片段，每个片段长度不超过 10。数据规模 n<=2 000 000 000。

下面的讲解中我们以 ATC,AAA,GGC,CT 这四个病毒片段为例，说明怎样像上面的题一样通过构图将问题转化为例题 8。我们找出所有病毒片段的前缀，把 n 位 DNA 分为以下 7 类：以 AT 结尾、以 AA 结尾、以 GG 结尾、以?A 结尾、以?G 结尾、以?C 结尾和以??结尾。其中问号表示“其它情况”，它可以是任一字母，只要这个字母不会让它所在的串成为某个病毒的前缀。显然，这些分类是全集的一个划分（交集为空，并集为全集）。现在，假如我们已经知道了长度为 n-1 的各类 DNA 中符合要求的 DNA 个数，我们需要求出长度为 n 时各类 DNA 的个数。我们可以根据各类型间的转移构造一个边上带权的有向图。例如，从 AT 不能转移到 AA，从 AT 转移到??有 4 种方法（后面加任一字母），从?A 转移到 AA 有 1 种方案（后面加个 A），从?A 转移到??有 2 种方案（后面加 G 或 C），从 GG 到??有 2 种方案（后面加 C 将构成病毒片段，不合法，只能加 A 和 T）等等。这个图的构造过程类似于用有限状态自动机做串匹配。然后，我们就把这个图转化成矩阵，让这个矩阵自乘 n 次即可。最后输出的是从??状态到所有其它状态的路径数总和。

题目中的数据规模保证前缀数不超过 100，一次矩阵乘法是三方的，一共要乘 log(n)次。因此这题总的复杂度是 100^3 * log(n)，AC 了。

【源代码】[2]

```
#include <cstdio>
#define SIZE (1<<m)
#define MAX_SIZE 32
using namespace std;

class CMatrix
{
  public:
    long element[MAX_SIZE][MAX_SIZE];
    void setSize(int);
    void setModulo(int);
```

[1] Matrix67 ［推荐］十个用矩阵乘法解决的经典题目 http://www.matrix67.com/blog/archives/276
[2] Matrix67 ［推荐］十个用矩阵乘法解决的经典题目 http://www.matrix67.com/blog/archives/276

```
    CMatrix operator* (CMatrix);
    CMatrix power(int);
  private:
    int size;
    long modulo;
};

void CMatrix::setSize(int a)
{
  for (int i=0; i<a; i++)
    for (int j=0; j<a; j++)
      element[i][j]=0;
  size = a;
}

void CMatrix::setModulo(int a)
{
  modulo = a;
}

CMatrix CMatrix::operator* (CMatrix param)
{
  CMatrix product;
  product.setSize(size);
  product.setModulo(modulo);
  for (int i=0; i<size; i++)
    for (int j=0; j<size; j++)
      for (int k=0; k<size; k++)
      {
        product.element[i][j]+=element[i][k]*param.element[k][j];
        product.element[i][j]%=modulo;
      }

  return product;
}

CMatrix CMatrix::power(int exp)
{
  CMatrix tmp = (*this) * (*this);
  if (exp==1) return *this;
  else if (exp & 1) return tmp.power(exp/2) * (*this);
  else return tmp.power(exp/2);
}
```

```
int main()
{
  const int validSet[]={0,3,6,12,15,24,27,30};
  long n, m, p;
  CMatrix unit;

  scanf("%d%d%d", &n, &m, &p);
  unit.setSize(SIZE);
  for(int i=0; i<SIZE; i++)
    for(int j=0; j<SIZE; j++)
      if( ((~i)&j) == ((~i)&(SIZE-1)) )
      {
        bool isValid=false;
        for (int k=0; k<8; k++)isValid=isValid||(i&j)==validSet[k];
        unit.element[i][j]=isValid;
      }

  unit.setModulo(p);
  printf("%d", unit.power(n).element[SIZE-1][SIZE-1] );
  return 0;
}
```

2、子串清除[1]

命题人：吴鹏

Clear.c/pas/cpp

【问题描述】

我们定义字符串 A 是字符串 B 的子串当且仅当我们能在 B 串中找到 A 串。现在给你一个字符串 A，和另外一个字符串 B，要你每次从 B 串中从左至右找第一个 A 串，并从 B 串中删除它，直到 A 串不为 B 串的子串，问你需要进行几次删除操作。

[数据范围]

目标串长度小于 256，输入文件小于 500kb。

[样例数据]

Clear.in:

abbbbc

hwyebpabbbbciabbbbcekrbbmqqesfvhwpvfclrc

Clear.out

2

【分析】

由于时限为 2 秒，本题用朴素算法即可解决，关键是优化算法，卡常数。

【骗分攻略】只需要按照问题的要求操作，每次自左向右寻找第一个子串并将其删除，

[1] 提交地址：http://www.rqnoj.cn/Problem_Show.asp?PID=476

采用朴素的字符串匹配算法。

【期望】时间 30 分钟，80 分

【源代码 1】笔者的代码，80 分，超时 2 个点。事实再次证明，“骗分”具有强大的生命力。

```
#include<stdio.h>
#include<string.h>
char a[300],b[520000];
int main()
{ int i,j,t,la,lb,num=0,flag;
  FILE *fp,*fp1;
  fp=fopen("clear.in","r");
  fp1=fopen("clear.out","w");
  fscanf(fp,"%s%s",a,b);
  la=strlen(a);
  lb=strlen(b);
  while(1)
  { flag=0;i=0;
  while(i<lb)
  { if(b[i]==a[0])
    { t=i+1;
      for(j=1;j<la;j++,t++)
      { while(t<lb&&b[t]==0)
         t++;
        if(t>=lb||b[t]!=a[j])
         break;
      }
      if(j==la)
      { flag=1;
        num++;
        for(;i<t;i++)
         b[i]=0;
        i=t;
      }
      else i++;
    }
    else i++;
  }
  if(flag==0)
   break;
  }
  fprintf(fp1,"%d",num);
  fclose(fp);
  fclose(fp1);
  return 0;
```

lose-01	(lo...	10.00	4.09s	1.15M	正确
lose-02	(lo...	10.00	0.02s	1.15M	正确
lose-03	(lo...	10.00	0.02s	1.15M	正确
lose-04	(lo...	10.00	0.02s	1.15M	正确
lose-05	(lo...	10.00	0.03s	1.15M	正确
lose-06	(lo...	10.00	0.05s	1.15M	正确
lose-07	(lo...	10.00	0.03s	1.15M	正确
lose-08	(lo...	10.00	1.17s	1.15M	正确
lose-09	(lo...	10.00	4.09s	1.15M	正确
lose-10	(lo...	10.00	4.13s	1.15M	正确

```
}
```

【源代码 2】90 分，错 1 个点，功亏一篑。

从根本上认识了这个算法，使用 KMP 算法优化匹配过程，速度很快，值得学习。

【期望】时间 60 分钟，100 分（小问题，克服就 AC 了）

Clear-01 (c...	10.00	0.02s	8.90M	正确
Clear-02 (c...	10.00	0.02s	8.90M	正确
Clear-03 (c...	10.00	0.02s	8.90M	正确
Clear-04 (c...	10.00	0.02s	8.90M	正确
Clear-05 (c...	10.00	0.02s	8.90M	正确
Clear-06 (c...	10.00	0.05s	8.90M	正确
Clear-07 (c...	10.00	0.03s	8.90M	正确
Clear-08 (c...	10.00	0.05s	8.90M	正确
Clear-09 (c...	0.00	0.03s	8.90M	不匹配
Clear-10 (c...	10.00	0.02s	8.90M	正确

```
#include "stdio.h"
#include "string.h"
#define N 1000000
int next[300];
char s1[300],s2[N];
int pre[N],ind[N];

void getnext(char *pipei)
{
    long j=-1,i=1;
    int len=strlen(pipei);
    memset(next,255,sizeof(next));
    for (;i<len;i++)
    {
        for (;(j>=0)&&(pipei[j+1]!=pipei[i]);j=next[j]);
        if (pipei[j+1]==pipei[i]) j++;
        next[i]=j;
    }
}

long KMP(char *mubiao,char *pipei)
{
    int j=-1,i=1,res=0;
    int k,l;
    int lenmb=strlen(mubiao+1),lenpp=strlen(pipei);
    pre[0]=pre[1]=0;
    for (;i<=lenmb;i++)
    {
        pre[i+1]=i;
        for (;(j>=0)&&(pipei[j+1]!=mubiao[i]);j=next[j]);
        if (pipei[j+1]==mubiao[i]) j++;
        if (j==(lenpp-1))
        {
            res++;
            k=i; l=lenpp;
            for(l=0;l<lenpp;l++)
                k=pre[k];
            pre[i+1]=k;
            if(!k)
```

```
                j=-1;
            else
                j=ind[k];
        }
        ind[i]=j;
    }
    return res;
}

main()
{
    freopen("clear.in","r",stdin);
    freopen("clear.out","w",stdout);
    gets(s1);
    gets(s2+1);
    getnext(s1);
    printf("%d\n",KMP(s2,s1));
    return 0;
}
```

【源代码 3】AC，贪心方法

本题源代码已经找不到了。
据称是采用了贪心方法。
评测机较慢，故放宽了时间限制。

Clear-01 (c...	10.00	0.02s	1.25M	正确
Clear-02 (c...	10.00	0.02s	1.25M	正确
Clear-03 (c...	10.00	2.09s	3.31M	正确
Clear-04 (c...	10.00	1.88s	1.86M	正确
Clear-05 (c...	10.00	1.11s	1.86M	正确
Clear-06 (c...	10.00	0.75s	11.83M	正确
Clear-07 (c...	10.00	0.81s	11.83M	正确
Clear-08 (c...	10.00	0.84s	11.83M	正确
Clear-09 (c...	10.00	1.59s	11.83M	正确
Clear-10 (c...	10.00	0.83s	1.86M	正确

3、Snow 的减肥计划

命题人：刘延珍

lose.c/cpp/pas

[背景]

Snow 最近超重了，但它又是鼎鼎大名的爱吃一族。为了健康和美味兼顾，他不得不损耗一些脑细胞来为每一顿饭想一个食谱。

[题目描述]

Snow 想吃的食物有 n 种，编号为 1~n，每一种食物都有一个基础肥胖指数 f[i]（1≤i≤n，f[i]＞0）。Snow 吃一顿饭需要 t 个连续时间单位，每个时间单位吃一种食物（不同的时间单位可以吃相同的食物）。但 Snow 吃的每种食物会和上一个时间单位吃的食物发生一定的作用（第一个时间单位吃的食物除外），导致这种的食物的肥胖指数发生改变。若上一个时间单位吃的食物种类为 i，现在所处时间单位吃的食物种类为 j（i 可以等于 j），则此时 j 的肥胖指数变为 a[i%m+1][j]（m 为一个任意给定的整数，保证 1≤m≤n）。Snow 的胃很奇特，在每个不同的时间单位会对食物产生不同的抵制作用，抵制作用的表现是使第 k 个时间单位所吃的食物的肥胖指数下降 s[k]（s[k]≥0，若 s[k]大于此时食物的肥胖指数，

则此食物最终肥胖指数为 0)。你的任务便是为 Snow 指定一个食谱，使每个时间单位 Snow 所吃食物的肥胖指数之和最小。

[输入文件](lose.in)

第 1 行，一个整数 n。

第 2 行，n 个整数，为 f[1]~f[n]。

第 3 行，一个整数 m。

以下 m 行，每行 n 个整数，其中第 i 行第 j 个整数表示 a[i][j]。

下一行为一个整数 t。

再下一行为 t 个整数，表示 s[1]~s[t]。

[输出文件](lose.out)

一个整数，表示最小的肥胖指数之和。

[样例输入]

```
4
5 10 15 20
1
4 7 13 17
3
1 2 3
```

[样例输出]

```
7
```

[说明]

第一个时间单位吃第一种食物，肥胖指数为 5-1=4；第二个时间单位吃第一种食物，肥胖指数为 4-2=2；第三个时间单位吃第一种食物，肥胖指数为 4-3=1。所以最小肥胖指数之和为 7。

[数据规模]

对于 10%数据，保证 n,m,t≤10。

对于 30%数据，保证 n≤100，m,t≤30。

对于 60%数据，保证 n,,m,t≤100。

对于 100%数据，保证 n≤1000，m,t≤100。

保证答案≤2147483647。

【分析】

本题是二维动态规划题。

【源代码 1】时间 40 分钟，70 分[1]

```
#include "stdio.h"
#include "stdlib.h"
#define OO 2147483647
#define N 1010
#define M 110
int f[M][N];
int a[M][N];
int org[N],s[M];
int n,t,m;
```

lose-01	(lo...	10.00	4.09s	1.15M	正确
lose-02	(lo...	10.00	0.02s	1.15M	正确
lose-03	(lo...	10.00	0.02s	1.15M	正确
lose-04	(lo...	10.00	0.02s	1.15M	正确
lose-05	(lo...	10.00	0.03s	1.15M	正确
lose-06	(lo...	10.00	0.05s	1.15M	正确
lose-07	(lo...	10.00	0.03s	1.15M	正确
lose-08	(lo...	10.00	1.17s	1.15M	正确
lose-09	(lo...	10.00	4.09s	1.15M	正确
lose-10	(lo...	10.00	4.13s	1.15M	正确

[1] 由于评测机 CPU 为 1.73GHz，时限稍微放宽。

```
int input()
{
    int i,j;
    scanf("%d",&n);
    for(i=1;i<=n;i++)
    {
        scanf("%d",&org[i]);
    }
    scanf("%d",&m);
    for(i=1;i<=m;i++)
    {
        for(j=1;j<=n;j++)
            scanf("%d",&a[i][j]);
    }
    scanf("%d",&t);
    for(i=1;i<=t;i++)
    {
        scanf("%d",&s[i]);
    }
}

int min(int a,int b){return a<b?a:b;}

int myabs(int a){return a<0?0:a;}

int work()
{
    int i,j,k;
    int ans=OO;
    for(i=1;i<=t;i++)
        for(j=1;j<=n;j++)
            f[i][j]=OO;
    for(i=1;i<=n;i++)
    {
        f[1][i]=myabs(org[i]-s[1]);
    }
    for(i=2;i<=t;i++)
    {
        for(j=1;j<=n;j++)
        {
            for(k=1;k<=n;k++)
            {
```

```
f[i][j]=min(f[i][j],f[i-1][k]+myabs(a[k%m+1][j]-s[i]));
            }
            if(i==t)
                ans=min(ans,f[i][j]);
        }
    }
    printf("%d\n",ans);
}

int main()
{
    freopen("lose.in","r",stdin);
    freopen("lose.out","w",stdout);
    input();
    work();
    return 0;
}
```

【源代码 2】时间 40 分钟，70 分

```
var n,m,t,i,j:integer;
    f:array[1..1000] of longint;
    a:array[1..100,1..1000] of longint;
    s:array[1..100] of longint;

    kao:array[0..100,1..1000] of longint;

procedure main;
var i,j,k,temp:longint;
begin
  for i:=1 to t do
    for j:=1 to n do kao[i,j]:=maxlongint;
  for i:=1 to n do
     if f[i]-s[1]>=0 then kao[1,i]:=f[i]-s[1]
       else kao[1,i]:=0;

  for i:=2 to t do
    for j:=1 to n do
       for k:=1 to n do
         begin
                 if a[(k mod m)+1,j]-s[i]>=0 then temp:=a[(k mod
m)+1,j]-s[i]
            else temp:=0;
                                if  kao[i-1,k]+temp<kao[i,j]  then
kao[i,j]:=kao[i-1,k]+temp;
```

```
        end;

  temp:=maxlongint;
  for i:=1 to n do if kao[t,i]<temp then temp:=kao[t,i];
  writeln(temp);
end;

begin
  assign(input,'lose.in');
  reset(input);
  assign(output,'lose.out');
  rewrite(output);

  readln(n);
  for i:=1 to n do read(f[i]);
  readln(m);
  for i:=1 to m do
    for j:=1 to n do
      read(a[i,j]);
  read(t);
  for i:=1 to t do read(s[i]);

  main;

  close(input);
  close(output);
end.
```

lose-01	(lo...	10.00	3.19s	7.36M	正确
lose-02	(lo...	10.00	0.05s	7.36M	正确
lose-03	(lo...	10.00	0.03s	7.36M	正确
lose-04	(lo...	10.00	0.02s	7.36M	正确
lose-05	(lo...	10.00	0.05s	7.36M	正确
lose-06	(lo...	10.00	0.06s	7.36M	正确
lose-07	(lo...	10.00	0.06s	7.36M	正确
lose-08	(lo...	10.00	0.92s	7.36M	正确
lose-09	(lo...	10.00	3.19s	7.36M	正确
lose-10	(lo...	10.00	3.19s	7.36M	正确

4、LBJ 的任务

命题人：孟赫

task.c

【背景】

自从 LBJ 加入大联盟以来，总是认真完成组织上的任务。这次倒 S 大联盟交给他一个艰巨的任务——****S 的犯罪事实。而 S 将她全部残害人员名单都放在了她的电脑里。

在一个伸手不见五指的黑夜里，一个胖胖的黑影闪过。突然，他消失了！

吧唧——啪——啪

“妈呀，谁扔的香蕉皮？”

许久许久……

那个黑影爬到了 S 的计算机前，打开了它。

闪光的显示器照亮了那黑影：啊，竟然是 LBJ！

“咦？竟然有密码！”LBJ 挠了挠头，“让我试试。”

LBJ 输入“ZJS”——“密码错误”;

“JSZ”——“密码错误”;

“SJZ”,”ZSJ”,”JZS”,”SZJ”都是错误。

LBJ 甚至把 ”ZBC”,”UGLY”都试过了，可还是错误；LBJ 这时输入“PangZiZhang”，登陆界面跳出一个对话框：“竟敢骂我！想要登陆吗？只要做出我的题!”。

一个算式映入 LBJ 的眼前，“只要你算出结果，我便让你登陆。”

由于 S 大脑比较缺魂儿，就只会用小写字母进行计算，且它认为 a-z 分别代表 0-25。她还认为乘法便是一位一位的乘。且位数不够在前面补’a’。下面给出实例：

abc+abc=(a+a)(b+b)(c+c)=ce;

bb*cc=(b*c)(b*c)=cc;

zz*zy=(z*z)(z*y)=yyc;

（其中 z*z=yb,z*y=xc,yba+xc=yyc。）

bc*bbb=(a*b)(b*b)(b*c)=bc;

（其中 a*b=a,b*b=b,b*c=c,所以结果为 abc,又因不能有多余的’a’，所以结果为 bc。）

LBJ 可没有多少时间了，请你帮帮他。

【输入数据】(task.in)：

一个字符串，包含小写字母’a’-‘z’,’+’,’*’,’(‘,’)’,长度不超过 4000 位。

【输出数据】(task.out)：

一个字符串，计算的结果，要求不允许有多余的’a’。

【样例输入】

abc+bcd*(hdj+jdb)+hgh*tsy+anv

【样例输出】

fymn

【分析】

考察表达式求值、高精度运算，应特别注意这里定义的“乘法”有所不同，求值的顺序要注意，不满足交换律，只能按照优先级顺序从左至右严格计算。应注意特殊情况的处理，例如为零时要输出’a’.

【骗分攻略】

本题主要考察细心。这里笔者采用了一种原创的表达式求值方法，读者可以参考。

【期望】50 分钟，100 分

【源代码】

```
#include<stdio.h>
int f[4010][500]={0};
int ord[4010]={0},use[4010]={0};
void mul(int out[],int a[],int b[])
{ int i;
  for(i=0;i<500;i++)
    out[i]=a[i]*b[i];
  for(i=0;i<499;i++)
  { out[i+1]+=out[i]/26;
    out[i]%=26;
```

```
  }
}
void add(int out[],int a[],int b[])
{ int i;
  for(i=0;i<500;i++)
    out[i]=a[i]+b[i];
  for(i=0;i<499;i++)
  { out[i+1]+=out[i]/26;
    out[i]%=26;
  }
}
int main()
{ int l,i,j,k,now=1,num=0,max=0,t=0,ans;
  char s[4010];
  FILE *fp,*fp1;
  fp=fopen("task.in","r");
  fp1=fopen("task.out","w");
  fscanf(fp,"%s",s);
  l=strlen(s);
  for(i=0;i<l;i++)
  { if(s[i]=='+'||s[i]=='*')
    { if(s[i]=='+')
       ord[i]=now;
      else ord[i]=now+1;
    }
    else if(s[i]=='(')
     now+=2;
    else if(s[i]==')')
     now-=2;
    else if(i==0||s[i-1]<'a'||s[i+1]>'z')
    { use[i]=1;
      for(j=i;j<l;j++)
       if(s[j]<'a'||s[j]>'z')
        break;
      t=0;j--;
      while(j>=i)
      { f[i][t]=s[j]-'a';
        t++;j--;
      }
    }
  }
  for(i=0;i<l;i++)
  { max=0;
    for(j=0;j<l;j++)
```

```
     if(ord[j]>ord[max])
      max=j;
    if(ord[max]==0)
     break;
    ord[max]=0;
    for(j=max-1;j>=0;j--)
     if(use[j])
      break;
    for(k=max+1;k<l;k++)
     if(use[k])
      break;
    if(s[max]=='+')
     add(f[max],f[j],f[k]);
    if(s[max]=='*')
     mul(f[max],f[j],f[k]);
    use[j]=use[k]=0;
    use[max]=1;
  }
  for(i=0;i<l;i++)
   if(use[i]!=0)
    break;
  ans=i;
  i=499;
  while(f[ans][i]==0)
   i--;
  if(i<0)
   fprintf(fp1,"a");
  while(i>=0)
   fprintf(fp1,"%c",f[ans][i--]+'a');
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

有趣的是，这道题出题人的标程、一位同学的程序得到的是另一组截然不同的答案，经过手算验证，推翻标程，上述程序是笔者 AC 的程序。经进一步分析，他们犯的同样错误是把这里“乘法”的法则搞错了，以为还是按照平时高精度运算来乘，导致出错。因此我们在做题时，一定要注意审题，观察题目中给出的定义、运算是否与我们平时的认识相同，而不能凭感觉。尤其是题目中出现“定义一种运算”“定义一种变量表示……”，更不能想当然，而要严格按照题目中的定义设计算法。

本次模拟赛由于准备仓促，所以收到的效果并不好，第二、三题甚至最后没有给出标准程序，但这次模拟赛的题目难度有一定水平，且题目也都有较好的区分度。

第一题考察递推、矩阵运算，是最难的一道题目。第二题考察字符串匹配，尤其是优化。

第三题考察动态规划，也需要优化。第四题考察表达式处理、高精度运算，是最简单的一道试题。四道试题各有侧重，虽然还是没有出现有关非线性数据结构的试题，但考察较为全面，有一定难度，使得“得分容易，得满分难”，也符合今后命题趋势。

这样的试题，适合使用本文的“骗分”思想。对于难度较大的竞赛，尽可能得到每道题的部分分，而不要追求一道题的满分，就是我们考试高分的秘诀。

9.6 NOIP2008[1]

NOIP2008，相信每位读到这里的 OIer 都感慨万千。也许您实现了心中的梦想，成功拿到省一等奖的证书，等待着大学的录取通知书；也许您站得更高，看得更远，正在同笔者一道向 NOI2009 冲刺；也许这是一次失利，一切重头再来，明年还有机会，车到山前必有路；也许这是您 OI 生涯一个并不愉快的终止符，正在向高考或其他学科竞赛冲刺……

无论结果如何，OI 始终陪伴你我。回顾 2008，展望 2009，我们会有崭新的收获与期待。

由于普及组的试题过于简单，提高组的试题难度又类似往年的普及组，本文只讨论提高组试题。

9.6.1 笨小猴

【问题描述】

输入一个由小写字母构成的字符串，统计出现最多与最少字母的个数，若两数之差为质数，输出“Lucky Word”和差值；否则输出“No Answer”和 0.

【题目类型】

模拟

【建议编程时间】

10 分钟（细心一些，避免出错）。

【解题分析】

1、读入字符串（文件）

2、构造一个数组，记录 a-z 各字符出现的次数。枚举字符串中每个字符，将该字符对应数组元素加一。

3、枚举数组中 a-z，找出最大值和非零最小值，求出它们的差。

4、判断差值是否为素数，数据规模很小，可用试除法。注意 0，1 的特殊情况。

5、输出，注意大小写、换行符。

【源代码】[2]

```
#include<stdio.h>
int prime(int n)
{ int i;
  if(n==1||n==0)
   return 0;
  for(i=2;i<n;i++)
   if(n%i==0)
```

[1] 改编自笔者《NOIP2008 解题报告》。

[2] NOIP2008 一节中涉及的源代码，除注释外，均为考试时真实的源程序，具有参考价值。

```
    return 0;
  return 1;
}
int main()
{ int n,i,l,num[1000]={0},max=0,min=10000;
  char a[1000];
  FILE *fp,*fp1;
  fp=fopen("word.in","r");
  fp1=fopen("word.out","w");
  fscanf(fp,"%s",a);
  l=strlen(a);
  for(i=0;i<l;i++)
   num[a[i]]++;
  for(i='a';i<='z';i++)
  { if(num[i]>max)
     max=num[i];
    if(num[i]!=0&&num[i]<min)
     min=num[i];
  }
  if(prime(max-min)==1)
   fprintf(fp1,"Lucky Word\n%d",max-min);
  else fprintf(fp1,"No Answer\n0");
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

【骗分攻略】

仅仅是这样一道简单的试题，就有不少选手因为很小的一处失误而丢掉分数。例如下面的 10 分程序：

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
int cmt[26];
int main()
{
    FILE *fin,*fout;
    char s[200];
    int len;
    int i;
    int max=-1,min=200,key;
    int flag=1;
    fin=fopen("word.in","r");
    fout=fopen("word.out","w");
    fscanf(fin,"%s",&s);
```

```
    len=strlen(s);
    for(i=0;i<26;i++)
      cmt[i]=0;
    for(i=0;i<len;i++)
      cmt[s[i]-'a']++;
    for(i=0;i<26;i++)
    {
                  if(cmt[i]>max)
                    max=cmt[i];
                  if(cmt[i]<min&&cmt[i]!=0)
                    min=cmt[i];
    }
    key=max-min;
    if(key<=3) flag=0; //判断素数出错：差为 2、3 时应为素数
    for(i=2;i<key;i++)
      if(key%i==0)
      {
                flag=0;
                break;

      }
    if(flag) fprintf(fout,"Lucky Word\n%d",key);
    else fprintf(fout,"No Answer");     //不服从输出规定，忘记 0
    fclose(fin);
    fclose(fout);
    return 0;
}
```

上述程序出现的严重问题，恰好是本文所述“细节错误”的两点：特殊情况、输入输出。这些问题只要经过实际测试数据调试，就能避免。竞赛时此选手一定是觉得题目太简单，未加调试就以为高枕无忧了。

9.6.2 火柴棒等式

【问题描述】
给 n 根火柴棒，问能拼出多少种形如“A+B=C”的等式。
【题目类型】
搜索
【建议编程时间】
30 分钟。若某些问题未考虑到，调试 20 分钟。[1]
【解题分析】
【算法一】
1、 读入整数 n，减去加号和等号需要的 4 根火柴。

[1] 笔者甚至用了接近 60 分钟时间，由于多处调试出现问题，反复修改。

2、采用搜索方法，先递归枚举火柴拆分成每个数字对应根数（存在数组里），再枚举加号和等号的位置，计算对应的 A，B，C，最后判断 A+B=C，统计总数。

3、输出 tot，注意换行。

【编程注意】

1、注意最高位不能为零，即除非该数为零，最高位不为零。

2、递归函数两个入口，表示当前剩余火柴根数、当前位数。枚举下一根火柴的数字，记录，递归（剩余-当前火柴，位数+1）。

3、当剩余根数为零时，枚举 A，B，C 的拆分方式。

4、计算 A 时，先将 A 设为首位，若首位为零且位数大于 1，则失败退出（最高位不为零）从首位+1 到末位：A*=10, A+=a[i]. 字符串转为整数的常用方法。

5、注意个数为 0 时的特殊处理。

【源代码】用时 0.12s，100 分

```
#include<stdio.h>
int tot=0,a[11]={6,2,5,5,4,5,6,3,7,6},f[20]={0};
void solve(int n,int now)
{ int i,j,k,x,y,z;
  if(n==0)
  { for(i=0;i<now-2;i++)
    { x=f[0];
      for(j=1;j<=i;j++)
      { x*=10;
        if(x==0)
         break;
        x+=f[j];
      }
      if(j<=i)
       continue;
      for(j=i+1;j<now-1;j++)
      { y=f[i+1];
        for(k=i+2;k<=j;k++)
        { y*=10;
          if(y==0)
           break;
          y+=f[k];
        }
        if(k<=j)
         continue;
        z=f[j+1];
        for(k=j+2;k<now;k++)
        { z*=10;
          if(z==0)
           break;
          z+=f[k];
        }
```

```
        if(k<now)
         continue;
        if(x+y==z)
         tot++;
      }
    }
    return;
  }
  for(i=0;i<10;i++)
   if(n>=a[i])
    { f[now]=i;
      solve(n-a[i],now+1);
      f[now]=0;
    }
}
int main()
{ int n;
  FILE *fp,*fp1;
  fp=fopen("matches.in","r");
  fp1=fopen("matches.out","w");
  fscanf(fp,"%d",&n);
  solve(n-4,0);
  fprintf(fp1,"%d",tot);
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

【算法二】

上面的算法是最简单的想法，只是简单的对火柴棒个数进行枚举，找出所有可能的组成数字情况，再检验是否满足等式。这也是人类解决此问题时自然的想法。

为什么一定要对不同的数字组成进行枚举，再检验等式呢？我们不妨换一个角度，充分抓住问题的关键，对不同的等式进行枚举，再检验火柴棒个数是否符合要求。这样转念一想，思路就豁然开朗了，编程也比前面简单得多。

这再次证明：搜索问题的**搜索顺序**是至关重要的，要考虑好将哪个条件作为枚举时的出发点，将哪个条件作为检验当前解正确性的工具。不同的搜索顺序，编程的复杂度、产生的效果截然不同。

那么我们思考时为什么没有采用算法二呢？显然，算法二的效率要比算法一低得多，算法二要对所有可能的算式进行枚举，而算法一只需枚举火柴棒个数等于定值的一小部分。人脑适合处理复杂而情况少的问题，因为逻辑思维能力强，但运算速度慢；电脑适合处理简单而情况多的问题，因为运算速度快，但逻辑思维差，需要程序员复杂的编程。因此我们在编写计算机程序时，应该注意到并利用好计算机的特点，不要总是使用人脑的观点要求电脑，要让计算机的 CPU 为我们工作，而不是我们自己为冗长的程序敲打键盘。

当然，算法一并不是一无是处，它更具备普适性，并且效率更高。对于更多的火柴棒，

例如 100 根，算法二会因为冗余枚举太多而无法使用；对于更复杂的等式，例如有五项而不是三项，也必须使用算法一，因为算法二同样存在枚举范围过于宽泛的问题。只是对于本题来说，对症下药的是算法二。

【编程注意】

（1）使用数组记录，使用循环统一判断，避免逐一处理。

（2）计算每个数使用的火柴棒个数，这可以用逐位取模相加的办法解决。

（3）循环枚举两个加数，注意到数字的范围<1000，判断等式是否符合给定的根数。

（4）累加答案、输出。

【源代码】[1]用时 0.14s，100 分

```
#include <iostream>
using namespace std;
int a[10]={6,2,5,5,4,5,6,3,7,6},n,k[2000];
int main(){
  freopen("matches.in","r",stdin);
  freopen("matches.out","w",stdout);
  memset(k,0,sizeof(k));
  k[0]=6;
  for (int i=1;i<2000;i++){
    for (int p=i;p>0;p/=10) k[i]+=a[p%10];
  }
  scanf("%d",&n); n-=4;
  int ans=0;
  for (int i=0;i<1000;i++)
    for (int j=0;j<1000;j++)
      if (k[i]+k[j]+k[i+j]==n) ans++;
  printf("%d\n",ans);
  return 0;
}
```

【骗分攻略】

观察到本题数据规模很小，可以使用“打表”的方法解决。程序不再给出，相信每个有 Online Judge“骗分”经验的读者都深有体会。但有趣的是，笔者所在地区本题满分的选手都没有打表，而打表的反而没有满分。这是因为即使是最差的搜索，对于本题较小的数据范围，也是能在瞬间解决的，根本没有必要交表。交表只是针对可能的输入少，所需时间又长的问题，为了避免超时而采用的权宜之计，通常用于数学类问题。如果时间效率很高，没有超时之虞，又何必耽误工夫去做一个“表”呢？所以“骗分”手段不能滥用，用不好反而会弄巧成拙。

【错例 1】例如这样一个得到 40 分的程序：

```
#include<stdio.h>
#include<stdlib.h>
int
ans[25]={0,0,0,0,0,0,0,0,0,0,0,0,1,2,8,9,6,9,30,37,46,72,94,135
};
```

[1] 来源：石家庄 郭子晨

```
int n;
int main()
{
    freopen("matches.in","r",stdin);
    freopen("matches.out","w",stdout);
    scanf("%d",&n);
    printf("%d\n",ans[n]);
    return 0;
}
```

后面的一些数据计算错误，一定是某种情况未考虑到。这样，错误的表即使打上去也没有意义。

事实上，笔者在开始编写此题时，也得到了相同的结果，后来通过笔算验证，发现一个重大漏洞，于是避免了这个错误。问题发生在“0”这个特殊数字上，由于标准的数字除一位数外，不能有前导的零，于是采用先构造数再验证等式搜索方法的程序必须注意特殊判断。但不要将一位数 0 也同时除去，否则又少了。

在排除错误的过程中，采用了上文所述的“输出语句”法，首先直接输出三个数，发现都满足等式，但火柴棒数少了一些，百思不得其解。然后采用更加精细的输出方式，将所有求出的解的各个数字依次输出，立即真相大白：将形如 01 的“数”也计算在内了。

【错例 2】下面这段程序是 90 分的。经检测，对于 22 根火柴棒的问题，比标准输出少计算了一种情况。再使用输出语句进行检测，发现缺少的恰好是 0+0=0 这种最特殊的情况。

针对这种错误，我们在编写程序之时，对于 0 之类的特殊情况要进行深入的思考，观察是否有关于“0”的已求出却不合题意，或符合题意但未求出的情况。

```
#include"stdio.h"
#include"stdlib.h"

int n;
int num[10]={6,2,5,5,4,5,6,3,7,6};
int match[12][100];
int ans=0;

void pretreatment(void);
int judge(int numb);

main()
{
   int i,j,i1,j1;
   freopen("matches.in","r",stdin);
   scanf("%d",&n);
   fclose(stdin);
   n-=4;
   freopen("matches.out","w",stdout);
   if(n<=7)
      printf("%d",0);
   else
```

```
   {
      pretreatment();
      for(i=2;i<=n-4&&i<=11;i++)
         for(j=2;j<=n-i-2&&j<=11;j++)
            for(i1=1;i1<=match[i][0];i1++)
               for(j1=1;j1<=match[j][0];j1++)
               {
                  if(judge(match[i][i1]+match[j][j1])==n-i-j)
                     ans++;
               }
      printf("%d",ans);
   }
   fclose(stdout);
   return 0;
}

void pretreatment(void)
{
   int i,j,k,l;
   for(i=0;i<10;i++)
   {
      l=++match[num[i]][0];
      match[num[i]][l]=i;
   }
   for(i=9;i>0;i--)
      for(j=0;j<10;j++)
      {
         if(num[i]+num[j]<=11)
         {
            l=++match[num[i]+num[j]][0];
            match[num[i]+num[j]][l]=i*10+j;
         }
      }
   for(i=9;i>0;i--)
      for(j=0;j<10;j++)
         for(k=0;k<10;k++)
            if(num[i]+num[j]+num[k]<=11)
            {
               l=++match[num[i]+num[j]+num[k]][0];
               match[num[i]+num[j]+num[k]][l]=i*100+j*10+k;
            }
}

int judge(int numb)
```

```
{
   int m=0;
   while(numb!=0)
   {
      m+=num[numb%10];
      numb/=10;
   }
   return m;
}
```

【错例 3】60 分。经测试，这个程序对 21、22、23、24 四个数据计算错误，都少算了一些。原因是，选手误以为数字一定是两位数，从而忽略了三位数的情况。例如 n=22 时，恰好少了 0+111=111，111+0=111，110+1=111，1+110=111 四种情况。

这提示我们，解决问题是一定不要想当然，尤其是搜索类问题，实际出现的结果可能超出我们直观想象的范围，应该在主观估计的基础上适当扩大搜索范围，并在调试程序时不仅注意得到解的正确性，还要注意解的完整性，对于小规模数据一定要用笔算验证。

```
const
  num:array[0..9]of integer=
    (6,2,5,5,4,5,6,3,7,6);
var
  i,j,now,ans:longint;
  n:longint;
function suan(x:longint):longint;
var i,j,sum:longint;
begin
  if x=0 then exit(num[0]);
  sum:= 0;
  while x>0 do begin
    inc(sum,num[x mod 10]);
    x:= x div 10;
  end;
  exit(sum);
end;

begin
  assign(input,'matches.in');reset(input);
  assign(output,'matches.out');rewrite(output);
  readln(n);
  close(input);
  dec(n,4);
  if n<=0 then begin
    writeln(0);
    close(output);
    halt;
  end;
```

```
  ans:= 0;
  for i:= 0 to 99 do begin  //Where are the 3-dight numbers?
    for j:= 0 to 99 do begin
      now:= i+j;
      if suan(i)+suan(j)+suan(now)<>n
        then continue
        else begin
          inc(ans);
        end;
    end;
  end;
  writeln(ans);
  close(output);
end.
```

【错例 4】30 分。这个程序在枚举上存在较为严重的问题。在循环枚举第一个数 a 的终止条件中，认为只要发生一次木棍长度之和超过限制，就退出循环。这样，当 n=13 时，首先枚举 0 就超过了限制，不再继续，后面的 1+1=2 当然就不可能被找到了。事实上，数大、数字之和大、木棍数多三者之间没有必然联系。

这使笔者想到一个笑话：某警察局接到一封恐吓信，称某地的定时炸弹将在小时、分钟的数字都相等时爆炸，于是警察局在 1：11，2：22 直到 5：55 都顺利排除了炸弹，因为不再有 6：66，便放松了警惕。但爆炸还是发生了，因为 11：11 还是存在的。这个笑话也启示我们，不能顺着原来的思路“想当然”，而要注意到问题规模扩大时，原有的规律可能不再适用，需要另辟蹊径。

```
#include"stdio.h"
int cost[10]={6,2,5,5,4,5,6,3,7,6};
int getlen(int a)
{if(a<10)
  return(cost[a]);
 else
 {int sum=0;
  while(a>0)
  {sum+=cost[a%10];
   a/=10;
  }
  return(sum);
 }
}
main()
{FILE *fpi,*fpo;
 fpi=fopen("matches.in","r");
 fpo=fopen("matches.out","w");
 int n;
 int ca=0,cb=0;
```

```
 int a=0,b=0;
 int cnt=0;
 fscanf(fpi,"%d",&n);
 if(n==14) fprintf(fpo,"%d\n",2);
 else if(n==18) fprintf(fpo,"%d\n",9);
 else
 {n-=4;
  for(a=0;(ca=getlen(a))&&ca+4<=n;a++)//错误所在
  {for(b=0;b<=a;b++)
   {cb=getlen(b);
    if(ca+cb+getlen(a+b)==n)
     if(a!=b)
      cnt+=2;
     else
      cnt++;
   }
  }
  fprintf(fpo,"%d\n",cnt);
 }
 fclose(fpi);
 fclose(fpo);
 return(0);
}
```

当然，如果是初学者，不会编写上述的搜索程序，可以使用“交表”的方法，首先将结果为 0 的情况、样例数据输出，然后对较小的数据用手算的方法得到解的个数，并加入表中。采用这种方法，要保证手工枚举的准确性，要不重不漏，否则数错一个就前功尽弃。

【小结】本题是一道较为基础的搜索问题，也没有超时问题的困扰，不需要时间复杂度的优化，本以为多数选手可以轻松 AC，但由于这样那样的原因，总是在细节上出现问题，本题的得分率并不高。

9.6.3 传纸条

【问题描述】
从 m*n 的矩阵中，只能向下、右两个方向走，找到两条互不相交的路径，使得两条路径上的权值之和最大。
【题目类型】
双线程动态规划
【建议编程时间】
40 分钟。这里的编程时间包括调试时间。
【算法一】
【空间复杂度】
约 27M，全局变量完全可承受。50*50*50*50*sizeof(int)=2.5*10^7.
【解题分析】

1、 读入矩阵，注意行列。
2、 采用一个四维数组记录当前两条路走到的位置(i1,j1,i2,j2)时取得的最大值，初始化为 0，表示不可能到达。(0,0,0,0)为 1，最后减 1 输出。
3、 一个四重循环枚举两条路分别走到的位置。由于每个点均从上或左继承而来，故内部有四个 if，分别表示两个点从上上、上左、左上、左左继承来时，加上当前两个点所取得的最大值。例如，f[i][j][k][l] = max{f[i][j][k][l], f[i-1][j][k-1][l] + a[i][j] + a[k][l]}，表示两点均从上面位置走来。
4、 输出右下角处的最大值 f[m][n][m][n]，注意换行。

【编程注意】

1、 在数组边界处特殊处理，避免数组越界。
2、 若待继承的点最大值为零，则停止判断，不能从这里走来。
3、 显然，非矩阵右下角的汇合点，两个位置不能重合（否则路径相交），若重合则最大值为 0，不可达。

【源代码】用时 1.02s，100 分

```
#include<stdio.h>
int f[51][51][51][51];
int main()
{ int n,m,i,j,k,l,a[100][100]={0};
  FILE *fp,*fp1;
  fp=fopen("message.in","r");
  fp1=fopen("message.out","w");
  fscanf(fp,"%d%d",&m,&n);
  for(i=0;i<m;i++)
   for(j=0;j<n;j++)
    fscanf(fp,"%d",&a[i][j]);
  f[0][0][0][0]=1;
  for(i=0;i<m;i++)
   for(j=0;j<n;j++)
    for(k=0;k<m;k++)
     for(l=0;l<n;l++)
     { if(i==k&&j==l&&!(i==m-1&&j==n-1))
         continue;
       f[i][j][k][l]=0;
       if(i>0&&k>0&&f[i-1][j][k-1][l]!=0&&f[i-1][j][k-1][l]+a[i][j]+a[k][l]>f[i][j][k][l])
        f[i][j][k][l]=f[i-1][j][k-1][l]+a[i][j]+a[k][l];
       if(i>0&&l>0&&f[i-1][j][k][l-1]!=0&&f[i-1][j][k][l-1]+a[i][j]+a[k][l]>f[i][j][k][l])
        f[i][j][k][l]=f[i-1][j][k][l-1]+a[i][j]+a[k][l];
       if(j>0&&k>0&&f[i][j-1][k-1][l]!=0&&f[i][j-1][k-1][l]+a[i][j]+a[k][l]>f[i][j][k][l])
        f[i][j][k][l]=f[i][j-1][k-1][l]+a[i][j]+a[k][l];
       if(j>0&&l>0&&f[i][j-1][k][l-1]!=0&&f[i][j-1][k][l-1]+a[i][j]+a[k][l]>f[i][j][k][l])
        f[i][j][k][l]=f[i][j-1][k][l-1]+a[i][j]+a[k][l];
     }
  fprintf(fp1,"%d",f[m-1][n-1][m-1][n-1]-1);
  fclose(fp);
```

```
  fclose(fp1);
  return 0;
}
```

【算法二】

【解题分析】

考虑到算法一中存在“不可能”的路线，即“两头所用时间不同，不可能同时到达”，浪费了计算时间，我们采用减少一维的办法，将第四维由前三维确定，程序时间复杂度由 O(n^4)降为 O(n^3)，实测结果时间相差 10 倍，也证明了去除不合理的状态有很大作用。

由于 NOIP2008 的数据较弱，两个程序都可以满分。但如果数据规模扩大，无论从时间上还是空间上，算法一都无法承受。

事实上笔者正是注意到试题数据弱的特点，估计 O(n^4)的算法即可 AC，所以直接编写了算法一，而没有考虑优化。这就是本文中“用时间复杂度换编程复杂度”的思想，只要不超时，就尽可能往简单处想，往简单处写。虽然在练习中应当“精益求精”，这样的思想似乎不好，但在竞赛中这种“不求精致，只求得分”的策略还是非常实用的。

【源代码】[1] 用时 0.10s，100 分

```
#include <iostream>
using namespace std;
int m,n,ma[52][52],d[100][52][52];
int main(){
  freopen("message.in","r",stdin);
  freopen("message.out","w",stdout);
  cin>>m>>n;
  for (int i=1;i<=m;i++)
    for (int j=1;j<=n;j++)
      scanf("%d",&ma[i][j]);
  memset(d,0,sizeof(d));
  for (int i=3;i<m+n;i++){
    int l=(i-n)>1?(i-n):1,r=(i-1)<m?(i-1):m;
    for (int j=l;j<r;j++)
      for (int k=j+1;k<=r;k++){
     if (d[i-1][j-1][k-1]>d[i][j][k]) d[i][j][k]=d[i-1][j-1][k-1];
       if (d[i-1][j-1][k]>d[i][j][k]) d[i][j][k]=d[i-1][j-1][k];
       if (d[i-1][j][k-1]>d[i][j][k]) d[i][j][k]=d[i-1][j][k-1];
       if (d[i-1][j][k]>d[i][j][k]) d[i][j][k]=d[i-1][j][k];
       d[i][j][k]+=ma[j][i-j]+ma[k][i-k];
      }
  }
  printf("%d\n",d[m+n-1][m-1][m]);
  return 0;
}
```

【骗分攻略】

本题是一道经典的双线程动态规划试题，每位参加 NOIP 的选手应该都训练过。但由于

[1] 来源：石家庄 郭子晨

犯了多种多样的错误，此题的得分率仍然不高。

【错例 1】80 分。有两个测试点，程序输出结果比标准输出略大。观察测试数据，发现第 7 个数据左下角为 43，恰好多了 43；第 9 个数据左下角为 29，恰好多了 29. 于是我们怀疑，这个程序当路径通过左下角时，会出现重复问题。

至于错误原因，可能是<与<=写错，循环控制变量中左下角被计算了两次。

```
#include"stdio.h"
#define N 51
main()
{FILE *fpi,*fpo;
 fpi=fopen("message.in","r");
 fpo=fopen("message.out","w");
 int f[2][N][N];
 int data[N][N];
 int m,n;
 int i,j,k;
 fscanf(fpi,"%d%d",&m,&n);
 for(i=0;i<m;i++)
  for(j=0;j<n;j++)
   fscanf(fpi,"%d",&data[i][j]);
 f[0][0][0]=0;
 for(k=1;k<m+n;k++)
  for(j=1;j<=k&&j<m;j++)
   for(i=j-1;i>=0;i--)
   {f[k&1][i][j]=0;
    if(j<k&&f[!(k&1)][i][j]>f[k&1][i][j])
    {f[k&1][i][j]=f[!(k&1)][i][j];
     if(i&&f[!(k&1)][i-1][j]>f[k&1][i][j])
      f[k&1][i][j]=f[!(k&1)][i-1][j];
    }
    if(i&&f[!(k&1)][i-1][j-1]>f[k&1][i][j])
     f[k&1][i][j]=f[!(k&1)][i-1][j-1];
    if(j!=i+1&&f[!(k&1)][i][j-1]>f[k&1][i][j])
     f[k&1][i][j]=f[!(k&1)][i][j-1];
    f[k&1][i][j]+=(data[i][k-i]+data[j][k-j]);
   }
 fprintf(fpo,"%d\n",f[!(k&1)][m-2][m-1]);
 fclose(fpi);
 fclose(fpo);
 return(0);
}
```

【错例 2】50 分。这个程序与错例 3 类似，都是未考虑路径重叠的情况。例如最后一个测试点，结果几乎是正确答案的 2 倍，再观察测试数据，是两条形如下图的斜线

1100000

0110000
0011000
0001100
0000110
0000011

所有元素的和，恰好就是标准输出；而程序的输出显然是将每个经过的点算了两次，效果与“两次动态规划相加”相同，会导致路径交叉问题。

这道题目很多选手都是因为没有看清题意“两条路径不允许交叉”而丢分。

```
#include "stdio.h"
#include "stdlib.h"
#define N 110
#define M 64
int map[N][M];
int vis[N][M];
int n,m;

int input()
{
    int i,j;
    scanf("%d%d",&n,&m);
    for(i=1;i<=n;i++)
    {
        for(j=1;j<=m;j++)
        {
            scanf("%d",&map[i+j][j]);
            vis[i+j][j]=1;
        }
    }
}

int update(int *dat,int w)
{
    if((*dat)<w) (*dat)=w;
    return 0;
}

int work()
{
    int i,j,k;
    int len,total;
    int f[N][M][M]={0};

    f[3][1][2]=map[3][1]+map[3][2];
```

```
    for(k=3;k<n+m;k++)
    {
        for(i=1;i<=m;i++)
        {
            for(j=1;j<=m;j++)
            {
                if(vis[k][i]&&vis[k][j])
                {

    update(&f[k+1][i][j],f[k][i][j]+map[k+1][i]+map[k+1][j]);

    update(&f[k+1][i+1][j+1],f[k][i][j]+map[k+1][i+1]+map[k+1][j+1
]);

    update(&f[k+1][i][j+1],f[k][i][j]+map[k+1][i]+map[k+1][j+1]);
                    if(i+1!=j)

    update(&f[k+1][i+1][j],f[k][i][j]+map[k+1][i+1]+map[k+1][j]);
                }//根本没有考虑路径重叠、相交问题
            }
        }
    }

    printf("%d\n",f[m+n-1][m-1][m]);
}

int main()
{
    freopen("message.in","r",stdin);
    freopen("message.out","w",stdout);
    input();
    work();
    return 0;
}
```

【错例 3】0 分。这个算法是典型的一种错误。程序运行后，程序输出远大于实际结果，甚至大于所有元素的和。到后面四个测试点，则出现了崩溃现象。这证明程序没有经过调试，可能是由于考场上时间不够导致的。该考生 NOIP2008 中只得了 130 分，其中最简单的“笨小猴”一题只得了 10 分，本题又出现了如此严重的问题，证明编程基本功不扎实，出现了算法实现上的问题。

选手希望实现的算法是，以走过的步数 k 划分阶段，枚举阶段的两个端点所到达的位置，并从两个位置的上、左分别继承，是一个 O(n^3)的动态规划算法。

（1）程序在枚举继承方向时，采用了 for 的方法，由于循环语句的常数复杂度很大，浪费了很多时间。由于只有 4 个方向，每个方向中处理又很简单，完全可以使用

4 个 if 代替。但如果方向数较多，例如有 8 个，这样的方法是可取的。

（2）对重复问题的处理有误。如果两条线来自同一个节点，则不合题意，应当舍去。选手可能理解错题意，认为交叉的点只算一次，于是减去了重复的部分。这提醒我们一定要注意审题，不能想当然。

（3）更新动态规划数组时出现问题，max 的括号应当在表达式的最右端，表示如果更优则更新。如果像程序中这样，则成为了将所有可行路线的当前和进行累加，这也是输出结果非常大的原因。这是本文中“匹配错误”一类问题的典范。

```
#include<stdio.h>
#include<stdlib.h>
#define N 60
int a[N][N];
int f[N][N][N];
int max(int a,int b){return a>b?a:b;}
int main()
{
 FILE *fin,*fout;
 int n,m;
 int k;
 int i,j,p,q;
 int s1,s2,s3;
 int c,d,e,g;
 fin=fopen("message.in","r");
 fout=fopen ("message.out","w");
 fscanf(fin,"%d%d",&n,&m);
 for(i=1;i<=n;i++)
  for(j=1;j<=m;j++)
   fscanf(fin,"%d",&a[i][j]);
 for(k=1;k<=n+m-1;k++)
 {
    for(i=1;i<=k;i++)
      if(i<=n)
        for(p=1;p<=k;p++)
          if(p<=n)
          {
            j=k+1-i;
            q=k+1-p;
            s1=s2=s3=0;
            for(c=-1;c<=0;c++)//多重循环完全可以改成 if
              for(d=-1;d<=0;d++)
                if(c+d==-1)
                  for(e=-1;e<=0;e++)
                     for(g=-1;g<=0;g++)
                       if(e+g==-1)
                       {
```

```
                        s1=f[k-1][i+c][j+d];
                        s2=f[k-1][p+e][q+g];
                        s3=s1+s2;
                        if(i+c==p+e&&q+g==j+d)//若重复，应该舍弃
                         s3-=a[i+c][p+e];
                       }
           f[k][i][p]=max(f[k][i][p],s3)+a[i][j]+a[p][q];//括号位置
         }
 }
 fprintf(fout,"%d",f[n+m-1][n][m]);
 fclose(fin);
 fclose(fout);
 return 0;
}
```

【错误算法】

（1）通过普通的动态规划算法，计算最大值和次大值。这样会导致大量的重复路径。

（2）先用动态规划计算出最大值，将路径上的点删去（标记为不可达），再计算最大值。这是一个错误的贪心算法。

0 1 0

0 2 1

1 2 0

对于上述数据，这个算法会先将路径 0-1-2-2-0 选出，于是将两侧的 1 隔开，只能选择其中一个。而事实上，通过 0-1-2-1-0 和 0-0-1-2-0 可以选出所有的点。当数据较大时，这样的贪心算法几乎不可能正确。

（3）采用图论方法，找到图中权值和最大的不自交环。可能图中最大的环并不通过左上角和右下角，所以仍然是错误的。

值得欣慰的是，上述错误算法在 NOIP2008 中极少出现，这证明选手对于错误算法举出反例、尽快排除是有一定能力的。

【小结】本题并不困难，关键是细心，以及初始化、边界位置的特殊处理等。需要注意的是，目前的试题采用“直接输出样例数据”的方法很难得分。

9.6.4 双栈排序

【问题描述】

通过两个栈 S1，S2，将一个给定的输入序列升序排列。定义 a 操作将元素压入 S1 栈，b 操作从 S1 栈中取出栈顶元素，c 操作将元素压入 S2 栈，d 操作从 S2 栈中取出栈顶元素。求字典序最小的操作序列。

【建议编程时间】

贪心算法 40 分钟（包括调试），可得 30 分。

【骗分攻略】（贪心算法，30 分）

1、读入序列

2、若能进入 S1 栈，则执行 a 操作，进入 S1 栈

3、重复执行 b 操作，将 S1 栈中当前元素弹出，直到不可弹出为止

4、若能进入 S2 栈，则执行 c 操作，进入 S2 栈
5、重复执行 d 操作，将 S2 栈中当前元素弹出，直到不可弹出为止
6、若两栈均无法进入，失败退出
7、输出操作序列
【判断是否能进栈】
若当前元素小于栈顶元素，则进栈，栈元素个数自增；否则不能进栈。因为栈中必须保持降序，这样才能保证依次输出时升序。
【判断是否能出栈】
用全局变量表示当前取出的最大元素。若栈内元素个数不为零，且当前栈顶元素等于最大元素+1（因为元素是 1-n 依次取出的），则取出该元素，最大元素自增，栈元素个数自减。
【编程注意】
（1）不要将栈 S1、S2 的顺序颠倒
（2）不要将进栈、出栈的顺序颠倒
（3）不要将两栈都进完后再一起出栈
（4）进栈只能进一次，而出栈要将可能的全部出完，不要仅弹出栈顶元素
【源代码】用时 0.03s，30 分

```
#include<stdio.h>
main()
{                                                                    int
i,j,n,a[1005]={0},x[1005]={0},y[1005]={0},tx=0,ty=0,now=1,opt=0,m
in=10000000;
  char op[4005]={0};
  FILE *fp,*fp1;
  fp=fopen("twostack.in","r");
  fp1=fopen("twostack.out","w");
  fscanf(fp,"%d",&n);
  for(i=0;i<n;i++)
   fscanf(fp,"%d",&a[i]);
  for(i=0;i<n;i++)
  { if(a[i]<min)//进 S1 栈
    { x[tx++]=a[i];
      op[opt++]='a';
      min=a[i];
    }
    else //进 S2 栈
    { y[ty++]=a[i];
      op[opt++]='c';
    }
    while(1)
    { if(tx>0&&now==x[tx-1])//出 S1 栈
      { op[opt++]='b';
        tx--;
        now++;
        min=10000000;
```

```
        for(j=0;j<tx;j++)
         if(x[j]<min)
          min=x[j];
      }
      else if(ty>0&&now==y[ty-1])//出 S2 栈
      { op[opt++]='d';
        ty--;
        now++;
      }
      else break;
    }
  }
  if(now>n)
  { for(i=0;i<opt-1;i++)
     fprintf(fp1,"%c ",op[i]);
    fprintf(fp1,"%c",op[opt-1]);
    fclose(fp);
    fclose(fp1);
    return 0;
  }
  fprintf(fp1,"0");
  fclose(fp);
  fclose(fp1);
  return 0;
}
```

【正确解题分析】（转载，感谢提供解题方法的大牛）
这道题大概可以归结为如下题意:
有两个队列和两个栈,分别命名为队列 1(q1),队列 2(q2),栈 1(s1)和栈 2(s2).最初的时候,q2,s1 和 s2 都为空,而 q1 中有 n 个数(n<=1000),为 1~n 的某个排列.
现在支持如下四种操作:
a 操作,将 q1 的首元素提取出并加入 s1 的栈顶.
b 操作,将 s1 的栈顶元素弹出并加入 q1 的队列尾.
c 操作,将 q1 的首元素提取出并加入 s2 的栈顶.
d 操作,将 s2 的栈顶元素弹出并加入 q1 的队列尾.
请判断,是否可以经过一系列操作之后,使得 q2 中依次存储着 1,2,3,…,n.如果可以,求出字典序最小的一个操作序列.

这道题的错误做法很多,错误做法却能得满分的也很多,这里就不多说了.直接切入正题,就是即将介绍的这个基于二分图的算法.
注意到并没有说基于二分图匹配,因为这个算法和二分图匹配无关.这个算法只是用到了给一个图着色成二分图.

第一步需要解决的问题是,判断是否有解.

考虑对于任意两个数 q1[i]和 q1[j]来说,它们不能压入同一个栈中的充要条件是什么(注意没有必要使它们同时存在于同一个栈中,只是压入了同一个栈).实际上,这个条件 p 是:存在一个 k,使得 i<j<k 且 q1[k]<q1[i]<q1[j].

首先证明充分性,即如果满足条件 p,那么这两个数一定不能压入同一个栈.这个结论很显然,使用反证法可证.
假设这两个数压入了同一个栈,那么在压入 q1[k]的时候栈内情况如下:
…q1[i]…q1[j]…
因为 q1[k]比 q1[i]和 q1[j]都小,所以很显然,当 q1[k]没有被弹出的时候,另外两个数也都不能被弹出(否则 q2 中的数字顺序就不是 1,2,3,…,n 了).
而之后,无论其它的数字在什么时候被弹出,q1[j]总是会在 q1[i]之前弹出.而 q1[j]>q1[i],这显然是不正确的.

接下来证明必要性.也就是,如果两个数不可以压入同一个栈,那么它们一定满足条件 p.这里我们来证明它的逆否命题,也就是"如果不满足条件 p,那么这两个数一定可以压入同一个栈."
不满足条件 p 有两种情况:一种是对于任意 i<j<k 且 q1[i]<q1[j],q1[k]>q1[i];另一种是对于任意 i<j,q1[i]>q1[j].
第一种情况下,很显然,在 q1[k]被压入栈的时候,q1[i]已经被弹出栈.那么,q1[k]不会对 q1[j]产生任何影响(这里可能有点乱,因为看起来,当 q1[j]<q1[k]的时候,是会有影响的,但实际上,这还需要另一个数 r,满足 j<k<r 且 q1[r]<q1[j]<q1[k],也就是证明充分性的时候所说的情况…而事实上我们现在并不考虑这个 r,所以说 q1[k]对 q1[j]没有影响).
第二种情况下,我们可以发现这其实就是一个降序序列,所以所有数字都可以压入同一个栈.
这样,原命题的逆否命题得证,所以原命题得证.

此时,条件 p 为 q1[i]和 q1[j]不能压入同一个栈的充要条件也得证.

这样,我们对所有的数对(i,j)满足 1<=i<j<=n,检查是否存在 i<j<k 满足 p1[k]<p1[i]<p1[j].如果存在,那么在点 i 和点 j 之间连一条无向边,表示 p1[i]和 p1[j]不能压入同一个栈.此时想到了什么?那就是二分图~
二分图的两部分看作两个栈,因为二分图的同一部分内不会出现任何连边,也就相当于不能压入同一个栈的所有结点都分到了两个栈中.
此时我们只考虑检查是否有解,所以只要 O(n)检查出这个图是不是二分图,就可以得知是否有解.

此时,检查有解的问题已经解决.接下来的问题是,如何找到字典序最小的解.
实际上,可以发现,如果把二分图染成 1 和 2 两种颜色,那么结点染色为 1 对应当前结点被压入 s1,为 2 对应被压入 s2.为了字典序尽量小,我们希望让编号小的结点优先压入 s1.
又发现二分图的不同连通分量之间的染色是互不影响的,所以可以每次选取一个未染色的编号最小的结点,将它染色为 1 并从它开始 DFS 染色,直到所有结点都被染色为止.这样,我们就得到了每个结点应该压入哪个栈中.接下来要做的,只不过是模拟之后输出序列啦~

还有一点小问题,就是如果对于数对(i,j),都去枚举检查是否存在 k 使得 p1[k]<p1[i]<p1[j]的话,那么复杂度就升到了 O(n^3).解决方法就是,首先预处理出数组 b,b[i]表示从 p1[i]到 p1[n]中的

最小值.接下来,只需要枚举所有数对(i,j),检查 b[j+1]是否小于 p1[i]且 p1[i]是否小于 p1[j]就可以了.

【源代码】[1] 代码中的 a 数组对应文中的队列 p1.
已经过掉所有标准数据,以及 5 7 2 4 1 6 3 这组让很多贪心程序挂掉的数据.

```
#include <iostream>
using namespace std;
const int nn = 1002, mm = nn * 2, inf = 1000000000;
int n, tot, now;
int a[nn], b[nn], head[nn], color[nn];
int adj[mm], next[mm];
int stack[3][nn];
bool result;
void addEdge(int x, int y) //加边
{
    ++ tot;
    adj[tot] = y;
    next[tot] = head[x];
    head[x] = tot;
}
bool dfs(int i) //DFS 染色,检查图是否是二分图的经典算法
{
    int temp = head[i];
    while (temp) //邻接表,检查每一条边
    {
        if (! color[adj[temp]]) //如果与当前结点的结点还未染色
        {
            color[adj[temp]] = 3 - color[i]; //进行染色
            dfs(adj[temp]); //DFS
        }
        if (color[adj[temp]] == color[i]) return false;
            //如果两个相邻结点染色相同,说明此图不是二分图,返回无解
        temp = next[temp];
    }
    return true;
}
int main()
{
    freopen("twostack.in", "r", stdin);
    freopen("twostack.out", "w", stdout);
    //输入
    scanf("%d", &n);
    for (int i = 1; i <= n; ++ i) scanf("%d", &a[i]);
```

[1] 第四题上述程序转载自 OIBH 论坛：http://www.oibh.org/bbs/。

```
    //预处理 b 数组
    b[n + 1] = inf;
     for (int i = n; i >= 1; -- i) b[i] = min(b[i + 1], a[i]);
//"min" in STL
    //枚举数对(i,j)并加边
    tot = 0;
    for (int i = 1; i <= n; ++ i)
        for (int j = i + 1; j <= n; ++ j)
            if (b[j + 1] < a[i] && a[i] < a[j])
            {
                addEdge(i, j);
                addEdge(j, i);
            }
    //DFS 染色
    memset(color, 0, sizeof(color));
    result = true;
    for (int i = 1; i <= n; ++ i) //每次找当前未染色的编号最小的结点,并
染颜色 1
        if (! color[i]) //当前位置尚未被染色
        {
            color[i] = 1;
             if (! dfs(i)) //染色时出现矛盾,此时图不是一个二分图,即无法分配
到两个栈中
            {
                result = false; //记录无解
                break;
            }
        }
    if (! result) //无解
        printf("0");
    else //有解
    {
        //模拟求解
        now = 1;
        for (int i = 1; i <= n; ++ i)
        {
            //将当前数字压入对应的栈
            if (color[i] == 1)
                printf("a ");
            else
                printf("c ");
            stack[color[i]][0] ++;
             stack[color[i]][stack[color[i]][0]] = a[i]; //this will
work even if stack[1][0] = 0
```

```
        //循环检查,如果可以的话就从栈顶弹出元素
                    while  (stack[1][stack[1][0]]  ==  now  ||
stack[2][stack[2][0]] == now)
        {
            if (stack[1][stack[1][0]] == now)
            {
                printf("b ");
                stack[1][0] --;
            }
            else if (stack[2][stack[2][0]] == now)
            {
                printf("d ");
                stack[2][0] --;
            }
            now ++;
        }
      }
   }
```

【骗分攻略】本题的 30 分贪心算法即为“骗分”的妙计。但能紧紧抓住这宝贵的 30 分，不让它从手中溜走，也是“骗分”的技巧。

【错例】20 分。本程序将 c 和 a 的优先级颠倒，导致第二个测试点错误。

```
#include<stdio.h>
#include<stdlib.h>
#define N 2000

int st1[N],s1;
int st2[N],s2;
int a[N],b[N],n;
int ope[N*2],ops;
int main()
{
 FILE *fin,*fout;
 int i,j;
 int flag;
 fin=fopen("twostack.in","r");
 fout=fopen("twostack.out","w");
 fscanf(fin,"%d",&n);
 for(i=1;i<=n;i++)
 {
     fscanf(fin,"%d",&a[i]);
     b[i]=0;
 }
```

```
for(i=1;i<=n;i++)
 for(j=i+1;j<=n;j++)
 if(a[i]>a[j])
  b[i]++;
s1=s2=0;
st1[0]=st2[0]=0;
a[0]=2147483640;
for(i=1;i<=n;i++)
{
    flag=0;
    if(b[i]>1&&a[i]<=a[st2[s2]])
    {
     st2[++s2]=i;
     ope[ops++]=2;
     flag=1;
    }
    if(b[i]<2&&a[i]<=a[st1[s1]])
    {
         st1[++s1]=i;
         ope[ops++]=0;
         flag=1;
    }
    if(!flag&&a[i]<a[st2[s2]])
    {
     st2[++s2]=i;
     ope[ops++]=2;
     flag=1;
    }
    if(!flag)
    {
     fprintf(fout,"0");
     fclose(fin);
     fclose(fout);
     return 0;
    }
    while(a[st1[s1]]<=a[st2[s2]]&&b[st1[s1]]==0&&s1)
    {
            ope[ops++]=1;
            s1--;
            for(j=1;j<=s1;j++)
             b[st1[j]]--;
            for(j=1;j<=s2;j++)
             b[st2[j]]--;
    }
```

```
     while(a[st2[s2]]<=a[st1[s1]]&&b[st2[s2]]==0&&s2)
     {
             ope[ops++]=3;
             s2--;
             for(j=1;j<=s1;j++)
              b[st1[j]]--;
             for(j=1;j<=s2;j++)
              b[st2[j]]--;
     }
 }
 while(s1||s2)
 {
    if(s1&&a[st1[s1]]<=a[st2[s2]])
    {
         ope[ops++]=1;
         s1--;
    }
    if(a[st1[s1]]>a[st2[s2]]&&s2)
    {
         ope[ops++]=3;
         s2--;
    }
 }
 for(i=0;i<ops-1;i++)
  fprintf(fout,"%c ",ope[i]+'a');
 fprintf(fout,"%c",ope[ops-1]+'a');
 fclose(fin);
 fclose(fout);
 return 0;
}
```

【小结】本题是一道难度适合 NOI 的试题，在 NOIP 中出现，绝对可以起到压轴题的作用。事实上，全国范围内普遍存在的 330 分，也印证了此题的难度。但采用 30 分的骗分算法，就可以划分出 330 与 300 的天壤之别。

9.6.5 NOIP 小结

以上四道题全部做完，需要约 140 分钟，距离三小时的时限还有 40 分钟，可以编一些特殊的、极端的数据对程序进行测试。

这个阶段要做好源代码备份，除非发现重大问题，不要修改程序源代码，尤其在离终场 10 分钟以内时，因为此时头脑紧张，极易改错。

这样下来，330 分不是大问题，在河北省就进入省队了。但是，很多平时成绩不错的大牛考试时粗心大意，丢分现象十分严重。这次考试的题目很水，更考察选手的细心认真程度。真正的高手不仅会做难题，水题也能保证不出错。只有不出错，才能省队，因为第四题竞赛时几乎没人想出正确解法；只有保证错误不足半道，才能一等奖，因为河北省一等奖分数线

是 280 分。实际上，对于如此普及组难度的水题，即使实力一般，也可能拿到不错的成绩。

希望 NOIP2008 成功的同学总结经验，再接再厉，创造更好的成绩；NOIP 失利的同学不要气馁，总结教训，认真细致，下次再争取。

9.7 NOI2008

9.7.1 假面舞会

【问题描述】

一年一度的假面舞会又开始了，栋栋也兴致勃勃的参加了今年的舞会。

今年的面具都是主办方特别定制的。每个参加舞会的人都可以在入场时选择一个自己喜欢的面具。每个面具都有一个编号，主办方会把此编号告诉拿该面具的人。

为了使舞会更有神秘感，主办方把面具分为 k (k≥3)类，并使用特殊的技术将每个面具的编号标在了面具上，只有戴第 i 类面具的人才能看到戴第 i+1 类面具的人的编号，戴第 k 类面具的人能看到戴第 1 类面具的人的编号。

参加舞会的人并不知道有多少类面具，但是栋栋对此却特别好奇，他想自己算出有多少类面具，于是他开始在人群中收集信息。

栋栋收集的信息都是戴第几号面具的人看到了第几号面具的编号。如戴第 2 号面具的人看到了第 5 号面具的编号。栋栋自己也会看到一些编号，他也会根据自己的面具编号把信息补充进去。

由于并不是每个人都能记住自己所看到的全部编号，因此，栋栋收集的信息不能保证其完整性。现在请你计算，按照栋栋目前得到的信息，至多和至少有多少类面具。由于主办方已经声明了 k≥3，所以你必须将这条信息也考虑进去。

【输入格式】

输入文件 party.in 第一行包含两个整数 n, m，用一个空格分隔，n 表示主办方总共准备了多少个面具，m 表示栋栋收集了多少条信息。

接下来 m 行，每行为两个用空格分开的整数 a, b，表示戴第 a 号面具的人看到了第 b 号面具的编号。相同的数对 a, b 在输入文件中可能出现多次。

【输出格式】

输出文件 party.out 包含两个数，第一个数为最大可能的面具类数，第二个数为最小可能的面具类数。如果无法将所有的面具分为至少 3 类，使得这些信息都满足，则认为栋栋收集的信息有错误，输出两个-1。

“AMD”杯 浙江 绍兴

第 25 届全国信息学奥林匹克竞赛 第一试 假面舞会 party

第 3 页 共 7 页

【输入样例一】

```
6 5
1 2
2 3
3 4
4 1
```

3 5
【输出样例一】
4 4
【输入样例二】
3 3
1 2
2 1
2 3
【输出样例二】
-1 -1
【数据规模和约定】
50%的数据，满足 n ≤ 300, m ≤ 1000;
100%的数据，满足 n ≤ 100000, m ≤ 1000000。

9.7.2 设计路线

【问题描述】
Z 国坐落于遥远而又神奇的东方半岛上，在小 Z 的统治时代公路成为这里主要的交通手段。Z 国共有 n 座城市，一些城市之间由双向的公路所连接。非常神奇的是 Z 国的每个城市所处的经度都不相同，并且最多只和一个位于它东边的城市直接通过公路相连。Z 国的首都是 Z 国政治经济文化旅游的中心，每天都有成千上万的人从 Z 国的其他城市涌向首都。
为了使 Z 国的交通更加便利顺畅，小 Z 决定在 Z 国的公路系统中确定若干条规划路线，将其中的公路全部改建为铁路。
我们定义每条规划路线为一个长度大于 1 的城市序列，每个城市在该序列中最多出现一次，序列中相邻的城市之间由公路直接相连(待改建为铁路)。并且，每个城市最多只能出现在一条规划路线中，也就是说，任意两条规划路线不能有公共部分。
当然在一般情况下是不可能将所有的公路修建为铁路的，因此从有些城市出发去往首都依然需要通过乘坐长途汽车，而长途汽车只往返于公路连接的相邻的城市之间，因此从某个城市出发可能需要不断地换乘长途汽车和火车才能到达首都。
我们定义一个城市的“不便利值”为从它出发到首都需要乘坐的长途汽车的次数，而 Z 国的交通系统的“不便利值”为所有城市的不便利值的最大值，很明显首都的“不便利值”为 0。小 Z 想知道如何确定规划路线修建铁路使得 Z 国的交通系统的“不便利值”最小，以及有多少种不同的规划路线的选择方案使得“不便利值”达到最小。当然方案总数可能非常大，小 Z 只关心这个天文数字 mod Q 后的值。
注意：规划路线 1-2-3 和规划路线 3-2-1 是等价的，即将一条规划路线翻转依然认为是等价的。两个方案不同当且仅当其中一个方案中存在一条规划路线不属于另一个方案。
【输入格式】
输入文件 design.in 第一行包含三个正整数 N、M、Q，其中 N 表示城市个数，

M 表示公路总数，N 个城市从 1~N 编号，其中编号为 1 的是首都。Q 表示上文提到的设计路线的方法总数的模数。接下来 M 行，每行两个不同的正数 ai、bi (1≤ ai , bi ≤ N)表示有一条公路连接城市 ai 和城市 bi。输入数据保证一条公路只出现一次。

【输出格式】

输出文件 design.out 应包含两行。第一行为一个整数，表示最小的“不便利值”。 第二行为一个整数，表示使“不便利值”达到最小时不同的设计路线的方法总数 mod Q 的值。

如果某个城市无法到达首都，则输出两行-1。

【输入样例】

5 4 100
1 2
4 5
1 3
4 1

【输出样例】

1
10

【样例说明】

以下样例中是 10 种设计路线的方法：

(1) 4-5
(2) 1-4-5
(3) 4-5, 1-2
(4) 4-5, 1-3
(5) 4-5, 2-1-3
(6) 2-1-4-5
(7) 3-1-4-5
(8) 1-4
(9) 2-1-4
(10) 3-1-4

【数据规模和约定】

对于 20%的数据，满足 N,M ≤ 10。
对于 50%的数据，满足 N,M ≤ 200。
对于 60%的数据，满足 N,M ≤ 5000。
对于 100%的数据，满足 1 ≤ N,M ≤ 100000，1 ≤ Q ≤ 120000000。

【评分方式】

每个测试点单独评分。对于每个测试点，第一行错则该测试点得零分，否则若第二行错则该测试点得到 40%的分数。如果两问都答对，该测试点得到 100%的分数。

9.7.3 志愿者招募

【问题描述】

申奥成功后，布布经过不懈努力，终于成为奥组委下属公司人力资源部门的主管。布布刚上任就遇到了一个难题：为即将启动的奥运新项目招募一批短期志愿者。经过估算，这个项目需要 N 天才能完成，其中第 i 天至少需要 Ai 个人。布布通过了解得知，一共有 M 类志愿者可以招募。其中第 i 类可以从第 Si 天工作到第 Ti 天，招募费用是每人 Ci 元。新官上任三把火，为了出色地完成自己的工作，布布希望用尽量少的费用招募足够的志愿者，但这并不是他的特长！于是布布找到了你，希望你帮他设计一种最优的招募方案。

【输入格式】

输入文件 employee.in 的第一行包含两个整数 N, M，表示完成项目的天数和可以招募的志愿者的种类。

接下来的一行中包含 N 个非负整数，表示每天至少需要的志愿者人数。

接下来的 M 行中每行包含三个整数 Si, Ti, Ci，含义如上文所述。为了方便起见，我们可以认为每类志愿者的数量都是无限多的。

【输出格式】

输入文件 employee.out 中仅包含一个整数，表示你所设计的最优方案的总费用。

【输入样例】

3 3
2 3 4
1 2 2
2 3 5
3 3 2

【输出样例】

14

【样例说明】

招募 3 名第一类志愿者和 4 名第三类志愿者。

【数据规模和约定】

30%的数据中，1 ≤ N, M ≤ 10，1 ≤ Ai ≤ 10；

100%的数据中，1 ≤ N ≤ 1000，1 ≤ M ≤ 10000，题目中其他所涉及的数据均不超过 231-1。

9.7.4 奥运物流

【问题描述】

2008 北京奥运会即将开幕，举国上下都在为这一盛事做好准备。为了高效率、成功地举办奥运会，对物流系统进行规划是必不可少的。

物流系统由若干物流基站组成，以 1…N 进行编号。每个物流基站 i 都有且仅有一个后继基站 Si，而可以有多个前驱基站。基站 i 中需要继续运输的物资都将被运往后继基站 Si，显然一个物流基站的后继基站不能是其本身。编号为 1 的物流基站称为控制基站，从任何物流基站都可将物资运往控制基站。注意控制基站也有后继基站，以便在需要时进行物资的流通。在物流系统中，高可靠性与低成本是主要设计目。对于基站 i，我们定义其“可靠性” R(i) 如下：

设物流基站 i 有 w 个前驱基站 1 2 , , w P P　　P ，即这些基站以 i 为后继基站，

则基

站 i 的可靠性 R(i)满足下式：

1

( ) ( )

w

i j

j

Ri C k RP

=

= + Σ

其中 Ci 和 k 都是常实数且恒为正，且有 k 小于 1。

整个系统的可靠性与控制基站的可靠性正相关，我们的目标是通过修改物流系统，即更改某些基站的后继基站，使得控制基站的可靠性 R(1)尽量大。但由于经费限制，最多只能修改 m 个基站的后继基站，并且，控制基站的后继基站不可被修改。因而我们所面临的问题就是，如何修改不超过 m 个基站的后继，使得控制基站的可靠性 R(1)最大化。

【输入格式】

输入文件 trans.in 第一行包含两个整数与一个实数，N, m, k。其中 N 表示基站数目，m 表示最多可修改的后继基站数目，k 分别为可靠性定义中的常数。

第二行包含 N 个整数，分别是 S1, S2…SN，即每一个基站的后继基站编号。

第三行包含 N 个正实数，分别是 C1, C2…CN，为可靠性定义中的常数。

【输出格式】

输出文件 trans.out 仅包含一个实数，为可得到的最大 R(1)。精确到小数点两位。

【输入样例】

4 1 0.5

2 3 1 3

10.0 10.0 10.0 10.0

【输出样例】

30.00

【样例说明】

原有物流系统如左图所示，4 个物流基站的可靠性依次为 22.8571，21.4286，25.7143，10。

最优方案为将 2 号基站的后继基站改为 1 号，如右图所示。 此时 4 个基站的可靠性依次为 30，25，15，10。

【数据规模和约定】

本题的数据，具有如下分布：

测试数据编号 N M

1 ≤ 6 ≤ 6

2 ≤ 12 ≤ 12

3 ≤ 60 0

4 ≤ 60 1

5 ≤ 60 N-2

6~10 ≤ 60 ≤ 60
对于所有的数据，满足 m ≤ N ≤ 60，Ci ≤ 106，0.3 ≤ k < 1，请使用双精度实数，无需考虑由此带来的误差。

9.7.5 糖果雨

【问题描述】
有一个美丽的童话：在天空的尽头有一个"糖果国"，这里大到摩天大厦，小到小花小草都是用糖果建造而成的。更加神奇的是，天空中飘满了五颜六色的糖果云，很快糖果雨密密麻麻从天而落，红色的是草莓糖，黄色的是柠檬糖，绿色的是薄荷糖，黑色的是巧克力糖……这时糖果国的小朋友们便会拿出大大小小的口袋来接天空中落下的糖果，拿回去与朋友们一起分享。
对糖果情有独钟的小 Z 憧憬着能够来到这样一个童话的国度。所谓日有所思，夜有所梦，这天晚上小 Z 梦见自己来到了"糖果国"。他惊喜地发现，任何时候天空中所有的云朵颜色都不相同，不同颜色的云朵在不断地落下相应颜色的糖果。更加有趣的是所有的云朵都在做着匀速往返运动，不妨想象天空是有边界的，而所有的云朵恰好在两个边界之间做着往返运动。每一个单位时间云朵向左或向右运动一个单位，当云朵的左界碰到天空的左界，它会改变方向向右运动；当云朵完全移出了天空的右界，它会改变方向向左运动。
我们不妨把天空想象为一个平面直角坐标系，而云朵则抽象为线段(线段可能退化为点)：
如上图，不妨设天空的左界为 0，右界为 len。图中共有 5 片云朵，其中标号为 1 的云朵恰好改变方向向右运动，标号为 2 的云朵恰好改变方向向左运动。忽略云朵的纵坐标，它们在运动过程中不会相互影响。
小 Z 发现天空中会不断出现一些云朵(某个时刻从某个初始位置开始朝某个方向运动)，而有的云朵运动到一定时刻就会从天空中消失，而在运动的过程中糖果在不断地下落。小 Z 决定拿很多口袋来接糖果，口袋容量是无限的，但袋口大小却是有限的。例如在时刻 T 小 Z 拿一个横坐标范围为[L, R]的口袋来接糖果，如果[L, R]存在一个位置 x，该位置有某种颜色的糖果落下，则认为该口袋可接到此种颜色的糖果。极端情况下，袋口区间可能是一个点，譬如[0,0]、[1,1]，但仍然可以接到相应位置的糖果。通常可以接到的糖果总数会很大，因而小 Z 想知道每一次(即拿出口袋的一瞬间)他的口袋可以接到多少种不同颜色的糖果。糖果下落的时间忽略不计。
【输入格式】
输入文件 candy.in 的第一行有两个正整数 n, len，分别表示事件总数以及天空的“边界”。
接下来 n 行每行描述一个事件，所有的事件按照输入顺序依次发生。每行的第一个数 k（k = 1，2，3）分别表示事件的类型，分别对应三种事件：插入事件，询问事件以及删除事件。输入格式如下：
事件类型 输入格式 说明
插入事件
(天空中出现了一片云朵)
1 Ti Ci Li Ri Di 时刻 Ti，天空中出现了一片坐标范围为[Li, Ri]，颜色为 Ci 的

云朵，初始的时候云朵运动方向为向左(Di = -1)或向右(Di = 1)。

满足 0 ≤ Li ≤ Ri ≤ len，Di = -1 或 1。

数据保证任何时刻空中不会出现两片颜色相同的云朵。

询问事件

(询问一个口袋可以接到多少种不同颜色的糖果)

2 Ti Li Ri 时刻 Ti，小 Z 用一个坐标范围为[Li, Ri]的大口袋去接糖果，

询问可以接到多少种不同的糖果。满足 0 ≤ Li ≤ Ri ≤ len。

删除事件

(天空中一片云朵消失了)3 Ti Ci 时刻 Ti，颜色为 Ci 的云朵从天空消失中。数据保证当前天空中一定存在一片颜色为 Ci 的云朵。

【输出格式】

对于每一个询问事件，输出文件 candy.out 中应包含相应的一行，为该次询问的答案，即口袋可以接到多少种不同的糖果。

【输入样例】

```
10 10
1 0 10 1 3 -1
2 1 0 0
2 11 0 10
2 11 0 9
1 11 13 4 7 1
2 13 9 9
2 13 10 10
3 100 13
3 1999999999 10
1 2000000000 10 0 1 1
```

【输出样例】

```
1
1
0
2
1
```

【样例说明】

共 10 个事件，包括 3 个插入事件，5 个询问事件以及 2 个删除事件。

时刻 0，天空中出现一片颜色为 10 的云朵，初始位置为[1, 3]，方向向左。

时刻 1，范围为[0, 0]的口袋可以接到颜色为 10 的糖果(云朵位置为[0, 2])。

时刻 11，范围为[0,10]的口袋可以接到颜色为 10 的糖果(云朵位置为[10, 12])。

时刻 11，范围为[0, 9]的口袋不能接到颜色为 10 的糖果(云朵位置为[10, 12])。

时刻 11，天空中出现一片颜色为 13 的云朵，初始位置为[4, 7]，方向向右。

时刻 13，范围为[9, 9]的口袋可以接到颜色为 10(云朵的位置为[8, 10])和颜色为 13(云朵的位置为[6, 9])两种不同的糖果。

时刻 13，范围为[10, 10]的口袋仅仅可以接到颜色为 10 的一种糖果(云朵的位置为[8, 10])，而不可以接到颜色为 13 的糖果(云朵的位置为[6, 9])，。

时刻 100，颜色为 13 的云朵从天空中消失。

时刻 1999999999，颜色为 10 的云朵从天空中消失。

时刻 2000000000，天空中又出现一片颜色为 10 的云朵，初始位置为[0, 1]，方向向右。

【数据规模和约定】

对于所有的数据，0 ≤ Ti ≤ 2000000000，1 ≤ Ci ≤ 1000000。

数据保证{Ti}为非递减序列即 T1 ≤ T2 ≤ … ≤ Tn-1 ≤ Tn。

对于所有的插入事件，令 Pi = Ri - Li，即 Pi 表示每片云朵的长度。

数据编号 n len Pi 数据编号 n len Pi

1 20 10 ≤ len 6 150000 1000 ≤ 3

2 200 100 ≤ len 7 200000 1000 ≤ 3

3 2000 1000 ≤ len 8 100000 1000 ≤ len

4 100000 10 ≤ len 9 150000 1000 ≤ len

5 100000 100 ≤ 2 10 200000 1000 ≤ len

9.7.6 赛程安排

【问题描述】

随着奥运的来临，同学们对体育的热情日益高涨。在 NOI2008 来临之际，学校正在策划组织一场乒乓球赛。小 Z 作为一名狂热的乒乓球爱好者，这正是他大展身手的好机会，于是他摩拳擦掌，积极报名参赛。

本次乒乓球赛采取淘汰赛制，获胜者晋级。恰好有 n (n 是 2 的整数次幂，不妨设 $n = 2^k$)个同学报名参加，因此第一轮后就会有 2k-1 个同学惨遭淘汰，另外 2k-1 个同学晋级下一轮；第二轮后有 2k-2 名同学晋级下一轮，… 依次类推，直到 k 轮后决出冠亚军：具体的，每个人都有一个 1~n 的初始编号，其中小 Z 编号为 1，所有同学的编号都不同，他们将被分配到 n 个位置中，然后按照类似下图的赛程进行比赛：

位置 1 位置 2 位置 3 位置 4 位置 5 位置 6 位置 7 位置 8

位置 1 和位置 2 的胜者

位置 3 和位置 4 的胜者

位置 5 和位置 6 的胜者

位置 7 和位置 8 的胜者

位置 1234 中的胜者 位置 5678 的胜者冠军

n=8 时比赛的赛程表

为了吸引更多的同学参加比赛，本次比赛的奖金非常丰厚。在第 i 轮被淘汰的选手将得到奖金 ai 元，而冠军将获得最高奖金 ak+1 元。显然奖金应满足 a1 < a2 < … < ak+1.

在正式比赛前的热身赛中，小 Z 连连败北。经过认真分析之后，他发现主要的失败原因不是他的球技问题，而是赢他的这几个同学在球风上刚好对他构成相克的关系，所以一经交手，他自然败阵。小 Z 思索：如果在正式比赛中能够避开这几位同学，该有多好啊！

假设已知选手两两之间交手的胜率，即选手 A 战胜选手 B 的概率为 PA,B (保证 PA,B + PB,A=1)。于是小 Z 希望能够通过确定比赛的对阵形势（重新给每个选手安排位置），从而能够使得他获得尽可能多的奖金。你能帮助小 Z 安排一个方案，使得他这场比赛期望获得的奖金最高么？

【输入格式】
这是一道提交答案型试题，所有的输入文件 match*.in 已在相应目录下。
输入文件 match*.in 第一行包含一个正整数 n，表示参赛的总人数，数据保证存在非负整数 k，满足 2k = n。
接下来 n 行，每行有 n 个 0 到 1 间的实数 Pi,j,, 表示编号为 i 的选手战胜编号为 j 的选手的概率，每个实数精确到小数点后两位。特别注意 Pi,i = 0.00。
接下来 k+1 行，每行一个整数分别为晋级各轮不同的奖金，第 i 行的数为 ai。
【输出格式】
输出文件 match*.out 包括 n 行，第 i 行的数表示位于第 i 个位置的同学的编号，要求小 Z 的编号一定位于第 1 个位置。
【输入样例】
4
0.00 0.70 0.60 0.80
0.30 0.00 0.60 0.40
0.40 0.40 0.00 0.70
0.20 0.60 0.30 0.00
1
2
3
【输出样例】
1
4
2
3
【样例说明】
第一轮比赛过后，编号为 1 的选手(小 Z)晋级的概率为 80%，编号为 2 的选手晋级的概率为 60%，编号为 3 的选手晋级的概率为 40%，编号为 4 的选手晋级的概率为 20%。
第二轮（决赛），编号为 1 的选手（小 Z）前两轮均获胜的概率为 80% * (60%*70% + 40%*60%) = 52.8%，因此，小 Z 在第一轮失败的概率 P1=1-0.8=0.2，第一轮胜出但第二轮败北的概率 P2=0.8-0.528=0.272，获得冠军的概率 P3=0.528。从而，期望奖金为 0.2*1 + (0.8-0.528)*2 + 0.528*3 = 2.328。
【如何测试你的输出】
我们提供 match_check 这个工具来测试你的输出文件是否可接受。 使用这个工具的测试方法是在终端中使用命令：
./match_check 测试数据编号
例如：./match_check 10 表示测试你的 match10.out 是否合法。
调用这个程序后，match_check 将根据你得到的输出文件给出测试的结果，其中包括：
非法退出： 未知错误；
Format error： 输出文件格式错误；
Not a permutation： 输出文件不是一个 1~n 的排列；
OK.Your answer is xxx：输出文件可以被接受，xxx 为对应的期望奖金。
【评分方法】

每个测试点单独评分。

对于每一个测试点，如果你的输出文件不合法，如文件格式错误、输出解不符合要求等，该测试点得 0 分。否则如果你的输出的期望奖金为 your_ans，参考期望奖金为 our_ans，我们还设有一个用于评分的参数 d，你在该测试点中的得分如下：

如果 your_ans > our_ans，得 12 分。

如果 your_ans < our_ans*d，得 1 分。

否则得分为：

_ _ * *8 2

_ _ *

your ans our ans d

our ans our ans d

−

\+ −

【提示】

“数学期望”

数学期望是随机变量最基本的数字特征之一。它反映随机变量平均取值的大小，又称期望或均值。它是简单算术平均的一种推广。例如某城市有 10 万个家庭，没有孩子的家庭有 1000 个，有一个孩子的家庭有 9 万个，有两个孩子的家庭有 6000 个，有 3 个孩子的家庭有 3000 个，则该城市中任一个家庭中孩子的数目是一个随机变量，它可取值 0，1，2，3，其中取 0 的概率为 0.01，取 1 的概率为 0.9，取 2 的概率为 0.06，取 3 的概率为 0.03，它的数学期望为 0×0.01＋1×0.9＋2×0.06＋3×0.03 等于 1.11，即此城市一个家庭平均有小孩 1.11 个。

本题中期望值的计算：假设小 Z 在第一轮被打败的概率为 P1，第一轮胜利且在第二轮被打败的概率为 P2，前两轮胜利且在第三轮被打败的概率为 P3……，那么小 Z 的期望奖金为：

P1 * a1 + P2 * a2 + …+ Pk+1 * ak+1

【特别提示】

请妥善保存输入文件*.in 和你的输出*.out，及时备份，以免误删。

9.8 WC2009

9.8.1 最短路问题

【问题描述】

一个 6×n 的方格，初始每个格子有一个非负权值。有如下两种操作形式：

改变一个格子的权值（改变以后仍然非负）；

求两个格子之间的最短路的权值。

【注解与任务】

任意格子 P 的坐标(xP, yP)满足 1 ≤ xP ≤ 6，1 ≤ yP ≤ n。格子 P 和 Q 的曼哈顿距离定义为|xP - xQ|+|yP - yQ|。一个有序方格序列(p1, p2, ..., pn)，若满足任意 pi 和

pi+1 的曼哈顿距离为 1，则称该序列为一条从 p1 到 pn 的路径，其权值为

d(p1)+d(p2)+...+d(pn)，其中 d(P)表示格子 P 的权值。两个格子 P 和 Q 之间的最短路定义为从 P 到 Q 权值最小的路径。

【输入文件】

第一行一个整数 n。接下来 6 行，每行 n 个整数，第 i+1 行第 j 个整数表示初始格子(i, j)的权值。接下来是一个整数 Q，后面的 Q 行，每行描述一个操作。

输入的操作有以下两种形式：

操作 1："1 x y c"(不含双引号)。表示将格子(x, y)的权值改成 c (1 ≤ x ≤ 6, 1 ≤ y ≤ n, 0 ≤ c ≤ 10000) 。

操作 2："2 x1 y1 x2 y2"(不含双引号)。表示询问格子(x1, y1)和格子(x2, y2)之间的最短路的权值。(1 ≤ x1, x2 ≤ 6, 1 ≤ y1, y2 ≤ n)

【输出文件】

对于每个操作 2，按照它在输入中出现的顺序，依次输出一行一个整数表示求得的最短路权值。

【样例输入】

```
5
0 0 1 0 0
0 1 0 1 0
0 2 0 1 0
0 1 1 1 0
0 0 0 0 0
1 1 1 1 1
5
2 1 2 1 4
1 1 1 10000
2 1 2 1 4
1 2 3 10000
2 1 2 3 3
```

【样例输出】

```
0
1
2
```

【数据约定】

测试数据规模如下表所示

数据编号	n	Q	数据编号	n	Q
1	10	20	6	10000	30000
2	100	200	7	35000	30000
3	1000	2000	8	50000	50000
4	10000	10000	9	100000	60000
5	10000	10000	10	100000	100000

【特别说明】

本题测试时将使用-O2 优化。

9.8.2 语音识别

【题目背景】

与机器进行语音交流，让机器明白你说什么，这是人们长期以来梦寐以求的事情。语音识别技术就是让机器通过识别和理解过程把语音信号转变为相应的文本或命令的高技术。

——百度百科

现在，我们需要你解决的是一个简化版的语音识别问题：

麦克风所录入的信息可以被认为是一些独立的信号，每个信号都按照其电平值被表示为一个非负整数，这些信号组成的有序序列就是麦克风输入的信号序列。

语音信号实例

信号序列可以用一个非负整数序列来描述，形如 A = {a1, a2, ... an}，信号序列 A 的子序列 A'是指 A 中的一段连续信号 A' = {ai, ai+1, ... aj-1, aj}。

实际情况中，麦克风录入的信号序列往往混有为数不多的噪声，为了在语音识别中能够处理噪声带来的问题，需要引入近似匹配的概念：

设 A、B 是两个信号序列，A 对于 B 近似匹配，是指从 A 中删除若干个信号之后，所得的信号序列恰好等于 B。我们把从 A 中删除信号的个数称为 A 对于 B 近似匹配的差别程度。值得注意的是，为了使得识别有意义，只有那些差别程度不大的近似识别才是有意义的。例如，从信号序列{1, 2, 0}中删除一个信号 2 就可以得到信号序列{1, 0}，因此{1, 2, 0}对于{1, 0}近似匹配，差别程度为 1；同样{1, 2, 0}对于{0}也是近似匹配的，其差别程度为 2，但是如果限定近似匹配的差别程度不能超过 1，那么{1, 2, 0}对于{0}的近似匹配就将被忽略。特别的，如果两个信号序列完全一致，那么这两个信号序列的匹配可以被认为是差别程度为 0 的近似匹配。

研究人员已经对很多日常使用的字进行了预处理，得到了和每个字相对应的信号序列，这些字的信号序列所组成的集合称为字典。令 A‘是信号序列 A 的一个子序列，如果 A‘对于信号序列 B 近似匹配，那么 A‘就是 B 在 A 中的一次近似出现。例如：

对于信号序列 A = {a1, a2, a3, a4, a5, a6} = {3, 3, 3, 1, 2, 0}，那么 A1 = {a1, a2, a3},

A2 = {a4, a5}, A3 = {a6}, A4 = {a1, a2}都是 A 的子序列。

考虑字典 Σ = {a = {1}, b = {1, 2}, c = {0}, d = {3, 3}}，且近似匹配的差别程度

上限为 1，那么 A1、A4 都是 d 在 A 中的一次近似出现，A2 是 a 或者 b 的近似出现，A3 是 c 的近似出现。

进一步可知字典中的每一个字都可以在信号序列中被检测出来。

a 被检测出 3 次，{a4},{a3, a4},{a4, a5}；

b 被检测出 3 次，{a4, a5}, {a3, a4, a5}, {a4, a5, a6}；

c 被检测出 2 次，{a6}, {a5, a6}；

d 被检测出 4 次，{a1, a2}, {a2, a3}, {a1, a2, a3}, {a2, a3, a4}。

为了对麦克风输入的信号序列作出尽量好的识别，一个最直观的想法是，从输入信号序列中能够识别出来的字越多越好。具体的说，如果能够从信号序列 A

= {a1, a2, ... an}中找出一组子序列 D1, D2, ... Ds，对于字典Σ，满足：

1) 对于任意 i、j，Di 和 Dj 没有相交部分；也就是说，如果 Di = {ap, ap+1, ... aq}，Dj = {au, au+1, ... av}，那么一定有区间[p, q]和区间[u, v]没有交集，即 p > v 或者 q < u。

2) 任意 Di 都可以在字典中找到一个字(设为 wordi）的信号序列，使得 Di 对于字 wordi 的信号序列近似匹配；

那么 D1->word1, D2->word2, ... Ds->words 被称为 A 的一种识别方案，把这些近似匹配按照出现先后排列起来就可得到识别结果，s 就是这种识别方案的长度。

最直观的想法就是希望能够找出一组最长的识别方案。

例如，考虑刚才的例子：{A1->d, A3->c}, {A4->d, A2->a, A3->c}，{A4->d, A2->b, A3->c}都是可行的识别方案，对应的识别结果分别是："dc", "dac", "dbc"。由于不存在长度超过 3 的识别方案，所以最长的识别方案的长度为 3，有两种识别结果"dac"和"dbc"。

【任务】

给定信号序列 A，字典Σ，和能够允许的近似匹配的差别程度的上限 K。要求计算：

字典中有多少字在 A 中近似出现，总的近似出现的次数是多少？

A 的最长的识别方案的长度是多少，在识别方案长度最长的前提下，能够识别出的本质不同的序列有多少个？

注意：如果两个识别方案分别对应识别结果 words = {word1, word2, ... wordp}和 words' = {word'1, word'2, ... word'q}，如果 words 和 words'不完全相同，则两个识别方案本质不同。

【输入文件】

第一行有三个整数 N、M、K，分别表示待测信号序列的长度、字典的大小和近似匹配差别程度的上限。

第二行有 N 个用空格隔开的非负整数，描述待测信号序列 A。

接下来 M 行每行描述字典的一个不同的字，对于其中每一行：

先输入一个非负整数 L，表示这个字对应信号序列的长度，接下来 L 个整数给出这个字所对应的信号序列。相邻的整数之间用空格隔开。

注意：不同的字可能对应相同或者类似的信号序列。

【输出文件】

输出文件一共包含两行：

第一行包含两个整数，分别表示字典中可以被检测出来的字的个数，以及近似出现的总次数；

第二行包含两个整数，分别表示最长识别方案的长度，以及能够识别出的本质不同的最长识别方案数（输出的方案数为你所得到的答案对 1 000 003 取模后的结果）。

注意：每行要求包含且仅包含两个整数，用一个空格隔开，不允许多余的空格和换行，不允许输出不完整。以下的输出均不合法（其中下划线表示空格，\n 表示回车）：

多空格：

100__100\n

100_100\n
多回车：
100_100\n
100_100\n
\n
多行首（尾）空格
_100_100\n
100_100_\n
输出不完整
100_100\n
100\n
【样例输入】
6 4 1
3 3 3 1 2 0
1 1
2 1 2
1 0
2 3 3
【样例输出】
4 12
3 2
【样例说明】
样例为在题目背景中给出的例子。
【评分标准】
每个测试点单独评分。
如果输出文件不存在、不合法则该测试点得 0 分，否则，你的得分为以下 4 项得分之和：
能够识别的字的个数正确(2 分)
近似出现的总次数正确(4 分)
最长识别方案的长度正确(2 分)
最长识别方案的个数正确（2 分）
【数据约定】
本题共有 10 个测试点，每个测试点 10 分，每个测试点单独评分
Test# N ≤ M ≤ K ≤ L ≤ Test# N ≤ M ≤ K ≤ L ≤
1 6 5 1 10 6 100000 20 2 15000
2 1000 10 5 100 7 100000 20 20 100000
3 50000 15 5 50000 8 100000 20 20 50000
4 10000 20 20 15000 9 100000 20 20 100000
5 50000 20 20 30000 10 100000 20 20 100000
其中 L 表示字典中所有字的信号总长度。

9.8.3 优化设计

【问题描述】

生活中总有些事情不能两全，所谓“鱼和熊掌”说的便是这个意思。建筑师小X 在最近一个设计中正面临着这样的问题而头痛不已：出于居住环境的考虑，楼间距离应该较大，而为了节约土地资源却应减小楼间距离；朝南的房间应该选择起居室还是卧室；等等……。

为了最优化设计方案，小X 决定将问题进行建模，将参数的选择抽象为布尔变量，将各种要求抽象为布尔表达式。目标即为选择参数的设置使得被满足的要求尽量多，对应的，在模型中选择布尔变量的一个赋值使得被满足的布尔表达式尽量多。现在请你帮助小X 设置方案中的参数，使得设计方案尽可能优化。

下面将形式化地描述小X 的模型。模型包含n 个布尔变量x1, x2, …, xn 和m 个布尔表达式E1, E2, …, Em。布尔变量的一个赋值为从{x1, x2, …, xn}到{True, False}的一个映射。一个布尔表达式E 被满足指该表达式值为真(True).

布尔表达式包含与(&)、或(|)、非(~)三种类型运算，可归纳定义如下：

基础：

1) 单个布尔变量xi 组成一个布尔表达式E。E 被满足当且仅当xi 取值为真。

归纳：

1) 若E 是合法布尔表达式，则E’=(E)也是布尔表达式。E’被满足当且仅当E 被满足。

2) 若E 是合法布尔表达式，则E’= ~E 也是布尔表达式。E’被满足当且仅当E 不被满足(即取值为假)。

3) 若E1、E2 是合法布尔表达式，则E’=E1&E2 也是布尔表达式。E’被满足当且仅当E1、E2 均被满足。

4) 若E1、E2 是合法布尔表达式，则E’=E1|E2 也是布尔表达式。E’被满足当且仅当E1 被满足或E2 被满足。

所有合法布尔表达式均由以上规则产生。一些合法布尔表达式的例子为：x1&~x2，(x1&~x2)|x3。

布尔表达式的运算符优先级由高到低分别为非、与、或。举例说明，表达式x1&~x2|~x3 等价于(x1&(~x2))|(~x3)，即非运算优先计算，与次之。

在本题中，目标即为找到一组赋值，使得给定的m 个布尔表达式中被满足的表达式尽可能多。

【输入文件】

这是一道提交答案型试题，所有的输入文件opt*.in 都已在目录下。

对于每个输入文件，文件第一行包含一个整数n，表示布尔变量的个数。第二行为一个整数m，为布尔表达式的个数。接下来m 行，每行为一个布尔表达式，表达式格式同上述定义，并请参看输入文件（表达式中不含空格）。

【输出文件】

对于每一个输入文件，你需要在对应目录下给出对应的输出文件（主文件名不变，扩展名为.out）。

输出文件包含n 行，每行为0 或1。第i 行表示对第i 个布尔变量的赋值(1表示赋值为True, 0 表示赋值为False).

【样例输入】

```
3
2
x1&x2&x3
```

~x1|x2

【样例输出】

1

1

1

【样例说明】

在本题中输出任意的 n 行 0 或 1 均为合法输出。

在本样例中若输出

1

1

1

则两个表达式均被满足；而若输出为

0

1

1

则仅有第二个表达式被满足；而若输出为

1

0

0

则没有表达式被满足。

【评分标准】

每个测试点单独评分。

对于每一个测试点，如果你给出的输出文件不合法，如文件格式错误、输出解不符合要求等，该测试点得 0 分。

否则设你输出的赋值能够使得 k 个表达式被满足，对于不同的测试点，我们还设有 10 个评分相关的常数 c1 ≤c2 ≤ c3 ≤ c4 ≤ c5 ≤ c6 ≤ c7 ≤ c8 ≤ c9 ≤ c10，你在该测试点中的得分取决于下列约定：

如果 k < c1，得 0 分。

如果 k ≥ c1，得 1 分。

如果 k ≥ c2，得 2 分。

如果 k ≥ c3，得 3 分。

如果 k ≥ c4，得 4 分。

如果 k ≥ c5，得 5 分。

如果 k ≥ c6，得 6 分。

如果 k ≥ c7，得 7 分。

如果 k ≥ c8，得 8 分。

如果 k ≥ c9，得 9 分。

如果 k ≥ c10，得 10 分。

如果满足多个条件，取得分最大者为最终得分。

【如何测试你的输出】

你可以使用 checker 程序检查你的输出，格式为：

./checker TestNo

其中 TestNo 为测试点编号。例如你已经得到了数据 5 的输出 opt5.out，可以

使用命令./checker 5 来测试你的输出是否合法。

【特别提示】

请妥善保存输入文件*.in 和你的输出*.out，及时备份，以免误删。

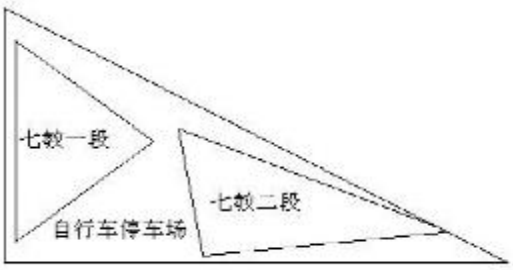
9.9 CTSC2008

9.9.1 三角形的教学楼

triangle

【问题描述】

清华大学拥有很多教学楼，但由于学生的课业负担较重，所以上自习的人很多，特别是期末考试前，教学楼的自习位置仍然是捉襟见肘。为迎接 2008 年北京奥运，改善同学们上自习的条件，学校决定新建第七教学楼。



第七教学楼由若干个小教学楼（下图中称为段）和一个大的自行车停车场组成，每个教学楼的俯视图都是一个三角形。而第七教学楼的自行车停车场则是包含所有小教学楼所构成的一个大三角形。如下图所示：

由于学校的总面积有限，所以七教的占地面积要尽量节省，即七教所有教学楼的面积以及与自行车停车场组成的大三角形的总面积要尽量小。

已知七教每个小教学楼俯视图的大小（以坐标形式给出），现在要求你设计出七教的停车场范围（停车场面积可以为零），你可以平移、旋转与翻转这些小教学楼的位置，但不能改变它们的大小和形状，使得停车场俯视图是一个大三角形，并且每一个小教学楼都在停车场中（注意：答案中小教学楼俯视图的边可以重合，但是任意两个小教学楼俯视图的公共面积必须为 0）。

你要确定每个教学楼的俯视图坐标与停车场俯视图坐标。使得它们构成的总面积和最小。

【输入文件】

这是一道提交答案型试题，所有的输入文件 triangle*.in 都已在目录下。

输入文件第一行为一个整数 N，表示小教学楼的个数。接下来 N 行，每行有用空格隔开的六个实数。第 i 行代表第 i 个小教学楼俯视图坐标的三个点的初始坐标 x0, y0, x1, y1, x2, y2。

（注意：初始小教学楼的俯视图可能会有教学楼相交的情形）

【输出文件】

输出文件 triangle*.out 应包含 N+1 行，第一行有六个实数 x0, y0, x1, y1, x2, y2，中间用空格隔开，代表你设计的自行车停车场的俯视图坐标，可以是无限大的平面上的任意位置。

接下来 N 行，每一行有六个实数 x0,y0,x1,y1,x2,y2，中间用空格隔开。第 i 行的六个实数代表第 i 个小教学楼最终被摆放的位置。输出文件中小教学楼的输出顺序必须与输入文件中小教学楼的输入顺序相同。

【样例输入】

2
-1 0 0 0 0 1
1 0 0 0 0 1

【样例输出】

0 1 -1 0 1 0
-1 0 0 0 0 1
1 0 0 0 0 1

【样例解释】

输入的两个教学楼如下图所示：

教学楼一段(-1 0 0 0 0 1)：

教学楼二段（1 0 0 0 0 1）

可以拼接到一个大的三角形停车场中（粗线显示），停车场的面积为零。

第七教学楼的最终坐标（0 1 -1 0 1 0）

【评分标准】

每个测试点单独评分。

对于每一个测试点，如果你给出的输出文件不合法，如文件格式错误、输出的解不符合要求等，该测试点得 0 分。否则设你的输出答案的三角形面积为 YourAns。

如果你的答案优于或等于我们的答案，你将得到 10 分，否则你的答案的得分按照如下公式计算：

【如何测试你的输出】

你可以使用 checker 程序检查你的输出，格式为：

./checker CaseNo

其中 CaseNo 为测试点的编号。例如你已经得到了数据 5 的输出 triangle5.out，可以使用命令./checker 5 来测试你的输出是否合法。checker 还会同时输出你的方案所占用的总面积。

对于如下错误，我们将会返回：

错误信息	错误原因
Wrong Test Number	输入的文件编号不合法
No output file	没有找到你的输出文件
Output format error	你的输出文件格式错误
The Ith triangle in your answer doesn't fit the input	你的输出文件中第 I 个小教学楼与输入文件中第 I 个小教学楼不相符，当且仅当它们对应的三条边有至少一条边的长度相差在 10-7 以上。
The Ith triangle in your answer doesn't in your big triangle	你的输出文件中第 I 个小教学楼不在你设计的停车场范围中。
The Ith triangle have common area with the Jth triangle	你的输出文件中第 I 个三角形与第 J 个小教学楼重叠。
Correct! The area of your	你的答案是正确的，答案面积是 xxx

answer is xxx	

（注：如果你的输出文件没有按照标准格式输出，可能还会引发其他问题）

【特别提示】

请妥善保存输入文件*.in 和你的输出*.out，及时备份，以免误删。

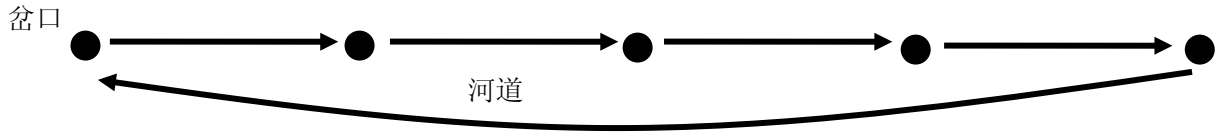
9.9.2 祭祀

river

【问题描述】

在遥远的东方，有一个神秘的民族，自称 Y 族。他们世代居住在水面上，奉龙王为神。每逢重大庆典，Y 族都会在水面上举办盛大的祭祀活动。

我们可以把 Y 族居住地水系看成一个由岔口和河道组成的网络。每条河道连接着两个岔口，并且水在河道内按照一个固定的方向流动。显然，水系中不会有环流（下图描述一个环流的例子）。



环流

由于人数众多的原因，Y 族的祭祀活动会在多个岔口上同时举行。出于对龙王的尊重，这些祭祀地点的选择必须非常慎重。准确地说，Y 族人认为，如果水流可以从一个祭祀点流到另外一个祭祀点，那么祭祀就会失去它神圣的意义。

族长希望在保持祭祀神圣性的基础上，选择尽可能多的祭祀的地点。

【输入文件】

输入文件 river.in 中第一行包含两个用空格隔开的整数 N、M，分别表示岔口和河道的数目，岔口从 1 到 N 编号。

接下来 M 行，每行包含两个用空格隔开的整数 u、v，描述一条连接岔口 u 和岔口 v 的河道，水流方向为自 u 向 v。

【输出文件】

第一行包含一个整数 K，表示最多能选取的祭祀点的个数。

接下来一行输出一种可行的选取方案。对于每个岔口依次输出一个整数，如果在该岔口设置了祭祀点，那么输出一个 1，否则输出一个 0。应确保你输出的 1 的个数最多，且中间没有空格。

接下来一行输出，在选择最多祭祀点的前提下，每个岔口是否能够设置祭祀点。对于每个岔口依次输出一个整数，如果在该岔口能够设置祭祀点，那么输出一个 1，否则输出一个 0。

注意：多余的空格和换行可能会导致你的答案被判断为错误答案。

【样例输入】

```
4 4
1 2
3 4
```

3 2
4 2

【样例输出】

2
1010
1011

【样例说明】

在样例给出的水系中，不存在一种方法能够选择三个或者三个以上的祭祀点。包含两个祭祀点的测试点的方案有两种：选择岔口 1 与岔口 3（如样例输出第二行），选择岔口 1 与岔口 4。

水流可以从任意岔口流至岔口 2。如果在岔口 2 建立祭祀点，那么任意其他岔口都不能建立祭祀点，但是在最优的一种祭祀点的选取方案中我们可以建立两个祭祀点，所以岔口 2 不能建立祭祀点。对于其他岔口，至少存在一个最优方案选择该岔口为祭祀点，所以输出为 1011。

【评分规则】

对于每个测试点：

如果你仅输出了正确的被选取的祭祀点个数，那么你将得到该测试点 30%的分数；

如果你仅输出了正确的被选取的祭祀点个数与一个可行的方案，那么你将得到该测试点 60%的分数；

如果你的输出完全正确，那么你将得到该测试点 100%的分数；

【数据规模】

N ≤ 100

M ≤ 1 000

9.9.3 奥运抽奖

volunteer

【问题描述】

距 2008 年北京奥运会开幕还有 90 天时，CTSC 准备为志愿者们举行一次抽奖活动。作为志愿者的一员，你对这次抽奖活动自然是万分期待。

CTSC 委员会介绍了抽奖活动的规则。设总共有 p 个参加抽奖的志愿者，开始时每一个志愿者领取一个 0 到 p-1 的 号码。任意两个志愿者领取的号码不同。屏幕的正中央是五福娃的头像，他们不停的眨眼欢迎大家。开始抽奖时，工作人员按下屏幕旁边的按钮，等待屏幕上的画面静止下来。这时，福娃们都停止眨眼了。当然，画面静止时，有的福娃的眼睛可能是睁开的，有的是闭上的。如果所有福娃的眼睛都闭上了，工作人员需要重新按一 下按钮。这样，直到至少有一个福娃的眼睛是睁开的。接着，工作人员开始观察有哪些福娃的眼睛是睁开的。

工作人员对五个福娃都标了号。贝贝、晶晶、欢欢、迎迎、妮妮的标号分别是 2、3、4、5、6（工作人员认为 0 和 1 都不是好数字）。定义幸运数字如下：

1、如果一个福娃的眼睛是睁开的，那么他（她）对应的标号就是幸运数字；

2、如果数字 11 和 12(可能相等)都是幸运数字，那么他们的乘积 也是幸运数字；

3、其他的数字都不是幸运数字。

用 L 表示所有数字的集合，例如，如果贝贝、晶晶的眼睛是睁开的，欢欢、迎迎、妮妮的眼睛是闭上的，则 L={2,3,4,6,8,9,12,…}。令 l(x)表示第 x 大的幸运数字。例如，

上面的例子中，l(1)=2，l(4)=6 等等。

接着，工作人员开始随机产生两个数，小的数是 a，大的数字是 b。定义集合 Ta, b 为：

$$T_{a,b}=\{d \mid d\in L, l(a) \mid d, d \mid l(b)\}$$　（其中 表示 x 整除 y）

定义一个自然数的有限子集的特征值 f 如下：

1、空集的特征值为 0，即 ；

2、对于非空集合 S，令 d 为 S 中的最小元素，则

$f(S)=d+f(S\setminus d)+q\times d\times f(S\setminus d)$。

其中， 表示把 S 删除元素 d 后的集合，q 是一个给定的非负整数。

在 a 和 b 产生以后，中奖的志愿者就确定了，他的号码是 $f(T_{a,b})$ 除以 p 的余数。工作人员会产生多次 a，b，这样就能形成多个中奖者。但是，抽奖现场的程序需要很长的时间才能算出中奖的志愿者。出于对中奖结果的热切期待，你便想要重新写一下计算程序，于是，你的目光移向了前面的键盘……。

【输入文件】

输入的第一行给出用空格隔开的 5 个数，每个数不是 0 就是 1，分别表示贝贝、晶晶、欢欢、迎迎、妮妮的眼睛是否睁开。0 对应眼睛闭上，1 对应眼睛睁开。5 个数不可能都是 0。

第二行给出了用空格隔开的两个数，p 和 q。 其中 p 表示参加抽奖的志愿者的人数，q 如前所述，用来计算集合的特征值。

第三行给出了数 n，表示抽取的 a 和 b 的次数。

接下来的 n 行，每一行有两个数 a、b，中间用空格隔开，表示一次抽奖产生的两个数。

【输出文件】

输出共 n 行，每一行一个整数，表示一次抽奖中中奖者的号码。顺序与输入的 n 对 a、b 一一对应。当然，一个人可能中奖多次。

【样例输入】

```
1 0 0 1 0
10001 2
3
1 10
2 12
4 15
```

【样例输出】

```
3265
5816
0
```

【样例说明】

贝贝和迎迎的眼睛是睁开的，因此，前面 15 个幸运数字是 2、4、5、8、10、16、20、25、32、40、50、64、80、100、125。l(1) = 2，l(10) = 40。既是 2 的倍数，又是 40 的约数的幸运数字有 2、4、8、10、20、40。所以 $T_{1,10}=\{2,4,8,10,20,40\}$。$T_{1,10}$ 的特征值的计算过程为：

$$f(\Phi)=0$$
$$f(\{40\})=40+0+2\times40\times0=40$$
$$f(\{20,40\})=20+40+2\times20\times40=1660$$
$$f(\{10,20,40\})=10+1660+2\times10\times1660=34870$$
$$f(\{8,10,20,40\})=8+34870+2\times8\times34870=592798$$
$$f(\{4,8,10,20,40\})=4+592798+2\times4\times592798=5335186$$
$$f(\{2,4,8,10,20,40\})=2+5335186+2\times2\times5335186=26675932$$

所以中奖者的号码就是 26675932 除以 10001 的余数——3265。

类似的，T2,12 = {4,8,16,32,64}，它的特征值是 21167932，除以 10001 的余数是 5816。而 T4,15 = Φ。

【数据规模】

有 20%的数据点满足：

1 ≤ a ≤ b ≤1000，n ≤ 2000；

所有的数据点均满足：

1 ≤ a ≤ b ≤ 100000，n ≤ 100000，p，q ≤ 2 × 109。

60%的数据点满足：p 是素数。

9.9.4 图腾

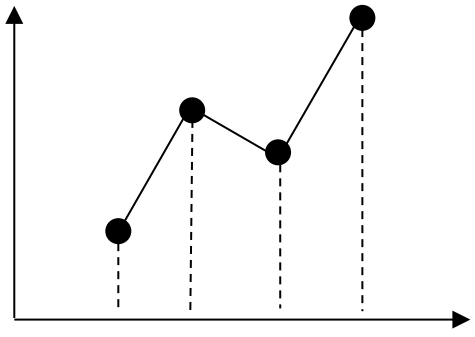
totem

【问题描述】

在完成了古越州圆盘密码的研究之后，考古学家小布来到了南美大陆的西部。相传很久以前在这片土地上生活着两个部落，一个部落崇拜闪电，另一个部落崇拜高山，他们分别用闪电和山峰的形状作为各自部落的图腾。

小布的团队在山洞里发现了一幅巨大的壁画，壁画上被标记出了 N 个点，经测量发现这 N 个点的水平位置和竖直位置是两两不同的。小布认为这幅壁画所包含的信息仅与这 N 个点的相对位置有关，因此不妨设坐标分别为(1, y1)，(2, y2)，...，(n, yn)，其中 y1~yn 是 1~N 的一个排列。

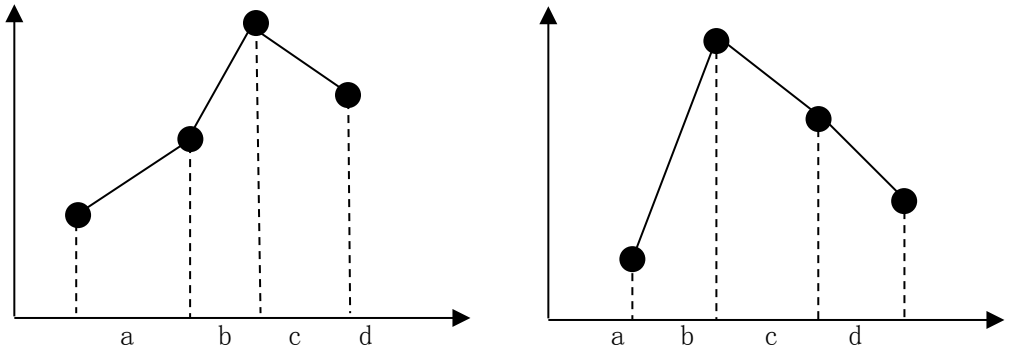
小布的团队打算研究在这幅壁画中包含着多少个图腾，其中闪电图腾的定义图示如下（图腾的形式只与 4 个纵坐标值的相对大小排列顺序有关）：



a　　b　　c　　d

即$1\le a<b<c<d\le N, y_a<y_c<y_b<y_d$

崇拜高山的部落有两个氏族，因而山峰图腾有如下两种形式，左边为 A 类，右边为 B 类（同样，图腾的形式也都只与 4 个纵坐标值的大小排列顺序有关）：



即$1 \le a < b < c < d \le N, y_a < y_b < y_d < y_c$

即$1 \le a < b < c < d \le N, y_a < y_d < y_c < y_b$

小布的团队希望知道，这 N 个点中两个部落图腾数目的差值。因此在本题中，你需要帮助小布的团队编写一个程序，计算闪电图腾数目减去山峰图腾数目的值，由于该值可能绝对值较大，本题中只需输出该值对 16777216 的余数（注意余数必为正值，例如-1 对 16777216 的余数为 16777215）。

【输入文件】

输入文件 totem.in 中第一行包含一个整数 N，为点的数目。

接下来一行包含 N 个整数，分别为 y1，y2，…，yn。保证 y1，y2，…，yn 是 1~N 的一个排列。

【输出文件】

输出文件 totem.out 仅包含一个数，表示闪电图腾数目与山峰图腾数目的差值对 16777216 的余数。

【样例输入一】

5

1 5 3 2 4

【样例输出一】

0

【样例输入二】

4

1 2 4 3

【样例输出二】

16777215

【样例说明】

样例一中共有 1 个闪电图腾（1324）和 1 个 B 类山峰图腾（1532）。

样例二中仅有一个 A 类山峰图腾（1243），故差值为-1，答案为 16777215。

【数据规模】

对于 10%的数据，N ≤ 600；

对于 40%的数据，N ≤ 5000；

对于 100%的数据，N ≤ 200000。

9.9.5 网络管理

network

【问题描述】

M 公司是一个非常庞大的跨国公司，在许多国家都设有它的下属分支机构或部门。为了让分布在世界各地的 N 个部门之间协同工作，公司搭建了一个连接整个公司的通信网络。该网络的结构由 N 个路由器和 N-1 条高速光缆组成。每个部门都有一个专属的路由器，部门局域网内的所有机器都联向这个路由器，然后再通过这个通信子网与其他部门进行通信联络。该网络结构保证网络中的任意两个路由器之间都存在一条直接或间接路径以进行通信。

高速光缆的数据传输速度非常快，以至于利用光缆传输的延迟时间可以忽略。但是由于路由器老化，在这些路由器上进行数据交换会带来很大的延迟。而两个路由器之间的通信延迟时间则与这两个路由器通信路径上所有路由器中最大的交换延迟时间有关。作为 M 公司网络部门的一名实习员工，现在要求你编写一个简单的程序来监视公司的网络状况。该程序能够随时更新网络状况的变化信息（路由器数据交换延迟时间的变化），并且根据询问给出两个路由器通信路径上延迟第 k 大的路由器的延迟时间。

【任务】

你的程序从输入文件中读入 N 个路由器和 N-1 条光缆的连接信息，每个路由器初始的数据交换延迟时间 Ti，以及 Q 条询问（或状态改变）的信息。并依次处理这 Q 条询问信息，它们可能是：

1. 由于更新了设备，或者设备出现新的故障，使得某个路由器的数据交换延迟时间发生了变化。
2. 查询某两个路由器 a 和 b 之间的路径上延迟第 k 大的路由器的延迟时间。

【输入文件】

输入文件 network.in 中的第一行为两个整数 N 和 Q，分别表示路由器总数和询问的总数。

第二行有 N 个整数，第 i 个数表示编号为 i 的路由器初始的数据延迟时间 Ti。

紧接着 N-1 行，每行包含两个整数 x 和 y。表示有一条光缆连接路由器 x 和路由器 y。

紧接着是 Q 行，每行三个整数 k、a、b。如果 k=0，则表示路由器 a 的状态发生了变化，它的数据交换延迟时间由 Ta 变为 b。如果 k>0，则表示询问 a 到 b 的路径上所经过的所有路由器（包括 a 和 b）中延迟第 k 大的路由器的延迟时间。注意 a 可以等于 b，此时路径上只有一个路由器。

【输出文件】

输出文件为 network.out。对于每一个第二种询问（k>0），输出一行。包含一个整数为相应的延迟时间。如果路径上的路由器不足 k 个，则输出信息“invalid request!”（全部小写不包含引号，两个单词之间有一个空格）。

【样例输入】

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

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

【样例输出】

```
3
2
2
invalid request!
```

【数据规模】

100% 测试数据满足 ， 。任意一个路由器在任何时刻都满足延迟时间小于 108。对于所有询问满足 。

40% 测试数据满足所有询问中 。即路由器的延迟时间不会发生变化。

10% 测试数据满足 。

9.9.6 唯美村落

village

【问题描述】

在经过无数科学家夜以继日的探索后，公元 6101 年，人类终于找到了地球外一个适合居住的星球 W。这个星球与地球有着几乎一样的自然环境。

正当无数地球人争着向 W 星球移民时，人类外星移民计划部的秘书栋栋向部长提议，移民应该有计划地分批进行，而不能一次移民过多。部长听取了栋栋的建议，决定先将 n 个人移民到 W 星球上最适合人居住的平原，以组成一个新的村落。经过认真的勘察，人类外星移民计划部在 W 星球上选择了一块 1v×1v 的正方形平原区域(其中 v 是一个长度单位)，所有人居住的位置必须在这块区域内(可以在边界上)。为了描述区域内一点的位置，人们以区域的一个顶点为原点，其邻边为正方向建立了一个二维的坐标轴，如下图所示：

为了使这个新村落能够和[illegible]令栋栋对村落进行规划：根据移民中 n 个人的关系来规定他们居住的位置。[illegible]两个人，他们之间存在两种关系：或者互相认识，或者互相不认识。通过长时间的研究，栋栋找出了一个能让村落最好发展的条件：不认识的人之间的距离越大越好，而认识的人之间的距离既不能太大，也不能太小。

为了量化这两个条件，栋栋定义了几个概念：

陌生距离 ：所有两两不认识的人之间的平均距离，用公式（1）表示：

$$\bar{d}=\frac{1}{C_1}\sum_{i,j\text{不认识}} D_{i,j} \qquad (1)$$

其中 C1 为不认识的人的对数，而 Di,j 表示 i 与 j 之间的距离，设 i 的坐标为(xi,yi)，j 的坐标为(xj,yj)，则

$$D_{i,j}=\sqrt{(x_i-x_j)^2+(y_i-y_j)^2} \qquad (2)$$

友好距离 ：所有两两认识的人之间的平均距离，用公式（3）表示：

$$\bar{a}=\frac{1}{C_2}\sum_{i,j\text{认识}} D_{i,j} \qquad (3)$$

其中 C2 为认识的人的对数。

友好变化系数 v：认识人之间的距离的标准差，用公式（4）表示：

$$v=\sqrt{\frac{1}{C_2}\sum_{i,j\text{认识}} (D_{i,j}-\bar{a})^2} \qquad (4)$$

栋栋给出了一个规划的目标：

使$(1.5-v)^4\bar{d}$尽可能大！

【任务】

这是一个提交答案的试题。

在试题目录下有 10 个输入数据 village1.in~village10.in，描述要移民的 n 个人，你需要产生对应的 village1.out~village10.out，告诉栋栋这 n 个人应该移到平原的哪个位置。

【输入文件】

从 village1.in~village10.in 中读入数据，数据的第一行为一个整数 n，表示部长准备移民多少人到星球 W。

接下来的一行为一个整数 m，表示这 n 个人中，有多少对两两认识的人。

接下来 m 行，每行两个整数 a, b，表示 a 与 b 互相认识。

输入数据保证同一对互相认识的人只被描述一次。即如果出现了 a, b，就不会出现 b, a 或再次出现 a, b。

【输出文件】

输出到 village1.out~village10.out 中。输出应该包含 n 行，每行两个实数，第 i 行的两个实数 xi 和 yi 表示第 i 个人居住的坐标。你可以根据自己的需要保留多位有效数字，但不能使用科学计数法。

【样例输入】

```
4
4
1 2
2 3
3 4
```

```
4 1
```

【样例输出】

```
0 0
0 1.0
1 1.000
0.00000 0.0
```

【如何评分】

对于每个测试点，我们都有一个预设的期望 e，如果你的解合法(即所有人的都在平原内)，你的得分为：

$$\left\lfloor \min\{\max\{(\frac{(1.5-v)^4\bar{d}-0.5e}{0.5e},0.1\},1\}\times \text{该测试点的分值} \right\rfloor$$

其中 **min**{*a*, *b*}表示 *a* 与 *b* 的较小值，**max**{*a*, *b*}表示 *a* 与 *b* 的较大值，$\lfloor x \rfloor$表示不大于 *x* 的最大整数。v、$\bar{d}$如前所述。

如果你的解非法，则此测试点得分为 0。

【提示】

在你提供的解中，可以有多于一个人居住在同一个位置。

9.10 IPSC2008

IPSC 是 Internet Problem Solving Contest[1]，国际网络程序设计竞赛。这项竞赛的试题全部是提交答案类试题，多数试题编写代码并不是很麻烦，也不需要多么高级的算法，且具有一定的趣味性，适合 NOIP、NOI 水平的选手练习。很多问题需要我们仔细用头脑考虑，设计全新的算法，或者“人机结合”处理问题，而不是盲目去编写程序。这样的试题，利于我们“骗分”的训练。

9.10.1 Army Strength

【问题描述】有两支军队，A、B 军队分别有 N、M 个士兵，每次将两支军队中士兵的战斗力最低的一个士兵杀死。如果战斗力最低的士兵在两只军队中都有，则杀死 B 军队中的那个士兵。如果一支军队的士兵被全部杀死，则另一支军队获胜。问最后哪只军队能够取得胜利？

【分析】

这个问题是十分简单的。既然战斗力最强的士兵永远不会被打败，哪只军队拥有最强的士兵，就可以无敌了。但要注意两支军队都有战斗力最强的士兵的情况，这时根据规则，杀死 B 军队中的士兵，A 获胜。

【源代码】[2]

```
#include<stdio.h>
int main()
{  int m,n,i,t,maxa=0,maxb=0;
    scanf("%d%d",&m,&n);
    for(i=0;i<m;i++)
    {   scanf("%d",&t);
        if(t>maxa)
            maxa=t;
    }
    for(i=0;i<n;i++)
    {  scanf("%d",&t);
        if(t>maxb)
            maxb=t;
    }
```

[1] 官方网址：http://ipsc.ksp.sk/ 读者可以到此网站查看问题的英文原版、输入输出和官方题解。

[2] 本节中的源代码，均忽略了“多组测试数据”的要求。

```
    if(maxa>=maxb)
        printf("Godzilla\n");
    else
        printf("MechaGodzilla\n");
}
```

9.10.2 Breaking In

【问题描述】

给定一个有向图，一个节点的重要度就是从该节点可以到达的节点的总数。要求找出最重要的节点（如果有多个，则全部输出）。

【分析】

首先考虑最朴素的算法：使用 Floyed 算法计算每个节点是否可达，简单的 count 一下就可以得出每个节点可达的数目。复杂度 $O(n^3)$.

但注意到，这道题中较大的一组测试数据规模可以达到点数 4583，边数 30159，显然 Floyed 是会超时的。但又注意到边数比点数大不了很多，所以这个图是稀疏图。

既然要求的是每个点可达节点的总数，就可以使用深度优先搜索的方法，对每个点搜索出所有可达节点即可。复杂度 O(nm)，由于是提交答案试题，可以 AC。

【使用 Floyed 算法的源代码】

```
#include<stdio.h>
int x[1000][1000]={0},num[1000]={0};
int main()
{ int n,m,i,j,k,a,b,max=0;
  scanf("%d%d",&n,&m);
  for(i=0;i<m;i++)
  { scanf("%d%d",&a,&b);
    x[b][a]=1;
  }
  for(k=1;k<=n;k++)
   for(i=1;i<=n;i++)
    for(j=1;j<=n;j++)
     if(x[i][k]&&x[k][j])
      x[i][j]=1;
  for(i=1;i<=n;i++)
  { for(j=1;j<=n;j++)
     if(x[i][j])
      num[i]++;
    if(num[i]>max)
     max=num[i];
  }
  for(i=1;i<=n;i++)
   if(num[i]==max)
    printf("%d ",i);
```

```
  system("pause");
}
```

【使用深度优先搜索的源代码】[1]

```
#include <stdio.h>
#include <string.h>

#define MAXN 10000
#define MAXM 100000

// Graph
int start[MAXN+1];
typedef struct { int from, to; } ARC;
ARC arc[MAXM];
int otherend[MAXM];
int cnt[MAXN];

// Data for
char visited[MAXN];

void visit(int p)
{
    int i;
    if (visited[p]) return;
    visited[p] = 1;
    for (i = start[p]; i < start[p+1]; i++)
        visit(otherend[i]);
}

int main()
{
    int t, n, m, i, j, k;

    scanf("%d", &t);
    while(t--)
    {
        scanf("%d %d", &n, &m);
        memset(start, 0, sizeof(start));
        for (i=0; i<m; i++)
        {
            scanf("%d %d", &arc[i].to, &arc[i].from);
            arc[i].to--;
            arc[i].from--;
            start[arc[i].from]++;
```

[1] 来源：IPSC 官方题解源代码

```
        }
        for (i=0; i<n; i++) start[i+1] += start[i];
        for (i=0; i<m; i++)
            otherend[--start[arc[i].from]] = arc[i].to;

// Run the searches
        for (i=0; i<n; i++)
        {
            memset(visited, 0, sizeof(visited));
            visit(i);
            cnt[i] = 0;
            for (j=0; j<n; j++)
                if (visited[j]) cnt[i]++;
        }

// Count
        m = 0;
        for (i=0; i<n; i++)
            if (cnt[i] > m) m = cnt[i];
// Print the best targets
        for (i=0; i<n; i++)
            if (cnt[i] == m)
                printf("%d ", i+1);
        printf("\n");
    }
    return 0;
}
```

9.10.3 Comparison Mysteries

【问题描述】

要求补全一段 Java 代码，利用浮点数的误差，实现(1)x+(-x)!=0　(2)x=y, y=z, x!=z 两个特点。补全的内容是变量类型和初值。

【代码 1】

```
import java.util.*;
public class C1 {
 public static void main(String args[]) {
 Scanner sc = new Scanner(System.in); sc.useLocale(Locale.US);
 ????? x = sc.?????();
 System.out.println( (x==-x) + " " + (x!=0) );
}
}
```

【代码 2】

```
import java.util.*;
public class C2 {
 public static void main(String args[]) {
 Scanner sc = new Scanner(System.in); sc.useLocale(Locale.US);
 ????? x = sc.?????();
 ????? y = sc.?????();
 ????? z = sc.?????();
 System.out.println( (x==y) + " " + (y==z) + " " + (x==z) );
}
}
```

【分析】

这道试题主要考察选手对程序设计语言的熟悉程度。首先需要指出，数学中的相等与计算机中的相等并不是相同的意思。当计算机中的两个数进行比较时，如果两个数类型不同，会将高精度数转化成低精度数所属的类型，并在同一类型下进行比较。这样就会发生精度损失，例如 double 的比较要使用误差处理，就是这个道理。

对于第一个问题，涉及到 x 与-x 的问题，我们需要了解计算机中计算负数时的流程：直接对符号位取反。而有符号整数在计算机中存储时，符号位表示正负，后面的若干二进制位表示数的绝对值。但 000....0 与 1000...0 似乎是相同的，都代表 0 吗？不是的。在计算机中，000...0（符号位为 0）代表整数 0，而 1000...0（符号位为 1）代表最小的整数-2147483648（如果使用 32 位 int）。这样，-2147483648 就存在问题：它符号位取反之后，得到的是本身，而不是 2147483648，因为计算机中不能存储+2^{32} 这样一个数，而误认为所有的 2^{32} 都是负的。

因此，对于第一个问题，填入 int -2147483648 即可。

对于第二个问题，我们需要使得 x=y, y=z, 而 x!=z. 那么只有利用浮点误差，注意到题目描述中，int 有 32 位有效数字，而 float 只有 24 位有效数字，因此可以让 int 与 float 进行比较，四舍五入丧失精度，从而“不相等”。在 int 与 float 比较时，全部变成 float，因此可以使 x、z 为相差不多的两个大整数，y 为 x、z 之间的一个实数，这样 x、y 比较时“相等”，y、z 比较时“相等”，而 x、z 比较时由于是整数，发现不等。

我们可以取 int x=1000000000, float y=1000000001, int z=1000000001. 这样可以得到 true true false 的结果。

类似这样的问题，在编写“正规”的题目时也可能出现，因此要特别小心。例如有一道题，要求读入两个 int64 范围内的整数，并输出它们的和。乍一看，可以直接使用 int64 解决，但不能 AC，因为两个 int64 之和可能超过 int64，因此必须使用高精度。（或者可以使用 unsigned int64，但需要复杂的判断处理）

9.10.4 Discover All Sets

【问题描述】

现有 K 张卡片，每张卡片有 N 种性质，每种性质有 M 种不同的取值。输入数据给出一个 K*N 的矩阵描述这些性质。

要求找到满足下列要求的 K-卡片组的个数：对于每种性质，或者 K 张卡片全部相同，或者 K 张卡片互不相同。

【分析】

对于简单的测试数据，简单的枚举每个 K-卡片组，并验证是否符合条件，是可以做到的；但大规模的数据显然不可行。由于是提交答案试题，打开文件进行观察，发现测试数据

的特征数都是偶数。这是否暗示着我们什么特殊的计算方式？

既然是偶数个特征，我们便设想二分处理。首先将 M 个特征分为两组，前 M/2 个和后 M/2 个。然后针对每 M/2 个特征进行枚举，通过计算 Hash 值进行合并。

9.10.5 Expected Cost

9.10.6 Find the Meaning

9.10.7 Game on Words

9.10.8 Hidden Text

9.10.9 Inventing Test Data

9.10.10 Just a Single Lie

9.10.11 K Equal Dights

9.10.12 Large Party

体验了提交答案试题的乐趣，读者是否感觉与传统的信息学竞赛试题有所不同？

以下是 IPSC2008 的成绩统计结果。[1]

Problem	Correct solutions + Wrong answers	
	Easy data set	Difficult data set
A	366+169	352+61
B	275+193	222+38
C	203+98	125+47
D	189+24	3+19
E	1+20	0+2
F	81+200	6+46
G	118+522	71+927
H	144+125	77+137
I	123+90	97+16
J	14+22	3+16
K	175+32	35+23
L	99+62	30+10

Total correct submissions: 2809
Total wrong answers: 2909
Total submissions: 5718

按照规则，每道题目有 10 次提交机会，加之竞赛时间较长，WA 的比例较小，这是与国内竞赛的不同之处。

另外，由于这项竞赛是网上竞赛，很多选手可以上网查询相关资料，这也是竞赛的得分偏高。但我们参加此类竞赛时，为了检验自身的实力，除非题目要求使用网络（例如 Problem F），尽量不要查询其他资料。

9.11 IOI2008

9.12 APIO2009

10 骗分的实质

10.1 Keep it simple and stupid

骗分路漫漫，我们一直寻求的“骗分的本质”，就是如何取得高分的本质方法。

“Keep it simple and stupid”是著名的“KISS 原则”，被广泛应用于管理学等社会学科，在自然科学的研究中，也有重要的参考价值。

[1] http://ipsc.ksp.sk/stats.php?arg_contest=ipsc2008

“骗分”，从本质上说，就是“Keep it simple and stupid”，就是“**从简单处出发考虑，尽量设计简单易行的算法**”。

简单，是自然界本质的诠释，众多自然科学的结论，无论是牛顿三定律，还是麦克斯韦的场方程，都遵循着自然美妙的原则。程序设计，也应该如此，事实证明也确实如此。很多经典的算法，源自于精妙的构思；很多骗分的魔术，源自于考场的创造。

我们追求“骗分”，实质上就是追求一种“简单”的算法，能够得到尽可能多的分数；即使不是骗分，算法优化的目的仍然是尽可能使得时间复杂度简单，或者编程复杂度简单。

“Keep it simple and stupid”，就是非完美算法的最好诠释。一个看似简单、笨拙的算法，可能在实际应用中有着重大意义。一个看似不完美的算法，可能在时间复杂度上有着完美算法无法比拟的优势，而精确度、优化程度又很可观。在这个方面，大型线性方程组的高斯消元法与迭代法的比较，就很能说明问题。

很多问题并不是那样复杂，退一步海阔天空，从另一个角度考虑，也许会柳暗花明。素数判定算法的探索优化历程，就是极好的范例。

本文已经介绍了一些效率不同的素数判断算法，从朴素的筛法，到近代的 Miller-Rabbin 算法，后者应用了费马小定理的有关内容；而 2002 年三位印度科学家简单易行的算法，又开辟了素数判定领域的新天地。

算法如下[1]：

```
Input: integer n>1
1. If (n is the form a^b, b>1) return COMPOSITE;
2. r=2;
3. While(r<n) {
4.    If( gcd(n,r)!=1 ) return COMPOSITE;
5.    If( r is prime )
6.       Let q be the largest prime factor of r-1;
7.       If( q>=4√r log n) and (n^((r-1)/q) mod r !=1 )
8.          Break;
9.    r=r+1;
10.}
11.For a=1 to 2√r log n
12.   If( (x-a)^n mod x^r-1,n ≠ x^n-a ) return COMPOSITE;
13.Return PRIME;
```

素数判定算法，在经历了长时间的“椭圆曲线算法”的探索后，又回到经典的初等数论算法，这体现了研究“在螺旋中上升”的特点，也体现了“简单”之美。

何止是科学研究，在信息学竞赛中，KISS 原则也是我们“骗分”的核心思想。

某同学总结出 NOIP 的经典类型：动贪、暴搜、裸做、交表。解释一下，就是动态规划、贪心、搜索、模拟、预计算。其实，真正能预计算并交表的可能性很小，但前四种类型是必出的。此外，一道图论（最短路、最小生成树）或树形结构（并查集、线段树）的题目也是必不可少的。至于数学题（组合）、计算几何、网络流、树状数组等高级算法、数据结构，考试的可能性较小，除非题目过难。

[1] 英文论文：http://www.irgoc.org/Article/UploadFiles/200610/20061028135903917.pdf

特别需要注意的是，一定要注意 NOIP 规则，不要乱用库函数、变量类型，例如 C/C++中的 int64 就不可用。另外，写错文件名、变量忘记定义、忘记 return 0 等低级错误一定不要犯。切记，程序调试用的输出语句 printf、中断语句 getch 等一定要在提交前彻底删除。

无论走到哪里，“Keep it simple and stupid”，都会指导我们探索的进程。

10.2 告别骗分

骗分，一个永恒的名词，自从“我是智障”之后，便广泛流传于信息学竞赛界，成为“考试策略”的代名词。骗分的历程就是探索的历程，骗分的努力对我们信息学竞赛选手来说，就是新的长征。

但我们不能沉浸在“骗分”的喜悦中，走上信息学的研究之路，只靠“骗分”的小技巧是不可行的。当我们通过“骗分”进入理想的学府，到了开创属于自己的一片天地的时候，就应该思考真正巧妙的算法，真正效率高的算法，而不是停留在“降低时间复杂度”的层面。

我们不难看到，现代信息学理论研究的很多问题，都是“用编程复杂度换时间复杂度”，为了一个小小的优化，程序规模可能增加一倍，两倍，甚至更多。有时甚至需要建立一套全新的理论来优化算法。

但是，无论走到哪里，骗分的实质——KISS 原则，都始终伴随我们。骗分的技巧不能终生捧住不放，但“骗分”的精神，“骗分”的思想，我们会受用终生。

参考文献

[1]刘汝佳、黄亮，《算法艺术与信息学竞赛》
[2]历届 NOIP、NOI、WC 试题与题解
[3]1999-2009 国家集训队论文集
[4]2009 信息学冬令营讲义

感谢

本文自二月二十一日动笔以来，或作或辍，绵延以至今日。其间点点滴滴，皆为笔者解题之心得，凡有所思，俱列于此文，未曾有所掩饰。

由笔者之浅陋，广考文献，摘先哲之精华，聚往昔之妙法，方得《骗分导论》，是故本文之作，不可不引经典，不可不念旧恩，遂谨此致谢。

笔者受父母之发肤，方可得今日之生；蒙祖父母之恩，方可致今日之势。夫初出于母体，祖父母恐余孤苦，弃天伦之乐，守幼子于室，关爱体察，无微不至；方年少时，启蒙有道，教子有方，故得今日余科学之境，自立于学子之林；品行方正，树余之正气，立诚之道德，出淤泥而不染，是树木则直，树人则正也；时至今日，凡十六年，以年迈之躯，躬亲抚养，任劳任怨，置己之辛苦于不顾，唯余学之深厚为忧，诚心之挚，九州共闻。每见其一步一趋，便得养育之艰，顿泪如泉涌。乌鸟报恩之情，是以区区不能废远。

笔者入高中，凡一年八月有余，雷刘蔡靳，启航领者；潘达引路，程序人生。余无恩师，无以至今日。传业解惑，授道明理，师恩浩荡，小生之心，一言难尽。

分班前后，好友如林，体察深情，无私相助，答疑解惑，同甘共苦，是以学之相伴也。余无挚友，无以克艰难。无一字虚言驾饰，诤友之诚，余以为知遇之恩，当以至性相报。

至于省队，沐浴清化。余本水平低贱，当赴高考，今过蒙拔擢，授以省队，岂敢盘桓，有所希冀。报养育之恩，扬母校之威，重任于肩，不敢稍有懈怠。

是故余将数月之心血公诸于世，作此《骗分导论》，以飨后人，望读者听之信之。余之希冀，以中华复兴为己任，若得九州共鸣，则养育之情，教导之恩，可来日相报矣。

后记——程序人生[1]

人生处处程序，程序片片人生。

当 NOIP 的终止哨吹响，又是一切似曾相识的开始。也许悲伤，也许畅快；也许为 NOI 而继续徜徉，也许永远作别那熟悉的机房。程序，字母与数字的集合，诠释着生命本初的向往。

有人问，学习信息学竞赛有什么用。显然，多数人就是为了保送一所好大学。更有甚者，“NOIP 一等奖证书一发，算法导论就扔进旧书箱”。我们不禁想，这样学到的一些所谓的程序，不过是无聊字母构成的一个个文件，实在没有什么深的意义；由此，中国学术的不正之风，中国难以拿到的诺贝尔奖，中国在科技领域仍然落后的事实，也便不难诠释了。

偌大的中国，竟不能得一个诺贝尔奖！观夫日本以弹丸之地，致科技之巅，获强国之位，岂为科技之原因乎？中华五千年灿烂，领先于世界，而今纵一奖而不可得，何也？功名过甚而钻研不足也。科技如此，竞赛何以例外？试使竞赛之生，得奖而掷学，不思持之以恒，则中华信息产业之复兴，又远于强国之林也。

我早就说过，我向来是不对当下某些人的素质抱有任何幻想的。但我也明白，单凭一己之力，难以改变现实，只能奋臂高呼，仅此而已。而此生最美的风景，则是自然，更进一步，则是科学。

富兰克林说过，他人生的最大动力是爱情、知识和对世界的同情心。我则认为，站在自然界浩瀚的大海面前，捡拾几颗美丽的珍珠，则是人生最大的财富。

最美的风景不在浮华的宫殿，最深的情感不在潇洒的语言。人类区别于动物的最大特征，就是感情系统。而将这个系统放在怎样的环境下运行，装入怎样的软件，则是由每个人自己决定的。这台系统是如此精密，它对客观世界的一切细小的变化了如指掌；这台系统又是如此强大，它对客观世界的一切复杂的事情洞察清晰。系统软件，则是人装入的思想与境界。

我为我的系统装入这样的程序，它不求功名，只求现在的快乐；它不求光耀，只求自然的探索。它不愿为那些所谓的“一等奖”徘徊于两点一线之间，只为心中的理想而进入那浩如烟海的竞赛书籍的世界。

人生俯仰，就是在构建属于自己的程序；程序纷繁，就是在控制人生的进程。

当程序调入内存，一个生命呱呱坠地。程序编写完毕，不容修改，正如先天的基因不能改造。算法的优劣，已经决定了这个程序的初步走向；但更大的变化空间，还在输入的数据：一组好的数据，适合的数据，能使程序瞬间出解，非常正确；而那些极限数据，或超时，或错误，难以保证程序的运行是否顺利完成。

当读入函数开始执行，教育构成一切变量的初始值，起到至关重要的作用。硬盘灯在闪

[1] http://boj.pp.ru/olympic/comp/programlife.htm

烁，那是记忆系统在疯狂的接受这个世界赋予他的一切。没有评判，没有选择，只有默默的接受，不知所云的接受，即使是机械的模仿。动作，语言，思想，感情，一项项参数从自然界的文件读入。内存中，一个世界的雏形，正在渐渐形成。

当主运行函数开始执行，奋斗的历程谱成一曲美妙的音乐，但静悄悄的，不为世人所知，用瓢泼的汗水挥洒在处理器的运行途中，内存中构建着未来的生命之塔。CPU 利用率达到 100%，内存占用达到峰值，这时系统最累最苦的时刻，但黑色的屏幕没有任何反应，就像隐匿深山的桃花源中发生的一切。算法的优劣，这时已见分晓，只是未被世人所知罢了。

当一切做好准备，输出函数便开始工作。渐渐为世人所知，成果缓缓浮现，一项项与输入迥然不同的输出，展现在社会的大屏幕上。程序的好与坏，对与错，价值连城与一文不值，也便开始接受世界的评判。输出的内容，同样不容更改。你不能在测评结束后修改主程序，你不能在结果输出后反悔执行的失误。

当输出宣告完毕，程序的使命便行将结束。它只是做几个收尾的工作，把文件关闭，以便后人使用；把内存释放，以便其他程序运行；最后，进程从任务管理器中消失，宣告了程序的终结。终结的程序并没有为自然界忘记，内存中还有运行过程的影子，只是很快就被重新分配初始化，于是程序运行的过程被自然界的操作系统所淡忘；但输出的文件，却永久保留在自然界的硬盘上，留待再次使用。

当又一个用户登录，看到这个程序生命中所输出的一切，就会生发出无限的感慨。或崇敬，或唾弃；或喜爱，或质疑；或光宗耀祖，或名声扫地。因为算法的优秀，有的输出百年不遇；但人们想找回程序运行的中间过程，甚至重新运行一遍，却又是那样的不可能：程序纷繁，那些内存的地址早已被覆盖千百次，不留任何痕迹。于是，算法变成了人类永恒的话题。为了一个良好的输出，主程序中深度优先，广度优先，在自然界的初值中反复探索、回溯、更新最优值，不惜 CPU 完全占用，不惜把其他进程的空间挤占，甚至试图获取操作系统的底层控制权限，俯瞰程序运行的一切。

这就是程序。人生的程序，与我们编写的程序相较，输入输出难以计数，函数语句冗长难懂，变量定义杂乱无规。这样，搞懂人生的程序，便成为难于上青天的任务。好在几千年的先哲们，已经零散的作出了片片研究成果，散落在文明的田野中等待捡拾，这些算法，或许值得我们学习，不说抄袭，但大可借鉴。

创造良好的算法，是人生追求的目标；而自己创造的算法，又指引生命之程序执行的方向。人生程序，程序人生。每当我看到那程序的窗体一闪而过，便想到内部蕴含的丰富的过程。也许程序的运行十分短暂，但输出将长久地停留在操作系统的桌面。

我为什么学习信息学竞赛，为什么初赛遇到如此大的风波临危不乱，为什么复赛进入省队却并未欢声雀跃，从人生的程序，也许可以找到答案。信息学竞赛，在人生的程序中，也许只是一个极小的函数；但每个函数都具有不可或缺的功能，缺少任何一条语句，就不能称作是完整的程序。

我愿用程序控制人生。选择一个优秀的算法，为全人类输出有价值的成果，就是我毕生的追求。我不在意执行过程中的千变万化，只为那输出一刻。把学习数学物理信息学三个学科的竞赛当作乐趣，获得如此令人羡慕的成绩，同时搞好文化课，包括费大力气开办这个博杰学习网，把我所知的一切毫无保留的告诉所有热衷于科学事业的人们……在常人看来或幼稚，或天真，或虚伪，或功名显赫，或不可思议的那些努力，其实都是为了那个人生终极的程序——为了给自然界的操作系统留下有意义的输出。我始终相信，有这个不言悔的信条，任何难以企及的高峰，都可以在开阔豁达的背景下攀登。

但愿每个人生都有美好的程序相伴。

最后，让我们与温总理一起《仰望星空》，彻悟人生程序，品味程序人生。

我仰望星空，
它是那样寥廓而深邃；
那无穷的真理，
让我苦苦地求索、追随。
我仰望星空，
它是那样庄严而圣洁；
那凛然的正义，
让我充满热爱、感到敬畏。
我仰望星空，
它是那样自由而宁静；
那博大的胸怀，
让我的心灵栖息、依偎。
我仰望星空，
它是那样壮丽而光辉；
那永恒的炽热，
让我心中燃起希望的烈焰、响起春雷。