

www.bn51.com  
站 爱 智 城 市 生 活

Programming Challenges

The Programming Contest Training Manual

# 挑战编程

## 程序设计竞赛训练手册

Steven S. Skiena  
Miguel A. Revilla 著

刘汝佳 译



清华大学出版社



Springer

Programming Challenges  
The Programming Contest Training Manual

# 挑战编程

## 程序设计竞赛训练手册

Steven S. Skiena 著  
Miguel A. Revilla

刘汝佳 译



清华大学出版社  
北京



Springer

English reprint edition copyright © 2009 by Springer-Verlag and TSINGHUA UNIVERSITY PRESS.

Original English language title from Proprietor's edition of the Work.

Original English language title: Programming Challenges: The Programming Contest Training Manual by Steven S. Skiena, Miguel A. Revilla, Copyright © 2009.

All Rights Reserved.

This edition has been authorized by Springer-Verlag (Berlin/Heidelberg/New York) for sale in the People's Republic of China only and not for export therefrom.

本书影印版由 Springer-Verlag 授权给清华大学出版社出版发行。

本书封面贴有清华大学出版社防伪标签，无标签者不得销售。

版权所有，侵权必究。侵权举报电话：010-62782989 13701121933

图书在版编目 (CIP) 数据

挑战编程：程序设计竞赛训练手册 / (美) 斯基纳 (Skiena, S. S.) , (西) 雷维拉 (Revilla, M. A.) 著；刘汝佳译. —北京：清华大学出版社, 2009.7

书名原文：Programming Challenges: The Programming Contest Training Manual  
ISBN 978-7-302-19797-3

I. 挑... II. ①斯... ②雷... ③刘... III. 程序设计 IV. TP311.1

中国版本图书馆 CIP 数据核字 (2009) 第 045370 号

责任编辑：龙啟铭

责任校对：徐俊伟

责任印制：王秀菊

出版发行：清华大学出版社

地 址：北京清华大学学研大厦 A 座

http://www.tup.com.cn 邮 编：100084

邮 编：100084

社 总 机: 010-62770175 邮 购: 010-62786544

投稿与读者服务: 010-62776969, c-service@tup.tsinghua.edu.cn

质量反馈: 010-62772015, zhiliang@tup.tsinghua.edu.cn

北京市人民文学印刷厂

装 订 者：三河市新茂装订有限公

经 销：全国新华书店

开本：185×230 印

版 次：2009年7月第1版 印 次：2009年7月

印數 1~3000

定 价 20.00 元

定价：35.00 元

---

中文字本由高志明校

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题,请调换。联系电话: (010) 62770177 转 3103 产品编号: 030502-01

## 译者序

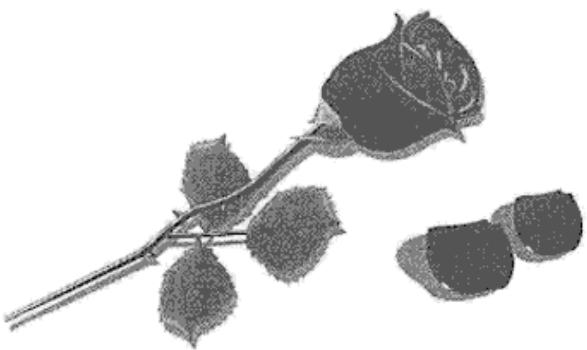
不管对初出茅庐的新人还是身经百战的老手，用“挑战”一词形容程序设计竞赛是再合适不过的了。酷爱编程的人们往往喜欢挑战，但大多数程序员对各种程序设计竞赛却是“敬而远之”，为什么会这样呢？原因在于，学习编程语言和软件开发的知识只是接受这些挑战的必要而非充分条件。要想在程序设计竞赛中脱颖而出，还需要更多的知识和技能。而这些知识和技能，却是很难在传统的课堂和教科书中学到的。

本书的目标读者便是那些已经具备初步的编程技能，对程序设计竞赛充满好奇，希望有机会武装自己、接受编程挑战的人，以及他们的老师和教练（甚至父母）。即使不参加任何竞赛，从本书的编程挑战中学到的东西，也会对程序员的职业生涯产生重要影响，更不用说这些挑战本身就是充满乐趣、引人入胜的。

本书文字精练、通俗易懂。尽管每一章都涉及一个不同的领域，但篇幅却短得甚至可以一口气读完。另外，所有题目均附有难度、流行度等客观评价系数，并可以在线提交。写出程序并不意味着完善的解决了难题，只有通过了评测系统的严格把关才能让人信服。

全书由刘汝佳主译，并得到王希、杨锐、尹淳兴、邱前皓等的大力协助。感谢两位作者算法大师 Steven S. Skiena 教授和在线评测系统 UVaOJ 的创立者 Miguel A. Revilla 教授邀请译者完成本书的翻译工作，提供了书的源程序和插图，并讨论书中的一些细节；感谢清华大学出版社的龙啟铭编辑，他对工作认真负责的态度和严谨的科学作风令人钦佩。尽管我们付出了许多努力，但译文中难免有翻译不当之处，敬请批评指正。

仅以此书献给我们的妻子们 Renee 和 Carmela，以及孩子们 Bonnie，Emilio 还有 Miguel。和书中的难题相比，抽出时间来陪伴这些我们所爱的人更具挑战。



## 前　　言

计算机编程能给人带来很多特殊的快乐。付出总会有回报，亲手做出一个有用的东西并看着它成功运转时，你为之满足；灵光一现，于是轻松解决一个困扰你多年的问题时，你为之兴奋；对美的执着追求能让一名普通黑客成为艺术家，而吝啬已然成为了一种美德，尤其是从那些经过锤炼后的精巧算法和简洁代码中榨取最后一滴“性能之油”。

国际编程竞赛题目中的游戏、谜题和挑战是体验这些快乐的绝佳途径，同时还能提高你的算法能力和编程技巧。本书包含了百余个历届比赛中出现的题目，并讨论了解决这些题目所需的理论和思维方式。读者可以在两个在线评测系统中提交其中任何一道题目的程序，并获得即时的自动评分。若能将评测系统和本书有机结合，你会亲身体会到：迎接挑战和提升编程水平是一件多么新奇、刺激的事啊！

本书可用于自学、讲授算法、编程类创新课程以及比赛的训练。

### 致读者

本书中的题目选自 Valladolid 大学在线评测系统 (<http://uva.onlinejudge.org/>) 中的上千道编程题目。到目前为止，该系统已经评测了来自 27 000 个注册用户的超过一百万份提交。我们只选择了精品中的精品，即那些最好玩、最刺激和最有意思的题目。

我们把这些题目分成若干个类别，然后提供了足够的教学材料（主要是数学和算法）来帮助你解决它们。书中配备了很多参考程序来更好地讲解重要的概念。阅读本书并尝试解决这些题目，你会对回溯法、动态规划等算法技巧以及数论、计算几何等高级话题有更加具体和深刻的理解。即使不参加编程竞赛，关注这些东西也是十分有益的。

多数题目都很生动。它们展示了计算机科学和数学中令人着迷的话题，有时还会以搞笑故事的面目出现。它们往往涉及一些有意思的领域来深入学习研究，因此在合适的时候，我们总是给出注解和进一步的阅读材料。

我们发现，主要接受项目开发和软件工程方面训练的人们通常忽视了算法的重要性。类似地，理论派算法研究者往往低估了把算法转化为程序的难度，也不清楚编程智慧如何化繁为简。

基于上述原因，本书的第一个部分主要关注编程技巧，包括数据类型和库函数的合理运用。这些是书中第二部分那些以算法为主线，难度更大章节的基础。要想成为一名全能的问题求解大师，必须同时精通这两个部分。

## 致教师

本书被设计为以下三种类型的课程教材：

- 偏重编程的算法课程。
- 偏重算法的编程课程。
- 训练学生参加 ACM/ICPC(ACM 国际大学生程序设计竞赛) 或 IOI(国际信息学奥林匹克) 的选修课。

这样的课程对所有相关人士来说都是愉快的。学生的潜能很容易被比赛本身的刺激所激发，并且每次提交通过一道题目时都将得到增强。最直白的程序可能会被测试系统返回“超时”信息，于是学生会很自然地寻找更有效的解法。正确的洞察力会让一个十余行的小程序取代一大片杂乱无章的代码。最优秀的学生甚至自愿尝试更多的题目来获得快感。

讲授这样的课程也是愉快的。很多题目十分精巧，尽管在算法和编程本质上并无特殊之处，但题目却令人耳目一新。对于这样的问题，要找到最佳解法需要洞察力和灵感。寻找这些题目的解法是令人兴奋的，而当我们所教的学生独立找到解法时，我们会更加兴奋。

从教育学的角度讲，本书的特点包括：

- 作为标准算法教材的补充——尽管本书体系完整，写作时仍然考虑到大多数学生已有了些算法基础。本书是作为传统算法课程的补充教材来设计（与定价）的，它用具体实现补充抽象描述，用实践经验补充理论体系。另外，它还涉及一些多数标准教材中都没有提到的有趣主题。
- 提供经典算法的完整实现——很多学生在把抽象的算法描述转化成能正确运行的代码时困难重重。为了帮助他们，我们精心编写了书中所讨论的所有重要算法的程序。这些程序只用到 C 语言的一个子集，因此 C++ 和 Java 程序员也能轻易看懂。书中的不少习题都只需适当修改这些例程即可解决，从而为初学者提供了一条宽敞的入门之路。
- 内置课程管理系统——我们已经打造了一个特别的课程管理系统，使得管理一门这样的课程易如反掌。所有测试和评分都是自动进行的，你完全不必操心！用我们的网站 <http://www.programming-challenges.com> 可以轻松管理花名册、布置作业、查看学生的程序和分数，甚至还能检测到疑似作业！
- 帮助程度参差不齐的学生——本书在题目选择时有意覆盖了一个较宽的难度范围。多数题目适合初学者，而其他题目即使对于备战国际竞赛的选手来说也是富有挑战的。我们为多数题目提供了解题提示。

为了帮助学生挑选适合自己的题目，我们给每道题目标注了三种类型的难度值。题目的流行度 (A, B, 或 C) 是指有多少人提交过此题，而成功率 (low 表示低, high 表示高) 是指这些人中正确解决此题的比率。最后，题目的等级 (从 1 到 4，大致表示从新手到高水平读者) 表示解题者应具有的水平等级。

致竞赛选手与教练

本书尤其适合作为高中和大学的编程竞赛培训材料。我们为数学和算法中的重要主题提供方便的总结和参考，还配备了合适的挑战题目来帮助你掌握这些内容。

自动评测系统像 ACM/ICPC 中的真人裁判一样检查所提交程序的正确性。只要你拥有该系统的个人账号，你就可以提交 C、C++、Pascal 或 Java 程序，然后等待系统告诉你这个程序是对还是错。系统还会为你保存一份统计信息，让你有机会与成千上万的其他用户进行对比。

为了帮助竞赛选手，我们请到了三个重要编程比赛——ACM 国际大学生程序设计竞赛 (ICPC)、国际信息学奥林匹克 (IOI)、TopCoder 编程挑战赛——的决赛选手分享经验，并把这些成功的秘密写在附录中。我们还介绍了这三项赛事的历史和参赛方法，最大限度地帮助你施展才华。

大约有 80% 的 ACM/ICPC 决赛选手用 Valladolid 在线评测系统进行训练。ICPC 全球总决赛常常在夏威夷这样令人神往的异国他乡举行，这无疑为选手们提供了额外的精神动力。加油吧！

### 相关网站

本书有两个配套网站。书中的所有题目均可在 <http://www.programming-challenges.com> 提交，该网站还提供了一些辅助材料，包括书中所有程序的电子版以及帮助教师们把书中的材料融入课堂的课程笔记。

本书中的所有题目（以及很多其他题目）也可以在 Valladolid 大学在线评测系统 <http://uva.onlinejudge.org/> 中提交。书中的所有习题均同时标注两个评测系统的 ID 号，供读者选择。

致谢

本书的成书在很大程度上归功于那些欣然授权把他们设计的竞赛题目用在本书和在线评测系统中的人们。本书中的题目由来自四大洲的至少 17 位命题者提供，尤其是 Gordon Cormack 和 Shahriar Manzoor，就好比 Sam Loyd 和 H. E. Dudeney<sup>1</sup>一样！

作者和题目的完整列表见附录，但我们要在这里特别感谢如下的竞赛组织者：Gordon Cormack (38 题), Shahriar Manzoor (28), Miguel Revilla (10), Pedro Demasi (8), Manuel Carro (4), Rujia Liu(刘汝佳) (4), Petko Minkov (4), Owen Astrakan (3), Alexander Denisjuk (3), Long Chong(龙翀) (2), Ralf Engels (2), Alex Gevak (1), Walter Guttmann (1), Arun Kishore (1), Erick Moreno (1), Udvrant Patik (1) 以及 Marcin Wojciechowski (1)。其中的部分题目由第三方设计，我们已在附录中向他们致谢。

<sup>1</sup> 译者注：两位都以提出了大量有趣但富有挑战性的数学谜题而著称。

其中一些题目的原作者已经难以考证。我们想尽办法找到每道题目的原作者，并得到了能代表作者发言的人授权。如有疏漏，在敬请原谅的同时还希望能告知我们，以便在致谢中补充。

在线评测系统项目是许多人辛勤劳动的结晶。Ciriaco García 是评测系统软件的主要开发人员，也是项目的核心支持人员。Fernando P. Nájera 开发了一系列辅助工具，使得评测系统更加友好。Carlos M. Casas 维护测试数据，确保它们在正确的前提下既公平又严格。José A. Caminero 和 Jesús Paúl 负责检查和完善题目描述，并测试参考程序。我们还要特别感谢 Miguel Revilla, Jr，他建立并维护着 <http://www.programming-challenges.com>。

本书曾作为 Stony Brook 2002 年春季学期的一门课程的教材。该课程由 Vinhthuy Phan 和 Pavel Sumazin 任教，并修改了诸多错误。我们今年很棒的两支参赛队伍的成员 (Larry Mak, Dan Ports, Tom Rothamel, Alexey Smirnov, Jeffrey Versoza 和 Charles Wright) 帮忙评阅了手稿，感谢他们的兴趣和反馈。Haowen Zhang 仔细阅读了手稿，测试了书中的程序，并简化了代码，为本书的修改做出了重要贡献。

感谢 Springer-Verlag 的 Wayne Yuhasz, Wayne Wheeler, Frank Ganz, Lesley Poliner 和 Rich Putter。没有他们的帮助，本书无法从手稿最终变成出版物。我们还要感谢 Gordon Cormack, Lauren Cowles, David Gries, Joe O'Rourke, Saurabh Sethia, Tom Verhoeff, Daniel Wright 和 Stan Wagon。他们认真仔细的评阅极大地提高了最终成品的质量。Fulbright Foundation 和 Valladolid 大学应用数学与计算系为两位作者面对面交流和一起工作提供了必要的支持。Citigroup CIB，通过 Peter Remch 和 Debby Z. Beckman 所做出的努力，为 Stony Brook 大学的 ACM ICPC 产生了深远的影响。它的参与促成了本书的写作。

Steven S. Skiena  
Stony Brook, NY

Miguel A. Revilla  
Valladolid, Spain

# 目 录

## 译者序

## 前言

<b>第 1 章 入门</b>	1
1.1 初识自动评测系统	1
1.1.1 评测系统反馈	1
1.2 挑选你的武器	3
1.2.1 程序设计语言	3
1.2.2 如何阅读本书的程序	4
1.2.3 标准输入输出	5
1.3 编程提示	6
1.4 基本数据类型	8
1.5 关于习题	10
1.6 习题	11
1.6.1 $3n + 1$ 问题 ( $3n+1$ Problem)	11
1.6.2 扫雷 (Minesweeper)	12
1.6.3 旅行 (The Trip)	13
1.6.4 液晶显示屏 (LC-Display)	14
1.6.5 图形化编辑器 (Graphical Editor)	15
1.6.6 解释器 (Interpreter)	16
1.6.7 将军 (Check the Check)	17
1.6.8 澳大利亚投票 (Australian Voting)	19
1.7 提示	20
1.8 注解	20
<b>第 2 章 数据结构</b>	22
2.1 基本数据结构	22
2.1.1 栈	22
2.1.2 队列	23
2.1.3 字典	25
2.1.4 优先队列	26

2.1.5 集合 .....	26
2.2 库函数 .....	27
2.2.1 C++ 标准模板库 .....	27
2.3 程序设计实例：纸牌大战 .....	28
2.4 准备行动 .....	29
2.5 字符串输入输出 .....	30
2.6 赢得战争 .....	32
2.7 测试与调试 .....	33
2.8 习题 .....	35
2.8.1 快乐的跳跃者 (Jolly Jumper) .....	35
2.8.2 扑克牌型 (Poker Hands) .....	35
2.8.3 罢工 (Hartals) .....	36
2.8.4 解密 (Crypt Kicker) .....	37
2.8.5 完美洗牌术 (Stack'em Up) .....	38
2.8.6 Erdős 数 (Erdős Numbers) .....	41
2.8.7 比赛记分板 (Contest Scoreboard) .....	42
2.8.8 Yahtzee 游戏 (Yahtzee) .....	43
2.9 习题 .....	45
2.10 注解 .....	45
<b>第 3 章 字符串 .....</b>	<b>47</b>
3.1 字符编码 .....	47
3.2 字符串的表示 .....	49
3.3 程序设计实例：公司更名 .....	49
3.4 模式查找 .....	51
3.5 字符串操作 .....	52
3.6 程序的完成 .....	54
3.7 字符串库函数 .....	54
3.8 习题 .....	56
3.8.1 WERTYU 键盘 (WERTYU) .....	56
3.8.2 寻找单词 (Where's Waldorf?) .....	57
3.8.3 公共排列 (Common Permutation) .....	58
3.8.4 解密 II(Crypt Kicker II) .....	59
3.8.5 自动评测脚本 (Automated Judge Script) .....	60
3.8.6 文件碎片 (File Fragmentation) .....	62
3.8.7 Doublet 序列 (Doublets) .....	63

3.8.8 Fmt 程序 (Fmt) .....	63
3.9 提示 .....	65
3.10 注解 .....	65
<b>第 4 章 排序 .....</b>	<b>66</b>
4.1 排序的应用 .....	66
4.2 排序算法 .....	67
4.3 程序设计举例：给绅士排名 .....	69
4.4 与排序相关的库函数 .....	71
4.5 给绅士排名 .....	72
4.6 习题 .....	75
4.6.1 Vito 家族 (Vito's Family) .....	75
4.6.2 煎饼堆 (Stacks of Flapjacks) .....	75
4.6.3 过桥 (Bridge) .....	76
4.6.4 最长打盹时间 (Longest Nap) .....	77
4.6.5 鞋匠的烦恼 (Shoemaker's Problem) .....	79
4.6.6 CDVII 高速公路 (CDVII) .....	80
4.6.7 龟壳排序 (ShellSort) .....	81
4.6.8 足球 (Football (aka Soccer)) .....	82
4.7 提示 .....	84
4.8 注解 .....	85
<b>第 5 章 算术与代数 .....</b>	<b>86</b>
5.1 机器算术 .....	86
5.1.1 整数库函数 .....	86
5.2 高精度整数 .....	87
5.3 高精度算术 .....	88
5.4 进制及其转换 .....	94
5.5 实数 .....	96
5.5.1 如何处理实数 .....	97
5.5.2 分数 .....	97
5.5.3 十进制实数 .....	98
5.6 代数 .....	98
5.6.1 多项式运算 .....	98
5.6.2 多项式求根 .....	99
5.7 对数 .....	100
5.8 实数函数库 .....	101

book1051.com  
数 拼 语 爱  
数 学 之 真 球 丽 色

5.9 习题 .....	101
5.9.1 小学生算术 (Primary Arithmetic) .....	101
5.9.2 反转相加 (Reverse and Add) .....	102
5.9.3 考古学家的烦恼 (The Archeologist's Dilemma) .....	103
5.9.4 仅由 1 组成的数 (Ones) .....	104
5.9.5 乘法游戏 (A Multiplication Game) .....	104
5.9.6 多项式的系数 (Polynomial Coefficients) .....	105
5.9.7 Stern-Brocot 代数系统 (The Stern-Brocot Number System) .....	105
5.9.8 两两之和 (Pairsumonious Numbers) .....	106
5.10 提示 .....	107
5.11 注解 .....	108
<b>第 6 章 组合数学 .....</b>	<b>109</b>
6.1 基本计数技巧 .....	109
6.2 递推关系 .....	110
6.3 二项式系数 .....	111
6.4 其他计数序列 .....	113
6.5 递归与数学归纳法 .....	115
6.6 习题 .....	116
6.6.1 斐波那契计数 (How Many Fibs?) .....	116
6.6.2 土地分割 (How Many Pieces of Land?) .....	116
6.6.3 数数 (Counting) .....	117
6.6.4 括号表达式 (Expressions) .....	118
6.6.5 完全树标号 (Complete Tree Labeling) .....	119
6.6.6 牧师数学家 (The Priest Mathematician) .....	119
6.6.7 自描述序列 (Self-describing Sequence) .....	120
6.6.8 数轴行走 (Steps) .....	121
6.7 提示 .....	122
6.8 注解 .....	122
<b>第 7 章 数论 .....</b>	<b>123</b>
7.1 素数 .....	123
7.1.1 寻找素数 .....	123
7.1.2 素数的个数 .....	124
7.2 整除性 .....	125
7.2.1 最大公约数 .....	125
7.2.2 最小公倍数 .....	127

7.3 模算术.....	127
7.4 同余.....	129
7.4.1 同余运算.....	129
7.4.2 求解线性同余式.....	130
7.4.3 不定方程.....	131
7.5 数论函数库.....	131
7.6 习题.....	132
7.6.1 开灯与关灯 (Light, More Light) .....	132
7.6.2 Carmichael 数 (Carmichael Numbers) .....	132
7.6.3 欧几里德问题 (Euclid Problem) .....	133
7.6.4 阶乘与整除 (Factovisors) .....	134
7.6.5 四素数之和 (Summation of Four Primes) .....	134
7.6.6 Smith 数 (Smith Numbers) .....	135
7.6.7 弹珠 (Marbles) .....	135
7.6.8 重新打包 (Repackaging) .....	136
7.7 提示.....	137
7.8 注解.....	137
<b>第 8 章 回溯法.....</b>	<b>138</b>
8.1 回溯法.....	138
8.2 构造所有子集.....	140
8.3 构造所有排列.....	141
8.4 程序设计举例：八皇后问题.....	143
8.5 搜索中的剪枝.....	144
8.6 习题.....	147
8.6.1 棋盘上的象 (Little Bishops) .....	147
8.6.2 15 数码游戏 (15-Puzzle Problem) .....	148
8.6.3 队伍 (Queue) .....	149
8.6.4 服务站 (Servicing Stations) .....	150
8.6.5 拔河 (Tug of War) .....	150
8.6.6 伊甸园 (Garden of Eden) .....	151
8.6.7 色彩缤纷游戏 (Color Hash) .....	153
8.6.8 拼接正方形 (Bigger Square Please...) .....	154
8.7 提示.....	156
8.8 注解.....	156

<b>第 9 章 图遍历</b>	157
9.1 图的不同属性	157
9.2 图的数据结构	158
9.3 图的遍历: 宽度优先	162
9.3.1 宽度优先遍历	162
9.3.2 遍历的应用	163
9.3.3 寻找路径	164
9.4 图的遍历: 深度优先	165
9.4.1 寻找环	166
9.4.2 连通分量	167
9.5 拓扑排序	168
9.6 习题	170
9.6.1 双着色 (Bicoloring)	170
9.6.2 摆弄轮子 (Playing With Wheels)	171
9.6.3 导游 (The Tourist Guide)	173
9.6.4 斜线迷宫 (Slash Maze)	174
9.6.5 递变阶梯 (Edit Step Ladders)	175
9.6.6 立方体之塔 (Tower of Cubes)	176
9.6.7 从黄昏到拂晓 (From Dusk till Dawn)	177
9.6.8 汉诺塔卷土重来! (Hanoi Tower Troubles Again!)	179
9.7 提示	179
<b>第 10 章 图算法</b>	181
10.1 图论	181
10.1.1 度的性质	181
10.1.2 连通性	182
10.1.3 图中的回路	182
10.1.4 平面图	183
10.2 最小生成树	183
10.3 最短路	186
10.3.1 Dijkstra 算法	186
10.3.2 每对结点之间的最短路	188
10.4 网络流和二分图匹配	191
10.5 习题	195
10.5.1 斑点 (Freckles)	195
10.5.2 项链 (The Necklace)	195

10.5.3 消防站 (Fire Station) .....	197
10.5.4 铁路 (Railroads) .....	198
10.5.5 战争 (War) .....	199
10.5.6 导游 (Tourist Guide) .....	201
10.5.7 丰盛的晚餐 (The Grand Dinner) .....	202
10.5.8 命题者的难题 (The Problem With the Problem Setter) .....	203
10.6 提示 .....	205
<b>第 11 章 动态规划 .....</b>	<b>206</b>
11.1 慎用贪心法 .....	206
11.2 编辑距离 .....	207
11.3 重建路径 .....	211
11.4 编辑距离的变种 .....	212
11.5 程序设计举例: 电梯优化 .....	214
11.6 习题 .....	218
11.6.1 越大越聪明? (Is Bigger Smarter?) .....	218
11.6.2 不同的子序列 (Distinct Subsequences) .....	219
11.6.3 重量和力量 (Weights and Measures) .....	219
11.6.4 单向 TSP(Unidirectional TSP) .....	220
11.6.5 切割小木棍 (Cutting Sticks) .....	221
11.6.6 渡船装载 (Ferry Loading) .....	222
11.6.7 筷子 (Chopsticks) .....	223
11.6.8 搬家大冒险: 第四部 (Adventures in Moving: Part IV) .....	224
11.7 提示 .....	225
11.8 注解 .....	225
<b>第 12 章 网格 .....</b>	<b>226</b>
12.1 矩形网格 .....	226
12.1.1 遍历 .....	227
12.1.2 对偶图及其表示 .....	228
12.2 三角形网格和六边形网格 .....	229
12.2.1 三角形网格 .....	229
12.2.2 六边形网格 .....	230
12.3 程序设计举例: 西餐碟的重量 .....	232
12.4 给圆打包 .....	234
12.5 经度和纬度 .....	236
12.6 习题 .....	236

www.pink1.com  
新课标  
教材  
同步  
课堂  
讲义  
课时  
训练  
卷子  
课件  
教案  
学案  
习题  
答案  
解析  
课时  
训练  
卷子  
课件  
教案  
学案  
习题  
答案  
解析

12.6.1 棋盘上的蚂蚁 (Ant on a Chessboard) .....	236
12.6.2 独轮车 (The Monocycle) .....	237
12.6.3 六角星 (Star) .....	239
12.6.4 蜜蜂 Maja(Bee Maja) .....	240
12.6.5 抢劫案 (Robbery) .....	241
12.6.6 $(2/3/4)$ -维立方体? ( $(2/3/4)$ -D Sqr/Rects/Cubes/Boxes?) .....	242
12.6.7 Dermuba 三角 (Dermuba Triangle) .....	243
12.6.8 航线 (Airlines) .....	244
12.7 提示 .....	246
<b>第 13 章 几何 .....</b>	<b>247</b>
13.1 直线 .....	247
13.2 三角形和三角学 .....	250
13.2.1 直角三角形和勾股定理 .....	251
13.2.2 三角函数 .....	251
13.2.3 解三角形 .....	252
13.3 圆 .....	253
13.4 程序设计举例: 超高速飞行 .....	255
13.5 三角函数库 .....	257
13.6 习题 .....	259
13.6.1 狗拿地鼠 (Dog and Gopher) .....	259
13.6.2 绳子王国的危机! (Rope Crisis in Ropeland!) .....	260
13.6.3 圆桌骑士 (The Knights of the Round Table) .....	260
13.6.4 巧克力片饼干 (Chocolate Chip Cookies) .....	261
13.6.5 生日蛋糕 (Birthday Cake) .....	262
13.6.6 最大/最小的盒子 (The Largest/Smallest Box ...) .....	263
13.6.7 要算积分吗? (Is This Integration?) .....	264
13.6.8 它有多大? (How Big Is It?) .....	265
13.7 提示 .....	266
<b>第 14 章 计算几何 .....</b>	<b>277</b>
14.1 线段及其相交 .....	277
14.2 多边形及旋转方向 .....	278
14.3 凸包 .....	270
14.4 三角剖分: 算法与相关问题 .....	274
14.4.1 Van Gogh 算法 .....	274
14.4.2 面积计算 .....	276

14.4.3 点定位	277
14.5 网格上的算法	278
14.5.1 范围查询	278
14.5.2 网格多边形与 Pick 定理	279
14.6 几何函数库	280
14.7 习题	280
14.7.1 新生集会 (Herding Frosh)	280
14.7.2 最近点对问题 (The Closest Pair Problem)	281
14.7.3 电锯惊魂 (Chainsaw Massacre)	282
14.7.4 冷热游戏 (Hotter Colder)	283
14.7.5 没用的瓷砖打包公司 (Useless Tile Packers)	284
14.7.6 雷达追踪 (Radar Tracking)	285
14.7.7 岛上的树 (Trees on My Island)	286
14.7.8 美味的牛奶 (Nice Milk)	287
14.8 提示	288
<b>附录 A</b>	<b>290</b>
A.1 ACM 国际大学生程序设计竞赛	290
A.1.1 准备	290
A.1.2 战略战术	292
A.2 国际信息学奥林匹克	293
A.2.1 如何参加	293
A.2.2 比赛形式	294
A.2.3 准备	295
A.3 Topcoder.com	295
A.4 念研究生吧!	296
A.5 题目致谢	297
<b>参考文献</b>	<b>300</b>

# 第1章 入 门

在本书的开始，我们将看到一些只用数组和迭代即可解决的基础编程题目。但是，基础并不意味着简单！通过这些题目，读者可以了解到自动评测系统对程序的严格要求，并认识到认真阅读和理解规范的重要性。另外，这些题目也为我们讨论最合适编程风格提供了一个良好的平台。

为了帮助读者入门，我们首先介绍自动评测系统和它的特性。接下来，在介绍本书的第一组习题之前将讨论基本的编程风格和数据结构。和本书的其余章节一样，我们在习题之后提供了提示和评注，以帮助读者进一步学习。

## 1.1 初识自动评测系统

本书最好与两个由 Miguel Revilla 管理的自动评测系统结合使用。挑战编程自动评测系统 (*The Programming Challenges judge*)<http://www.programming-challenges.com> 专门为本书而运行，而 Valladolid 大学自动评测系统 (*The Universidad de Valladolid judge*)<http://uva.onlinejudge.org/> 虽然界面略有不同，却拥有成百上千道本书没有的题目。

本书中的所有题目均可以在两个评测系统中提交，但出于清晰性的考虑，本书对这些题目的文字做了少量修订。尽管两个系统的题目描述存在差异，但本质是相同的。理论上，能在其中一个系统中通过测试的程序也能在另外一个系统中通过。两个系统都在不断发展，因此使用方法可能会有变化。请在使用前阅读两个网站各自的说明<sup>1</sup>。

不管使用哪个系统，第一步都是注册帐号。请注意，两个系统的用户数据目前是独立的，这意味着你需要多花一点时间在两个网站分别注册，才能享受它们各自的特色服务。

挑战编程自动评测系统 (<http://www.programming-challenges.com>) 提供了和本书习题风格一致的环境，还包含了一个特别的课程管理系统，而 Valladolid 大学自动评测系统 (<http://uva.onlinejudge.org/>) 则拥有更多的题目，为读者提供了更大的挑战。二者都可以方便地通过 Web 提交，并能很快看到评测结果。

### 1.1.1 评测系统反馈

两个系统对于“正确程序”的定义都是相当苛刻的，因此正确地理解题目说明 (problem specification) 十分重要。不要对题目说明中没有明确规定的东西做出任何假设。例如，不要

<sup>1</sup> 在本书英文版出版后，Valladolid 自动评测系统已经发生了很大变化。根据原作者的建议，译者根据网站现在的情况重写了本节。

假定输入已排好序、图是连通的或者题目中的整数都是不太大的正数，除非题目说明中作了明确规定。

正如 ACM 国际大学生程序设计竞赛 (ACM/ICPC) 中的裁判一样，自动评测系统只给你极少的反馈信息，包括：

- *Accepted (AC)* — 正确。祝贺你！你的程序是正确的，并且运行时间和内存开销均不超过相应的限制。
- *Presentation Error (PE)* — 格式错。你的程序输出了正确的结果，但是输出格式并非严格满足题目说明。请检查空格或换行、左右对齐等。
- *Accepted (PE)* — 几乎正确。你的程序有格式错，但是评测系统在给出警告信息的同时仍然认为该程序正确。不要介意，因为很多题目说明中的输出格式规定本身就有有着或多或少的歧义。你的程序很可能只是在行末多加了一个空格，因此大可就此打住，宣告胜利。
- *Wrong Answer (WA)* — 答案错。你的程序对一组或者多组评测系统内部的非公开测试数据给出了错误的结果，因此还需要更进一步的调试。
- *Compile Error (CE)* — 编译错。编译器无法成功地编译你的程序，错误信息将返回给用户。编译过程中产生的警告信息并不会导致此错误。
- *Runtime Error (RE)* — 运行错。你的程序在运行结束之前由于段错误 (segmentation fault)、浮点异常 (floating point exception) 或者其他类似的问题异常终止。程序终止时的信息将返回给用户。建议检查无效指针引用 (invalid pointer reference) 或者除零错误 (division by zero)。
- *Time Limit Exceeded (TL)* — 超时。你的程序在某一组或多组数据上运行的时间过长，因此该程序可能存在效率问题。注意，在一组数据上超时并不意味着在其他数据上正确！
- *Memory Limit Exceeded (ML)* — 超内存。你的程序试图使用超过评测系统规定大小的内存。
- *Output Limit Exceeded (OL)* — 超输出。你的程序输出了过多的信息。这通常意味着你的程序陷入了一个带输出的无限循环中。
- *Restricted Function (RF)* — 你的程序试图使用一些被测试系统禁止的非法的函数，例如`fork()`或`fopen()`。请不要使用这些函数。
- *Submission Error (SE)* — 你错误地指定了一个或多个提交信息域，例如错误的 ID 或者题目编号。

再次强调：如果你的程序被认为是错误的，评测系统并不会告诉你错在哪个数据，也不会告诉你为什么是错的。即使你对你的程序很有把握，评测系统仍然可能始终对它说不。也许这只是因为你忽略了一个边界情况或者做了一个不成立的假定，但是不对代码进行修改就直接重新提交在绝大多数情况下都是不可取的。再次仔细读题，进一步确认题目的意思和

你先前所理解的是否完全一致。

系统有时也会返回一些上面没有列举出来的特殊反馈信息。这些信息往往和你的程序无关，请阅读网站上的说明。

## 1.2 挑选你的武器

一般来说，你应当选择你最熟悉的语言来写程序。评测系统目前接受 C、C++、Pascal 和 Java 语言的程序。对于大多数人来说，最熟悉的编程语言已经包含在其中了。这些语言各有自身擅长的领域，但是就本书所讨论的问题求解 (problem-solving) 而言，所有四种语言都可以很好地完成任务。问题求解的关键在于算法，而语言之间的可移植性、模块性和效率等方面差异已经变得不那么重要了。

### 1.2.1 程序设计语言

评测系统所支持的四种语言是在四个不同的时期，以不同的目标为导向所设计出来的：

- *Pascal* — 作为 20 世纪 80 年代教育领域最流行的编程语言，Pascal 语言鼓励结构化的编程习惯。尽管它已经不再流行，甚至几乎处于消亡的边缘，但在东欧和中学教育中仍然处于重要地位。
- C — 作为 UNIX 操作系统的原始语言，C 语言为有经验的程序员提供了足够强大的力量来做任何他们想做的事，包括用无效的指针引用和强制类型转换来让程序崩溃。20 世纪 90 年代，面向对象编程技术的进步使得 C 语言逐步发展成了一个崭新的，更先进的 …
- C++ — 第一个成功的、商业化的、面向对象的程序设计语言在成功实现对 C 语言向后兼容的同时引入了新的数据抽象和继承机制。C++ 成为了 20 世纪 90 年代中后期教育和工业领域的主流语言，但它的地位慢慢地被另一个语言所动摇 …
- *Java* — 作为一门功能强大的通用程序设计语言，它拥有一个特殊的安全机制来避免一些常见的编程错误，如数组下标越界或者非法指针访问。

注意上面所提到的每种编程语言都有一些与编译器和系统相关的特性，因此在你的本地机器上可以正确运行的程序不一定能在评测系统中运行。请阅读评测系统中关于编程语言的说明，尤其是当你使用 Java 的时候。

到 2002 年 12 月为止，评测系统收到了超过 1 250 000 份提交。其中接近一半的程序用 C++ 语言写成，另外还有三分之一是 C 语言。只有很少部分的提交使用 Java 语言，但请注意系统从 2001 年 11 月才开始支持 Java 语言。

图 1.1 把这些提交更加细致地按月统计。早期 C 一直是最受欢迎的语言，但在 1999 年底被 C++ 超过。从整体上看，系统的繁忙程度逐年增加。

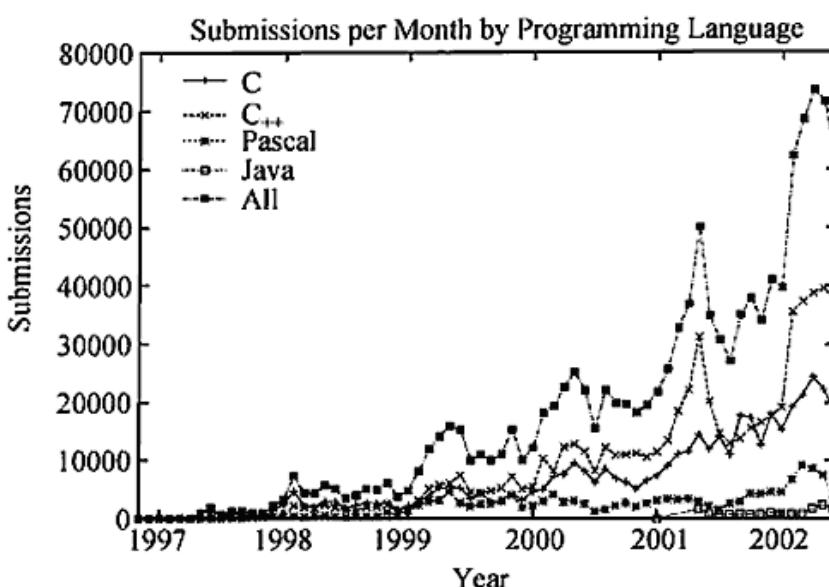


图 1.1 2002 年 12 月之前自动评测系统所接收到的提交

表 1-1 2002 年 12 月为止评测系统给出的结果

语言	总数	AC	PE	WA	CE	RE	TL	ML	OL	RF
C	451447	31.9%	6.7%	35.4%	8.6%	9.1%	6.2%	0.4%	1.1%	0.6%
C++	639565	28.9%	6.3%	36.8%	9.6%	9.0%	7.1%	0.6%	1.0%	0.7%
Java	16373	17.9%	3.6%	36.2%	29.8%	0.5%	8.5%	1.0%	0.5%	2.0%
Pascal	149408	27.8%	5.5%	41.8%	10.1%	6.2%	7.2%	0.4%	0.4%	0.5%
总计	1256793	29.7%	6.3%	36.9%	9.6%	8.6%	6.8%	0.5%	1.0%	0.6%

### 1.2.2 如何阅读本书的程序

本书中包含若干编程实例，用于展示一些编程技巧和基础算法的完整实现。所有的代码均可以在<http://www.programming-challenges.com>上下载，供读者使用和实验。这些程序被成千上万的学生使用和阅读，因此建议读者到该网站查阅最新的勘误表和代码的修订版本。

本书的所有代码均用 C 语言的一个子集实现。我们希望这样的代码可以比较容易地被所有读者理解。C 本身是 C++ 的一个子集，语法和 Java 有不少的相似之处。我们在本书的所有代码中都刻意回避了 C 语言所特有的语法以及指针和动态内存分配，而剩下来的部分对于所有四种语言的用户来说，都应当是熟悉的。

为了帮助读者更好地阅读本书的程序，下面是一些提示：

- 参数传递 — C 语言统一采用传值 (call-by-value) 方式来进行参数传递。这看上去似乎意味着函数不能有任何副作用，但实际并非如此。如果你想修改某一个参数，可以把指向它的指针传递给一个函数。尽管函数无法修改这个指针，但它可以修改该指针所指向的内容，也就是你想修改的那个参数。

在本书中，指针唯一的作用就是参数传递。指向  $x$  的指针写作  $\&x$ ，而指针  $p$  所指向的内容写作  $*p$ 。这里的星号代表“取内容”运算，而不是算术乘法，请读者注意不要

混淆。

- **数据类型** — C 语言支持一些基本数据类型，包括 `int`、`float` 和 `char`。更高精度的整数和浮点数分别为 `long` 和 `double` 类型。未说明返回类型的函数均返回 `int` 类型。
- **数组** — C 语言的数组下标范围总是 0 到  $n - 1$ ，其中  $n$  是数组中的元素个数。因此如果为了使用方便而用 1 作为起始下标，最好是定义一个  $n + 1$  个元素的数组。C 语言不提供运行时刻的数组下标范围检查，因下标越界而引起的程序崩溃非常常见。

在实际应用中，第一个元素的位置并不总是一致的。尽管从 0 开始是 C 语言的传统风格，但有时候把 1 作为开始会让代码更加清晰和简单。在这种情况下，我们愿意为此浪费一个内存单元，来获取这样的简洁性。请读者在阅读本书代码时注意这一点，不要被这种处理方法所迷惑。

- **运算符** — C 语言有一些看上去很奇怪的运算符。例如，整数取余（即取模）运算符为%，在条件语句中常见的逻辑与（logical-and）和逻辑或（logical-or）运算分别为 `&&` 和 `||`。

### 1.2.3 标准输入输出

UNIX 程序员大都对流式输入输出的管道程序十分熟悉，它们的输出经常会作为另外一些程序的输入。人们习惯于把很多这样的小程序串在一起完成任务，而不是制造一个复杂的、功能完善的软件系统并试图覆盖所有的应用。

这样的“小工具哲学”近来受到了图形用户界面（graphical user interface）的冲击。很多程序员习惯于为每个程序包装上一个“指向 - 单击”的图形界面，但这样将使程序之间的数据传输变得非常困难。处理程序的文本输出是很容易的，但面对一张图片的时候，除了“用眼睛看”之外，你还能做什么呢？

和很多正规的 ACM/ICPC 比赛一样，评测系统要求程序从标准输入（standard input）读取数据，并把结果打印到标准输出（standard output）中。程序不允许打开任何文件，也不能执行一些特定的系统调用，如进程和网络相关的函数<sup>2</sup>。

标准输入输出（standard input/output）的使用在 C、C++ 与 Pascal 语言中都是很简单的。图 1.2 展示了如何用这三种语言分别写一个程序，每行读取两个整数，打印它们之差的绝对值，直到文件结束。请读者注意不同语言是如何判断文件结束的。大多数题目通过指定测试数据个数或者规定数据结束标记来避免程序显式的判断文件结束。

大多数语言都提供了强大的格式化 I/O 函数。如果善加利用，可能只需要一行代码就能完成某些看上去很复杂的输入输出任务。这些任务对于既不熟悉这些函数、又不看手册（manual）的程序员来说通常意味着痛苦的解析（parsing）和格式化（formatting）。

可惜的是，Java 的标准输入输出并不简单。<http://www.programming-challenges.com> 包含了一个 35 行长的 Java 输入输出模板以供参考，而其他注意事项可以在<http://uva>.

<sup>2</sup> 如果你希望在本地采用文件输入输出来测试程序，而在评测机器上自动改为标准输入输出，只需测试 `ONLINE_JUDGE` 符号是否已定义。

[onlinejudge.org](http://onlinejudge.org)/找到。

```

#include<stdio.h>

int main() {
    long p,q,r;

    while (scanf("%ld %ld",&p,&q)
           !=EOF) {
        if (q>p) r=q-p;
        else r=p-q;

        printf("%ld\n",r);
    }
}

#include<iostream.h>
void main()
{
    long long a,b,c;

    while (cin>>a>>b) {
        if (b>a)
            c=b-a;
        else
            c=a-b;
        cout << c << endl;
    }
}

```

图 1.2 C (左)、C++ (中)、Pascal (右) 中的标准输入输出

### 1.3 编程提示

本书不是要教你如何编程，而是要教你如何更好地编程。我们假定你已经熟悉了一些基本概念，如变量 (variable)，条件语句 (例如 if-then-else, case)，循环语句 (例如 for-do, while-do, repeat-until)，子程序 (subroutine) 和函数 (function)。如果你对这些概念并不熟悉，也许你挑错了书。但我们仍然建议你把本书买下来备用。

你应当弄清楚你已经学会的东西到底有多大威力。在理论上，所有有趣的程序和算法都可以由你在第一门程序设计课程中学到的东西构建起来。现代编程语言的强大特性对于构建这些“有趣的东西”并不是必须的，它们只是让这个构建过程更清晰、更合理。

打个比方，成为优秀作家的关键不是持续地学习新词汇，而是努力寻找自己想说的。同样的，学习了一两门程序设计课程之后，你好比是已经学会了让别人能听懂自己所需的所有单词。剩下的问题正是本书所要努力解决的：找一些有趣的东西来说。

我们提供一些底层的编程技巧来帮助读者编写高质量的程序。下面将看到的例子均来自于评测系统接收到的实际提交代码。

- 先写注释 — 在写程序或者子程序之前首先用几句话描述你将要写出的代码应完成怎样的任务。这一点很重要，因为如果你不能轻易的写出这些描述性注释，你很可能并不十分清楚你到底想做什么。一般来讲，调试注释比调试程序要容易得多，而且这部分的时间花费是值得的。当然，在比赛现场的紧张环境和压力下，选手们往往不肯为写注释

而花时间，但在节省这一点微不足道的时间的同时，风险也随之而来。

- 给每个变量编写注释 — 在声明每个变量的时候写一个单行注释来表明它的用途。同样的道理，如果你无法轻易地描述这个变量，你甚至可能会怀疑这个变量存在的必要性。由于你很可能会花一点调试时间来与这个程序作进一步的近距离接触，用这个方法适度提高程序的可读性以减少调试时间是很划算的。
- 使用符号来表示常量 — 无论何时，只要你需要使用常量（输入规模、数学常量、数据结构大小等）时，就应在程序的开头部分以符号的形式定义它们。在程序中使用不一致的常量可能会引入极其隐蔽的错误。当然，你应当记得在程序中每个需要使用常量的地方引用相应的符号而不是这些常量本身，否则符号常量就失去了它们的意义 …
- 不要滥用枚举类型 — 枚举类型（即符号变量，例如布尔型（true, false））能提高复杂程序的可读性，但在短程序中往往是不必要的。下面的代码表示了一堆牌的花色，即草花（club）、方块（diamond）、红心（heart）和黑桃（spade）：

```
switch(cursuit) {  
    case 'C':  
        newcard.suit = C;  
        break;  
    case 'D':  
        newcard.suit = D;  
        break;  
    case 'H':  
        newcard.suit = H;  
        break;  
    case 'S':  
        newcard.suit = S;  
  
    ...
```

与原始的字符表示法（'C'，'D'，'H'，'S'）相比，枚举类型（C,D,H,S）的引入不仅没有提高可读性，反而带来了更多的出错机会。

- 用子程序来避免冗余代码 — 请读者阅读下面一段维护矩形棋盘状态的代码，思考你将如何缩短并简化它：

...

```
while (c != '0') {  
    scanf("%c", &c);  
    if (c == 'A') {  
        if (row-1 >= 0) {
```

```
    temp = b[row-1][col];
    b[row-1][col] = ' ';
    b[row][col] = temp;
    row = row-1;
}
}
else if (c == 'B') {
    if (row+1 <= BOARDSIZE-1) {
        temp = b[row+1][col];
        b[row+1][col] = ' ';
        b[row][col] = temp;
        row = row+1;
    }
}
...

```

在完整的程序中，一共有四个这样的代码段，每个代码段有三行，但作用仅仅是把一个值移动到某个相邻的单元格中。这段代码不仅长，而且把其中任何一个 + 或者 - 敲错都可能是致命的。相反，如果写一个“移动并交换”的子过程，只需要用正确的参数调用四次即可。

- 输出有意义的调试信息 — 花一点时间学习如何在你的系统中调试程序，例如在特定的语句或条件下中断程序运行，然后检查所有相关变量的值。一般来说这比加入大量输出语句的方法更快更简单，但如果你确实需要输出调试信息，应尽量让这些信息有意义。输出所有相关变量，并在打印数值之前标注相应的变量名，否则你将会很容易迷失在大量无意义的调试信息中。

大多数学习计算机科学的学生习惯于使用面向对象程序设计 (*object-oriented programming*)。尽管它在构建大型的、可重用的程序方面非常有用，但本书中的绝大多数程序只需要短小但精巧的程序即可解决。面向对象程序设计中的一些基本假定在本领域中并不成立，因此定义一些复杂的新对象（使用已有对象不算）一般来说是浪费时间。

请注意，我们并不是要大家摒弃良好的编程风格，而是提倡根据任务规模选择合适的编程方式。这才是成功的关键。

## 1.4 基本数据类型

数组与链表相比有一个重要的优势：简单。很多在基于指针的数据结构中经常出现的错误根本不可能在静态数组中出现。

职业选手的成熟标志是能保持问题的简单性。这对于刚刚接触一个新领域的人来说相当具有挑战性。有一个很经典的例子：在学习完一系列的热带疑难杂症之后，一个年轻的医学博士担心患有鼻塞和疹子的病人都有可能感染依波拉病毒 (Ebola virus) 或者淋巴腺鼠疫 (bubonic plague)，但有经验的医生只会给这样的病人开一瓶阿司匹林 (aspirin)。

同样地，你可能最近刚刚学会平衡排序二叉树 (balanced binary search tree)、异常处理 (exception handling)、并行处理 (parallel processing) 以及对象继承 (object inheritance) 的若干模型。这些都很有用、很重要，但并不见得是解决一个简单问题的最佳途径。

因此，尽管基于指针的数据结构在你无法预知最大输入规模，或者需要支持快速查找与更新的时候十分有用，但本书中的很多题目已经指明了最大规模，而且时限相当宽松。你的程序并不会因为特别快速而获得额外的奖励。

那么，什么才是简单、成熟的通往数据结构之路呢？首先，要熟悉程序设计语言中内置的原始数据类型。理论上，你可以用下述类型构造出任意的数据结构：

- 数组 — 数组允许我们按位置而非按内容访问数据，正如门牌号码让我们按照地址而非户主名字来访问。数组用来存放单一类型的元素序列，如整数，实数或者记录 (record)<sup>3</sup> 等复合对象。字符数组可以用来表示文本串，而文本串数组几乎可以用来表示任何东西。

哨兵 (*sentinel*) 是一个用来简化数组访问的常见技巧。哨兵是一个特殊元素，在不用显式测试下标范围的情况下保证程序不会访问到非法地址。考虑在一个包含  $n$  个元素的有序数组  $a$  中插入元素  $x$  的情形。我们可以像左边的代码那样显式的测试下标是否到达最大允许值：

```
i = n;                                i = n;
while ((a[i]>=x) && (i>=1)) {          a[0] = SMALLINT
    a[i+1] = a[i];                      while (a[i] >= x) {
    i=i-1;                            a[i+1] = a[i];
}                                         i=i-1;
a[i+1] = x;                           }
                                         a[i+1] = x;
```

也可以像右边的代码那样在  $a[0]$  处设置一个保证比所有元素小的整数  $SMALLINT$  作为哨兵。合理的使用哨兵并确保数组的大小略大于它实际被使用的部分往往可以避免很多边界错误。

- 多维数组 — 尽管在提到二维数组时，人们总是会不自觉地想到棋盘、图像这样的矩形网格结构，但更一般地，多维数组可以用来聚集同构 (*homogeneous*) 数据。例如， $x - y$  平面上的  $n$  个点所组成的数组可以被看作一个  $n \times 2$  数组，其中  $A[i][j]$  中的第二维 (0 或 1) 表示该元素是相应点的  $x$  坐标还是  $y$  坐标。

<sup>3</sup> 译者注：这是 Pascal 语言的概念，在 C/C++ 语言中对应于结构体。

- 记录 — 记录用来聚集异构 (*heterogeneous*) 数据。例如人事记录把名字、身份证号码、身高和体重捆绑到一起。在大程序中，记录往往对整个程序的概念清晰性起到了重要作用，但在中等规模的程序中也许把各个域分解成多个独立的数组会更加方便。

用记录还是用多维数组，哪个更好呢？这个问题的答案并不总是那么清晰。还是回到刚才所讨论过的  $x - y$  平面上点的表示。最显然的表示莫过于下面的记录（结构体），而不是一个双元素数组：

```
struct point {  
    int x, y;  
};
```

记录表示法的主要优势在于  $p.x$  和  $p.y$  看上去更加接近于我们的日常用法，但问题在于你无法方便地遍历一个记录中的各个变量。而在数组中，遍历元素是非常方便的。

假设你准备把程序扩展到支持三维、甚至任意维空间中。当然，你可以简单地在记录中增加一个域，但接下来你还需要修改几乎所有的函数，把  $x$  和  $y$  分量做的事情对于  $z$  分量重复一遍。另一方面，如果你采用的是数组表示法，则把距离函数从二维改到三维仅需修改一个常量：

```
typedef int point[DIMENSION];  
  
double distance(point a, point b)  
{  
    int i;  
    double d=0.0;  
  
    for (i=0; i<DIMENSION; i++)  
        d = d + (a[i]-b[i]) * (a[i]-b[i]);  
  
    return( sqrt(d) );  
}
```

在第 2 章中，我们将会看到一些相对高级的数据结构，它们都可以由这些原始数据类型构造出来。这些数据结构将让我们工作在一个更高的抽象层次中，但当简单数据类型完全可以胜任时，不要害怕使用它们。

## 1.5 关于习题

本书每章的末尾均有一个富有挑战性的编程题集。这些题目都是从 Valladolid 评测系统的上千道题目中精选出来的清晰而精巧的、难度各异的问题。我们尤其关注这些题目中闪现

出的智慧的火花，是它们让这些题目成为了真正的挑战。

我们所选的每个问题的描述均经过了编辑，在尽量保持原文风格的同时提高了准确性和可读性。我们同时给出了每道题目在两个评测系统中各自的题目编号，它们是成功提交的重要保障。第一个编号是<http://www.programming-challenges.com>上的，而第二个编号是<http://uva.onlinejudge.org/>上的。

为了帮助读者对题目的相对难度有一个更好的了解，每道题目用三种不同的方式进行标注。首先，每题用 A、B、C 三个等级衡量正确提交的数量。A 类题目往往更加容易解决，或者比 B 类题目更吸引人。第二，提交通过率分为高 (high)、中 (average) 和低 (low) 三种。低通过率可能是因为过分苛刻的测试数据，或者题目需要用到一些精妙的算法，甚至可能是数据曾经出现过错误。因此，建议读者不要过分地关注通过率。最后，我们给出一个从 1 到 4 的主观评价，表示解题所需的学术水平。数值越大，表示这个问题越复杂。

祝大家好运，“切题”愉快！

## 1.6 习 题

### 1.6.1 $3n + 1$ 问题 (3n+1 Problem)

PC/UVa 题号: 110101/100, 流行度: A, 通过率: low 难度: 1

考虑如下的序列生成算法：从整数  $n$  开始，如果  $n$  是偶数，把它除以 2；如果  $n$  是奇数，把它乘 3 加 1。用新得到的值重复上述步骤，直到  $n = 1$  时停止。例如， $n = 22$  时该算法生成的序列是：

22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1

人们猜想（没有得到证明）对于任意整数  $n$ ，该算法总能终止于  $n = 1$ 。这个猜想对于至少 1 000 000 内的整数都是正确的。

对于给定的  $n$ ，该序列的元素（包括 1）个数被称为  $n$  的循环节长度。在上述例子中，22 的循环节长度为 16。输入两个数  $i$  和  $j$ ，你的任务是计算  $i$  到  $j$ （包含  $i$  和  $j$ ）之间的整数中，循环节长度的最大值。

#### 输入

输入每行包含两个整数  $i$  和  $j$ 。所有整数大于 0，小于 1 000 000。

#### 输出

对于每对整数  $i$  和  $j$ ，按原来的顺序输出  $i$  和  $j$ ，然后输出二者之间的整数中的最大循环节长度。这三个整数应该用单个空格隔开，且在同一行输出。对于读入的每一组数据，在输出中应位于单独的一行。

**样例输入**

1 10  
100 200  
201 210  
900 1000

**样例输出**

1 10 20  
100 200 125  
201 210 89  
900 1000 174

**1.6.2 扫雷 (Minesweeper)**

PC/UVa 题号: 110102/10189, 流行度: A, 通过率: high 难度: 1

你玩过扫雷游戏吗? 这个有趣的小游戏来自于某个快被忘却的操作系统。游戏的目标是找出一个  $n \times m$  矩阵内的所有地雷。在本题中, 你需要为每个单元格统计出它周围的地雷个数。每个单元格最多有 8 个相邻的单元格。左图的  $4 \times 4$  矩阵有两个地雷, 用 “\*” 表示。计算结果为右边的矩阵:

*...	*100
....	2210
.*..	1*10
....	1110

**输入**

输入将包含若干个矩阵。对于每一个矩阵, 第一行包含两个数字  $n$  和  $m(0 < n, m \leq 100)$ , 分别代表这个矩阵的行数和列数。接下来的  $n$  行每行包含  $m$  个字符, 即该矩阵。安全格子用 “.” 表示, 有地雷的格子用 “\*” 表示。当  $n = m = 0$  时, 表示输入结束。你的程序不应处理这一行。

**输出**

对于每一个矩阵, 首先在单独的一行里打印序号: Field # $x$ : 其中  $x$  是数据编号, 从 1 开始。接下来的  $n$  行中, 读入的 “.” 应被该位置周围的地雷数所代替。输出的每两个相邻矩阵必须用一个空行隔开。

**样例输入**

4 4  
\*...  
....  
.\*\*.  
....  
3 5

**样例输出**

Field #1:  
\*100  
2210  
1\*10  
1110

```
**...
.....
.*...
0 0
Field #2:
**100
33200
1*100
```

### 1.6.3 旅行 (The Trip)

PC/UVa 题号: 110103/10137, 流行度: B, 通过率: average 难度: 1

有一个旅行俱乐部每年组织学生到不同的地方旅行。俱乐部成员所去过的地方包括印地安那波利斯、菲尼克斯、纳什维尔、费城、圣何塞和亚特兰大。今年春天他们还准备到爱因霍芬旅行。

他们一开始就说好要平摊所有费用，但是每次花费都平摊是不现实的。因此，每个学生先为某些特定的项目（例如吃饭、住酒店、打车和购买机票）垫付一些钱，当旅行结束后根据每个学生垫付的金额多退少补，使得每个人的支出差距在 1 分钱以内。在过去，重新分配钱币是一件乏味并且耗时的工作。你的任务是根据一份清单计算出为了平摊费用（差距在 1 分钱以内）所必须的最小总“交易”金额。

#### 输入

输入将包括若干组旅行的数据。每一组数据的第一行为一个正整数  $n$ ，代表这次旅行中的学生人数。接下来的  $n$  行每一行包含了一个学生的支出，精确到分。学生人数不超过 1000，并且每个学生的支出不超过 \$1 000.00。在最后一组数据结尾后还有单独的一行，包含一个 0。

#### 输出

对于每一组数据，只输出一行，即让每个学生平摊支出所需的最小总“交易”金额，以美元计，精确到分。

#### 样例输入

```
3
10.00
20.00
30.00
4
15.00
15.01
3.00
3.01
0
```

#### 样例输出

```
$10.00
$11.99
```

#### 1.6.4 液晶显示屏 (LC-Display)

PC/UVa 题号: 110104/706, 流行度: A, 通过率: average 难度: 1

你的一个朋友刚买一台新电脑, 而在此之前他用过的最强大的机器是一个袖珍式计算器。他有点失望, 因为和电脑屏幕相比, 他更喜欢袖珍式计算器的液晶显示屏。为了让他高兴, 你要写一个程序让电脑屏幕像液晶屏一样显示数字。

#### 输入

输入包含若干行, 每一行包含两个整数  $s$  和  $n$ , 其中  $n(0 \leq n \leq 99\ 999\ 999)$  是需要显示的数,  $s(1 \leq s \leq 10)$  是显示的大小。当某一行只包括两个 0 时, 输入终止, 这两个 0 不必处理。

#### 输出

模拟液晶屏显示输入文件中的指定整数  $n$ 。用  $s$  个 “-” 表示水平线段,  $s$  个 “|” 表示竖直线段。每一个阿拉伯数字占用  $s+2$  列和  $2s+3$  行。注意那些数字中的空白之处要填上空格, 并且两个数字之间必须有一个空列。在每一个整数后面输出一个空行。你将在下面的输出样例中看到每个阿拉伯数字的输出方法。

#### 样例输入

```
2 12345
3 67890
0 0
```

#### 样例输出

```
-- -- --
| | | | |
| | | | |
-- -- --
| | | | |
| | | | |
-- -- --
--- --- ---
| | | | |
| | | | |
| | | | |
--- --- ---
| | | | |
| | | | |
| | | | |
--- --- ---
```

### 1.6.5 图形化编辑器 (Graphical Editor)

PC/UVa 题号: 110105/10267, 流行度: B, 通过率: low 难度: 1

正如文本编辑器允许我们修改文本一样，Photoshop 这样的图形化编辑器允许我们修改位图图像。在本题中，图像被表示成一个  $M \times N$  的像素数组，每个像素有一种颜色。

你的任务是编写一个程序，模拟一个简单的交互式图形化编辑器。

输入

输入包括一个指令序列，每条指令占单独的一行。每行的第一个字符总是大写字母，代表这条指令的类型。如果该指令需要参数，将会在同一行中陆续给出，用空格隔开。

像素坐标用列号和行号这两个整数表示，列号在 1 到  $M$  范围内，行号在 1 到  $N$  范围内 ( $1 \leq M, N \leq 250$ )。坐标原点在左上角。颜色用大写字母表示。

编辑器接受以下命令：

I M N	创建一个新的 $M \times N$ 图像, 所有像素为白色 (0)。
C	清除图像, 把所有像素涂成白色 (0), 图像大小不变。
L X Y C	把像素 $(X, Y)$ 涂成颜色 (C)。
V X Y1 Y2 C	画一条竖直线段, 颜色为 (C), 列号为 X, 从第 Y1 行到第 Y2 行 (包括 Y1 和 Y2 行)。
H X1 X2 Y C	画一条水平线段, 颜色为 (C), 行号为 Y, 从第 X1 列到第 X2 列 (包括 X1 和 X2 列)。
K X1 Y1 X2 Y2 C	画一个实心矩形, 颜色为 (C), 左上角为 $(X_1, Y_1)$ , 右下角为 $(X_2, Y_2)$ 。
F X Y C	把区域 $R$ 填充为颜色 (C)。 $R$ 定义如下: 若像素 $(X, Y)$ 属于 $R$ , 则所有与 $(X, Y)$ 有公共边且颜色相同的像素也属于 $R$ 。
S NAME	原样输出文件名以及整幅图像的像素矩阵。
X	退出程序。

输出

对于每条 `S NAME` 指令，原样输出文件名 `NAME` 和当前图像的内容。输出的每一行表示图像每一行中各个像素的颜色，参见样例输出。

如果指令的第一个字符不是 I、C、L、V、H、K、F、S、X之一，忽略此命令，直接跳到下一

行输入进行处理。

### 样例输入

```
I 5 6
L 2 3 A
S one.bmp
G 2 3 J
F 3 3 J
V 2 3 4 W
H 3 4 2 Z
S two.bmp
X
```

### 样例输出

```
one.bmp
00000
00000
0A000
00000
00000
00000
two.bmp
JJJJJ
JJZZJ
JWJJJ
JWJJJ
JJJJJ
JJJJJ
```

### 1.6.6 解释器 (Interpreter)

PC/UVa 题号: 110106/10033, 流行度: B, 通过率: low 难度: 2

某台电脑有 10 个寄存器和 1000 个内存单元的存储器。每一个寄存器或者内存单元可以保存一个 0 到 999 的三位十进制整数。指令被编码为三位十进制数字并存储在内存单元中。编码规则如下:

100	表示停机
2dn	表示将寄存器 $d$ 的值设为 $n$ (0 到 9)
3dn	表示给寄存器 $d$ 的值增加 $n$
4dn	表示给寄存器 $d$ 的值乘以 $n$
5ds	表示将寄存器 $s$ 的值拷贝到寄存器 $d$
6ds	表示将寄存器 $s$ 的值加到寄存器 $d$
7ds	表示将寄存器 $s$ 的值乘到寄存器 $d$
8da	表示将地址为“寄存器 $a$ 的值”的内存单元的值拷贝到寄存器 $d$
9sa	表示将寄存器 $s$ 的值拷贝到地址为“寄存器 $a$ 的值”的内存单元
0ds	表示若寄存器 $s$ 的值不为 0, 则跳转到地址为“寄存器 $d$ 的值”的内存单元处继续执行

所有寄存器初始值都为 000。存储器的初始值将从输入数据读入。第一条指令总是在存储器地址 0 处执行。所有运算结果都为除以 1000 后的余数。

输入

输入第一行仅为一个正整数  $n$ , 表示测试数据组数。然后是一个空行, 接下来是数据。每两组测试数据间均有一个空行。

每组测试数据包含最多 1000 个三位非负整数，代表从地址 0 开始的存储器的值。未指定的其他内存单元初始化为 000。

输出

对于每行测试数据输出一个整数：停机之前执行的总指令数（包括停机指令本身）。假设程序总是会终止。相邻两组数据间用单个空行隔开。

### 样例输入

1

299

492

495

399

492

495

399

283

279

689

078

100

000

600

888

## 样例输出

16

### 1.6.7 将军 (Check the Check)

PC/UVa 题号: 110107/10196, 流行度: B, 通过率: average 难度: 1

你的任务是写一个程序，读入一个国际象棋棋局并且判断棋盘上的王是否被攻击（被将军）。如果王所处的位置能在一步之内被对手占领，我们称这个王被“将军”了。

大写字母代表白子，小写字母代表黑子。白方总是位于棋盘下部，而黑方总是位于棋盘上部。

如果你不熟悉国际象棋，下面是不同棋子的走子方式：

**卒 (p 或 P)**：笔直向前移动，一次一格。但在吃子时，只能吃掉斜前方一格的棋子。在这个问题中你只用考虑吃子的情况。

**马 (n 或 N)**：像中国象棋的马一样走“日”字形。只有马可以越过其他棋子（没有中国象棋中蹩腿的规则）。

**象 (b 或 B)**：可以沿对角线向前、向后走任意格。

**车 (r 或 R)**：可以沿水平或竖直线向前、向后走任意格。

**后 (q 或 Q)**：可以沿水平、竖直线或者对角线向任意方向走任意格。

**王 (k 或 K)**：可以沿水平、竖直线或者对角线向任意方向走一格。

各种棋子移动方法如下图所示，“\*”代表棋子一步之内可以占领的格子。

卒	车	象	后	王	马
.....	...*....	.....*	...*...*	.....	.....
.....	...*....	*.....*	*...*...*	.....	.....
.....	...*....	.*....*	.**.*..	.....	..*.*...
.....	...*....	...*.*...	..***...	...***...	.*...*..
...p....	***r****	...b....	***q****	..*k*...	..n....
...*.*...	...*....	...*.*...	...***...	...***...	.*...*..
.....	...*....	.*....*	.**.*..	.....	..*.*...
.....	...*....	*.....*	*...*...*	.....	.....

注意只有马可以越过其他棋子。卒的移动方向取决于它属于哪一方：黑方的卒只能向下吃掉对角线方向的子，白方的卒只能向上吃掉对角线方向的子。上面的示例是黑方的卒，用小写“p”表示。

## 输入

输入包含多个棋盘，每个棋盘由 8 行 8 列字母表示。“.” 代表没有棋子，大小写字母按照前面的定义表示不同棋子。不会出现任何无效字符，也不会出现双方的王同时被将的情况。输入数据最后为一个仅包含“.” 的空棋盘（不需处理），所有其他棋盘有且仅有一个黑王和一个白王。

## 输出

对读入的每个棋盘，输出如下信息之一 ( $d$  代表从 1 开始的局面编号)，分别表示白王被将军、黑王被将军和双方均未被将军：

Game #d: white king is in check.

Game #d: black king is in check.

Game #d: no king is in check.

### 样例输入

..k.....

ppp.pppp

.....

.R...B..

.....

.....

PPPPPPPP

K.....

rnbqk.nr

ppp..ppp

...p...

..p....

.bPP....

.....N..

PP..PPPP

RNBQKB.R

### 样例输出

Game #1: black king is in check.

Game #2: white king is in check.

### 1.6.8 澳大利亚投票 (Australian Voting)

PC/UVa 题号: 110108/10142, 流行度: B, 通过率: low 难度: 1

澳大利亚投票系统要求选民们将所有候选人按愿意选择的程度排序，一张选票就是一个排序。一开始，每张选票的首选项将被统计。若有候选人得票超过 50%，他将直接胜出；否则，所有并列得票最低的候选人出局，而那些将出局候选人排在第一位的选票将被重新统计为排名最高的未出局候选人。这一筛选过程将持续进行，直到某个候选人得到超过 50%

的选票，或所有候选人得票相同。

### 输入

输入第一行包含一个正整数，代表测试数据组数。然后是一个空行，接下来是数据。每两组测试数据间用空行隔开。

每组数据第一行为一个整数  $n \leq 20$ ，代表候选人数目。接下来  $n$  行按顺序给出每个候选人的名字，最长 80 个字符，可以是任何可打印字符。接下来是至多 1000 行，每行描述一张选票的内容。每张选票包含 1 到  $n$  的某一排列。第一个为首选，第二个次选，以此类推。

### 输出

对于每组数据，输出数据可能是单独一行，即胜出者的名字；或者若干行，分别包含所有平局的候选人。每两组数据间用一个空行隔开。

#### 样例输入

```
1
3
John Doe
Jane Smith
Jane Austen
1 2 3
2 1 3
2 3 1
1 2 3
3 1 2
```

#### 样例输出

```
John Doe
```

1.3 如果无法完全平均分配，哪些人应该得到多余的钱？

1.5 应该如何处理F命令？同时保留新老版本的图像是否会简化问题？

## 1.8 注解

1.1  $3n+1$  问题（又称 Collatz 问题）至今仍悬而未决。[Lag85] 是一篇不错的数学调研。1999 年还召开了一次 Collatz 问题的国际会议，读者可以在 <http://www.math.grinnell.edu/chamberl/conf.html> 找到该会议的相关信息。

- 1.2 扫雷一致性问题 (the minesweeper consistency problem) 输入一个  $m \times n$  矩形网格, 每个单元格要么被标记 0 到 8 的整数, 要么被标记为有雷, 要么被标记为未知, 要求判断是否存在一个布局满足所有这些标记。Clay 数学研究所 (<http://www.claymath.org/>) 为该问题的有效算法提供 \$1 000 000 的奖金。

但请不要太兴奋。该问题已经被证明是 NP- 完全的 [Kay00], 也就是说在现有的计算理论发生革命性变化之前不会找到该问题的有效算法。关于 NP- 完全性的更多讨论参见 [GJ79]。

1.6 在 Java 这样的语言中, 用软件解释的虚拟机是可移植性的关键。为诸如 PDP-8 这样老式、已废弃、但结构简单的处理器编写机器语言模拟器是一件有趣的事。更有意思的是, 在高速硬件的帮助下, 你写出来的虚拟 PDP-8 将比原始的机器快很多!

1.7 当你写出一个合法走步生成器 (这正是本题的核心) 之后, 你的程序离“自动下棋”已经不远了! 关于计算机程序如何自动下国际象棋 (chess) 和西洋跳棋 (checkers), 并战胜人类世界冠军的故事参见 [New96, Sch97]。

1.8 投票系统的数学分析是一个引人入胜的课题。阿罗不可能性定理 (Arrow's impossibility theorem) 说明了任何一个投票系统都不可能同时具备五个直观上“显然应该有”的特性。关于社会选择理论的更多有趣的数学讨论参见 [COM94]。

## 第2章 数据结构

数据结构是复杂算法的核心。数据结构的选择会对算法实现的复杂性产生巨大的影响。选择了正确的数据结构，编程会十分容易；选择了错误的数据结构，则需要大量的时间和代码量作为决策失误的代价。

在本章中，你将复习到一些每个程序员都应熟悉的基础数据结构。我们将以一个孩子们喜欢的扑克牌游戏作为背景展开讨论。很多经典的编程题目都是以游戏为背景的。几乎所有人在初学编程的课程中都会接触到汉诺塔 (Hanoi Tower)、骑士周游、八皇后这样的游戏。

### 2.1 基本数据结构

我们首先介绍栈 (stack)、队列 (queue)、字典 (dictionaries)、优先队列 (priority queues)、集合 (sets) 等最重要的数据结构的抽象操作 (abstract operations)，接下来简单描述从头实现这些操作的最简单的方法。

请注意，C++ 和 Java 这样的现代面向对象程序设计语言都已经在它们的标准库中实现了基础数据结构。我们将在 2.2 节中简单地介绍它们。每个程序员都应该花一些时间来熟悉这些数据结构，而不是每次都从头实现。当你很好地熟悉了这些库的使用方法后，在阅读本节时便可专注于这些数据结构所擅长的领域而非实现细节。

#### 2.1.1 栈

在栈和队列中，元素被取出的顺序取决于它们的插入顺序，而不是这些元素的内容。栈 (stack) 满足后进先出 (*last-in, first-out, LIFO*) 的特性。栈的抽象操作包括：

- $Push(x,s)$  — 在栈  $s$  的顶部插入元素  $x$ 。
- $Pop(s)$  — 返回并删除栈  $s$  顶部的元素。
- $Initialize(s)$  — 创建一个空栈。
- $Full(s), Empty(s)$  — 测试栈是否满 (无法继续 Push) 或者空 (无法继续 Pop)。

注意标准的栈和队列均不支持元素的查找操作。

抽象操作的定义帮助我们在不了解实现细节的情况下使用和重用栈模块。最简单的实现是用一个数组 (array)，加上一个变量来记录当前栈顶元素的下标。栈还可以用链表 (linked lists) 实现，好处是只要内存足够就不会溢出 (overflow)。

栈可以用来表示成堆物体，例如一叠晚餐用的盘子。新洗好的盘子被放到栈顶，而有人饿了的时候可以直接从顶部取出一个干净的盘子来用。在这个例子中栈是合适的，因为这些

盘子的顺序是无关紧要的。因此，栈的一个重要应用是作为顺序无关元素的容器，因为它的实现特别简单。

栈顺序对于嵌套结构 (nested structure) 的处理十分重要，包括带括号的公式 (parenthesized formula)(遇“(”入栈，遇“)”出栈)，递归程序调用 (过程进入时入栈，过程退出时出栈)，图的深度优先遍历 (depth-first traversal)(发现新结点时入栈，最后一次离开结点时出栈)。

### 2.1.2 队列

队列 (*Queues*) 满足先进先出 (*first-in, first-out, FIFO*) 顺序。它看上去比后进先出更加公平，因此超市收银台前面的人排成队列而不是栈。扑克牌堆也可以用队列来建模，因为我们通常在牌堆的顶部拿牌，然后放到最底部。在第 9 章中，FIFO 队列被用来实现图的宽度优先遍历 (breadth-first search)。队列的抽象操作包括：

- *Enqueue*(*x,q*) — 在队列 *q* 的末尾插入元素 *x*。
- *Dequeue*(*q*) — 返回 (并删除) 队列 *q* 的首部元素。
- *Initialize*(*q*), *Full*(*q*), *Empty*(*q*) — 和栈的同名操作类似。

在实现上，队列比栈要困难一点，因为首尾两端均可以操作。最简单的实现方法是采用一个数组 (array)，直接在一端进行插入，并采用移动元素的方法填补因删除而出现的空挡。

然而，每次删除均移动所有元素是一个极大的浪费。如何才能做得更好呢？维护两个下标，分别表示第一个元素 (*first*) 和最后一个元素 (*last*)<sup>1</sup>，则插入和删除均不涉及元素移动操作，只需修改相应的下标。我们完全没有必要显式的清理以前用过的内存单元，但这样的实现方法导致元素出队以后留下了一串无法重用的空间。

环形队列 (circular queues) 较好地解决了这个问题。注意队首和队尾两个指针好比在环形的跑道上赛跑，当队首超过队尾一圈以后 (此时队满)，两个指针的位置关系和队空时完全一样！换句话说，无法仅通过两个指针的当前位置区分开队空和队满，需要借助其他方法。其中最简单的方法是另外设置一个变量记录队列中的元素个数。

```
typedef struct {
    int q[QUEUESIZE+1];           /* body of queue */
    int first;                    /* position of first element */
    int last;                     /* position of last element */
    int count;                    /* number of queue elements */
} queue;

init_queue(queue *q)
```

<sup>1</sup> 译者注：另一个方法是记录队首 *head* 和队尾 (最后一个元素的后一个位置)*tail*，则在逻辑上元素所处的区间是 [*head,tail*]。

```

{
    q->first = 0;
    q->last = QUEUESIZE-1;
    q->count = 0;
}

enqueue(queue *q, int x)
{
    if (q->count >= QUEUESIZE)
        printf("Warning: queue overflow enqueue x=%d\n",x);
    else {
        q->last = (q->last+1) % QUEUESIZE;
        q->q[ q->last ] = x;
        q->count = q->count + 1;
    }
}

int dequeue(queue *q)
{
    int x;

    if (q->count <= 0) printf("Warning: empty queue dequeue.\n");
    else {
        x = q->q[ q->first ];
        q->first = (q->first+1) % QUEUESIZE;
        q->count = q->count - 1;
    }

    return(x);
}

int empty(queue *q)
{
    if (q->count <= 0) return (TRUE);
    else return (FALSE);
}

```

}

队列是链表比数组更加容易实现的少数数据结构之一，因为可以省略对队空和队满的特殊判断。

### 2.1.3 字典

字典 (*dictionaries*) 根据内容读写数据，而不像栈和队列那样根据元素的位置。字典支持三个主要操作：

- $Insert(x, d)$  — 在字典  $d$  中插入元素  $x$ 。
- $Delete(x, d)$  — 在字典  $d$  中删除元素  $x$  或者  $x$  所指向的元素。
- $Search(k, d)$  — 查找关键字为  $k$  的元素。如果成功，则返回该元素。

数据结构课程会讲授很多种实现字典的方法，包括无序链表和有序链表、无序数组和有序数组、伸展树 (splay tree)、AVL 树 (AVL tree)、红黑树 (red-black tree) 以及各种散列 (hashing) 技术。

算法分析 (algorithm analysis) 主要关注时空性能 (performance)，即达到三种操作代价的最佳平衡 (trade-off)。但在实际中，我们更希望在满足运行时间限制的前提下采用最简单的方法。解决问题的最佳途径取决于在执行过程中该数据结构将如何变化。

- 静态字典 — 静态字典 (static dictionaries) 一旦建立，永不更改。因此它将只支持查找操作，而不支持插入和删除。

静态字典通常用数组来实现，唯一的问题是要不要将数据排序，以便使用二分查找。为了让程序尽量简单，除非时限特别紧，否则只有在  $n > 100$  时才值得使用二分查找。如果查找操作不会执行很多次的话，甚至在  $n = 1000$  或者更大的情况下，无序数组上的顺序查找就足够解决问题。

排序和二分查找几乎总是比人们想象的更加难以调试。C、C++ 和 Java 语言都提供了排序和查找的库函数，参见第 4 章。

- 半动态字典 — 半动态字典 (semi-dynamic dictionaries) 支持插入和查找，但不支持删除。仍然可以用线性表实现半动态字典。如果事先知道被插入元素的上限，我们可以用数组实现它，否则只能用链表。

哈希表 (hash tables) 能够很好地实现字典，特别是在不需要支持删除操作的时候。它的基本思想是设计一个函数，根据键值直接算出元素在某数组中的具体位置，而不需要访问其他元素。为了防止表过大，我们必须允许冲突 (*collisions*)，即多个元素可以被映射到同一位置。

使用哈希表的两个步骤是：(1) 定义哈希函数 (hash function)，把键值映射到一个特定范围内的整数；(2) 申请一个至少和该范围一样大的数组，使得哈希函数的值可以作为数组下标使用。

基本的哈希函数把键值转化成整数，然后对哈希表的大小取模。哈希表的大小一

般取素数 (至少不要取 1000 这样明显的合数), 这可以避免一些潜在的问题。字符串可以看成是一个基数为字母表大小 (alphabet-size) 的进位制系统。例如  $e$  是第 5 个字母,  $s$  是第 19 个字母,  $t$  和  $v$  分别是第 20 和 22 个字母, 因此字符串 “steve”  $\Rightarrow 26^4 \times 19 + 26^3 \times 20 + 26^2 \times 5 + 26^1 \times 22 + 26^0 \times 5 = 9\,038\,021$ 。快速取模的技巧将在第 7 章中讨论。

在不需要删除的情况下, 开放地址法 (*open addressing*) 是一个优秀而容易实现的冲突解决策略。当冲突出现后, 开放地址法使用一个简单的规则计算下一个尝试的位置 (例如总是把元素放到下一个空位), 而查找的时候只需从哈希函数对应的位置开始线性查找。如果在找到元素之前发现空位, 则元素不存在。

当使用开放地址法时, 元素的删除变得非常困难, 因为直接删除元素将破坏插入元素时生成的链条关系, 从而让一些元素不可访问。高效的关键是申请一个足够大的、包含很多“空洞”的哈希表。请不要吝啬这些空间, 否则你将付出巨大的性能代价。

- 全动态字典 — 哈希表也可以很好地实现全动态字典 (fully dynamic dictionaries), 但需要用链地址法 (*chaining*) 来解决冲突。我们为每个表项设立一个链表, 插入、删除和查询都可以方便的完成。如果哈希函数足够优秀, 键值的哈希函数值将在表里均匀分布, 因此每个链表都会足够小, 使得三种操作都能快速完成。

#### 2.1.4 优先队列

优先队列 是一个支持以下三种操作的元素集合:

- $Insert(x,p)$  — 在优先队列  $p$  中插入元素  $x$ 。
- $Maximum(p)$  — 返回优先队列  $p$  中最大的元素。
- $ExtractMax(p)$  — 返回并删除优先队列  $p$  中的最大元素。

我们经常用优先队列来维护日程表 (schedule) 或者日历 (calendar), 因为它们可以方便地计算出下一个事件发生的时刻。

优先队列还用在计算几何中常用的扫描算法 (*sweep-line algorithms*) 中。典型的方法是以  $x$  坐标为优先级, 用优先队列储存还没有遇到过的点, 然后离散地移动扫描线。

最著名的优先队列实现莫过于二叉堆 (binary heap) 了, 它可以通过结点的上调、下调操作来高效的实现。二叉堆巧妙而高效, 但在时间紧迫的比赛中还是容易写错。当插入次数不太多的情况下也可以考虑用有序数组来代替。

#### 2.1.5 集合

集合 (更恰当的说法是子集 (*subsets*)) 是由给定全集  $U$  的若干元素组成的无序集。它们和字典的区别在于至少有一种隐式的需求来对不在该子集中的  $U$  的元素进行编码。

子集的基本操作有:

- $Member(x,S)$  — 元素  $x$  是子集  $S$  中的元素吗?

- $\text{Union}(A, B)$  — 返回子集  $A \cup B$ ，即所有在  $A$  中或者在  $B$  中的元素所组成的  $U$  的子集。
- $\text{Intersection}(A, B)$  — 返回子集  $A \cap B$ ，即所有在  $A$  中且在  $B$  中的元素所组成的  $U$  的子集。
- $\text{Insert}(x, S), \text{Delete}(x, S)$  — 在子集  $S$  中插入/删除元素  $x$ 。

当全集太大甚至无穷时，一个简单的方法是用字典来实现子集。如果字典是基于比较的<sup>2</sup>，则并集 (union) 和交集 (intersection) 都可以用类似有序表合并的算法来实现：在合并表中至少出现过一次的元素属于并集；恰好出现两次的属于交集。

当全集很小而且不会变化时，位向量 (bit vectors) 是一个很方便的表示方法。若全集包含  $n$  个元素，则  $n$  位向量的第  $i$  个位等于 1 当且仅当  $i \in S$ 。插入和删除只需要翻转相应的位 (bit)，而交集和并集只需要计算两个位向量的“按位与”(bitwise AND) 和“按位或”(bitwise OR)。位向量表示法是相当紧凑的，例如只需要 1000 个 32 位整数即可表示一个包含 32 000 个元素的  $U$  的任意子集。<sup>3</sup>

## 2.2 库 函 数

C++ 和 Java 等现代的面向对象程序设计语言在它们的标准库中实现了这些基础数据结构。

### 2.2.1 C++ 标准模板库

C 函数库中很难有栈和队列这样的通用数据结构 (general-purpose data structures)，因为在 C 语言中我们无法用编程的方式自动获取参数的数据类型。换句话说，我们只能分别定义像 `push_int()` 和 `push_char()` 这样的函数。所有的数据类型都必须在编译时确定。不仅每个可能的数据类型都要定义一次，而且还无法照顾到用户自定义类型。

模板 (*templates*) 是 C++ 中的重要机制，它可以定义针对通用数据类型的类或者函数，只在编译时将它们参数化成具体的类型。C++ 标准模板库 (*Standard Template Library, STL*) 除了提供了刚才所提到的所有数据结构外，还包含了很多其他实用的类和函数。下面的例子：

```
#include <stack>
using namespace std;
stack<int> S;
stack<char> T;
```

<sup>2</sup> 译者注：有序表或者排序二叉树。但哈希表不是基于比较的。

<sup>3</sup> 译者注：如果只有 50 个元素，则一个 64 位整数便足够了，在很多时候这是很方便的。空间不那么紧张时也可以用数组来代替位向量，每个元素而不是每个位非 0 即 1。尽管浪费了空间，但换来了程序的简洁性。

声明了两个栈，各自采用不同的数据类型。

推荐的 STL 参考资料包括：[MDS01] 和 <http://www.sgi.com/tech/stl/>。下面是其中一些最常用的数据结构的简单描述：

- 栈 — 方法包括 `S.push()`, `S.top()`, `S.pop()` 和 `S.empty()`。`top` 函数返回栈顶元素但不删除它，而 `pop` 函数删除栈顶元素但不返回，因此你应当先 `top` 再 `pop`[Seu63]。它的内部结构确保了它永远不会满（只要内存足够）。
- 队列 — 方法包括 `Q.front()`, `Q.back()`, `Q.push()`, `Q.pop()` 和 `Q.empty()`，功能与栈类似。
- 字典 — STL 包含许多种字典类容器，包括 `hash_map`，一个用哈希表实现的键 – 值映射<sup>4</sup>。方法包括 `H.erase()`, `H.find()` 和 `H.insert()`。
- 优先队列 — 用 `priority_queue<int> Q;` 来定义，方法包括：`Q.top()`, `Q.push()`, `Q.pop()` 和 `Q.empty()`。
- 集合 — 集合用基于比较的关联容器 (*associative containers*) 表示，声明方法为 `set<key, comparison> S;`。集合算法包括 `set_union` 和 `set_intersection`，还有其他一些标准集合运算。

## 2.3 程序设计实例：纸牌大战

在一个称为“纸牌大战”(war) 的儿童游戏中，一副标准的 52 张牌被发给两个游戏者 (1 和 2)，每人 26 张。双方都不能看自己的牌，而只能将它们正面朝下摞成一堆。游戏的目标是赢得所有的牌。

游戏开始时，两个游戏者同时把他们各自的第一张牌从自己的牌堆顶部拿出，正面朝上放在桌子上。点数较高的人把两张牌都拿走，正面朝下放到他自己的牌堆底部。点数从大到小依次为：*A*、*K*、*Q*、*J*、*T*、9、8、7、6、5、4、3、2。花色在本游戏中没有意义。双方重复这个过程，直到一方赢得所有的牌，游戏结束。

如果双方取出的牌相等，则称为出现战争 (*war*)。双方再各取出两张牌放在桌子上，第一张正面朝下，第二张正面朝上，并且比较第二张牌的点数。如果点数仍然相同则继续这一过程（再各取两张牌，第一张正面朝下，第二张正面朝上，比较第二张的点数），否则点数大的一方把桌子上的所有牌按照顺序（游戏者 1 的第一张牌，游戏者 2 的第一张牌，游戏者 1 的第二张牌等）拿走放到自己的牌堆底部。在战争的过程中，如果有一方无牌可出，则另一方获胜；如果双方均无牌可出，则游戏以平局结束。

玩过几次的人都会知道，这个游戏常常会持续很长时间都结束不了。你的任务就是编程模拟这一过程，最后给出游戏结束之前所用的步数。

<sup>4</sup> 译者注：这并不是一个标准的 STL 容器，并不保证能在比赛中使用。

以下为题解

如何解读这样的题目描述？我们建议你在设计、编码、测试和调试时参考下述建议：

- **仔细读题** — 仔细阅读题目中的每一行，并当评测系统返回错误信息后重读一遍。背景信息可以适当地略读，因为很多文字实际上对程序没有影响。对于输入输出格式和样例一定要格外仔细。但是 …
  - **不要假设** — 阅读和理解题目描述 (对应于现实中的各类 “说明书”(specification)) 是解题的重要环节。题目描述往往隐藏着陷阱。

不要觉得某些样例所具有的性质会被所有数据满足。要特别留意未指名的输入顺序、无界的整数、超长的行、负数等。没有被明确禁止的输入都应该被看作是允许出现的！

- 不要刻意追求效率 — 时间效率一般来讲并不是大问题，除非我们在多项式可解的题目里使用了指数时间算法。不要太担心你的程序会超时，除非你有足够的证据。请根据题目中关于最大输入规模的信息来选择尽量简单但不会超时的算法。

尽管在实际玩的时候，这个纸牌游戏看上去总是需要很长时间（事实上，它甚至可能无限循环），但是对于本题的描述，我们没有必要担心程序的时间效率。

## 2.4 准备行动

应该用何种数据结构来表示一副牌？这取决于你将对这副牌做什么。你会洗牌么？会比较这些牌的的大小么？会搜寻一张特定的牌么？你希望执行的动作定义了数据结构的操作。

在本题中，一堆牌的主要操作是从顶部拿走牌以及往底部插入新牌。这正是我们前面讲过的 FIFO 队列的基本操作。

但其实这里还有一个更加基础的问题：如何表示一张牌？注意，每张牌有花色和点数之分。我们有多种选择，例如用两个字符分别表示花色和点数。尽管游戏本身完全与每张牌的花色无关，但如果我们将完全抛弃花色，将会遇到麻烦：我们无法打印每次出的牌，也无法在调试的时候直观地验证模拟过程确实按照我们所设想的进行。另一种方法是用 0 到 51 之间的不同整数来表示不同的牌，然后用一组映射关系在整数与牌之间自由转换。

游戏的主要操作是比较牌的点数大小，但是在“花色 - 点数”的双字符组表示中，点数的比较似乎会显得比较笨拙，因为我们无法简单地利用字符比较，而需要手工指定一些规则。

为此，我们再次考虑刚才所说的映射方法，它是一种有用的通用编程技巧。只要我们可以为某集合  $S$  的所有元素设计两个函数  $ranking$  和  $unranking$ ，满足对于所有  $s \in S$  都有  $s = unrank(rank(s))$ ，对于不同的  $s \in S$ ， $rank(s)$  总是取不同的整数。换句话说，可以把  $rank$  看作是一个没有冲突的哈希函数。

应该如何做映射呢？我们首先把点数从小到大排序，然后利用“每种点数恰好有四种花色”的事实简单地利用乘法和除法完成映射。

```
#define NCARDS 52      /* number of cards */
#define NSUITS 4        /* number of suits */

char values[] = "23456789TJQKA";
char suits[] = "cdhs";

int rank_card(char value, char suit)
{
    int i,j;          /* counters */

    for (i=0; i<(NCARDS/NSUITS); i++)
        if (values[i]==value)
            for (j=0; j<NSUITS; j++)
                if (suits[j]==suit)
                    return( i*NSUITS + j );

    printf("Warning: bad input value=%d, suit=%d\n",value,suit);
}

char suit(int card)
{
    return( suits[card % NSUITS] );
}

char value(int card)
{
    return( values[card/NSUITS] );
}
```

映射函数很容易在排列 (permutations)、子集 (subsets) 和绝大多数组合对象 (combinatorial objects) 上实现。这是一种通用的编程技巧，可以简化很多数据类型的运算。

## 2.5 字符串输入输出

本题的输入由两行组成。第一行表示游戏者 1 的牌堆，第二行表示游戏者 2 的牌堆。下面是三个游戏输入的实例：

4d Ks As 4h Jh 6h Jd Qs Qh 6s 6c 2c Kc 4s Ah 3h Qd 2h 7s 9s 3c 8h Kd 7h Th Td  
 8d 8c 9c 7c 5d 4c Js Qc 5s Ts Jc Ad 7d Kh Tc 3s 8s 2d 2s 5h 6d Ac 5c 9h 3d 9d  
 6d 9d 8c 4s Kc 7c 4d Tc Kd 3s 5h 2h Ks 5c 2s Qh 8d 7d 3d Ah Js Jd 4c Jh 6c Qc  
 9h Qd Qs 9s Ac 8h Td Jc 7s 2d 6s As 4h Ts 6h 2c Kh Th 7h 5s 9c 5d Ad 3h 8s 3c  
 Ah As 4c 3s 7d Jc 5h 8s Qc Kh Td 3h 5c 9h 8c Qs 3d Ks 4d Kd 6c 6s 7h Qh 3c Jd  
 2h 8h 7s 2c 5d 7c 2d Tc Jh Ac 9s 9c 5s Qd 4s Js 6d Kc 2s Th 8d 9d 4h Ad 6h Ts

很多题目都需要读取非数值 (non-numerical) 数据。文本串将在第 3 章中进行深入讨论，但是请记住，你有很多方法可以读取文本串：

- 一次只读一个字符 (例如 C 中的 `getchar()` 函数)，然后一次处理一个。
- 一次读一个格式化后的标记 (token)<sup>5</sup> (例如 C 中的 `scanf()` 函数)，然后再处理。
- 一次读一整行 (例如 C 中的 `fgets()` 函数)，然后访问其中的字符或子串来解析它。
- 如果你的程序设计语言支持，你还可以直接使用更为“现代化”的输入输出方式，例如流。但你仍然需要决定你的输入单元是字符、串或者是其他。

在下面的代码中，我们选择了第一个方案：一次只读一个字符。为了让代码更清晰，我们显式的处理了行尾标记 (C 语言中的'\n')：

```
main()
{
    queue decks[2];                      /* player's decks */
    char value,suit,c;                   /* input characters */
    int i;                                /* deck counter */

    while (TRUE) {
        for (i=0; i<=1; i++) {
            init_queue(&decks[i]);

            while ((c = getchar()) != '\n') {
                if (c == EOF) return;
                if (c != ' ') {
                    value = c;
                    suit = getchar();
                    enqueue(&decks[i],rank_card(value,suit));
                }
            }
        }
    }
}
```

<sup>5</sup> 译者注：这个词在国内的文献中多被直接引用，这里暂时翻译为“标记”，但后文将保留英文单词 token。

```
    war(&decks[0],&decks[1]);  
}  
}
```

注意我们把两堆牌表示成队列数组而不是两个单独的队列，这样就可以只用一份代码处理两个牌堆了。

## 2.6 赢得战争

在完成数据结构的设计之后，war函数的编写就相当简单了。注意战争的过程也可以用队列操作建模，所以直接使用前面的抽象数据类型即可。

```
war(queue *a, queue *b)  
{  
    int steps=0;                      /* step counter */  
    int x,y;                          /* top cards */  
    queue c;                          /* cards involved in the war */  
    bool inwar;                       /* are we involved in a war? */  
  
    inwar = FALSE;  
    init_queue(&c);  
  
    while ((!empty(a)) && (!empty(b) && (steps < MAXSTEPS))) {  
        steps = steps + 1;  
        x = dequeue(a);  
        y = dequeue(b);  
        enqueue(&c,x);  
        enqueue(&c,y);  
        if (inwar) {  
            inwar = FALSE;  
        } else {  
            if (value(x) > value(y))  
                clear_queue(&c,a);  
            else if (value(x) < value(y))  
                clear_queue(&c,b);  
            else if (value(y) == value(x))  
                inwar = TRUE;  
        }  
    }  
}
```

```
    inwar = TRUE;
}

}

if (!empty(a) && empty(b))
    printf("a wins in %d steps \n",steps);
else if (empty(a) && !empty(b))
    printf("b wins in %d steps \n",steps);
else if (!empty(a) && !empty(b))
    printf("game tied after %d steps, |a|=%d |b|=%d \n",
           steps,a->count,b->count);
else
    printf("a and b tie in %d steps \n",steps);

}

clear_queue(queue *a, queue *b)
{
    while (!empty(a)) enqueue(b,dequeue(a));
}
```

## 2.7 测试与调试

当自动评测系统告诉你提交没有通过后，调试往往是很让人郁闷的。系统不会告诉你程序对于哪组数据出错，而你也不能提交一个“假”的程序来蒙混过关。

因此在提交之前，你应当系统地测试你的程序。自己发现低级错误往往能节省宝贵的时间，特别是在错误提交会导致罚时的比赛中。以下是一些设计测试数据的建议：

- 样例 — 绝大多数题目里都包含样例输入和输出。通过样例数据是程序正确的必要（但不充分）条件。
- 错误输入 — 如果题目指明你的程序必须用某种特定的方式处理非法输入，你应当进行相应的测试。
- 边界条件 — 很多 bug 都是“失之毫厘”的。你应当用空输入、一个元素、两个元素、零值等特殊数据来测试一些边界条件。
- 正确答案已知的用例 — 设计好数据的关键之一是确保答案正确。你的数据设计重点应该放在那些足够简单、可以用心算或者笔算得到结果的数据。你应当仔细、完整地分析算法的期望行为，以免误信了那些看上去可信 (plausible-looking) 的错误输出。

- 正确答案未知的大规模数据 — 一般来说，只有小规模数据能够手算出来，因此验证大规模的数据往往是困难的。尽管如此，你还是应当测试一些容易生成的大数据，例如随机数据或者有规律的数据，至少要确保你的程序既不会崩溃，也不会做出一些明显错误的蠢事。

测试是查错 (revealing bugs) 的艺术，而调试是排错 (exterminating bugs) 的艺术。我们专门为了本节而设计了这道编程题目，但是作者自己花在调试上的时间却比事先所预料的要长。这并不奇怪，因为所有的程序员都是乐天派。但我们如何才能避免落入这样的陷阱呢？

- 熟悉调试器 — 大多数编程环境都配备了源码级调试器 (source-level debugger)。你可以在特定的地方或者在满足特定逻辑表达式时终止程序执行，然后查找或修改特定变量的值。建议你尽快学习使用它们。学得越早，你节省的时间就会越多。
- 打印数据结构 — 在调试“纸牌大战”的时候，我们的队列实现有一个细小的问题。这几乎只能通过显示队列中的内容来发现错误。建议对各种非平凡 (non-trivial) 的数据结构编写它们特有的打印函数，因为源码调试器很难有效地自动显示它们。
- 严格测试不变量 — 纸牌的双向映射函数是一个潜在的错误源。一个典型的情况是：*rank* 和 *unrank* 函数之间并非可逆关系。不变量 (*invariant*) 是指一个程序总是满足的条件，和具体输入无关。下面是一个简单的不变量测试：

```
for (i=0; i<NCARDS; i++)
    if (i != rank_card(value(i), suit(i)))
        printf("Error: rank_card(%c,%c)=%d not %d\n", value(i),
               suit(i), rank_card(value(i), suit(i)), i);
```

上述代码完整地测试了可逆映射的正确性。

- 静态查错 — 可能最简单有效的查错方法是仔细阅读代码。错误往往隐藏在那些特别丑陋的，或者看上去很“高级”但实际上难以理解的代码中。
- 编写有意义的调试输出语句 — 在程序中增加调试输出语句通常会带来麻烦，但又往往是不得不的。它们可以通过有效的使用源码调试器来减少，但如果你仍然决定要用，需要让它们发挥出最大的作用。例如，在输出一个变量值的同时输出变量的名称和在程序中的位置。一般情况下，你很容易迷失在海量的调试输出中，但是如果输出了一些有意义的文本，就可以比较容易地在输出中找出有用信息。

当你不再需要一些调试输出语句之后，你应当注释掉它们而不是直接删除。因为在很多情况下，你还会再次需要它们。

- 开数组时留出一定的余量 — “失之毫厘”的错误往往难以找到。尽管我们可以通过清晰的思考来避免它们，但思考需要花时间。相反，浪费一点内存却是无关紧要的。开数组的时候留出一定你认为用不上的空间将大大减少此类错误的可能性。

- 确保 *Bug* 真的是 *Bug* — 在调试“纸牌大战”的时候，我们花了不少时间来动态跟踪程序，试图找出我们程序死循环的原因，但最后却发现对于随机初始局面，游戏永不停止的概率惊人的大。事实上，我们的程序根本没有错！请从这个例子中吸收教训：每次收牌的时候不是按固定顺序，而是洗牌后插入队列，以减少死循环的可能性。

## 2.8 习题

### 2.8.1 快乐的跳跃者 (Jolly Jumper)

PC/UVa 题号: 110201/10038, 流行度: A, 通过率: average 难度: 1

对于一个包含  $n > 0$  个元素的整数序列，如果序列中相邻元素之差的绝对值取遍从 1 到  $n - 1$  的所有整数，那么这个序列就叫做 jolly jumper。例如：1 4 2 3 就是一个 jolly jumper，因为相邻元素之差的绝对值分别为 3、2、1。这个定义意味着所有单元素序列都是 jolly jumper。写一个程序来判断一个序列是不是 jolly jumper。

输入

输入的每行包含一个整数  $n \leq 3000$ ，然后是  $n$  个整数，表示一个输入序列。

输出

对于输入的每一行，输出一行“Jolly”或者“Not jolly”来表示它是否为 jolly jumper。

## 样例输入

## 样例输出

4	1	4	2	3	
5	1	4	2	-1	6

Jolly  
Not jolly

### 2.8.2 扑克牌型 (Poker Hands)

PC/UVa 题号: 110202/10315, 流行度: C, 通过率: average 难度: 2

一副纸牌包括了 52 张牌。每一张纸牌的花色为草花、方块、红桃或黑桃中的一种(在输入中分别用C、D、H、S来描述)。每一张纸牌的点数为2到10中的某一个数字,或是J、Q、K、A(用2、3、4、5、6、7、8、9、T、J、Q、K、A来表示)。为了给牌型计分,牌的点数从小到大如上排列,其中2最小而A最大。在本题中,花色对大小毫无影响。

一手牌为一副牌中的五张。下面从小到大定义九种牌型。对于不同种类的牌型，前面的小于后面的；对于相同种类的牌型，比较规则见该牌型的描述。

**最大牌** 不属于下述分类的牌型都属于此类。这手牌的大小取决于最大牌的大小。如果最大牌一样大，则比较次大牌的大小，以此类推。

**一对** 五张牌中恰好有两张点数相同的牌 (对子)。这手牌的大小取决于对子的点数大小，然后是其他牌的大小 (按从大到小顺序比较，与“最大牌”相同)。

**两对** 包含两个不同的对子。这手牌的大小取决于较大点数的对子，然后是另一个对子的点数，最后是剩下那张牌的大小。

**三张** 包含三张点数相同的牌。这手牌的大小只取决于这三张牌的点数。

**顺子** 五张牌的点数为连续值。这手牌的大小只取决于最大牌的点数。

**同花** 五张牌的花色相同。这手牌的大小按照与“最大牌”相同的规则确定。

**葫芦** 其中三张点数相同，另两张形成一个对子。这手牌的大小只取决于“三张”的点数。

**四张** 其中四张点数相同。这手牌的大小只取决于这四张牌的点数。

**同花顺** 五张牌为点数连续的同花。这手牌的大小取决于最大牌的点数。

你的任务是比较若干对牌型，分别指出哪一方比较大 (如果有的话)。

## 输入

输入包含若干行，每行包含 10 张牌。前五张牌属于黑方 (*Black*)，后五张牌属于白方 (*White*)。

## 输出

对于每行输入，输出一行，为下列三种情况之一：

Black wins.

White wins.

Tie.

### 样例输入

2H 3D 5S 9C KD 2C 3H 4S 8C AH  
2H 4S 4C 2D 4H 2S 8S AS QS 3S  
2H 3D 5S 9C KD 2C 3H 4S 8C KH  
2H 3D 5S 9C KD 2D 3H 5C 9S KH

### 样例输出

White wins.  
Black wins.  
Black wins.  
Tie.

### 2.8.3 罢工 (Hartals)

PC/UVa 题号: 110203/10050, 流行度: B, 通过率: high 难度: 2

孟加拉国的政党用对经济具有极大破坏力的规律性罢工来显示他们的威力。对于本题来说，每个政党的罢工行为都可以用一个称为罢工指数的正整数  $h$  来刻画，表示该政党预定每  $h$  天举行一次罢工。

假设有 3 个政党， $h_1 = 3, h_2 = 4, h_3 = 8$ ，其中  $h_i$  表示第  $i$  个政党的罢工指数。我们可以模拟出这 3 个政党在 14 天中的行为。模拟总是从某个星期天开始，并且在星期五和星期六不能有任何罢工。

星期	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	Su	Mo	Tu	We	Th	Fr	Sa	Su	Mo	Tu	We	Th	Fr	Sa
政党 1		x		x			x			x		x		
政党 2			x				x				x			
政党 3							x							
罢工数		1	2					3	4			5		

在这 14 天以内将发生 5 次罢工 (分别在第 3、4、8、9 和 12 天)。第 6 天没有罢工，因为它是星期五。因此，在两周的时间内，总共罢工了 5 天。给出每个政党的罢工指数和时间  $N$ ，计算出在  $N$  天以内存在罢工的天数。

## 输入

输入的第一行只有一个整数  $T$ , 表示接下来要处理的数据组数。每组数据的第一行有一个整数  $N(7 \leq N \leq 3650)$ , 代表需要模拟的天数。接下来的一行包括另一个整数  $P(1 \leq P \leq 100)$ , 代表政党的个数。接下来的  $P$  行中, 第  $i$  行包含了一个正整数  $h_i$ (保证不是 7 的倍数), 表示第  $i$  个政党的罢工指数( $1 \leq i \leq P$ )。

输出

对于每一组数据，在单独的一行输出一个整数，代表存在罢工的天数。

### 样例输入

2

14

3

3

4

8

100

4

18

1

3

20

## 样例输出

5

15

#### 2.8.4 解密 (Crypt Kicker)

PC/UVa 题号: 110204/843, 流行度: B, 通过率: low 难度: 2

重排字母表中的各个字母是一种常用但不安全的加密方式。具体来说，字母表中的每个

字母始终都被另外某个字母替换。为了保证加密过程是可逆的，任何两个字母都不能被同一个字母替换。

你的任务是将密文破解成原文。假设每一行用的是不同的替换方案，并且原文中的每一个单词都在一个已知的字典内。

### 输入

输入的第一行有一个整数  $n$ ，接下来的  $n$  行给出字典。其中每行为一个由小写字母组成的单词，按字典序排列。这  $n$  个单词组成的字典包含了原文中所有可能出现的单词。接下来有若干行，每行都是一个加密后的文本。

词典中最多包含 1000 个单词。每个单词不超过 16 个字母。密文只包含小写字母和空格，长度不超过 80。

### 输出

输出每行密文的原文。如果有多种解密方案，可以输出任意一种；如果无解，则在输出时用星号来替换每一个输入的字母。

### 样例输入

```
6
and
dick
jane
puff
spot
yertle
bjvg xsb hxsn xsb qymmm xsb rqat xsb pnetfn
xxxx yyy zzzz www yyyy aaa bbbb ccc dddddd
```

### 样例输出

```
dick and jane and puff and spot and yertle
***** *** **** *** ***** *** **** *****
```

## 2.8.5 完美洗牌术 (Stack'em Up)

PC/UVa 题号: 110205/10205, 流行度: B, 通过率: average 难度: 1

在一个大城市中有很多赌场。在其中一个赌场中，荷官 (dealer) 会出老千。她拥有若干种完美的洗牌方法，每种方法都能保证洗牌后所有牌按照一种预先设计好的顺序排列。例如，“底牌”洗牌法就是将最下面的牌移到最上面。当把各种洗牌法组合起来时，这位无耻的

荷官几乎能将一副牌洗成她想要的任何一种顺序。

你被保安经理 (security manager) 雇来监视荷官。你有一份荷官的洗牌法清单以及能让你分辨出任何情况下她所使用的洗牌法的视觉线索。你的工作是预测洗牌后的各张扑克牌的排列顺序。

一副标准的牌包括 52 张牌，分 13 种点数和 4 种花色。点数用 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace 表示；花色用 Clubs, Diamonds, Hearts, Spades 表示。每张牌都可通过点数和花色唯一确定，用 <value> of <suit> 表示。例如 “9 of Hearts” 或者 “King of Spades”。在本题中，一副新的纸牌先按花色的首字母，然后按点数从小到大排列。

### 输入

输入第一行只有一个正整数，代表要处理的测试数据组数。接下来是一个空行。在每相邻两组数据之间也有一个空行。

每组数据的第一行包含一个正整数  $n \leq 100$ ，代表荷官所掌握的洗牌法数目。接下来有  $n$  组整数，每一组都是 1 到 52 的某种排列。在该排列中，整数  $i$  在位置  $j$  表示洗牌前的第  $i$  张牌在洗牌后位于第  $j$  张。

接下来有若干行，每一行都有一个 1 到  $n$  之间的整数  $k$ ，表示荷官使用了第  $k$  种洗牌法。

### 输出

对于每组数据，假设荷官总是使用一副新牌（排列方式见题目描述）。输出洗牌完成后整副牌的排列方式。相邻两组数据的输出之间应用一个空行隔开。

### 样例输入

```
1

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

### 样例输出

```
King of Spades
2 of Clubs
4 of Clubs
```

www.niuniu51.com

5 of Clubs  
6 of Clubs  
7 of Clubs  
8 of Clubs  
9 of Clubs  
10 of Clubs  
Jack of Clubs  
Queen of Clubs  
King of Clubs  
Ace of Clubs  
2 of Diamonds  
3 of Diamonds  
4 of Diamonds  
5 of Diamonds  
6 of Diamonds  
7 of Diamonds  
8 of Diamonds  
9 of Diamonds  
10 of Diamonds  
Jack of Diamonds  
Queen of Diamonds  
King of Diamonds  
Ace of Diamonds  
2 of Hearts  
3 of Hearts  
4 of Hearts  
5 of Hearts  
6 of Hearts  
7 of Hearts  
8 of Hearts  
9 of Hearts  
10 of Hearts  
Jack of Hearts  
Queen of Hearts  
King of Hearts

Ace of Hearts

2 of Spades

3 of Spades

4 of Spades

5 of Spades

6 of Spades

7 of Spades

8 of Spades

9 of Spades

10 of Spades

Jack of Spades

Queen of Spades

Ace of Spades

3 of Clubs

### 2.8.6 Erdös 数 (Erdös Numbers)

PC/UVA 题号: 110206/10044, 流行度: B, 通过率: low 难度: 2

匈牙利数学家 Paul Erdös(1913 – 1996) 是 20 世纪最著名的数学家之一。任何能够荣幸成为 Erdös 的合作作者的数学家都受到了人们的尊敬。

可惜的是，并不是每个人都能幸运的与 Erdös 一同写论文。因此，大多数人所能做的最好的事，便是与某个曾经与 Erdös 一同发表过论文的人一起发表论文。这样就产生了所谓的Erdös 数 (*Erdös numbers*)。所有与 Erdös 共同发表过论文的作者的 Erdös 数为 1。没有与 Erdös 共同发表过文章，但曾与 Erdös 数为 1 的人合作发表了文章的作者的 Erdös 数为 2，以此类推。

你的任务是写一个根据一份论文与科学家的清单计算出每位科学家的 Erdös 数的程序。

#### 输入

输入第一行为测试数据组数。每组测试数据包含一个论文数据库和一个人名列表。第一行包含两个自然数  $P$  和  $N$ 。接下来的  $P$  行是论文数据库，每行按如下方式描述一篇论文：

Smith, M.N., Martin, G., Erdos, P.: Newtonian forms of prime factors

注意 “ö” 这样的元音变音 (umlauts) 将写成 “o” 这样的简单形式。在  $P$  篇论文之后的  $N$  行描述作者，格式如下：

Martin, G.

## 输出

对于每组输入数据，你需要输出一行字符串“Scenario *i*”(*i* 为测试数据序号)，然后是一个人名与对应 Erdős 数的列表。输出中各人名的顺序应和输入保持一致。这里的 Erdős 数应该是基于论文数据库所列论文得出的。与 Erdős 完全没有关系的作者的 Erdős 数为无穷大(infinity)。

## 样例输入

```
1
4 3
Smith, M.N., Martin, G., Erdos, P.: Newtonian forms of prime factors
Erdos, P., Reisig, W.: Stuttering in petri nets
Smith, M.N., Chen, X.: First order derivates in structured programming
Jablonski, T., Hsueh, Z.: Selfstabilizing data structures
Smith, M.N.
Hsueh, Z.
Chen, X.
```

## 样例输出

```
Scenario 1
Smith, M.N. 1
Hsueh, Z. infinity
Chen, X. 2
```

### 2.8.7 比赛记分板 (Contest Scoreboard)

PC/UVa 题号: 110207/10258, 流行度: B, 通过率: average 难度: 1

想参加 ACM 国际大学生程序设计竞赛 (ACM/ICPC) 吗？那么你最好了解一下如何计分。参赛队首先按解题数(越多越好)降序排列，然后按罚时(penalty time)(越少越好)升序排列。如果两个或两个以上队伍的解题数和罚时均相同，在显示时队伍序号小的排在前面。

一道题目被某队“解出”，当且仅当这支队伍对该题的提交(submission)中至少有一份的评测结果为“正确”。罚时的计算方式为：第一份正确提交被收到时距比赛开始的分钟数，加上在此之前每份非正确提交的 20 分钟罚时。未解决的题目不计罚时。

## 输入

输入第一行包含一个整数，表示测试数据的组数。接下来是一个空行。以后每两组测试数据间均用一个空行隔开。

每组数据是一个评测队列的快照，包含编号为 1 到 100 的部分或全部参赛队对于题目 1 到 9 的提交。每行包含三个数和一个字母，按队伍 题目 时间 状态的顺序排列，其中状态为 C、I、R、U、E 中的一种，分别代表正确，错误，答疑请求，未评测和违规提交。后三种情况对评分没有影响。

在输入数据中，各提交按照收到顺序排列。

### 输出

对于每组数据，输出一个记分板，按前面描述的规则排名。每行包含参赛队编号，已解决的题目数以及总累计罚时。鉴于有些队并未真正到场参加比赛，你只需输出那些至少提交过一次的队伍排名。

相邻两组数据的输出之间应用一个空行隔开。

### 样例输入

1

1 2 10 I

3 1 11 C

1 2 19 R

1 2 21 C

1 1 25 C

### 2.8.8 Yahtzee 游戏 (Yahtzee)

PC/UVa 题号: 110208/10149, 流行度: C, 通过率: average 难度: 3

Yahtzee 是一个用五个骰子来玩的游戏，共掷 13 轮。同样的，计分卡也包含 13 项。在每一轮中，游戏者可以任意指定一个计分项，并按照相应的规则计分，但在整个游戏的 13 轮中，每个计分项只能被选一次。13 个计分项规则如下<sup>6</sup>：

- 计一 - 所有点数为一的骰子点数和
- 计二 - 所有点数为二的骰子点数和
- 计三 - 所有点数为三的骰子点数和
- 计四 - 所有点数为四的骰子点数和
- 计五 - 所有点数为五的骰子点数和
- 计六 - 所有点数为六的骰子点数和
- 机会 - 所有骰子点数和
- 三同 - 若掷出至少三个相同点数的骰子，计所有骰子点数和

<sup>6</sup> 译者注：在传统的 Yahtzee 游戏中，小顺和大顺分别是 30 和 40 分，葫芦只有 25 分。五同也称 Yahtzee，当第一次被计分后每次掷出都将获得 100 分奖励。

- 四同 - 若掷出至少四个相同点数的骰子，计所有骰子点数和
  - 五同 - 若所有骰子点数相同，计 50 分
  - 小顺 - 若四个骰子成顺 (即 1, 2, 3, 4 或 2, 3, 4, 5 或 3, 4, 5, 6)，计 25 分
  - 大顺 - 若五个骰子成顺 (1, 2, 3, 4, 5 或 2, 3, 4, 5, 6)，计 35 分
  - 葫芦 - 如果三个骰子点数相同而另两个骰子点数也相同，计 40 分  
在后六种计分项中，如果条件不满足，则计 0 分。

游戏的总分即所有 13 项的和。若前六个计分项的得分之和大于或等于 63，则将在总分中加上 35 分作为奖励。

你的任务是计算一次给定的完整游戏所可能得到的最大总分。

输入

每行输入数据包含五个 1 到 6 之间的整数，表示每轮投掷骰子的情况。输入的每 13 行组成了 一局完整的游戏。

输出

对于每局游戏，输出一行 15 个数：每个计分项的得分（按题目描述的顺序）、奖励分（0 或 35）以及总分。如果有多种可能情况，可以输出任意一种。

### 样例输入

1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 2 3 4 5  
1 1 1 1 1  
6 6 6 6 6  
6 6 6 1 1  
1 1 1 2 2

1 1 1 2 3  
1 2 3 4 5  
1 2 3 4 6  
6 1 2 6 6  
1 4 5 5 5  
5 5 5 5 6  
4 4 4 5 6  
3 1 3 6 3  
2 2 2 4 6

### 样例输出

1 2 3 4 5 0 15 0 0 0 25 35 0 0 90  
3 6 9 12 15 30 21 20 26 50 25 35 40 35 327

## 2.9 提 示

- 2.2 我们是否可以用一个数值来表示扑克牌，从而让比较更加容易？
- 2.3 是否必须先构造出实际的表格，才能统计出罢工的数量？
- 2.4 根据重复字母和长度把单词划分成等价类是否划算？
- 2.7 有多个关键字时，最简单的排序方法是什么？
- 2.8 是必须枚举每一轮和计分项之间的所有映射方式，还是可以直接为每一轮选择最合适 的计分项？

## 2.10 注 解

- 2.1 本题中的 jolly jumper 是优美图标号的特殊情形。一个图被称为优美 (graceful) 的当且仅当存在一种给  $n$  个顶点进行整数标号的方式，使得所有  $m$  条边的端点标号之差的绝对值取遍 1 到  $m$  之间的所有整数值。本题中的 jolly jumper 正是包含  $n$  个结点的路径的优美标号。有一个著名的优美树猜想 (*graceful tree conjecture*)：每棵树均存在优美标号。优美图是一个不错的本科生研究课题，读者可参考 Gallian 的动态调研 [Gal01]，以获得一份易于理解的开放问题列表。
- 2.5 洗牌术中的数学是一个引人入胜的课题。完全洗牌 (*perfect shuffle*) 将一堆牌均分成  $A$  和  $B$ ，然后将两堆牌交替放置： $A$  的第一张牌， $B$  的第一张牌， $A$  的第二张牌 … 令人惊奇的是，八次完全洗牌后将回到这堆牌的初始状态！这一结论可以用模算术 (modular arithmetic) 或者置换中有关循环的理论来证明。关于完全洗牌的更多讨论参见 [DGK83, Mor98]。

2.6 本书第一作者的 Erdős 数为 2, 因此第二作者的 Erdős 数不超过 3。Erdős 闻名于世的主要原因是提出组合数学、图论、数论中优美、易于理解但是难以解决的问题。有兴趣的读者可以阅读他那些广为传颂的传记 [Hof99, Sch00] 来进一步了解这个魅力四射的人物。

第3章 字符串

文本串 (text strings) 是一个越来越重要的基础数据结构。Google 这样的 Internet 搜索引擎在浩如烟海的文档中搜索，几乎是瞬间完成。人类基因组 (genome) 序列由多达 30 亿个字符组成，描述了构成人体的所有蛋白质。当在这样的串中搜寻有趣的模式 (pattern) 时，我们实际上是在探索生命的奥秘。

和它们相比，解决本章中的编程问题所需的投入要小得多。但这些问题能让读者了解字符和文本串在现代计算机中的表示，并学到搜索和操纵这些数据的精巧算法。有兴趣的读者可以阅读 [Gus97]，该书包含了字符串算法的若干高级话题的讨论。

### 3.1 字符编码

字符编码(*character codes*)是一个特定字符集(alphabet)中的符号与数之间的映射。从根本上来说，计算机是用来处理数值数据的。对于一个给定字符集，它仅仅知道每个可能的数值各与哪个字符相关联。当改变一个打印程序所用的字体(font)时，仅仅是每个字符所对应的图像位图(image bit-map)被改变。当把操作系统的默认语言从英文切换到俄文时，仅仅是每个字符的编码映射发生了改变。

如果要和字符串打交道，了解各种字符编码的设计是很有帮助的。美国标准信息交换码(*American Standard Code for Information Interchange, ASCII*)是一个包含 $2^7 = 128$ 个字符的单字节编码。<sup>1</sup>但一个字节一共有8个比特，因此最高位的比特被固定为0。

图 3.1 是一个 ASCII 字符编码表。相邻两个表项组成一对，其中左边是编码值（十进制），右边是与之相关联的符号。编码和符号之间的对应关系并不是任意的。它有若干有趣的特性，让编程更加简单：

- 所有不可打印字符 (non-printable characters) 要么前 3 位均为 0, 要么最低 7 位均为 1。这使得在打印文本之前可以很容易地把其中的“废物”清理掉, 尽管似乎很少有程序是这么做的。
  - 大写字母、小写字母和数字都是连续出现的。因此, 当我们需要遍历所有字母/数字时, 只需从第一个字符<sup>2</sup> (例如 “a”) 循环到最后一个字符 (例如 “z”)。

<sup>1</sup> 请注意，ASCII 有很多变种。其中最重要的也许是 ISO Latin-1 字符集（也称 ISO-8859-1），它是一个完整的 8 位编码，包含了欧洲语言中带重音符号的字母（accented characters）。

<sup>2</sup> 译者注：这里混合使用字符和它们的 ASCII 码。例如，给字符做加减运算的时候，实际上是指对它们的编码运算。

0	NUL	1	SOH	2	STX	3	ETX	4	EOT	5	ENQ	6	ACK	7	BEL
8	BS	9	HT	10	NL	11	VT	12	NP	13	CR	14	SO	15	SI
16	DLE	17	DC1	18	DC2	19	DC3	20	DC4	21	NAK	22	SYN	23	ETB
24	CAN	25	EM	26	SUB	27	ESC	28	FS	29	GS	30	RS	31	US
32	SP	33	!	34	"	35	#	36	\$	37	%	38	&	39	'
40	(	41	)	42	*	43	+	44	,	45	.	46	.	47	/
48	0	49	1	50	2	51	3	52	4	53	5	54	6	55	7
56	8	57	9	58	:	59	:	60	<	61	=	62	>	63	?
64	@	65	A	66	B	67	C	68	D	69	E	70	F	71	G
72	H	73	I	74	J	75	K	76	L	77	M	78	N	79	O
80	P	81	Q	82	R	83	S	84	T	85	U	86	V	87	W
88	X	89	Y	90	Z	91	[	92	/	93	]	94	-	95	-
96	'	97	a	98	b	99	c	100	d	101	e	102	f	103	g
104	h	105	i	106	j	107	k	108	l	109	m	110	n	111	o
112	p	113	q	114	r	115	s	116	t	117	u	118	v	119	w
120	x	121	y	122	z	123	{	124	-	125	}	126	~	127	DEL

图 3.1 ASCII 编码表

- 上述连续编码的另一个好处是，当我们需要把字符（例如“I”）转化成它在对应字符序列中的序号（8，如果“A”的序号是0的话），只需用它减去第一个字符（“A”）的编码即可。
- 当我们需要把一个大写字母（例如“C”）转化为小写字母时，只需给它加上第一个大小写字母的编码差（“C”-“A”+“a”）。类似的，字符x是大写当且仅当它在“A”和“Z”之间。
- 当给定字符编码之后，我们可以预测把文本文件按普通方法排序后的结果。按照字典序（alphabetical order），“x”、“3”和“C”中的哪一个排在最前面？按字典序排序的意思是按照字符编码排序。如果需要按照另一个对照序列（collating sequence）排序，需要用到更为复杂的排序函数。我们将在第4章中讨论。
- 不可打印字符中的换行符（10）和回车符（13）用来给文本行分界。UNIX 和 Windows 对这两个字符的用法不同<sup>3</sup>，这无疑增加了在两个操作系统间移动文件的痛苦。

更为现代的国际化字符编码，例如 Unicode<sup>4</sup>，用两个甚至三个字节来表示一个字符，理论上可以表示世界上任何语言的所有字符。尽管如此，古老而优秀的 ASCII 依旧被保留在 Unicode 当中。例如在 UTF-8 中，当首字节的最高位为0时，文本将被解释为单字节字符。这样，我们可以在为成千上万的新字符打开一扇门的同时继续使用更为简单、节省内存的编码。

这些都使得用不同的编程语言操纵文本的方法也不同。相对较老的语言，如 Pascal、C 和 C++ 把 char 类型固定为8位，因此适合处理原始文件（raw file），包括那些并非可打印的二进制文件。但 Java 不同，它的字符采用 Unicode 编码。当你需要切换编程语言的时候，应当注意这个差别。

<sup>3</sup> Windows 下每行以 13, 10(回车、换行) 结束，UNIX 以 10(换行) 结束，Mac 以 13(回车) 结束。

<sup>4</sup> 译者注：包括 UTF-8, UTF-16 和 UTF-32 三种，参见 <http://www.unicode.org/unicode/standard/principles.html>。

字符串是字符序列，串中各字符的顺序至关重要。了解你所用的编程语言如何表示字符串是很重要的，因为有几种不同的表示方法。

- 以空字符结尾的数组 — C/C++ 把字符串看作字符数组。当遇到空字符 “\0”(即 ASCII 码为 0 的字符) 时字符串终止。如果你忘记在最后一个字符之后显式的添加一个空字符，那么它之后的若干 (通常是非可打印的) 字符也会被算作字符串的一部分。在定义字符串的时候，你必须为它分配一个足够大的数组来容纳最长可能的串 (包括结尾的空字符)，除非你想让程序崩溃。这种表示法的优点是所有单个字符都可以通过下标直接访问，和普通数组一样。
- 带长度信息的数组 — 另一种方案是在数组的第一个位置保存字符串长度，从而避免了在串尾增加结束标记的需要。这大概是 Java 虚拟机内部表示字符串的方法，尽管在用户视角下，字符串是定义了若干运算和方法的对象。
- 字符链表 — 文本串可以用链表来表示，但我们通常并不这么做，因为它的空间利用率太低 (需要为每个单字节字符关联一个多字节的指针)。不过如果你需要经常在文本串的中间插入和删除子串的话，字符链表也许有用。

字符串的表示法很大程度上决定了字符串能方便、高效支持的操作有哪些。试对如下属性比较上述三种数据结构：

- 哪种方案最节省空间？适用于什么大小的字符串？
- 哪种方案限制了它所能表示的字符串内容？
- 哪些方案支持在常数时间内访问第  $i$  个字符？
- 哪些方案能高效地检查第  $i$  个字符是否在字符串长范围之内，从而避免越界错误？
- 哪些方案能高效地在第  $i$  个位置删除或插入字符？
- 如果串长不超过 255(例如 Windows 操作系统的文件名)，哪种方案比较合适？

### 3.3 程序设计实例：公司更名

随着合并和收购事件的日益频繁，公司更名事件已经屡见不鲜了。除此之外，某些公司还会试图用改名来摆脱昔日的坏名声，或是在名字后面加个后缀 “.com” 来提高股价！

这些变化使得人们阅读一些早期文档时难以把一系列公司的曾用名和现在的新名字对应起来。你的公司 Digiscam (以前叫 Algorist Technologies ) 需要你写一个程序来维护公司的更名数据库，并且用文本替换的方式更新一批“过时”的文档中的公司名。

你的程序应当接受一系列的公司更名事件，然后是若干待更新的文本。只有精确匹配的字符串才能被替换。最多有 100 个更名事件，每行文本不超过 1000 个字符。一个样例输

- 50 -

入如下：

4

"Anderson Consulting" to "Accenture"

## "Enron" to "Dynegy"

"DEC" to "Compaq"

"TWA" to "American"

5

Anderson Accounting begat Anderson Consulting, which offered advice to Enron before it DECLARED bankruptcy, which made Anderson

Consulting quite happy it changed its name  
in the first place!

相应的输出应该是：

Anderson Accounting begat Accenture, which offered advice to Dynegy before it CompaqLARED bankruptcy, which made Anderson Consulting quite happy it changed its name in the first place!

本题并不要求你考虑单词分隔符(如空格),因此把 DECLARED 转换成 CompaqLARED 实际上是正确的。

以下为题解

为了解决这个问题，我们需要哪些类型的字符串操作？我们必须能够读取和储存一些字符串，在它们之中寻找一些模式，修改它们，并最终把结果打印出来。

注意输入被分成两个部分。其中第一部分，即更名数据库，必须在转换文本之前全部读取并保存下来。下面是相关数据结构：

```
#include <string.h>

#define MAXLEN          1001    /* longest possible string */
#define MAXCHANGES      101     /* maximum number of name changes */

typedef char string[MAXLEN];

string mergers[MAXCHANGES][2]; /* store before/after corporate names */
int nmergers;                /* number of different name changes */
```

我们用字符串二维数组来表示字典。不必把关键字按照任何一种特定的方式排序，因为我们在处理每行文本的时候都会重新扫描整个数据库。

读取公司名称列表略显复杂，因为我们需要对读取的每行进行解析，来获取引号之间的部分。一个技巧是忽略第一个引号之前的文本，然后顺次搜集字符，直到第二个引号出现：

```
read_changes()
```

-f-

```
int i; /* counter */

scanf("%d\n", &nmergers);
for (i=0; i<nmergers; i++) {
    read_quoted_string(&(mergers[i][0]));
    read_quoted_string(&(mergers[i][1]));
```

```
int i=0;          /* counter */  
char c;          /* latest character */
```

```
while ((c=getchar()) != '\n') ;  
while ((c=getchar()) != '\n') {  
    s[i] = c;  
    i = i+1;
```

}

1

解题所需的高级操作将在接下来的几节中介绍。

### 3.4 模式查找

在文本  $t$  中查找模式串  $p$  的最简单算法莫过于尝试着用模式串匹配文本串的每个位置，检查模式串中的每个字符是否都和它对应的文本串字符一致：

/\* Return the position of the first occurrence of the pattern p  
in the text t, and -1 if it does not occur.

```
/*
int findmatch(char *p, char *t)
{
    int i,j;                      /* counters */
    int plen, tlen;                /* string lengths */

    plen = strlen(p);
    tlen = strlen(t);

    for (i=0; i<=(tlen-plen); i=i+1) {
        j=0;
        while ((j<plen) && (t[i+j]==p[j]))
            j = j+1;
        if (j == plen) return(i);
    }

    return(-1);
}
```

注意这段例程只寻找精确匹配。如果某个字母在模式串中是大写，而文本串中是小写，就不算匹配。更加严重的是，如果公司名跨越了两行（见样例输入），那么也将检测不出这个匹配。如果要加入对这两种情况的处理，你只需把文本串/模式串的比较代码  $t[i+j]==p[j]$  修改一下。用同样的技巧，你还可以处理通配符 (*wild card*)，它能匹配任何字符。关于近似串匹配 (approximate string matching) 的一般概念和讨论参见第 11 章。

刚才介绍的朴素算法在最坏情况下需要  $O(|p||t|)$  的运算时间。你能否构造出一个任意长度的模式串和文本，使得朴素算法在经过这样长时间的运算后仍然找不到匹配？在通常情况下，朴素算法要高效得多，因为一旦发现单个字符失配，就可以立刻换下一个位置检查。更加复杂的线性时间模式匹配算法是存在的。[\[Gus97\]](#) 对此进行了完整的讨论。一般说来，这些算法已经在你最喜欢的编程语言的字符串库函数中实现了。

### 3.5 字符串操作

在本节中，我们假设字符串用以空字符结束的单字节字符序列表示，正如 C 语言中的那样。

把字符串当作数组来处理使得很多操作都变得十分简单：

- **计算字符串长度** — 从头扫描字符串中的每个字符，每遇到一个字符就给计数器加 1，直到空字符。
- **拷贝字符串** — 除非你的编程语言支持用一条语句拷贝整个数组，否则你需要显式的用循环语句一次拷贝一个字符。不要忘记为新字符串申请足够大的空间，更不要忘记串尾的空字符！
- **反转字符串** — 最简单的方法是按照从右往左的顺序把原串拷贝到另一个数组中。每个字符的新位置可以借助字符串长度计算出。不要忘记在新串的末尾增加空字符！字符串的反转操作也可以利用交换字符的方式原地 (in place) 进行<sup>5</sup>，但同时会破坏原来的串。

作为一个例子，我们实现了如下的函数，把一个字符串中特定起始位置和长度的子串用另一个字符串代替。我们需要在公司更名程序中用到它。容易出错的地方在于，为了得到新串，我们需要移动剩下的字符。如果替换成的子串比原始子串长，应把原串的后缀往右移，以腾出空间。如果替换成的子串比原始子串短，应把原串的后缀往左移，以免出现空档。

```
/* Replace the substring of length xlen starting at position pos in
   string s with the contents of string y.
*/
replace_x_with_y(char *s, int pos, int xlen, char *y)
{
    int i;                                /* counter */
    int slen,ylen;                         /* lengths of relevant strings */

    slen = strlen(s);
    ylen = strlen(y);

    if (xlen >= ylen)
        for (i=(pos+xlen); i<=slen; i++) s[i+(ylen-xlen)] = s[i];
    else
        for (i=slen; i>=(pos+xlen); i--) s[i+(ylen-xlen)] = s[i];

    for (i=0; i<ylen; i++) s[pos+i] = y[i];
}
```

另一个方法是在一个临时缓冲区里构造新串，然后用它覆盖 *s* 的所有字符。

<sup>5</sup> 译者注：即不借助于辅助空间。



### 3.6 程序的完成

到现在为止，所有用到的子程序都已经完成，剩下的部分就变得十分简单了：

```

main()
{
    string s;                      /* input string */
    char c;                        /* input character */
    int nlines;                    /* number of lines in text */
    int i,j;                       /* counters */
    int pos;                       /* position of pattern in string */

    read_changes();
    scanf("%d\n",&nlines);
    for (i=1; i<=nlines; i=i+1) {           /* read text line */
        j=0;
        while ((c=getchar()) != '\n') {
            s[j] = c;
            j = j+1;
        }
        s[j] = '\0';

        for (j=0; j<nmergers; j=j+1)
            while ((pos=findmatch(mergers[j][0],s)) != -1) {
                replace_x_with_y(s, pos,
                                     strlen(mergers[j][0]), mergers[j][1]);
            }

        printf("%s\n",s);
    }
}

```

### 3.7 字符串库函数

不管你用的是 C、C++ 还是 Java，都需要了解编程语言为字符和字符串提供的各种函数和类。你没有必要重新实现它们。

## C 的字符串函数库

C 语言中既有字符函数库，又有字符串函数库。其中，字符函数库 ctype.h 包含一些基于字符编码的简单测试和操作。根据 C 语言的规定，任何非 0 整数值都代表“真”，0 代表“假”。

```
#include <ctype.h>      /* include the character library */

int isalpha(int c);      /* true if c is either upper or lower case */
int isupper(int c);      /* true if c is upper case */
int islower(int c);      /* true if c is lower case */
int isdigit(int c);      /* true if c is a numerical digit (0-9) */
int ispunct(int c);      /* true if c is a punctuation symbol */
int isxdigit(int c);     /* true if c is a hexadecimal digit (0-9,A-F) */
int isprint(int c);      /* true if c is any printable character */

int toupper(int c);      /* convert c to upper case -- no error checking */
int tolower(int c);      /* convert c to lower case -- no error checking */
```

请认真检查每个函数的定义，在此之前不要假定它的作用和你想象的完全相同。

下列函数在 C 语言字符串函数库 string.h 中。该函数库还有很多其他函数和功能，读者不妨一试。

```
#include <string.h>      /* include the string library */

char *strcat(char *dst, const char *src);      /* concatenation */
int strcmp(const char *s1, const char *s2);      /* is s1 == s2? */
char *strcpy(char *dst, const char *src);      /* copy src to dist */
size_t strlen(const char *s);                  /* length of string */
char *strstr(const char *s1, const char *s2);  /* search for s2 in s1 */
char * strtok(char *s1, const char *s2);        /* iterate words in s1 */
```

## C++ 的字符串函数库

除了支持 C 语言风格的字符串外，C++ 还拥有一个字符串类，包含如下操作（还有更多操作未列出）：

```
string::size()          /* string length */
string::empty()          /* is it empty */

string::c_str()          /* return a pointer to a C style string */
```

```
string::operator [](size_type i)           /* access the ith character */

string::append(s)             /* append to string */
string::erase(n,m)           /* delete a run of characters */
string::insert(size_type n, const string&s) /* insert string s at n */

string::find(s)
string::rfind(s)            /* search left or right for the given string */

string::first()
string::last()               /* get characters, also there are iterators */
```

字符串类为连接和比较操作重载了相应的运算符。

### Java 的字符串对象

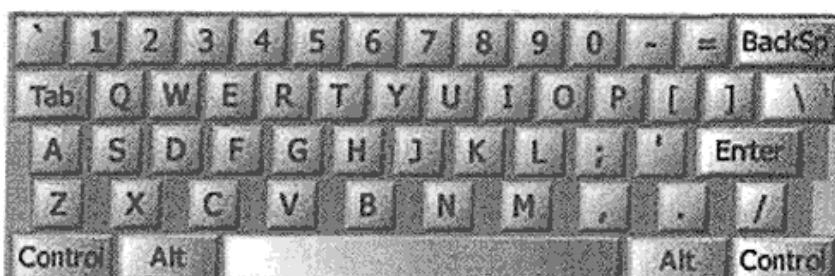
Java 字符串都是第一类对象 (first-class objects), 不管是由 String 类还是 StringBuffer 类实例化而来。String 类适用于静态的、永远不变的字符串, 而 StringBuffer 是为动态字符串而设计的。回忆前面所说的, Java 支持 Unicode, 它的字符都是 16 位的。

java.text 包中包含了更加高级的字符串操作, 包括对日期和其他结构化文本的解析。

## 3.8 习题

### 3.8.1 WERTYU 键盘 (WERTYU)

PC/UVa 题号: 110301/10082, 流行度: A, 通过率: high 难度: 1



一个常见的打字错误是把手放在了正确位置的右边一格, 然后将 “Q” 打成 “W”, 将 “J” 打成 “K” 等。你的任务是翻译在这种情况下打印出的信息。

输入

输入包含若干行。每一行可能包括数字、空格、大写字母（除了“Q”，“A”和“Z”）或者是上面出现过的标点符号（除了后引号‘）。键名为单词的（Tab, BackSp, Control等）也不会出现在输入中。

输出

你需要根据上面所给出的键盘布局，将每个字母或标点用它左边的一个键替换。空格在输出文件中应原样输出。

## 样例输入

## 样例输出

O S. GOMR YPFSU/

I AM FINE TODAY.

### 3.8.2 寻找单词 (Where's Waldorf?)

PC/UVa 题号: 110302/10010, 流行度: B, 通过率: average 难度: 2

给出一个由字母组成的  $m \times n$  矩阵和一份单词列表，在格子中找出每一个可以被找到的单词的位置。

单词必须和矩阵中一条不间断的直线匹配，大小写不敏感（即大写字母和相应的小写字母被认为是相同的）。这条直线可以是水平、竖直、对角线的八个方向之一。

## 输入

输入第一行只有一个正整数，表示测试数据的组数。接下来有一个空行。相邻两组数据之间也有一个空行。

每组数据的第一行只有两个十进制数  $m$  和  $n(1 \leq m, n \leq 50)$ 。接下来的  $m$  行每行有  $n$  个字母，即一个包含大小写字母的矩阵。下一行只有一个整数  $k(1 \leq k \leq 20)$ 。接下来的  $k$  行是待查找的单词，每行一个。这些单词只包含大小写字母，没有空格、连字符或者其他非字母的字符。

输出

对于每组数据中的每个单词，输出一对整数代表它在矩阵中的位置，以一个空格隔开。第一个数代表这个单词的首字母在第几行 ( $1$  代表最上面的一行,  $m$  代表最下面的一行)。第二个数代表这个单词的首字母在第几列 ( $1$  代表最左边的一列,  $n$  代表最右边的一列)。如果一个单词在表格中出现不止一次，输出最上边的位置 (即最靠近矩阵顶端的位置)。如果有两个或两个以上的位置都在最上边，输出它们之中最左边的那个。保证所有的单词都可以在表格中找到。

输出时相邻两组数据用一个空行隔开。

## 样例输入

1  
8 11  
abcDEFGhigg  
hEbkWalDork  
FtyAwaldORm  
FtsimrLqsrc  
byoArBeDeyv  
Klcbqwicomk  
strEBGadhrb  
yUiqlxcnBjf  
4  
Waldorf  
Bambi  
Betty  
Dagbert

## 样例输出

2 5  
2 3  
1 2  
7 8

## 3.8.3 公共排列 (Common Permutation)

PC/UVA 题号: 110303/10252, 流行度: A, 通过率: average 难度: 1

给出两个字符串  $a$  和  $b$ , 输出一个满足如下条件的最长串  $x$ :  $x$  的某两个排列分别是  $a$  和  $b$  的(不必连续的)子序列 (subsequence)。

## 输入

输入包括若干组数据, 每组数据均由连续的两行组成。换句话说, 第1行和第2行是一组测试数据, 第3行和第4行是另一组测试数据等。输入的每行都是一个由小写字母组成的字符串, 其中第一行代表  $a$ , 第二行代表  $b$ 。每个字符串的长度不超过 1000。

## 输出

对于每组数据, 在单独一行输出  $x$ 。如果有多个  $x$  满足条件, 输出字典序最小的一个。

## 样例输入

pretty  
women

## 样例输出

e  
nw

```

walking           et
down
the
street

```

### 3.8.4 解密 II(Crypt Kicker II)

PC/UVa 题号: 110304/850, 流行度: A, 通过率: average 难度: 2

重排字母表中的各个字母是一种常用但不安全的加密方式。具体来说，字母表中的每个字母始终都被另外某个字母替换。为了保证加密过程是可逆的，任何两个字母都不能被同一个字母替换。

密码学中有一种强有力的破解手段称为已知明文攻击 (*known-plaintext attack*)。在这种攻击手段中，密码专家先要设法知道原文中的短语或句子，然后通过观察密文来推测加密方法。

你的任务是将密文破解成原文。假设每一行用的是相同的替换方案，并且其中一行的原文是 `the quick brown fox jumps over the lazy dog`。

#### 输入

输入第一行只有一个正整数，表示测试数据的组数。接下来有一个空行。相邻两组数据之间也有一个空行。

每组数据由若干行组成，用上面提到的方法加密。密文由小写字母和空格组成，并且每行不超过 80 个字符。每组数据的输入最多有 100 行。

#### 输出

对于每一组数据，依次输出每一行的原文。如果有多种解密方式，可以输出任何一种。如果无法解密，输出

`No solution.`

相邻两组数据的输出之间用一个空行隔开。

#### 样例输入

1

```

vtz ud xnm xugm itr pyy jttk gmv xt otgm xt xnm puk ti xnm fprxq
xnm ceuob lrtzv ita hegfd tsmr xnm ypwq ktj
frtjrpqgguvj otvxmdxd prm iev prmvx xnmq

```

#### 样例输出

```
now is the time for all good men to come to the aid of the party
```

```
the quick brown fox jumps over the lazy dog
programming contests are fun aren't they
```

### 3.8.5 自动评测脚本 (Automated Judge Script)

PC/UVa 题号: 110305/10188, 流行度: B, 通过率: average 难度: 1

程序设计竞赛中的真人裁判是出了名的挑剔。为了代替他们的工作, 请编写一个自动评测脚本, 用来评测参赛队伍所提交的程序。

你的程序需要输入正确答案和选手程序的输出, 返回 Accepted、Presentation Error 或 Wrong Answer 之一。这些返回值的定义如下:

**Accepted:** 当选手输出和正确答案完美匹配时返回 “Accepted”。所有字符和它们的出现顺序必须完全一致。

**Presentation Error:** 如果所有数字字符和它们出现的顺序完全一致, 但至少有一个非数字字符不一致, 返回 “Presentation Error”。例如, “15 0” 和 “150” 应该返回 “Presentation Error”, 但 “15 0” 和 “1 0” 应该返回 “Wrong Answer”(见下)。

**Wrong Answer:** 其他情况都应该返回 “Wrong Answer”。

#### 输入

输入包含多组数据。每组数据的第一行包含一个正整数  $n < 100$ , 即正确答案的行数。以下  $n$  行为正确答案。接下来的一行包含一个正整数  $m < 100$ , 即选手输出的行数。以下  $m$  行为选手输出。整个输入以  $n = 0$  的数据结束, 你不必处理这组数据。

每一行均不超过 100 个字符。

#### 输出

对于每组数据, 输出以下情况之一:

Run #x: Accepted

Run #x: Presentation Error

Run #x: Wrong Answer

其中  $x$  代表测试数据序号 (从 1 开始编号)。

#### 样例输入

```
2
The answer is: 10
```

#### 样例输出

```
Run #1: Accepted
Run #2: Wrong Answer
```

The answer is: 5 Run #3: Presentation Error  
2 Run #4: Wrong Answer  
The answer is: 10 Run #5: Presentation Error  
The answer is: 5 Run #6: Presentation Error  
2  
The answer is: 10  
The answer is: 5  
2  
The answer is: 10  
The answer is: 15  
2  
The answer is: 10  
The answer is: 5  
2  
The answer is: 10  
The answer is: 5  
3  
Input Set #1: YES  
Input Set #2: NO  
Input Set #3: NO  
3  
Input Set #0: YES  
Input Set #1: NO  
Input Set #2: NO  
1  
1 0 1 0  
1  
1010  
1  
The judges are mean!  
1  
The judges are good!  
0

### 3.8.6 文件碎片 (File Fragmentation)

PC/UVa 题号: 110306/10132, 流行度: C, 通过率: average 难度: 2

你有一个学生物化学的朋友, 手拿计算机文件穿梭于实验室间的时候不慎跌倒。所有文件散落一地, 摔坏了。他拾起所有文件碎片, 然后找你帮忙把它们恢复原状。

幸运的是, 所有文件都是完全相同的, 每个文件都恰好断裂成两个碎片, 并且所有文件碎片都已被找到。不幸的是, 并不是所有文件都在同一位置断裂, 而且当它们散落一地之后, 碎片的顺序被完全打乱。

原始的二进制文件碎片已经转化成了由 ASCII 字符1和0组成的比特串。你的任务是确定每个文件的内容 (也用比特串表示)。

#### 输入

输入第一行包含一个正整数, 表示测试数据的组数。接下来是一个空行。相邻两组数据之间也有一个空行。

每个数据包含若干个“文件碎片”, 每个碎片单独占一行, 以文件结束符或者空行结束。每个文件碎片由 ASCII 字符1和0组成。

#### 输出

对于每组输入, 输出一行由 ASCII 字符1和0组成的串, 表示原始文件的内容。如果输入中有  $2N$  个文件碎片, 则必然存在一种方法把它们分成  $N$  个有序对, 分别顺次连接后将得到输出串的  $N$  份拷贝。如果答案不唯一, 可以输出其中的任何一个。

原始文件不超过 144 个, 且文件大小不超过 256 字节。

相邻两组数据的输出之间应用一个空行隔开。

#### 样例输入

1

011  
0111  
01110  
111  
0111  
10111

#### 样例输出

01110111

### 3.8.7 Doublet 序列 (Doublets)

PC/UVa 题号: 110307/10150, 流行度: C, 通过率: average 难度: 3

一个 *doublet* 是一对长度相同、只有一个字母不同的单词，例如 “booster” 和 “rooster”、“rooster” 和 “roaster” 以及 “roaster” 和 “roasted”。

你有一本不超过 25 143 个小写单词的词典，每个单词不超过 16 个字母。给你若干对单词，要求对于每对单词，找出由第一个单词开始、第二个单词结束的最短单词序列，使得该序列中的每对相邻单词都是 doublet。例如，对于 “booster” 和 “roasted” 这对单词，一个合法的序列为 (“booster”、“rooster”、“roaster”、“roasted”)，如果上述单词都在字典内。

#### 输入

输入包含字典和一些单词对。字典由若干个单词组成，每行一个，以一个空行结束。接下来是若干对单词，每对单词占据单独的一行，用单个空格隔开。

#### 输出

对于每组输入的单词对，输出一个由第一个单词开头、第二个单词结尾的单词序列。每对相邻的单词都必须是 doublet。

如果最短单词序列不唯一，可以输出任何一种。如果无解，输出一行：“No solution.”。相邻两组数据的输出之间用一个空行隔开。

#### 样例输入

```
booster
rooster
roaster
coasted
roasted
coastal
postal
```

#### 样例输出

```
booster
rooster
roaster
roasted
No solution.
```

```
booster roasted
coastal postal
```

### 3.8.8 Fmt 程序 (Fmt)

PC/UVa 题号: 110308/848, 流行度: C, 通过率: low 难度: 2

Unix 系统的 *fmt* 程序读入若干行文本，将它们合并后重新换行输出一个新的文本，使得每行的字符数尽量接近，但不超过 72 个字符。合并与换行规则如下：

- 可以从输入文本中的任意一个空格处换行，然后去掉前一行的行末空格和新行的行首空格。
- 输入中的换行符可以去掉，除非 (1) 它在一个空白行 (blank line) 或者空行 (empty line) 的末尾，或者 (2) 它的下一个字符为空格或者另一个换行符。当换行符需要被去掉时，你应该用一个空格来代替它。
- 输出中每行的行末空格都应被删除。
- 输入中的所有长度超过 72 的单词必须在输出中单独占一行。  
你可以假定输入文本中不含有水平制表符。

### 样例输入

Unix fmt

The unix fmt program reads lines of text, combining and breaking lines so as to create an output file with lines as close to without exceeding 72 characters long as possible. The rules for combining and breaking lines are as follows.

1. A new line may be started anywhere there is a space in the input. If a new line is started, there will be no trailing blanks at the end of the previous line or at the beginning of the new line.

2. A line break in the input may be eliminated in the output, provided it is not followed by a space or another line break. If a line break is eliminated, it is replaced by a space.

### 样例输出

Unix fmt

The unix fmt program reads lines of text, combining and breaking lines so as to create an output file with lines as close to without exceeding 72 characters long as possible. The rules for combining and breaking lines are as follows.

1. A new line may be started anywhere there is a space in the input.

If a new line is started, there will be no trailing blanks at the end of the previous line or at the beginning of the new line.

2. A line break in the input may be eliminated in the output, provided it is not followed by a space or another line break. If a line break is eliminated, it is replaced by a space.

## 3.9 提 示

- 3.1 你打算用硬编码 (hard-coded) 逻辑来替换字符, 还是觉得用已初始化数组的表驱动 (table-driven) 策略能让问题简化?
- 3.2 你能否写出一个带参数的比较函数, 使得所有八个方向的比较只需八次参数不同的函数调用? 用整数对  $(\delta_x, \delta_y)$  而不是名称来表示各个方向是否合适?
- 3.3 是否能把两个单词内的字母分别重排, 使得二者的公共排列更加明显?
- 3.5 在判断“格式错”的时候, 只比较数字字符的最简单的方法是什么?
- 3.6 你是否可以轻易地弄清哪些碎片应该拼到一起, 甚至确定它们的顺序?
- 3.7 是否可以为本题建立一个与路径有关的图论模型? 先读读第 9 章也许对解决本题有所帮助。我们将在该部分介绍图的数据结构和遍历算法。

## 3.10 注 解

- 3.4 尽管在数千年前就已诞生, 密码学 (cryptography) 近期的重大变革源于计算方法的进步和新算法的出现。如果对这个引人入胜的领域感兴趣, 请阅读 Schneier 的 [Sch94] 和 Stinson 的 [Sti02]。
- 3.8 文本排版程序的黄金标准 (gold standard) 是 L<sup>A</sup>T<sub>E</sub>X。本书正是用它排版的。L<sup>A</sup>T<sub>E</sub>X构建于 T<sub>E</sub>X之上。后者出自计算机科学界闻名于世的大师 Don Knuth(高德纳) 之手。高德纳的著作《计算机程序设计艺术》[Knu73a, Knu81, Knu73b] 从出版至今, 30 年来一直无法被超越, 已被世人奉为经典。

## 第4章 排序

排序是计算机科学中最基础的算法问题，也是编程题目的重要来源。原因有两点。首先，排序能有效地完成每个程序员都可能遇到的众多任务。只要发现你的任务是排序问题的特殊情况，就可以合理地使用库函数来减小工作量。其次，人们已经发明出了许多不同的排序算法，每种算法都有它自身的独到之处。大多数算法设计方法与技巧都体现在了不同的排序算法中，包括分治法，随机化，增量法以及高级数据结构的使用。很多有趣的编程题目或者数学题都是基于算法的这些性质。

在本章中，我们将看到排序的一些主要应用以及大多数重要的排序算法背后的理论。最后，我们将描述常见程序设计语言中与排序相关的库函数，并且展示如何用它们来解决一个不算简单的题目。

### 4.1 排序的应用

理解排序的关键在于懂得如何用它完成一些重要的编程任务：

- 唯一性测试 — 如何测试一个给定序列  $S$  中的元素是否两两互异？把它们按照升序或者降序排列，则重复元素一定会排在一起。接下来，只需做一遍扫描，检查是否存在  $1 \leq i < n$  满足  $S[i] = S[i+1]$  即可。
- 删除重复元素 — 对于  $S$  中的每个重复出现的元素，如何删除多余的拷贝，只保留一份？排序和扫描仍然适用。注意这里的扫描需要维护两个下标： $\text{back}$  指向已处理完的部分（它是原序列的前缀）的最后一个元素， $i$  指向下一个需要考虑的元素。如果  $S[\text{back}]$  不等于  $S[i]$ ，则给  $\text{back}$  加 1，并且把  $S[i]$  拷贝到  $S[\text{back}]$ 。
- 按优先级处理事件 — 假设我们有一些需要完成的工作，每项工作都有截止期。按照截止期（或者相关的准则）排序后，这些工作就已经处于正确的处理顺序了。优先队列在维护有插入删除的日程表时十分有用，但如果事件始终保持不变，只做一次排序就完全足够了。
- 中位数/顺序选择 — 假设我们要寻找序列  $S$  中的第  $k$  大元素。把元素降序排列后，所求元素就是  $S[k]$ 。作为特殊情况，这个方法也可以用来（尽管效率差了点）求最小、最大值和中位数。
- 频率统计 — 哪个元素在序列  $S$  中出现得最频繁（即：如何寻找序列的模式（mode）？排序后，一次线性扫描可以统计出每个元素出现的频率。
- 恢复原始顺序 — 如果出于某种原因，我们把一些元素做了重排，如何恢复它们原来的顺序？给每个元素增加一个附加域，使得原始顺序下的第  $i$  个元素的附加域值为  $i$ 。只

要在重排的时候让附加域和数据一起移动，按照附加域排序即可让它们恢复原来的顺序。

- 集合的交与并 — 如何求两个多重集的交与并？如果二者都用升序序列表示，则只需不断拿走两个序列首部元素中的较小值，放到新序列的尾部（对于交集，只有当两个首部元素相同时才进行此操作），并且删除序列首元素（不相同时只删除较小值，相同时一起删除）。
- 寻找目标数对 — 给定一个目标数  $z$ ，如何检测是否存在两个整数  $x, y \in S$  使得  $x + y = z$ ？为了更加高效地完成这个目标，我们并不直接判断所有可能的数对，而是把  $S$  里的所有整数按照升序排序，然后扫描。当  $s[i]$  随着  $i$  的增加而增加时，它的可能搭档（即满足  $s[j] = z - s[i]$  的  $j$ ）递减。只要在  $i$  增加时合理地控制  $j$  递减，便可在一次扫描内找到所有可能的解。
- 高效查找 — 如何高效地判断一个元素  $s$  是否在序列  $S$  中？“把序列排序后用二分查找来回答询问”也许是排序算法最常见的应用了。但是不要忘记刚才讨论过的其他内容！

## 4.2 排序算法

你很可能已经听说过很多不同的排序算法了。还记得冒泡排序、插入排序、选择排序、堆排序、归并排序、快速排序、基数排序、桶排序、希尔排序、树的中序遍历以及排序网络吗？你很可能在还没有看完这个列表之前，目光就已经开始有呆滞的迹象了。干嘛要学用这么多种不同的方法来做同一件事？更何况，绝大多数编程语言已经配备了排序相关的库函数。

学习排序算法的真正原因在于这些算法背后的思想。这些思想广泛地应用于很多其他的问题中。堆排序的精髓在于数据结构；快速排序的精髓在于随机化；归并排序的精髓在于分治法。懂得了这些之后，你将拥有一大类算法工具。

下面，我们讨论几个特别具有指导意义的算法。请留意每个算法所具备的有用特性（例如，最小化数据搬移的工作量）。

- 选择排序 — 该算法把输入数组分成有序和无序两部分，然后在每次迭代中选择无序部分的最小元素，并移动到有序部分的尾部：

```
selection_sort(int s[], int n)
{
    int i, j;                  /* counters */
    int min;                   /* index of minimum */

    for (i=0; i<n; i++) {
        min=i;
```

```
    for (j=i+1; j<n; j++)
        if (s[j] < s[min]) min=j;
    swap(&s[i],&s[min]);
}
}
```

选择排序进行了大量的比较，但如果我们只关心数据移动的次数，它是相当高效的。算法只进行了  $n - 1$  次交换操作，而在最坏情况下这是必须的：考虑对一个逆序排列进行排序的情形。选择排序还展示了高级数据结构的威力：如果用高效的优先队列来维护无序部分，则  $O(n^2)$  的简单选择排序摇身一变，成为  $O(n \lg n)$  的堆排序！

- 插入排序 — 该算法同样维护有序和无序两部分。在每一次迭代中，无序部分中的下一个元素被插入到有序部分中的合适位置：

```
insertion_sort(int s[], int n)
{
    int i,j; /* counters */

    for (i=1; i<n; i++) {
        j=i;
        while ((j>0) && (s[j] < s[j-1])) {
            swap(&s[j],&s[j-1]);
            j = j-1;
        }
    }
}
```

排列中的逆序对 (*inversion*) 是顺序不正确的一对元素，即满足  $i < j$  但  $p[i] > p[j]$  的二元组  $(i, j)$ 。插入排序中的每次交换恰好消除一个逆序对，而且没有其他元素被移动，因此交换总次数等于逆序对数。由于逆序对很少，插入排序对于“几乎有序”的输入非常有效。

- 快速排序 — 该算法把“给一个大数组排序”的任务归约为“给两个小数组排序”。归约的关键在于一个称为序列划分 (*partition*) 的步骤，它把序列分成两个子序列：不超过轴值 (*pivot*) 的部分和严格大于轴值的部分。由于每个子序列中的元素都不会被移出，接下来只需分别排序即可。为了方便程序实现，我们在函数 `quicksort` 的参数列表中包含了待排序子序列中的第一个元素 (*l*) 和最后一个元素 (*h*) 在数组 *s* 中的下标。

```
quicksort(int s[], int l, int h)
{
    int p; /* index of partition */
```

```
if ((h-1)>0) {
    p = partition(s,l,h);
    quicksort(s,l,p-1);
    quicksort(s,p+1,h);
}

int partition(int s[], int l, int h)
{
    int i;                      /* counter */
    int p;                      /* pivot element index */
    int firsthigh;               /* divider position for pivot */

    p = h;
    firsthigh = l;
    for (i=l; i<h; i++)
        if (s[i] < s[p]) {
            swap(&s[i],&s[firsthigh]);
            firsthigh++;
        }
    swap(&s[p],&s[firsthigh]);

    return(firsthigh);
}
```

快速排序的有趣之处有几点：如果实现得当，它是最快的内排序算法 (in-memory sorting algorithm)；它还是展示递归 (recursion) 威力的一个优美实例；partition 算法本身还可以独立解决很多问题，例如把仅包含 0 和 1 的数组排序，让所有的 0 和 1 分别连续。

### 4.3 程序设计举例：给绅士排名

年轻貌美的 Polly 小姐身边从来都不缺少绅士追求者。事实上，她最大的烦恼在于如何弄清楚他们中的哪些人是最棒的。聪明的她想用一个计算机程序来给这些绅士们排名，而且成功地说服了你来替她编写这个程序。

Polly 喜欢跳舞，她心中的理想伴侣应该拥有 1 米 80 的身高。因此，她的第一条排序准则是：身高尽量接近她心中的理想值（在差值相同的情况下，略高还是略低是没有区别的）。身高差值相同的绅士，体重不超过 75 千克的比超过的好。不超过 75 千克的绅士中，体重越接近这个值越好；而体重超过 75 千克的绅士中，体重越轻越好。如果以上准则仍然无法区分某两个绅士，那么在输出时先按姓（last name），再按名（如果需要的话）排序输出。

Polly 只关心最后的排序结果，因此输入文件：

George Bush	195	110
Harry Truman	180	75
Bill Clinton	180	75
John Kennedy	180	65
Ronald Reagan	165	110
Richard Nixon	170	70
Jimmy Carter	180	77

的正确输出为：

Clinton, Bill  
Truman, Harry  
Kennedy, John  
Carter, Jimmy  
Nixon, Richard  
Bush, George  
Reagan, Ronald

#### 以下为题解

本题的关键在于用一个定义在多个域上的复杂准则来排序。至少有两种常见做法。第一种做法是多趟排序。先按最不重要的关键字排序，然后是倒数第二重要的关键字等。最后才是按最重要的关键字排序。

为何用这样的次序排序？次要的关键字仅仅用来给基于主关键字的排序结果打破平局（break tie）。只要使用稳定（stable）的排序算法（即保持关键字相同的元素在输入中的顺序），我们对次要关键字的排序结果将保留到最后，只要它和最终结果有关。

不是所有的排序算法都是稳定的：事实上，大部分快速的算法都是不稳定的！插入排序和选择排序都可以实现成稳定的<sup>1</sup>，但快速排序不是稳定的。在使用排序函数之前，请仔细阅读相关文档，而不要直接假定它们是稳定的。

第二种做法，也是我们将选用的方法，是将这个准则实现成一个复杂的比较函数。这样

<sup>1</sup> 译者注：前面的插入排序实现并不是稳定的，读者可以思考如何把它改成稳定的。

可以方便地使用我们即将讨论的：C 语言自带的排序函数。

## 4.4 与排序相关的库函数

只要有可能，请尽量使用编程语言自带的排序和查找库函数。

### C 中的排序与查找

头文件 `stdlib.h` 声明了排序和查找的相关函数。对于排序，有这样一个函数 `qsort`：

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
           int (*compare) (const void *, const void *));
```

使用 `qsort` 的关键是弄清楚各个参数的含义。待排序数组的首地址为 `base`，元素个数为 `nel`，每个元素占 `width` 个字节。因此，待排序的元素可以是单字节字符、4 字节的整数，或者是 100 个字节的记录。只需要修改 `width` 即可。

元素之间的序关系由函数 `compare` 指定。该函数接受两个待比较元素的指针（别忘了每个元素占 `width` 个字节），当第一个元素小于<sup>2</sup> 第二个元素时返回负数，大于时返回正数，相等时返回 0。

下面是一个按升序给整数排序的比较函数：

```
int intcompare(int *i, int *j)
{
    if (*i > *j) return (1);
    if (*i < *j) return (-1);
    return (0);
}
```

使用这个比较函数，我们可以把包含 `cnt` 个整数的数组 `a` 按升序排列：

```
qsort((char *) a, cnt, sizeof(int), intcompare);
```

我们将在 4.5 节中看到一个更加复杂的实例。函数 `qsort` 的名称表明了它使用了快速排序算法，但这一点通常和用户无关。

注意，`qsort` 将破坏待排序数组的内容，因此如果你需要恢复那些元素本来的顺序，要么在排序之前保留原数组的一个备份，要么使用我们在 4.1 节中提到的增加附加域的方法。

在有压力的情况下，二分查找相当容易写错。最好的方法是不要自己实现，而是直接使用 `stdlib.h` 提供的 `bsearch()`。除了新增参数 `key` 表示待查找元素之外，其他参数和 `qsort` 并无区别。若要在刚刚那个排序好的数组中查找，可以用：

<sup>2</sup> 译者注：这里的“小于”关系应当理解为“在排序结果中位于前面”。

```
bsearch(key, (char *) a, cnt, sizeof(int), intcompare);
```

### C++ 中的排序与查找

我们在 2.2.1 小节中讨论过的 C++ 标准模板库 (STL) 包含了和排序、查找相关的很多函数。每一个立志学好 C++ 的程序员都应当熟悉 STL。

为了用 STL 来排序，我们要么使用类定义的缺省比较函数 (即 < 运算符)，要么传入一个比较函数 op：

```
void sort(RandomAccessIterator bg, RandomAccessIterator end) void
sort(RandomAccessIterator bg, RandomAccessIterator end,
      BinaryPredicate op)
```

STL 还提供了稳定排序函数，其中关键字相同的元素之间的相对顺序保持不变。这对于前面提到的多准则 (multiple criteria) 排序是有用的。

```
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end)
void stable_sort(RandomAccessIterator bg, RandomAccessIterator end,
                  BinaryPredicate op)
```

还有一些 STL 函数实现了 4.1 节中提到的部分排序应用，包括：

- 第  $n$  小元素 — 函数 nth\_element 返回指定容器中的第  $n$  小元素。
- 集合运算 — 函数 set\_union、set\_intersection、set\_difference 返回两个容器的并、交和差。
- 删除重复元素 — 函数 unique 删除所有连续出现的重复元素。

### Java 中的排序与查找

java.util.Arrays 类包含了排序和查找所用到的很多方法：

```
static void sort(Object[] a)
static void sort(Object[] a, Comparator c)
```

上述方法用对象的默认顺序<sup>3</sup> 或者自定义比较器 c。上述两个方法均采用稳定排序算法。

类似的，有序数组上的二分查找也有相应的方法：

```
binarySearch(Object[] a, Object key)
binarySearch(Object[] a, Object key, Comparator c)
```

## 4.5 给绅士排名

下面回到 Polly 的程序。该程序的难点在于让多准则的排序尽量简单。首先，我们需要定义基本的数据结构：

<sup>3</sup> 译者注：Java 语言内置的基本数据类型和类都有默认顺序。自定义类需要实现 Comparable 接口。

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

#define NAMELENGTH      30          /* maximum length of name*/
#define NSUITORS        100         /* maximum number of suitors */

#define BESTHEIGHT      180         /* best height in centimeters */
#define BESTWEIGHT       75          /* best weight in kilograms */

typedef struct {
    char first[NAMELENGTH];           /* suitor's first name */
    char last[NAMELENGTH];           /* suitor's last name */
    int height;                      /* suitor's height */
    int weight;                      /* suitor's weight */
} suitor;

suitor suitors[NSUITORS];           /* database of suitors */
int nsuitors;                      /* number of suitors */
```

接下来需要读取输入。注意我们并没有保存每个人的真实身高和体重！与其在比较函数中直接实现 Polly 小姐的标准，倒不如在输入的时候就修改身高和体重，使得二者都越小越好<sup>4</sup>。

```
read_suitors()
{
    char first[NAMELENGTH], last[NAMELENGTH];
    int height, weight;

    nsuitors = 0;

    while (scanf("%s %s %d %d\n", suitors[nsuitors].first,
                suitors[nsuitors].last, &height, &weight) != EOF) {
        suitors[nsuitors].height = abs(height - BESTHEIGHT);
        if (weight > BESTWEIGHT)
            suitors[nsuitors].weight = weight - BESTWEIGHT;
```

<sup>4</sup> 译者注：对于体重不超过 75 千克的绅士，体重是越大越好，因此相反数越小越好；由于负数小于正数，他们总是比超重的绅士好。

```
    else
        suitors[nsuitors].weight = - weight;

    nsuitors++;
}

}
```

请注意我们用 `scanf` 来逐一读取各个 token，而非单个字符。

核心的比较函数读取一对追求者 *a* 和 *b*，比较是 *a* 更优秀，*b* 更优秀，还是同样优秀。为了满足 `qsort` 的需求，我们需要在这三种情况下分别返回 -1, 1 和 0。以下是一个参考实现：

```
int suitor_compare(suitor *a, suitor *b)
{
    int result; /* result of comparison */

    if (a->height < b->height) return(-1);
    if (a->height > b->height) return(1);

    if (a->weight < b->weight) return(-1);
    if (a->weight > b->weight) return(1);

    if ((result=strcmp(a->last,b->last)) != 0) return result;

    return(strcmp(a->first,b->first));
}
```

在输入过程和比较函数都准备好之后，剩下的工作仅仅是编写一个主程序，输入数据、排序，最后产生 Polly 想要的绅士排名。下面是一个参考实现：

```
main()
{
    int i; /* counter */
    int suitor_compare();

    read_suitors();

    qsort(suitors, nsuitors, sizeof(suitor), suitor_compare);
```

```

for (i=0; i<nsuitors; i++)
    printf("%s, %s\n", suitors[i].last, suitors[i].first);
}

```

## 4.6 习 题

### 4.6.1 Vito 家族 (Vito's Family)

PC/UVa 题号: 110401/10041, 流行度: A, 通过率: high 难度: 1

著名的大盗 Vito Deadstone 将到纽约去。在那里，他有一个庞大的家族，所有人都住在 Lamafia 大道<sup>5</sup>。因为会经常拜访这些亲戚，他希望找一间离他们很近的房子。

事实上，Vito 希望让他的房子离所有亲戚的距离之和尽量小。他命令你写一个程序来解决这一问题。

#### 输入

输入包含若干个测试点。第一行为测试点的数量。

对于每组数据，你将会得到整数  $r$  ( $0 < r < 500$ )，表示 Vito 的亲戚数量以及他们所居住的街编号 (也是整数)  $s_1, s_2, \dots, s_i, \dots, s_r$ ，其中  $0 < s_i < 30\,000$ 。可能会有不同的亲戚住在同一条街上。

#### 输出

对于每一组数据，你的程序要输出从 Vito 的房子离所有亲戚家的总距离的最小值。 $s_i$  和  $s_j$  两条街之间的距离  $d_{ij} = |s_i - s_j|$ 。

#### 样例输入

```

2
2 2 4
3 2 4 6

```

#### 样例输出

```

2
4

```

### 4.6.2 煎饼堆 (Stacks of Flapjacks)

PC/UVa 题号: 110402/120, 流行度: B, 通过率: high 难度: 2

在架子上烤一堆完美的煎饼是一件技巧性很高的工作，因为无论你有多努力，烤出的饼的直径都会各不相同。为了整洁，你可以将所有的饼按大小排序，使得每个饼下面所有饼的直径都比它大。

<sup>5</sup> 译者注：粗略地说，纽约的曼哈顿区被南北走向的大道 (Avenue) 和东西走向的街 (Street) 分成有规律的网格。换句话说，每一条大道可以看成一条数轴，数轴上的位置用街号来标识。

为这堆饼排序是通过一系列的“翻转 (flip)”操作来实现的。有一把铲子可以用来插在两张饼之间，并将铲子之上的所有饼翻转过来。每次翻转操作用最下面一张被翻转的饼在整个堆饼的位置来描述。如果一共有  $n$  张饼，最底部的饼位置为 1，最上面的饼位置为  $n$ 。

一堆煎饼由从上到下各张煎饼的直径来表示。例如，下面的 3 堆饼，在最左边的一堆中，最上面的一张直径为 8：

8	7	2
4	6	5
6	4	8
7	8	4
5	5	6
2	2	7

最左边一堆可以通过操作 $flip(3)$ 变成中间的一堆，再通过操作 $flip(1)$ 变成右边的一堆。

输入

输入包含了若干堆饼的数据。每堆饼的张数在 1 到 30 之间，并且每个饼的直径是 1 到 100 之间的整数。输入用文件结束符终止。每堆饼的数据都在单独一行中给出，每行的第一个数代表最上面的那张饼的直径，最后一个数代表最下面那张饼的直径。相邻两个数被一个空格隔开。

输出

对于每一堆饼，你的程序要先用单独一行输出原始的顺序，然后在下一行输出能够让最大的饼在最下面，最小的饼在最上面的翻转操作序列。翻转序列应该以 0 结束，表示不必再进行多余的翻转。当一堆饼被排列好以后，不要再进行多余的动作。

### 样例输入

1	2	3	4	5
5	4	3	2	1
5	1	2	3	4

## 样例输出

```

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

```

### 4.6.3 过桥 (Bridge)

PC/UVa 题号: 110403/10037, 流行度: B, 通过率: low 难度: 3

有  $n$  个人希望在晚上通过一座桥。在任何时刻，最多只能有两个人在桥上，而且必须要有手电筒才能过桥。一共只有一个手电筒，所以必须安排某种顺序，使得手电筒可以被带

回去让更多的人过桥(手电筒必须由人带回,不可以从对岸扔过去)。

每个人都有不同的过桥时间，两个人一起过桥所花的时间等于其中较慢的一个。你的任务是要找出能在最短时间内使所有人都过桥的方案。

## 输入

输入第一行只有一个整数，代表测试点的数量。接下来有一个空行。相邻两组数据之间也有一个空行。

每组数据的第一行有一个整数  $n$ , 接下来有  $n$  行, 每一行给出一个人的过桥时间 (单位: 秒)。人数不超过 1000, 每个人的过桥时间不超过 100 秒。

输出

对于每组数据，第一行输出所有人过桥的总时间。接下来输出过桥策略，每一行有一个或两个整数，代表某个人或某两个人结伴过桥。每个人用过桥时间代表（尽管可能不同人的过桥时间相同，但这无关紧要）。

请注意过桥的方向是交替的，因为必须要将手电筒传回去让更多的人过桥。如果有多种方案，输出任何一种都可以。

相邻两组数据的输出之间应有一个空行。

### 样例输入

1 4 1 2 5 10

## 样例输出

17  
1 2  
1  
5 10  
2  
1 2

#### 4.6.4 最长打盹时间 (Longest Nap)

PC/UVa 题号: 110404/10191, 流行度: B, 通过率: average 难度: 1

教授们总是因为日程上排满了工作和会面而终日忙碌。P 教授喜欢在白天打盹，但他的安排实在太满了，以至于他并没有太多机会来打盹。

可是，他太想每天都能打个盹了。他想从满满的日程安排中挤出一段最长的空余时间来打盹，写一个程序来帮他吧。

输入

输入包含多组测试数据，每组数据表示一天的情况。每组数据的第一行包含一个正整数

数  $s \leq 100$ , 表示当天日程中的事件数。接下来  $s$  行按照  $\text{time1 time2 appointment}$  的格式描述事件, 其中  $\text{time1}$  表示事件开始的时间,  $\text{time2}$  表示事件结束的时间。所有的时间都按照  $hh:mm$  的格式表示; 结束时间一定严格在开始时间之后, 二者用一个空格隔开。

所有时间都会不早于 10:00, 不晚于 18:00, 因此你的结果也必须在这个区间之内, 即打盹不能在 10:00 之前开始, 也不能延续到 18:00 之后。

事件描述  $\text{appointment}$  可以是任意的字符序列, 但保证在一行之内。你可以假定每行总是不超过 255 个字符,  $10 \leq hh \leq 18$  以及  $0 \leq mm < 60$ 。数据输入的顺序是任意的, 你不能假定任何特定的顺序。输入以文件结束符终止。

## 输出

对于每组数据, 你应该输出如下一行内容:

`Day #d: the longest nap starts at hh : mm and will last for [H hours and] M minutes.`

其中  $d$  表示测试数据编号 (从 1 开始),  $hh : mm$  为打盹开始的时间。打盹持续时间的表示方法按照如下规则:

1. 若总时间  $X$  不超过 60 分钟, 则只输出 “ $X$  minutes”。
2. 若总时间  $X$  至少为 60 分钟, 输出 “ $H$  hours and  $M$  minutes”, 其中

$$H = \left\lfloor \frac{X}{60} \right\rfloor \quad M = X \bmod 60$$

不用担心单词的单复数变化。例如你应当打印 “1 minutes” 或者 “1 hours”, 尽管它们在词法上是错误的。

打盹的持续时间按结束时间减去开始时间计算。如果前一个事件在 14:00 结束而下一个事件在 14:47 开始, 那么能够打盹的时间就是  $14:47 - 14:00 = 47$  分钟。

如果出现并列最长的可能打盹时间, 输出最早的那个。你可以假定这个教授不会一整天都忙, 至少会有一段可以打盹的时间。

## 样例输入

4

10:00 12:00 Lectures

12:00 13:00 Lunch, like always.

13:00 15:00 Boring lectures...

15:30 17:45 Reading

4

10:00 12:00 Lectures

12:00 13:00 Lunch, just lunch.

13:00 15:00 Lectures, lectures... oh, no!

16:45 17:45 Reading (to be or not to be?)

4

10:00 12:00 Lectures, as everyday.

12:00 13:00 Lunch, again!!!

13:00 15:00 Lectures, more lectures!

15:30 17:15 Reading (I love reading, but should I schedule it?)

1

12:00 13:00 I love lunch! Have you ever noticed it? :)

### 样例输出

Day #1: the longest nap starts at 15:00 and will last for 30 minutes.

Day #2: the longest nap starts at 15:00 and will last for 1 hours and 45 minutes.

Day #3: the longest nap starts at 17:15 and will last for 45 minutes.

Day #4: the longest nap starts at 13:00 and will last for 5 hours and 0 minutes.

### 4.6.5 鞋匠的烦恼 (Shoemaker's Problem)

PC/UVa 题号: 110405/10026, 流行度: C, 通过率: average 难度: 2

一个鞋匠有  $n$  个必须完成的订单。鞋匠一天只能处理一个订单，但是一个订单可能需要很多天才能完成。对于第  $i$  个订单，整数  $T_i$  ( $1 \leq T_i \leq 1\,000$ ) 代表鞋匠完成这件工作所需要的天数。

但是，订单太多是需要付出代价的。

每个客户都认为自己的订单应该被马上处理。因此，对于第  $i$  件工作，在开始着手这件工作之前，每天都要付罚金  $S_i$  ( $1 \leq S_i \leq 10\,000$ ) 美分。写一个程序来帮这个鞋匠找出所付罚金最少的订单处理次序。

### 输入

输入第一行只有一个正整数，代表测试点的数量。接下来有一个空行。在相邻两组测试数据之间也会有一个空行。每组数据的第一行将给出订单的总数  $n$  ( $1 \leq n \leq 1\,000$ )。接下来的第  $i$  行有两个整数，即完成第  $i$  件工作所需时间  $T_i$  和延迟这项工作每天需要交的罚金  $S_i$ 。

### 输出

对于每组测试数据，你的程序要输出缴纳罚金最少的工作次序。每件工作用它在输入文件中的序号（从 1 开始）表示。所有的序号应在同一行输出，并且相邻两个序号之间应用一

个空格隔开。如果有多种可能的解，输出字典序最小的一种。

相邻两组数据的输出间应有一个空行。

### 样例输入

```
1
4
3 4
1 1000
2 2
5 5
```

### 样例输出

```
2 1 3 4
```

## 4.6.6 CDVII 高速公路 (CDVII)

PC/UVa 题号: 110406/10138, 流行度: C, 通过率: low 难度: 2

罗马的道路以可靠的施工著称。可惜的是，可靠的施工并不便宜，一些统治者决定用自动化的收费来回收成本。

CDVII 是一条特别的收费公路，它的票价结构如下：在这条公路上行驶每公里所付的费用由车辆开上这条公路的时间而定。每个进出口的照相机将记录下所有进出车辆的牌照号码。每个月的账单会被寄给这些车主，告诉他们每一公里所需要付的费用（以进入时刻的资费标准计），加上每次通过该公路需付的一美元以及月租费两美元（只要至少进出过一次 CDVII）。你的工作是要根据给出的汽车牌照照片准备这样的一份账单。

### 输入

输入第一行只有一个整数，代表测试数据的数量。接下来有一个空行。相邻两组数据之间也有一个空行。每组数据由两个部分构成：票价结构和牌照照片。票价结构在一行中给出，包含 24 个非负整数，分别代表从 00:00 到 00:59 的收费（美分/公里）、从 01:00 到 01:59 的收费等一天 24 小时各自的资费标准。每张照片包含车辆的牌照号码（不超过 20 个字母或数字）、时间和日期（mm:dd:hh:mm）、单词 enter 或 exit（代表进入或离开）以及出入口的位置（离公路某一端的距离，单位：公里）。所有的日期都在同一个月内。每一个“enter”的记录只有当同一辆车的下一条记录是“exit”时与之配对。不是成对的记录应当忽略。假设同一辆车不会在同一时间出现两项纪录。时间用 24 小时制表示。照片记录不会超过 1000 个。

### 输出

对于每一组测试数据，按字典序输出每一个进出过 CDVII 的车辆的牌照号码和费用。相邻两组数据的输出间应有一个空行。

### 样例输入

1

```
10 10 10 10 10 10 20 20 20 15 15 15 15 15 15 15 15 20 30 20 15 15 10 10 10  
ABCD123 01:01:06:01 enter 17  
765DEF 01:01:07:00 exit 95  
ABCD123 01:01:08:03 exit 95  
765DEF 01:01:05:59 enter 17
```

### 样例输出

```
765DEF $10.80  
ABCD123 $18.60
```

### 4.6.7 龟壳排序 (ShellSort)

PC/UVA 题号: 110407/10152, 流行度: B, 通过率: average 难度: 2

乌龟王想要调整他的宝座 (由乌龟们堆叠而成), 好让他最亲近最尊贵的宰相们离顶端更近一些。每次调整操作可以这样改变乌龟们在宝座中的顺序: 一只乌龟爬出宝座, 然后爬过其他乌龟, 到达宝座最顶端。

给出一个宝座中乌龟的初始顺序和目标顺序, 你的任务是找出一个最短的操作序列, 使宝座变为目标状态。

### 输入

输入第一行包含一个整数  $k$ , 表示测试数据的组数。每组测试数据的第一行包含一个整数  $n$ , 表示宝座中乌龟的数量。接下来的  $n$  行描述宝座中乌龟的初始顺序。每行为一只乌龟的名字, 从宝座的顶端开始一直到底端。每只乌龟都有唯一的名字, 为一个长度不超过 80 的字符串, 只包含英文字母、数字、空格符和句点 (“.”)。接下来  $n$  行描述目标状态宝座中乌龟的顺序, 仍然从顶端到底端列出乌龟的名字。每组数据严格包含  $2n + 1$  行。乌龟总数  $n$  不会超过 200。

### 输出

对于每组数据, 输出包含一个乌龟名字的序列, 每个名字单独占一行, 表示依次爬出宝座的乌龟名字。你应保证在将宝座从初始状态转化为目标状态的前提下让调整序列尽可能短。如果有多种可能的调整方式, 输出任何一个序列均可。

相邻两组测试数据的输出间用一个空行分隔。

样例输入

2  
3  
Yertle  
Duke of Earl  
Sir Lancelot  
Duke of Earl  
Yertle  
Sir Lancelot  
9  
Yertle  
Duke of Earl  
Sir Lancelot  
Elizabeth Windsor  
Michael Eisner  
Richard M. Nixon  
Mr. Rogers  
Ford Perfect  
Mack  
Yertle  
Richard M. Nixon  
Sir Lancelot  
Duke of Earl  
Elizabeth Windsor  
Michael Eisner  
Mr. Rogers  
Ford Perfect  
Mack

样例输出

Duke of Earl  
Sir Lancelot  
Richard M. Nixon  
Yertle

#### 4.6.8 足球 (Football (aka Soccer))

PC/UVa 题号: 110408/10194, 流行度: B, 通过率: average 难度: 1

足球是全世界最流行的运动, 尽管美国人坚持叫它“英式足球 (*soccer*)”。一个像巴西这样捧过五次世界杯的国家会因为拥有很多国家级和地方级联赛而难以管理。你的任务是写一个程序读入联赛名称、队名和比赛情况, 输出联赛的当前排名。

一支球队在一场比赛中进球数多于对手就能赢得这场比赛, 少于对手就会输掉这场比赛

最新書  
發售  
上架  
站  
·83·

赛，进球数相等为平局。胜一场得 3 分，平一场得 1 分，输一场得 0 分。

队伍按如下规则排名 (按照从前往后的顺序):

1. 得分从高到低。
  2. 获胜场数从多到少。
  3. 总净胜球数 (进球数 - 失球数) 从多到少。
  4. 总进球数从多到少。
  5. 比赛场数从少到多。
  6. 大小写不敏感的字典序。

输入

输入第一行仅包含一个整数  $N(0 < N < 1000)$ 。接下来为  $N$  个联赛的描述，以联赛的名称开头。联赛名称由至多 100 个字母、数字、空格等组成，占单独的一行。接下来一行一个数  $T(1 < T \leq 30)$ ，表示该联赛参赛队伍的数量。在接下来  $T$  行中，每行为一支队伍的名称。队名最多有 30 个字符，可以包含 ASCII 码大于或等于 32 (空格) 的字符中除了“#”和“@”的任意字符。

在队名描述之后是一个独占一行的非负整数  $G$ , 表示该联赛中已经结束的比赛场数。 $G$  不会超过 1000。接下来的  $G$  行按如下格式给出比赛结果:

team\_name\_1#goals1@goals2#team\_name\_2

例如，Team A#3@1#Team B 表示一场 Team A 和 Team B 之间的比赛，其中 Team A 进了 3 个球，Team B 进了一个球。所有进球数均为小于 20 的非负整数。你可以假定所有比赛结果中出现的队名都在前面的队伍列表中，而且一个队不会和自己比赛。

输出

对于每个联赛，在单独的一行中输出联赛名称。接下来的  $T$  行应输出在上述规则下的队伍排名。注意当需要按字典序排名时，你不应当区分大小写。每行输出格式如下：

[a]) Team.name [b]p, [c]g ([d]-[e]-[f]), [g]gd ([h]-[i])

其中 [a] 是队伍排名, [b] 是总分, [c] 为比赛场次, [d] 为胜场数, [e] 为平局数, [f] 为负场数, [g] 为净胜球, [h] 为进球数, [i] 为失球数。

以上各项之间用一个空格隔开，相邻联赛的输出之间用一个空行隔开。参见样例输出。

### 样例输入

## 样例输出

World Cup 1998 - Group A      1) Brazil 6p, 3g (2-0-1), 3gd (6-3)  
4                                    2) Norway 5p, 3g (1-2-0), 1gd (5-4)  
Brazil                            3) Morocco 4p, 3g (1-1-1), 0gd (5-5)  
Norway                          4) Scotland 1p, 3g (0-1-2), -4gd (2-6)  
Morocco  
Scotland                        Some strange tournament  
6                                1) Team D 4p, 2g (1-1-0), 1gd (2-1)  
Brazil#2@1#Scotland            2) Team E 3p, 2g (1-0-1), 0gd (3-3)  
Norway#2@2#Morocco            3) Team A 3p, 3g (0-3-0), 0gd (3-3)  
Scotland#1@1#Norway            4) Team B 1p, 1g (0-1-0), 0gd (1-1)  
Brazil#3@0#Morocco            5) Team C 1p, 2g (0-1-1), -1gd (3-4)  
Morocco#3@0#Scotland  
Brazil#1@2#Norway  
Some strange tournament  
5  
Team A  
Team B  
Team C  
Team D  
Team E  
5  
Team A#1@1#Team B  
Team A#2@2#Team C  
Team A#0@0#Team D  
Team E#2@1#Team C  
Team E#1@2#Team D

## 4.7 提 示

- 4.1 直观来看, Vito 的房子应当位于所有亲戚位置在某种意义上的“平均”。是均值吗? 还是中位数? 或者是其他什么?
- 4.3 按照过桥速度排序是否有助于决定谁和谁配对?
- 4.4 排序对解题有帮助吗?
- 4.5 为了帮助解题, 应当给各项工作按照时间长度还是罚款金额排序? 或者两个都要考虑?
- 4.6 能否把一条日期/时间信息转化成单个整数, 以便操作?

4.7 在什么情况下，我们无需移动一只乌龟？

4.8 对于这样一个复杂的排名系统，应如何简化比较函数的编写？

## 4.8 注解

4.2 用最少的反转次数给煎饼排序是一个著名的问题，因为 Bill Gates 所发表过的惟一的论文 [GP79] 便是关于它的！这个问题除了本身具有数学上的研究价值外，在重建物种（例如人类和老鼠）之间的进化史上也有着有趣的应用。基因组反转 (genome reversal) 事件将会反转一个生物体 DNA 序列中的一段基因。经过漫长时间的进化，这些罕见的事件将对生物体产生重大的影响，因此，重建反转顺序是一个重要的问题。关于基因组反转的更多讨论，参见 [Ber01, Gus97]。

## 第5章 算术与代数

编程技巧和数学能力之间历来都有着重要的联系。事实上，计算机最初便是数学家们为了能够更快捷地运算而制造出来的。Pascal(他原本作为一个数学家闻名于世，直到很多年后人们用他来命名一门编程语言)早在1645年就用齿轮制造出了一个加法机器。像Turing和Von Neumann这样的计算机科学界的先驱科学家们，在纯数学上的造诣不亚于他们在计算机领域中取得的成就。

我们将在本章探索一些算术和代数中的编程挑战。它们看上去是数学中最基础的部分，但其中的一些算法不可避免地用到了一些诸如数论这样的高级话题，这也证明了算术和代数并不像它们看上去的那么简单。

### 5.1 机器算术

每个编程语言都有整数类型，支持四种基本运算：加法、减法、乘法和除法。这些运算一般直接映射到硬件级的算术运算指令，因此整数的大小范围取决于相应的处理器。

现在的PC机一般是32位的，即标准整数类型支持的范围大致是 $\pm 2^{31} = \pm 2\,147\,483\,648$ 。因此，在普通机器上可以轻易数到十亿以上。

多数编程语言支持32位，甚至64位整数。 $2^{63} = 9\,223\,372\,036\,854\,775\,808$ ，比“万亿”还要大好多个数量级。即使用当今最先进的计算机来数数，数到这个数之前你恐怕早已不愿等待。因此，这样大的数多出现在数学研究和编程竞赛中。

常规的32位整数一般用4个连续字节来表示，而64位整数需要8个字节。当需要保存的数范围小，数量又很多时，它们就显得有些浪费了。比如，计算机图像经常用单字节颜色(即256级灰度)矩阵表示，以节省内存。

正整数的表示并无特别之处，但负整数的表示往往需要一点技巧，例如补码(two's-complement)表示法。尽管看起来不那么直观，却很方便用硬件实现算术运算。

浮点数将在5.5节中讨论。浮点数据类型所能表示的范围相当之大，特别是采用双精度浮点类型时。但请注意，表示范围大的原因是采用了科学计数法 $a \times 2^c$ 。由于a和c的位数均有限制，浮点数的精度(precision)仍然是有限的。如果要精确地计数，请不要用浮点数，而使用整数。

#### 5.1.1 整数库函数

每种编程语言都把基本算术运算作为基础操作，而更高级的数学函数由标准库提供。

请留意数学库中与整数相关的函数。例如，C/C++ 头文件 `stdlib.h` 中声明了计算绝对值和生成随机数的函数，而 `math.h` 则定义了取上整、下整、算术平方根和幂函数。

Java 中的整数类更加高级。事实上，`java.math.BigInteger` 类提供了高精度整数，比我们下一节将要开发的高精度整数要完善得多。你应当尽量使用它，但同时也不要忘记，在一些特定应用和题目中，你仍然需要自己编写高精度整数和相应的操作。

## 5.2 高精度整数

表示大整数需要把它的各位数字串起来。有两种可能的方案：

- **数字数组** — 最简单的方法是数字数组，其中数组右起的第一个元素表示最低位（即个位）。可以维护数位数，以避免不必要的运算。
- **数字链表** — 如果真的需要处理任意精度的整数（即，数位数没有上界），动态结构是必须的。但请注意，100 000 位的整数对于大多数应用来说已经足够，用 100 000 个字节足以存下它。这个内存开销对于现代计算机来说可以忽略不计。

我们将在本节实现“数字数组”表示法的主要算术运算。动态内存分配和链表尽管看上去可以根据需要无限制地使用内存，但由于指针本身也会占用空间，链式结构往往浪费了大量的内存。

动态内存真正的好处在于为你提供足够的灵活性，把内存用到真正需要的地方。如果你需要创建一个巨大的高精度整数数组，其中一些整数很大，但绝大多数都很小，就可以考虑用数字链表的方式，因为无法同时给每个整数分配很大的空间<sup>1</sup>。

高精度整数的数据结构如下：

```
#define MAXDIGITS      100          /* maximum length bignum */

#define PLUS            1           /* positive sign bit */
#define MINUS           -1          /* negative sign bit */

typedef struct {
    char digits[MAXDIGITS];        /* represent the number */
    int signbit;                  /* PLUS or MINUS */
    int lastdigit;                /* index of high-order digit */
} bignum;
```

注意每个数字（0~9）用一个单字节字符表示<sup>2</sup>。尽管做运算时需要更加小心，但为此换

<sup>1</sup> 译者注：也可以继续使用数字数组，但应根据需要分配数组空间。或者使用像 STL 中的 `vector` 这样的动态数组。

<sup>2</sup> 译者注：数字 1 用 ASCII 码为 1 的字符，而不是‘1’来表示。

来空间节省是值得的。把 1 和 -1 分别作为正数和负数的 signbit 值比较方便，因为在乘法里可以直接将两个 signbit 相乘，来得到乘积的 signbit。

没有人规定必须用 10 进制来表示高精度整数。事实上，适当的扩大基数可以减少数位，从而提高效率。尽管如此，10 进制表示在打印输出方面还是要方便一些的：

```
print_bignum(bignum *n)
{
    int i;

    if (n->signbit == MINUS) printf("- ");
    for (i=n->lastdigit; i>=0; i--)
        printf("%c", '0'+ n->digits[i]);

    printf("\n");
}
```

为了简单起见，我们的代码忽略了溢出的可能性。

### 5.3 高精度算术

我们在学校里学到的第一批算法就是整数的四则运算：加、减、乘、除。我们学会了如何执行它们，哪怕对其中的原理还不甚理解。

下面，我们来回顾一下这些在小学里学到的算法，重点在理解它们为什么是正确的以及如何把它们“教”给计算机执行。对于所有四则运算，假定操作数和结果满足等式  $c = a * b$ ，其中 \* 是 +、-、\* 或者 /。

- 加法 — 整数加法从右往左进行，每一位相加时产生的进位向左传递。负数的出现使得问题变得复杂起来，需要仔细判断，以便确定何时转化成减法运算。

```
add_bignum(bignum *a, bignum *b, bignum *c)
{
    int carry;                      /* carry digit */
    int i;                           /* counter */

    initialize_bignum(c);

    if (a->signbit == b->signbit) c->signbit = a->signbit;
    else {
```

```

        if (a->signbit == MINUS) {
            a->signbit = PLUS;
            subtract_bignum(b,a,c);
            a->signbit = MINUS;
        } else {
            b->signbit = PLUS;
            subtract_bignum(a,b,c);
            b->signbit = MINUS;
        }
        return;
    }

c->lastdigit = max(a->lastdigit,b->lastdigit)+1;
carry = 0;

for (i=0; i<=(c->lastdigit); i++) {
    c->digits[i] = (char)
        (carry+a->digits[i]+b->digits[i]) % 10;
    carry = (carry + a->digits[i] + b->digits[i]) / 10;
}

zero_justify(c);
}

```

注意上述代码中的一些细节。符号位 signbit 的处理并不是一件容易的事情。在一些特殊情况下，我们修改了操作数的符号位，并且交换了操作数的顺序，但在修改符号位时应更加小心。

加法过程本身是相当简单的。由于结果的所有数字都初始化为 0，最高位的进位无需特殊处理（它将和初始化后的 0 相加）。zero\_justify 函数调整 lastdigit 的值，来避免前导零（leading zeros）的出现。每个运算之后调用 zero\_justify 总是有益无害的，特别是它还能自动把 -0 更正为 0：

```

zero_justify(bignum *n)
{
    while ((n->lastdigit > 0) && (n->digits[ n->lastdigit ]==0))
        n->lastdigit--;
}

```

```
    if ((n->lastdigit == 0) && (n->digits[0] == 0))
        n->signbit = PLUS;      /* hack to avoid -0 */
    }

• 减法 — 减法比加法容易写错，因为它涉及借位。为了保证借位能够成功终止，最好确保被减数不小于减数。
```

```
subtract_bignum(bignum *a, bignum *b, bignum *c)
{
    int borrow;                  /* anything borrowed? */
    int v;                      /* placeholder digit */
    int i;                      /* counter */

    if ((a->signbit == MINUS) || (b->signbit == MINUS)) {
        b->signbit = -1 * b->signbit;
        add_bignum(a,b,c);
        b->signbit = -1 * b->signbit;
        return;
    }

    if (compare_bignum(a,b) == PLUS)  {
        subtract_bignum(b,a,c);
        c->signbit = MINUS;
        return;
    }

    c->lastdigit = max(a->lastdigit,b->lastdigit);
    borrow = 0;

    for (i=0; i<=(c->lastdigit); i++) {
        v = (a->digits[i] - borrow - b->digits[i]);
        if (a->digits[i] > 0)
            borrow = 0;
        if (v < 0) {
            v = v + 10;
            borrow = 1;
        }
        c->digits[i] = v;
    }
}
```

```

    }

    c->digits[i] = (char) v % 10;

}

zero_justify(c);
}

```

- 比较 — 确定两个数的大小关系需要“比较”运算。比较操作从符号位开始，按从左到右进行。

```

compare_bignum(bignum *a, bignum *b)
{
    int i;                                /* counter */

    if ((a->signbit==MINUS) && (b->signbit==PLUS)) return(PLUS);
    if ((a->signbit==PLUS) && (b->signbit==MINUS)) return(MINUS);

    if (b->lastdigit > a->lastdigit) return (PLUS * a->signbit);
    if (a->lastdigit > b->lastdigit) return (MINUS * a->signbit);

    for (i = a->lastdigit; i>=0; i--) {
        if (a->digits[i] > b->digits[i])
            return(MINUS * a->signbit);
        if (b->digits[i] > a->digits[i])
            return(PLUS * a->signbit);
    }

    return(0);
}

```

- 乘法 — 乘法看上去比加法和减法高级。像罗马人这样高深莫测的民族并不擅长乘法，尽管他们刻在建筑墙角石上的数字显得格外引人注目。

罗马人的问题在于他们并不使用进位制数字系统 (radix number system)。尽管乘法可以用反复相加的方式来完成，但这个方法实在是慢得让人无法忍受。用反复加法来计算 999 999 的平方需要上百万次运算，但用学校里学的“一行一行算”的方法却很容易得出结果：

```
multiply_bignum(bignum *a, bignum *b, bignum *c)
{
    bignum row;                      /* represent shifted row */
    bignum tmp;                      /* placeholder bignum */
    int i,j;                         /* counters */

    initialize_bignum(c);

    row = *a;

    for (i=0; i<=b->lastdigit; i++) {
        for (j=1; j<=b->digits[i]; j++) {
            add_bignum(c,&row,&tmp);
            *c = tmp;
        }
        digit_shift(&row,1);
    }

    c->signbit = a->signbit * b->signbit;

    zero_justify(c);
}
```

每次操作都需要把第一个乘数左移一个数位，然后把左移后的数乘以  $d$  的积加到最终结果中，其中  $d$  是第二个乘数对应的数字。尽管用反复加法来实现这一步看起来不够精巧，但由于对于第二个乘数的每个数字，内层循环最多执行 9 次，实际上并没有浪费太多的时间。左移一个数位等价于用它乘以进位制的基数，在十进制里就是乘以 10：

```
digit_shift(bignum *n, int d)          /* multiply n by  $10^d$  */
{
    int i;                            /* counter */

    if ((n->lastdigit == 0) && (n->digits[0] == 0)) return;

    for (i=n->lastdigit; i>=0; i--)
        n->digits[i+d] = n->digits[i];
```

```

    for (i=0; i<d; i++) n->digits[i] = 0;

    n->lastdigit = n->lastdigit + d;
}

```

- 除法 — 尽管学生和计算机架构师都对长除法“心存恐惧”，其实它可以用一个比人们预想的要简单些的循环来完成。如果被除数太大，反复作减法仍然慢得难以忍受。在下面的代码中，我们不断将余数左移，加上被除数的下一个数字，然后一次次尝试减去除数。这个方法比学校里学到的“试除法”更容易编程：

```

divide_bignum(bignum *a, bignum *b, bignum *c)
{
    bignum row;                      /* represent shifted row */
    bignum tmp;                      /* placeholder bignum */
    int asign, bsign;                /* temporary signs */
    int i,j;                         /* counters */

    initialize_bignum(c);

    c->signbit = a->signbit * b->signbit;

    asign = a->signbit;
    bsign = b->signbit;

    a->signbit = PLUS;
    b->signbit = PLUS;

    initialize_bignum(&row);
    initialize_bignum(&tmp);

    c->lastdigit = a->lastdigit;

    for (i=a->lastdigit; i>=0; i--) {
        digit_shift(&row,1);
        row.digits[0] = a->digits[i];
        c->digits[i] = 0;
    }
}

```

```
        while (compare_bignum(&row, b) != PLUS) {
            c->digits[i]++;
            subtract_bignum(&row, b, &tmp);
            row = tmp;
        }

        zero_justify(c);

        a->signbit = asign;
        b->signbit = bsign;
    }
}
```

上述代码完成整数除法，并丢弃余数。如果你希望计算  $a \div b$  的余数，可以用  $a - b \times (a \div b)$  代替。更为精巧的方法将在 7.3 节中讨论。当操作数中存在负数时，商和余数均没有清楚的定义<sup>3</sup>。因此如果上述代码给出的结果和编程语言不一致，请不要感到奇怪。

- 乘方 — 乘方的定义就是反复作乘法，因此在设计时将遇到和反复加法相同的性能问题。解决方法是注意到

$$a^n = a^{n/2} \times a^{n/2} \times a^{n \bmod 2}$$

因此只需对数级别的乘法次数。

## 5.4 进制及其转换

选取的基数 (*base*) 不同，用来表示一个特定整数的数字串也会不同。常用的基数包括：

- 二进制 — 二进制数用数字 0 和 1 组成。它是计算机内部整数的表示方式，因为 0 和 1 很自然地与开/关、高/低状态相对应。
- 八进制 — 八进制可以用来简化二进制数的读写，因为它从右往左的每一个数位对应了二进制表示中的三个连续的比特。例如， $11111001_2 = 371_8 = 249_{10}$ 。有趣的是，目前唯一的与进制转换相关的笑话用到了八进制。为何程序员总是把圣诞节和万圣节前夕弄混？答：因为  $31_{Oct} = 25_{Dec}$ <sup>4</sup>
- 十进制 — 我们平时都用十进制，因为我们用 10 根手指头数数。古玛雅人用二十进制数，大概是因为他们手足并用。

<sup>3</sup> 译者注：另外，除数为 0 时除法将出错。由于上述代码没有检测这种情况，读者在调用之前需要确保除数不为 0。

<sup>4</sup> 译者注：圣诞节是 12 月 25 日，万圣节前夕是 10 月 31 日。12 月的英文是 December，简称 Dec. 10 月的英文是 October，简称 Oct；十进制的英文是 Decimal，简称也是 Dec，八进制的英文是 Octal，简称也是 Oct。

- 十六进制 — 二进制的一个更加紧凑的缩写法。应注意 10 ~ 15 之间的数字用字母“A~F”表示。
- 其他进制 — 有时你会遇到基数更大的进位制。三十六进制是用数字和字母所能表示的最大进位制，但如果你有办法显示更多不同的符号，理论上你可以用任意大的整数作为进位制的基数。

把一个  $a$  进制整数  $x$  转化成  $b$  进制整数  $y$  通常有两种方法：

- 自左向右扫描 — 我们首先求出  $y$  的最高位。它是满足如下条件的整数  $d_l$ :

$$(d_l + 1)b^k > x \geq d_l b^k$$

其中， $1 \leq d_l \leq b - 1$ 。理论上，它可以用试错法找出，但需要比较不同基数的整数之间的大小，比较麻烦。这和前面介绍的长除法类似。

- 自右向左扫描 — 我们首先求出  $y$  的最低位，即  $x$  除以  $b$  的余数。这恰好是 7.3 节中将要讨论的问题。更美妙的是，由于在求余算法中  $x$  是一位一位处理的，对于大整数来讲也丝毫不成问题。

自右向左的扫描法类似于我们将普通整数转化成高精度整数的过程。将待转化的整数除以 10 取余数（用 % 运算符），就得到了最低位的数字。

```
int_to_bignum(int s, bignum *n)
{
    int i;                      /* counter */
    int t;                      /* int to work with */

    if (s >= 0) n->signbit = PLUS;
    else n->signbit = MINUS;

    for (i=0; i<MAXDIGITS; i++) n->digits[i] = (char) 0;

    n->lastdigit = -1;

    t = abs(s);

    while (t > 0) {
        n->lastdigit++;
        n->digits[n->lastdigit] = (t % 10);
        t = t / 10;
    }
}
```

```
if (s == 0) n->lastdigit = 0;  
}
```

将 10 换成其他数，我们就能把这个整数转化成其他进制数。

## 5.5 实 数

与实数有关的数学分支对于理解现实世界来说相当重要。牛顿在发现基本运动定律之前先创立了微积分。对积分和方程组求解的需求遍布科学的各个角落。早期的计算机主要就是用来处理实数计算问题的。

用计算机处理实数是一件富有挑战性的事，因为浮点算术只能保证有限的精度。你需要牢记：计算机处理的实数并不是真正的实数：

- 数学的很多分支依赖于实数的稠密性 (*continuity*)，即对于任意实数  $a < b$ ，存在实数  $c$  介于二者之间。但这个结论对于计算机表示的实数来说并不成立。
- 很多算法假设算术运算都是精确的，但对于计算机来说这也不一定成立。加法结合律确保了在数学上：

$$(a + b) + c = a + (b + c)$$

但由于舍入误差的存在，计算机中的实数并不一定满足上式。

我们将遇到不同类型的实数：

- 整数 — 整数是用来计数的实数， $-\infty, \dots, -2, -1, 0, 1, 2, \dots, \infty$ 。特殊的整数集合包括自然数 (*natural numbers*) (从 0 开始的整数)、正整数 (*positive integers*) (从 1 开始的整数)，尽管这些记号并不是通用的。

整数的一个限制是它们之间有间隙。曾经有一篇愚人节报道的标题是“科学家们发现了 6 和 7 之间的一个新数”。这很可笑，因为任何两个有理数  $x$  和  $y$  之间本来就有一个有理数 (例如  $(x + y)/2$ )。在 6 和 7 之间发现一个整数才算得上新闻。

- 有理数 — 有理数是那些可以写成两个整数之比的数。换句话说， $c$  是有理数当且仅当存在整数  $a$  和  $b$ ，使得  $c = a/b$ 。每个整数自身都是有理数，因为整数  $c$  可以写成  $c/1$ 。如果我们允许假分数 (*improper fractions*) 的存在，即满足  $a \geq b$  的分数  $a/b$ ，有理数和分数 (*fraction*) 就成了同义词。
- 无理数 — 很多有用的数都不是有理数，例如  $\pi = 3.141\ 592\ 6\dots$ 、 $\sqrt{2} = 1.414\ 21\dots$  以及  $e = 2.718\ 28\dots$ 。可以证明，不存在整数  $x$  和  $y$ ，使得  $x/y$  与上述任何一个数相等。

那么，应该如何在计算机内表示这些数呢？如果你确实需要保留任意精度的数值，可以用泰勒级数展开 (*Taylor series expansions*) 来计算它们，但一般来说，用十多位有效数字来近似它们已经足够了。

### 5.5.1 如何处理实数

浮点数的内部表示法和机器、语言与编译器都相关。这使得处理浮点数成了一个非常头疼的问题。

越来越多的厂商开始遵守 IEEE 规范，但在对运算结果的精度要求很高时，你几乎一定会遇到麻烦。浮点数用科学记数法表示，即  $a \times 2^c$ ，其中尾数 (*mantissa*)  $a$  和指数 (*exponent*)  $c$  均有比特数限制。对两个数量级相差巨大的实数进行运算往往会引起上溢 (overflow) 或者下溢 (underflow) 错误，因为尾数部分所拥有的比特数不足以保存答案的全部。

上述问题成为了很多由舍入误差引起的错误的根源。其中最容易出错的情况是测试两个实数是否相等。由于尾数的最后几位几乎总是不精确的，严格测试比较两个实数的尾数和指数往往没有实际意义。不要把一个实数和 0 (或其他实数) 进行严格的相等测试，而应该看二者之间差的绝对值是否小于某一个很小的正数  $\epsilon$ 。

很多题目要求在输出实数时保留到小数点后的某一位。为此，我们需要区分四舍五入 (*rounding*) 和截尾 (*truncating*)。截尾是函数 `floor` 完成的操作，它去掉了实数的小数部分。舍入用来得到一个更加接近的个位数。为了把实数  $X$  舍入保留到小数点后  $k$  个数字，采用以下公式：

$$\text{round}(X, k) = \text{floor}(10^k X + (1/2))/10^k$$

而在输出时，最好用你的编程语言自带的格式化输出函数。

### 5.5.2 分数

精确的有理数  $x/y$  最好用一对整数  $x$  和  $y$  来表示，其中  $x$  被称为分子 (*numerator*)， $y$  被称为分母 (*denominator*)。

有理数  $c = x_1/y_1$  和  $d = x_2/y_2$  的四则运算很容易编写：

- 加法 — 在相加之前应当先通分：

$$c + d = \frac{x_1 y_2 + x_2 y_1}{y_1 y_2}$$

- 减法 — 和加法类似，因为  $c - d = c + (-1) \times d$ ：

$$c - d = \frac{x_1 y_2 - x_2 y_1}{y_1 y_2}$$

- 乘法 — 分子分母对应相乘：

$$c \times d = \frac{x_1 x_2}{y_1 y_2}$$

- 除法 — 除以一个分数等价于乘以它的倒数 (*reciprocal*)：

$$c/d = \frac{x_1}{y_1} \times \frac{y_2}{x_2} = \frac{x_1 y_2}{x_2 y_1}$$

这个式子为何是正确的呢？因为根据上述公式， $d(c/d) = c$ ，这正是除法的定义。

直接实现上述运算很容易引起溢出，必须在每次运算之后把结果约分 (*reduce*) 成最简分数，例如把  $2/4$  替换成  $1/2$ 。约分的方法是将分子和分母同时除以它们的最大公约数 (*greatest common divisor*)，即能同时整除分子和分母的最大整数。

用试错法或者穷举法求最大公约数需要花费大量的时间。推荐使用高效、易于编程的欧几里德算法，我们将在 7.2.1 小节中详细介绍它。

### 5.5.3 十进制实数

实数的十进制表示法是有理数的特殊情形。它由两部分组成：整数部分和小数部分。因此， $\pi$  的前 5 个有效数字可以用分数表示为：

$$3.141\bar{5} = (3/1) + (141\bar{5}/10\,000) = 6\,283/2\,000$$

如果最右端非 0 数字在小数点后第  $i$  位，则分数表示法的分母就是  $10^{i+1}$ 。

把有理数写成十进制实数的形式并不困难，只需用分子除以分母即可。问题在于，很多分数不存在有限的十进制表示。例如， $1/3=0.333\,333\,3\dots$ ,  $1/7=0.142\,857\,142\,857\,142\,857\,14\dots$  尽管保留十几位有效数字对于一般的应用而言已经足够，但有时我们仍然需要精确地表示，即  $1/30=0.\overline{03}$ ,  $1/7=0.\overline{142\,857}$ 。

如何把分数转化成十进制循环小数？可以用模拟长除法的方式。为了得到分数  $1/7$  的十进制展开式，我们用 7 来除 1.000 000 0...。每次用 10 来乘以余数，加上被除数的下一位数字（当处理到小数点以后，这个数字将总是 0），然后用这个所得的和数来除以除数。注意当这个和数出现循环之后，此后的所有运算将和第一次出现它之后的情形完全一样。因此，商中的数字也将开始无限地循环下去。

下面讨论循环小数化分数的方法。对于纯循环小数（从小数点后第 1 位开始循环） $x$ ，假设循环节为  $R$ ，它包含  $l$  个数字，我们有  $10^l x - x = R$ ，因此  $x = R/(10^l - 1)$ 。例如，若  $x = 0.123\,123\,123\dots$ ，则  $R = 123$ ,  $l = 3$ ，因此  $x = 123/999 = 41/333$ 。混循环小数的情形类似，读者可以自行推导。

## 5.6 代 数

尽管高中代数的内容主要和包含加法和乘法的方程有关，代数的主要研究对象是群环域等代数结构。多项式是一个重要的数学工具，它的定义为  $P(x) = c_0 + c_1 x + c_2 x^2 + \dots$ ，其中  $x$  是变量， $c_i$  是第  $i$  项  $x^i$  的系数。多项式的次数是使得  $c_i$  非零的最大  $i$ 。

### 5.6.1 多项式运算

$n$  次一元多项式的最自然的表示法是一个包含从  $c_0 \sim c_n$  这  $n+1$  个系数的数组。这使得基本算术运算变得十分简单：

- 求值 — 对于给定的  $x$ , 可以用朴素的方法计算  $P(x)$ , 即独立地计算出每一项  $c_i x^i$ , 然后把它们加起来。问题在于, 这个方法用了  $O(n^2)$  次乘法操作, 但实际上仅  $O(n)$  次就足够了。加速的关键在于等式  $x^i = x^{i-1} \times x$ , 因此只要按照次数从小到大的顺序计算各个项, 保留  $x$  的当前幂, 每计算一项都只需要两次乘法:  $x^{i-1} \times x$ , 然后  $c_i \times x^i$ 。

还有一种方法, 即引入洪特规则 (*Horner's rule*)。它完成的工作和刚才相同, 但是更加巧妙:

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_0 = ((a_n x + a_{n-1}) x + \dots) x + a_0$$

- 加减法 — 多项式加减法比高精度整数的加减法还要简单, 因为连进位和借位都不需要考虑。只需要从 0 到最高次项依次把对应的系数相加或相减即可。
- 乘法 — 计算多项式  $P(x)$  和  $Q(x)$  乘积的方法是从两个多项式各任取一项乘起来得到结果中的一项, 然后把所有这样的项加起来。即:

$$P(x) \times Q(x) = \sum_{i=0}^{\text{degree}(P)} \sum_{j=0}^{\text{degree}(Q)} (c_i c_j) x^{i+j}$$

上述两两配对相乘的操作称为卷积 (*convolution*)。本书中涉及的其他卷积包括整数乘法 (数字两两配对) 和字符串匹配 (模板和文本可能的位置两两配对)。用快速傅立叶变换 (fast Fourier transform, FFT) 可以在  $O(n \log n)$  而非朴素的  $O(n^2)$  时间内计算出卷积, 尽管这已超出本书范围。但了解这个问题的本质和相关的工具仍然是有益的。

- 除法 — 相比之下, 多项式除法要困难很多, 因为多项式对除法并不封闭。注意,  $1/x$  可以看成多项式, 也可以看成非多项式, 因为它可以写成  $x^{-1}$ 。但  $2x/(x^2 + 1)$  无论如何也无法看成多项式, 它是一个有理式 (*rational function*)。

多项式有时会是稀疏的 (*sparse*), 即大多数系数满足  $c_i = 0$ 。如果足够稀疏, 多项式应该表示成由系数 — 次数值对组成的链表。多元多项式 (*multivariate polynomials*) 包含超过一个变元。二元多项式  $f(x, y)$  可以用系数矩阵  $C$  表示, 其中  $C[i][j]$  为项  $x^i y^j$  的系数。

### 5.6.2 多项式求根

给定多项式  $P(x)$  和目标数  $t$ , 多项式求根 (*root finding*) 的任务是找到满足  $P(x) = t$  的一个或所有  $x$ 。

如果  $P(x)$  是一次多项式, 根是  $x = (t - a_0)/a_1$ , 其中  $a_i$  是  $P(x)$  中  $x$  项的系数。如果  $P(x)$  是二次多项式, 则只需解一个二次方程 (*quadratic equation*):

$$x = \frac{-a_1 \pm \sqrt{a_1^2 - 4a_2(t - a_0)}}{2a_2}$$

三四次方程也都存在求根公式 (尽管很复杂), 但“公式法”已经走到尽头。可以证明, 五次或更高次的方程不存在封闭形式 (*closed form*) 的解。

正因为如此，高于二次的方程通常只能用数值方法求解。任何一本数值分析教材都会介绍很多求根算法，包括 Newton 迭代法 (Newton's method)、Newton-Raphson 法以及诸如数值稳定性这样的潜在陷阱。在这些算法中，最基本的思想是二分查找。假设函数  $f(x)$  在  $[l, u]$  内单调递增 (*monotonically increasing*)，即对于所有  $l \leq i \leq j \leq u$ ，都有  $f(i) \leq f(j)$ 。假设一定存在  $x \in [l, u]$  满足  $f(x) = t$ ，如何求出它呢？比较  $f((l+u)/2)$  和  $t$ 。如果  $t < f((l+u)/2)$ ，则  $x$  一定位于  $[l, (l+u)/2]$  内，否则它一定位于  $[(l+u)/2, u]$  内。重复以上过程，直到  $x$  所藏身的区间足够小，不需继续细分为止。

上述算法可以用来计算不小于 1 的任意实数  $t$  的平方根，因为这等价于找出方程  $x^2 = t$  在  $[1, t]$  内的根。尽管如此，利用下一节将介绍的指数对数，我们可以更方便地计算出  $t$  的算术  $i$  次根  $t^{(1/i)}$ 。

## 5.7 对 数

你可能曾注意到计算器上的 `log` 和 `exp` 按钮，不过很可能从未用过它们。你甚至不清楚为什么会有这两个按钮。简单的说，对数 (*logarithm*) 函数是指数函数的反函数。 $b^x = y$  几乎等价于  $x = \log_b y$ 。<sup>5</sup>

在上式中， $b$  被称为对数的底 (*base*)。由于历史原因，有两个特殊的底尤其重要：自然对数 (*natural log*) (用  $\ln x$  表示) 的底是  $e = 2.718\ 28\dots$ ， $\ln x$  的反函数为  $\exp x = e^x$ ，合起来得到：

$$\exp(\ln x) = x$$

常用对数 (*common logarithm*) (用  $\lg x$  表示) 的底是 10。在计算器发明之前，常用对数非常重要<sup>6</sup>，但现在已经大不如前。对数为我们提供了一个手算大整数乘积的方法，比如隐式的用计算尺 (slide rule)，或者显式的用对数表 (book of logarithms)。

对数可以用来简化乘法和乘方运算。回忆  $\log_a xy = \log_a x + \log_a y$ ，即乘积的对数等于对数的和。上式的直接推论是

$$\log_a n^b = b \cdot \log_a n$$

那么，如何利用  $\exp(x)$  和  $\ln(x)$  对任意正数  $a$  和  $b$  计算  $a^b$  呢？我们知道

$$a^b = \exp(\ln(a^b)) = \exp(b \ln a)$$

因此求幂只需一次乘法，并将上述两个函数各调用一次即可。

我们可以用这个方法计算平方根，因为  $\sqrt{x} = x^{1/2}$ 。类似的，我们也可以计算其他的分数幂。正因为如此，任何一个合理的编程语言的数学库中一定包含 `ln` 和 `exp` 函数。注意它

<sup>5</sup> 译者注：尽管二者所适用的变量范围不尽相同。

<sup>6</sup> 本书的原作者曾亲身经历过 1972 之前的那个时代。

们都属于复杂的数值函数(用泰勒展开计算),不可避免地包含一定的数值误差,因此不要期望 $\exp(0.5 \ln 4)$ 的结果严格等于2。

关于对数的另一个重要性质是如下的换底公式:

$$\log_a b = \frac{\log_c b}{\log_c a}$$

因此把对数 $\log b$ 的底从 $a$ 换成 $c$ 只需要除以 $\log_c a$ 。借助这个方法,我们很容易用自然对数计算常用对数,反之亦然。

## 5.8 实数函数库

### C/C++ 中的数学库

标准的 C/C++ 数学库中有一些函数在处理实数时很有用:

```
#include <math.h>           /* include the math library */

double floor(double x);    /* chop off fractional part of x */
double ceil (double x);   /* raise x to next largest integer */
double fabs(double x);    /* compute the absolute value of x */

double sqrt(double x);    /* compute square roots */
double exp(double x);     /* compute e^x */
double log(double x);     /* compute the base-e logarithm */
double log10(double x);   /* compute the base-10 logarithm */
double pow(double x, double y); /* compute x^y */
```

### Java 中的数学库

`java.lang.Math` 类除了包含所有上述函数外,还有一些其他功能。例如,`round` 函数有四舍五入的功能。

## 5.9 习 题

### 5.9.1 小学生算术 (Primary Arithmetic)

PC/UVA 题号: 110501/10035, 流行度: A, 通过率: average 难度: 1

孩子们在学习多位数加法时被告知要从右至左将每一位数依次相加。很多人认为进位(把1从某个数位进到下一位)是一个重大的挑战。你的工作是要统计出每个加法题目所需

的进位次数，以便让教育者们评估这些题目的难度。

### 输入

输入的每一行包含两个 10 位以下的非负整数。输入的最后一行为“0 0”。

### 输出

对于除了“0 0”之外的每行输入，统计把输入的两个数加起来所需的进位次数，按照样例输出的格式打印出来。

#### 样例输入

```
123 456
555 555
123 594
0 0
```

#### 样例输出

```
No carry operation.
3 carry operations.
1 carry operation.
```

### 5.9.2 反转相加 (Reverse and Add)

PC/UVA 题号: 110502/10018, 流行度: A, 通过率: low 难度: 1

反转相加 (*reverse and add*) 操作从某个数开始，先将它每一位数的顺序颠倒过来，然后把这个数和原数相加。如果和不是一个回文数（即从左向右读和从右向左读不是同一个数），就重复这一操作，直到它变成一个回文数。

例如，从 195 开始，我们将在 4 次操作后得到最终结果 9 339。

$$\begin{array}{r} 195 & 786 & 1\ 473 & 5\ 214 \\ 591 & 687 & 3\ 741 & 4\ 125 \\ + \_ & + \_ & + \_ & + \_ \\ 786 & 1\ 473 & 5\ 214 & 9\ 339 \end{array}$$

这种方法几乎能让所有的整数在几次操作之内变成回文数。但也有一些数例外。196 是第一个至今尚未得到回文数的数。同时，人们也无法证明它永远都得不到回文数。

写一个程序输入一个数，然后输出最终得到的回文数（如果存在的话）和在此之前加法的次数。

假设输入数据中的每一个数都可以在 1 000 次加法内得到一个不超过 4 294 967 295 的回文数。

### 输入

输入第一行包含一个整数  $N(0 < N \leq 100)$ ，表示测试数据的组数。接下来的  $N$  行中，每行包含一个整数  $P$ 。

**输出**

对于每个输入整数，输出得到回文数所需的最少加法次数以及得到的回文数。二者应以一个空格隔开。

**样例输入**

3  
195  
265  
750

**样例输出**

4 9339  
5 45254  
3 6666

**样例输出****5.9.3 考古学家的烦恼 (The Archeologist's Dilemma)**

PC/UVa 题号: 110503/701, 流行度: A, 通过率: low 难度: 1

一个考古学家在寻找外星生物曾到过地球的证据。他偶然发现一面破损的墙上有一串串奇怪的数字。左侧的数字都是完整的，但不幸的是，很多数的右边部分都因石头被腐蚀而丢失了。她发现保存完好的数都是 2 的幂，所以猜测所有的数都是 2 的幂。为了坚定信心，她选取了一份数的清单，每个数中清晰可辨的数字个数总是严格地小于已丢失的数字个数。请你为清单中的每个数找出一个尽量小的 2 的幂，使得它左侧的数字和清单吻合。

写一个程序，对于一个给定的整数，找出最小的指数  $E$  (如果存在)，使得  $2^E$  从最高位开始的若干个数字等于给定整数 (注意：一半以上的数字已经丢失)。

**输入**

输入每行包含一个不超过 2 147 483 648 的正整数  $N$ 。

**输出**

对于每个输入的整数，输出一行，包含最小正整数  $E$ ，使得  $2^E$  左端的若干位数字正好和  $N$  相同。如果不存在，输出 “no power of 2”。

**样例输入**

1  
2  
10

**样例输出**

7  
8  
20



## 5.9.6 多项式的系数 (Polynomial Coefficients)

PC/UVa 题号: 110506/10105, 流行度: B, 通过率: high 难度: 1

计算如下多项式展开后某个特定项的系数:

$$P = (x_1 + x_2 + \dots + x_k)^n$$

## 输入

输入包含若干组数据, 每组有两行。第一行为两个整数  $n$  和  $k(0 < k, n < 13)$ , 分别表示多项式的次数和变元数。第二行为  $k$  个非负整数  $n_1, \dots, n_k$ , 满足  $n_1 + \dots + n_k = n$ 。

## 输出

对于每组输入数据, 输出一行, 仅包含一个整数, 即多项式  $(x_1 + x_2 + \dots + x_k)^n$  展开后  $x_1^{n_1} x_2^{n_2} \dots x_k^{n_k}$  这一项的系数。

## 样例输入

```
2 2
1 1
2 12
1 0 0 0 0 0 0 0 0 1 0
```

## 样例输出

```
2
2
```

## 5.9.7 Stern-Brocot 代数系统 (The Stern-Brocot Number System)

PC/UVa 题号: 110507/10077, 流行度: C, 通过率: high 难度: 1

*Stern-Brocot* 树是一种生成所有非负最简分数  $\frac{m}{n}$  的美妙方式。其基本思想是从  $(\frac{0}{1}, \frac{1}{0})$  这两个分数开始, 根据需要反复执行如下操作: 在相邻分数  $\frac{m}{n}$  和  $\frac{m'}{n'}$  之间插入  $\frac{m+m'}{n+n'}$ 。

例如, 第一步将在  $\frac{0}{1}$  和  $\frac{1}{0}$  之间得到一个新的分数:

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

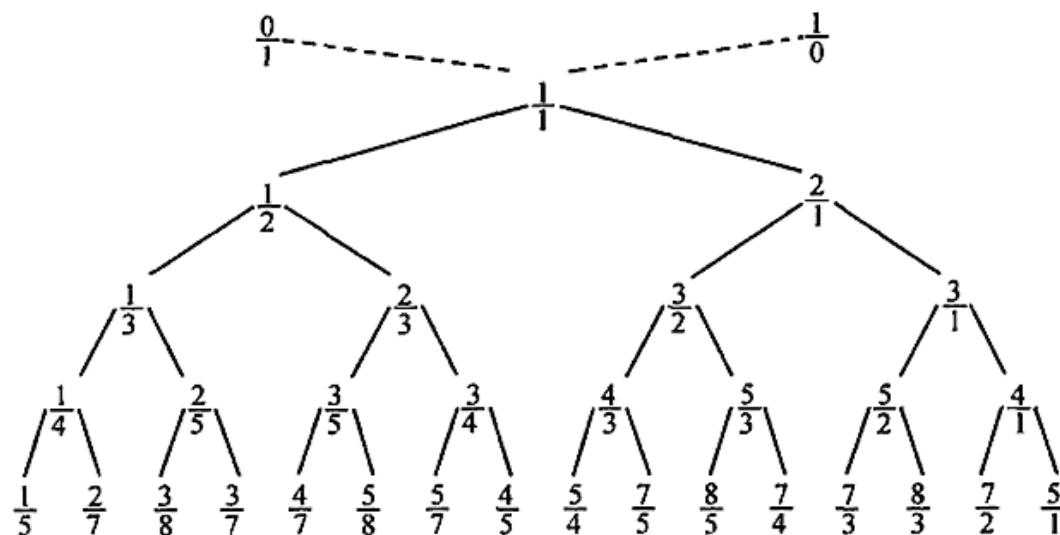
然后下一步将得到两个新分数:

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

接下来是 4 个新分数:

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

你可以将整个序列看作一个无限延伸的二叉树, 它的顶部看来是这样的:



构造过程是保序的，因此同一个分数不会在多个地方出现。

事实上，我们可以把 Stern-Brocot 树看成一个表示有理数的代数系统，因为每个正的最简分数在这棵树中恰好出现一次。我们用字母“L”和“R”分别表示从树根开始的一步“往左走”和“往右走”，则一个 L 和 R 组成的序列唯一地确定了树中的一个位置。例如，LRRL 表示从  $\frac{1}{1}$  往左走到  $\frac{1}{2}$ ，然后往右走到  $\frac{2}{3}$ ，再往右走到  $\frac{3}{4}$ ，最后往左走到  $\frac{5}{7}$ 。我们可以把 LRRL 看作  $\frac{5}{7}$  的一种表示法。几乎每个正分数均有唯一的方法表示成一个由 L 和 R 组成的序列。

唯一的例外是  $\frac{1}{1}$ ，它对应于空串。我们用 I 来表示它，因为它看起来像 1，而且是单位 (identity) 的首字母。

给定一个分数，输出它在 Stern-Brocot 代数系统中的表示法。

### 输入

输入包含多组数据。每组数据仅一行，包含两个互素的正整数  $m$  和  $n$ 。 $m = n = 1$  表示输入结束，你不必对它进行处理。

### 输出

对于输入的每组数据，输出一行，表示该分数在 Stern-Brocot 代数系统中的表示法。

#### 样例输入

```
5 7
878 323
1 1
```

#### 样例输出

```
LRRL
RRLRRLRLLLLRLRLLLRRR
```

### 5.9.8 两两之和 (Pairsumonious Numbers)

PC/UVa 题号: 110508/10202, 流行度: B, 通过率: high 难度: 4

$n$  个整数两两相加可以得到  $n(n - 1)/2$  个和。你的任务是根据这些和来找出原来的  $n$  个整数。

### 输入

输入每行以一个整数  $n(2 < n < 10)$  开头，接下来是  $n(n - 1)/2$  个整数，代表两两相加的和。相邻整数以空格隔开。

### 输出

对于输入的每一行，输出一行，包含  $n$  个整数，按非降序排列。如果有多组解，任意输出一组即可。如果无解，输出“Impossible”。

### 样例输入

```
3 1269 1160 1663
3 1 1 1
5 226 223 225 224 227 229 228 226 225 227
5 216 210 204 212 220 214 222 208 216 210
5 -1 0 -1 -2 1 0 -1 1 0 -1
5 79950 79936 79942 79962 79954 79972 79960 79968 79924 79932
```

### 样例输出

```
383 777 886
Impossible
111 112 113 114 115
101 103 107 109 113
-1 -1 0 0 1
39953 39971 39979 39983 39989
```

## 5.10 提 示

- 5.1 我们需要实现一个完整的高精度加法，还是有更简单的方法来获取进位次数？
- 5.3 我们需要实现一个完整的高精度乘法，还是可以利用 2 的幂的某些性质来简化问题？
- 5.4 是否需要计算出完整的答案，以便统计位数？
- 5.5 解决这样的一般性问题是否更为简单：从  $x$  开始做乘法，谁将获胜？
- 5.6 必须计算出完整的展开式，还是有更简单的方法可以只计算我们感兴趣的系数？二项式定理有用吗？
- 5.8 是否有必要穷举所有可能性？如果是，先看看第 8 章中介绍的回溯法。

### 5.11 注 解

5.2 历时 3 年的计算机搜索证实了从 196 开始加到 2 百万位之内无法得到回文数。可以预见，搜索的时间越长，回文数出现的可能性越少。

详情参见 <http://www.fourmilab.ch/documents/threeyears/threeyears.html>。

## 第6章 组合数学

组合数学是关于计数的数学分支，有不少基本计数问题贯穿整个计算机科学与编程领域。

组合数学中的问题以需要智慧和洞察力而著称。当你用正确的角度看待一个问题时，答案就会变得显而易见。这个现象使得这些问题特别适合程序设计竞赛，因为一旦发现了问题的本质，“生成并统计解”的复杂程序将变成一个简单的公式调用。在一些特殊情况下，你甚至可以脱离电脑，只用纸、笔或者袖珍计算器来直接算出一些相对较小的答案。请记住，评测系统不会读心术，也不会看你的程序来了解你的意图——它只检查输出。

### 6.1 基本计数技巧

在这里，我们复习一些基本计数原理和公式。你也许见过它们，不过可能已经忘却了。具体来说，一共有三个基本计数原理，从它们衍生出了大量的计数公式。对一个具体问题来讲，应当弄清楚哪些原理适用于该题。

- 乘法原理 (*product rule*) — 如果集合  $A$  里有  $|A|$  个元素，集合  $B$  里有  $|B|$  个元素，那么一共有  $|A| \times |B|$  种方法在  $A$  和  $B$  中各取一个元素组合起来。例如，如果你有 5 件衣服，4 件裤子，那么就有  $5 \times 4 = 20$  种着装方法。
- 加法原理 (*sum rule*) — 如果集合  $A$  里有  $|A|$  个元素，集合  $B$  里有  $|B|$  个元素。假设两个集合没有公共元素，那么一共有  $|A| + |B|$  个元素出现在  $A$  或  $B$  里。例如，如果你有 5 件衣服，4 件裤子，洗衣店不小心弄坏了一件，那么一共有 9 种被弄坏的可能。<sup>1</sup>
- 容斥原理 — 加法原理可以推广到两个集合有重叠部分的情形。即

$$|A \cup B| = |A| + |B| - |A \cap B|$$

例如，若  $A$  表示我的衣服颜色集合， $B$  表示裤子颜色集合。只要知道了颜色相同的衣服和裤子的数目，我就可以用容斥原理算出衣服和裤子的颜色总数。这个公式的原理是：把两个集合的颜色个数相加之后，有一些颜色被加了两遍（即衣服和裤子中都有的那些颜色），把它们减掉即可。容斥原理很容易推广到三个或更多集合的情形：

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

<sup>1</sup> 不过在日常生活中，洗衣店似乎总是爱挑你最心爱的那件，然后把它弄坏！

在组合计数时，弄清楚哪些元素“算重了”是一项需要细心和技巧的工作，因此用容斥原理解题并不是一件容易的事情。另一个强大的计数工具是一一映射。一一映射 (*bijection*) 是两个集合之间的一一对应关系。只要你在两个集合之间建立了这样的映射关系，对其中一个集合计数的同时，也就得到了另一个集合的大小。

例如，假设全班每个同学今天都穿着裤子，那么只要数出他们一共穿着多少条裤子，就知道班里有多少个学生。这是因为裤子和学生之间存在着一一映射。如果把裤子改成袜子，或者允许学生穿裙子，这个方法就不奏效了。

为了建立一一映射，我们需要了解一些容易计数的常见集合，以便将其他对象映射过来。以下就是这样一些组合对象。你还应该清楚对象数目的增长情况，以便知道穷举法何时失效：

- 排列 — 排列 (*permutation*) — 是将一些元素排成一列，每个元素恰好用一次。若有  $n$  个不同元素，则有  $n! = \prod_{i=1}^n i$  种不同的排列。三个元素的  $3! = 6$  种排列分别为：123、132、213、231、312 和 321。当  $n = 10$  时， $n! = 3\ 628\ 800$ ，已经接近穷举法的极限了。
- 子集 — 子集 (*subset*) — 是在  $n$  个元素中任选一些（可以全选也可以不选）。若有  $n$  个不同元素，则有  $2^n$  种不同的子集。三个元素的  $2^3 = 8$  个子集分别为：1、2、3、12、13、23、123 和空集 —— 千万不要漏掉空集。当  $n = 20$  时， $2^n = 1\ 048\ 576$ ，已经接近穷举法的极限了。
- 字符串 — 字符串 (*string*) — 是若干可重复的元素所组成的序列。若一共有  $m$  种元素，则有  $m^n$  个长度为  $n$  的序列。从集合 123 选出 3 个元素所能组成的 27 个字符串分别为 111、112、113、121、122、123、131、132、133、211、212、213、221、222、223、231、232、233、311、312、313、321、322、323、331、332 和 333。长度为  $n$  的二元串（只包含两种不同字符）个数等于  $n$  个不同元素的子集数目（想一想，为什么？），当  $m$  更大时增长得更快。

## 6.2 递推关系

递推关系使得很多递归结构的计数变得容易。递归结构包括树、列表、合式表达式和分治算法等 —— 它们无处不在。

什么是递推关系？它是一个用自己定义自己的方程式。为什么说它有用？因为很多普通函数都能轻而易举地用递推关系来表示！所有多项式都可以用递推关系表示，包括线性函数：

$$a_n = a_{n-1} + 1, a_1 = 1 \longrightarrow a_n = n$$

所有指数函数也都能用递推关系表示：

$$a_n = 2a_{n-1}, a_1 = 2 \longrightarrow a_n = 2^n$$

最后,一些用常规方法不太容易描述的、奇特而有趣的函数也能用递推来表示:

$$a_n = n a_{n-1}, a_1 = 1 \longrightarrow a_n = n!$$

因此,递推关系是表示函数的一种通用方法。很多计数问题的解都能方便地表达成递推关系的形式。求解递推关系,得到封闭形式 (closed form) 往往需要技巧,但正如我们即将看到的,用程序很容易算出递推关系中的各项数值,即使它并不存在一个简洁的封闭形式。

### 6.3 二项式系数

在组合计数中,二项式系数 (*binomial coefficients*) 起到了至关重要的作用。 $\binom{n}{k}$  表示  $n$  个元素取  $k$  个的方案总数。它所能解决的问题包括:

- 委员会 — 从  $n$  个人里选出  $k$  个组成一个委员会,有多少种方案? 根据定义,答案就是  $\binom{n}{k}$ 。
- 穿越网格的路径数 — 从一个  $n \times m$  网格的左上角走到右下角,只能沿着网格线往下或往右走,有多少种方法? 每条路径恰好包含  $n+m$  步,  $n$  次往下,  $m$  次往右。确定了其中的哪  $n$  步往下,也就确定了这条路径。因此一共有  $\binom{n+m}{n}$  条路径。
- $(a+b)^n$  的系数 — 观察下式:

$$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

对于一般问题, $(a+b)^n$  的展开式中,  $a^k b^{n-k}$  项的系数是多少? 答案是  $\binom{n}{k}$ , 因为它代表了从  $n$  个因子中选  $k$  个  $a$  的方案数。

- *Pascal* 三角形 — 你肯定在高中里见过如下图形。其中每个数都是它上面相邻两个数之和:

		1				
		1	1			
		1	2	1		
		1	3	3	1	
		1	4	6	4	1
		1	5	10	10	5
						1

*Pascal* 为什么要发明它? 因为这张表构造出了二项式系数! 表中第  $(n+1)$  行恰好是  $\binom{n}{i}$  当  $i$  取遍  $0 \leq i \leq n$  时的所有值。这个三角形里蕴含着很多有趣的等式,例如第  $(n+1)$  行的所有数之和等于  $2^n$ 。

如何计算二项式系数？由于  $\binom{n}{k} = n! / ((n - k)!k!)$ ，理论上可以直接用阶乘计算。但这样做有一个严重的问题：中间的运算结果很容易溢出，即使最后答案在整数范围之内。

一个更加保险的做法是利用 Pascal 三角形中蕴含的递推关系，即：

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

上式为何成立？考虑第  $n$  个元素是否被选中。如果选，则只需在剩下的  $n - 1$  个元素中选  $k - 1$  个。如果不选，则还需在剩下的  $n - 1$  个元素中选  $k$  个。两种情况没有重复、没有遗漏，因此二者的和就是方案总数。

未指定边界情况的递推是不完整的。有哪些二项式系数是不需要递推就能直接算出的？在递归展开过程中，左式最终会变成  $\binom{n-k}{0}$ 。在一个集合中选 0 个东西有几种方法？恰好有一种，即什么都不选。如果这还不够有说服力，把  $\binom{m}{1} = m$  作为边界条件也可以。类似的，右式最终会变成  $\binom{k}{k}$ 。在一个有  $k$  个元素的集合中选  $k$  个东西有几种方法？恰好有一种，即全部都选。加上边界情况后，上述递推关系才完整地定义了所有二项式系数。

计算这种递推式的最佳方法是建立一张包含所有可能值的表格（至少包含所有需要计算的值）。你可以从下面的程序中看到我们是如何做到这一点的。

```
#define MAXN 100 /* largest n or m */

long binomial_coefficient(n,m)
int n,m; /* computer n choose m */
{
    int i,j; /* counters */
    long bc[MAXN][MAXN]; /* table of binomial coefficients */

    for (i=0; i<=n; i++) bc[i][0] = 1;

    for (j=0; j<=n; j++) bc[j][j] = 1;

    for (i=1; i<=n; i++)
        for (j=1; j<i; j++)
            bc[i][j] = bc[i-1][j-1] + bc[i-1][j];
```

```

    return( bc[n][m] );
}

```

像这样计算递推式的程序是动态规划的基础。我们将在第 11 章中进一步学习。

## 6.4 其他计数序列

还有一些计数序列也频繁地出现在各类应用中，它们都可以用递推关系轻松算出。精于组合计数的人们都已将这些序列熟记于心，并用来帮助他们解决计数问题：

- *Fibonacci* 数 — Fibonacci 数列由递推关系  $F_n = F_{n-1} + F_{n-2}$  和初始条件  $F_0 = 0$  和  $F_1 = 1$  定义。它的应用广泛，因为它大概能算是最简单、有趣的递推关系了。该数列的开头几项分别是：0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …。在很多数学等式中都能发现 Fibonacci 数的身影，而它们自身也拥有很多有趣的性质。 $F_n$  拥有如下难以猜测但容易推导的封闭形式：

$$F_n = \frac{1}{\sqrt{5}} \left( \left( \frac{1+\sqrt{5}}{2} \right)^n - \left( \frac{1-\sqrt{5}}{2} \right)^n \right)$$

上述封闭形式有一些重要的推论。由于  $(1 - \sqrt{5})/2$  的值在 -1 和 1 之间，它的任意次幂也在这个范围之内。因此，第一项  $\phi^n$  (其中  $\phi = (1 + \sqrt{5})/2$ ) 占主导地位，因此可以用来估计  $F_n$ ，误差在正负 1 之间。

- *Catalan* 数 — 如下的递推关系和封闭形式定义了 *Catalan* 数：

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{1}{n+1} \binom{2n}{n}$$

涉及 *Catalan* 数的组合问题数量惊人。它的前几项是：2, 5, 14, 42, 132, 429, 1430, …，而  $C_0 = 1$ 。

用  $n$  对括号可以构造出多少平衡的括号表达式？例如， $n = 3$  时有五种方法：((()) $、$ (( $)$ ) $、$ ( $)$ (( $)$  $、$ ( $)$ ( $)$  $、$ 和 ( $)$ ( $)$ ( $)$  $)$ 。最左边的括号  $l$  一定和某个右括号  $r$  配对，它们合在一起把表达式划分成了两个平衡的部分：在  $l$  和  $r$  之间的部分以及  $r$  右边的部分。如果左边部分有  $k$  对括号，则右边部分有  $n - k - 1$  对括号，因为  $l$  和  $r$  已经用了一对括号。两个部分也必须是平衡的括号表达式，因此有如下递推关系：

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

这正是 *Catalan* 数。

同样的推理也适用于凸多边形的三角剖分计数、 $n+1$  个叶子的有根二叉树计数，以及不经过主对角线上方的网格穿越路线计数。*Catalan* 数有一个漂亮的封闭形式： $C_n =$

$$\frac{1}{n+1} \binom{2n}{n}.$$

- 欧拉数 — 欧拉数 (*Eulerian Number*)<sup>2</sup>  $\begin{Bmatrix} n \\ k \end{Bmatrix}$  表示元素  $1, 2, \dots, n$  的排列中, 恰好包含  $k$  次下降 ( $p_j > p_{j+1}$ ) (即包含  $k+1$  上升段)<sup>3</sup> 的排列个数<sup>4</sup>。为了得到它的递推关系, 考虑  $1, \dots, n-1$  的每个排列  $p$ , 总有  $n$  个位置可以插入元素  $n$ 。注意到  $n$  比已有元素  $1, \dots, n-1$  都大, 所以若  $p$  已经包含了  $k$  次下降, 则  $n$  必须插入到  $k+1$  个上升段中的某一个的最后面 (有  $k+1$  种方法) 才能保持下降数不增加; 另一方面, 若  $p$  只包含  $k-1$  次下降, 则  $n$  必须插入到  $k$  个上升段中的某一个的中间或者整个排列的开头 (有  $n-k$  种方法) 才能让下降数加 1。因此,  $\begin{Bmatrix} n \\ k \end{Bmatrix} = (k+1) \begin{Bmatrix} n-1 \\ k \end{Bmatrix} + (n-k) \begin{Bmatrix} n-1 \\ k-1 \end{Bmatrix}$ 。你能构造出四个元素、两个段的所有 11 种排列吗?
  - 第一类 *Stirling* 数 — 有两种不同的 *Stirling* 数。第一类 *Stirling* 数, 即  $\begin{bmatrix} n \\ k \end{bmatrix}$ , 表示  $n$  个元素组成的, 恰好包含  $k$  个循环 (permutation cycle)<sup>5</sup> 的排列<sup>6</sup>。为了得到递推关系, 考虑第  $n$  个元素是否形成一个单元素循环。如果是, 则有  $\begin{bmatrix} n-1 \\ k-1 \end{bmatrix}$  种方法来安排剩下的元素, 以形成  $k-1$  个循环。如果不是, 第  $n$  个元素可以在  $\begin{bmatrix} n-1 \\ k \end{bmatrix}$  种已有排列的基础上插入到任意一个循环的任何位置, 以用  $n-1$  个元素构造  $k$  个循环。因此,

$$\binom{n}{k} = \binom{n-1}{k-1} + (n-1) \binom{n-1}{k}$$

四个元素、两个循环的排列共有 11 种。

- 集合划分 — 第二类 Stirling 数  $\{n\}_k$  表示把  $n$  个元素划分成  $k$  个非空集合的方案数。例如, 有七种方法把四个元素划分成两个非空集合:  $(1)(234)$ 、 $(12)(34)$ 、 $(13)(24)$ 、 $(14)(23)$ 、 $(123)(4)$ 、 $(124)(3)$  和  $(134)(2)$ 。对于  $n - 1$  个元素、 $k$  个集合的任意方案, 第  $n$  个元素可以插入到  $k$  个集合中的某一个, 或者被分配到一个新集合中。和第一类 Stirling 数类似, 这也可以用一个递推关系表示:  $\{n\}_k = k \{n - 1\}_{k - 1} + \{n - 1\}_k$ 。作为一个特例,  $\{n\}_2 = 2^{n - 1} - 1$ ,

<sup>2</sup> 译者注：还有很多组合数都称为“欧拉数”，请不要与这里介绍的欧拉数弄混。

<sup>3</sup> 译者注：例如 3,6,1,4,5,2 包含两个下降（6 到 1, 5 到 2），三个上升段（3,6；1,4,5；2）。

<sup>4</sup> 译者注：还有一个略有不同的记号  $A(n, k)$  (有时也记为  $A_{n,k}$ )，它表示包含  $k$  个上升段的排列个数。不难看出， $A(n, k) = \binom{n}{k-1}$ 。

<sup>5</sup> 译者注：为了理解循环，首先应该把排列看作置换。例如排列  $\{3, 2, 4, 1\}$  表示置换  $1 \rightarrow 3, 2 \rightarrow 2, 3 \rightarrow 4, 4 \rightarrow 1$ 。接下来可以把置换分解成循环乘积的形式。其中每个循环  $(a_1 a_2 \dots a_k)$  表示  $a_1 \rightarrow a_2, a_2 \rightarrow a_3, \dots, a_{k-1} \rightarrow a_k, a_k \rightarrow a_1$ 。例如  $\{3, 2, 4, 1\}$  可以分解为  $(134)(2)$ ，它包含两个循环。

<sup>6</sup> 译者注：事实上，第一类 Stirling 数有带符号和无符号两个版本，这里特指无符号第一类 Stirling 数。

因为元素 2 到  $n$  的任何一个非空集合都可以与 (1) 合并, 这定义了一个集合划分。该划分的第二个部分包含了所有不在第一部分出现的元素。

- 整数拆分 ——  $n$  的整数拆分是总和等于  $n$  的若干正整数的无序多重集。例如, 5 有七种整数拆分法, 即: (5)、(4, 1)、(3, 2)、(3, 1, 1)、(2, 2, 1)、(2, 1, 1, 1) 和 (1, 1, 1, 1, 1)。给它计数的最简单方法是定义函数  $f(n, k)$ , 代表  $n$  的整数拆分中, 最大项不超过  $k$  的方案数。对于任意满足条件的拆分, 最大项要么达到这个限制, 要么小于它, 因此  $f(n, k) = f(n - k, k) + f(n, k - 1)$ 。边界条件是  $f(n, 1) = 1, f(0, 0) = 1, f(1, 1) = 1$  和  $f(n, k) = f(n, n)(k > n)$ 。

推荐有兴趣的读者阅读 [GKP89], 以了解这些序列的更多知识, 另外一些有趣的计数序列, 还可到 <http://www.research.att.com/~njas/sequences/> 在线访问 Sloane 的整数序列手册 (*Handbook of Integer Sequences*)。它几乎能识别出任何有趣的整数序列。

## 6.5 递归与数学归纳法

数学归纳法是求解递推关系的有力工具。当我们在高中时第一次学到数学归纳法时, 它简直就像是在变魔术。只需要验证 1 或 2 这样简单的情况, 假设 (*assumed*) 它对于到  $n - 1$  为止的一切整数成立, 然后就能用这个假设证明它对任意  $n$  都成立。这也算证明? 荒唐!

当我们在大学里第一次学到递归程序设计时, 它同样像是在变魔术。你的程序只需处理输入为 1 或 2 这样的简单情形。在其他情况下, 把输入分解成规模较小的若干部分, 然后让这个程序自己调用自己, 从而解决那些分解后的小规模问题。这也算程序? 荒唐!

二者看上去同样不可思议的原因是: 递归就是数学归纳法! 二者都有边界情况和一般情况, 其中一般情况把问题逐步分解成越来越小的子问题, 而初始条件 (或边界条件) 将终止递归。当你理解了递归或数学归纳法中的其中一个后, 你应当能够触类旁通, 想通另一个也成立的原因。

求解递推关系的一个有效方法是猜测出通解, 然后用数学归纳法证明。为了更好地猜测, 你可能需要把一些小规模的解罗列到一张表格中, 然后凝神注目, 直到找出其中的规律。

例如, 考虑如下的递推关系:

$$T_n = 2T_{n-1} + 1, T_0 = 0$$

它的前几项如下表:

$n$	0	1	2	3	4	5	6	7
$T_n$	0	1	3	7	15	31	63	127

你能猜到解吗? 你应当能注意到, 这些数值好像每次都加倍了 (正如我们在递推关系中直接看到的。这并不稀奇), 但又不完全是  $2^n$ 。用不了多长时间, 你就应当能得出如下猜

想:  $T_n = 2^n - 1$ 。最后, 我们只需证明这个猜想, 用归纳法的三个步骤来完成:

1. 验证归纳基础:  $T_0 = 2^0 - 1 = 0$ 。
2. 假设  $T_{n-1}$  满足公式。
3. 根据假设证明:

$$T_n = 2T_{n-1} + 1 = 2(2^{n-1} - 1) + 1 = 2^n - 1$$

在上述方法中, 猜测解的形式往往是最困难的部分。这也是技巧和经验派上用场的地方。最好多琢磨那些小规模的值, 并且对解的大致形式心里有底。

## 6.6 习题

### 6.6.1 斐波那契计数 (How Many Fibs?)

PC/UVa 题号: 110601/10183, 流行度: B, 通过率: average 难度: 1

我们知道, 斐波那契数的定义为:

$$\begin{aligned}f_1 &:= 1 \\f_2 &:= 2 \\f_n &:= f_{n-1} + f_{n-2} \quad (n \geq 3)\end{aligned}$$

给定两个整数  $a$  和  $b$ , 统计区间  $[a, b]$  内有多少个斐波那契数。

#### 输入

输入包含多行, 每行为一组数据, 包含两个不含前导 0 的非负整数  $a$  和  $b$  ( $a \leq b \leq 10^{100}$ )。输入以  $a = b = 0$  结束, 你不应当处理这一行。

#### 输出

对于每组数据, 输出一行, 仅包含一个整数, 表示满足  $a \leq f_i \leq b$  的  $f_i$  的个数。

#### 样例输入

10 100  
1234567890 9876543210  
0 0

#### 样例输出

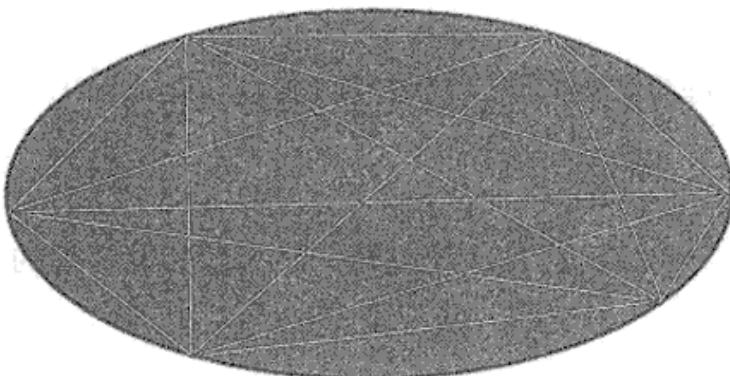
5  
4

### 6.6.2 土地分割 (How Many Pieces of Land?)

PC/UVa 题号: 110602/10213, 流行度: B, 通过率: average 难度: 2

你有一块椭圆形的土地。你可以在它的边界上任意挑选  $n$  个点, 然后用直线段连接每

两个不同的点对，共得到  $n(n - 1)/2$  条线段。如果你精心挑选这  $n$  个点的位置，这些线段最多可以把土地分成多少个部分？



$n = 6$  时的一种土地分割方案

### 输入

输入第一行包含一个整数  $s(0 < s < 3\,500)$ ，即数据组数。接下来的  $s$  行每行包含一个整数  $n(0 \leq n < 2^{31})$ 。

### 输出

对于每组数据，输出一行，仅包含一个整数，表示  $n$  个点的连线最多能把土地划分成多少块。

#### 样例输入

```
4
1
2
3
4
```

#### 样例输出

```
1
2
4
8
```

### 6.6.3 数数 (Counting)

PC/UVa 题号: 110603/10198, 流行度: B, 通过率: high 难度: 2

Gustavo 会数数，但还不太会写。他已经学了数字 1、2、3 和 4，但总是把 1 和 4 搞混。他认为 4 只是 1 的另一种写法。

他最近沉浸在他自创的一个小游戏中：用他学过的 4 个数字组成一些数，然后把这些数字加起来。例如：

$$132 = 1 + 3 + 2 = 6$$

$$112314 = 1 + 1 + 2 + 3 + 1 + 1 = 9 \text{ (别忘了 Gustavo 认为 4 是 1 的另一种写法)}$$

Gustavo 想知道有多少个数的数字之和（按照他自己的逻辑）恰好为  $n$ 。例如，当  $n = 2$  时，有 5 个数：11、14、41、44 和 2。可是，当  $n$  比 2 大的时候，他就算不过来了。于是，他

请求你的帮助。

## 输入

输入包含多组数据，每组数据包含一个整数  $n(1 \leq n \leq 1000)$ 。输入以文件结束符结尾。

## 输出

对于每个整数，输出一行，仅包含一个整数，表示有多少个数的数字之和（按照 Gustavo 的逻辑）为  $n$ 。

### 样例输入

2  
3

### 样例输出

5  
13

## 6.6.4 括号表达式 (Expressions)

PC/UVa 题号: 110604/10157, 流行度: C, 通过率: average 难度: 2

$X$  表示所有合法的括号表达式 (*correctly built parenthesis expressions*) 组成的集合。 $X$  中的元素均为只包含字符 “(” 和 “)” 的字符串，定义如下：

- 空串属于  $X$ 。
- 如果  $A$  属于  $X$ ，则  $(A)$  也属于  $X$ 。
- 如果  $A$  和  $B$  都属于  $X$ ，则它们的连接串  $AB$  也属于  $X$ 。

例如，表达式  $()((())()$  和  $((()())()$  都是合法的括号表达式，因此属于集合  $X$ 。表达式  $(())()$  和  $)()$  都不是合法的括号表达式，因此不属于集合  $X$ 。

合法的括号表达式  $E$  的长度 (*length*) 是它所包含的括号 (字符) 个数。 $E$  的深度 (*depth*)  $D(E)$  递归定义如下：

$$D(E) = \begin{cases} 0 & \text{若 } E \text{ 为空} \\ D(A) + 1 & \text{若 } E = (A), \text{ 且 } A \text{ 在 } X \text{ 中} \\ \max(D(A), D(B)) & \text{若 } E = AB, \text{ 且 } A \text{ 和 } B \text{ 都在 } X \text{ 中} \end{cases}$$

例如， $()((())()$  的长度为 8，深度为 2。编程读入  $n$  和  $d$ ，统计长度为  $n$ ，深度为  $d$  的合法括号表达式的数目。

## 输入

输入包含若干行，每行为一组数据，包含两个整数  $n$  和  $d$ ，满足  $2 \leq n \leq 300, 1 \leq d \leq 150$ 。输入中可能包含空行，你应当忽略它们。

## 输出

对于每组输入，输出一行，仅包含一个整数，表示长度为  $n$ ，深度为  $d$  和合法括号表达式的个数。

### 样例输入

```
6 2
300 150
```

### 样例输出

```
3
1
```

说明：长度为 6、深度为 2 的三个合法括号表达式分别是  $((())$ 、 $(()()$  和  $(()()$ 。

## 6.6.5 完全树标号 (Complete Tree Labeling)

PC/UVa 题号: 110605/10247, 流行度: C, 通过率: average 难度: 2

完全  $k$  叉树 (complete  $k$ -ary tree) 是一种特殊的  $k$  叉树，它的所有叶子位于同一层，并且所有内部结点恰好有  $k$  个因子。很容易算出这样的树中有多少个结点。

给出深度  $d$  和分支因子  $k$ ，你需要统计有多少种方法给一棵完全  $k$  叉树中的每个结点标号。标号原则是：每个结点的标号小于它所有后代的标号 ( $k = 2$  时，这正是二叉堆所具有的堆性质)。你的任务是统计标号的方案数。对于一棵  $n$  个结点的树，标号范围是  $(1, 2, 3, \dots, n - 1, n)$ 。

## 输入

输入包含多组数据。每组数据单独占一行，包含两个整数  $k$  和  $d$ ，其中  $k > 0$  是分支因子， $d > 0$  是深度。输入满足  $k \times d \leq 21$ 。

## 输出

对于每组数据，输出一行，仅包含一个整数，即标号方案数。

### 样例输入

```
2 2
10 1
```

### 样例输出

```
80
3628800
```

## 6.6.6 牧师数学家 (The Priest Mathematician)

PC/UVa 题号: 110606/10254, 流行度: C, 通过率: high 难度: 2

汉诺塔 (*Towers of Hanoi*) 背后的传说已经为人们所熟知。有人说，当 Brahmin 的和尚们得知移动全部 64 个圆盘所需的时间后，他们决定寻找一个更快的方法。



## 四根柱的汉诺塔

正在这时，寺庙里的一位牧师出现了。他告诉和尚们，只需加一根柱子，在每秒一个圆盘的速度下一个下午就可以完成任务。具体策略如下：

- 将顶部的若干个盘子（设为  $k$  个盘子）移动到其中一个不是目标的柱子。
  - 用标准的汉诺塔移动策略把剩下的  $n - k$  个盘子移到目标柱子。
  - 用这里描述的四柱汉诺塔移动策略把第一步移出的  $k$  个盘子移到目标柱子。

通过计算最优的  $k$  值，牧师算出了最少需要 18 433 次移动就能完成任务。换句话说，只需 5 小时 7 分 13 秒就能解决原本需要五千年的任务。

试着领会这个聪明的牧师的方法，并计算在有四根柱子的情况下移动的步数。移动的规则和传统汉诺塔一样，即每一步只能移动一个圆盘，并且只能放在一个大于它的圆盘上方。当然，最后目的是要在遵守牧师的策略的前提下，让移动的步数最少。

输入

输入包含若干行，每行包含一个整数  $0 \leq N \leq 10\,000$ ，即需要移动的圆盘数。输入以文件结束符终止。

输出

对于每组数据，输出一行，仅包含一个整数，即移动圆盘的最少步数。

### 样例输入

## 样例输出

1

1

2

3

28

769

64

18433

### 6.6.7 自描述序列 (Self-describing Sequence)

PC/UVa 题号: 110607/10049, 流行度: C, 通过率: high 难度: 2

Solomon Golomb 的自描述序列 (*self-describing sequence*)  $\langle f(1), f(2), f(3), \dots \rangle$  是唯一一个具有如下性质的不下降正整数序列：对于任意正整数  $k$ ，该序列恰好包含  $f(k)$  个  $k$ 。不难

得出，该序列的开头一定如下表所示：

$n$	1	2	3	4	5	6	7	8	9	10	11	12
$f(n)$	1	2	2	3	3	4	4	4	5	5	5	6

写一个程序，对于给定的  $n$  计算出  $f(n)$  的值。

### 输入

输入包含多组数据。每组数据在单独的一行中包含一个整数  $n$  ( $1 \leq n \leq 2\,000\,000\,000$ )。  
输入以  $n = 0$  结束，你不应当处理这一行。

### 输出

对于每组数据，输出一行，包含一个整数  $f(n)$ 。

#### 样例输入

100  
9999  
123456  
1000000000  
0

#### 样例输出

21  
356  
1684  
438744

### 6.6.8 数轴行走 (Steps)

PC/UVa 题号：110608/846，流行度：A，通过率：high 难度：2

你需要在数轴上从一个整数位置  $x$  走到另一个整数位置  $y$ 。每一步的长度必须是非负整数，且要么比上一步步长恰好大 1，要么相等，要么小 1。

若第一步和最后一步的长度均必须是 1，从  $x$  到  $y$  最少需要多少步？

### 输入

输入第一行包含一个整数  $n$ ，即测试数据的组数。以下每行为一组数据，包含两个整数  $x, y$  ( $0 \leq x \leq y < 2^{31}$ )。

### 输出

对于每组数据，输出一行，仅包含一个整数，即从  $x$  到  $y$  所需的最小步数。

#### 样例输入

3  
45 48  
45 49  
45 50

#### 样例输出

3  
3  
4

## 6.7 提 示

- 6.1 能否借助于  $F_n$  的封闭形式来避免高精度算术?
- 6.2 能否找到所求值的递推公式?
- 6.3 能否找到所求值的递推公式?
- 6.4 能否找到递推公式? 也许那会是 Catalan 数的二维版本?
- 6.5 能否找到所求值的递推公式?
- 6.7 可以显式的生成这个序列, 还是由于内存限制的关系必须使用某种更巧妙的方法?
- 6.8 最优解对应的步长序列具有怎样的特点?

## 6.8 注 解

- 6.6 尽管本题要求在给定策略下寻找最快的方案, 但人们还不知道这个策略本身是否最优!  
关于更多讨论, 参见 [GKP89]。

## 第7章 数论

数论也许是最有趣和优美的数学分支。欧几里德对于素数无穷性的证明在今天看来仍和两千年前一样清晰和优雅。“当  $n > 2$  时方程  $a^n + b^n = c^n$  是否有整数解”这样“面善”的问题往往无比艰深。事实上，这就是著名的费马大定理 (Fermat's last theorem)！

若想训练正规、严密的思维方式，数论是绝佳的选择，因为数论中的证明往往是清晰和无可驳斥的。学习整数是一件很有意思的事情，因为它们具体而且重要。发现整数的新奇性质相当于发现了大自然中又一奇妙的现象。

长久以来，计算机都被用来辅助数论研究。在大整数上进行有趣的数论计算需要很高的效率。所幸的是，有很多精妙的算法能够帮上忙。

### 7.1 素数

对于整数  $p > 1$ ，若它只能被 1 和他自身整除，则称  $p$  为素数 (*prime number*)。换句话说，若  $p$  是素数，且  $p = a \cdot b$  (其中  $a$  和  $b$  都是整数，且  $a \leq b$ )，则  $a = 1$ ,  $b = p$ 。前 10 个素数分别为：2, 3, 5, 7, 11, 13, 17, 19, 23 和 29。

素数的重要性在算术基本定理 (*fundamental theorem of arithmetic*) 中得以体现。这个定理的名字听上去很高级，但它的内容不过是：每个整数表示成素数乘积的方式只有一种。例如，105 的唯一分解式是  $3 \times 5 \times 7$ , 32 的唯一分解式是  $2 \times 2 \times 2 \times 2 \times 2$ 。 $n$  的唯一分解式中的那些素数称为  $n$  的素因数分解 (*prime factorization*)。在素因数分解中，素数是没有顺序之分的。为了方便起见，我们一般按照从小到大的顺序将它们列出。但是每个素数出现的次数很重要，否则 4 和 8 就没有区别了。

如果一个素数  $p$  在整数  $x$  的素因数分解中出现，则称它为  $x$  的因数 (*factor*)，也称约数。大于 1 且不是素数的整数称为合数 (*composite*)。

#### 7.1.1 寻找素数

测试一个给定整数  $x$  是否为素数的最简单的方法是不断的试除：从最小可能的约数开始，尝试所有可能整除它的数。由于 2 是唯一的偶素数，所以当验证了  $x$  不是偶数后，只需尝试奇数。另外，试除法只需做到  $\sqrt{x}$  即可。如果不超过  $\sqrt{x}$  的大于 1 的整数都不是  $x$  的约数，则  $x$  一定是素数。我们用反证法来证明：若  $x$  是合数，但它最小的非平凡因子  $p$  大于  $\sqrt{x}$ 。如果是这样， $x/p$  也能整除  $x$ ，而且至少和  $p$  一样大（否则与  $p$  是最小非平凡因子矛盾）。但两个大于  $\sqrt{x}$  的整数之积肯定大于  $x$ ，与  $p \times x/p = x$  矛盾。

分解素因数不仅需要找到第一个素因子，还需要把它除干净后继续寻找：

```
prime_factorization(long x)
{
    long i;                      /* counter */
    long c;                      /* remaining product to factor */

    c = x;
    while ((c % 2) == 0) {
        printf("%ld\n", 2);
        c = c / 2;
    }

    i = 3;
    while (i <= (sqrt(c)+1)) {
        if ((c % i) == 0) {
            printf("%ld\n", i);
            c = c / i;
        }
        else
            i = i + 2;
    }

    if (c > 1) printf("%ld\n", c);
}
```

测试终止条件  $i > \sqrt{c}$  会遇到点小麻烦，因为 `sqrt()` 会有精度问题。为了保险起见，我们多做了一次试除。另一个方法是避免浮点运算，而改用  $i*i > c$  作为终止条件。可是，当  $i$  很大时，乘法可能会溢出。注意到  $(i+1)^2 = i^2 + 2i + 1$ ，因此给  $i^2$  加上  $i + i + 1$  就得到了  $(i+1)^2$ 。这样，我们就避免了乘法。

为了提高效率，我们可以把 `sqrt(c)` 的计算放到循环体外，仅当  $c$  改变时才更新它的值。不过，现在的程序在我的机器上已经可以瞬间判断出 2 147 483 647 是素数了，没必要加这么多优化。存在一些巧妙的随机算法能更有效地测试超大整数是否为素数，但对于普通的问题规模，就不需要学习这些算法了。

### 7.1.2 素数的个数

一共有多少个素数？之所以这么问，是因为随着数值增大，素数越来越稀少。或许，素数

会最终消失？答案是：不会。欧几里德用一个精彩的反证法证明了素数是无穷多的。参加编程竞赛并不需要学习这个证明，但一般认为，每个受过高等教育的人都应该熟知它。下面我们就给出这个证明。

假设结论不成立，即只有有限多的素数， $p_1, p_2, \dots, p_n$ 。令  $m = 1 + \prod_{i=1}^n p_i$ ，即所有素数的乘积，再加 1。因为这个  $m$  比所有的素数都大，因此一定是合数。换句话说，某个素数能够整除它。

哪一个素数能整除它呢？不难知道， $m$  不能被  $p_1$  整除，因为  $m$  除以  $p_1$  的余数为 1。同样的， $m$  也不能被  $p_2$  整除，因为余数也是 1。事实上，当  $m$  除以任何一个  $p_i (1 \leq i \leq n)$  时，余数总是 1。如果  $m$  真的是合数，唯一的可能性是： $p_1, p_2, \dots, p_n$  并没有包含所有的素数。但这又和假设矛盾。结论：素数有无穷多个！证毕！<sup>1</sup>

素数不仅是无穷多的，而且还不算特别少。不超过  $x$  的素数大约有  $x/\ln x$  个。换句话说，大约每  $\ln x$  个整数中就有一个是素数。

## 7.2 整除性

整数的整除性在数论中占有重要地位。对于整数  $a$  和  $b$ <sup>2</sup>，若存在整数  $k$  使得  $a = bk$ ，则称  $b$  整除 (*divides*)  $a$  (用  $b|a$  来表示)。 $b|a$  也可以说成  $b$  是  $a$  的约数 (*divisor*)，或者  $a$  是  $b$  的倍数 (*multiple*)。

作为该定义的简单推论，任何非零整数的最小非负约数是 1。原因是，不存在整数  $k$  使得  $a = 0 \cdot k$ 。

如何找出一个给定整数的所有约数？根据唯一分解定理， $x$  能唯一的表示成它的素因数的乘积。每个约数都是这些素因数的某个子集的元素之积。这些子集可以用回溯法来构造 (将在第 8 章中讨论)，但请注意小心处理重复的素因数<sup>3</sup>。例如 12 的素因数分解式包含三项：(2, 2, 3)，但 12 只有 6 个约数：(1, 2, 3, 4, 6, 12)。

### 7.2.1 最大公约数

因为 1 能整除所有整数，因此任意两个整数  $a$  和  $b$  的最小公约数都是 1。相比之下，最大公约数 (*greatest common divisor*) (也称 *gcd*) 就要有趣得多了。对于一个分数  $x/y$ ，例如  $24/36$ ，它的既约形式 (*reduced form*) 可以通过把分子和分母同时除以  $\gcd(x, y)$  (在这个例子中是 12) 来得到。如果两个整数的最大公约数是 1，我们称二者是互素 (*relatively prime*) 的。

求解最大公约数的 Euclid 算法被人们认为是历史上第一个有趣的算法。求解最大公约数的朴素算法是测试第一个整数的所有约数，看它们是否能整除第二个整数，或者分别求

<sup>1</sup> 我们准备了一个小问题来测试你是否很好地理解了这个证明：假设我们取前  $n$  个素数，乘在一起，然后加 1。这样得到的结果一定是素数吗？给出证明或者反例。

<sup>2</sup> 译者注：本章只考虑非负整数。

<sup>3</sup> 译者注：如果需要在没有给出素因数分解式的情况下枚举所有的约数，则只需简单的用试除法即可。

出二者的素因数分解式，然后把所有公共素因子乘起来。遗憾的是，两种方法的运算量都很大。

Euclid 算法建立在两个事实的基础上。首先，如果  $b|a$ ，则  $\gcd(a, b) = b$ 。道理显而易见：如果  $b$  整除  $a$ ，则存在整数  $k$ ，使得  $a = bk$ ，因此  $\gcd(bk, b) = b$ 。其次，如果存在整数  $t$  和  $r$ ，使得  $a = bt + r$ ，则  $\gcd(a, b) = \gcd(b, r)$ 。为什么？根据定义， $\gcd(a, b) = \gcd(bt + r, b)$ 。由于  $bt$  是  $b$  所有约数的倍数， $a$  和  $b$  的所有公约数都应该能整除  $r$ 。

Euclid 算法是递归的，它不停地把较大的整数替换成它除以较小整数的余数。这种做法往往把其中一个参数缩小一半左右，因此在对数级别的迭代次数后达到边界。考虑下面的例子。令  $a = 34398$ ,  $b = 2132$ 。

$$\begin{aligned}\gcd(34398, 2132) &= \gcd(34398 \bmod 2132, 2132) = \gcd(2132, 286) \\ \gcd(2132, 286) &= \gcd(2132 \bmod 286, 286) = \gcd(286, 130) \\ \gcd(286, 130) &= \gcd(286 \bmod 130, 130) = \gcd(130, 26) \\ \gcd(130, 26) &= \gcd(130 \bmod 26, 26) = \gcd(26, 0)\end{aligned}$$

因此， $\gcd(34398, 2132) = 26$ 。

除了  $\gcd(a, b)$  之外，Euclid 算法还能给出更多的信息。它能找出两个  $x$  和  $y$ ，使得

$$a \cdot x + b \cdot y = \gcd(a, b)$$

它将在求解线性同余式中发挥重要的作用。我们知道， $\gcd(a, b) = \gcd(b, a')$ ，其中  $a' = a - b\lfloor a/b \rfloor$ 。根据数学归纳法原理，假设我们已经找出了整数  $x'$  和  $y'$ ，使得

$$b \cdot x' + a' \cdot y' = \gcd(a, b)$$

把  $a'$  的表达式代入上式，得到：

$$b \cdot x' + (a - b\lfloor a/b \rfloor) \cdot y' = \gcd(a, b)$$

重新整理后即可得到  $x$  和  $y$  的表达式。我们还需要验证算法的边界情况，不过这很容易： $a \cdot 1 + 0 \cdot 0 = \gcd(a, 0)$ 。

对于前面的例子， $34398 \times 15 + 2132 \times -242 = 26$ 。下面是该算法的一个实现：

```
/*
   Find the gcd(p,q) and x,y such that p*x + q*y = gcd(p,q)
*/
long gcd(long p, long q, long *x, long *y)
{
    long x1,y1; /* previous coefficients */
```

```

long g;                                /* value of gcd(p,q) */

if (q > p) return(gcd(q,p,y,x));

if (q == 0) {
    *x = 1;
    *y = 0;
    return(p);
}

g = gcd(q, p%q, &x1, &y1);

*x = y1;
*y = (x1 - floor(p/q)*y1);

return(g);
}

```

### 7.2.2 最小公倍数

两个整数的另一个有用函数是最小公倍数 (*least common multiple*) (*lcm*)，同时也是能被这两个数整除的最小整数。例如，24 和 36 的最小公倍数是 72。

当我们需要计算两个不同周期事件的公共周期时，需要用到最小公倍数。总统选举 (4 年一次) 和人口普查 (10 年一次) 多长时间同时举行一次？二者每 20 年同时举行一次，因为  $\text{lcm}(4, 10) = 20$ 。

显然  $\text{lcm}(x, y) \geq \max(x, y)$ 。类似的，因为  $x \cdot y$  是  $x$  和  $y$  的公倍数， $\text{lcm}(x, y) \leq xy$ 。只有当  $x$  和  $y$  有非平凡因子时，它们的最小公倍数才会严格地小于这个值。

这与 Euclid 算法一起为我们提供了一个计算最小公倍数的有效算法，即  $\text{lcm}(x, y) = xy/\text{gcd}(x, y)$ 。[\[Dij76\]](#) 中介绍了一个更为精巧的算法，它避免了乘法，因此不会溢出<sup>4</sup>。

## 7.3 模 算 术

在第 5 章中，我们复习了加法、乘法等整数的基本算术运算。但在有些时候，我们并不关心完整的答案，而只需要它们除以某个整数后的余数。例如，假设你今年的生日在星期三，那么明年的生日会是星期几呢？为了算出结果，你只需要知道两年生日之间的间隔（要么是

<sup>4</sup> 译者注：只需按照  $\text{lcm}(x, y) = (x/\text{gcd}(x, y)) \times y$  的计算顺序，只要结果不超界，中间结果也不会溢出。

365, 要么是 366) 除以 7 的余数。答案是星期三加上一 ( $365 \bmod 7$ ) 或者二 ( $366 \bmod 7$ ), 即星期四或者星期五 (具体结果还取决于计算是否受到闰年影响)。

高效计算这些余数的关键是模算术 (*modular arithmetic*)。当然, 理论上我们可以显式的计算出完整结果, 然后求余数, 但对于足够大的整数, 用模算术直接对余数进行运算要容易得多。

在模算术中, 除数称为模 (*modulus*), 余数称为剩余 (*residue*)。高效模算术的关键在于懂得在一个给定的模下, 基本的加法、减法和乘法运算是如何进行的:

- 加法 —  $(x + y) \bmod n$  等于多少? 我们可以把它变成

$$((x \bmod n) + (y \bmod n)) \bmod n$$

来避免大整数加法。如果妈妈给我 \$123.45, 爸爸给我 \$94.67, 我将拥有多少零钱?

$$(12,345 \bmod 100) + (9,467 \bmod 100) = (45 + 67) \bmod 100 = 12 \bmod 100$$

- 减法 — 减法只是加上一个负数。花掉 \$52.53 以后, 我将拥有多少零钱?

$$(12 \bmod 100) - (53 \bmod 100) = -41 \bmod 100 = 59 \bmod 100$$

注意负数模  $n$  可以通过加上  $n$  的正整数倍来转换成正数模  $n$ 。并且在这个例子中, 这样做是有实际意义的。一般来讲, 我们需要让剩余总是保持在 0 和  $n - 1$  之间, 以将我们需要处理的整数范围减到最小限度。

- 乘法 — 由于乘法只是重复加法,

$$xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$$

如果你每小时能挣 \$17.28, 经过 2 143 小时后会拥有多少零钱?

$$(1\,728 \times 2\,143) \bmod 100 = (28 \bmod 100) \times (43 \bmod 100) = 4 \bmod 100$$

更进一步, 由于乘方运算只是重复的乘法,

$$x^y \bmod n = (x \bmod n)^y \bmod n$$

而乘方是得到超大整数的最快途径, 这正是模算术真正发挥作用的地方。

- 除法 — 除法的处理要困难很多, 我们将在 7.4 节中继续讨论。

模算术有很多有趣应用, 包括:

- 求最后一位数 —  $2^{100}$  的最后一位数是什么? 当然我们可以利用高精度乘法把这个具体值求出来, 然后看看最后一位是什么, 但是何必呢? 我们用手算就能完成。所求结果实

际上是  $2^{100} \bmod 10$ 。我们只需不断平方，并在每步后求  $\bmod 10$  的值，就可以很快达到目的：

$$\begin{aligned}2^3 \bmod 10 &= 8 \\2^6 \bmod 10 &= 8 \times 8 \bmod 10 \rightarrow 4 \\2^{12} \bmod 10 &= 4 \times 4 \bmod 10 \rightarrow 6 \\2^{24} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\2^{48} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\2^{96} \bmod 10 &= 6 \times 6 \bmod 10 \rightarrow 6 \\2^{100} \bmod 10 &= 2^{96} \times 2^3 \times 2^1 \bmod 10 \rightarrow 6\end{aligned}$$

- **RSA 加密算法** — 大整数模算术的经典应用之一是公钥加密算法 RSA。首先把原文编码成整数  $m$ ，然后计算它的  $k$  次方，其中  $k$  就是人们常说的公钥（或密钥），然后把结果对模  $n$  求剩余。由于  $m$ 、 $n$  和  $k$  都是巨大的整数，高效的计算  $m^k \bmod n$  需要我们刚才介绍的方法。
- **日历计算** — 正如在前面那个关于生日的例子中所看到的，计算若干天之后是星期几，或者若干秒之后的时间，都是模算术的应用。

## 7.4 同余

同余 (*congruences*) 是表示模算术的另一个记号：当  $m|(a-b)$  时，我们说  $a \equiv b \pmod{m}$ 。根据定义，若  $a \bmod m = b$ ，则  $a \equiv b \pmod{m}$ 。

尽管同余并不是一个全新的思想，记号的重要性不容忽视。它让我们考虑模  $n$  后等于某个特定值的整数集合，并且用方程的形式来表示它们。假设  $x$  是一个变量，哪些整数  $x$  满足同余式  $x \equiv 3 \pmod{9}$ ？

对于这样简单的同余式，答案也很简单。显然  $x = 3$  本身就是一个解。另外，增加或者减少模（在这个例子中是 9）将得到另外一个解。因此，通解可以表示成  $9y + 3$ ，其中  $y$  是任意整数。

如何求解复杂一些的同余式，如  $2x \equiv 3 \pmod{9}$  和  $2x \equiv 3 \pmod{4}$ ？尝试几次后，你将发现第一个方程的通解是  $9y + 6$ ，而第二个方程无解。

同余式的两大问题是：同余变形；解方程。我们将在接下来的小节中讨论它们。

### 7.4.1 同余运算

同余式支持加法、减法、乘法和一种受限的除法。上述运算的共同前提是：两个同余式的模必须相同：

- 加减法 — 假设  $a \equiv b \pmod{n}$ ,  $c \equiv d \pmod{n}$ , 则  $a + c \equiv b + d \pmod{n}$ 。例如, 假设  $4x \equiv 7 \pmod{9}$ , 并且  $3x \equiv 3 \pmod{9}$ , 那么

$$4x - 3x \equiv 7 - 3 \pmod{9} \rightarrow x \equiv 4 \pmod{9}$$

- 乘法 — 若  $a \equiv b \pmod{n}$ , 用它自己加自己  $d$  次后可以得到  $a \cdot d \equiv b \cdot d \pmod{n}$ 。事实上, 更一般的乘法也成立, 即  $a \equiv b \pmod{n}$  和  $c \equiv d \pmod{n}$  可以推出  $ac \equiv bd \pmod{n}$ 。
- 除法 — 可惜, 我们并不能简单地无视同余式中公因子的存在。注意到  $6 \cdot 2 \equiv 6 \cdot 1 \pmod{3}$ , 但显然  $2 \not\equiv 1 \pmod{3}$ 。为了看清问题的所在, 我们重新用乘法的逆来定义除法, 即  $a/b$  等价于  $ab^{-1}$ 。因此, 我们只需要找到满足  $bb^{-1} \equiv 1 \pmod{n}$  的逆  $b^{-1}$ , 就能计算出  $a/b \pmod{n}$ 。然而, 逆并不总是存在的。不信的话, 试试  $2x \equiv 1 \pmod{4}$ 。

所幸的是, 我们可以把同余式  $ad \equiv bd \pmod{dn}$  化简为  $a \equiv b \pmod{n}$ 。换句话说, 我们可以把三项同时除以三者的公因子 (如果存在的话)。因此,  $170 \equiv 30 \pmod{140}$  可以推出  $17 \equiv 3 \pmod{14}$ 。然而, 若  $\gcd(a, n)$  不能整除  $b$ , 同余式  $a \equiv b \pmod{n}$  一定无解。

#### 7.4.2 求解线性同余式

线性同余式是指形如  $ax \equiv b \pmod{n}$  的方程。求解它就是要找出哪些整数  $x$  满足该方程。

并不是所有方程都有解。我们已经看到了一些对于特定模不存在逆的整数, 换句话说, 方程  $ax \equiv 1 \pmod{n}$  无解。事实上,  $ax \equiv 1 \pmod{n}$  有解当且仅当模和一次项系数互素, 即  $\gcd(a, n) = 1$ 。我们可以先用 Euclid 算法找到  $a \cdot x' + n \cdot y' = \gcd(a, n) = 1$  的解, 进而完成求逆:

$$ax \equiv 1 \pmod{n} \rightarrow ax \equiv a \cdot x' + n \cdot y' \pmod{n}$$

显然,  $n \cdot y' \equiv 0 \pmod{n}$ , 因此所求的逆就是 Euclid 算法输出的  $x'$ 。

对于更加一般的方程, 需要根据  $a$ 、 $b$  和  $n$  的关系分成三种情况讨论:

- $\gcd(a, b, n) > 1$  — 我们可以将三者同时除以这个公约数, 得到一个等价的新同余式。这是新模下的唯一解, 等价于  $(\pmod{n})$  下的  $\gcd(a, b, n)$  个不同解。
- $\gcd(a, n)$  不能整除  $b$  — 正如刚才所说, 同余式无解。
- $\gcd(a, n) = 1$  — 在模  $n$  意义下只有唯一解  $x = a^{-1}b$ , 因为  $aa^{-1}b \equiv b \pmod{n}$ 。前面已经说过, 在这种情况下, 逆是存在的, 并且可以用 Euclid 算法求出。

中国剩余定理 (*Chinese remainder theorem*) 让我们能够处理不同模下的同余方程组。假设存在整数  $x$  满足  $x \equiv a_1 \pmod{m_1}$  和  $x \equiv a_2 \pmod{m_2}$ 。则当  $m_1$  和  $m_2$  互素时,  $x$  在模  $m_1 m_2$  意义下有唯一解。

为了找出满足条件的  $x$ , 进而解决两个同余式组成的方程组, 首先要从两个同余式

$m_2 b_1 \equiv 1 \pmod{m_1}$  和  $m_1 b_2 \equiv 1 \pmod{m_2}$  解出  $b_1$  和  $b_2$ 。接下来，很容易验证：

$$x = a_1 b_1 m_2 + a_2 b_2 m_1$$

是两个原始同余式的公共解。更进一步，这个方法很容易推广到任意数量的同余式，只要它们的模两两互素。

#### 7.4.3 不定方程

不定方程 (*Diophantine equations*) 是变量被限制在整数范围内的方程。例如，Fermat 大定理就是关于不定方程  $a^n + b^n = c^n$  的。在实数范围内求解这些方程往往并不困难，但如果所有变量必须是整数，问题的难度陡然提升。

不定方程难以处理，因为整数除以整数不一定仍是整数。然而，一些特殊的不定方程是容易解出来的，而且它们出现得很频繁。

最重要的不定方程是线性不定方程  $ax - ny = b$ ，其中  $x$  和  $y$  是整数变量， $a$ 、 $b$  和  $n$  是整数常数。幸运的是，这个不定方程和线性同余式  $ax \equiv b \pmod{n}$  是等价的，因此可以用刚才介绍的方法求解。

更高级的不定方程分析已经超出了本书范围，有兴趣的读者可以阅读标准的数论参考书，包括 Niven 和 Zuckerman 的 [ZMNN91] 以及 Hardy 和 Wright 的 [HW79]。

## 7.5 数论函数库

Java 的 BigInteger 类 (`java.math.BigInteger`) 包含不少有用的数论函数，其中最重要的莫过于对高精度算术运算 (已在第 5 章中讨论) 的基本支持了。它还包括一些纯粹的数论函数：

- 最大公约数 — `BigInteger gcd(BigInteger val)` 返回一个 BigInteger，它是 `abs(this)` 和 `abs(val)` 的最大公约数。
- 乘方取模 — `BigInteger modPow(BigInteger exp, BigInteger m)` 返回一个 BigInteger，它是 `thisexp mod m` 的值。
- 同余取逆 — `BigInteger modInverse(BigInteger m)` 返回一个 BigInteger，它是  $this^{-1} \pmod{m}$  的值。换句话说，该函数求解同余式  $y \cdot this \equiv 1 \pmod{m}$ ，并且返回满足上式的整数  $y$  (如果存在)。
- 素数测试 — `public boolean isProbablePrime(int certainty)` 进行随机化素数测试。当这个 BigInteger 可能是素数时返回 `true`，当它一定是合数时返回 `false`。如果返回 `true`，则它确实是素数的概率  $\geq 1 - 1/2^{certainty}$ 。



## 7.6 习题

### 7.6.1 开灯与关灯 (Light, More Light)

PC/UVa 题号: 110701/10110, 流行度: A, 通过率: average 难度: 1

有一个叫做 Mabu 的人在大学的走廊里面反复开灯和关灯。每个灯泡都有它自己的开关，用来改变它的状态。如果灯是关着的，那么可以通过按这个开关来打开它，再按一次则会将它关上。最初的时候所有的灯都是关上的。

Mabu 做了一件很奇特的事情：如果走廊里有  $n$  个灯泡，他就在走廊里来回走  $n$  次。在他走第  $i$  次的时候，只按动可以被  $i$  整除的位置（位置编号为 1 到  $n$ ）上的开关。在他走回初始位置时，他不会按动任何开关。第  $i$  次走动被定义为走到走廊尽头（同时做着上述奇特的事情），然后走回来。你的任务是确定最后一个灯泡的最终状态。它是开着，还是关着的？

输入

输入每一行将给出一个走廊里的电灯总数  $n$  ( $n \leq 2^{32} - 1$ )。 $n = 0$  表示输入结束，你的程序不应处理这一行。

输出

如果灯泡开着，输出“yes”；如果是关着，输出“no”。每个数据在单独的一行输出。

### 样例输入

3	no
6241	yes
8191	no

### 7.6.3 Carmichael 数 (Carmichael Numbers)

PC/UVa 题号: 110702/10006, 流行度: A, 通过率: average 难度: 2

密码学中的一些算法用到了很大的素数，但是要检验这些大数是不是素数却不容易。

存在一些速度快、准确性高的随机素数测试，如费马测试。 $a$  是一个 2 到  $n - 1$  之间的随机数，其中  $n$  就是那个需要测试素数性的整数。如果  $n$  满足以下的等式，则  $n$  可能是素数：

$$a^n \bmod n = a$$

如果一个数可以多次通过费马测试，那么它是素数的可能性很高。

最新題庫

不幸的是，这里有一个坏消息：存在一些特定的合数  $n$ ，当  $a$  取遍小于  $n$  的所有可能值时都能通过测试。这些数被称为 Carmichael 数。

写一个程序来判断一个数是不是 Carmichael 数。

输入

输入包含若干行，每一行包含一个正整数  $n$  ( $2 < n < 65\,000$ )。 $n = 0$  为输入结束标志，你的程序不应当处理它。

输出

对于输入的每个数，按题样例输出一行，说明这个数是否为 Carmichael 数。

### 样例输入

1729  
17  
561  
1109  
431  
0

## 样例输出

The number 1729 is a Carmichael number.  
17 is normal.  
The number 561 is a Carmichael number.  
1109 is normal.  
431 is normal.

### 7.6.3 欧几里德问题 (Euclid Problem)

PC/UVa 题号: 110703/10104, 流行度: A, 通过率: average 难度: 1

欧几里德曾提出：对于任何正整数  $A$  和  $B$ ，一定存在一对整数  $X$  和  $Y$  使得  $AX + BY = D$ ，其中  $D$  是  $A$  和  $B$  的最大公约数。给定  $A$  和  $B$ ，你的任务是找出相应的  $X$ 、 $Y$  和  $D$ ，使得  $|X| + |Y|$  最小。

输入

输入包含若干行，每行有一对整数  $A$  和  $B$ ，用空格隔开 ( $A, B < 1\,000\,000\,001$ )。

输出

对于输入的每一行，输出一行，包含三个整数  $X$ 、 $Y$  和  $D$ ，用单个空格隔开。如果有多个组满足条件的  $X$  和  $Y$ ，应当保证  $X \leq Y$ 。

### 样例输入

4 6  
17 17

## 样例输出

$$\begin{matrix} -1 & 1 & 2 \\ 0 & 1 & 17 \end{matrix}$$

#### 7.6.4 阶乘与整除 (FactoVisors)

PC/UVa 题号: 110704/10139, 流行度: A, 通过率: average 难度: 2

对于一非负整数  $n$ ,  $n$  的阶乘定义如下:

$$0! = 1$$

$$n! = n \times (n - 1)! \quad (n > 0)$$

如果存在一个整数  $k$  使得

$$k \times a = b$$

则称  $a$  可以整除  $b$ 。

#### 输入

输入包含若干行, 每行有两个小于  $2^{31}$  的非负整数  $n$  和  $m$ 。

#### 输出

对于输入的每一行, 按照样例格式输出一行, 说明  $m$  是否可以整除  $n!$ 。

#### 样例输入

```
6 9
6 27
20 10000
20 100000
1000 1009
```

#### 样例输出

```
9 divides 6!
27 does not divide 6!
10000 divides 20!
100000 does not divide 20!
1009 does not divide 1000!
```

#### 7.6.5 四素数之和 (Summation of Four Primes)

PC/UVa 题号: 110705/10168, 流行度: A, 通过率: average 难度: 2

Waring 素数猜想是说: 每个奇数要么是素数, 要么可以写成三个素数之和。Goldbach 猜想是说: 每个偶数都能写成两个素数之和。这两个问题 200 年来尚未解决。

在本题中, 我们的任务要简单一点: 把一个给出的整数写成四个素数之和。

#### 输入

输入每一行都有一个整数  $n$  ( $n \leq 10\,000\,000$ )。输入以文件结束符结尾。

#### 输出

对于输入的每个数  $n$ , 输出一行, 包含和为  $n$  的四个素数。如果  $n$  不能写成 4 个素数之和, 则在单独的一行输出 “Impossible.”。如果有多种解法, 输出任意一种即可。

**样例输入**

24

36

46

**样例输出**

3 11 3 7

3 7 13 13

11 11 17 7

### 7.6.6 Smith 数 (Smith Numbers)

PC/UVa 题号: 110706/10042, 流行度: B, 通过率: average 难度: 1

数学家 Albert Wilansky 在 1982 年翻阅电话簿时偶然发现他姐夫 H. Smith 的电话有以下的奇特性: 电话号码的各个数字之和等于该号码所有素因子的每个数字之和。懂了么? Smith 的电话号码是 493-7775。它的素因子分解式为:

$$4937775 = 3 \cdot 5 \cdot 5 \cdot 65837$$

电话号码的各个数字之和为  $4 + 9 + 3 + 7 + 7 + 7 + 5 = 42$ , 而所有素因子的各个数字之和为  $3 + 5 + 5 + 6 + 5 + 8 + 3 + 7 = 42$ 。Wilansky 将这类数按姐夫的姓氏命名为 Smith 数。

因为每个素数都满足这个性质, 所以 Wilansky 将它们排除在 Smith 数的定义之外。其他的 Smith 数还包括 6 036 和 9 985。

Wilansky 没有找到过比刚才那个电话号码更大的 Smith 数, 你能帮助他吗?

#### 输入

输入包含若干组数据, 其中第一行为测试数据组数。每个数据位于单独的一行中, 包含一个小于  $10^9$  的正整数。

#### 输出

对于每一个输入值  $n$ , 计算出大于  $n$  的最小 Smith 数, 并在单独的一行输出它。假设这样的数总是存在的。

**样例输入**

1

4937774

**样例输出**

4937775

### 7.6.7 弹珠 (Marbles)

PC/UVa 题号: 110707/10090, 流行度: B, 通过率: low 难度: 1

我收藏了一些弹珠 (彩色的小玻璃球), 并且想买一些盒子来装它们。这些盒子有两种型号:

类型 1: 每个盒子  $c_1$  美元, 可以恰好装  $n_1$  个弹珠。

类型 2：每个盒子  $c_2$  美元，可以恰好装  $n_2$  个弹珠。

我想将每个盒子都装满，并且花费最少的钱来买盒子。请帮我找出最好的方法来把这些弹珠装到盒子里。

### 输入

输入包含多组数据，每组数据的开头是一个整数  $n(1 \leq n \leq 2\,000\,000\,000)$ 。第二行为  $c_1$  和  $n_1$ ，第三行为  $c_2$  和  $n_2$ 。这里的  $c_1, c_2, n_1, n_2$  都是小于  $2\,000\,000\,000$  的正整数。

$n$  为 0 标志着输入的结束。

### 输出

对于每组数据，输出最小花费的方案（两个非负整数  $m_1$  和  $m_2$ ，其中  $m_i$  表示第  $i$  种盒子的数量）。如果不存在解决方案，则输出“failed”。

你可以假定：如果有解，方案唯一。

#### 样例输入

```
43
1 3
2 4
40
5 9
5 12
0
```

#### 样例输出

```
13 1
failed
```

### 7.6.8 重新打包 (Repackaging)

PC/UVa 题号: 110708/10089, 流行度: C, 通过率: low 难度: 2

杯子制造协会 (Association of Cup Makers, ACM) 生产三种不同规格的杯子（规格 1、规格 2、规格 3），并以不同的包装出售。每种包装用 3 个正整数  $(S_1, S_2, S_3)$  来表示，其中  $S_i(1 \leq i \leq 3)$  表示该包装中规格  $i$  的杯子数量。不幸的是，没有一种包装中  $S_1 = S_2 = S_3$ 。

市场调研发现：这种  $S_1 = S_2 = S_3$  的包装需求量很大。为了抓住这个机会，ACM 决定在它的庞大库存中拆开一些没有出售的包装，重新组合后得到这种大需求量的包装。例如，假设 ACM 公司有以下的库存： $(1, 2, 3)$ ,  $(1, 11, 5)$ ,  $(9, 4, 3)$  和  $(2, 3, 2)$ 。我们可以将三个  $(1, 2, 3)$ , 一个  $(9, 4, 3)$  和两个  $(2, 3, 2)$  的包装拆开，然后重新组合成 16 个  $(1, 1, 1)$  的包装。你也可以把它们重新组合为八个  $(2, 2, 2)$ , 四个  $(4, 4, 4)$ , 两个  $(8, 8, 8)$ , 一个  $(16, 16, 16)$  等包装。注意拆封后的所有杯子都必须被重新组合到新的包装中，即不能浪费任何一个杯子。

ACM 请你写一个程序来判断是否能够从库存包装中选取一些，重新打包后得到三种规格的杯子数目相等的包装。

**输入**

输入包含若干组数据。每组数据的第一行为一个整数  $N$  ( $3 \leq N \leq 1\,000$ )，表示库存中不同种类包装的数量。接下来的  $N$  行每行有 3 个正整数，分别代表该包装中规格 1, 2, 3 这 3 种杯子的数量。对于每组数据，相同规格的包装不会重复出现。

当  $N$  为 0 时，输入结束。

**输出**

对于每组数据，如果可以按希望将它们重新包装，则输出 “Yes”，否则输出 “No”。

**样例输入**

```
4
1 2 3
1 11 5
9 4 3
2 3 2
4
1 3 3
1 11 5
9 4 3
2 3 2
0
```

**样例输出**

```
Yes
No
```

**7.7 提 示**

- 7.1 是否可以不通过模拟开关灯过程直接找出第  $n$  个灯泡的状态？
- 7.2 如何高效计算  $a^n \pmod n$ ？
- 7.3 本书介绍的方法是否能保证给出绝对值之和最小的解？
- 7.4 是否可以在不显式计算  $n!$  的情况下进行整除性测试？
- 7.7 是否可以在不考虑花费的情况下先求出通解？其中哪个解是最便宜的？
- 7.8 是否可以用本章介绍的方法求解不定方程？

**7.8 注 解**

- 7.5 Goldbach 猜想和 Waring 猜想几乎肯定是正确的，但这更多是因为穷举法提供的实证数据，而不是素数所具备的某些高深的性质。假定小于  $n$  的素数个数为  $n/\ln n$ ，读者可以自行估算每个数期望的表示方案总数。当  $n = 1\,000\,000$  之前都没有发现反例的情况下，在更大范围内寻找反例真的有前途吗？
- 7.6 可以在 [Wil82, McD87] 等论文中找到关于 Smith 数的更多性质。

## 第8章 回溯法

由于现代计算机的超高速度，穷举法已经成为了一个解决问题的有效途径。例如，为了统计一个集合中的元素个数，有时直接构造出它们比使用深奥的组合手段来进行计数要容易得多。当然，这要求元素不能太多，至少要少到运算能够终止的程度。

现代的个人计算机的时钟频率约为 1 GHz<sup>1</sup>，即每秒运算 10 亿次。由于大多数有用的操作需要数百条指令（甚至更多），所以每秒能搜索几百万个元素就不错了。

你应当知道一百万到底有多大（或者有多小）。一百万种排列约是 10 个元素的所有排列总数（元素不能再多了）。一百万种子集约是 20 个元素的所有组合（元素不能再多了）。如果需要求解更大规模的问题，我们需要在搜索中剪枝，以确保我们只会搜索到有必要搜索的元素。

在本章中，我们介绍穷举搜索中的回溯法，并为它设计有效的剪枝策略，让它尽可能的强大。

### 8.1 回溯法

回溯法是一种枚举状态空间中所有可能状态的系统方法。它是一个一般性的算法框架（或称技巧），应用时须具体问题具体分析。

对于一般情况，我们把解表示成向量  $a = (a_1, a_2, \dots, a_n)$ ，其中每个元素  $a_i$  取自一个有限有序集  $S_i$ 。这样的向量可以表示一个排列，其中  $a_i$  是排列中的第  $i$  个元素；它也可以表示一个子集  $S$ ，其中  $a_i$  为真当且仅当全集中的第  $i$  个元素在  $S$  中；它甚至可以表示游戏中的行动序列或者图中的路径，其中  $a_i$  表示序列中的第  $i$  个元素。

在回溯算法的每一步，我们从一个给定的部分解（partial solution） $a = (a_1, a_2, \dots, a_k)$  开始，尝试着在最后添加元素来扩展这个部分解。扩展之后，我们必须测试它是否为一个完整的解——如果是的话，需要输出这个解、更新解的计数器或者其他任何你想做的事。如果仍不完整，我们必须检查这个部分解是否仍有可能扩展成完整解。如果有可能，递归下去；如果没有可能，从  $a$  中删除新加入的最后一个元素，然后尝试该位置上的其他可能性（如果有的话）。

下面是代码。我们使用了一个全局的 `finished` 标志来允许提前终止回溯过程，它的设置时机和具体应用相关。

<sup>1</sup> 译者注：近年来，这个时钟频率不断提高，但目前仍处于这个数量级。

```

bool finished = FALSE;           /* found all solutions yet? */

backtrack(int a[], int k, data input)
{
    int c[MAXCANDIDATES];        /* candidates for next position */
    int ncandidates;             /* next position candidate count */
    int i;                      /* counter */

    if (is_a_solution(a,k,input))
        process_solution(a,k,input);
    else {
        k = k+1;
        construct_candidates(a,k,input,c,&ncandidates);
        for (i=0; i<ncandidates; i++) {
            a[k] = c[i];
            backtrack(a,k,input);
            if (finished) return; /* terminate early */
        }
    }
}

```

算法中和具体应用相关的部分包含三个子程序：

- `is_a_solution(a,k,input)` — 该函数测试向量  $a$  的前  $k$  个元素是否为一个完整解。最后一个参数 `input` 用来给这个函数传递一些其他信息。例如，可以用它来指定目标解的长度  $n$ 。在构造  $n$  个数的排列，或者  $n$  个数的子集时，这是有用的，不过对于构造游戏行动序列这样的解长度可变的应用，这个  $n$  就没有意义了。
- `construct_candidates(a,k,input,c,ncandidates)` — 该函数根据  $a$  的前  $k - 1$  个元素值，把第  $k$  个元素的所有候选值放到数组  $c$  中。候选值的数量为 `ncandidates`。同样的，`input` 可以用来传递附加信息，特别是目标解的长度。
- `process_solution(a,k)` — 当一个完整解被构造出来后，该函数输出、计数或者用其他某种方式处理这个解。注意这里没有必要用 `input`，因为  $k$  本身就是解的长度。

回溯法穷举了所有可能性，并以此保证了算法的正确性；它从不重复访问任何一个状态，并以此保证了算法的效率。

请注意递归是如何优美地实现回溯算法的。递归调用的每层都有自己的候选数组  $c$ ，因此每个位置上尚未考虑的候选集不会相互影响。我们将在第 9 章中看到一个和 `backtrack`

本质相同的递归算法。回溯可以看成是在隐式图 (implicit graph) 而非显式图上深度优先搜索的过程。

我们提供上述三个函数的两套具体实现。它们分别枚举  $n$  个元素的所有子集和所有排列。

## 8.2 构造所有子集

刚才已经提到过，只需枚举长度为  $n$  的布尔向量的全部  $2^n$  种情况，我们就得到了  $n$  个元素的所有子集。其中向量的第  $i$  维表示第  $i$  个元素是否在子集中。

用回溯框架中的记号来说， $S_k = (\text{true}, \text{false})$ ， $a$  是解的充要条件是  $k \geq n$ 。函数 `is_a_solution()`, `construct_candidates()` 和 `process_solution()` 的实现都非常简单——事实上，“每构造出一个子集就把它打印出来”是三个函数中最复杂的一个！

```
is_a_solution(int a[], int k, int n)
{
    return (k == n);                                /* is k == n? */
}

construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    c[0] = TRUE;
    c[1] = FALSE;
    *ncandidates = 2;
}

process_solution(int a[], int k)
{
    int i;                                         /* counter */

    printf("{");
    for (i=1; i<=k; i++)
        if (a[i] == TRUE) printf(" %d", i);

    printf(" }\n");
}
```

最后，我们需要用正确的参数对 `backtrack` 进行初始调用。具体来说，你应当提供一个指向空向量的指针，用  $k = 0$  表示向量为空，并指明全集中的元素个数：

```
generate_subsets(int n)
{
    int a[NMAX]; /* solution vector */

    backtrack(a, 0, n);
}
```

{1, 2, 3} 各个子集的生成顺序是怎样的？答案的关键在于 `construct_candidates` 中生成的候选集中各元素的顺序。由于 `true` 总在 `false` 之前，“全为真” 对应的全集最先生成，“全为假” 对应的空集最后生成：

```
{ 1 2 3 }
{ 1 2 }
{ 1 3 }
{ 1 }
{ 2 3 }
{ 2 }
{ 3 }
{ }
```

### 8.3 构造所有排列

构造所有排列和构造子集类似，唯一的区别是候选集取决于部分解中的已有值。为了避免产生重复元素，我们需要确保排列的第  $i$  个元素与之前的所有元素均不相同。

用回溯框架中的记号来说， $S_k = \{1, \dots, n\} - a$ ， $a$  为解的充要条件是  $k = n$ ：

```
construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    int i; /* counter */
    bool in_perm[NMAX]; /* who is in the permutation? */

    for (i=1; i<NMAX; i++) in_perm[i] = FALSE;
    for (i=1; i<k; i++) in_perm[a[i]] = TRUE;

    *ncandidates = 0;
```

```
    for (i=1; i<=n; i++)  
        if (in_perm[i] == FALSE) {  
            c[*ncandidates] = i;  
            *ncandidates = *ncandidates + 1;  
        }  
    }
```

为了测试  $i$  是否为排列第  $k$  个元素的候选值，我们可以依次比较排列的前  $k - 1$  个元素是否与  $a$  相同，但更好的做法是用一个位向量（参见第 2 章）来维护部分解中已有的元素集合。这样，判断一个值是否在候选集中只需常数时间<sup>2</sup>。

剩下的工作就是完成 `process_solution` 和 `is_a_solution` 两个函数以及编写初始调用。这些和“构造子集”中的相应部分并无本质区别：

```
process_solution(int a[], int k)  
{  
    int i; /* counter */  
  
    for (i=1; i<=k; i++) printf("%d", a[i]);  
  
    printf("\n");  
}  
  
is_a_solution(int a[], int k, int n)  
{  
    return (k == n);  
}  
  
generate_permutations(int n)  
{  
    int a[NMAX]; /* solution vector */  
  
    backtrack(a, 0, n);  
}
```

注意上述代码按字典序 (*lexicographic*) 给出各个排列，即：123, 132, 213, 231, 312，以及 321.

<sup>2</sup> 译者注：更常见的做法是动态维护这个位向量，而不是每次重新生成。

## 8.4 程序设计举例：八皇后问题

八皇后问题是一个经典的谜题，它的目标是在  $8 \times 8$  棋盘上放置八个皇后，使得任意两个皇后不会相互攻击。换句话说，任意两个皇后不会在同一行、同一列或者同一条  $45^\circ$  斜线上，参见图 8.1。不计其数的人们曾研究过这个问题，其中既包括高斯这样伟大的数学家，也包括只上过基础程序设计课程的学生。

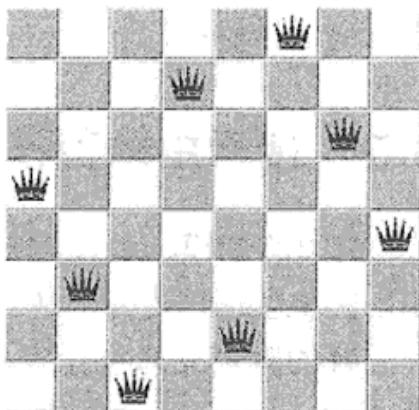


图 8.1 八皇后问题的一个解

没有什么可以阻挡我们考虑比 8 大的棋盘规模。 $n$  皇后问题要求统计出有多少种不同的方法可以在  $n \times n$  棋盘上放置  $n$  个互不攻击的皇后。即使是对很小的  $n$ ，解的总数已经大到无法把它们全部打印出来了。但如果只需计数，可以做到多大的  $n$  呢？

---

### 以下为题解

---

解决这类问题的时候，通常需要手算一些小规模的情况以获取灵感。很快你就会发现， $n = 2$  时无解，因为第二个皇后无论如何都会攻击第一个——要么在同一行，要么在同一列，要么在同一斜线。 $n = 3$  时的回旋余地要大一些了，但是尝试一段时间后你会发现还是无解。我们建议你手算出  $n = 4$  的一个解，这是有解的最小  $n$  了。

在实现 `backtrack` 框架之前，我们需要仔细寻找一个最简洁、高效的方式把解表示成向量。应如何合理地表示  $n$  皇后问题的解？这个解有多大？

最容易想到的表示法是利用前面提到的子集生成器，其中  $a_i$  为真当且仅当第  $i$  个方格里有一个皇后。这需要我们给每个格子赋予 1 到  $n^2$  之间的唯一编号。`false` 总是在当前格子的候选集中，而 `true` 在候选集中当且仅当这之前的皇后均不攻击这个格子。当所有  $n^2$  个格子被赋值之后还需要检查有多少格子为 `true`。只有当恰好  $n$  个格子为 `true` 时才是一个真正的合法解。

这是一个好的表示法吗？看起来它不是很简洁，因为在合法解中几乎所有的元素都是

*false*。换句话说，这种表示法的代价太高。粗略的算，对于  $8 \times 8$  棋盘，约有  $2^{64} \approx 1.84 \times 10^{19}$  个不同的布尔向量。尽管不是所有向量都被最终构造出来，但这仍然是一个很吓人的数字。

如果用解向量的第  $i$  个元素显式的给出第  $i$  个皇后的格子编号呢？在这种表示法下， $a_i$  是一个 1 到  $n^2$  之间的整数，当且仅当所有  $n$  个位置被赋值时  $a$  对应一个合法解。第  $i$  个位置的候选集为前  $i - 1$  个皇后没有攻击到的所有格子。

这个表示法和第一种相比是否更好？对于  $8 \times 8$  棋盘，它“只有”  $64^8 \approx 2.81 \times 10^{14}$  种可能的解向量。这是一个很大的改进，但是离通常情况下所能承受的最大搜索空间大小——百万结点（数量级约为  $10^6$ ），仍然相距甚远。要想提高回溯法的效率，我们需要剪枝（prune），即在构造出非法解之前尽可能早的把它们排除掉。

## 8.5 搜索中的剪枝

组合爆炸告诉我们：大多数搜索空间的规模随着问题规模呈指数增长。因此，即使是很小规模的问题，也常常无法在我们所能容忍的时间内求解。为了提高回溯法的效率，我们必须在发现一条搜索路径不会到达合法解的时刻立即终止它，即在搜索空间中剪枝。

剪枝（prune）这个术语相当形象。园丁会对他的树进行“剪枝”，即砍掉那些已经死掉或者长得奇形怪状的枝叶，使得树的营养和能量可以提供给最需要它们的地方。类似的，backtrack 的递归调用定义了一棵树。当发现某处的候选扩展集为空时可以对这棵树进行剪枝，从而避免让它不受控制的任意生长。

对于刚才描述的表示法，应如何进行剪枝？首先，可以去掉一些重复解。皇后没有编号，因此  $a_1 = x, a_2 = y$  和  $a_1 = y, a_2 = x$  是等价的。如果放任不管，每个解将以不同面目出现  $8! = 40,320$  次！这个问题很容易解决，只需确保  $a_i > a_{i-1}$  即可。这个简单的改进把搜索空间的大小缩小到了  $\binom{64}{8} = 4.426 \times 10^9$ 。

深入分析后不难得到一个更好的表示法。注意： $n$  皇后问题的合法解中，每行恰好有一个皇后。这是因为如果某一行没有皇后，则一定有一行至少有两个皇后（否则皇后总数不足  $n$ ），但同一行不能有两个皇后。这样，第  $i$  个皇后的位置只能在第  $i$  行的这 8 个格子中，搜索空间的大小进一步缩小到了  $8^8 \approx 1.677 \times 10^7$ 。尽管这个数字还不够小，但已经可以承受了。

事实上，我们还能做得更好！由于任意两个皇后所在的列都不相同，合法解中各个皇后所在的列一定是这  $n$  列的一个排列。这样一来，搜索空间的大小只剩  $8! = 40\,320$ ，对于任何一台普通机器来说已经微不足道了。

现在，我们终于可以编程实现这一算法了。最关键的函数是候选集生成器。我们依次判断当前行上的第  $k$  个格子是否被前面某个皇后攻击。如果被攻击，则判断下一个格子；如果没有被攻击，则加入到候选集中：

```

construct_candidates(int a[], int k, int n, int c[], int *ncandidates)
{
    int i,j;                      /* counters */
    bool legal_move;               /* might the move be legal? */

    *ncandidates = 0;
    for (i=1; i<=n; i++) {
        legal_move = TRUE;
        for (j=1; j<k; j++) {
            if (abs((k)-j) == abs(i-a[j])) /* diagonal threat */
                legal_move = FALSE;
            if (i == a[j])                 /* column threat */
                legal_move = FALSE;
        }
        if (legal_move == TRUE) {
            c[*ncandidates] = i;
            *ncandidates = *ncandidates + 1;
        }
    }
}

```

由于我们只需要计数，而不必打印解，剩下的函数都很简单了：

```

process_solution(int a[], int k)
{
    int i;                         /* counter */

    solution_count++;
}

is_a_solution(int a[], int k, int n)
{
    return (k == n);
}

nqueens(int n)

```

```
{  
    int a[NMAX]; /* solution vector */  
  
    solution_count = 0;  
    backtrack(a, 0, n);  
    printf("n=%d  solution_count=%d\n", n, solution_count);  
}
```

作者的笔记本电脑可以在瞬间内得到  $n = 9$  的解, 但当  $n = 10$  时, 由于大量的计算产生了足够的热量, CPU 风扇开始工作了。 $n = 14$  时, 计算需要好几分钟才能结束, 而且风扇的噪声实在烦人, 我们很快对更大的  $n$  失去了兴趣。下面是运行结果:

```
n=1  solution_count=1  
n=2  solution_count=0  
n=3  solution_count=0  
n=4  solution_count=2  
n=5  solution_count=10  
n=6  solution_count=4  
n=7  solution_count=40  
n=8  solution_count=92  
n=9  solution_count=352  
n=10  solution_count=724  
n=11  solution_count=2680  
n=12  solution_count=14200  
n=13  solution_count=73712  
n=14  solution_count=365596
```

更高效的程序当然可以算到更大一点的  $n$ 。在候选集生成器中, 一旦发现有皇后攻击当前格子, 循环即可跳出。通过更复杂的剪枝, 运行时间可以进一步缩短。在上述代码中, 当第  $k$  行没有候选元素时回溯, 但如果后面的行 (例如第  $k + 2$  行) 候选集为空, 在第  $k$  行上的任何工作都是徒劳的。越早发现这种情况越好<sup>3</sup>。

我们还可以利用对称加速。任何合法解旋转  $90^\circ$  或者关于棋盘中心求对称方案后都将得到合法解。如果细心设计解的生成算法, 我们可以在每个等价类中只生成一个解, 但统计时考虑该等价类中的所有解。这将大大减小搜索量。

不断优化搜索程序是一件有趣的事。为什么不现在就来求解  $n$  皇后问题, 看看 1 分钟

<sup>3</sup> 译者注: 但发现这种情况是有代价的。例如, 前向检查算法维护每一行的候选集 (而不仅仅是当前行), 当发现任一个候选集为空时回溯。实验比较发现, 由于维护候选集需要时间, 前向检查算法并不比普通回溯法快。

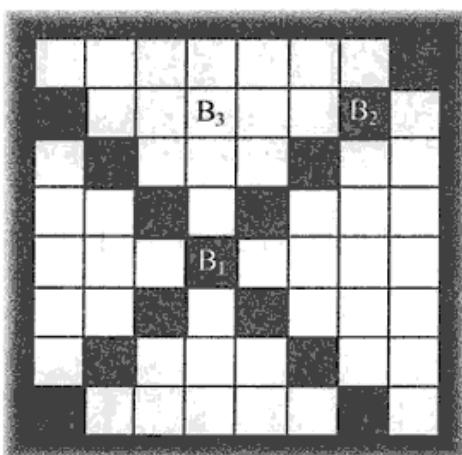
内能够算出多大的  $n$  呢？不过，不要期望比我们做到的好太多，因为对于我们所解决的问题规模，求解  $n+1$  所需的计算量比  $n$  要大十倍左右。因此，能够成功求解的  $n$  每增加一点点，都算得上是一个巨大的进步。

## 8.6 习 题

### 8.6.1 棋盘上的象 (Little Bishops)

PC/UVa 题号：110801/861，流行度：C，通过率：high 难度：2

国际象棋中的象在棋盘上总是沿对角线方向移动。若两个象相互处在对方可以到达的位置上，就可以互相攻击。如下图所示，黑色的格子表示象  $B_1$  能够到达的位置。象  $B_1$  和  $B_2$  间可以互相攻击，但  $B_1$  和  $B_3$  间不可以。 $B_2$  和  $B_3$  间同样不可以互相攻击。



现在给你两个整数  $n$  和  $k$ ，求出将  $k$  个象摆放在一个  $n \times n$  的棋盘上，并保证它们相互不能攻击的方案数。

#### 输入

输入包含多组数据。每组数据包含一行两个整数  $n(1 \leq n \leq 8)$  和  $k(0 \leq k \leq n^2)$ 。输入以一行两个 0 作为结束标记。

#### 输出

对于每组数据，输出一行一个整数，即在指定大小的棋盘上摆放象时互不攻击的方案总数。结果保证不超过  $10^{15}$ 。

#### 样例输入

8 6  
4 4  
0 0

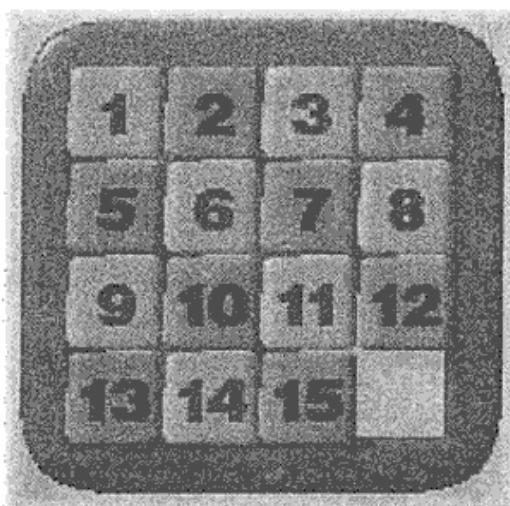
#### 样例输出

5599888  
260

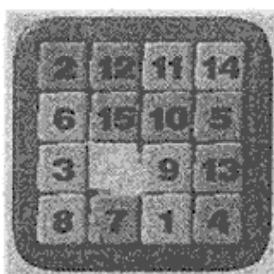
### 8.6.2 15 数码游戏 (15-Puzzle Problem)

PC/UVa 题号: 110802/10181, 流行度: B, 通过率: average 难度: 3

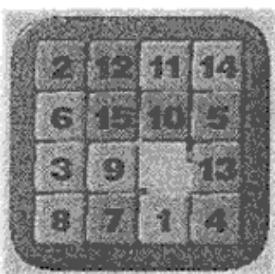
15 数码问题是一个很流行的游戏: 即便你没有听过这个名字, 你也一定见过。它由 15 个滑块构成, 每个滑块标有 1 到 15 之间的一个不同数码, 所有滑块排列在一个边长为 4 的正方形外框中, 留下一个空位。游戏的目标是移动滑块使他们排列成如下形式:



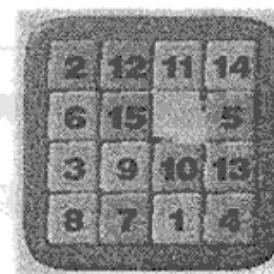
唯一符合规则的移动方式是将空位旁的 2, 3 或 4 个滑块中的一个移动到空位。考虑下面的一组移动:



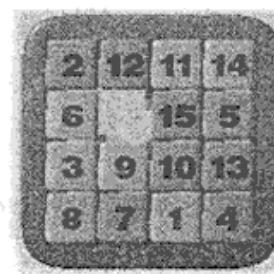
一个随机布局



空位右移 (R)



空位上移 (U)



空位左移 (L)

我们用与空位交换的邻位滑块来表示移动方式。可能的值为 “R”, “L”, “U” 和 “D”, 分别表示空位往右, 左, 上, 下移动。

给出一个 15 数码的初始布局, 找出一种移动方法使其转化为目标布局。输入数据中所有有解的问题均可以在不超过 45 步内解决, 你的解最多只能用 50 步。

#### 输入

输入数据第一行包含一个整数  $n$ , 代表数据组数。接下来  $4n$  行描述  $n$  个问题, 每 4 行描述一个。0 表示空位。

#### 输出

对于每组输入数据, 输出一行。如果给定的初始布局无解, 输出 “This puzzle is not solvable.”。如果有解, 输出一个可行序列来描述操作过程。

## 样例输入

2
2 3 4 0
1 5 7 8
9 6 10 12
13 14 11 15
13 1 2 4
5 0 3 7
9 6 10 12
15 8 11 14

## 样例输出

LLLD RDRDR

This puzzle is not solvable.

### 8.6.3 队伍 (Queue)

PC/UVa 题号: 110803/10128, 流行度: B, 通过率: high 难度: 2

有  $N$  个身高各异的人站成一排。如果某人比他左边所有的人都高，那么他可以往左看到整个队列以外，否则他的视线会被挡住；同样的，如果某人比他右边所有的人都高，那么他可以往右看到整个队列以外。

犯罪就在这里发生了。队列左边有一个人用回旋镖杀死了队列右边的一个人。此时恰有  $P$  人向左的视野没有被遮挡，同时恰有  $R$  人向右的视野没有被遮挡，这些人都可能是案件的目击证人。

辩护律师委托你算出这  $N$  个人有多少种可能的排列方法满足给定的  $P$  和  $R$ 。

输入

输入第一行包含一个整数  $T$  ( $1 \leq T \leq 10\,000$ ), 表示输入数据组数。

每组数据包含一行三个整数。第一个整数  $N$  表示队列中的人数 ( $1 \leq N \leq 13$ )。第二个整数对应左侧视野不受遮挡的人数  $P$ 。第三个数对应右侧视野不受遮挡的人数  $R$ 。

输出

对于每组数据，输出  $N$  人中  $P$  人左侧不受遮挡且  $R$  人右侧不受遮挡的排列种数。

### 样例输入

## 样例输出

3

90720

10 4 4

1026576

11 3



#### 8.6.4 服务站 (Servicing Stations)

PC/UVa 题号: 110804/10160, 流行度: B, 通过率: low 难度: 3

一家公司在  $N$  个小镇里销售个人电脑 ( $3 \leq N \leq 35$ )，这些小镇编号为  $1, 2, \dots, N$ 。有  $M$  条直达道路连接这些小镇。公司决定建一些服务站，保证对任意一个小镇  $X$ ，要么该镇自己有服务站，要么直接与之相邻的某个小镇内有服务站。

写一个程序计算该公司最少需要建多少个服务站。

输入

输入包含多组数据。每组数据以小镇数  $N$  以及道路数  $M$  开头，用空格隔开。接下来  $M$  行每行包含一个整数对，表示两个被直接连接的小镇编号，用空格隔开。输入数据以  $N = 0$  且  $M = 0$  作为结束标志。

输出

对于每组数据，输出一行，表示最小的服务站数量。

### 样例输入

8	12
1	2
1	6
1	8
2	3
2	6
3	4
3	5
4	5
4	7
5	6
6	7
6	8
0	0

## 样例输出

2

### 8.6.5 拔河 (Tug of War)

PC/UVa 题号: 110805/10032, 流行度: B, 通过率: low 难度: 2

拔河是一种主要靠体力对抗的游戏。人们分为两队，往相反的方向拉同一根绳子，成功将绳子拉到自己一方的队将获胜。

某公司的野餐会上将举行一次拔河比赛。他们想把参与者们尽可能分为实力相当的两支队伍。每个人都必须在其中一支队伍里，两队的人数差距不能超过一人，且两队的队员总体重应该尽量接近。

输入

输入以单独一个正整数开头，表示接下来的测试数据组数，然后是一个空行。

每组数据的第一行为一个整数  $n$ , 表示参加野餐会的人数。接下来的  $n$  行每行为一个人的重量, 用 1 到 450 之间的整数表示。参加野餐会的人数至多为 100。

相邻两组数据间用一个空行隔开。

输出

对于每组数据，输出单独一行，包含两个数，即两队各自的队员总体重。先输出总重较小的一个队的总体重。

相邻两组数据的输出之间用一个空行隔开。

## 样例输入

1

## 样例输出

190 200

3

100

90

200

#### 8.6.6 伊甸园 (Garden of Eden)

PC/UVa 题号: 110806/10001, 流行度: B, 通过率: average 难度: 2

细胞自动机是物理系统的一种理想化的数学模型。在这个模型中，空间和时间均是离散的，而其中的物理量均属于一个有限的离散值集合。细胞自动机由一组格子组成，每一格处于有限种状态中的一种。整个自动机的状态完全由每一格的状态决定。细胞自动机的演化过程是离散的，每一格的状态由它相邻格子前一时刻的状态决定。每个自动机都有它自己的一些规则来决定它的演化。

对于大部分细胞自动机，总有一些不可达到的状态：根据演化规则，没有任何一种状态可以演化成为这种状态。这些状态被称为伊甸园，因为它们只可能作为初始状态而存在。例如，考虑一个“把每个格子都变为 0”的平凡演化规则。对于这个自动机，任何一种包含非零格的状态都是伊甸园。

一般情况下，寻找给定状态的前驱（或者报告它没有前驱）是一个非常困难，需要很大计算量的问题。为了简单起见，我们只研究一维二元有限细胞自动机。换句话说，格子的数

目是有限的，且排成一圈，每格状态只可能是“0”或“1”。为了进一步简化问题，每个格子的状态仅与前一时刻自己和左右直接相邻格这三个状态有关。

注意：由于格子排成一圈，最后一个格子与第一个格子相邻。

### 问题描述

给定一个环状二元细胞自动机，你要确定一个给定状态是否是一个伊甸园。一个自动机用一组演化规则来描述。例如，下表描述了自动机  $Cell = XOR(Left, Right)$  对应的演化规则。

Left [i - 1]	Cell [i]	Right [i + 1]	New State	
0	0	0	0	$0 * 2^0$
0	0	1	1	$1 * 2^1$
0	1	0	0	$0 * 2^2$
0	1	1	1	$1 * 2^3$
1	0	0	1	$1 * 2^4$
1	0	1	0	$0 * 2^5$
1	1	0	1	$1 * 2^6$
1	1	1	0	$0 * 2^7$
				90 = Automaton Identifier

由于做了简化，本题只有 256 种不同的自动机。通过把新状态组成的向量按照表中的对应关系解释成一个二进制数，我们可以得到每个自动机的编码。上例中的自动机编码 90，而单元自动机（所有状态演化为其本身的自动机）编码为 204。

### 输入

输入包含多组数据，每组数据用单独的一行描述一个细胞自动机以及一个待判断的状态。其中第一项为待处理的自动机编码，第二项为一个正整数  $N (4 \leq N \leq 32)$ ，表示该自动机的格子数。最后一项为一个由 0 和 1 组成的长度为  $N$  的字符串。输入数据以文件结束符结尾。

### 输出

若输入数据所描述的状态为伊甸园，输出字符串 GARDEN OF EDEN，否则输出字符串 REACHABLE。每组数据的输出单独占一行。

#### 样例输入

```
0 4 1111
204 5 10101
255 6 000000
154 16 1000000000000000
```

#### 样例输出

```
GARDEN OF EDEN
REACHABLE
GARDEN OF EDEN
GARDEN OF EDEN
```

### 8.6.7 色彩缤纷游戏 (Color Hash)

PC/UVa 题号: 110807/704, 流行度: B, 通过率: average 难度: 3

“色彩缤纷”是一个由两个转轮组成的游戏。两个转轮都可以沿顺时针或逆时针方向转动。总共有 21 块彩色的滑块，其中 10 块是曲边三角形，另外 11 块则是分隔块。图 8.2 左边展示的是游戏的终态。注意一次旋转需要转过一个三角块和一个分割快。

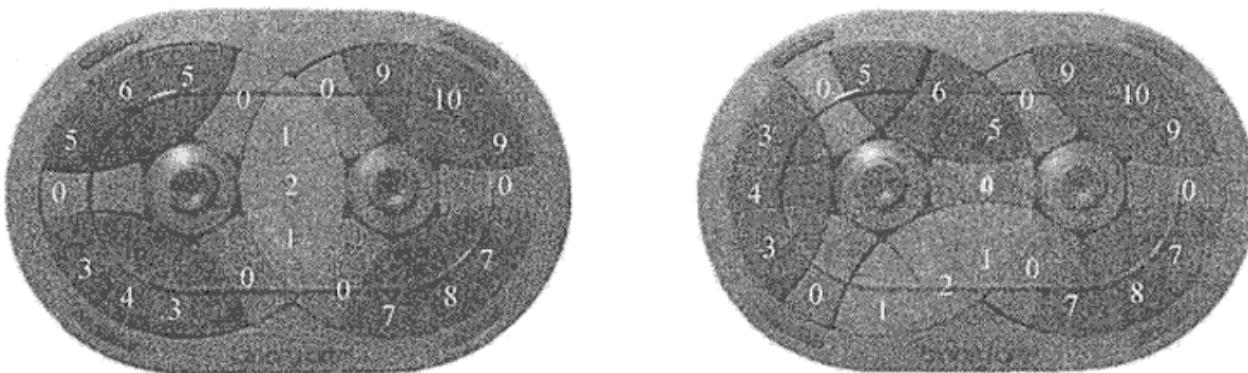


图 8.2 终态 (左), 在终态下将左侧转轮顺时针旋转一次 (右)

你的任务是写一个程序读入拼板的状态，输出最短的旋转方案将其变为终态。我们用下面的数值来代表不同的滑块：

- 0 灰色分隔块
- 1 黄色三角块
- 2 黄色分隔块
- 3 青色三角块
- 4 青色分隔块
- 5 紫色三角块
- 6 紫色分隔块
- 7 绿色三角块
- 8 绿色分隔块
- 9 红色三角块
- 10 红色分隔块

一个拼板的状态由 24 个整数来描述；前 12 个整数描述左侧转轮的状态，后 12 个整数描述右侧转轮的状态。第一个整数代表左侧转轮右下角的分隔块，接下来的 11 个整数顺时针描述整个左侧转轮。第 13 个整数描述右侧转轮左下角的分隔块，接下来 11 个整数逆时针描述整个右侧转轮。

这样，终态状态应该表示成：

0 3 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1

而如果将终态的左侧转轮顺时针旋转一次 (如上面的右图)，拼板就变成了：

2 1 0 3 4 3 0 5 6 5 0 1 0 7 8 7 0 9 10 9 0 5 0 1

### 输入

输入包含多个拼板。第一行为一个整数  $n$ , 表示拼板的个数。接下来的  $n$  行, 每行包含 24 个由空格分隔的整数, 描述一个拼板的初始状态。

### 输出

对于每个拼板初态, 输出一行, 包含一个整数, 即旋转方法的编码。每种旋转由 1 到 4 表示:

- 1 左侧转轮顺时针旋转
- 2 右侧转轮顺时针旋转
- 3 左侧转轮逆时针旋转
- 4 右侧转轮逆时针旋转

数字之间不应有空格。由于可能出现多解的情况, 你输出的解应该让方案编码最小。解法不超过 16 步。

如果找不到可行解, 输出 “NO SOLUTION WAS FOUND IN 16 STEPS”。如果输入状态已经是终态, 你应输出 “PUZZLE ALREADY SOLVED”。

### 样例输入

```
3
0 3 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1
0 3 4 5 0 3 6 5 0 1 2 1 0 7 8 7 0 9 10 9 0 1 2 1
0 9 4 3 0 5 6 5 0 1 2 1 0 7 8 7 0 9 10 3 0 1 2 1
```

### 样例输出

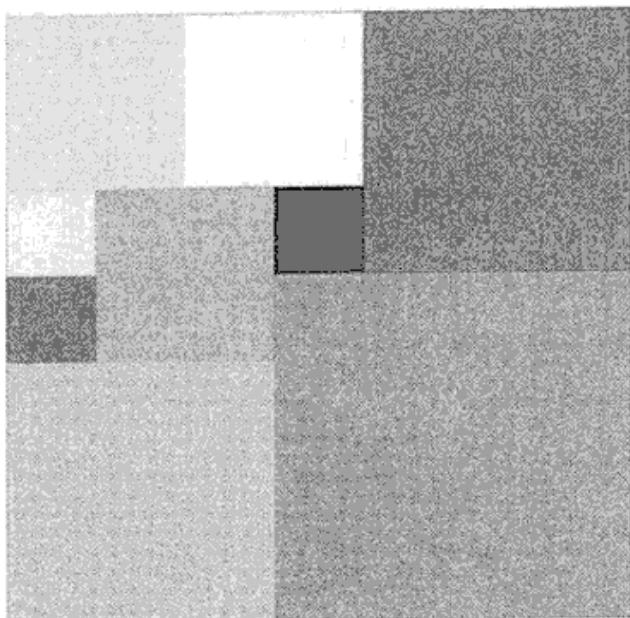
```
PUZZLE ALREADY SOLVED
1434332334332323
NO SOLUTION WAS FOUND IN 16 STEPS
```

### 8.6.8 拼接正方形 (Bigger Square Please...)

PC/UVa 题号: 110808/10270, 流行度: C, 通过率: high 难度: 3

Tomy 有很多正方形纸片。这些纸片的边长为 1 到  $N - 1$  不等, 且每种他都有无数张。但是他并不满足。他想要一张更大的 —— 边长为  $N$  的纸片。

可以把已有纸片拼接成他想要的大正方形。例如, 一个边长为 7 的正方形可以通过如下 9 个更小的正方形拼接而成:



在拼接出的正方形中间不能有空隙，不能有纸片超出正方形，且纸片不能相互重叠。并且，Tomy 想要用尽可能少的纸片来拼出这个大的正方形。你能帮助他吗？

### 输入

输入第一行有一个单独的整数  $T$ ，表示测试数据的组数。每组数据为一个单独的整数  $N(2 \leq N \leq 50)$ 。

### 输出

对于每组数据，输出一行，包含一个整数  $K$ ，表示最少需要的纸片数。接下来  $K$  行，每行三个整数  $x, y, l$ ，表示纸片左上角的坐标 ( $1 \leq x, y \leq N$ ) 以及纸片的边长。

#### 样例输入

3  
4  
3  
7

#### 样例输出

4  
1 1 2  
1 3 2  
3 1 2  
3 3 2  
6  
1 1 2  
1 3 1  
2 3 1  
3 1 1  
3 2 1  
3 3 1  
9

1 1 2  
1 3 2  
3 1 1  
4 1 1  
3 2 2  
5 1 3  
4 4 4  
1 5 3  
3 4 1

## 8.7 提 示

- 8.1 如何修改八皇后问题的解法来求解本题？把白象和黑象分开有用吗？
- 8.2 搜索过程中的重复状态既降低了搜索效率，又可能会增加解的长度。如何能够避免它们？
- 8.3 为了高效的搜索，应如何表示解？最好是把完整的排列构造出来，还是说可以只关心视野不被遮挡的人有哪些，而不必构造排列？
- 8.5 题目的规模即使对于加入了大量精巧剪枝的回溯法来说也许还是太大了。能否算出前  $i$  个人的各种子集所能得到的总重量，而无需显式的枚举所有  $2^i$  个子集？注意这些人能得到的不同总重量的个数远小于  $2^i$ 。
- 8.7 应如何表示解的状态？
- 8.8 优先尝试放置大正方形是否会比较好？

## 8.8 注 解

- 8.1 本题存在一个不错的组合方法，而不需要穷举。这使得对于更大棋盘的计数成为可能。UVa 评测系统中的 10237 题扩大了题目规模，使得组合方法变得必要了。
- 8.4 若图  $G$  的结点集  $S$  满足：每个结点要么属于  $S$ ，要么存在一个相邻结点属于  $S$ ，则称  $S$  为  $G$  的支配集 (*dominating set*)。寻找最小支配集的问题是 NP- 完全的。若必须找到最优解，穷举是不可避免的。
- 8.6 细胞自动机被用来作为很多不同自然现象的模型。推荐有兴趣的读者阅读 Wolfram 的那本有争议的书《*A New Kind of Science*》[Wol02]，来了解人们在这一问题上的不同看法。
- 8.7 作为“色彩缤纷游戏”的创造者，Binary Arts 公司还设计了很多不错的其他组合谜题。该公司的网站是 [www.puzzles.com](http://www.puzzles.com)。去看看吧。

## 第9章 图 遍 历

图 (graph) 是计算机科学的一个重要主题 —— 它为运输系统、电路、人际交互、电信网络的组织结构提供了一种抽象描述。很多不同的结构都可以用统一的形式建模，而这种能力也是受过良好教育的程序员的威力所在。

在本章中，我们将着重介绍只涉及基础图算法知识的题目。这在本章中特指合理运用图数据结构和遍历算法所能解决的那些问题。在第 10 章中，我们将看到一些涉及高级图算法的问题，如最小生成树、最短路和网络流。

### 9.1 图的不同属性

图  $G = (V, E)$  包含一个顶点 (*vertex*) 集  $V$  和一个边 (*edge*) 集  $E$ ，其中每条边是  $V$  中两个点的有序对或无序对。在给公路网建模时，顶点可以表示城市或者岔路口，而边表示两顶点之间的公路。在分析计算机程序的源代码时，顶点可以表示一个代码行，从行  $x$  连一条边到  $y$  表示执行完行  $x$  后，可能会接着执行行  $y$ 。在分析人际交互时，顶点往往表示人，边表示人与人之间的相互关联。

图有一些基本属性，它们将影响到可用来表示该图的数据结构和适用的算法。解决任何图问题的第一步就是弄清楚你将处理的图具有哪些属性：

- 无向图和有向图 —— 如果边  $(x, y) \in E$  总是可以推出  $(y, x)$  也在  $E$  中，我们称图  $G = (V, E)$  是无向图 (*undirected graph*)。否则，我们称该图是有向图 (*directed graph*)。城市之间的公路网络一般是无向的，因为宽阔的道路上一般同时有去往两个方向的车道。城市内部的街道网络一般是有向的，因为至少有一些小巷或者单行道只允许一个方向通过。程序流图一般是有向的，因为程序默认的行为是执行完一行后执行下一行，而不是反过来。
- 加权图和无权图 —— 在加权图中， $G$  中的每条边（或顶点）都被赋予了一个数值（或称权）。与具体应用相关，公路网上的典型权值包括距离、行驶时间、最大客运量等。在无权图中，点和边没有权上的区别。

当寻找两个结点之间的最短路时，加权图和无权图之间的区别就很明显了。在无权图中，最短路一定包含最少数目的边，且可以用本章后面将讨论的宽度优先搜索找到。加权图上的最短路需要更复杂的算法，将在第 10 章中讨论。

- 有环图和无环图 —— 无环图 不包含任意环。树是连通无向图。树是我们感兴趣的图中最简单的一种。它天生是一种递归结构，因为任意地删除一条边就会把树拆分成几棵更

小的树。

有向无环图一般缩写为 DAG，它们常常在调度问题中出现，其中有向边  $(x, y)$  表示  $x$  必须在  $y$  之前出现。有一个称为拓扑排序 (*topological sorting*) 的操作把一个 DAG 的所有结点根据优先级约束排序。拓扑排序是很多 DAG 算法的第一步，我们将在 9.5 节中讨论它。

- 简单图和非简单图 — 有一些特殊类型的边会让事情变得麻烦起来。自环 (*self-loop*) 是一个结点  $x$  到自己的边  $(x, x)$ 。如果一条边  $(x, y)$  在图中出现了多次，我们就说它是重边 (*multi-edge*)。

在实现图算法时需要小心处理这两种边。我们把既没有自环也没有重边的图称为简单图 (*simple graph*)。

- 嵌入图和拓扑图 — 嵌入图 (*embedded*) 是指点和边都被赋予几何位置的图。因此，把图画出来以后得到的就是该图的一个嵌入，这个嵌入方式对于算法来说可能很重要，也可能没有太大意义。

有时，图的拓扑结构可以由嵌入图的几何属性完全定义。例如，给定平面上的一个点集，要求一条访问所有点的最短回路（即旅行商问题 (*traveling salesman problem*)），则隐含的拓扑图是连接每对结点后得到的完全图，该图上的边权一般被定义为边的两个端点之间的欧几里德距离。

另一个从几何得到拓扑的例子来源于网格。 $n \times m$  网格上的很多问题都涉及相邻点之间的行走，因此该图的边已经在几何中隐式定义了。

- 隐式图和显式图 — 很多图都不是先被完整地构造出来后再遍历，而是在使用的时候逐步扩展。一个典型的例子是回溯法对应的隐式图。这个图的结点是状态向量，而有向边连接一个状态和由它生成的另一个状态。一般来讲，直接处理隐式图比显式地把它构造完整再分析要容易一些。
- 有标号图和无标号图 — 在有标号图 (*labeled graph*) 中，每个顶点被赋予一个唯一的名称（或称标识符），把它和其他顶点区分开。在无标号图 (*unlabeled graph*) 中，所有顶点都是相同的。

在实际应用中，大多数图的顶点很自然地被标了号，比如运输网络中的城市名。图中的一个常见问题是同构判定 (*isomorphism testing*)，即在忽略标号的情况下判断两个图的拓扑结构是否完全相等。这类问题一般需要由回溯法解决，即尝试两个图中的所有标号方案，判断得到的有标号图是否相同。

## 9.2 图的数据结构

可以用多种方法来表示图。下面，我们介绍 4 种有用的表示法。假设图  $G = (V, E)$  有  $n$  个顶点和  $m$  条边。

- 邻接矩阵 — 可以用  $n \times n$  矩阵  $M$  来表示图  $G$ , 其中元素  $M[i, j]$  为 1 当且仅当  $(i, j)$  是图  $G$  中的一条边。这样的表示法能快速回答询问 “ $(i, j)$  在图  $G$  中吗?”, 并支持边的快速插入和删除。但是, 当结点数很大而边数相对较少时, 这种表示法会占用过多不必要的空间。

考虑纽约曼哈顿的街道图。两条街道的交叉口是该图的一个结点，相邻交叉口之间用边相连。这张图有多大呢？曼哈顿由 15 条大道 (*avenue*) 组成，每条大道与大约 200 条街 (*street*) 交叉，因此一共约 3 000 个结点和 6 000 条边（每个结点有四个相邻结点，且每条边同时属于两个结点）。这样小的数据量可以很方便、有效地储存下来，但是邻接矩阵将有  $3\,000 \times 3\,000 = 9\,000\,000$  个位置，几乎所有位置都是空的！

- 邻接表（链表实现）— 为了有效地储存稀疏图，我们为每个结点开一个链表，保存它的所有相邻点。邻接表需要用到指针，不过当你熟悉链式结构以后就不成问题了。

在邻接表中，询问一条特定边  $(i, j)$  是否在图  $G$  中就不那么容易了（我们必须在相应链表中一项一项地查找），但好在大部分图算法都用不到这样的询问，而是用宽度优先遍历或深度优先遍历的框架对所有边进行一遍扫描，当访问一条边时对它进行处理，所以邻接表仍然是很实用的。

- 邻接表（矩阵实现）— 邻接表也可以用矩阵实现，从而避免了烦人的指针。我们用一个数组（实际上是矩阵中的一行）来储存一个结点的邻接点列表。除此之外，还需要一个计数器  $k$  来表明只有该数组的前  $k$  个元素是有用的。这样，我们仍然可以枚举从一个点出发的所有相邻点，但这次只需枚举数组下标，而不用任何指针操作。

表面上，这种表示法好像综合了邻接矩阵的弱点（占用空间大）和邻接表的弱点（查询边需要在表中检索），但实际上它自有妙用。首先，它是最容易编程实现的数据结构，特别是对于“一旦建立，永不改变”的静态图。其次，空间问题也可以通过为每行动态申请空间的方式来缓解<sup>1</sup>。

为了证明这种表示法的适用性，接下来的所有算法都将用它来实现。

- 边列表 — 还有一种更简单的数据结构：只维护所有边组成的数组或者链表。这种方法不像上述其他结构那样灵活，甚至连处理“ $x$  的相邻点有哪些”这样的询问都无法高效回答，但对于 Kruskal 算法这样的简单过程来说已经足够。

刚才说过，我们用矩阵来实现邻接表，并作为表示图的基本数据结构。不难把这些代码转化成使用链表的邻接表。邻接表和邻接矩阵的参考代码可以在很多书中找到，例如 [Sed01]。

我们用如下数据类型来表示一个图。图的结点数保存在 `nvertices` 中，每个结点被赋予一个 1 到 `nvertices` 之间的唯一整数。边用  $\text{MAXV} \times \text{MAXDEGREE}$  的二维数组表示，因此每个结点最多只能和 `MAXDEGREE` 个结点相邻。如果要表示任意简单图，应当把 `MAXDEGREE` 定义成 `MAXV`，但如果已知最大度数比较小，就没有这个必要了：

<sup>1</sup> 译者注：这可以用 STL 中的 vector 方便地实现。

```
#define MAXV          100           /* maximum number of vertices */
#define MAXDEGREE      50            /* maximum vertex outdegree */

typedef struct {
    int edges[MAXV+1][MAXDEGREE];   /* adjacency info */
    int degree[MAXV+1];             /* outdegree of each vertex */
    int nvertices;                 /* number of vertices in graph */
    int nedges;                    /* number of edges in graph */
} graph;
```

有向边  $(x,y)$  被表示成  $x$  的邻接表  $\text{graph-} \rightarrow \text{edges}[x]$  中的一个整数  $y$ 。 $\text{degree}$  数组储存每个结点的出度，它也是  $\text{graph-} \rightarrow \text{edges}[x]$  的有效表项数。无向边  $(x,y)$  在邻接表中出现两次，一次是  $x$  表中的  $y$ ，一次是  $y$  表中的  $x$ 。

为了展示这种数据结构的用法，下面给出了从文件读取一张图的方法。储存图的典型文件格式如下：第一行包含两个整数，即结点数和边数。以下每行描述一条边，即边两端的两个结点编号。

```
read_graph(graph *g, bool directed)
{
    int i;                      /* counter */
    int m;                      /* number of edges */
    int x, y;                   /* vertices in edge (x,y) */

    initialize_graph(g);

    scanf("%d %d", &(g->nvertices), &m);

    for (i=1; i<=m; i++) {
        scanf("%d %d", &x, &y);
        insert_edge(g, x, y, directed);
    }

initialize_graph(graph *g)
{
    int i;                      /* counter */
```

```

g -> nvertices = 0;
g -> nedges = 0;

for (i=1; i<=MAXV; i++) g->degree[i] = 0;
}

```

关键函数是 `insert_edge`。我们用布尔标志 `directed` 来表明是需要插入一条边还是两条，并用递归来解决了这一问题：

```

insert_edge(graph *g, int x, int y, bool directed)
{
    if (g->degree[x] > MAXDEGREE)
        printf("Warning: insertion(%d,%d) exceeds max degree\n",x,y);

    g->edges[x][g->degree[x]] = y;
    g->degree[x]++;

    if (directed == FALSE)
        insert_edge(g,y,x,TRUE);
    else
        g->nedges++;
}

```

输出整张图只需一个二重循环：

```

print_graph(graph *g)
{
    int i,j; /* counters */

    for (i=1; i<=g->nvertices; i++) {
        printf("%d: ",i);
        for (j=0; j<g->degree[i]; j++)
            printf(" %d",g->edges[i][j]);
        printf("\n");
    }
}

```

### 9.3 图的遍历：宽度优先

大多数图算法的基本操作是完整、系统的遍历图。我们希望用某种定义好的顺序访问每个结点和边各一次且仅一次。主要有两类遍历算法：宽度优先搜索 (*breadth-first search, BFS*) 和深度优先搜索 (*depth-first search, DFS*)。对于某些问题，遍历的具体方式无关紧要，但是在另一些问题中，不同遍历算法之间的差异是决定性的。

两种遍历有一个共同点：必须在访问一个结点后给它加上一个标记，使得它不会被重复访问。否则，我们的程序就像被困在迷宫中一样无法退出。BFS 和 DFS 的区别仅在于访问结点的顺序。

BFS 适用的场合是：(1) 访问结点和边的顺序无关紧要，或者 (2) 我们想找出无权图中的最短路。

#### 9.3.1 宽度优先遍历

宽度优先遍历代码 `bfs` 使用两个布尔数组来保存每个结点的信息。第一次访问一个结点时，它会被标记为 `discovered` (已发现)，当从它出发的所有出边都已处理完毕后，把它标记为 `processed` (已处理)。因此在遍历中，每个结点都会经历一个从“未发现”到“处理中”，再到“已处理”的过程。这个信息也可以用一个枚举类型来表示，这里我们使用了两个布尔变量。

当一个结点被发现时，它将被放到一个队列中（比如，我们在 2.1.2 小节中实现的队列）。由于我们按照先进先出的方式处理结点，越早发现的结点将越早被处理，也就越接近宽度优先搜索树的树根：

```
bool processed[MAXV]; /* which vertices have been processed */
bool discovered[MAXV]; /* which vertices have been found */
int parent[MAXV]; /* discovery relation */

bfs(graph *g, int start)
{
    queue q; /* queue of vertices to visit */
    int v; /* current vertex */
    int i; /* counter */

    init_queue(&q);
    enqueue(&q, start);
    discovered[start] = TRUE;
```

```

while (empty(&q) == FALSE) {
    v = dequeue(&q);
    process_vertex(v);
    processed[v] = TRUE;
    for (i=0; i<g->degree[v]; i++)
        if (valid_edge(g->edges[v][i]) == TRUE) {
            if (discovered[g->edges[v][i]] == FALSE) {
                enqueue(&q,g->edges[v][i]);
                discovered[g->edges[v][i]] = TRUE;
                parent[g->edges[v][i]] = v;
            }
            if (processed[g->edges[v][i]] == FALSE)
                process_edge(v,g->edges[v][i]);
        }
    }
}

initialize_search(graph *g)
{
    int i; /* counter */

    for (i=1; i<=g->nvertices; i++) {
        processed[i] = discovered[i] = FALSE;
        parent[i] = -1;
    }
}

```

### 9.3.2 遍历的应用

代码 bfs 的具体功能取决于函数 `process_vertex()` 和 `process_edge()`。通过修改这两个函数，我们可以按照自己的意愿来定制宽度优先搜索。例如，以下实现：

```

process_vertex(int v)
{
    printf("processed vertex %d\n",v);
}

```

```
process_edge(int x, int y) {
    printf("processed edge (%d,%d)\n",x,y);
}
```

会输出每个结点和每条边各一次；而以下实现：

```
process_vertex(int v)
{
}

process_edge(int x, int y)
{
    nedges = nedges + 1;
}
```

将统计出边的数目。不同的算法对应于结点和边的不同处理方式，这两个函数把这个灵活性从其他代码中分离出来，让我们可以更加方便地定制遍历代码。

最后一个可以定制的地方是布尔函数 `valid_edge`，它可以帮助我们忽略图中一些特定的边。在大多数情况下，函数 `valid_edge` 应该对所有边返回 `true`。在这种情况下，宽度优先搜索将对所有边进行。唯一的例外是 `netflow`，我们将在 10.4 节中讨论它。

### 9.3.3 寻找路径

`bfs()` 中设置的 `parent` 数组对寻找路径很有帮助。发现结点  $i$  的父结点就是 `parent[i]`。每个结点都是在遍历中被发现的，因此除了遍历的起点外，每个结点都有一个父结点。父子关系定义了一个树结构，其中遍历的起点是整棵树的根。

由于结点按照根到它们的距离从小到大的顺序被发现，这棵树有一个重要性质。对于任意结点  $x \in V$ ，沿着树上的边从根走到  $x$  所经过的边数（或等价地，中间结点数）是原图中所有根到  $x$  的路径中最小的。

如何构造出从根到  $x$  的最短路？如果从根出发，我们就很难知道应该顺着哪条边走才能到达  $x$ ，但如果从  $x$  出发，只需顺着父结点组成的链不断往回走，走到根的同时也就可以构造出这条最短路。

由于我们需要的路是从根到  $x$ ，和构造最短路的方向相反，因此需要（1）保存整条路径，然后用栈来反转它，或者（2）用递归实现，借助递归中的隐式栈来完成反向功能：

```
find_path(int start, int end, int parents[])
{
    if ((start == end) || (end == -1))

```

```
printf("\n%d",start);
else {
    find_path(start,parents[end],parents);
    printf(" %d",end);
}
}
```

对于一个网格上的例子 (如图 9.1 所示) 来说, 以左下角为起点, 我们的算法生成了如下的父亲数组:

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

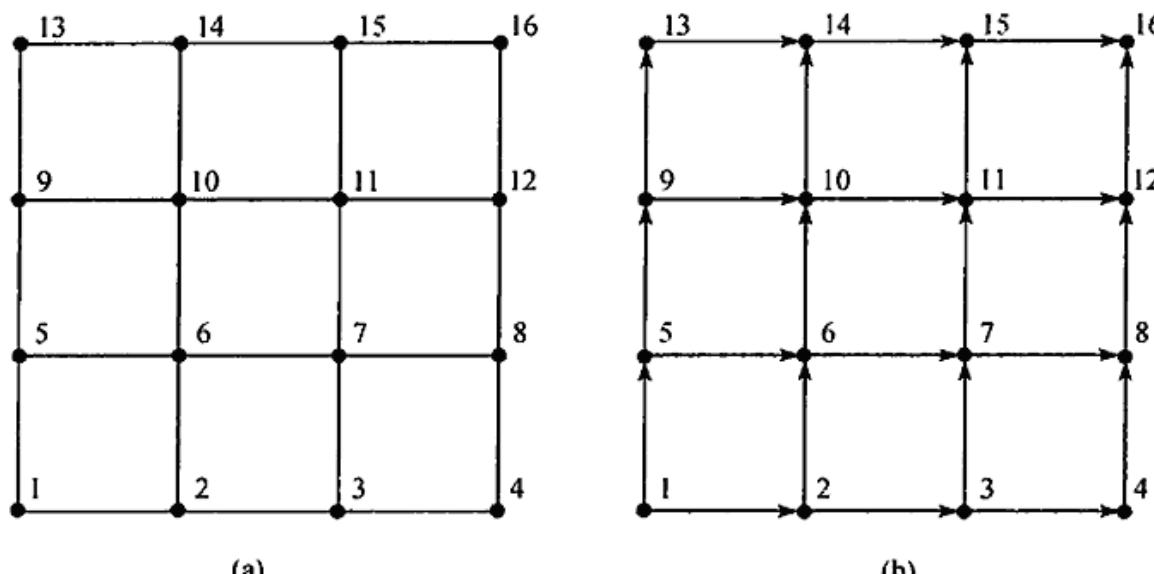


图 9.1 无向  $4 \times 4$  网格图 (图 (a)), 按照“从小指向大”的方式给边定向后得到的 DAG (图 (b))

对于从左下角到右上角的最短路, 可以从父亲数组得到如下的路径:  $\{1, 2, 3, 4, 8, 12, 16\}$ 。当然, 最短路并不是唯一的。事实上, 这个图中从左下角到右上角的最短路径条数已经在 6.3 节中讨论过了。

用宽度优先遍历寻找  $x$  到  $y$  的最短路径的算法有两点需要注意: 首先, 只有从  $x$  开始遍历, 得到的最短路径才是有用的。其次, BFS 只有在无权图中才能得到最短路。对于加权图, 我们将在 10.3.1 小节中介绍最短路算法。

## 9.4 图的遍历：深度优先

深度优先算法的思想和回溯法本质相同。两个算法都是尽量往前走, 实在无法继续前进时才后退, 用这种方式来穷举所有可能性。理解这两个算法的最简单途径都是递归。

深度优先遍历可以看作是把宽度优先遍历中的队列替换成栈后的产物。代码 `dfs` 的优美之处在于用递归消除了显式使用栈的必要：

```
dfs(graph *g, int v)
{
    int i;                      /* counter */
    int y;                      /* successor vertex */

    if (finished) return;        /* allow for search termination */

    discovered[v] = TRUE;
    process_vertex(v);

    for (i=0; i<g->degree[v]; i++) {
        y = g->edges[v][i];
        if (valid_edge(g->edges[v][i]) == TRUE) {
            if (discovered[y] == FALSE) {
                parent[y] = v;
                dfs(g,y);
            } else
                if (processed[y] == FALSE)
                    process_edge(v,y);
        }
        if (finished) return;
    }

    processed[v] = TRUE;
}
```

有根树是一种特殊的图（有向无环，入度最多为 1，每个顶点的各出边定义好了顺序<sup>2</sup>）。中序、先序和后序遍历在本质上都属于 DFS，它们的区别仅是在处理每个结点时按照何种顺序访问出边。

#### 9.4.1 寻找环

无向图的深度优先遍历把所有边分为两类，树边 (*tree edges*) 和反向边 (*back edges*)。父

<sup>2</sup> 译者注：这里特指有序的有根树。在无序有根树中，每个结点的各子树之间并无顺序之分。

子关系所涉及的边都是树边，发现这些边的同时也发现了新结点。反向边连接了一个结点和它在 DFS 树上的祖先。

每条边必定属于这两类边之一，这是 DFS 的一个奇妙性质。为什么一条边不能连接同一棵树上的兄弟结点或者一个结点和它的非直接后代结点呢？在 DFS 中，从给定结点  $v$  可达的所有结点都会在  $v$  处理完毕之前被发现，因此这样的情况在无向图中不可能出现。有向图上的 DFS 要复杂一些，但仍然拥有良好的结构。

反向边是在无向图中找环的关键。如果没有反向边，则所有边都是树边，而树中不存在环。另一方面，任何一条从  $x$  到它祖先  $y$  的反向边对应了一个环（从  $x$  沿着反向边走到  $y$ ，然后沿着树边回到  $x$ ）。这样的环可以用下面的代码 `dfs` 得到：

```
process_edge(int x, int y)
{
    if (parent[x] != y) { /* found back edge! */
        printf("Cycle from %d to %d:", y, x);
        find_path(y, x, parent);
        finished = TRUE;
    }
}

process_vertex(int v)
{
```

我们用一个标志 `finished` 让程序找到第一个环后立即退出。对于我们的  $4 \times 4$  网格图，第一个环是 3 4 8 7，其中 (7,3) 是那条反向边。

#### 9.4.2 连通分量

无向图的连通分量 (*connected component*) 是满足“任意两个结点之间都有通路”的极大点集。这些连通分量在图中各成“一片”，不同的“片”之间没有任何边相连。

有无数看上去很复杂的问题都可以归结为寻找或者统计连通分量。例如，判断像魔方、15 数码问题这样的谜题从某一个初始解开始是否有解实际上是在问初始状态和目标状态是否在同一连通分量中<sup>3</sup>。

用深度优先搜索和宽度优先搜索都可以求出连通分量，因为在这个问题上，遍历顺序是不重要的。我们随便从一个结点开始遍历，则此次遍历时所发现的结点与开始结点同属一个连通分量。然后随便找一个未发现的结点（如果有的话）开始遍历，直到所有结点都被发现

<sup>3</sup> 译者注：尽管这样的归约意义不大，因为这些问题对应的隐式图太大。

为止：

```
connected_components(graph *g)
{
    int c;                      /* component number */
    int i;                       /* counter */

    initialize_search(g);

    c = 0;
    for (i=1; i<=g->nvertices; i++)
        if (discovered[i] == FALSE) {
            c = c+1;
            printf("Component %d:",c);
            dfs(g,i);
            printf("\n");
        }
}

process_vertex(int v)
{
    printf(" %d",v);
}

process_edge(int x, int y)
{}
```

连通分量的变种将在 10.1.2 小节中讨论。

## 9.5 拓 扑 排 序

拓扑排序是有向无环图 (DAG) 上的基本操作。它把所有结点排成一列，使得所有有向边在序列中都是从左指向右的。显然，包含有向环的图不存在这样的排列方法，因为不断地从左往右走不可能回到起点！

拓扑排序的重要之处在于，它给出了一个方法，使得可以在处理每个结点的后继之前处理该结点本身。假设边表示优先级约束，即  $(x,y)$  表示作业  $x$  必须在作业  $y$  之前完成，则

任何一个拓扑排序都对应一个合法的调度方案。事实上，一个 DAG 往往有很多不同的拓扑排序。

但典型的应用并不仅限于此。假设我们需要在 DAG 上找到从  $x$  到  $y$  最短（或最长）的路径。在拓扑排序中位于  $y$  之后的结点肯定不属于所求路径的一部分，因为一旦到达这些点，就永远回不了  $y$  了。我们可以按照拓扑序从左到右处理各个结点，考虑它们的出边，更新从  $x$  到这些出边的右端点的最短（或最长）路径长度。这样，当我们需要用到某一个最短（或最长）路径长度时，它已经更新为正确的值了。

可以用深度优先遍历来有效实现拓扑排序。不过，可以通过每个结点的入度分析来得到一个更直观的算法。如果某结点没有入边（即入度为 0），我们可以放心地把它放到拓扑排序的首部。删除它的所有出边后将得到一些新的入度为 0 的点，再如法炮制。重复这一过程，直到所有点都被放入拓扑序列中。如果在此之前无法找到入度为 0 的点，则表明此图中有环，不是 DAG。

细节参见下面的代码实现：

```
topsort(graph *g, int sorted[])
{
    int indegree[MAXV];           /* indegree of each vertex */
    queue zeroin;                /* vertices of indegree 0 */
    int x, y;                    /* current and next vertex */
    int i, j;                    /* counters */

    compute_indegrees(g,indegree);
    init_queue(&zeroin);
    for (i=1; i<=g->nvertices; i++)
        if (indegree[i] == 0) enqueue(&zeroin,i);

    j=0;
    while (empty(&zeroin) == FALSE) {
        j = j+1;
        x = dequeue(&zeroin);
        sorted[j] = x;
        for (i=0; i<g->degree[x]; i++) {
            y = g->edges[x][i];
            indegree[y]--;
            if (indegree[y] == 0) enqueue(&zeroin,y);
        }
    }
}
```

```
    }

    if (j != g->nvertices)
        printf("Not a DAG -- only %d vertices found\n", j);
}

compute_indegrees(graph *g, int in[])
{
    int i, j; /* counters */

    for (i=1; i<=g->nvertices; i++) in[i] = 0;

    for (i=1; i<=g->nvertices; i++)
        for (j=0; j<g->degree[i]; j++) in[g->edges[i][j]]++;
}
```

有几点是值得注意的。首先应该计算所有结点的入度，因为 `degree` 保存的是出度而不是入度。在无向图中，出度和入度相等，而在有向图中二者是不同的。

其次，尽管我们这里用了一个队列来维护所有入度为 0 的结点，但其实这只是因为我们已经在 2.1.2 小节中实现过队列了。事实上，任何容器都可以用在这里，因为各个结点的处理顺序并不重要。不同的结点处理顺序会得到不同的拓扑排序。

结点处理顺序的影响通过图 9.1 很容易看出。既然所有边都是从编号小的点连向编号大的点，升序 {1, 2, ..., 15, 16} 显然就是一个拓扑排序，但由于我们的程序不断删除对角线，它得到的将是

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

还有很多其他可能的拓扑排序。

最后，注意我们的程序并没有真正地从图中删除任何边！只需要考虑它们对每个点入度的影响，遍历而不是删除图中的所有边。

## 9.6 习题

### 9.6.1 双着色 (Bicoloring)

PC/UVa 题号: 110901/10004, 流行度: A, 通过率: high 难度: 1

四色定理 (*four-color theorem*) 说的是：对于任意一个平面地图，可以只用四种颜色为它着色，使得每个区域均被着色，且每两个相邻区域的颜色都不同。定理提出 100 多年后的

1976年，人们才成功地借助计算机证明了它。

现在请你来解决这个问题的一个简化版本：判断一个给定的平面地图是否可以只用红色和黑色着色，使得每两个相邻区域的颜色不同。假设地图总是连通的，无向的，且不包含自环（即从某个顶点到它自己的边）。

输入

输入包含若干组数据。每组数据的第一行有一个整数  $n(1 < n < 200)$ , 表示顶点的数量。每个顶点用  $0 \sim (n - 1)$  之间的整数表示。第二行为一个整数  $l$ , 表示边的数量。接下来的  $l$  行每行用两个顶点的序号来描述一条边。

当  $n = 0$  时，输入结束。你的程序不应处理这一行。

输出

判断输入的图是否可以双着色，并按照样例的格式输出结果。

### 样例输入

3  
3  
0 1  
1 2  
2 0  
9  
8  
0 1  
0 2  
0 3  
0 4  
0 5  
0 6  
0 7  
0 8  
0

## 样例输出

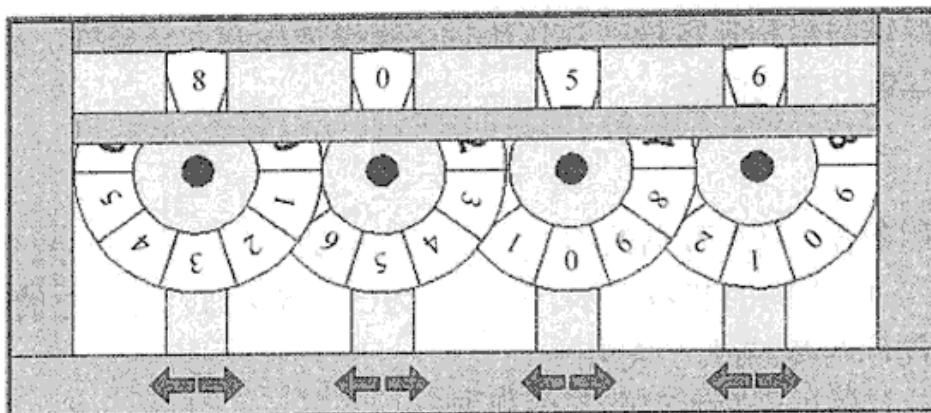
NOT BICOLORABLE.  
BICOLORABLE

### 9.6.2 摆弄轮子 (Playing With Wheels)

PC/UVa 题号: 110902/10067, 流行度: C, 通过率: average 难度: 2

下图是一个数学仪器：连续的数字 0 ~ 9 按顺时针顺序被印在每个齿轮的外围。每个齿轮的最高的数字一起构成了一个四位数。例如，下面的这些齿轮构成的整数为 8056。每个

齿轮有两个按钮：按左箭头时齿轮将按顺时针方向移动一个字母，按右箭头时则是沿相反的方向运动。



开始时，齿轮的顶端数字构成  $S_1S_2S_3S_4$ 。给你  $n$  个禁止的数  $F_{i_1}F_{i_2}F_{i_3}F_{i_4}$  ( $1 \leq i \leq n$ ) 和一个目标数  $T_1T_2T_3T_4$ ，你的任务是在中途不能得到禁止数的情况下，尽量少地按下按钮来得到目标数。

### 输入

输入第一行有一个整数  $N$ ，代表测试数据的数量。接下来有一个空行。

每组数据的第一行描述了这些齿轮的初始状态，用四个数字表示，其中每两个相邻数字用一个空格隔开。下面一行为目标数。第三行有一个整数  $n$ ，表示禁止数的个数。接下来的  $n$  行每行有一个禁止数。相邻两组输入数据之间有一个空行。

### 输出

对于每一组输入数据，输出一行，表示需要按下按钮的最少次数。如果无法完成任务，输出 “-1”。

#### 样例输入

```
2
8 0 5 6
6 5 0 8
5
8 0 5 7
8 0 4 7
5 5 0 8
7 5 0 8
6 4 0 8
0 0 0 0
```

#### 样例输出

```
14
-1
```

```

5 3 1 7
8
0 0 0 1
0 0 0 9
0 0 1 0
0 0 9 0
0 1 0 0
0 9 0 0
1 0 0 0
9 0 0 0

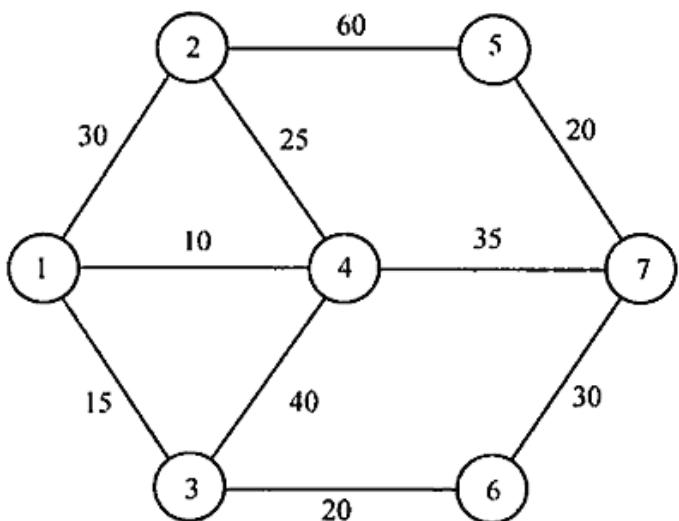
```

### 9.6.3 导游 (The Tourist Guide)

PC/UVa 题号: 110903/10099, 流行度: B, 通过率: average 难度: 3

Mr. G. 在孟加拉国的一家旅游公司工作。他当前的任务是带一些游客去一个遥远的城市。和所有国家一样，一些城市之间有双向道路。每对相邻城市之间都有一条公交线路，每条线路都规定了自己的最大乘客数目。Mr. G. 有一份包含城市间道路状况和公交车最大载客量的地图。

往往无法一次性地将所有乘客带往目的地。例如，在下面 7 个城市的地图中，边代表道路，每条边上的数字代表这条道路上公交车的最大载客量。



如果 Mr. G. 要将 99 位乘客从城市 1 带到城市 7，则至少要往返 5 次（他必须陪同每一群游客）。最佳路线是 1 - 2 - 4 - 7。

帮助 Mr. G. 找出将所有游客带到目的地，且往返次数最少的线路。

#### 输入

输入包含若干组数据。每组数据的第一行有两个整数  $N(N \leq 100)$  和  $R$ ，分别表示城市的数量和道路的数量。接下来的  $R$  行每行有 3 个整数  $(C_1, C_2, P)$ ，其中  $C_1$  和  $C_2$  为城市编

号,  $P(P > 1)$  是该道路上公交车的最大载客量。各城市编号为  $1 \sim N$  的连续整数。第  $R+1$  行包含 3 个整数  $(S, D, T)$ , 分别表示出发城市, 目的城市的编号和游客的数量。

$N = R = 0$  时输入结束。

### 输出

对于每组输入数据, 按照样例格式先输出一行数据编号, 接下来在单独的一行中输出最少的往返次数。

在每组数据的输出后打印一个空行。

### 样例输入

```
7 10
1 2 30
1 3 15
1 4 10
2 4 25
2 5 60
3 4 40
3 6 20
4 7 35
5 7 20
6 7 30
1 7 99
0 0
```

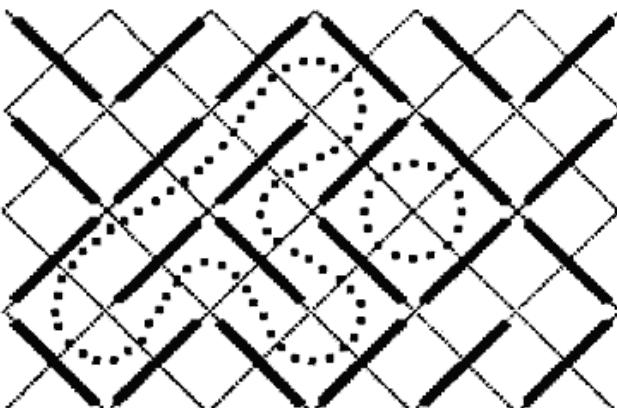
### 样例输出

```
Scenario #1
Minimum Number of Trips = 5
```

#### 9.6.4 斜线迷宫 (Slash Maze)

PC/UVa 题号: 110904/705, 流行度: B, 通过率: average 难度: 2

通过用斜线 (/) 和反斜线 (\) 来填充一个矩形, 你可以得到一个小迷宫, 例如:



正如你所看到的，迷宫中的通路从不分叉，因此整个迷宫中只有（1）环状通路以及（2）从一个地方进入并从另外一个地方出去的通路。我们只对环感兴趣。在上述例子中有两个环。

你的任务是要写一个程序来计算最长的环的长度。环的长度被定义为组成这个环的小正方形（在图中用灰色线条标记）的数量。在上述例子中，长环的长度为 16，短环的长度为 4。

### 输入

输入包含若干组数据。每组数据的第一行两个整数  $w$  和  $h$  ( $1 \leq w, h \leq 75$ )，分别代表迷宫的宽度和高度。接下来的  $h$  行，每行有  $w$  个字符，描述整个迷宫。这些字符都是 “/” 或 “\”。

当  $w = h = 0$  时输入结束。这一行不必处理。

### 输出

对于每个迷宫，先输出一行 “Maze # $n$ :”，其中  $n$  为迷宫序号。然后输出一行 “ $k$  Cycles; the longest has length  $l$ .”，其中  $k$  为迷宫中环的数量， $l$  是最长环的长度。如果这个迷宫中没有环，输出 “There are no cycles.”。

在每组数据的输出后打印一个空行。

### 样例输入

```
6 4
\\//\\/
\\\\//\\
//\\//\\
\\\\///\\
3 3
///
\\/
\\\
0 0
```

### 样例输出

```
Maze #1:
2 Cycles; the longest has length 16.

Maze #2:
There are no cycles.
```

### 9.6.5 递变阶梯 (Edit Step Ladders)

PC/UVa 题号: 110905/10029, 流行度: B, 通过率: low 难度: 3

递变 (edit step) 是指通过增加、减少或改变单词  $x$  中的一个字母，使它变成字典中的另一个单词  $y$ 。比如将 dig 变为 dog，将 dog 变成 do 都是递变。递变阶梯 (edit step ladder) 是一个按字典序排列的单词序列  $w_1, w_2, \dots, w_n$ ，满足对于从 1 到  $n - 1$  的所有  $i$ ，单词  $w_i$  到

$w_{i+1}$  都是一次递变。

给出一部字典，你要计算其中最长的递变阶梯。

### 输入

输入文件是一部字典：每行都是一个由小写字母构成的单词。所有单词按照字典序排列。所有的单词长度都不超过 16，且字典中最多有 25 000 个单词。

### 输出

输出一个整数，代表最长的递变阶梯包含的单词数量。

#### 样例输入

```
cat
dig
dog
fig
fin
fine
fog
log
wine
```

#### 样例输出

```
5
```

### 9.6.6 立方体之塔 (Tower of Cubes)

PC/UVa 题号：110906/10051，流行度：C，通过率：high 难度：3

给你  $N$  个质量各异的彩色立方体。每个立方体都不是单色的——事实上，它们的每一面都被涂上了不同的颜色。你的工作是要用这些立方体建一座最高的塔，使得（1）每个立方体上面的那个立方体都比它轻，（2）每个立方体（除了塔底的立方体）底面的颜色必须和它下面那个立方体的顶面颜色相同。

### 输入

输入包含若干组数据。每组数据的第一行为一个整数  $N(1 \leq N \leq 500)$ ，表示立方体的数量。接下来的  $N$  行中，第  $i$  行描述了第  $i$  个立方体。该描述中的六个数分别表示这个立方体前、后、左、右、上、下这几个面的颜色。为了方便，颜色都用 1 ~ 100 之间的整数代表。假设输入立方体的质量是递增的，即第 1 个是最轻的而第  $N$  个是最重的。

当  $N = 0$  时输入结束。

### 输出

对于每组数据，按照样例格式先在单独的一行输出数据的序号，然后在下一行输出塔的最大高度。接下来按从上到下的顺序描述塔的每一层。每行描述一个立方体：先给出它的

输入序号，然后是空格和一个表示着色方式的字符串 (front, back, left, right, top 或者 bottom)，分别表示该立方体在塔中的顶面是输入中的哪个面。如果有多种解，任意输出一种即可。

相邻两组数据的输出之间用一个空行隔开。

### 样例输入

```
3
1 2 2 2 1 2
3 3 3 3 3 3
3 2 1 1 1 1
10
1 5 10 3 6 5
2 6 7 3 6 9
5 7 3 2 1 9
1 3 3 5 8 10
6 6 2 2 4 4
1 2 3 4 5 6
10 9 8 7 6 5
6 1 2 3 4 7
1 2 3 3 2 1
3 2 1 1 2 3
0
```

### 样例输出

Case #1	
2	
2 front	
3 front	
Case #2	
8	
1 bottom	
2 back	
3 right	
4 left	
6 top	
8 front	
9 front	
10 top	

### 9.6.7 从黄昏到拂晓 (From Dusk till Dawn)

PC/UVa 题号: 110907/10187, 流行度: B, 通过率: average 难度: 3

Vladimir 皮肤苍白，牙齿奇长并且已经 600 岁了，但这些对他来说都不成问题，因为他是一个吸血鬼。Vladimir 是一个相当合格的吸血鬼。事实上，他还是一个出色的医生。他总是上夜班，因此能与同事们广交朋友。他喜欢在宴会上表演一个令人瞠目结舌的把戏：用味觉分辨血型。Vladimir 喜欢旅行，但是作为一个吸血鬼，他需要解决三大问题：

1. 他只能坐火车，因为他要随身带着他的棺材。幸运的是，由于在长线投资中赚足了钱，他总是可以坐上头等车厢。
2. 他只能从黄昏旅行到拂晓，即只能在傍晚 6 点到早上 6 点期间旅行。其他时间他只能待在火车站。
3. 他每天正午 (12:00) 必须从棺材中取一升鲜血来喝。

请帮助 Vladimir 寻找一条两个指定城市之间的最短线路，把他所需携带的血量降低到最小限度。如果他带的血太多，人们一定会嘲笑他：“你带这么多血做什么？”

### 输入

输入第一行有一个整数，代表测试数据的数量。每组数据的第一行有一个整数，表示火车路线的数量。每条路线用两个城市的名称、出发时间和到达时间来描述，其中时间都是整点时刻（24 小时制）。记住，Vladimir 不能乘坐在 18:00 前出发或在 6:00 后到达的列车。

城市数量不超过 100，路线的数量不超过 1000。所有线路耗时都在 1 小时以上，24 小时以下，但是 Vladimir 只能乘坐从黄昏到拂晓这 12 个小时内的列车。

所有城市名的长度都不超过 32。每组数据最后一行的两个城市分别表示 Vladimir 的出发地和目的地。

### 输出

对于每组数据，按照样例格式先输出一行数据序号，然后在下一行输出“Vladimir needs # litre(s) of blood.”，或者“There is no route Vladimir can take.”。

#### 样例输入

```
2
3
Ulm Muenchen 17 2
Ulm Muenchen 19 12
Ulm Muenchen 5 2
Ulm Muenchen
10
Lugoj Sibiu 12 6
Lugoj Sibiu 18 6
Lugoj Sibiu 24 5
Lugoj Medias 22 8
Lugoj Medias 18 8
Lugoj Reghin 17 4
Sibiu Reghin 19 9
Sibiu Medias 20 3
Reghin Medias 20 4
Reghin Bacau 24 6
Lugoj Bacau
```

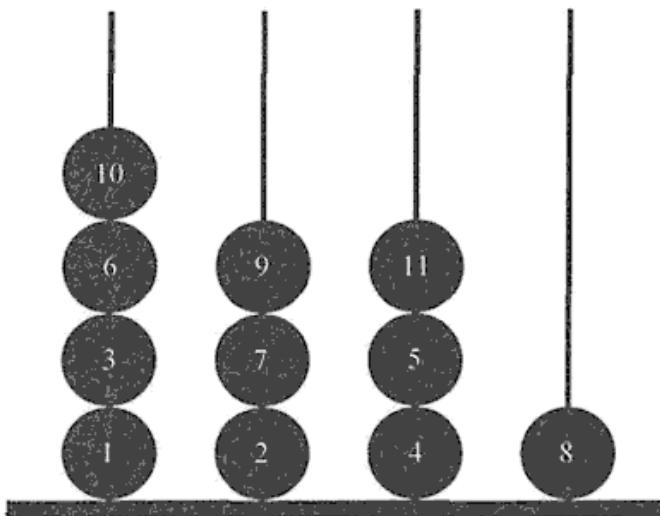
#### 样例输出

```
Test Case 1.
There is no route Vladimir can take.
Test Case 2.
Vladimir needs 2 litre(s) of blood.
```

### 9.6.8 汉诺塔卷土重来! (Hanoi Tower Troubles Again!)

PC/UVa 题号: 110908/10276, 流行度: B, 通过率: high 难度: 3

汉诺塔问题有很多有趣的变体。本题包含了  $N$  根柱子和编号为  $1, 2, 3, \dots, \infty$  的球。当两个球的编号和不是完全平方数 (即某个整数  $c$  的平方  $c^2$ ) 时, 它们会互相排斥而无法接触。



游戏者必须将这些球按顺序一个个地放到柱子上 (先放第一个球, 然后第二个, 接着第三个等)。当找不到不排斥的放法时, 游戏结束。

游戏的目标是要将尽可能多的球放到柱子上。上面的例子给出了 4 根柱子时的最优方案。

#### 输入

输入第一行为一个整数  $T(1 \leq T \leq 50)$ , 表示测试数据的数量。每组数据只有一个整数  $N(1 \leq N \leq 50)$ , 表示可用的柱子数量。

#### 输出

对于每组数据, 输出一行, 表示可以被放在柱子上的球的最大数量。如果有无穷多个球可以被放上去, 则输出 “-1”。

#### 样例输入

2  
4  
25

#### 样例输出

11  
337

## 9.7 提 示

9.1 可以在一次遍历中完成整张图的着色吗?

9.2 题目背后隐藏着怎样一张图?

9.3 本题可以归约到连通性判断吗？

9.4 把图显式地表示出来比较方便，还是直接处理斜线矩阵就可以了？

## 9.5 题目背后隐藏着怎样一张图？

9.6 是否能用这些立方体构成一张有向图，使得所求的塔对应于图上的一条路径？

9.7 本题可以表示成无权图上的问题，并用 BFS 来解决吗？

9.8 本题中的约束能否用 DAG 来有效建模?

## 第10章 图 算 法

第 9 章中介绍的图表示和遍历算法为图结构上的计算提供了基本工具。在本章中，我们介绍一些相对高级的关于图的理论和算法。

图论 (*graph theory*) 是一门关于图的结构和性质的学科，它为图的讨论提供了相应的数学语言。解决很多问题的关键是首先识别出问题背后隐藏的图论概念，然后借用经典算法来解决建模出的图论问题。

我们首先简要回顾一下图论中的基本内容，然后介绍一些重要的图算法，如最小生成树、最短路和最大流。

## 10.1 图 论

在本节中，我们将快速地回顾图论中的一些基本内容。更加详细的讨论参见图论方面的优秀书籍 [PS03, Wes00]。我们还将给出一些相关算法的框架，它们在前一章代码的基础上不难实现。

### 10.1.1 度的性质

图是由结点和边组成的。结点最简单的属性就是它的度数 (*degree*)，即它所关联的边数。

结点度数有一些重要性质。在无向图中，所有结点的度数之和等于边数的两倍。这是因为每条边在两端的度数中各被统计了一次。这个结论的一个推论是：在无向图中，度数为奇数的点有偶数个。在有向图中，所有结点的入度之和等于出度之和。结点度数的奇偶性在 10.1.3 小节即将讨论的欧拉回路中扮演了重要角色。

树 (*trees*) 是不含环的无向连通图。结点度数在树结构的分析中占有重要地位。度数为 1 的结点称为叶子 (*leaf*), 而每棵  $n$  结点树恰好包含  $n - 1$  条边, 因此所有非平凡树中至少有两个叶子。删除叶子将得到一棵小一点的树, 它只是把原来那棵树稍微裁剪了一下, 并不会引起剩下的结点不连通。

有根树 (*rooted trees*) 是除根之外每个点的入度均为 1 的有向图。叶子是那些出度为 0 的点。二叉树 (*binary trees*) 就是一种有根树。如果每个结点的出度要么为 0 要么为 2，则至少一半的结点都是叶子。

无向图  $G = (V, E)$  的生成树 (*spanning tree*) 是边的一个子集  $E' \subset E$ , 使得  $E'$  组成一棵以  $V$  为结点集的树。任何连通图都有生成树。例如, 宽度优先遍历和深度优先遍历得到的搜索树都是原图的生成树。加权图的最小生成树 (*minimum spanning tree*) 是一个重要问题。

题，我们将在 10.2 节中讨论它。

### 10.1.2 连通性

无向图是连通的 (*connected*) 当且仅当任意两个结点之间都有通路。生成树的存在性保证了连通性。我们已经在 9.4.2 小节中给出了一个基于深度优先遍历的连通分量寻找算法。

除此之外，还有一些重要的概念值得注意。点/边连通度 (*vertex/edge connectivity*) 是最少的点 (边) 的数目，使得删除了这些点 (边) 之后图不再连通。如果删除了某个结点之后图不再连通，我们称该点是一个割顶 (*articulation vertex*)，不存在割顶的图称为双连通的 (*biconnected*)。如果删除了某条边后图不再连通，我们称该点是一个桥 (*bridge*)，不存在桥的图称为边-双连通的。

割顶和桥的检测算法可以简单地利用枚举法完成：对于每个结点和每条边，试着从图中删除，然后判断剩下的图是否仍然连通。别忘了在下次测试之前把刚才删除的点或边重新插入到图中！

在有向图中，我们最关心的是强连通分量 (*strongly connected components*)，即把图划分成若干部分，使得在每个部分中任意两个结点都能互达。公路网应当是强连通的，否则就会存在一些地方，一旦过去了就回不来了（除非在单行道逆行）！

下面的思路可以用来找到有向图中的强连通分量。首先用深度优先遍历任意找一个有向环（一条反向边加上 DFS 树中的路径），则环中的所有点应该共属同一个强连通分量。接下来把环收缩 (*shrink*) 成一个点，然后重复这一过程，直到图中不存在有向环。此时，图中的每个结点对应一个强连通分量。

### 10.1.3 图中的回路

不是树的无向连通图中都有回路，其中那些访问了所有点或边的回路尤其重要。

欧拉回路 (*eulerian cycle*) 是访问图中每条边恰好一次的回路。孩子们喜欢的“一笔画”就是欧拉回路 (或道路) 的典型例子, 其中顶点是图画中的交叉点, 边是图画中的线段。在理想情况下, 邮递员的路线也应该是欧拉回路, 这样他就可以在回家之前访问每条街道 (边), 而且不用重复经过任何一条街。

一个无向连通图包含欧拉回路的充分必要条件是：每个结点的度数均为偶数。为什么？一方面，回路每经过一个结点都有一个“一进一出”的过程，因此所有的度数都是偶数。另一方面，只要度数全是偶数，我们就可以一次找一个环，然后合并成欧拉回路。具体来说，可以用 9.4.1 小节中的算法用 DFS 找一个环，然后在图中删除这个环。不难证明，删除之后，每个点的度数仍然是偶数。重复这一过程，直到我们把原图中的所有边分成了若干个边不相交 (*edge-disjoint*) 的环。接下来，在这些环的公共点处把它们连接起来，就得到了一条欧拉回路。

在有向图中，条件变为了：每个结点的入度等于出度<sup>1</sup>。删环过程将保持这个性质，因此上述方法对于有向图仍然适用。欧拉道路 (*eulerian paths*) 是指访问每条边恰好一次的道路，它和欧拉回路的区别在于终点不一定和起点重合。这样，恰好可以有两个结点不满足出度和入度相等的条件，其中出度比入度大 1 的是欧拉道路的起点，入度比出度大 1 的是终点。

哈密顿回路 (*hamiltonian cycle*) 是指经过每个结点恰好一次的回路。旅行商问题的目标是找出加权图中所有哈密顿回路中最短的一条。图  $G = (V, E)$  上的欧拉回路问题可以归约到一个新图  $G' = (V', E')$  上的哈密顿回路问题，其中  $V'$  中的每个结点对应于  $E$  中的一条边，而  $E'$  中的边  $(e_1, e_2)$  表示  $G$  中包含两条有公共端点的边  $e_1$  和  $e_2$ 。

遗憾的是，目前还没有找到寻找哈密顿回路的有效算法。因此，当遇到这个问题时，你有两个选择。如果图足够小，可以用回溯法解决。每个哈密顿回路对应于顶点的一个排列，当从上一个结点到当前结点的边不存在时回溯。如果图太大，就需要重新对问题建模，比如把问题转化成另一个图上的欧拉回路问题。

#### 10.1.4 平面图

平面图 (*planar graphs*) 是那些可以画在平面上, 使得任意两条边不交叉的图。很多常见的图都是可平面 (*planar*) 的。所有的树都是可平面的: 你能设计出树的平面嵌入算法吗? 没有立交桥的公路网都是平面图。凸多面体的邻接结构也能导出一个平面图。

平面图有若干重要性质。首先，它的点数  $n$ 、边数  $m$  和面数  $f$  关系密切。欧拉公式 (*Euler's formula*) 表明： $n - m + f = 2$ 。树有  $n - 1$  条边，因此树的平面嵌入恰好有一个面，即无限面。立方体的任意嵌入 (8 点 12 边) 都是恰好 6 个面，就像每个玩色子的人所知道的一样。

平面性检测和寻找边不相交的图绘制方案都存在有效算法，但是它们实现起来都很复杂。不过，用欧拉公式可以方便地证明某些特定的图不是可平面的。对于  $n > 2$ ，每个平面图最多包含  $3n - 6$  条边。作为推论，每个平面图至少有一个点的度数不超过 5，而且删除这个点后的图仍然保持这一性质。检查一个图绘制方案是否为平面嵌入等价于检查一个线段集合中是否存在两条相交线段。我们将在几何算法部分中讨论这一问题。

## 10.2 最小生成树

图  $G = (V, E)$  的生成树 (*spanning tree*) 是  $E$  的一个子集, 其中的边形成了连接  $V$  中所有点的树。对于带边权的图, 我们对其中的最小生成树 (*minimum spanning tree*) 尤其感兴趣, 它是边权之和最小的生成树。

如果我们希望用长度最短的道路、电线或者管道连接一个点集（表示城市、交叉路口或者其他位置）中的所有点，最小生成树正是我们所需要的。如果只要求边数最少，任意生成

<sup>1</sup> 谱者注：别忘了大前提是这个有向图的基图连通。

树都满足条件；但如果要边权之和最小，只有最小生成树符合要求。

计算最小生成树的主流算法有两个：Kruskal 算法和 Prim 算法。二者在大多数算法课程中都有介绍。在这里，我们只介绍 Prim 算法，因为它更容易实现<sup>2</sup>，并且只需少量改动就可以变成 Dijkstra 算法。

首先，我们必须扩充第 9 章中的图表示法来支持加权图。当时每条边只保存了另一个端点的编号，而现在需要把它替换成一个记录，以便加上权重信息：

```

typedef struct {
    int v;                                /* neighboring vertex */
    int weight;                            /* edge weight */
} edge;

typedef struct {
    edge edges[MAXV+1][MAXDEGREE]; /* adjacency info */
    int degree[MAXV+1];                /* outdegree of each vertex */
    int nvertices;                     /* number of vertices in graph */
    int nedges;                         /* number of edges in graph */
} graph;

```

然后更新相应的初始化和遍历算法。这些都不困难。

Prim 算法从一个给定起点开始一步一步生长最小生成树。在算法的每次迭代中，我们往生成树中增加一个结点。贪心算法可以保证正确性：我们总是加入所有“从树中指向树外”的边中权值最小的一条。

最简单的实现方法需要给每个结点赋予一个布尔变量，表明它是否在树中（代码中的数组 `intree`），然后在每次迭代中遍历所有边，找出恰好有一个 `intree` 结点的边中权值最小的一个。

我们的实现方法比这智能：它为每个树外结点保存了从树出发到该结点的最小边（代码中的 `distance` 数组），则每次迭代只需求出这些边中的最小值 `dist`，然后更新 `distance` 数组。由于每次只在树中增加一个点 `v`，所以只需更新从该点出发的所有相邻点 `w` 的 `distance` 值：

<sup>2</sup> 译者注：实用的 Prim 算法需要用到二叉堆或者类似的数据结构，但 Kruskal 算法只用到并查集和排序，容易实现且往往更加高效。

```
int distance[MAXV];           /* vertex distance from start */
int parent[MAXV];             /* record tree topology */
int v;                         /* current vertex to process */
int w;                         /* candidate next vertex */
int weight;                    /* edge weight */
int dist;                      /* shortest current distance */

for (i=1; i<=g->nvertices; i++) {
    intree[i] = FALSE;
    distance[i] = MAXINT;
    parent[i] = -1;
}

distance[start] = 0;
v = start;

while (intree[v] == FALSE) {
    intree[v] = TRUE;
    for (i=0; i<g->degree[v]; i++) {
        w = g->edges[v][i].v;
        weight = g->edges[v][i].weight;
        if ((distance[w] > weight) && (intree[w]==FALSE)) {
            distance[w] = weight;
            parent[w] = v;
        }
    }
    v = 1;
    dist = MAXINT;
    for (i=1; i<=g->nvertices; i++)
        if ((intree[i]==FALSE) && (dist > distance[i])) {
            dist = distance[i];
            v = i;
        }
}
```

}

最小生成树的权值或者这棵树本身有两种方法可以得到。最简单的方法是在上述过程中增加几条语句来打印选中的边，或者把边权累加到一个变量中供以后使用。另一种方法是利用 `parent` 数组，由它可以构造出完整的最小生成树。

最小生成树有一些不错的特性，可以用来解决很多相关问题：

- **最大生成树** — 假定我们找了一家“邪恶”的电话公司把一些住房连接起来。该公司得到的费用将正比于电缆的总长度。很自然的，公司希望得到尽量多的钱，但又不能太明目张胆（因此电缆不能成环），所以最大生成树 (*maximum spanning tree*) 成为了它的首选。只需在所有边权的前面加一个“负号”，求出的最小生成树就是原图的最大生成树。

大多数图算法在碰到负数的时候都会存在问题。事实上，当边权有负数时，最短路算法会遇到麻烦，刚才那种“边权符号取反”的技巧就不管用了。

- **乘积最小的生成树** — 假设所有边权均为正，要求一棵边权乘积尽量小的生成树。由于  $\lg(a \cdot b) = \lg(a) + \lg(b)$ ，只要把边权用它的对数代替，得到的新图的最小生成树就是原图乘积最小的生成树。
- **最小瓶颈生成树** — 有时我们需要最大边权最小的生成树。事实上，最小生成树就满足这个性质。根据 Kruskal 算法的正确性很容易证明这一点。

当边权被解释为费用、容量和力量时，这样的瓶颈生成树有一些有趣的应用。另一个解决这个问题的方法是从小到大枚举最大边权，把权比它更大的边全部删除，然后判断图是否连通<sup>3</sup>（这可以用简单的 BFS/DFS 完成）。

如果所有  $m$  条边权各不相同，则最小生成树是唯一的；否则，Prim 算法可能会需要在多条权相同的边中任意选择。具体的选择方案决定了最后得到的是哪一个最小生成树。

## 10.3 最 短 路

我们已经在 9.3.1 小节中讨论了无权图中的最短路问题。宽度优先遍历能很好地完成这个工作，但也仅此而已。BFS 不能找出加权图中的最短路径，因为  $a$  到  $b$  的权和最小的路径不一定包含的边数最少。很多人开车都喜欢走小路，尽管他们总是拐来拐去的（边数较多），但能更快到达目的地（权和更小）。

我们将在本节实现两种不同的算法在加权图中寻找最短路。

### 10.3.1 Dijkstra 算法

如果要在带点权和/或边权的图中寻找两个结点之间的最短路，Dijkstra 是再合适不过的了。

给定一个起点  $s$ ，该算法找出从  $s$  到所有其他结点（包括终点  $t$ ）的最短路。

<sup>3</sup> 译者注：事实上，如果把这个过程看成是从小到大加入边，这正是 Kruskal 算法！

算法思想和 Prim 算法很接近。每次迭代，我们都恰好在以  $s$  为根的最短路树（它包含所有已经算出最短路的结点。在这棵树上，从  $s$  到每个其他结点的唯一简单路径就是原图中从  $s$  到这些点的最短路）中增加一个点。和 Prim 算法一样，对于树外的每个结点，我们维护目前为止发现的最短路，然后按照路径长度从小到大的顺序把这些结点移到树中。

Dijkstra 算法和 Prim 算法的区别在于按照何种顺序把树外的结点移到树中。在最小生成树问题中，我们只关心把结点从树外移到树内的最后一条边的权值；在最短路经问题中，我们关心的是从起点出发到达各个树外结点的路径总长。这个总长既和最后一条边的权值有关，也和从起点到新增点前一个点的路径长度有关。

事实上，这个改动非常小。下面是 Dijkstra 算法的完整实现，它在 Prim 算法的基础上仅仅改动了三行——其中一行只是修改了函数名称<sup>4</sup>。

```
dijkstra(graph *g, int start)          /* WAS prim(g,start) */
{
    int i,j;                            /* counters */
    bool intree[MAXV];                 /* is vertex in the tree yet? */
    int distance[MAXV];                /* vertex distance from start */
    int v;                             /* current vertex to process */
    int w;                             /* candidate next vertex */
    int weight;                        /* edge weight */
    int dist;                          /* shortest current distance */

    for (i=1; i<=g->nvertices; i++) {
        intree[i] = FALSE;
        distance[i] = MAXINT;
        parent[i] = -1;
    }

    distance[start] = 0;
    v = start;

    while (intree[v] == FALSE) {
        intree[v] = TRUE;
        for (i=0; i<g->degree[v]; i++) {
```

<sup>4</sup> 译者注：另一个需要注意的地方是 MAXINT 的大小。为了保证加法  $distance[v]+weight$  不至于溢出后变成负数，这里的 MAXINT 不应该设置成最大可以表示的整数值。

```
w = g->edges[v][i].v;
weight = g->edges[v][i].weight;
/* CHANGED */ if (distance[w] > (distance[v]+weight)) {
/* CHANGED */     distance[w] = distance[v]+weight;
parent[w] = v;
}
}

v = 1;
dist = MAXINT;
for (i=1; i<=g->nvertices; i++) {
    if ((intree[i]==FALSE) && (dist > distance[i])) {
        dist = distance[i];
        v = i;
    }
}
}
```

如何运用 dijkstra 求出从 start 到某个给定结点 t 的最短路长度呢？答：所求值就是 `distance[t]`。那如何构造出这条具体的路径呢？顺着 `parent` 从 t 倒着走到 start（如果 `parent` 等于 -1，表示最短路不存在）即可，和 9.3.3 小节中的 `find_path()` 函数完全一样。

和 Prim 算法不同的是，Dijkstra 算法只适用于没有负权边的图。原因在于，在算法执行的过程中，我们可能遇到一条“负得特别厉害”的边，甚至让从 s 到某些“已经求出最短路”的点的最短路发生改变。事实上，从你家到邻居家最省钱的路径是绕到一家银行去取出足够的钱后再回来。

在大多数应用中，我们所处理的图都不包含负权边，因此刚才的讨论仅仅在理论上是重要的。下面即将讨论的 Floyd 算法能够正确地处理负权边，除非图中包含负权环 (*negative cost cycles*)。在这种情况下，最短路结构将受到严重冲击。如果你账户里的钱有无穷多，你甚至可以无限次地经过银行取钱，永远都不到达目的地！

### 10.3.2 每对结点之间的最短路

很多应用都需要知道一个给定图上每对结点之间的最短路长度。例如，你可能需要求出某个图的“中心点”，即“到其他所有点的最长或平均距离”最小的结点。这个中心点可能会是开展新业务的最佳场所。又或者，你想求出图的直径 (*diameter*)，即每对结点的最短路长度的最大值。它可以用来衡量一封信或一个网络包从任意起点到任意终点的最长可能时间。

一种解决方法是从所有 n 个可能的起点出发分别调用 Dijkstra 算法，但是稍显麻烦。另

一方面，接下来要介绍的 Floyd 算法可以从图的边权矩阵直接算出距离矩阵，其简洁程度令人吃惊。

Floyd 算法很自然地用到了邻接矩阵。考虑到算法结果本身也包含  $n^2$  个距离，邻接矩阵的空间开销并不算奢侈。在代码中，adjacency-matrix 数据类型为矩阵静态分配了足够大的空间，并保存了图中的结点数：

```
typedef struct {
    int weight[MAXV+1][MAXV+1];      /* adjacency/weight info */
    int nvertices;                   /* number of vertices in graph */
} adjacency_matrix;
```

邻接矩阵表示中的一个关键问题是：如何表示图中不存在的边？在无权图中，一般用 1 表示图中的边，用 0 表示不在图中的边。不能用这个方法表示加权图，否则原本不在图中的边会被认为是可以“免费”或者“瞬间”经过的，权为 0 的边！正确的处理方法是把所有不在图中的边设置为 MAXINT。这样，我们不仅可以检测一条边是否在图中，而且在最短路问题中可以自动忽略它们，因为只要 MAXINT 的值大于图的直径<sup>5</sup>，这些边一定不会被选用。

```
initialize_adjacency_matrix(adjacency_matrix *g)
{
    int i,j;                      /* counters */

    g->nvertices = 0;

    for (i=1; i<=MAXV; i++)
        for (j=1; j<=MAXV; j++)
            g->weight[i][j] = MAXINT;
}

read_adjacency_matrix(adjacency_matrix *g, bool directed)
{
    int i;                         /* counter */
    int m;                          /* number of edges */
    int x,y,w;                     /* placeholder for edge/weight */

    initialize_adjacency_matrix(g);
```

<sup>5</sup> 译者注：“无穷大不够大”是几乎所有人都犯过的错误，请读者留意。

```

scanf("%d %d\n", &(g->nvertices), &m);

for (i=1; i<=m; i++) {
    scanf("%d %d %d\n", &x, &y, &w);
    g->weight[x][y] = w;
    if (directed==FALSE) g->weight[y][x] = w;
}
}

```

目前的代码还没有什么技术含量。接下来，如何根据邻接矩阵求解最短路？Floyd 算法的第一步是把所有结点编号为  $1 \sim n$ 。这些编号不是用来标识这些结点，而是用来指定一个处理它们的顺序。

算法包含  $n$  次迭代，其中第  $k$  次迭代给每对结点计算出“只允许前  $k$  个结点作为路径中间点”的最短路。当  $k = 0$  时，不允许任何结点作为路径中间点，因此所有最短路经就是原图中的那些单独的边。换句话说，初始的最短路矩阵就等于邻接矩阵。每一次开始新的迭代，允许结点集就扩大了一点。从  $i$  途径  $k$  到达  $j$  的路可能比原来那些从  $i$  到  $j$  只经过结点  $1$  到  $k - 1$  的路更短，即：

$$W[i, j]^k = \min(W[i, j]^{k-1}, W[i, k]^{k-1} + W[k, j]^{k-1})$$

思路很直观，但算法的正确性并不是那么显然的，请读者自行思考。尽管如此，程序的简洁和优美却是毋庸置疑的<sup>6</sup>：

```

floyd(adjacency_matrix *g)
{
    int i,j;                      /* dimension counters */
    int k;                          /* intermediate vertex counter */
    int through_k;                 /* distance through vertex k */

    for (k=1; k<=g->nvertices; k++)
        for (i=1; i<=g->nvertices; i++)
            for (j=1; j<=g->nvertices; j++) {
                through_k = g->weight[i][k]+g->weight[k][j];
                if (through_k < g->weight[i][j])
                    g->weight[i][j] = through_k;
            }
}

```

<sup>6</sup> 译者注：在代码中，迭代变量  $k$  并没有作为数组下标。这样做的正确性也不是显然的，请读者自行思考。

P1051.COM  
最新  
教材  
课件  
习题  
答案  
网盘  
资源

}

上述代码只算出了距离矩阵，但并没有给出实际的路径。如果需要实际路径，可以用 Dijkstra 算法<sup>7</sup>。不过请注意，大部分需要每对结点间最短路经的应用只需要最终的距离矩阵，而这个任务正是 Floyd 算法的专职。

Floyd 算法还有一个重要应用：计算有向图的传递闭包 (*transitive closure*)。在分析有向图时，我们经常需要知道从一个给定结点出发能到达哪些点。

例如，考虑  $n$  个人组成的勒索图 (*blackmail graph*)，其中有向边  $(i, j)$  表示  $i$  抓住了  $j$  的把柄，可以任意摆布他。你想雇佣这  $n$  个人中的一个来做你的代表。谁的勒索能力最强？

表面上，度数最大的结点是一个不错的选择，但更强大的候选者可以通过“链式反应”间接勒索更多的人。Steve 可能仅仅能直接勒索 Miguel 一个人，但如果 Miguel 可以勒索其他所有人，则 Steve 就是你的最佳选择。

从任意单个结点可达的结点集能用 BFS 和 DFS 算出，但所有结点对之间的可达关系可以更加简单的从距离矩阵中直接得到。如果运行 Floyd 算法之后  $i$  到  $j$  的最短路长度仍然是 MAXINT，则从  $i$  到  $j$  没有有向路径；若小于 MAXINT，则从  $i$  可以到达  $j$ （在勒索图中，这代表  $i$  可以直接或间接地控制  $j$ ）。

## 10.4 网络流和二分图匹配

你可以把边上带正权的图想象成管道网络，其中  $(i, j)$  的边权衡量该管道的容量 (*capacity*)。容量可以想象成管道截面积的函数——一根粗管道可以在单位时间内通过 10 个单位的流量，而一根细管道只能通过 5 个单位。给定加权图  $G$  和两个结点  $s$  和  $t$ ，网络流问题 (*network flow problem*) 的目标是求出在满足每根管道的容量限制的条件下，单位时间内从  $s$  运送到  $t$  的最大流量。

尽管网络流问题本身有着很多有趣的直接应用，它的重要性更多地体现在解决其他图论问题的能力上。一个经典问题是二分图匹配。图  $G = (V, E)$  的匹配 (*matching*) 是边集的子集  $E' \subset E$ ，其中的任意两条边都没有公共端点。换句话说，匹配将图中的一些顶点配对，使得原图中的每个顶点最多在一个点对中出现。

如果图  $G$  的顶点可以分成两个集合  $L$  和  $R$ ，使得  $G$  中的所有边都是其中一个端点在  $L$  中，另一个端点在  $R$  中，则称图  $G$  是二分图 (*bipartite graph*) 或二着色图 (*two-colorable graph*)。很多真实应用中的图都是二分图。例如，用一些结点表示待完成的工作，剩下的结点表示能完成这些工作的工人。边  $(j, p)$  表示工作  $j$  可以由工人  $p$  完成。另一个例子是用一些结点表示男人，另一些结点表示女人，边表示有可能结婚的一对。这两个图中的匹配将分

<sup>7</sup> 译者注：Floyd 是一种动态规划算法。学习完动态规划之后，请读者自行思考如何修改上述代码，直接给出实际路径。

别解释成任务分配方案和相容的婚姻集合。

二分图最大匹配可以用网络流求解。建立源 (*source*)  $s$ , 并从它出发向  $L$  中的所有结点连一条有向弧, 容量均为 1。建立汇 (*sink*)  $t$ , 并从  $R$  中的所有结点出发分别向它连一条有向弧, 容量均为 1。这样, 原图  $G$  中的最大匹配就可以从新图中  $s$  到  $t$  的最大流得到。对于原图中任意匹配的每条边  $(i, j)$ , 让新图中的  $(s, i), (i, j), (j, t)$  三条弧均满载, 则原图的任意匹配总是能对应新图的一个可行流; 另一方面, 对于新图中的任意可行流, 满足  $i \in L, j \in R$  的所有满载弧  $(i, j)$  构成了原图的一个匹配。换句话说, 原图的匹配和新图的可行流一一对应。原图的最大匹配自然对应着新图的最大流。

最容易实现的网络流算法是 Ford-Fulkerson 增广路算法。对于每条弧, 我们同时记录它的流量和残余容量 (*residual capacity*)。因此, 我们需要修改 *edge* 数据结构, 加上一个新的域:

```
typedef struct {
    int v;                      /* neighboring vertex */
    int capacity;                /* capacity of edge */
    int flow;                    /* flow through edge */
    int residual;                /* residual capacity of edge */
} edge;
```

我们每次从源到汇寻找一条可以增加总流量的路径, 并且用它增广。当没有增广路 (*augmenting path*) 存在时算法终止, 此时的流就是最大流。

```
netflow(flow_graph *g, int source, int sink)
{
    int volume;                  /* weight of the augmenting path */

    add_residual_edges(g);

    initialize_search(g);
    bfs(g, source);

    volume = path_volume(g, source, sink, parent);

    while (volume > 0) {
        augment_path(g, source, sink, parent, volume);
        initialize_search(g);
```

```
bfs(g,source);
volume = path_volume(g, source, sink, parent);
}
}
```

从源到汇的任意增广路都能增加总流量，因此我们可以借用现成的 bfs。需要注意的是，我们只能沿着“还能增广”（即残余容量为正数）的边走，因此需要在以下函数中判断残余容量是否为正，以帮助 bfs 区分开饱和边 (*saturated edges*) 和非饱和边 (*unsaturated edges*)：

```
bool valid_edge(edge e)
{
    if (e.residual > 0) return (TRUE);
    else return(FALSE);
}
```

增广的过程把尽量多的残余容量转化成正流量。增广路的容量等于整条路中残余容量的最小值，正如车流的速度取决于最拥挤的路段。

```
int path_volume(flow_graph *g, int start, int end, int parents[])
{
    edge *e; /* edge in question */
    edge *find_edge();

    if (parents[end] == -1) return(0);

    e = find_edge(g,parents[end],end);

    if (start == parents[end])
        return(e->residual);
    else
        return( min(path_volume(g,start,parents[end],parents),
                    e->residual));
}

edge *find_edge(flow_graph *g, int x, int y)
{
    int i; /* counter */

```

```
        for (i=0; i<g->degree[x]; i++)
            if (g->edges[x][i].v == y)
                return( &g->edges[x][i] );
        return(NULL);
    }
```

沿着有向弧  $(i, j)$  多运送一个单位的流量将让  $(i, j)$  的残余容量减小一个单位，但  $(j, i)$  的残余容量将增加一个单位。换句话说，增广时不仅要找到增广路上每条弧本身（也称为前向弧 (*forward edge*)），还需要找到它们各自的反向弧 (*reverse edge*)。

```

augment_path(flow_graph *g, int start, int end, int parents[], int volume)
{
    edge *e;                                /* edge in question */
    edge *find_edge();

    if (start == end) return;

    e = find_edge(g,parents[end],end);
    e->flow += volume;
    e->residual -= volume;

    e = find_edge(g,end,parents[end]);
    e->residual += volume;

    augment_path(g,start,parents[end],parents,volume);
}

```

在初始化时，每条有向边  $e = (i, j)$  都需要拆成两条弧  $(i, j)$  和  $(j, i)$ ，其中  $(i, j)$  的初始残余容量为  $e$  的容量， $(j, i)$  的初始残余容量为 0。<sup>8</sup> 所有弧的初始流均设为 0。<sup>9</sup>

网络流是一个相对高级的算法设计技巧，判断一个特定问题是否可以使用网络流算法求解需要经验。推荐读者阅读 Cook 和 Cunningham 的 [CC97]，以及 Ahuja、Magnanti 和 Orlin 的 [AMO93]，以更加深入地了解这个主题。

<sup>8</sup> 译者注：相同的多条弧可以合并成一条，容量相加；方向相反的两条弧不能抵消，二者的初始残余容量均为各自的容量。这些都是初学者在初始化中容易忽略的问题。

<sup>9</sup> 译者注：事实上，任意可行流都可以作为算法的初始流。快速构造接近最大流的可行流能大大提高算法效率。

## 10.5 习 题

### 10.5.1 斑点 (Freckles)

PC/UVa 题号: 111001/10034, 流行度: B, 通过率: average 难度: 2

在 “Dick Van Dyke” 的某一集中, 小 Richie 把他爸爸背上的斑点连成了独立钟<sup>10</sup>的图案。但是其中一个斑点变成了伤疤, 所以 Richie 的尝试失败了。

将 Dick 的背看成一个平面, 上面有不同位置的斑点  $(x, y)$ 。你的任务是要告诉 Richie 应该怎样连接所有点, 使得所用的墨水最少。Richie 总是用直线段连接两个点, 在画不同线段之间可以将笔提起。当 Richie 结束画线时, 任何一个斑点必须能够经过所画的线到达任何另外一个斑点。

#### 输入

输入第一行只有一个正整数, 代表测试数据的数量。接下来有一个空行。

每组数据的第一行有一个整数  $n(0 < n \leq 100)$ , 表示 Dick 背上的斑点数量。接下来的每行有两个实数  $(x, y)$ , 分别代表他背上每个斑点的位置。

相邻两组输入数据之间有一个空行。

#### 输出

对于每组数据, 你的程序要输出一个保留两位小数的实数, 表示将所有斑点连接起来所需线段的最小总长度。

相邻两组数据的输出之间应有一个空行。

#### 样例输入

```
1
3
1.0 1.0
2.0 2.0
2.0 4.0
```

#### 样例输出

```
3.41
```

### 10.5.2 项链 (The Necklace)

PC/UVa 题号: 111002/10054, 流行度: B, 通过率: low 难度: 3

我妹妹有一条由彩色珠子串成的漂亮项链。如图所示, 项链中每两个相邻珠子的接触点

<sup>10</sup> 译者注: 美国费城独立厅的大钟, 1776 年 7 月 4 日鸣此钟宣布美国独立, 1835 年被损。

颜色相同：



噢不！突然有一天，项链断掉了，所有的珠子都散开了，滚得到处都是。妹妹找到了所有她能找到的珠子，但她无法确定是否把珠子都找齐了。她想知道，只用那些已经找到的珠子是否也可以穿出一条具有相同特点的项链。如果可以的话，应该怎样安排这些珠子的顺序？

写一个程序来解决这个问题。

### 输入

输入第一行有一个整数  $T$ ，表示测试数据的个数。每组数据的第一行为一个整数  $N(5 \leq N \leq 1000)$ ，表示妹妹已经找到的珠子的数量。接下来的  $N$  行每行有两个整数，代表珠子两端的颜色。颜色用  $1 \sim 50$  间的整数表示。

### 输出

对于每组数据，像样例那样先输出数据编号。如果不可能穿出项链，在单独的一行输出“*some beads may be lost*”。否则输出  $N$  行，按项链中的顺序输出每个珠子两端的颜色。对于所有的  $i(1 \leq i \leq N - 1)$ ，第  $i$  行的第二个整数必须和第  $i + 1$  行的第一个整数相同，第  $N$  行的第二个整数必须和第 1 行的第一个整数相同。如果有多个解，任何输出一组均可。

相邻两组数据之间应输出一个空行。

#### 样例输入

```
2
5
1 2
2 3
3 4
4 5
5 6
5
2 1
2 2
3 4
3 1
2 4
```

#### 样例输出

```
Case #1
some beads may be lost

Case #2
2 1
1 3
3 4
4 2
2 2
```

### 10.5.3 消防站 (Fire Station)

PC/UVa 题号: 111003/10278, 流行度: B, 通过率: low 难度: 2

某城市中有一些消防队。居民抱怨说一些房子离最近的消防站都很远，所以必须建一个新的消防站。你要选择这个新消防站的位置，让它离最缺少服务的居民区最近。

这个城市最多有 500 个路口，被长度不同的道路连接。汇集在同一个路口的道路数不会超过 20。房子和消防站的位置都在路口上。假定在每个路口至少有一栋房子，且在同一个路口可以有多个消防站。

#### 输入

输入第一行只有一个整数，代表测试数据的数量，接下来有一个空行。相邻两组数据之间也有一个空行。

每组数据的第一行有两个正整数：现有的消防站数量  $f$  ( $f \leq 100$ ) 和路口的数量  $i$  ( $i \leq 500$ )。路口用  $1 \sim i$  的连续整数表示。接下来的  $f$  行，每一行有一个数，表示该消防站所在路口的编号。后面的若干行，每行有 3 个正整数：一个路口的编号，另一个路口的编号，和连接这两个路口道路的长度。所有的道路都是双向的（至少消防车可以从两边通过），并且任何两个路口之间都是可达的。

#### 输出

对于每一组数据，输出新消防站所在的路口编号（使得任意路口到最近消防站的最大距离最小）。如果有多个路口满足条件，输出编号最小的一个。相邻两组数据的输出应被一个空行隔开。

#### 样例输入

1

1 6

2

1 2 10

2 3 10

3 4 10

4 5 10

5 6 10

6 1 10

#### 样例输出

5

#### 10.5.4 铁路 (Railroads)

PC/UVa 题号: 111004/10039, 流行度: C, 通过率: average 难度: 3

明天早上 Jill 必须要从 Hamburg 到 Darmstadt 去参加程序设计区域赛。她害怕因迟到而无法参加比赛，所以要寻找能够最早到达 Darmstadt 的火车。但是她不喜欢太早出发，所以如果有多种方案同时到达 Darmstadt，她会选择最晚出发的一个。

Jill 请你帮助她解决这个问题。她会给你一份列车时刻表，你的任务是根据它计算最早的到达时间和对应的最快乘车方式。幸运的是，Jill 在换乘上很有经验，所以她可以不消耗任何时间换乘列车。

#### 输入

输入第一行是一个整数，表示测试数据的数量。每个测试数据有三个部分。第一部分是所有被铁路连接的城市名称：首先是一个整数  $C(1 < C \leq 100)$ ，接下来的  $C$  行为城市的名字。所有的名称中只包含字母。

第二部分为列车时刻表：首先是一个整数  $T(T \leq 1000)$ ，下面是  $T$  列火车的描述。每组描述首先是一个整数  $t_i(t_i \leq 100)$ ，接下来是  $t_1$  行，每行有一个时间和一个城市的名字，表示乘客可以在这个时间在这个城市上车或下车。

最后一个部分有三行：第一行是最早可能的出发时间，第二行是她的出发城市，第三行是目的地。出发城市和目的地总是不同的。

#### 输出

在每组数据的开头先输出一行 “Scenario  $i$ ”， $i$  为从 1 开始的数据编号。

如果有解，则应按样例格式输出两行，包含用 0 补齐的时间戳和位置，并用空格缩进。如果无法在同一天（午夜前）之内到达，输出 “No connection”。

在每组数据的输出后打印一个空行。

#### 样例输入

```
2
3
Hamburg
Frankfurt
Darmstadt
3
2
0949 Hamburg
1006 Frankfurt
```

#### 样例输出

```
Scenario 1
Departure 0949 Hamburg
Arrival 1411 Darmstadt

Scenario 2
No connection
```

2

1325 Hamburg

1550 Darmstadt

2

1205 Frankfurt

1411 Darmstadt

0800

Hamburg

Darmstadt

2

Paris

Tokyo

1

2

0100 Paris

2300 Tokyo

0800

Paris

Tokyo

### 10.5.5 战争 (War)

PC/UVa 题号: 111005/10158, 流行度: B, 通过率: average 难度: 3

一场战争在  $A$  国和  $B$  国间开始了。作为一位  $C$  国的好公民, 你决定为你的国家秘密地参加  $A$  国与  $B$  国间的和谈。和谈中还有  $n$  个人, 但你不知道他们分别属于哪个国家。你可以看到他们互相交谈, 并能通过观察双方在一对一交谈时的表现猜测他们是敌人还是朋友。

你的国家需要知道某些特定的两个人之间的关系: 到底是属于同一国, 还是互相敌对, 因此需要你在和谈期间接收政府传来的问题, 并根据你当前的观察做出回答。

正式的来说, 考虑一个可以进行如下操作的黑盒:

<code>setFriends(<math>x,y</math>)</code>	$x$ 和 $y$ 属于同一国家
<code>setEnemies(<math>x,y</math>)</code>	$x$ 和 $y$ 属于不同国家
<code>areFriends(<math>x,y</math>)</code>	仅当你确信 $x$ 和 $y$ 为朋友时返回 true
<code>areEnemies(<math>x,y</math>)</code>	仅当你确信 $x$ 和 $y$ 为敌人时返回 false

若前两种操作与你现有的结论相冲突, 你应该报错。“朋友”关系 (符号为  $\sim$ ) 和“敌对”关系 (符号为  $*$ ) 具有如下性质:

$\sim$  是等价关系, 即:

1. 若  $x \sim y$  且  $y \sim z$ , 则  $x \sim z$  (朋友的朋友也是朋友)
2. 若  $x \sim y$ , 则  $y \sim x$  (朋友是相互的)
3.  $x \sim x$  (每个人都是他自己的朋友)

\* 满足对称性和反自反性:

1. 若  $x * y$ , 则  $y * x$  (敌对是相互的)
2.  $x * x$  总为假 (没有人和自己敌对)
3. 若  $x * y$ , 且  $y * z$ , 则  $x \sim z$  (敌人的敌人是朋友)
4. 若  $x \sim y$ , 且  $y * z$ , 则  $x * z$  (敌人的朋友是敌人)

操作 `setFriends(x,y)` 和 `setEnemies(x,y)` 不能破坏上述性质。

### 输入

输入第一行单独包含一个整数  $n$ , 表示人数。接下来每一行有三个数,  $c\ x\ y$ , 其中  $c$  为操作编号,

```
c = 1,    setFriends  
c = 2,    setEnemies  
c = 3,    areFriends  
c = 4,    areEnemies
```

$x$  和  $y$  分别为操作的参数, 它们都是区间  $[0, n)$  内的整数, 表示两个不同的人。最后一行为  $0\ 0\ 0$ 。

输入中所有整数由至少一个空格或换行符隔开。最多可有 10 000 个人, 但操作总数没有限制。

### 输出

对于每个 `areFriends` 和 `areEnemies` 操作, 输出 “0” (表示假) 或者 “1”(表示真)。对于每个与当前结论冲突的 `setFriends` 或 `setEnemies` 操作, 你的程序应当输出 “-1”, 但该指令不应对后续操作的执行产生任何影响。成功的 `setFriends` 或 `setEnemies` 不会产生输出。

输出中的相邻整数间应换行。

#### 样例输入

```
10  
1 0 1  
1 1 2  
2 0 5  
3 0 2
```

#### 样例输出

```
1  
0  
1  
0  
0
```

3 8 9	-1
4 1 5	0
4 1 2	
4 8 9	
1 8 9	
1 5 2	
3 5 2	
0 0 0	

### 10.5.6 导游 (Tourist Guide)

PC/UVa 题号: 111006/10199, 流行度: B, 通过率: average 难度: 3

Rio de Janeiro 是一个非常漂亮的城市, 但是这里的景点太多了, 让你觉得晕头转向。幸运的是, 你的朋友 Bruno 答应做你的向导。

可惜 Bruno 是个差劲的司机。他已经交过很多次罚款, 所以渴望避免更多的罚款。因此他想知道哪些地方有监控摄像头, 这样在驶过这些地方的时候会更小心。城市中的摄像头分布是颇有讲究的, 使得每个司机在从一个城区到另一个城区的时候总是无法避开它们。城区  $C$  将安装一个摄像头的充要条件是存在另两个城区  $A$  和  $B$ , 使得从  $A$  到  $B$  必须经过  $C$ 。

比如, 假设有 6 个城区 ( $A, B, C, D, E$  和  $F$ ) 由 7 条双向道路相连:  $B - C, A - B, C - A, D - C, D - E, E - F$  和  $F - C$  都有双向的道路。 $C$  地必须安装摄像头, 因为从  $A$  到  $E$  必须经过  $C$ 。在这个例子中,  $C$  是唯一需要安装摄像头的城区。

现在给你这个城市的地图, 你要通过一个程序来判断哪些城区有摄像头, 以此帮助 Bruno 避免罚款。

#### 输入

输入包含若干城市的地图, 其中每个地图的开头有一个整数  $N(2 < N \leq 100)$ , 代表这个城市中城区的数量。接下来的  $N$  行每行有一个不同城区的名称, 均由不超过 30 个小写字母构成。下面有一个非负整数  $R$ , 代表城市中道路的数量。下面的  $R$  行中, 每行描述一条连接两个城区的道路。

道路描述中的城区名称都是正确的, 并且不会有连接两个相同城区的道路。当  $N$  为 0 时, 输入结束。

#### 输出

对于每份地图, 你需要输出一行:

City map #d: c camera(s) found

其中  $d$  为地图编号 (从 1 开始),  $c$  为摄像头总数。接下来的  $c$  行按字典序输出这些城区的名称。输出的相邻两组数据之间应有一个空行。

**样例输入**

```
6
sugarloaf
maracana
copacabana
ipanema
corcovado
lapa
7
ipanema copacabana
copacabana sugarloaf
ipanema sugarloaf
maracana lapa
sugarloaf maracana
corcovado sugarloaf
lapa corcovado
5
guanabarabay
downtown
botanicgarden
colombo
sambodromo
4
guanabarabay sambodromo
downtown sambodromo
sambodromo botanicgarden
colombo sambodromo
0
```

**样例输出**

```
City map #1: 1 camera(s) found
sugarloaf

City map #2: 1 camera(s) found
sambodromo
```

### 10.5.7 丰盛的晚餐 (The Grand Dinner)

PC/UVa 题号: 111007/10249, 流行度: C, 通过率: high 难度: 4

每支参加今年 ACM/ICPC 世界总决赛的队伍都被邀请参加颁奖仪式后的宴会。为了

使不同参赛队成员更好地交流，同一队的任何两个人都不允许坐在同一桌。

给出每个参赛队的人数(包括选手、教练、替补和观察员)和每张桌子的座位容量,判断是否可能按刚才叙述的规则安排座位。如果存在一种方案,则输出对应的安排方式。如果有多种安排方法,任何一种都可以。

输入

输入包含若干组数据。每组数据的第一行有两个整数  $M$  和  $N$ , 其中  $1 \leq M \leq 70, 1 \leq N \leq 50$ , 分别表示参赛队的数量和桌子的数量; 第二行有  $M$  个整数, 其中第  $i$  个数  $m_i$  表示第  $i$  个参赛队的人数, 每个队不超过 100 人; 第三行有  $N$  个整数, 其中第  $j$  个数  $n_j$  ( $2 \leq n_j \leq 100$ ) 表示第  $j$  张桌子的座位数量。

当  $M$  和  $N$  都为 0 时，输入结束。

输出

对于每组测试数据，输出一行为 1 或者 0，表示是否存在这样的安排方式。如果存在，则在下面输出  $M$  行，其中第  $i$  行中的每个数，表示第  $i$  个参赛队中的成员坐在哪些桌子（桌子从 1 开始编号）。

### 样例输入

4	5			
4	5	3	5	
3	5	2	6	4
4	5			
4	5	3	5	
3	5	2	6	3
0	0			

## 样例输出

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

#### 10.5.8 命题者的难题 (The Problem With the Problem Setter)

PC/UVa 题号: 111008/10092, 流行度: C, 通过率: average 难度: 3

想参加今年程序设计区域赛的学生实在是太多了，因此我们决定举办一场资格赛来选出最有潜力的选手。这场资格赛中最多可以包含 20 种类别的 100 道题。我的任务就是为比赛分配题目。

一开始，我还以为工作非常简单，因为别人已经提供了大约 1000 道已经分好类别的题目。但着手工作后，我发现原作者经常在种类列表中写下多种类别。因为每道题都只能作为一种类型使用，并且每种类型题的数量是给定的，完成这个工作并不简单。

### 输入

输入包含若干组数据，每组数据的第一行有两个整数  $n_k$  和  $n_p$ ，其中  $n_k$  为分类数， $n_p$  为题目数。问题分类数在 2 和 20 之间，而题目总数不超过 1 000。

第二行有  $n_k$  个正整数，其中第  $i$  个整数表示在最终比赛试卷中，第  $i$  类 ( $1 \leq i \leq n_k$ ) 题目所应有的数量。假设所有这  $n_k$  个整数之和不超过 100。接下来的  $n_p$  行中，第  $j$  行 ( $1 \leq j \leq n_p$ ) 为第  $j$  个题目的分类信息。每个题目的分类信息以一个正整数开头，表示这道题所属的类型总数，接下来是具体的类别编号。

当  $n_k$  和  $n_p$  均为 0 时输入结束。

### 输出

对于每组数据，输出一行，表示是否可以从给出的题目中挑选出一些来组成这份试卷，1 代表可以，0 代表不行。

如果有解，用接下来的  $n_k$  行打印出方案，其中第  $i$  行表示所有第  $i$  类题的题目序号。题目序号都是不大于  $n_p$  的正整数，并用单个空格隔开。如果有多组解，任意输出一组即可。

#### 样例输入

```
3 15
3 3 4
2 1 2
1 3
1 3
1 3
1 3
3 1 2 3
2 2 3
2 1 3
1 2
1 2
2 1 2
2 1 3
2 1 2
1 1
3 1 2 3
3 15
7 3 4
```

#### 样例输出

```
1
8 11 12
1 6 7
2 3 4 5
0
```

2 1 2  
1 1  
1 2  
1 2  
1 3  
3 1 2 3 2 2 3  
2 2 3  
1 2  
1 2  
2 2 3  
2 2 3  
2 1 2  
1 1  
3 1 2 3  
0 0

## 10.6 提 示

- 10.1 Richie 要解决的是本章中的哪个问题？
- 10.2 本题可以建模成哈密顿回路还是欧拉回路问题？
- 10.3 如何利用最短路信息来帮助我们决定新消防站的位置？
- 10.4 本题可以建模成最短路问题吗？起点是什么？如何在到达时间最早的前提下求出最晚出发的方案？
- 10.5 可以通过传递闭包来传播对话中隐含的结论吗？
- 10.6 摄像头位置对应于哪一个图论概念？
- 10.7 贪心法可以解决本题吗？或者我们需要用到网络流这样更复杂的算法？
- 10.8 本题可以用网络流解决吗？是否存在更初等的解法？

第11章 动态规划

作为算法设计者和程序员，我们经常需要编程寻找一些问题的最优解。通常很容易得到优秀而可行的解，但要保证程序总是返回绝对意义上的最优解，往往需要对问题进行深入的分析和思考。

动态规划是求解最优化问题的一个非常强大、通用的工具。它对于字符串这样“从左到右排列好”的元素序列尤其有效。一旦学透，动态规划是很好用的，但很多人在尝试理解它时遇到了重重困难。

在看到足够的例子之前，你会觉得动态规划跟变魔术一样。我们建议你首先回顾一下第6章中的二项式系数，琢磨一下我们是如何用“部分解”来逐步递推出我们想要的最终答案的。接下来复习一下10.3.2小节中学过的Floyd这个每对结点间的最短路算法。再然后，就轮到学习本章中的两个例题了。其中第一个题目是动态规划的经典应用，几乎所有的教科书里都有讲解；第二个题目相对比较特别，你需要用动态规划来设计一个全新的算法。

## 11.1 慎用贪心法

很多问题都可以归结为寻找满足特定约束条件的最优解。处理这类问题有很多方法。例如，第 8 章中的回溯法所解决的问题常常要求我们找出最大、最小或者得分最高的方案。回溯法搜索所有可能的解，并从中挑出我们所想要的，因此保证得到正确的答案。但这个方法只适用于很小的问题实例。

很多重要的图论问题都存在正确并且高效的算法，包括第 10 章中讨论过的最短路、最小生成树、匹配等。在自己设计算法之前最好看看需要完成的任务是否可以转化成这些经典问题。如果可以的话，直接套用经典算法即可。

贪心 (*greedy*) 算法在每次决策时做出局部最优选择。例如，求解从  $x$  到  $y$  的最短路的一种容易想到的方法就是从  $x$  出发反复沿着最短边走，直到走到  $y$ 。很直观，可惜是错的！事实上，没有正确性证明的贪心法很有可能是错误的。

那我们该怎么办呢？动态规划为我们提供了一个设计新算法的方式，不仅系统地搜索了所有可能性（从而保证正确性），而且通过保存中间结果来避免重复计算（从而保证效率）。

动态规划建立在递归的基础上，它需要把整个问题的解用若干小规模子问题的解来表示。我们已经介绍过的回溯法和图的深度优先遍历都属于这样的递归过程。

要想让这种递归算法高效执行，程序中必须储存足够的信息以避免重复计算。为什么图的深度优先遍历算法高效？因为算法每访问一个结点后就给它加上标记，防止重复访问。为

什么纯粹的回溯法运算量巨大？因为它搜索了所有可能的路径/解，而不是只考虑那些以前没有考虑过的。

动态规划是一个高效实现递归算法的技巧<sup>1</sup>，它的核心在于保存中间结果。首先需要发现朴素的递归算法一而再再而三地计算着一些相同的子问题，接下来需要把答案改放在表格中而不是重新计算。这样，算法就变得高效了。为了理解下面的例子，你最好先自行寻找一个递归解法。只有在设计出正确算法以后才需要考虑加速的问题。

## 11.2 编辑距离

毫无疑问，在文本串中寻找模式是一个很重要的问题。尽管我们已经在第3章中讨论过模式匹配算法，当时的眼光局限在精确匹配上，即模式串 $s$ 必须一字不差地包含在文本串 $t$ 中。可惜，现实并不那么简单。文本或模式串中的拼写错误让我们不得不找出“相似”而非“相等”的匹配；基因序列或语言文字的演化让我们常常用古老的模式在脑海里搜索：“尔等休要取人性命”变成了“你们别杀人”。

如果我们要处理非精确匹配，首先就应该定义一个代价函数，衡量两个字符串“差得有多远”，即两个字符串之间的距离度量。一个合理的距离度量是把一个字符串变到另一个字符串所需的最小修改次数。有三种常见的修改方式：

- 替换 — 把模式串  $s$  中的一个字符替换成文本串  $t$  中的一个不同字符，例如把 “shot” 变成 “spot”。
  - 插入 — 在模式串  $s$  中插入一个新字符，使得它和  $t$  更相近。例如把 “ago” 变成 “agog”。
  - 删除 — 在模式串  $s$  中删除一个字符，使得它和  $t$  更相近。例如把 “hour” 变成 “our”。

给出一个具体的字符串相似度定义需要为上述三种变换操作分别设置代价。如果每个操作的代价均设为一，则相应的度量称为编辑距离 (*edit distance*)。其他代价值会得到另一些有趣的结果，我们将在 11.4 节中讨论。

如何计算编辑距离？注意到对源串的任何修改方式都可以改写成从左到右的顺序，因此可以基于以下事实设计递归算法：源串的最后一个字符要么直接替换成目标串中的最后一个字符（如果本来就相等则不必操作），要么在源串的最后插入一个字符，要么删除源串的最后一个字符。不管是哪种方式，在这之前都需要递归计算两个更短的字符串的编辑距离<sup>2</sup>。只要递归计算出这些情况各自对应的编辑距离，从中选出最小的一个，就能知道两个串的编辑距离了。由于在任何时候，需要计算的两个字符串都是原来两个字符串的前缀，我们用  $i$  和  $j$  分别表示目前递归计算距离的两个字符串为  $s[1..i]$  和  $t[1..j]$ ：

```
#define MATCH 0 /* enumerated type symbol for match */
```

<sup>1</sup> 译者注：这里特指那些存在大量重复计算的递归。并不是每个递归算法都可以用动态规划高效实现的。

<sup>2</sup> 译者注：以替换为例，若要把  $s$  的前  $i$  个字符变成  $t$  的前  $j$  个字符，只需先把  $s$  的前  $i-1$  个字符变成  $t$  的前  $j-1$  个字符，最后把  $s$  的第  $i$  个字符替换成  $t$  的第  $j$  个字符。

```
#define INSERT      1      /* enumerated type symbol for insert */
#define DELETE       2      /* enumerated type symbol for delete */

int string_compare(char *s, char *t, int i, int j)
{
    int k;                  /* counter */
    int opt[3];             /* cost of the three options */
    int lowest_cost;         /* lowest cost */

    if (i == 0) return(j * indel(' '));
    if (j == 0) return(i * indel(' '));

    opt[MATCH] = string_compare(s,t,i-1,j-1) + match(s[i],t[j]);
    opt[INSERT] = string_compare(s,t,i,j-1) + indel(t[j]);
    opt[DELETE] = string_compare(s,t,i-1,j) + indel(s[i]);

    lowest_cost = opt[MATCH];
    for (k=INSERT; k<=DELETE; k++)
        if (opt[k] < lowest_cost) lowest_cost = opt[k];

    return( lowest_cost );
}
```

这个程序完全正确（请自行确认这一点），但是慢得让人无法忍受。在我们的机器上，即使比较两个只有 11 个字符的字符串也需要好几秒钟，当字符串变长时几乎相当于永远运行不完。

为什么会这么慢？算法的运行时间是指数级别的，因为它有着太多的重复计算。递归函数每次都有三个分支，对于两个长度为  $n$  的字符串，运行时间至少为  $3^n$ ——事实上，这还只是保守估计，因为在大多数情况下，函数调用时  $i$  和  $j$  只会减小其中的一个，而不是同时减小。

那么，应该如何让这个算法变得实用呢？关键是要发现大部分递归调用都在做着重复工作。我们是怎么知道这一点的？由于参数  $s$  和  $t$  从未修改，最多只有  $|s| \cdot |t|$  种不同的  $(i, j)$  对，因此一共只有这么多种本质不同的递归调用<sup>3</sup>。只需把这些  $(i, j)$  对所对应的结果保存在一张表里，我们就能用查表代替重复递归。

<sup>3</sup> 译者注：这个推理成立的前提是函数 `string_compare` 不能有副作用。换句话说，它不能修改全局变量。

下面给出这个算法的实现，它是一种基于表格的动态规划。表格是一个  $|s| \cdot |t|$  的二维矩阵  $m$ ，其中每个单元格代表一个子问题。单元格中同时包含这个子问题的最优解和一个相当于“父亲指针”的变量，表明我们是如何到达这个单元格的：

```
typedef struct {
    int cost;           /* cost of reaching this cell */
    int parent;         /* parent cell */
} cell;

cell m[MAXLEN+1][MAXLEN+1];      /* dynamic programming table */
```

这个算法的动态规划版本和递归版本有三个区别。第一，用查表的方式代替递归调用来得到中间值；第二，对于每个单元格更新 `parent` 域的值，以便构造编辑序列；第三，用更一般的 `goal_cell()` 函数，而不是仅仅返回 `m[|s|][|t|].cost`。这使得我们可以将这个程序用到更多题目中。

注意我们使用字符串的方式比较特别。事实上，我们假定字符串的前面多了一个空格字符，因此 `s` 的第一个真实字符是 `s[1]`。读入时需要这样：

```
s[0] = t[0] = ' ';
scanf("%s", &(s[1]));
scanf("%s", &(t[1]));
```

为什么要这么做？这样我们让矩阵 `m` 和字符串的索引保持一致，让代码更清楚——别忘了我们必须把 `m` 的第 0 行和第 0 列留出来保存和空串匹配的边界值。另一个方案是用通常的处理方法保存字符串，但是在使用下标时进行调整。

```
int string_compare(char *s, char *t)
{
    int i, j, k;           /* counters */
    int opt[3];            /* cost of the three options */

    for (i=0; i<MAXLEN; i++) {
        row_init(i);
        column_init(i);
    }

    for (i=1; i<strlen(s); i++)
```

```

    for (j=1; j<strlen(t); j++) {
        opt[MATCH] = m[i-1][j-1].cost + match(s[i],t[j]);
        opt[INSERT] = m[i][j-1].cost + indel(t[j]);
        opt[DELETE] = m[i-1][j].cost + indel(s[i]);

        m[i][j].cost = opt[MATCH];
        m[i][j].parent = MATCH;
        for (k=INSERT; k<=DELETE; k++)
            if (opt[k] < m[i][j].cost) {
                m[i][j].cost = opt[k];
                m[i][j].parent = k;
            }
    }

    goal_cell(s,t,&i,&j);
    return( m[i][j].cost );
}

```

请注意矩阵的填充顺序。在计算  $(i, j)$  的值之前，我们需要知道另外三个元素的值： $(i - 1, j - 1)$ 、 $(i, j - 1)$  和  $(i - 1, j)$ 。满足这点的任何计算顺序都可以，包括上述代码中使用的行优先顺序。

下面是一个例子，展示了从“thou shalt not”到“you should not”<sup>4</sup> 的五个步骤中每步的 cost 和 parent：

	y	o	u	-	s	h	o	u	l	d	-	n	o	t	
:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
t:	1	1	2	3	4	5	6	7	8	9	10	11	12	13	
h:	2	2	2	3	4	5	5	6	7	8	9	10	11	12	13
o:	3	3	2	3	4	5	6	5	6	7	8	9	10	11	12
u:	4	4	3	2	3	4	5	6	5	6	7	8	9	10	
-:	5	5	4	3	2	3	4	5	6	6	7	7	8	9	10
s:	6	6	5	4	3	2	3	4	5	6	7	8	8	9	10
h:	7	7	6	5	4	3	2	3	4	5	6	7	8	9	10
a:	8	8	7	6	5	4	3	3	4	5	6	7	8	9	10
l:	9	9	8	7	6	5	4	4	4	5	6	7	8	9	
t:	10	10	9	8	7	6	5	5	5	5	6	7	8	8	
-:	11	11	10	9	8	7	6	6	6	6	5	6	7	8	
n:	12	12	11	10	9	8	7	7	7	7	6	5	6	7	
o:	13	13	12	11	10	9	8	7	8	8	8	7	6	5	6
t:	14	14	13	12	11	10	9	8	8	9	9	8	7	6	5

	y	o	u	-	s	h	o	u	l	d	-	n	o	t	
:	-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
t:	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
h:	2	0	0	0	0	0	0	0	1	1	1	1	1	1	1
o:	2	0	0	0	0	0	0	0	1	1	1	1	1	0	1
u:	2	0	2	0	1	1	1	1	0	1	1	1	1	1	1
-:	2	0	2	2	0	1	1	1	1	0	0	0	1	1	1
s:	2	0	2	2	2	0	1	1	1	0	0	0	0	0	0
h:	2	0	2	2	2	2	0	1	1	1	1	1	1	0	0
a:	2	0	2	2	2	2	2	0	0	0	0	0	0	0	0
l:	2	0	2	2	2	2	2	2	0	0	0	1	1	1	1
t:	2	0	2	2	2	2	2	2	0	0	0	0	0	0	0
-:	2	0	2	2	0	2	2	2	0	0	0	0	0	1	1
n:	2	0	2	2	2	2	2	2	0	0	0	0	2	0	1
o:	2	0	0	2	2	2	2	0	0	0	0	2	2	0	1
t:	2	0	2	2	2	2	2	2	0	0	0	2	2	2	0

<sup>4</sup> 译者注：都是“你不该”的意思。前者是古英语，后者是现代英语。

上面的动态规划实现返回了最优解的代价，但没有给出解本身。知道从“thou shalt not”到“you should not”只需五个步骤倒是不错，但到底是哪五个步骤呢？

一个动态规划问题的可行解可以用动态规划矩阵中从初始状态（两个空串 $(0, 0)$ ）到目标状态（两个完整串 $(|s|, |t|)$ ）的路径来描述。构造解的关键是重建最优路径上每一步的决策。这些决策已经储存在了每个单元格的 parent 域中。

重建这些决策的方式是从目标状态出发，顺着 parent 指针往回走，直到回到初始单元格。 $m[i, j]$  的 parent 域告诉我们 $(i, j)$  处的操作是 MATCH(匹配)、INSERT(插入) 还是 DELETE(删除)。从“thou-shalt-not”到“you-should-not”的操作序列为 DSMMMMMISMSM~~MM~~——表示删除第一个“t”，把“h”替换成“y”，接下来的五个字符不变，然后插入一个“o”，把“a”替换成“u”，最后把“t”替换成“d”。

往回走意味着重建出的解也是逆序的。不过递归可以帮助我们把解反向：

```
reconstruct_path(char *s, char *t, int i, int j)
{
    if (m[i][j].parent == -1) return;

    if (m[i][j].parent == MATCH) {
        reconstruct_path(s, t, i-1, j-1);
        match_out(s, t, i, j);
        return;
    }

    if (m[i][j].parent == INSERT) {
        reconstruct_path(s, t, i, j-1);
        insert_out(t, j);
        return;
    }

    if (m[i][j].parent == DELETE) {
        reconstruct_path(s, t, i-1, j);
        delete_out(s, i);
        return;
    }
}
```

包括编辑距离在内的很多问题，不需要 `parent` 数组也能重建解。诀窍在于：回过头去，看看从哪个单元格过来的代价和最优解一致。

## 11.4 编辑距离的变种

上面给出的最优化和解重建过程引用了一些尚未定义的函数。它们可以分为以下几类：

- 表格的初始化 — 函数 `row_init()` 和 `column_init()` 分别初始化动态规划表格的第 0 行和第 0 列。对于字符串编辑距离问题，单元格  $(i, 0)$  和  $(0, i)$  对应于其中一个串取前  $i$  个字符，另一个串为空的子问题。这些子问题是易于求解的： $(i, 0)$  或  $(0, i)$  都是恰好需要  $i$  次插入/删除。

```
row_init(int i)           column_init(int i)
{
    m[0][i].cost = i;
    if (i>0)
        m[0][i].parent = INSERT;
    else
        m[0][i].parent = -1;
}
{
    m[i][0].cost = i;
    if (i>0)
        m[i][0].parent = DELETE;
    else
        m[0][i].parent = -1;
}
```

- 罚款值 — 函数 `match(c, d)` 和 `indel(c)` 分别提供了把字符  $c$  替换成  $d$  的代价以及插入/删除字符  $c$  的代价。对于标准的编辑距离问题，`indel` 总是返回 1，而 `match` 的计算方式为：两个字符相等时代价为 0，否则为 1。更细致的代价函数也是存在的，例如当替换前后的两个字符在标准键盘上位置相近，或者读音/外形相近，则替换代价比正常情况要小。

```
int match(char c, char d)      int indel(char c)
{
    if (c == d) return(0);
    else return(1);
}
```

- 目标格 — 函数 `goal_cell` 返回最优解所对应的单元格编号。对于编辑距离，单元格可由两个输入串各自的长度确定，但对于其他应用，目标单元格并不能事先确定。

```
goal_cell(char *s, char *t, int *i, int *j)
{
    *i = strlen(s) - 1;
    *j = strlen(t) - 1;
```

}

- 追溯动作 — 函数 `match_out`、`insert_out` 和 `delete_out` 在追溯的过程中执行相应的编辑动作。在编辑距离问题中，这些动作可能是打印出操作的名称或者涉及的字符，具体采取哪种方案取决于应用本身的需求。

```

insert_out(char *t, int j)           match_out(char *s, char *t,
{                                     int i, int j)
    printf("I");
}
                                         if (s[i]==t[j]) printf("M");
                                         else printf("S");
delete_out(char *s, int i)          }
{
    printf("D");
}

```

对于我们的编辑距离问题来说，上述函数都很简单。但是，我们必须承认，编写正确的边界条件和下标操作是有一定难度的。尽管一旦懂得了动态规划的精髓，算法设计和编码都不是坏事，但要确保所有细节都正确无误，缜密的思考和充分的测试是必不可少的。

初看起来，把如此简单的算法分解成这么多基础模块有点小题大做。但我们很快就能看到，这样做可以让好几种重要的问题变成编辑距离的特殊情况，只需在部分模块中做出少量改动即可解决它们<sup>5</sup>。

- 子串匹配 — 假设我们想知道一个较短的模板 `s` 在一个长文本 `t` 中的何处以最接近的形式出现。例如当模板为“Skiena”时，字符串 `Skienna`, `Skenna`, `Skina`, ... 都算是它的接近形式。直接使用编辑距离的效果差强人意，原因是编辑的代价大都花在了“删除文本中多余的字符”上。

我们希望在文本中匹配之前的多余字符不计代价，同样的，目标状态也不一定是两个串同时终止的情况，而应该是模板串终止时代价最小的位置。修改相应的两个函数就得到了以下的正确解法：

```

row_init(int i)
{
    m[0][i].cost = 0;          /* note change */
    m[0][i].parent = -1;       /* note change */
}

goal_cell(char *s, char *t, int *i, int *j)

```

<sup>5</sup> 译者注：但请注意，用这样的方式解决那些问题不一定是渐进意义下最优的。

```
{  
    int k; /* counter */  
  
    *i = strlen(s) - 1;  
    *j = 0;  
    for (k=1; k<strlen(t); k++)  
        if (m[*i][k].cost < m[*i][*j].cost) *j = k;  
}
```

- 最长公共子序列 — 我们常常需要找到两个单词之间最长的、不一定连续的公共字符序列。“democrat” 和 “republican”的最长公共子序列 (*longest common subsequence, LCS*) 是 *eca*。

公共子序列是在编辑序列中所有保持不变的字符序列。要让这个序列最长，须应当禁止不同字符的替换操作。这可以通过修改匹配代价函数实现：

```
int match(char c, char d)  
{  
    if (c == d) return(0);  
    else return(MAXLEN);  
}
```

事实上，只需把替换操作的代价设置为比一次插入加一次删除还要大。这样，最优编辑序列就一定不会包含任何替换操作了。

- 最长上升子序列 — 如果一个序列的第  $i$  个元素总是不小于第  $i - 1$  个元素，称该序列为单调递增 (*monotonically increasing*) 的。最长上升子序列 (*longest increasing subsequence*) 问题要求从输入串  $S$  中删除尽量少的元素，使得剩下的子序列是单调递增的。因此，“243517698”的一个最长上升子序列是“23568”。

事实上，它可以转化成最长公共子序列问题，其中第二个字符串是  $S$  中元素按照从小到大的顺序排列而成。二者的任意公共序列保证 (a) 在  $S$  中递增，并且 (b) 各字符在原串中的位置递增。因此其中最长的序列即为所求。当然，这个方法还能求出最长下降子序列，只需在排序时改为从大到小即可。

正如你所看到的，如此简单的编辑距离程序可以方便地完成不少令人吃惊的事情。诀窍在于：把你所要解决的问题转化成近似串匹配问题的特例。

## 11.5 程序设计举例：电梯优化

我在一幢摩天大厦里工作，楼里的电梯慢得像蜗牛。最要命的是，每当我从一楼乘电梯

到顶楼时，总会有人按下一连串的按钮（比如 13 楼、14 楼、15 楼），使得电梯连停三次。其实只要大家都同意，可以只按 14 楼。去 13 楼和 15 楼的人们也在 14 楼出电梯，走一层楼并没有什么大不了的。

你的任务是写一个电梯优化程序。所有人事先输入他们各自的目标楼层（若有多人的目标楼层相同，人数也将被统计），然后电梯自主决定它会在哪些楼层停下。电梯最多只能停  $k$  次，目标是最小化所有人步行的楼层总和。

假设往上步行和往下步行的代价相同——就当锻炼身体了。基于这一点，当有多种方案同时使得步行楼层总和最小时，电梯停靠的位置应该尽量低，因为电力消耗比较小。注意电梯停靠的楼层不一定是某人的目标楼层。假设一共有两人，分别去 27 楼和 29 楼，电梯可以停在 28 楼。

---

#### 以下为题解

---

这是一个用动态规划巧妙地解决实际问题的例子。如何得知它能用动态规划解决？得知以后，又如何具体解决问题？

前面曾经讲过，动态规划基于递归。第  $k$  次停留的最优位置取决于前  $k - 1$  次停留的所有方案的代价。如果你能告诉我所有相关部分解的最优代价，我就能对最后一次停留做出正确的决策。

高效的动态规划算法往往要求输入有序（*ordered*）。所有人的目标楼层可以从低到高排序，这对于动态规划来说是十分重要的。考虑在  $f_1$  楼第一次停留，然后在  $f_2$  楼第二次停留的方案。对于那些目标楼层不比  $f_1$  高的人来说，第二次停在哪里都无所谓。换句话说，整个问题可以分解成若干部分。如果我们需要在  $f_2$  楼上加上第三次停留，则此次停留的最优位置和  $f_1$  无关。

到这里，我们已经嗅到了动态规划的气味。整个算法是怎样的？我们需要为“部分解”下一个定义，以便为规模更大的问题做出决策。这个定义如何——

$m[i][j]$  表示电梯恰好停  $j$  次，最后一次停在  $i$  楼时，所有人步行总层数的最小值。

这个定义能用来计算第  $j + 1$  次停留的位置吗？答案是肯定的。根据定义，这第  $j + 1$  次停留的楼层编号必须比第  $j$  次停留的楼层编号（即  $i$ ）大。不难看出，这次新的停留只影响到那些目标楼层超过  $i$  的人。为了具体计算新的停留对这群人有着怎样的影响，我们需要把目标楼层在  $i$  和新停留的楼层之间的人根据他们离哪层楼近分成两部分。对应的递推关系如下：

$$m_{i,j+1} = \min_{k=0}^i (m_{k,j} - \text{floors\_walked}(k, \infty) + \text{floors\_walked}(k, i) + \text{floors\_walked}(i, \infty))$$

这个递推关系是什么意思？如果最后一次停留在第  $i$  楼，前一次停留的楼层  $k$  一定满足  $k < i$ 。这样的方案代价是多少？我们必须从  $m_{k,j}$  中减去目标楼层超过  $k$  的人们的步行层数和（即  $\text{floors\_walked}(k, \infty)$ ），并代之以在第  $i$  楼处停留时（暂时）为这些人节省的开销（即

`floors_walked(k, i) + floors_walked(i,  $\infty$ )`。

关键在于函数 `floors_walked(a, b)`，它的含义是计算目标楼层在电梯的两次停留  $a$  和  $b$  之间的那些人步行的层数之和。这其中的每个人都从  $a$  和  $b$  中离目标楼层较近的一层楼下电梯，然后步行到目的地：

```
int floors_walked(int previous, int current) {
    int nsteps=0;           /* total distance traveled */
    int i;                  /* counter */

    for (i=1; i<=nriders; i++)
        if ((stops[i] > previous) && (stops[i] <= current))
            nsteps += min(stops[i]-previous, current-stops[i]);

    return(nsteps);
}
```

当你认同上述逻辑之后，算法的实现是相当简单的。我们用两个独立的全局矩阵来保存代价和父结点指针：

```
#define NFLOORS      110      /* the building height in floors */
#define MAX_RIDERS    50       /* what is the elevator capacity? */

int stops[MAX_RIDERS];          /* what floor does everyone get off? */
int nriders;                   /* number of riders */
int nstops;                     /* number of allowable stops */

int m[NFLOORS+1][MAX_RIDERS];   /* dynamic programming cost table */
int p[NFLOORS+1][MAX_RIDERS];   /* dynamic programming parent table */
```

优化函数是递推关系的直接实现。唯一需要注意的是循环的顺序，应当保证所有值在被使用之前都已经算好：

```
int optimize_floors()
{
    int i,j,k;           /* counters */
    int cost;             /* costs placeholder */
    int laststop;         /* the elevator's last stop */
```

```

for (i=0; i<=NFLOORS; i++) {
    m[i][0] = floors_walked(0,MAXINT);
    p[i][0] = -1;
}

for (j=1; j<=nstops; j++)
    for (i=0; i<=NFLOORS; i++) {
        m[i][j] = MAXINT;
        for (k=0; k<=i; k++) {
            cost = m[k][j-1] - floors_walked(k,MAXINT) +
                floors_walked(k,i) + floors_walked(i,MAXINT);
            if (cost < m[i][j]) {
                m[i][j] = cost;
                p[i][j] = k;
            }
        }
    }

laststop = 0;
for (i=1; i<=NFLOORS; i++)
    if (m[i][nstops] < m[laststop][nstops])
        laststop = i;

return(laststop);
}

```

最后，让我们来重建解。方法和上一个例子完全相同：顺着父指针不断往回走：

```

reconstruct_path(int lastfloor, int stops_to_go)
{
    if (stops_to_go > 1)
        reconstruct_path(p[lastfloor][stops_to_go], stops_to_go-1);

    printf("%d\n", lastfloor);
}

```

我们用上述程序计算一个十层欧洲建筑（地面被看成是“零楼”）的电梯调度方案，其中

1 楼到 10 楼中每层楼都恰好有一个人想去。如果电梯只能停一次，应该停到 7 楼，总开销是 18(去往 1、2、3 楼的旅客不坐电梯，直接上楼)。如果可以停两次，应该停到 3 楼和 8 楼，总开销是 11。如果可以停三次，则应该停到 3 楼、6 楼和 9 楼，总开销为 7。

## 11.6 习题

### 11.6.1 越大越聪明? (Is Bigger Smarter?)

PC/UVa 题号: 111101/10131, 流行度: B, 通过率: high 难度: 2

一些人认为，大象的体型越大，脑子越聪明。为了反驳这一错误观点，你想要分析一组大象的数据，找出尽量多的大象组成一个体重严格递增但 IQ 严格递减的序列。

#### 输入

输入包含若干大象的数据，每行一头大象，直到输入结束。每头大象的数据包括两个整数：第一个是以千克为单位的体重，第二个是以整百为单位的 IQ 指数。两个整数均在 1 到 10000 之间。输入最多包含 1000 头大象。两头大象可能有相同的体重，或者相同的 IQ，甚至体重和 IQ 都相同。

#### 输出

输出第一行应当包括一个整数  $n$ ，为找到的大象序列的长度。接下来  $n$  行，每行包含一个正整数，表示一头大象。用  $W[i]$  和  $S[i]$  表示输入数据中第  $i$  行的两个数，则若找到的这一序列为  $a[1], a[2], \dots, a[n]$ ，则必须有：

$$W[a[1]] < W[a[2]] < \dots < W[a[n]] \text{ 和 } S[a[1]] > S[a[2]] > \dots > S[a[n]]$$

这里的  $n$  应当是最大可能值。所有不等关系均为严格不相等：体重严格递增，而 IQ 严格递减。

如果存在多组解，你的程序只需输出任何一个解。

#### 样例输入

6008 1300  
6000 2100  
500 2000  
1000 4000  
1100 3000

#### 样例输出

4  
4  
5  
9  
7

6000 2000  
8000 1400  
6000 1200  
2000 1900

www.pku.edu.cn

### 11.6.2 不同的子序列 (Distinct Subsequences)

PC/UVa 题号: 111102/10069, 流行度: B, 通过率: average 难度: 3

序列  $S$  的一个子序列为删掉  $S$  中的零个或几个元素后的序列。严格地说: 对于序列  $Z = z_1 z_2 \dots z_k$  和  $X = x_1 x_2 \dots x_m$ , 当且仅当存在一个严格递增的  $X$  下标序列  $\langle i_1, i_2, \dots, i_k \rangle$ , 使得对于所有的  $j = 1, 2, \dots, k$ , 均有  $x_{i_j} = z_j$ , 称  $Z$  为  $X$  的一个子序列。例如,  $Z = bcd$  就是  $X = abcdbab$  的一个子序列, 对应的下标序列为  $\langle 2, 3, 5, 7 \rangle$ 。

你的任务是写一个程序, 统计  $Z$  在  $X$  中作为不同的子序列 (即对应不同的下标序列) 出现了多少次。

#### 输入

输入第一行包含一个数  $N$ , 表示测试数据的个数。每组数据的第一行包含一个字符串  $X$ , 完全由小写英文字母组成并且长度不超过 10000。接下来的一行为另一个字符串  $Z$ , 长度不超过 100, 同样由小写英文字母组成。保证  $Z$  或它的任意前缀、后缀在  $X$  中出现的次数均不超过  $10^{100}$ 。

#### 输出

对于每组数据, 输出单独的一行, 表示  $Z$  在  $X$  中出现的次数。

#### 样例输入

2  
babgbag  
bag  
rabbbit  
rabbit

#### 样例输出

5  
3

### 11.6.3 重量和力量 (Weights and Measures)

PC/UVa 题号: 111103/10154, 流行度: C, 通过率: average 难度: 3

一只叫 Mack 的乌龟, 为了不被压碎, 听取了你关于“应按何种顺序堆叠乌龟来建造 Yertle 王的宝座”的建议。有 5607 只乌龟听从 Yertle 王号令, 它们的体重和力量各异。你的任务是将乌龟们堆得尽可能高。

**输入**

输入包含多行，每行包含由一个或多个空格隔开的两个整数，表示一只乌龟的体重和力量。体重单位为克。力量表示这只乌龟能够承受的包括自身在内的最大重量，单位也为克。即一只体重 300 克，力量 1000 克的乌龟可以承受 700 克的乌龟踩在它上面。最多有 5607 只乌龟。

**输出**

输出单独一个数，表示在每只乌龟的承受范围内可以堆叠的最大乌龟数。

**样例输入**

```
300 1000
1000 1200
200 600
100 101
```

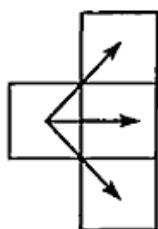
**样例输出**

```
3
```

**11.6.4 单向 TSP(Unidirectional TSP)**

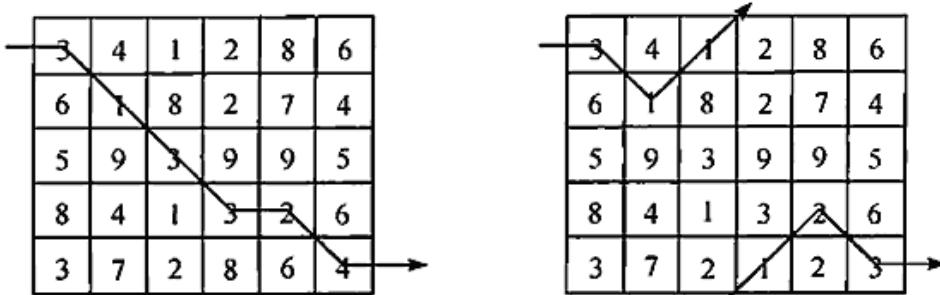
PC/UVa 题号: 111104/116, 流行度: A, 通过率: low 难度: 3

给定一个  $m \times n$  的数字矩阵，你需要写一个程序计算一条从左到右走过矩阵且权和最小的路径。一条路径可以从第 1 列的任意位置出发，到达第  $n$  列的任意位置。每一步为从第  $i$  列走到第  $i + 1$  列的相邻行（水平移动或者沿 45 度斜线移动）。第一行和最后一行看作是相邻的，即你应当把这个矩阵看成是一个卷起来的圆筒。如下为合法的走法：



路径的权和为所有经过的  $n$  个方格中整数的和。

两个略有不同的  $5 \times 6$  矩阵的最小路径如下。只有矩阵中最下面一行的数不同。右边矩阵的路径利用了第一行与最后一行相邻的性质。



## 输入

输入包含多个矩阵。每个矩阵的第一行为两个数  $m$  和  $n$ , 分别表示矩阵的行数和列数。接下来的  $m \cdot n$  个整数按行优先的顺序排列, 即前  $n$  个数组成第一行, 接下来  $n$  个数组成第二行, 以此类推。相邻整数间用一个或多个空格隔开。注意: 这些数不一定是正数。输入中可能有一个或多个矩阵描述, 直到输入结束。

每个矩阵的行数均在 1 到 10 之间, 列数在 1 到 100 之间。路径的权和不会超过 30 个比特。

## 输出

对每个矩阵输出两行。第一行为最小权和的路径, 第二行为该路径的权和。路径由  $n$  个整数组成 (相邻整数间用一个或多个空格隔开), 表示路径经过的行号。如果权和最小的路径不止一条, 输出字典序最小的一条。

### 样例输入

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

### 样例输出

```
1 2 3 4 4 5
16
1 2 1 5 4 5
11
1 1
19
```

## 11.6.5 切割小木棍 (Cutting Sticks)

PC/UVa 题号: 111105/10003, 流行度: B, 通过率: average 难度: 2

你想把一根木棍切成几截。最合适公司“相似体切割机构”(Analog Cutting Machinery, 缩写 ACM) 根据被切割的木棍长度收费。他们使用的切割锯一次只能切一下。

很容易知道, 不同的切割顺序使得收费不同。例如, 对于一根长 10 米的棍子, 需要在离一端 2 米, 4 米和 7 米的位置切割, 有几种选择。可以第一次在 2 米处切割, 接下来在 4 米

处切割，然后在 7 米处切割。这样的价格是  $10 + 8 + 6 = 24$ ，因为一开始时木棍长 10 米，切割后长 8 米，再次切割后长 6 米。另一种方法是在 4 米处切割，然后 2 米，然后 7 米。这样的价格是  $10 + 4 + 6 = 20$ ，比刚才的方案便宜。

你的老板要求你写一个程序，找出对于任意给定棍子可能的最小切割费。

### 输入

输入包含多组数据。每组数据的第一行包含一个正整数  $l$ ，表示需要切割的棍子的长度。 $l < 1000$ 。接下来的一行包含需要切割的次数  $n$  ( $n < 50$ )。

下面一行包含  $n$  个正整数  $c_i$  ( $0 < c_i < l$ )，表示需要切割的位置，按严格递增序给出。

输入数据中  $l = 0$  表示输入结束。

### 输出

按样例格式输出切割棍子的最小花费。

#### 样例输入

```
100
3
25 50 75
10
4
4 5 7 8
0
```

#### 样例输出

```
The minimum cutting is 200.
The minimum cutting is 22.
```

### 11.6.6 渡船装载 (Ferry Loading)

PC/UVa 题号: 111106/10261, 流行度: B, 通过率: low 难度: 3

渡轮被用来在河流或其他水域间运输汽车。一般来说，渡轮宽到足以容纳两条贯穿整个船体的汽车车道。两列汽车从渡轮一头驶入，等到渡轮到港后再从另一头驶出。

等待登船的汽车排成一列，工作人员引导每辆车驶向左舷或者右舷的车道，使它们平衡。队列中每辆车的长度不同，工作人员先检查队列并估计每辆车的长度。在此基础上，决定哪辆车上哪边的车道，并在船身长度的限制下，尽可能使更多的车能够登船。写一个程序确定车辆登船的方案使得能够登船的车尽可能多。

### 输入

输入数据以单独一行一个正整数开头，表示测试数据的组数。接下来是一个空行。

每组数据的第一行包含一个 1 到 100 之间的整数，即渡轮的长度（单位：米）。接下来队列中的每辆车占一行，用一个 100 到 3000 之间（含 100 和 3000）的整数描述车的长度（单

位：厘米）。每组数据最后一行为一个数 0。汽车必须按顺序登船，并且船任何一侧车道上汽车的总长度不能超过船身的长度。登船的汽车要尽可能的多，从队列中的第一辆车开始顺次登船直到下一辆车无法登船。

相邻两组输入数据间有一个空行。

## 输出

对于每组数据，输出的第一行为渡轮可以装载的汽车数量。接下来对于每辆可以登船的汽车，按照出现在输入数据中的顺序，输出一行“port”表示该车驶向左舷车道，“starboard”表示该车驶向右舷车道。如果有多种满足条件的安排方式，输出任意一种。

相邻两组数据的输出间用一个空行隔开。

### 样例输入

```
1
50
2500
3000
1000
1000
1500
700
800
0
```

### 样例输出

```
6
port
starboard
starboard
starboard
port
port
```

## 11.6.7 筷子 (Chopsticks)

PC/UVa 题号: 111107/10271, 流行度: B, 通过率: average 难度: 3

在中国，人们通常用一双筷子吃饭。但是 L 先生与众不同：他用三支筷子吃饭。其中的一支长筷子用来叉取大块的食物，而剩下两支用作普通筷子。两支普通筷子的长度应尽可能的接近，而长的那支只要是三支中最长的就可以了。对于一副筷子的长度  $A, B, C (A \leq B \leq C)$ ，定义函数  $(A - B)^2$  为这副筷子的“难用度”。

L 先生邀请  $K$  个人参加他的生日聚会，并且想将他用筷子的方法介绍给其他人。他需要准备  $K + 8$  副这样的筷子（给他自己、妻子、儿子、女儿、父亲、母亲、岳父、岳母以及  $K$  名客人）。但是 L 先生的筷子有很多不同的长度！给出所有筷子的长度，他需要找到一种组合  $K + 8$  副筷子的方法，使得难用度的总和最小。

输入

输入第一行为一个单独的整数  $T$ , 表示测试数据的组数 ( $1 \leq T \leq 20$ )。每组数据开头两个整数  $K$  和  $N$  ( $0 \leq K \leq 1000, 3K + 24 \leq N \leq 5000$ ) 为宾客的总数和筷子的支数。接下来  $N$  个正整数  $L_i$ , 按不下降顺序给出筷子的长度 ( $1 \leq L_i \leq 32000$ )。

输出

对于每组数据，输出一行包含最小的难度总和。

### 样例输入

1  
1 40  
1 8 10 16 19 22 27 33 36 40 47 52 56 61 63 71 72 75 81 81 84 88 96 98  
103 110 113 118 124 128 129 134 134 139 148 157 157 160 162 164

## 样例输出

23

注：样例输入所对应的一种最优组合方案是 (8, 10, 16), (19, 22, 27), (61, 63, 75), (71, 72, 88), (81, 81, 84), (96, 98, 103), (128, 129, 148), (134, 134, 139) 和 (157, 157, 160)。

#### 11.6.8 搬家大冒险：第四部 (Adventures in Moving: Part IV)

PC/UVa 题号: 111108/10201, 流行度: A, 通过率: low 难度: 3

你正在考虑租用一辆搬家卡车来帮你从滑铁卢搬到一个大城市。鉴于当今飞涨的油价，你想知道这个大家伙要吃掉你多少油。

这辆卡车每走一公里消耗一公升的汽油。油箱的容量为 200 公升。当你从滑铁卢租到它的时候，油箱是半满的。当你到达大城市去还卡车的时候，油箱必须至少是半满，否则租赁公司会敲诈你更多的油费。你希望尽可能少花油费，但你也不想在中途因为没油而抛锚。

输入

输入开头为单独的一行，包含一个正整数，表示测试数据的组数，紧跟着是一个空行。

每组数据包含若干个整数。第一个整数为滑铁卢到大城市的距离 (单位: 公里), 不超过 10000。接下来为至多 100 个加油站的信息, 描述途中的加油站情况, 按照距离由近到远排列。每个加油站包括离滑铁卢的距离 (单位: 公里) 以及每公升汽油的价格 (单位: 10 美分), 不超过 2000。

相邻两组数据之间用一个空行隔开。

**输出**

对于每组数据，输出从滑铁卢到大城市最少需要花费的油费。如果在上述的限制下无法从滑铁卢到大城市，输出“Impossible”。

相邻两组数据的输出之间用一个空行隔开。

**样例输入**

1  
500  
100 999  
150 888  
200 777  
300 999  
400 1009  
450 1019  
500 1399

**样例输出**

450550

## 11.7 提 示

- 11.1 本题是否可以归约到某种形式的字符串匹配问题？
- 11.3 输入的原始顺序有用吗？或者说，我们可以随意重排它们？如果答案是肯定的，怎样的重排是最有用的？
- 11.4 为了选出最优的最后一步，需要知道关于路径前面部分的哪些信息？
- 11.5 是否可以根据“每次切割将得到两根更小的木棍”来构造一个递归算法？
- 11.6 是否可以总是把下一辆车塞到空间最大的车道里？试说明原因。是否可以充分利用“两个车道里的车身总长均为整数”这一条件？
- 11.7 如果没有第三支筷子，如何解决本题？
- 11.8 为了选出最优的最后一步，需要知道“到达特定地点并拥有特定剩余油量”的费用的哪些信息？

## 11.8 注 解

- 11.3 关于 Yertle 龟王的更多讨论参见 [Seu58]。

## 第12章 网 格

不是极坐标太复杂，而是笛卡尔坐标没有达到它有权拥有的复杂性。

——[美]Kleppner, [美]Kolenkow, “An Introduction to Mechanics”<sup>1</sup>

大量的自然结构都离不开网格。棋盘是网格，典型的城区往往也是网格。事实上，网格上最自然的距离度量常常被称为“曼哈顿”距离。经纬度系统定义了地球表面上的网格，尽管这个表面不是平面，而是球面。

网格无处不在，因为它是把空间分成若干区域，以方便定位的最自然的方式。在极限意义下，每个“单元格”变成了单独的点，但本章只讨论较粗糙的网格，它们的单元格都是有形状的。在正规网格 (*regular grids*) 中，每个单元格的形状是全等的，并且以一种规则的图案排列。由于简单，矩形网格是最常见的网格，但请不要忽略基于三角形的六边形网格 (*hexagonal grids*) 的重要性。事实上，勤劳的蜜蜂日复一日、年复一年向人们所展现着的，正是六边形蜂房的高效。

### 12.1 矩形网格

用过绘图纸的读者一定不会对矩形网格陌生。在这样的网格中，单元格通常由等间距的横线和竖线所围成。尽管不等间距的网格中单元格大小有所差异，但单元格之间的拓扑关系依然不变。三维网格由等间距的层（每层都是平面网格）通过垂直于每层的垂线连接而成，三维网格中有“面”的概念，由每对“面相邻”的单位立方体定义。

平面网格有三个重要的组成部分：顶点、边和单元格的内部。有时，我们关心的是单元格的内部，比如在一些几何应用中，每个单元格描述的是的一个区域；有时，我们关心的是网格的顶点，比如在棋盘上定位棋子；又有时，我们关心的是网格的边，例如在一个充满建筑物（占据单元格的内部）的城市中寻找一条路径。

除了边界上的之外，平面网格中的每个顶点恰好与四条边、四个单元格的内部区域接触。三维空间里，每个内部点恰好接触六条边和八个单元格。在  $d$  维空间中，每个内部点恰好接触  $2d$  条边和  $2^d$  个单元格。平面网格中的内部单元格接触八个其他单元格，其中四个以顶点相连，另外四个以边相连。在三维空间里，每个内部单元格接触 26 个其他单元格，其中 6 个以面相连，12 个以边相连，8 个以顶点相连。

<sup>1</sup> 译者注：该书中译本《力学引论》于 1980 年由人民教育出版社出版，译者宁元源、刘爱晖。

### 12.1.1 遍历

我们常常需要遍历  $n \times m$  网格中的所有单元格。任何一种遍历方式都可以看作是从  $nm$  种有序数对到 1 和  $nm$  之间唯一整数的映射。对于一些特定的场合，遍历的顺序至关重要，例如动态规划中的求值顺序。最重要的遍历方式包括 —

- 行优先 — 把矩阵分成若干行，其中前  $m$  个遍历的元素属于第一行，接下来的  $m$  个元素属于第二行，以此类推。大多数现代的程序设计语言都采用这种方式用一段连续内存单元表示二维矩阵。

```
(1,1)      row_major(int n, int m)
(1,2)      {
(1,3)          int i,j;           /* counters */
(2,1)
(2,2)          for (i=1; i<=n; i++)
(2,3)              for (j=1; j<=m; j++)
(3,1)                  process(i,j);
(3,2)      }
(3,3)
```

- 列优先 — 把矩阵分成若干列，其中前  $m$  个遍历的元素属于第一列，接下来的  $m$  个元素属于第二列，以此类推。这只需要交换行优先遍历法中循环的嵌套次序。当你需要优化缓存 (cache) 性能，或者通过指针运算访问特定元素时，编译器到底使用行优先顺序还是列优先顺序表示矩阵就变得重要了。

```
column_major(int n, int m)
(1,1)      {
(2,1)          int i,j;           /* counters */
(3,1)
(1,2)          for (j=1; j<=m; j++)
(2,2)              for (i=1; i<=n; i++)
(3,2)                  process(i,j);
(1,3)      }
(2,3)
(3,3)
```

- 蛇形 — 和行优先类似，但并不是每行都按照从左到右的顺序遍历，而是每到新的一行就改变方向。想象一台两个方向都能工作的打字机，则这样的顺序能让打印的时间最小化。

```
snake_order(int n, int m)
```

```

{
(1,1)         int i,j;          /* counters */
(1,2)
(1,3)         for (i=1; i<=n; i++)
(2,3)             for (j=1; j<=m; j++)
(2,2)                 process(i, j + (m+1-2*j) * ((i+1) % 2));
(2,1)         }
(3,1)
(3,2)
(3,3)

```

- 对角线序 — 沿着对角线往上和往下。注意  $n \times m$  网格共有  $m+n-1$  条对角线，它们包含的元素个数不尽相同。写一个正确的对角线遍历程序比它“看上去”的难度要大。

```

diagonal_order(int n, int m)
(1,1) {
(2,1)     int d,j;          /* diagonal and point counters */
(1,2)     int pcount;        /* points on diagonal */
(3,1)     int height;        /* row of lowest point */
(2,2)
(1,3)     for (d=1; d<=(m+n-1); d++) {
(4,1)         height = 1 + max(0,d-m);
(3,2)         pcount = min(d, (n-height+1));
(2,3)         for (j=0; j<pcount; j++)
(4,2)             process(min(m,d)-j, height+j,);
(3,3)     }
(4,3) }

```

### 12.1.2 对偶图及其表示

平面矩形网格最自然的表示方式是二维矩阵。根据场合的不同，可以用  $m[i][j]$  表示第  $(i, j)$  个顶点，或者第  $(i, j)$  个面。在两个维度中选取某一个坐标上增加  $\pm 1$ ，可以得到所有相邻的四个面。

在解决平面划分问题时，一个有用的概念是对偶图 (*dual graph*)，平面划分中的每个区域在对偶图中对应一个结点，在平面划分中相邻的区域所对应的顶点在对偶图中以边相连。

四色定理说的是，每个平面图都可以用四种颜色着色，使得任意一对相邻区域的颜色都不同。换一个角度，这也相当于给该平面图的对偶图进行顶点着色，而这个对偶图本身也是可平面的（想一想，为什么）。

注意矩形网格和六边形网格的对偶图分别是比原网格略小一点的矩形网格和六边形网  
格。换句话说，不管我们采用怎样的结构，顶点的连通性可以用来表示面连通性。

当矩形网格中的边上带权时，邻接表示法是一种自然的方式。最简单的实现方法也许是使用一个三维数组  $m[i][j][d]$ ，其中  $d$  取值有四种（东、南、西、北），分别表示了从点  $(i, j)$  出发沿着四个方向伸展出的四条边。

## 12.2 三角形网格和六边形网格

还有一些重要的正规网格，比如三角形网格 和六边形网格。二者的联系相当紧密 —— 事实上，把三角形网格中的一些顶点拿掉，就得到了六边形网格。

### 12.2.1 三角形网格

三角形网格由三簇等距直线组成，水平的称为“行”，由水平线逆时针旋转  $60^\circ$  得到的线称为“列”，再逆时针旋转  $60^\circ$  得到的线称为“斜线”，如图 12.1 所示。网格中的每个顶点都是三个方向各一条线的交叉点，因此每个面都是等边三角形。每个顶点  $v$  和六个其他顶点相邻，分别对应于三条轴线上的上端和下端。

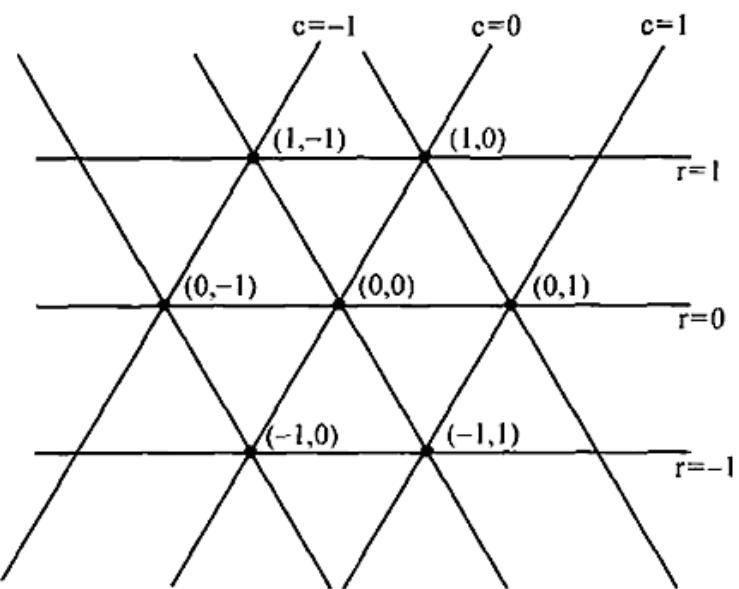


图 12.1 三角形网格的行-列坐标系

为了处理三角形网格，我们不仅要知道每个顶点的所有相邻点，而且要知道它们的地理位置。这需要我们同时关注两种不同的坐标系统：

- 三角形/六边形坐标系 — 在这个坐标系中，其中一个顶点被称为原点，坐标为  $(0,0)$ 。在给其他点分配坐标时应该让每个点的相邻点坐标容易计算。在标准的矩形网格坐标系中， $(x,y)$  的四个相邻点分别为给行列坐标的其中之一增加  $\pm 1$  所得到。

尽管网格中的每个顶点有三条线穿过，只需其中的任意两条就可以确定它的位置。在我们的坐标系中，顶点的位置只由行和列确定，如图 12.1 所示。顶点  $(x, y)$  位于原点

上方第  $x$  行, 右边第  $y$  列 ( $60^\circ$  的斜线)。顶点  $v$  的相邻点坐标可以通过在  $v$  的坐标基础上加上以下各数对得到 (逆时针排列):  $(0, 1), (1, 0), (1, -1), (0, -1), (-1, 0)$  和  $(-1, 1)$ 。

有多种方法表示三角形网格中面的坐标, 参见 [LR76]。

- 几何坐标 — 正规三角形网格中的顶点都对应着平面上的几何位置。如图 12.1 所示, 三角形网格中顶点交错的排列在平面上, 因为轴和轴之间的夹角为  $60^\circ$ , 而不是矩形网格中的  $90^\circ$ 。

假设每个顶点和六个相邻点的距离都是  $d$ , 且网格原点  $(0, 0)$  恰好与笛卡尔坐标系的原点重合。经过简单的三角运算可知, 三角网格坐标系中的点  $(x_t, y_t)$  在平面上的坐标为

$$(x_g, y_g) = (d(y_t + (x_t \cos(60^\circ))), dx_t \sin(60^\circ))$$

由于  $\cos(60^\circ) = 1/2$ ,  $\sin(60^\circ) = \sqrt{3}/2$ , 上述计算不需要调用三角函数。

在下小节中, 我们将给出六边形网格中的类似代码。

### 12.2.2 六边形网格

在三角形网格中每隔一个顶点删除一个顶点, 剩下的图形是一个六边形网格, 如图 12.2 所示。网格中的每个面都是正六边形, 且每个六边形和另外六个六边形相邻。每个顶点的度数均为 3, 因为六边形网格是三角形网格的对偶图。

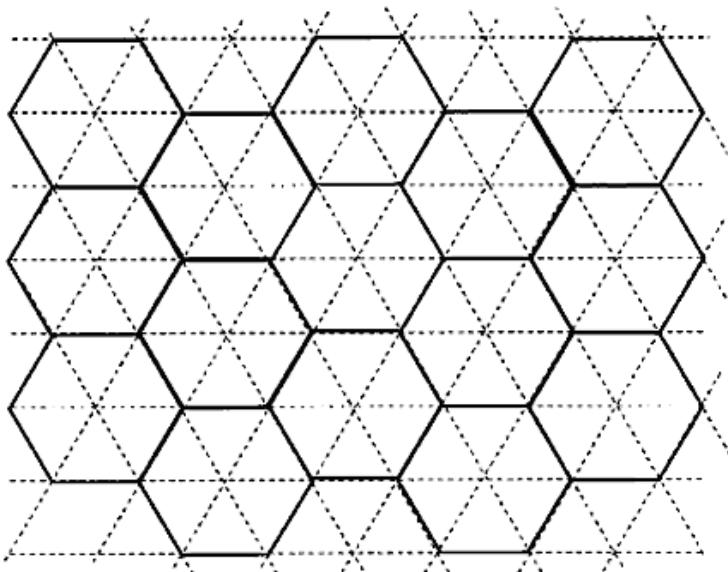


图 12.2 在三角形网格中删除一些顶点可以得到六边形网格

六边形网格有很多有趣且有用的性质, 这和六边形比正方形 “更接近圆” 有关。周长一定时, 圆的面积最大, 是最能有效利用材料的建筑结构。六边形网格比矩形网格更加坚固, 这也是蜂房被建成六边形的另一个原因。最小化单位面积的周长对减小图形的视觉失真 (*visual artifacts*) 起到了重要作用, 因此很多电脑游戏采用六边形区块 (*tile*) 而非矩形区块渲染。

在上一节中，我们讨论了三角形/六边形坐标和笛卡尔坐标之间的转换方法。假设两个坐标系的原点  $(0, 0)$  重合，然后想象以每个顶点为圆心有一些全等的圆，没有缝隙的布满整个平面。六边形坐标  $(x_h, y_h)$  表示圆心坐标在行号为  $x_h$ ，列号为  $y_h$  的位置。该点的几何位置和圆半径  $r$  有关，它正好是上一节中直径  $d$  的一半：

```
hex_to_geo(int xh, int yh, double r, double *xg, double *yg)
{
    *yg = (2.0 * r) * xh * (sqrt(3)/2.0);
    *xg = (2.0 * r) * xh * (1.0/2.0) + (2.0 * r) * yh;
}

geo_to_hex(double xg, double yg, double r, double *xh, double *yh)
{
    *xh = (2.0/sqrt(3)) * yg / (2.0 * r);
    *yh = (xg - (2.0 * r) * (*xh) * (1.0/2.0)) / (2.0 * r);
}
```

写成行-列这样的二维形式之后，我们可以有效地用矩阵  $m[row][column]$  来表示一个六边形网格。用三角形网格中相邻顶点的坐标差公式，我们很容易得到每个六边形面的六个邻居。

可是这里有一个问题。在六边形坐标系下，满足  $0 \leq hx \leq x_{\max}$ ,  $0 \leq hy \leq y_{\max}$  的所有六边形  $(hx, hy)$  的圆心排列成菱形，而不是像矩形网格那样排列成边平行于坐标轴的矩形。但是，对很多应用来说，我们希望把这些圆装在矩形而不是菱形里。为了解决这个问题，我们定义数组坐标  $(ax, ay)$  表示左下角在  $(0, 0)$  点的，边平行于坐标轴的矩形中各个圆心在矩阵中的位置：

```
array_to_hex(int xa, int ya, int *xh, int *yh)
{
    *xh = xa;
    *yh = ya - xa + ceil(xa/2.0);
}

hex_to_array(int xh, int yh, int *xa, int *ya)
{
    *xa = xh;
    *ya = yh + xh - ceil(xh/2.0);
}
```

图 12.3 展示了一个六边形网格，其中每个圆心除了标明了六边形坐标之外，还在它的下面用斜体给出了矩阵中的数组下标（如果二者不相同）。

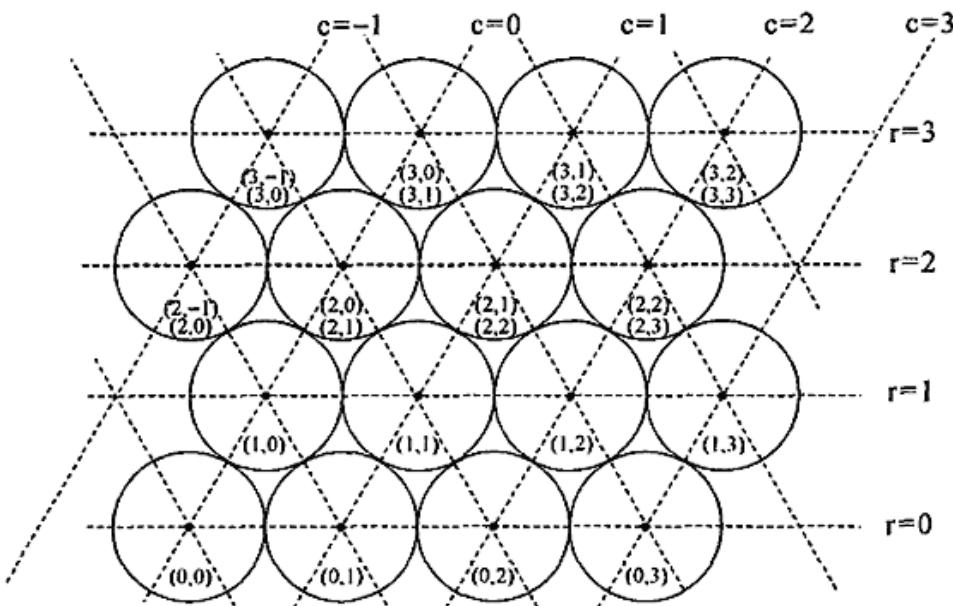


图 12.3 六边形坐标系下的圆盘填装 (disk packing) 以及和六边形坐标不同的数组下标  
(在六边形坐标下, 用斜体表示)

### 12.3 程序设计举例：西餐碟的重量

一家西餐碟制造商准备进入竞争激烈的校园食堂市场。食堂所购买的西餐碟尺寸统一，因此可以整齐的叠在一起。同学们总会在食物大战 (food fights) 中打碎很多碟子，因此给食堂提供新碟是个不错的生意。

我们公司打算依靠独特的打包技术在市场中占领一席之地。由于六边形网格比矩形网格密，我们把碟子打包在一个  $l \times w$  的盒子中，如图 12.4(l) 所示。每个碟子的半径均为  $r$ ，最下面一行恰好有  $p = \lfloor w/(2r) \rfloor$  个碟子。根据  $w$  和  $r$  的关系不同，有可能每行都恰好有  $p$

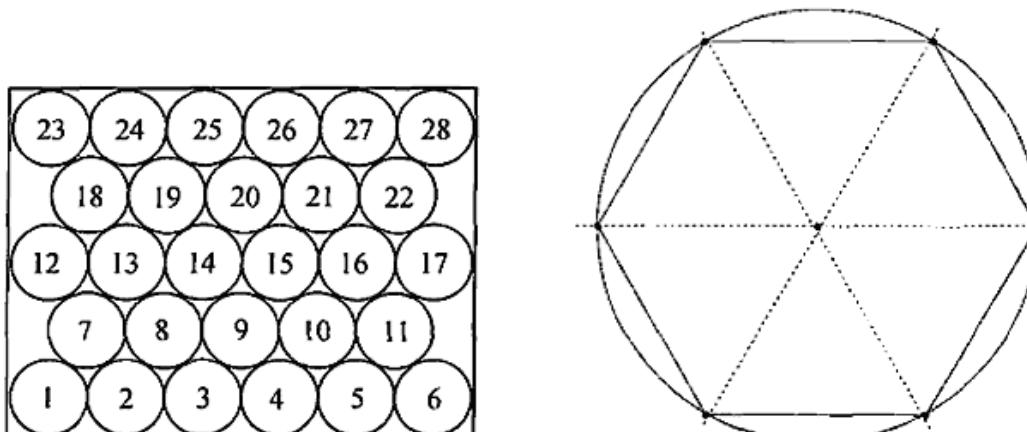


图 12.4 把碟子打包放在盒子中 (l)。把六边形网格看成打包方案 (r)

个碟子，也有可能从下数第奇数行有  $p$  个碟子，而第偶数行有  $p - 1$  个碟子。每个碟子都有一个唯一编号，如图 12.4(1) 所示。在不超过长度  $l$  的限制下，堆砌的行数应该尽量多。

管理者想知道：对于给定的长和宽，最多可以装多少个碟子。他们还需要知道一个碟子上方叠了多少个其他的碟子，以确保整个包装不会易碎。你的任务是计算出哪个碟子承受着最大的压力，它的上方有多少个碟子，以及整个包装里一共有多少层，多少个碟子。

---

以下为题解

---

第一个问题是计算出基于六边形网格的堆砌方案一共能装下多少行。这正是三角学的用武之地。最下层的碟子直接放在盒子底部，该层上所有圆心组成的中心线在底部之上  $r$  个单位，其中  $r$  是碟子的半径。相邻行的中心线之间的垂直距离是  $2r \sin(60^\circ) = 2r\sqrt{3}/2$ 。我们可以把因子 2 约去，不过公式的三角意义就没有这么明显了。

```
int dense_layers(double w, double h, double r)
{
    double gap; /* distance between layers */

    if ((2*r) > h) return(0);

    gap = 2.0 * r * (sqrt(3)/2.0);
    return( 1 + floor((h-2.0*r)/gap) );
}
```

盒里能装下的碟子数目同时取决于层数和每层的碟子数。我们总是把最下行（第 0 行）的碟子尽量往左放，这样可以在满足宽度限制的前提下放尽量多的碟子。编号为奇数的行整体往右偏移了  $r$  个单位，如果最右边的碟子超出了边界，则必须把它拿掉：

```
int plates_per_row(int row, double w, double r)
{
    int plates_per_full_row; /* plates in full/even row */

    plates_per_full_row = floor(w/(2*r));

    if ((row % 2) == 0) return(plates_per_full_row);

    if (((w/(2*r))-plates_per_full_row) >= 0.5) /* odd row full */
        return(plates_per_full_row);
    else
```

```

    return(plates_per_full_row - 1);
}

```

可以用坐标系来简化计算碟子受到的压力。在没有宽度限制的情况下，碟子  $(r, c)$  受到它上一层（第  $r+1$  层）中两个碟子的压力，即  $(r+1, c-1)$  和  $(r+1, c)$ 。一般地，碟子  $(r, c)$  受到它上面第  $i$  层（总第  $r+i$  层）中  $i+1$  个碟子的（间接）压力。当然，要去掉那些超出盒子边界的碟子——它们并不存在。用数组坐标可以很方便地看出一个碟子是否超出边界，因此可以用它计算产生压力的椎体被盒子切割后的剩下部分：

```

int plates_on_top(int xh, int yh, double w, double l, double r)
{
    int number_on_top = 0;           /* total plates on top */
    int layers;                   /* number of rows in grid */
    int rowlength;                /* number of plates in row */
    int row;                      /* counter */
    int xla,yla,xra,yra;         /* array coordinates */

    layers = dense_layers(w,l,r);

    for (row=xh+1; row<layers; row++) {
        rowlength = plates_per_row(row,w,r) - 1;

        hex_to_array(row,yh-(row-xh),&xla,&yla);
        if (yla < 0) yla = 0;           /* left boundary */

        hex_to_array(row,yh,&xra,&yra);
        if (yra > rowlength) yra = rowlength; /* right boundary */

        number_on_top += yra-yla+1;
    }

    return(number_on_top);
}

```

## 12.4 给圆打包

六边形网格和圆盘填装之间有着有趣而重要的联系。网格中每个顶点  $v$  的六个相邻点

到  $v$  的距离相同，因此可以以  $v$  为圆心作一个穿过它们的圆，如图 12.4(r) 所示。每个圆盘和六个相邻圆盘相切，如图 12.3 所示。

圆盘填装问题要求我们比较两种不同的方法来给一些全等的圆盘打包。第一种方案是把圆心排列成矩形网格，第二种方案是排列成六边形网格。哪种方法能装更多的圆盘？很容易利用刚才设计的过程计算两种方案的圆盘数：

```
/* How many radius r plates fit in a hexagonal-lattice packed w*h box? */

int dense_plates(double w, double l, double r) {
    int layers; /* number of layers of balls */

    layers = dense_layers(w,l,r);

    return (ceil(layers/2.0) * plates_per_row(0,w,r) +
            floor(layers/2.0) * plates_per_row(1,w,r));
}

/* How many radius r plates fit in a grid-lattice packed w*h box? */

int grid_plates(double w, double h, double r)
{
    int layers; /* number of layers of balls */

    layers = floor(h/(2*r));

    return (layers * plates_per_row(0,w,r));
}
```

当盒子足够大时，六边形网格总是能比矩形网格装下更多的圆盘。事实上，在渐进意义上，六边形装填是最密集的圆盘装填方式，而它在三维空间中的类比是最密集的圆球装填方式。

由于边缘效应， $4 \times 4$  盒子用正方形装填方案能装下 16 个单位直径的圆盘，但用六边形装填方案只能装下 14 个。但  $10 \times 10$  的盒子用六边形方案能装 105 个，已经比正方形方案多了 5 个。对于更大的盒子，六边形方案的好处更加明显： $100 \times 100$  的盒子用六边形方案能装 11 443 个，而正方形方案只能装 10 000 个。这样，先进的打包技术为我们带来了巨大的优势。

## 12.5 经度和纬度

经纬度系统是一个非常重要的网格，它指明了地球表面上每个点的位置。

和赤道平行的东西向的环线称为纬线 (*lines of latitude*)。赤道的纬度为  $0^\circ$ ，而北极和南极分别是北纬  $90^\circ$  和南纬  $90^\circ$ 。

连接北极和南极的线称为经线 (*lines of longitude/lines of meridians*)。本初子午线 (*prime meridian*) 穿过英国的格林尼治天文台，它的经度为  $0^\circ$ 。整个经度的范围从西经  $180^\circ$  到东经  $180^\circ$ 。

地球表面上的任意点都是一条纬线和一条经线的交点。例如，曼哈顿的中心位于北纬  $40^\circ 47'$ ，西经  $73^\circ 58'$ 。

一个常见问题是寻找地球表面上两点之间的最短飞行距离。用一个穿越球心的平面去截球体，截面称为大圆 (*great circle*)。 $x$  和  $y$  之间的最短路长度是穿过  $x$  和  $y$  的唯一大圆上的劣弧长，除非  $x$ 、 $y$  和球心共线。

我们用空间几何的观点重述这个结果。设点  $p$  的纬度和经度组成的二元组是  $(p_{\text{lat}}, p_{\text{long}})$ ，球的半径为  $R$ ，则  $p$  和  $q$  之间的球面距离为 (其中角均用弧度表示)：

$$d(p, q) = R \arccos(\sin(p_{\text{lat}}) \sin(q_{\text{lat}}) + \cos(p_{\text{lat}}) \cos(q_{\text{lat}}) \cos(p_{\text{long}} - q_{\text{long}}))$$

## 12.6 习题

### 12.6.1 棋盘上的蚂蚁 (Ant on a Chessboard)

PC/UVa 题号: 111201/10161, 流行度: B, 通过率: high 难度: 1

一天，一只叫 Alice 的蚂蚁来到一个  $M \times M$  的棋盘上。她想要探索棋盘上的每个格子，于是从一个角落出发，开始了她的旅行。

Alice 从方格  $(1, 1)$  出发。一开始，她向上走一格，然后向右走一格，再向下走一格。接下来，她向右走一格，向上走两格，向左走两格。就这样，她每一轮从她所在的角落向外推进一行和一列。

例如，她的前 25 步会像如下的样子，方格中的数字表示她到达这格时的步数。

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

她第 8 步走到了  $(2, 3)$ , 而第 20 步时则走到了  $(5, 4)$ 。你的任务是求出在一个给定的时刻她在什么位置。假定棋盘足够大, 因此不必担心她会走出棋盘。

输入

输入包含多行，每行一个整数  $N$ ，表示她行走的步数，满足  $1 \leq N \leq 2 \times 10^9$ 。 $N = 0$  表示数据结束。

输出

对于每组输入数据，输出一行两个数  $x$  和  $y$ ，分别表示  $N$  步后的列与行的编号。这两个数应用单个空格隔开。

### 样例输入

8	2 3
20	5 4
25	1 5
0	

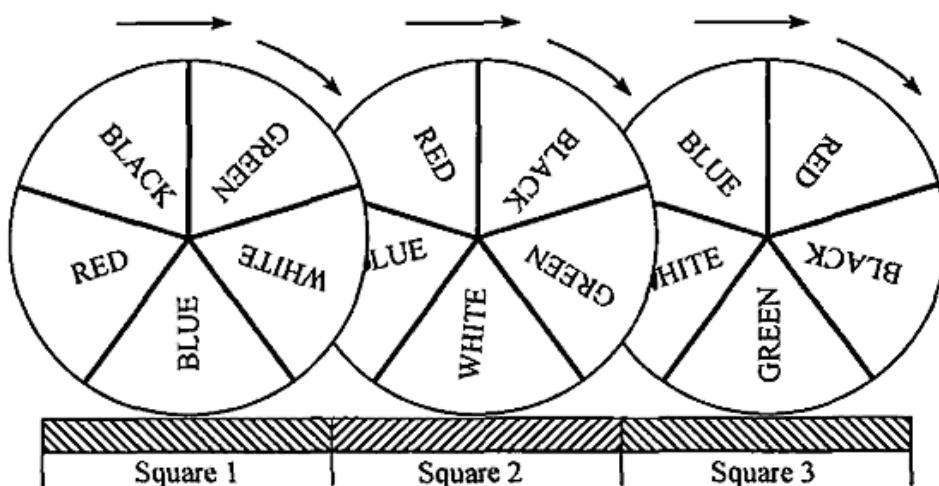
## 样例输出

23  
54  
15

### 12.6.2 独轮车 (The Monocycle)

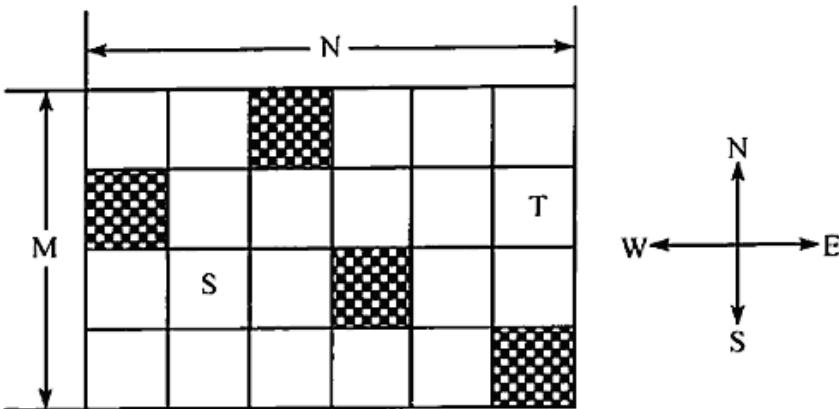
PC/UVa 题号: 111202/10047, 流行度: C, 通过率: average 难度: 3

独轮车是一种只有一个轮子的单车。我们将讨论的是一种特殊的独轮车，它有一个实心的彩色轮子，有如下图的五种颜色：



颜色的边界在圆心处形成相等的角度 ( $72^\circ$ )。一个独轮车手骑着这架独轮车在一块  $M \times N$  的方砖地面上行进。方砖的大小恰好使独轮车从一个方砖中心前进到相邻方砖中心时轮子旋转  $72^\circ$ 。效果如上图所示。当轮子停在方格 1 的中心时，蓝色 (BLUE) 部分的中心正好着地。而当轮子移动到下一个方砖 (方格 2) 时，则是白色 (WHITE) 部分的中心着地。

地面上一些方块有障碍物，车手无法到达这些地方。车手从某一格出发并试图在最短时



间到达一个目标格。在每一格，他要么向前移动到下一格，要么留在同一格并向左或向右转 $90^\circ$ 。每一种动作都会花去他一秒钟。他总是面向北方出发并使车轮的绿色(GREEN)部分中心着地。同样的，在到达目标格的时候也必须是绿色部分中心着地，但不用管他朝向什么方向。

请帮助这个独轮车手判断能否到达目的地，如果能够到达，计算出最少需要多长时间。

输入

输入可能包含多组数据。

每组数据第一行包含两个整数  $M$  和  $N(1 \leq M, N \leq 25)$ , 表示地面的大小。接下来的  $M$  行每行  $N$  个字符, 描述每格的状态。字符 “#” 表示障碍物, 其他情况均为可通行的格。车手出发的位置标识为 “S”, 终点标识为 “T”。

$M = N = 0$  时输入结束。

输出

对于每组数据，首先输出一行数据的编号（参见样例）。如果目标格可以到达，按样例格式输出到达需要的最短时间。否则输出“destination not reachable”。

相邻两组数据的输出间用一个空行分隔。

### 样例输入

1 3  
S#T  
10 10  
#S.....#  
#.#.##.##  
.##.##.##  
##.##..#.##  
# .. # ##

## 样例输出

Case #1  
destination not reachable

Case #2  
minimum time = 49 sec

```

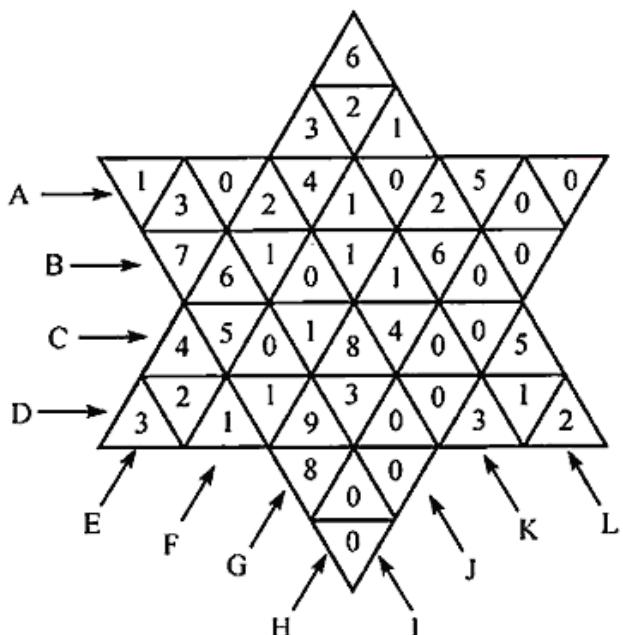
#.....##.
..##.##...
#.###...#.
#.....###T
 0 0

```

### 12.6.3 六角星 (Star)

PC/UVa 题号: 111203/10159, 流行度: C, 通过率: average 难度: 2

一块六角星形板上有 48 个三角形小格。每格里写着 0 到 9 之间的一个数字。每一格都同时属于两列或三列。这些列由字母  $A$  到  $L$  表示。如下图，写着 9 的格子同时属于列  $D$ ，列  $G$  和列  $I$ ，而写着 7 的格子同时属于列  $B$  和列  $I$ 。



对于每一列，我们统计出该列中最大的数字。这里列 A 最大数字是 5, B 是 7, E 是 6, H 是 0, J 是 8。

写一个程序，读入所有 12 列每列最大的数字，计算板上所有数字和可能的最大值和最小值。

输入

输入每行包含 12 个数字，用空格隔开。第一个数字是列 A 中的最大值，第二个是列 B 中的最大值，以此类推，最后一个数是列 L 中的最大值。

输出

对于输入的每行，输出一行，包含两个整数，分别表示该板数字和的最小可能值和最大可能值，用一个空格隔开。如果无解，输出“NO SOLUTION”。

**样例输入**

5 7 8 9 6 1 9 0 9 8 4 6

**样例输出**

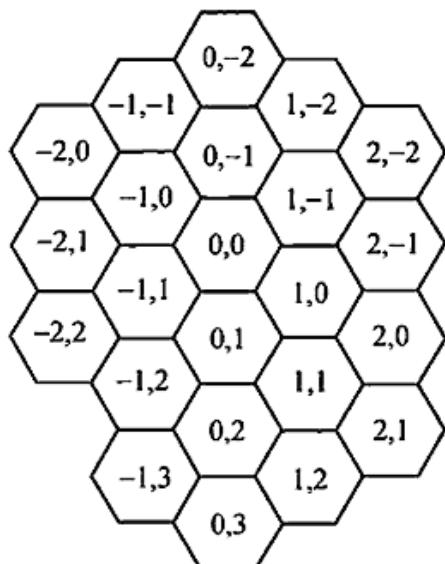
40 172

**12.6.4 蜜蜂 Maja(Bee Maja)**

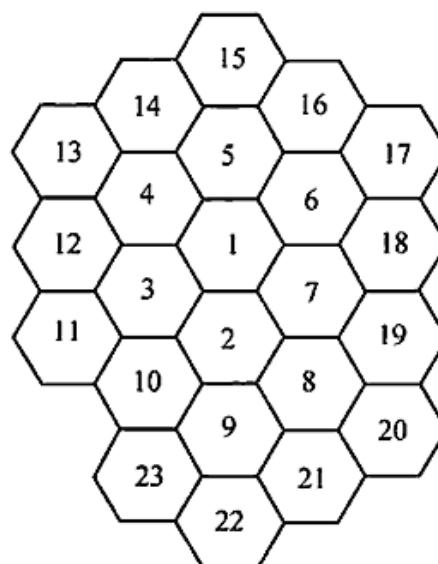
PC/UVa 题号: 111204/10182, 流行度: B, 通过率: high 难度: 2

Maja 是一只蜜蜂。她与成千只蜜蜂共同生活在一个六角形的蜂房里。但是 Maja 遇到了一个问题。她的朋友 Willi 告诉她约会的地点，但是 Willi (他是一只雄蜂) 和 Maja (她是一只工蜂) 各有一套不同的坐标系统：

- Maja 的坐标系统 — Maja(左) 根据建立在整个蜂巢上的，先进的二维坐标系直接飞向某一特定蜂房。
- Willi 的坐标系统 — Willi(右) 没有那么聪明。他只是沿着顺时针方向从蜂巢中心旋转着向外走。



Maja 的系统



Willi 的系统

帮助 Maja 将 Willi 的坐标系统转化成她自己的坐标系统。写一个程序，对给定的蜂房编号返回其在 Maja 坐标系中的坐标。

**输入**

输入包含一个或多个整数，每个一行，表示待转换的蜂房编号。所有蜂房编号均小于 100 000。

**输出**

输出 Willi 的编号对应 Maja 坐标系中的坐标，每个坐标为一对整数，在输出中占单独的一行。

**样例输入**

```
1
2
3
4
5
```

**样例输出**

```
0 0
0 1
-1 1
-1 0
0 -1
```

**12.6.5 抢劫案 (Robbery)**

PC/UVa 题号: 111205/707, 流行度: B, 通过率: average 难度: 3

Robostop 督察很生气, 因为昨晚有一家银行被抢劫, 而劫匪逃掉了。为阻止劫匪外逃, 所有的出城道路都被尽快封锁了。接下来, 督察开始询问城中的每一个人以寻找罪犯, 但他只得到了一些“我们没看见”这样的回答。

Robostop 下决心要找出劫匪逃离的确切路径。他请你写一个程序来分析所有他掌握的信息, 找出给定时刻劫匪所在位置。

被抢银行所在的城市是矩形的。所有的出城道路在一个特定的时间段  $t$  内都是封锁的, 在这期间有数个格式为“The robber isn't in the rectangle  $R_i$  at time  $t_i$ ”这样的观察汇报, 表示在时间  $t_i$ , 劫匪不在矩形  $R_i$ 。假定歹徒在单位时间只能移动单位距离, 试着找出歹徒在每一时刻的确切位置。

**输入**

输入包含多宗抢劫案。每一宗的第一行包含三个整数  $W$ ,  $H$  和  $t$  ( $1 \leq W, H, t \leq 100$ ), 其中  $W$  和  $H$  分别描述城市这个矩形的宽和高,  $t$  为道路被封锁的时间。

接下来的一行包含一个整数  $n$  ( $0 \leq n \leq 100$ ), 督察得到的信息数目。接下来的  $n$  行每行包含 5 个整数  $t_i, L_i, T_i, R_i, B_i$ , 其中  $t_i$  为观察时刻 ( $1 \leq t_i \leq t$ ), 而  $L_i, T_i, R_i, B_i$  分别为观察到的矩形区域的左、上、右、下边界。点  $(1, 1)$  在城市的左上角, 而点  $(W, H)$  在城市的右下角。信息的含义是: 在时刻  $t_i$ , 劫匪不在这个矩形内。

当  $W = H = t = 0$  时输入结束。

**输出**

对于每宗抢劫案, 输出一行“Robbery # $k$ :”, 其中  $k$  是案件编号。接下来有三种可能的情况:

如果劫匪不可能仍在城中, 则输出“The robber has escaped.”。

在其他情况下, 假设劫匪仍在城中。对每一个可以确切推断出劫匪位置的时刻输出一行“Time step  $\tau$ : The robber has been at  $x, y$ .”。其中  $x$  和  $y$  分别为时刻  $\tau$  劫匪所在的列与行。输出按照时间  $\tau$  排序。

如果无法推断出任何结果，输出“Nothing known.”。但愿督察不会更加生气。  
每组数据的输出后应增加一个空行。

### 样例输入

```
4 4 5
4
1 1 1 4 3
1 1 1 3 4
4 1 1 3 4
4 4 2 4 4
10 10 3
1
2 1 1 10 10
0 0 0
```

### 样例输出

Robbery #1:

```
Time step 1: The robber has been at 4,4.
Time step 2: The robber has been at 4,3.
Time step 3: The robber has been at 4,2.
Time step 4: The robber has been at 4,1.
```

Robbery #2:

The robber has escaped.

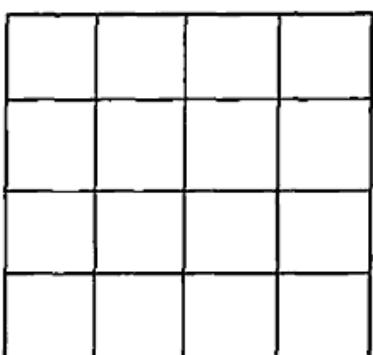
### 12.6.6 (2/3/4)-维立方体?((2/3/4)-D Sqr/Rects/Cubes/Boxes?)

PC/UVa 题号: 111206/10177, 流行度: B, 通过率: high 难度: 2

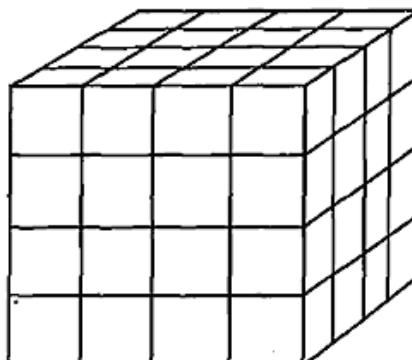
下图  $4 \times 4$  方阵中有多少个正方形和多少个矩形？或许对于这么小的方阵你可以用手数出来，但是如果是一个  $100 \times 100$  的方阵呢？

如果是更高的维度呢？你能数出一个  $10 \times 10 \times 10$  的立方体中有多少个大小不同的立方体或长方体吗？你能数出一个  $5 \times 5 \times 5 \times 5$  的四维超立方体中有多少个超立方体或者超长方体吗？

你的程序需要高效，因此你需要巧妙的设计算法。假定正方形不属于长方形，立方体不属于长方体，超立方体不属于超长方体。



4×4 方阵



4×4×4 立方体

### 输入

输入数据每行包含一个整数  $N(0 \leq N \leq 100)$ , 表示方阵、立方体或超立方的边长。上例中  $N = 4$ 。输入数据至多 100 行。

### 输出

对于输入的每行, 输出一行六个整数  $S_2, R_2, S_3, R_3, S_4, R_4$ , 其中  $S_2$  表示二维 ( $N \times N$ ) 方阵中正方形的数量,  $R_2$  表示长方形的数量,  $S_3, R_3, S_4, R_4$  对应三维和四维情况。

#### 样例输入

```
1
2
3
```

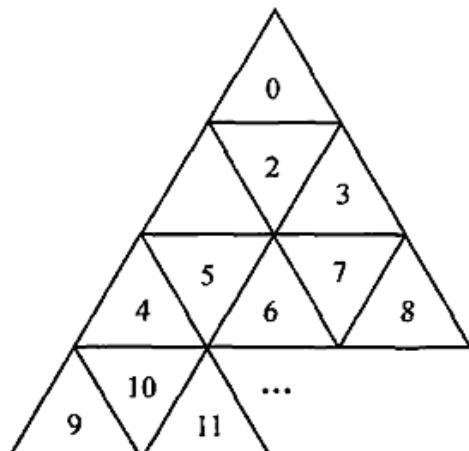
#### 样例输出

```
1 0 1 0 1 0
5 4 9 18 17 64
14 22 36 180 98 1198
```

### 12.6.7 Dermuba 三角 (Dermuba Triangle)

PC/UVa 题号: 111207/10233, 流行度: C, 通过率: high 难度: 2

Dermuba 三角形是在 L-PAX 星球上一种闻名于整个 Geometria 星系间的平面三角型区域。Dermuba 人生活在边长恰好为 1 千米的等边三角形土地上。房子总是被建造在三角形土地的外心处。他们的房子被编号如下图:



当 Dermuba 人相互拜访时，他们从自己的房子沿最短路径走到目的地的房子。显然最短路径是连接两所房子的直线距离。你的任务来了：写一个程序，根据房子的编号计算两所房子间的最短路径。

### 输入

输入包含多行，每行包含两个非负整数  $n$  和  $m$ ，表示起始和目的房子的编号，其中  $0 \leq n, m \leq 2\,147\,483\,647$ 。

### 输出

对于输入的每行，输出给定房子间最短路径，四舍五入保留小数点后三位（单位：千米）。

#### 样例输入

0 7	1.528
2 8	1.528
9 10	0.577
10 11	0.577

#### 样例输出

### 12.6.8 航线 (Airlines)

PC/UVa 题号: 111208/10075, 流行度: C, 通过率: high 难度: 3

某家一流的航空公司聘请你来写一个程序完成如下查询：给定城市位置以及直达航线的列表，从一个特定城市到另一个特定城市的最短航行距离是多少？城市的位置用经纬度给出。

从一个城市到达另一个城市可以搭乘直达航班（如果有的话），也可以进行一系列的转机后到达。

当从  $X$  直飞  $Y$  时，飞行距离为两地的球面距离（地球表面连接两点的曲线段中最短的距离）四舍五入到的最近整数。

地球近似成一个半径 6 378 千米的正球体， $\pi$  的近似值为 3.141592653589793。

### 输入

输入可能包含多组数据。每组数据第一行包含三个整数  $N \leq 100, M \leq 300, Q \leq 10000$ ，其中  $N$  表示城市数， $M$  表示直达航线数目， $Q$  表示查询次数。

接下来的  $N$  行为城市列表。其中第  $i$  行包含一个字符串  $c_i$  和两个实数  $lt_i$  和  $ln_i$ ，分别为城市名称，纬度和经度。城市名最多为 20 个字符且不会包含空格。纬度在  $-90^\circ$ （南极）到  $+90^\circ$ （北极）之间。经度在  $-180^\circ$  到  $+180^\circ$  之间，负数和正数分别表示西经和东经。

接下来的  $M$  行为航线列表。其中第  $i$  行包含两个城市名  $a_i$  和  $b_i$ ，表示城市  $a_i$  到  $b_i$  有一条直达航线。两个城市名都包含在上面的城市列表中。接下来的  $Q$  行为查询列表。其中

第  $i$  行包含城市名  $a_i$  和  $b_i$ , 查询从城市  $a_i$  到  $b_i$  乘客需要飞行的最短距离。数据保证  $a_i$  与  $b_i$  不相同且均包含在城市列表中。

$N = M = Q = 0$  时输入数据结束。

### 输出

对于每组数据, 首先按照样例格式输出数据编号(从 1 开始)。接下来对于输入的每个查询, 输出一行, 给出乘客从城市  $a_i$  到城市  $b_i$  要飞行的最短距离(单位: 千米)。如果无法从  $a_i$  到  $b_i$ , 输出“no route exists”。

相邻两组数据的输出之间应用一个空行隔开。

### 样例输入

```
3 4 2
Dhaka 23.8500 90.4000
Chittagong 22.2500 91.8333
Calcutta 22.5333 88.3667
Dhaka Calcutta
Calcutta Dhaka
Dhaka Chittagong
Chittagong Dhaka
Chittagong Calcutta
Dhaka Chittagong
5 6 3
Baghdad 33.2333 44.3667
Dhaka 23.8500 90.4000
Frankfurt 50.0330 8.5670
Hong_Kong 21.7500 115.0000
Tokyo 35.6833 139.7333
Baghdad Dhaka
Dhaka Frankfurt
Tokyo Hong_Kong
Hong_Kong Dhaka
Baghdad Tokyo
Frankfurt Tokyo
Dhaka Hong_Kong
Frankfurt Baghdad
```

Baghdad Frankfurt

0 0 0

### 样例输出

Case #1

485 km

231 km

Case #2

19654 km

no route exists

12023 km

## 12.7 提 示

- 12.1 必须显式的在图上走出一条路径，还是可以用公式直接算出终点是哪格？
- 12.2 如何建图来刻画出车轮的五色结构？
- 12.3 是否可以独立算出每列中的最小值和最大值？
- 12.4 如果我们无法找到在 Willi 系统中直接计算坐标的公式，如何合理地使用数据结构来模拟它的移动过程？
- 12.5 如何建图来同时刻画时间和空间？
- 12.6 二维、三维的面投影公式如何扩展到四维？每个超立方体是否仍然可以用两个对角顶点确定？
- 12.7 如何将前面的三角坐标系统和这个新系统相互转化？
- 12.8 通常根据经纬度计算距离的公式是否仍然适用？或者，这些公式会遇到点麻烦？题目背后隐藏着怎样的一张图？

## 第13章 几何

柏拉图学院 (Plato's academy) 门口的牌子上刻着一句话：“Let no one who is ignorant of geometry enter here.”(不懂几何者不得入内。) 程序设计竞赛的组织者似乎也赞同这一观点，因此几乎每场比赛都有至少一道几何题目。

从根本上讲，几何是一门可视化的学科，它要求作图，然后细心研究。几何编程的困难之一在于一些“容易作图求解”的操作（例如求两条直线的交点）。对于编程来说，正确的实现这些运算并不容易。

每个人在高中课程里都学过几何，但随着时间的流逝，这些几何知识慢慢地被人们所遗忘。本章力图用一些关于直线、点、圆等常见几何对象的编程问题唤醒你的记忆。当解决这样一些问题之后，你应该能够重新胸有成竹地大踏步走进柏拉图学院。

几何的内容博大精深。线段和多边形的问题将在第 14 章专门讨论。

### 13.1 直线

两点之间的直线距离最短。直线 (*lines*) 两端均无限延伸，而线段 (*line segments*) 则长度有限。这里的讨论围绕平面上的直线进行。

- 表示 — 直线有两种不同的表示方法：点对和直线方程。每条直线  $l$  可以完全用直线上的一对点  $(x_1, y_1)$  和  $(x_2, y_2)$  来表示。直线还可以完全用直线方程<sup>1</sup>表示，如  $y = mx + b$ ，其中  $m$  是直线的斜率 (*slope*)， $b$  是  $y$  轴上的截距 (*intercept*)，即直线穿过  $y$  轴的唯一一点  $(0, b)$ 。直线  $l$  的斜率  $m = \Delta y / \Delta x = (y_1 - y_2) / (x_1 - x_2)$ ，截距  $b = y_1 - mx_1$ 。

然而，竖线无法用上面的方程表示，因为除以  $\Delta x$  会导致除零错误。方程  $x = c$  表示在  $(c, 0)$  处穿越  $x$  轴的竖线。这种特殊情况，或者称为退化 (*degeneracy*)，需要在编程时特别注意。因此，我们在今后的讨论中统一采用一般式  $ax + by + c = 0$  作为直线的表示法，因为它涵盖了平面上所有可能的直线：

```
typedef struct {
    double a;           /* x-coefficient */
    double b;           /* y-coefficient */
    double c;           /* constant term */
} line;
```

<sup>1</sup> 译者注：这里特指斜截式。

若将三个系数同时乘以一个非零常数，我们可以得到同一条直线的另一个表示方法。为了得到标准化表示 (*canonical representation*)，我们规定当  $y$  的系数非零时必须为 1，而当  $y$  的系数为 0 时  $x$  的系数必须为 1：

```
points_to_line(point p1, point p2, line *l)
{
    if (p1[X] == p2[X]) {
        l->a = 1;
        l->b = 0;
        l->c = -p1[X];
    } else {
        l->b = 1;
        l->a = -(p1[Y]-p2[Y])/(p1[X]-p2[X]);
        l->c = -(l->a * p1[X]) - (l->b * p1[Y]);
    }
}
```

```
point_and_slope_to_line(point p, double m, line *l)
{
    l->a = -m;
    l->b = 1;
    l->c = -((l->a*p[X]) + (l->b*p[Y]));
}
```

- 相交 — 两条不同的直线要么拥有唯一的交点 (*intersection point*)，要么平行 (*parallel*) (即没有交点)。平行线的斜率相同，但截距不同。根据定义，它们永不相交。

```
bool parallelQ(line l1, line l2)
{
    return ( (fabs(l1.a-l2.a) <= EPSILON) &&
             (fabs(l1.b-l2.b) <= EPSILON) );
}
```

```
bool same_lineQ(line l1, line l2)
{
    return ( parallelQ(l1,l2) && (fabs(l1.c-l2.c) <= EPSILON) );
}
```

点  $(x', y')$  在直线  $l : y = mx + b$  上当且仅当把  $x'$  代入公式中的  $x$  后, 右式等于  $y'$ 。直线  $l_1 : y = m_1x + b_1$  和  $l_2 : y_2 = m_2x + b_2$  的交点  $(x, y)$  同时满足两个方程, 联立解得:

$$x = \frac{b_2 - b_1}{m_1 - m_2}, \quad y = m_1 \frac{b_2 - b_1}{m_1 - m_2} + b_1$$

```
intersection_point(line l1, line l2, point p)
{
    if (same_lineQ(l1,l2)) {
        printf("Warning: Identical lines, all points intersect.\n");
        p[X] = p[Y] = 0.0;
        return;
    }

    if (parallelQ(l1,l2) == TRUE) {
        printf("Error: Distinct parallel lines do not intersect.\n");
        return;
    }

    p[X] = (l2.b*l1.c - l1.b*l2.c) / (l2.a*l1.b - l1.a*l2.b);

    if (fabs(l1.b) > EPSILON) /* test for vertical line */
        p[Y] = - (l1.a * (p[X]) + l1.c) / l1.b;
    else
        p[Y] = - (l2.a * (p[X]) + l2.c) / l2.b;
}
```

- 角度 — 任何两条(不同的)不平行直线以特定角度相交。对于一般式, 直线  $l_1 : a_1x + b_1y + c_1 = 0$  和  $l_2 : a_2x + b_2y + c_2 = 0$  的交角的正切<sup>2</sup>为

$$\tan \theta = \frac{a_1b_2 - a_2b_1}{a_1a_2 + b_1b_2}$$

而对于斜截式, 公式为  $\tan \theta = (m_2 - m_1)/(m_1m_2 + 1)$ 。

两条直线垂直(*perpendicular*)当且仅当二直线以直角相交。例如, 正交坐标系的  $x$  轴和  $y$  轴相互垂直, 直线  $y = x$  和  $y = -1/x$  也是。垂直于直线  $l : y = mx + b$  的所有直线都可以表示为  $y = (-1/m)x + b'$ , 其中  $b'$  可取任意值。

<sup>2</sup>译者注: 垂直线的交角正切为正无穷、平行直线的交角正切为 0。

- 最近点 — 一个有用的子问题是：在直线  $l$  上找一点，使得它离给定点  $p$  最近。这个最近点在“过  $p$  并垂直于  $l$ ”的直线上，因此可以用前面已经介绍过的程序实现：

```
closest_point(point p_in, line l, point p_c)
{
    line perp; /* perpendicular to l through (x,y) */

    if (fabs(l.b) <= EPSILON) { /* vertical line */
        p_c[X] = -(l.c);
        p_c[Y] = p_in[Y];
        return;
    }

    if (fabs(l.a) <= EPSILON) { /* horizontal line */
        p_c[X] = p_in[X];
        p_c[Y] = -(l.c);
        return;
    }

    point_and_slope_to_line(p_in, 1/l.a, &perp); /* normal case */
    intersection_point(l, perp, p_c);
}
```

- 射线 — 它们是从一个点  $v$  出发的半直线，其中  $v$  称为射线的顶点 (*origin*)。射线可以用一个直线方程、顶点和方向来描述，也可以只记录顶点和射线上的另一点。

## 13.2 三角形和三角学

角 (*angle*) 由两条顶点相同的射线组成。三角学 (*trigonometry*) 是研究角及其度量的数学分支。

有两种常见的单位来度量角：弧度 (*radians*, 简称为 “*rad*”) 和角度(degrees, 通常简称为 “度”)。在平面上，角的范围是  $0 \sim 2\pi$  rad，或者等价地， $0 \sim 360^\circ$ 。从计算的角度讲，用弧度表示角比较好，因为 13.5 节中介绍的三角库函数都只用弧度 (rad) 表示角，但我们不得不承认，角度 ( $^\circ$ ) 对人来讲更加自然。由于历史原因，角度的分数部分用分 (*minutes*) 来表示，它代表  $1/60^\circ$ 。但同时处理 “度” 和 “分” 太麻烦了，因此在计算中应尽量用弧度 (即使用度数，也要用实数表示)。

三角形（“三个角组成的几何形状”）的几何性质和三角学密切相关，我们将在接下来的几个小节中一同讨论。

### 13.2.1 直角三角形和勾股定理

直角 (*right angle*) 是指  $90^\circ$  或者  $(\pi/2)\text{rad}$ 。两条垂直直线（如正交坐标轴）把  $360^\circ = 2\pi \text{ rad}$  分成了四个直角。

有公共顶点的两条射线实际上定义了两个角，内角 (*internal angle*) 为  $a \text{ rad}$ ，外角 (*external angle*) 为  $(2\pi - a) \text{ rad}$ 。通常我们关心的是内角。三角形的内角和为  $180^\circ = \pi \text{ rad}$ ，因此平均内角为  $60^\circ = (\pi/3) \text{ rad}$ 。三内角都相等的三角形称为等边三角形 (*equilateral triangle*)，我们已经在 12.2 节中讨论过它。

有一个内角为直角的三角形称为直角三角形 (*right triangle*)。直角三角形很容易处理，因为它满足勾股定理 (*Pythagorean theorem*)，所以任意给出两条边长都可以算出第三条边。具体来说， $|a|^2 + |b|^2 = |c|^2$ ，其中  $a$  和  $b$  是两条较短边（直角边）的长度， $c$  是最长边（斜边 (*hypotenuse*)）的长度。

勾股定理对于分析三角形大有帮助，而学会三角将使你如虎添翼。

### 13.2.2 三角函数

三角函数正弦 (*sine*) 和余弦 (*cosine*) 由圆心在  $(0, 0)$  处的单位圆上的点的  $x$  坐标和  $y$  坐标定义，如图 13.1(a) 所示。因此，正弦和余弦函数的取值在  $-1$  和  $1$  之间。另外，两个函数可以相互转化： $\cos(\theta) = \sin(\theta + \pi/2)$ 。

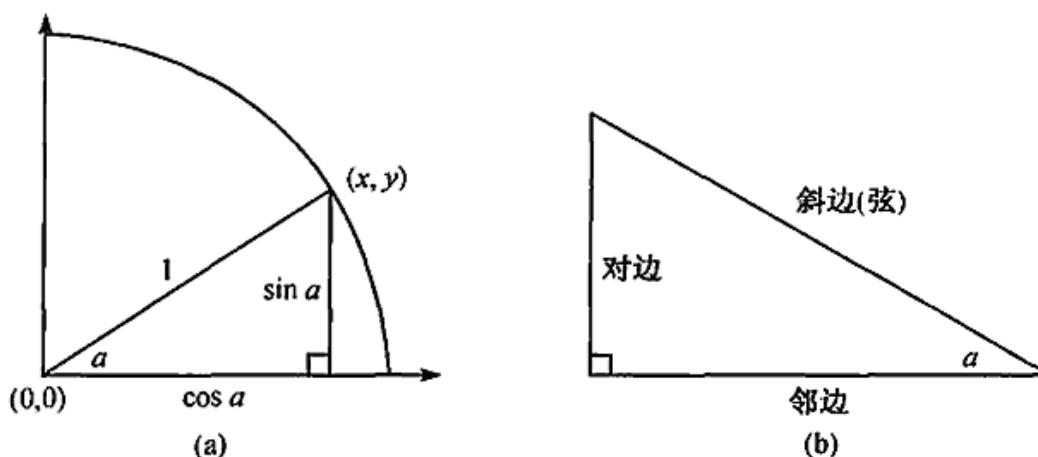


图 13.1 正弦和余弦的定义 (l)。给直角三角形的边标号 (r)

除此之外，最重要的三角函数当属正切 (*tangent*)，它是正弦和余弦的比值。因此， $\tan(\theta) = \sin(\theta)/\cos(\theta)$ 。注意当  $\cos(\theta) = 0$  时正切无定义，在  $[0, 2\pi)$  中对应两个值  $\theta = \pi/2$  和  $\theta = 3\pi/2$ 。

上述函数都很重要，因为它们把直角三角形  $T$  的任意两边长和两个锐角联系在了一起。直角三角形的斜边最长，它和直角相对；对于给定锐角  $a$ ，两条直角边中的其中一条是它的

新概念  
几何  
第 13 章 几何

对边 (*opposite edge*)，另一条是邻边 (*adjacent edge*)，如图 13.1(b) 所示。因此，

$$\cos(a) = \frac{|\text{邻边}|}{|\text{斜边}|}, \quad \sin(a) = \frac{|\text{对边}|}{|\text{斜边}|}, \quad \tan(a) = \frac{|\text{对边}|}{|\text{邻边}|}$$

读者应当记住这些关系。这里有个技巧，就是记住一位著名的印第安酋长 Soh-Cah-Toa<sup>3</sup>，它的名字中每个音节对应了一个关系。例如，“Cah” 表示余弦 ( $C$ ) 等于邻边 ( $A$ ) 除以斜边 ( $H$ )。

如果没有反函数来把  $\cos(\theta)$ 、 $\sin(\theta)$  和  $\tan(\theta)$  的函数值映射回内角本身，Soh-Cah-Toa 酋长的用处将大打折扣。上述三个函数的反函数<sup>4</sup>分别称为  $\arccos$ 、 $\arcsin$  和  $\arctan$ 。有了它们，给定直角三角形的任意两条边长，我们很容易算出它的所有内角。

三角函数一般用泰勒级数展开 (*Taylor series expansions*) 来计算的，但别担心：你最喜欢的编程语言的数学库中肯定包含了它们。三角函数并不是数值稳定的，因此使用的时候须多加小心。不要假定  $\theta$  和  $\arcsin(\sin(\theta))$  精确相等，尤其是对于较大和较小的角。

### 13.2.3 解三角形

有两个强大的三角公式可以用来计算三角形中的边和角。正弦定理 (*Law of Sines*) 描述的是三角形三边和它们的对角之间的关系。设三个角为  $A$ 、 $B$  和  $C$ ，对边分别为  $a$ 、 $b$  和  $c$ (如图 13.2(a))，有

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

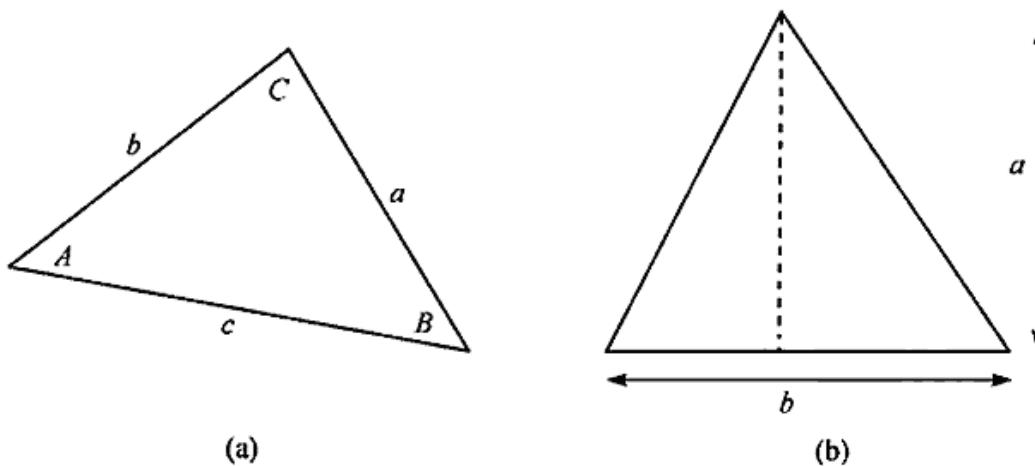


图 13.2 解三角形中的记号 (a) 和面积的计算 (b)

余弦定理 (*Law of Cosines*) 是勾股定理推广到非直角三角形的产物。设三个角为  $A$ 、 $B$  和  $C$ ，对边分别为  $a$ 、 $b$  和  $c$ ，有

$$a^2 = b^2 + c^2 - 2bc \cos A$$

<sup>3</sup> 译者注：尽管这个名字只是为了记忆三角函数而杜撰的，S.K.Martz 于 2001 年创作了一个关于 Soh-Cah-Toa 的童话故事，有兴趣的读者可以访问 [www.themathlab.com](http://www.themathlab.com) 阅读这个故事。

<sup>4</sup> 译者注：严格地说，它们只是原函数在特定区间内的反函数。

“解三角形”指的是给出三角形三边三角中的若干个量，求解其他量的过程。可大致分为两类：

- 已知两角一边，求其他量 — 第三个角很容易求出，因为三角形的三内角和为  $180^\circ = \pi$  rad。计算另两条边的一个方法是利用正弦定理。
- 已知两边一角，求其他量 — 如果已知角是两条已知边的夹角，可以用余弦定理算出第三条边，然后用正弦定理算出其他角。否则，可以用正弦定理和内角和性质计算出所有角，然后再用正弦定理计算出第三条边。

三角形  $T$  的面积  $A(T)$  由公式  $A(T) = (1/2)ab$  给出，其中  $a$  是高， $b$  是底。三角形的三条边都可以作为底，而对应的高就是与底边相对的顶点到底边的距离，如图 13.2(b) 所示。高可以由三角公式或者勾股定理轻易算出（具体方法取决于三角形的已有信息）。

另一个计算三角形面积的方法是直接利用坐标表示。用线性代数 (linear algebra) 和行列式 (determinants)，可以证明三角形  $T = (a, b, c)$  的有向面积 (*signed area*)  $A(T)$  满足：

$$2 \cdot A(T) = \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = a_x b_y - a_y b_x + a_y c_x - a_x c_y + b_x c_y - c_x b_y$$

把这个公式推广到高维情形，可以算出  $d$  维空间的单纯形 (*simplex*) 体积的  $d!$  倍。

注意到有向面积可以是负数，因此在使用具体面积之前需要取绝对值。这不是一个 bug，而是一种特性。我们将在 14.1 节中看到如何用有向面积定义出重要的几何操作。

```
double signed_triangle_area(point a, point b, point c)
{
    return( (a[X]*b[Y] - a[Y]*b[X] + a[Y]*c[X]
             - a[X]*c[Y] + b[X]*c[Y] - c[X]*b[Y]) / 2.0 );
}

double triangle_area(point a, point b, point c)
{
    return( fabs(signed_triangle_area(a,b,c)) );
}
```

## 13.3 圆

圆 (*circle*) 是到定点 (称为圆心 (*center*)) 距离为定长 (称为半径 (*radius*)) 的点集。圆盘 (*disk*) 包含圆和它的内部 (*interior*)，即到圆心距离不超过  $r$  的点集。

- 表示 — 圆有两种基本表示法：圆周上的三个点，或者圆心和半径。对于大多数应用，圆心和半径表示法最方便：

```
typedef struct {
    point c;                      /* center of circle */
    double r;                     /* radius of circle */
} circle;
```

由圆心半径表示法很容易得到圆的方程。由于  $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$  表示两点之间的距离，所以圆心为  $(x_c, y_c)$ ，半径为  $r$  的圆方程为： $r = \sqrt{(x - x_c)^2 + (y - y_c)^2}$ ，或者等价地写成  $r^2 = (x - x_c)^2 + (y - y_c)^2$ ，从而避免了平方根运算。

- 周长和面积 — 不少与圆有关的量都很容易计算。面积  $A$  和周长  $C$  的计算都依赖于“魔力数”圆周率  $\pi = 3.141\ 592\ 6$ 。具体来说， $A = \pi r^2$ ， $C = 2\pi r$ 。在众人面前流利地背诵很多位圆周率通常是一个炫耀你记忆力超强的好办法。圆的直径 (*diameter*) 是  $2r$ ，它也是圆内最长的线段长度。
- 切线 — 大多数直线  $l$  和圆周的交点数为 0 或者 2。第一种情况表明直线和圆  $c$  完全不相交，而第二种情况表明直线从圆  $c$  的内部穿过。还有一种情况是直线  $l$  恰好与圆  $c$  的边界接触，但没有穿过它的内部。这样的直线称为切线 (*tangent lines*)。

过点  $O$  与圆  $c$  相切的直线  $l$  如图 13.3 所示。圆  $c$  和线  $l$  的切点在过圆心  $c$ 、与  $l$  垂直的直线上。由于以  $r$ ,  $d$  和  $x$  为边的三角形是直角三角形，可以用勾股定理算出  $x$ ，然后计算出切点或角  $a$ 。点  $O$  到圆心的距离  $d$  可以用距离公式算出。

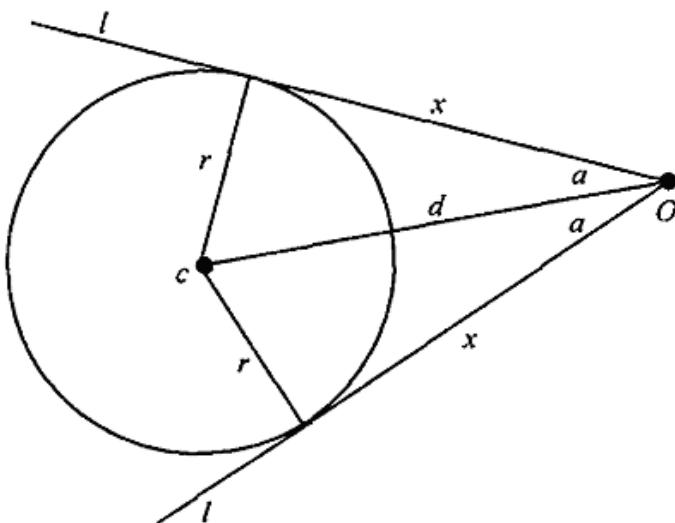


图 13.3 过点  $O$  的圆的切线

- 圆的位置关系 — 半径  $r_1$  和  $r_2$  不同的两个圆  $c_1$  和  $c_2$  可以通过多种方式相交。两圆相交<sup>5</sup>当且仅当圆心距不超过  $(r_1 + r_2)$ 。较小的圆 (不妨设为  $c_1$ ) 被包含在另一个圆  $c_2$  内当且仅当圆心距不超过  $r_2 - r_1$ 。另一种情况是  $c_1$  和  $c_2$  相交于两点。如图 13.4 所示，两

<sup>5</sup> 译者注：这里指圆盘相交。

个交点和两圆圆心分别形成一个三角形，两个三角形的所有边长均已知（分别是 $r_1$ ,  $r_2$  和圆心距 $d$ ），因此三角形各内角与交点坐标也能计算出来。

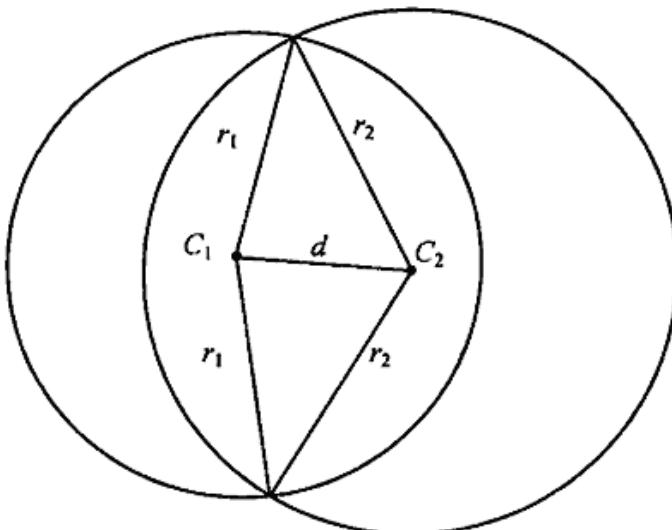


图 13.4 两圆的交点

## 13.4 程序设计举例：超高速飞行

超人最酷的两项能力是常人所没有的：透视眼和超高速飞行。其他能力就没那么吸引了：或许我们也能做到在电话亭里换衣服——只要我们加油干。

超人准备向人们展示他的超能力。首先是透视眼表演，从当前位置 $s = (x_s, y_s)$ 看到目标位置 $t = (x_t, y_t)$ ，其间会有若干圆形障碍物挡住普通人的视线。由于超人的透视能力并不是无限的，他需要在表演前计算出视线将穿越的障碍长度，以决定是否进行这个表演。

接下来是飞行表演。他能看穿物体，但身体却不能穿越。按照计划，他应该（如图 13.5 所示）沿着连接 $s$  和 $t$  的直线飞行，但由于圆形障碍物的存在，沿直线飞行可能会撞上它们。此时，超人需要沿着障碍物的边界飞，直到他回到既定的路线上来，继续沿直线飞行。尽管这不是避开障碍物从 $s$  到 $t$  的最短路线，但你也不能说超人是典型的四肢发达，头脑简单——毕竟在碰到障碍物后，他总是会选择两条路线中较短的一条。

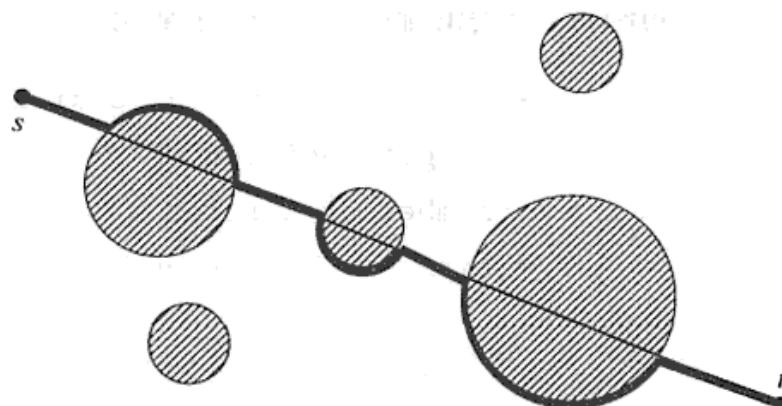


图 13.5 超人的飞行计划和透视长度

你可以假定圆形障碍物两两不相交，起点和终点均在障碍物外。障碍物用圆心坐标和半径表示。你的任务是计算出超人的视线穿越障碍的长度以及飞行总距离。

### 以下为题解

求解这个问题需要三个基本几何运算。我们需要 (1) 判断给定圆是否和给定直线  $l$  相交在起点和终点中间, (2) 计算  $l$  和圆相交的弦长以及 (3) 计算圆被  $l$  截得的较小部分的弧长。

第一个任务相对简单。求出圆心到直线的最短距离, 如果距离小于半径, 则相交, 否则不相交。交点在起点和终点之间当且仅当圆心到  $l$  的最近点在  $s$  和  $t$  之间<sup>6</sup>。

计算与相交有关的量看上去要困难一些。一种常见的方法是先计算出圆和直线的交点坐标。尽管这只需联立直线和圆的方程, 求解整理后的二次方程, 但整个求解过程将会显得乱糟糟的, 完全失去了几何美。往往可以避开坐标求解, 用更简单、直观和优美的方法求解几何问题。

图 13.6 描述了这样的一个简单方法。弦长等于图中的  $2x$ 。我们知道, 图上标注的  $d$  在垂直于  $l$  的直线上, 它与  $l$  形成的四个角都是直角。因此, 以  $r$ 、 $d$  和  $x$  为边组成了两个直角三角形, 用勾股定理很容易算出  $x$ 。

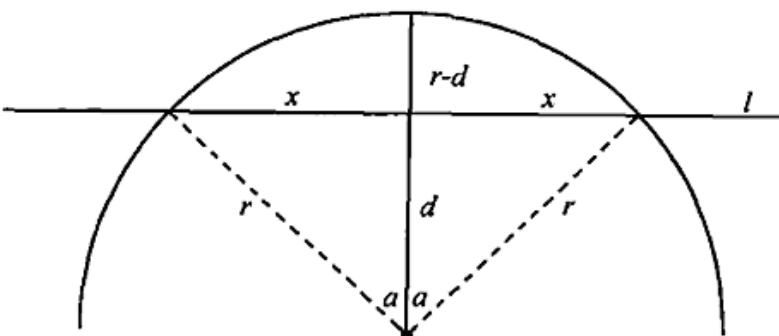


图 13.6 计算直线和圆相交时的弦长和弧长

较短弧长可以通过上述直角三角形中的角  $\alpha$  得到。所求弧对应的圆心角等于  $2\alpha \text{ rad}$ , 因此弧长等于圆周长  $2\pi r$  的  $(2\alpha)/(2\pi)$  倍。角  $\alpha$  很容易用反三角函数算出。

这样一来, 借助前面介绍的例程, 本题的解答就变得非常简单了:

```
point s;                      /* Superman's initial position */
point t;                      /* target position */
int ncircles;                 /* number of circles */
circle c[MAXN];               /* circles data structure */

superman()
```

<sup>6</sup> 译者注: 如果没有“起点和终点保证不在障碍物中”这个条件, 则此结论不成立。

```

line l;                      /* line from start to target position */
point close;                 /* closest point */
double d;                     /* distance from circle-center */
double xray = 0.0;            /* length of intersection with circles */
double around = 0.0;           /* length around circular arcs */
double angle;                 /* angle subtended by arc */
double travel;                /* total travel distance */
int i;                        /* counter */

double asin(), sqrt();
double distance();

points_to_line(s,t,&l);

for (i=1; i<=ncircles; i++) {
    closest_point(c[i].c,l,close);
    d = distance(c[i].c,close);
    if ((d>=0) && (d < c[i].r) && point_in_box(close,s,t)) {
        xray += 2*sqrt(c[i].r*c[i].r - d*d);
        angle = acos(d/c[i].r);
        around += ((2*angle)/(2*PI)) * (2*PI*c[i].r);
    }
}

travel = distance(s,t) - xray + around;
printf("Superman sees thru %7.3lf units, and flies %7.3lf units\n",
      xray, travel);
}

```

## 13.5 三角函数库

不同编程语言的三角函数库往往十分接近。在使用之前，请确认库函数用角度还是弧度来表示角，以及反三角函数返回值的范围。注意，反正弦、反余弦函数并不是在一个完整的  $360^\circ = 2\pi$  rad 周期内可逆，因此你还需要知道库函数使用哪一个  $180^\circ = \pi$  rad 的半周期。

## C/C++ 中的三角函数

标准 C/C++ 数学库 `math.h` 包含了所有标准的三角函数。别忘了在编译时连接数学库<sup>7</sup>:

```
#include <math.h>

double cos(double x);           /* compute the cosine of x radians */
double acos(double x);          /* compute the arc cosine of [-1,1] */

double sin(double x);           /* compute the sine of x radians */
double asin(double x);          /* compute the arc sine of [-1,1] */

double tan(double x);           /* compute the tangent of x radians */
double atan(double x);          /* compute the principal arctan of x */
double atan2(double y, double x); /* compute the arc tan of y/x */
```

库中有两个反正切函数的主要目的是正确的确定返回角所在的象限 (*quadrant*)。它同时取决于 *x* 和 *y* 的符号。

## Java 中的三角函数

Java 三角函数在 `java.lang.Math` 类中。所有角均用弧度表示，但也提供角度和弧度相互转换的函数。所有函数均是该类的静态成员函数，功能和 C 函数非常类似:

<code>double cos(double a)</code>	Return the trigonometric cosine of angle a.
<code>double acos(double a)</code>	Return the arc cosine of angle a, an angle in [0,pi].
<code>double sin(double a)</code>	Return the trigonometric sine of angle a.
<code>double asin(double a)</code>	Return the arc sine of angle a, an angle in [-pi/2,pi/2].
<code>double tan(double a)</code>	Return the trigonometric tangent of angle a.
<code>double atan(double a)</code>	Return the arc tangent of angle a, an angle in [-pi/2,pi/2]
<code>double atan2(double a, double b)</code>	Convert (b, a) to polar (r, theta)
<code>double toDegrees(double angrad)</code>	Convert a radian angle to degrees.
<code>double toRadians(double angdeg)</code>	Convert a degree angle to radians.

<sup>7</sup> 译者注: gcc 在编译 C++ 程序时会自动连接数学库, 但编译 C 程序时默认不连接数学库。在编译选项中添加 “-lm” 可手动连接。

## 13.6 习 题

### 13.6.1 狗拿地鼠 (Dog and Gopher)

PC/UVA 题号: 111301/10310, 流行度: A, 通过率: average 难度: 1

在一块地里有一只狗和一只地鼠。狗想要吃掉地鼠，而地鼠则想要通过地里的几个地洞跑到一个安全的地方。

狗和地鼠都不擅长数学；但是，也都不是彻底的笨蛋。地鼠会选一个地洞然后以恒定的速度沿直线向这个洞跑去。而狗则非常擅长解读肢体语言，它能预见地鼠想要向哪一个洞跑，然后以两倍于地鼠的速度向这个洞跑去。如果狗先到达这个洞，地鼠就会被吃掉；否则，地鼠就可以逃离。

地鼠请你替它选择一个可以逃离的洞口（如果这样的洞口存在）。

#### 输入

输入包含多组数据。每组数据第一行包含一个整数和四个实数。整数  $n$  表示该组数据中一共有多少个地洞。接下来的四个实数首先标明地鼠的坐标  $(x, y)$ ，然后是狗的坐标  $(x, y)$ 。接下来  $n$  行每行包含两个整数，表示一个地洞的坐标  $(x, y)$ 。所有距离均以米为单位，并已四舍五入精确到毫米。相邻两组数据间用一个空行隔开，输入数据以文件结束符结尾。

#### 输出

对于每组输入数据输出一行。如果地鼠能够逃离，则输出一行 “The gopher can escape through the hole at  $(x, y)$ .”。其中  $(x, y)$  为地洞坐标（精确到毫米）。否则，输出一行 “The gopher cannot escape.”。如果地鼠可以通过不止一个地洞逃离，输出输入中最早出现的地洞。每组数据最多包含 1 000 个地洞且坐标值在 -10 000 和 +10 000 之间。

#### 样例输入

```
1 1.000 1.000 2.000 2.000
1.500 1.500
```

```
2 2.000 2.000 1.000 1.000
1.500 1.500
2.500 2.500
```

#### 样例输出

```
The gopher cannot escape.
The gopher can escape through the hole at (2.500,2.500).
```

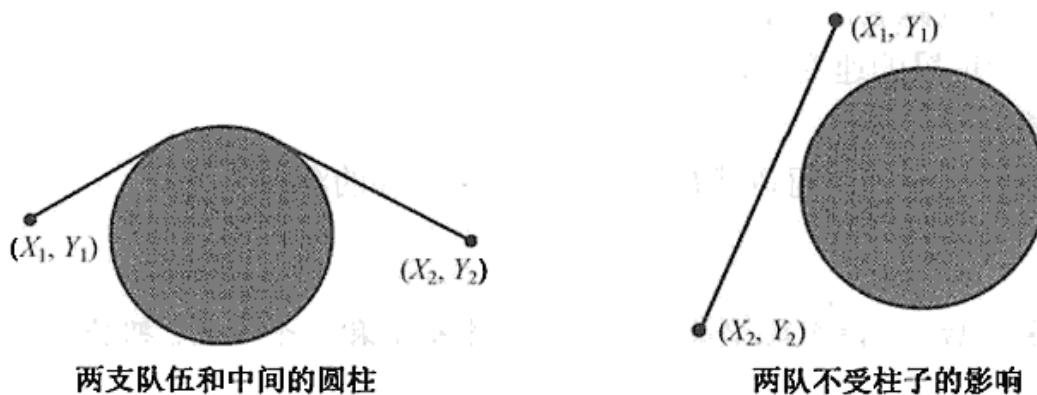
### 13.6.2 绳子王国的危机! (Rope Crisis in Ropeland!)

PC/UVa 题号: 111302/10180, 流行度: B, 通过率: average 难度: 2

拔河在绳子国很流行, 就像板球在孟加拉国很流行一样。两队人抓住绳子的两端使劲拉, 将绳子拉向自己一方的队伍获胜。

由于绳子短缺, 国王下令参赛队不能购买长于他们需要的绳子。

拔河比赛在一个大房间里举行, 其中有一根巨大的圆形柱子。如果两队分别在柱子的相对两面, 他们所拉的绳子就不能形成一条直线。给出两个队的位置, 找出进行拔河比赛需要的最小绳子长度。你可以假设用一个点代表队伍的位置。



#### 输入

输入数据第一行包含一个整数  $N$ , 表示测试数据的组数。接下来  $N$  行, 每行包含 5 个整数  $X_1, Y_1, X_2, Y_2, R$ , 其中  $(X_1, Y_1)$  和  $(X_2, Y_2)$  为两队位置的坐标, 而  $R > 0$  为柱子的半径。

柱子的中心总是坐标原点, 假定不会有某队在柱子里面。所有输入数据除  $N$  外均为实数, 绝对值均不超过 10 000。

#### 输出

对于每组数据, 输出一行, 包含一个实数, 为最小绳子长度 (四舍五入保留三位小数)。

#### 样例输入

```
2
1 1 -1 -1 1
1 1 -1 1 1
```

#### 样例输出

```
3.571
2.000
```

### 13.6.3 圆桌骑士 (The Knights of the Round Table)

PC/UVa 题号: 111303/10195, 流行度: A, 通过率: average 难度: 2

亚瑟王想在一个有三角形天窗的房间里做一张圆桌。他想要阳光能够照到圆桌上。尤其

在正午太阳在头顶的时候，他希望整个圆桌能够被笼罩在阳光下。

因此圆桌只能被放在房间中一个特定的三角形区域内。亚瑟王当然希望做一张满足条件的最大的圆桌。

由于大法师梅林在外就餐，请你写一个程序来算出这个最大圆桌的半径。

### 输入

输入数据为多组，每组由三个实数  $(a, b, c)$  表示，代表三角形区域的三边长。边长不超过 1 000 000，保证  $\max(a, b, c) \leq (a + b + c)/2$ 。

输入数据用文件结束符结尾。

### 输出

对于每组数据，输出如下一行：

The radius of the round table is:  $r$

其中  $r$  表示最大圆桌的半径，四舍五入保留三位小数。

### 样例输入

12.0 12.0 8.0

### 样例输出

The radius of the round table is: 2.828

### 13.6.4 巧克力片饼干 (Chocolate Chip Cookies)

PC/UVa 题号: 111304/10136, 流行度: C, 通过率: average 难度: 3

做巧克力片饼干需要混合面粉、盐、油、苏打粉和巧克力片，把它们揉成一个面团，然后擀成边长不超过 50cm 的方饼。从这张饼上切出一个一个的小圆饼放在饼干纸上，烘烤 20 分钟。烤好的饼干从烤箱中取出，冷却后就可以吃了。

我们感兴趣的 是在和好面团擀出面饼之后切圆饼的过程。在面饼里可以清楚地看到每片巧克力片，所以我们只需要选择按下模具的位置，使切割出来的巧克力片尽可能多。

### 输入

输入数据开头单独一行一个整数表示测试数据的组数。接下来是一个空行。每两组数据间均用一个空行隔开。

每组数据包含若干行，每行两个实数，表示在面饼表面的巧克力片的位置坐标  $(x, y)$ 。坐标在 0.0 ~ 50.0 之间 (单位: cm)。每片巧克力片可以看作一个点。至多有 200 片巧克力片，且位置各异。

**输出**

对于每组数据输出单独一个整数：在一片直径 5cm 的饼干上至多有多少片巧克力片。切出的饼干不一定要完全在方饼内部，也就是说，可以有一边是直的。

相邻两组数据的输出之间应用一个空行隔开。

**样例输入**

1

```
4.0 4.0
4.0 5.0
5.0 6.0
1.0 20.0
1.0 21.0
1.0 22.0
1.0 25.0
1.0 26.0
```

**样例输出**

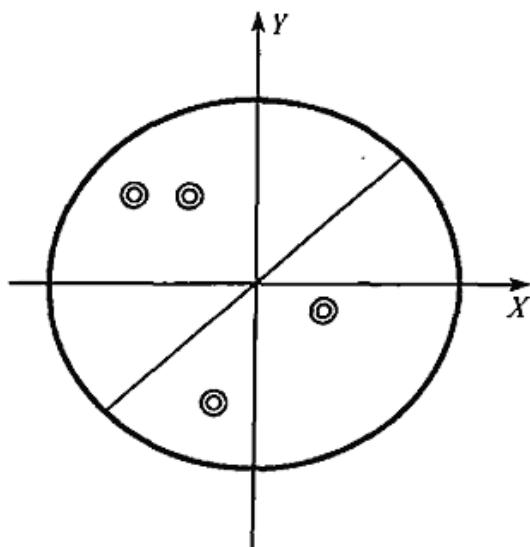
4

**13.6.5 生日蛋糕 (Birthday Cake)**

PC/UVa 题号: 111305/10167, 流行度: C, 通过率: average 难度: 2

Lucy 和 Lily 是双胞胎姐妹。今天是她们的生日，妈妈为她们买了一个生日蛋糕。

蛋糕上有  $2N$  颗樱桃 ( $1 \leq N \leq 50$ )。妈妈希望能够沿着直线一刀把蛋糕切成相等的两份，并且有相等数量的樱桃。你可以帮助她吗？



蛋糕半径为 100，中心坐标为  $(0, 0)$ 。每颗樱桃的位置由整数坐标  $(x, y)$  给出。你要用  $Ax + By = 0$  这样的形式表示这条直线， $A$  和  $B$  均为  $[-500, 500]$  范围内的整数。直线上不

能有樱桃。输入保证至少有一组解。

### 输入

输入包含多组数据。每组数据第一行为整数  $N$ 。接下来  $2N$  行，每行一个坐标  $(x, y)$ ，表示一颗樱桃的位置， $x$  和  $y$  由空格分开。 $N = 0$  表示输入结束。

### 输出

对于每组数据，输出一行，包含由一个空格分隔的两个整数  $A$  和  $B$ 。如果有多个解，输出任意一组。

#### 样例输入

```
2
-20 20
-30 20
-10 -50
10 -5
0
```

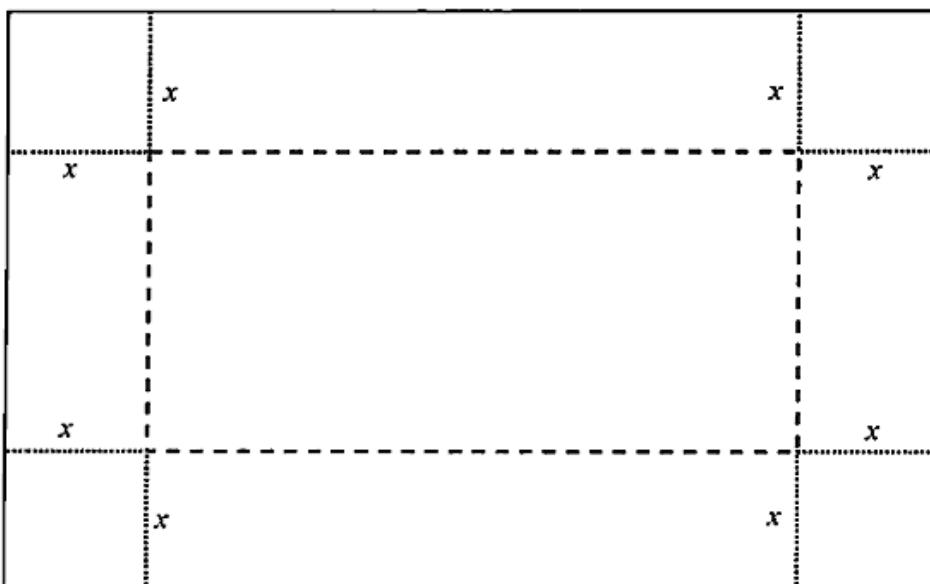
#### 样例输出

```
0 1
```

### 13.6.6 最大/最小的盒子 (The Largest/Smallest Box...)

PC/UVa 题号: 111306/10215, 流行度: A, 通过率: average 难度: 2

下图展示了一个长  $L$ , 宽  $W$  的矩形纸板, 厚度为 0。如图中虚线所示切去四角的四个  $x \times x$  的方块, 就可以把纸板折成一个没有盖子的纸盒。



给出纸板的长和宽, 求出  $x$  的值, 使得纸盒的容积最大/最小。

## 输入

输入数据包含多行。每行包含两个正实数  $L(0 < L < 10\,000)$  和  $W(0 < W < 10\,000)$ ，分别表示纸板的长和宽。

## 输出

对于输入数据的每行，输出一行，包含两个或多个实数（四舍五入保留三位小数），相邻两个数之间用一个空格分隔。第一个数表示使纸盒容积最大的  $x$  的值，后面的数表示使纸盒容积最小的  $x$  的值，按升序排列。

### 样例输入

1 1  
2 2  
3 3

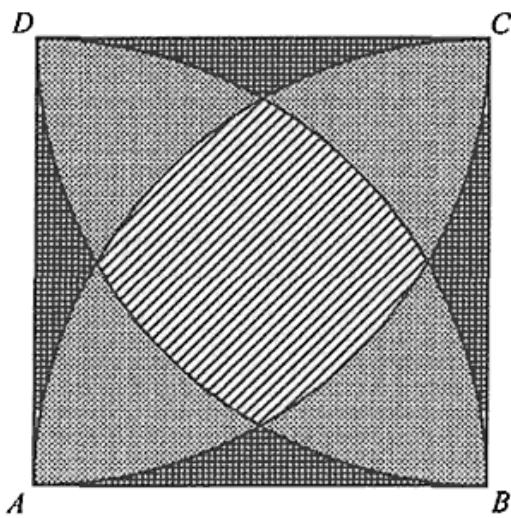
### 样例输出

0.167 0.000 0.500  
0.333 0.000 1.000  
0.500 0.000 1.500

## 13.6.7 要算积分吗?(Is This Integration?)

PC/UVa 题号: 111307/10209, 流行度: A, 通过率: high 难度: 3

下图是一个正方形  $ABCD$ ，其中  $AB = BC = CD = DA = a$ 。分别以  $A$ 、 $B$ 、 $C$ 、 $D$  为圆心，以  $a$  为半径作四条弧。以顶点  $A$  为圆心的弧开始于相邻顶点  $B$ ，结束于相邻顶点  $D$ 。其他弧用类似方法画出。以此形成了三种不同的阴影部分。你要算出各不同阴影部分的总面积。



## 输入

输入数据每行包含一个实数  $a$ ，表示正方形的边长。 $0 \leq a \leq 10\,000.0$ 。输入数据以文件结束符结尾。

## 输出

对于每组数据，单独输出一行，包含三个实数，为上图中不同区域的面积。其中第一个数为图中斜线部分的面积，第二个数为图中打点部分的面积，第三个数为剩下部分的面积。每个实数四舍五入保留三位小数。

### 样例输入

```
0.1
0.2
0.3
```

### 样例输出

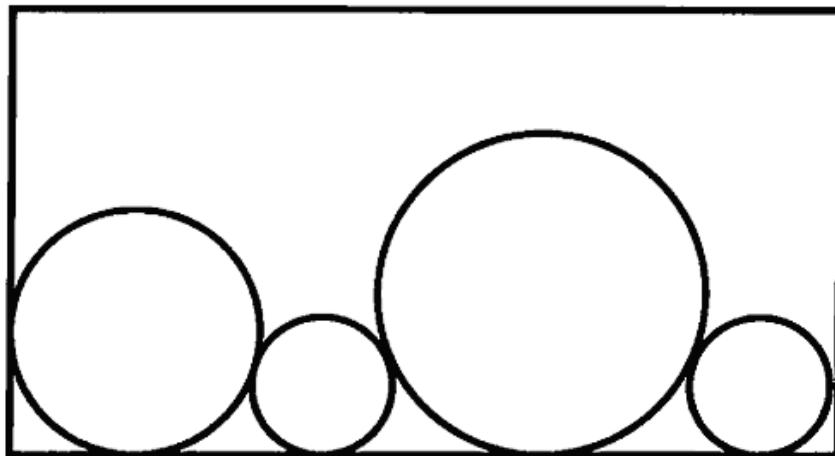
```
0.003 0.005 0.002
0.013 0.020 0.007
0.028 0.046 0.016
```

## 13.6.8 它有多大?(How Big Is It?)

PC/UVa 题号: 111308/10012, 流行度: B, 通过率: low 难度: 3

Ian 要到加利福尼亚去，因此需要将他的东西打包，包括他收藏的很多圆环。给你一个圆环集合，编程找出一个最小矩形可以容纳所有的圆环。

所有的圆必须与盒子的底部相切。下图展示了一种合理的打包方式，尽管不是最优的。正如你可能已经察觉到的，在最优方案中，每个圆都至少和另外一个圆外切。



## 输入

输入数据第一行包含一个正整数  $n(n \leq 50)$ ，表示接下来测试数据的组数。以下  $n$  行每行包含一系列由空格分隔的数。每行第一个数为正整数  $m(m \leq 8)$ ，表示这一行接下来还有多少个数。接下来的  $m$  个数为需要打包在一个盒子里的各个圆的半径。这些数不一定是整数。

## 输出

对于每组数据，输出能够将这些圆打包的最小矩形的大小。每组数据单独输出一行，精确到小数点后 3 位。输出数据不能有前导的 0，除非这个数小于 1(如 0.543)。

最新編  
網文大書站  
擇言堂

样例输入	样例输出
3	9.657
3 2.0 1.0 2.0	16.000
4 2.0 2.0 2.0 2.0	12.657
3 2.0 1.0 4.0	

### 13.7 提 示

- 13.1 离地鼠最近的洞真的是它最安全的去处吗？
  - 13.2 计算柱子之间的公切线是否有用？
  - 13.3 最大的圆桌肯定和三角形的几条边接触？
  - 13.4 是否总是能移动圆形模具，使得边界上有一些巧克力片？如果是，至少能让边界上有几片？这些巧克力片的位置和模具的半径是否定义了所有“有用的”模具放置方案？
  - 13.5 如果切割线方程的系数不必是整数，则本题一定有解——你能证明这一点吗？除了穷举所有可能的  $A$  和  $B$  外，是否有更加高效的方法？
  - 13.6 当盒子的容积为零时， $x$  的值是多少？微积分是否能帮助我们求出最大容积？
  - 13.7 是否可以借助容斥原理把复杂的区域变成一些容易计算面积的部分？
  - 13.8 从大到小依次放置这些圆比较好，还是交叉放置比较好？这些圆的顺序是否重要？回溯法能解决本题吗？

## 第14章 计算几何

几何计算在计算机图形学、机器入学、计算机辅助设计等领域中扮演着越来越重要的角色，因为真实物体都是具有几何形状的。但现实世界中的物体不是由两端无限延伸的直线组成，多数计算机程序采用直线段的排列来表示物体的几何形状。任意闭合曲线或封闭图形都可以用曲线段序列来表示，我们称由直线段序列表示的几何形状为多边形 (*polygons*)。

在本书中，计算几何 (computational geometry) 是指离散线段和多边形的几何。这是一个有趣的学科，但是一般来说不属于大学中的必修课程。计算几何为有志在比赛中取得好成绩的选手提供了高人一筹的机会，它本身也是一个当今算法研究的热门领域。有一些计算几何方面的优秀著作 [O'R00, dBvKOS00]，但本章的内容足以引领你入门。

### 14.1 线段及其相交

线段 (*line segment*)  $s$  是直线  $l$  在两个给定点之间 (包含这两个点) 的部分。因此，线段最自然的表示法就是一对端点：

```
typedef struct {
    point p1,p2;           /* endpoints of line segment */
} segment;
```

线段最基本的几何操作——判断两条线段是否相交，因易错的特殊情况而复杂得令人吃惊。两条线段可能位于平行线上，因此没有交点；一条线段可能与另一条线段在后者的端点处相交；两条线段甚至可能会部分重叠，使得它们的交集是一条线段而非一个孤立的点。

几何中的特殊情况 (也称退化) 使得“为计算几何算法编写鲁棒的代码实现”复杂了许多。处理退化情形往往是痛苦的。仔细阅读题目描述，看看它是否指明不存在平行直线或者部分重叠的线段。如果没有上述保证，你最好采用防御式编程<sup>1</sup>，显式的处理这些特殊情况。

处理退化的正确方式是把所有运算都建立在少量精心打造的几何基本操作上。在第 13 章中，我们实现了一个可以处理竖线 (斜率为无穷大) 的一般化直线数据类型，现在是时候让它发挥作用了——把已有的直线求交函数一般化成线段求交函数：

```
bool segments_intersect(segment s1, segment s2) {
    line l1,l2;           /* lines containing the input segments */
```

<sup>1</sup> 译者注：防御式编程是指先特判出输入的特殊情况并作相应处理，最后才是处理正常数据的主过程；进攻式编程是直接执行主过程，实在无法执行下去（例如程序抛出了异常）时才作特殊处理（例如捕获异常）。

```
point p; /* intersection point */

points_to_line(s1.p1,s1.p2,&l1);
points_to_line(s2.p1,s2.p2,&l2);

if (same_lineQ(l1,l2)) /* overlapping or disjoint segments */
    return( point_in_box(s1.p1,s2.p1,s2.p2) ||
           point_in_box(s1.p2,s2.p1,s2.p2));

if (parallelQ(l1,l2)) return(FALSE);

intersection_point(l1,l2,p);

return(point_in_box(p,s1.p1,s1.p2) && point_in_box(p,s2.p1,s2.p2));
}
```

我们首先用直线求交函数找到两条线段所在直线的交点（如果存在的话）。如果有交点，剩下的问题就是判断该交点是否分别位于两条线段上。这可以进一步简化为：判断交点是否在线段的包围盒（bounding box）内，该包围盒由线段的两个端点所定义：

```
bool point_in_box(point p, point b1, point b2) {
    return( (p[X] >= min(b1[X],b2[X])) && (p[X] <= max(b1[X],b2[X]))
        && (p[Y] >= min(b1[Y],b2[Y])) && (p[Y] <= max(b1[Y],b2[Y])) );
}
```

线段相交还可以简洁的用“三个有序点的旋转方向”来测试（这个基本操作将在下节介绍）。不过我们认为 `point_in_box` 更加直观。

## 14.2 多边形及旋转方向

多边形 (*polygons*) 是不自交的闭合折线段。闭合指的是折线的第一个端点和最后一个端点重合，不自交指的是不同线段只在端点处相交<sup>2</sup>。

多边形是描述平面图形的基本结构。除了显示的列举出所有线段（也称为边 (*edges*)）之外，还可以按照多边形边界上的顺序隐式的列举出所有  $n$  个顶点，用这个顶点列表来描述多边形。这样，对于任意  $0 \leq i \leq n - 1$ ，第  $i$  个顶点和第  $(i + 1)$  个顶点之间的线段是折线

<sup>2</sup> 译者注：并且非相邻线段不能相交，哪怕只在端点处相交。

的一部分。这里的下标均在  $\text{mod } n$  意义下进行，以确保最后一个点和第一个点之间也有边相连：

```
typedef struct {
    int n;                      /* number of points in polygon */
    point p[MAXPOLY];          /* array of points in polygon */
} polygon;
```

多边形  $P$  是凸的 (*convex*) 当且仅当  $P$  内任两个顶点的连线完全位于  $P$  的内部；这表明：凸多边形的所有内角均小于  $180^\circ = \pi$  弧度。

计算三个有序点之间的转角并不那么容易。幸运的是，在大多数几何算法中，我们可以用“逆时针旋转”谓词 `ccw(a,b,c)` 来避开实际的转角值计算。该函数判断：站在点  $a$  处往点  $b$  的方向望去，点  $c$  是否位于  $\vec{ab}$  的左侧。如果是，则从  $a$  到  $b$  后逆时针旋转一个小于  $180$  度的角后能到  $c$ ，该谓词因此得名。如果并不是，则要么  $c$  位于  $\vec{ab}$  右侧，要么三点共线。

以上谓词可以用 13.2.3 小节介绍的 `signed_triangle_area()` 公式来实现。当  $c$  位于  $\vec{ab}$  左侧时有向面积为正，三点共线时面积为零。为了减小浮点误差带来的影响，我们在实数比较时使用了一个很小的正常数  $\epsilon$ ，而不是直接与零比较。这只是权益之计，因为要把包含浮点运算的几何代码写得近乎完美的鲁棒几乎是不可能的。尽管如此，上述技巧总比什么都不做要强得多。

```
bool ccw(point a, point b, point c) {
    double signed_triangle_area();

    return (signed_triangle_area(a,b,c) > EPSILON);
}
```

```
bool cw(point a, point b, point c) {
    double signed_triangle_area();

    return (signed_triangle_area(a,b,c) < -EPSILON);
}
```

```
bool collinear(point a, point b, point c) {
    double signed_triangle_area();

    return (fabs(signed_triangle_area(a,b,c)) <= EPSILON);
```

}

### 14.3 凸 包

凸包在计算几何中的地位相当于排序在算法中的地位，它是处理非结构化数据的第一步，也是后续复杂处理的基础。点集  $S$  的凸包 (*convex hull*)  $C(S)$  是包含  $S$  的最小凸多边形，如图 14.1(a) 所示。

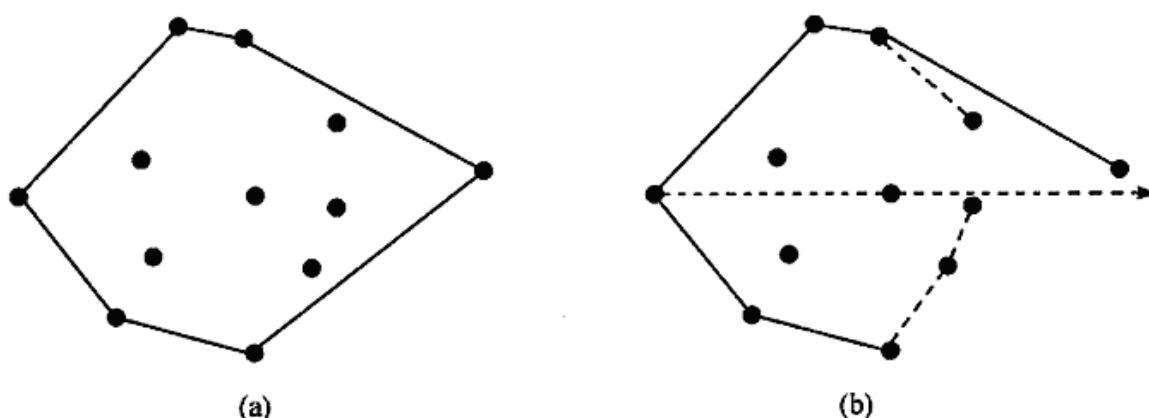


图 14.1 点集的凸包 (a), 插入最右点引起凸包变化 (b)

几乎可以这样说：有多少种排序算法，就有多少种凸包算法。其中，我们即将实现的 Graham 扫描算法先按照极角或从左到右的顺序给点集排序，然后按照该顺序依次在凸包中插入各个点。插入新点后，原来凸包上的无用点将被删除。

我们的代码实现依照 Gries 和 Stojmenović [GS87] 的版本，以最左点中的最低点作为参考点，按照极角从小到大给点集排序。注意到最左点和最低点都一定在凸包中，因为其他点所组成的多边形无法包含它们。为了确定极角计算的参考点，当有多个点都是“最左点”时，应取其中的最低点。这样的选择对于正确的处理退化输入是必要的。

算法的主循环按照到参考点（它也是凸包中的第一个点）的极角从小到大的顺序依次插入各个点。该顺序保证了新插入的点总是在已插入点组成的凸包边界上。这个新插入的点可能会形成三角形，把一些原先凸包上的点包含在内，因此这些点都应从凸包中删除。这些待删除的点都在当前凸包链的尾部，它们是“幸存”于凸包上的最新插入点。是否删除点的关键是：新插入的点与链上最后两个点形成的角是否小于 180 度——别忘了凸多边形的内角均小于 180 度。如果角度太大，链上最后一个点必须被删除。重复这一过程，直到这个角度小于 180 度，或者所有点都已处理完。可以用 `ccw()` 谓词来判断角是否过大：

```
point first_point;           /* first hull point */

convex_hull(point in[], int n, polygon *hull) {
    int i;                    /* input counter */
```

```
int top;           /* current hull size */
bool smaller_angle();

if (n <= 3) {      /* all points on hull! */
    for (i=0; i<n; i++)
        copy_point(in[i],hull->p[i]);
    hull->n = n;
    return;
}

sort_and_remove_duplicates(in,&n);
copy_point(in[0],&first_point);

qsort(&in[1], n-1, sizeof(point), smaller_angle);

copy_point(first_point,hull->p[0]);
copy_point(in[1],hull->p[1]);

copy_point(first_point,in[n]); /* sentinel for wrap-around */
top = 1;
i = 2;

while (i <= n) {
    if (!cw(hull->p[top-1], hull->p[top], in[i]))
        top = top-1; /* top not on hull */
    else {
        top = top+1;
        copy_point(in[i],hull->p[top]);
        i = i+1;
    }
}

hull->n = top;
}
```

该实现的优美之处在于它很自然的避免了多数退化问题。一个尤其容易出错的退化情况是三点共线，特别是当其中一个点正是凸包起始点（最左点中的最低点）时。稍有大意，我们就会在凸包的一条边上留下三个共线点，但实际上只有两个端点属于凸包<sup>3</sup>。

为了解决这一问题，我们在按极角排序时规定：当极角相等时以到初始点的距离为顺序排序。只要最远的点最后插入，我们就能保证它能留在凸包上，而不是被同一极角的其他点所代替：

```
bool smaller_angle(point *p1, point *p2) {
    if (collinear(first_point,*p1,*p2)) {
        if (distance(first_point,*p1) <= distance(first_point,*p2))
            return(-1);
        else
            return(1);
    }

    if (ccw(first_point,*p1,*p2))
        return(-1);
    else
        return(1);
}
```

最后一种退化情况是关于重复点的：三个点重合时，转角如何定义？为了避免这一问题，我们在给输入点排序以找到凸包起始点的同时去掉重复点<sup>4</sup>：

```
sort_and_remove_duplicates(point in[], int *n) {
    int i;                      /* counter */
    int oldn;                    /* number of points before deletion */
    int hole;                    /* index marked for potential deletion */
    bool leftlower();

    qsort(in, *n, sizeof(point), leftlower);

    oldn = *n;
    hole = 1;
```

<sup>3</sup> 译者注：有些题目特别指明要输出位于凸包边界上的所有点，此时就还应该输出中间那个点。

<sup>4</sup> 译者注：这里给出的代码排序了两次，但其实只按极角排序一次即可，不必先按坐标排序。尽管如此，排序两次的代码更加容易理解。

```

for (i=1; i<oldn; i++) {
    if ((in[hole-1][X]==in[i][X]) && (in[hole-1][Y]==in[i][Y]))
        (*n)--;
    else {
        copy_point(in[i],in[hole]);
        hole = hole + 1;
    }
}
copy_point(in[oldn-1], in[hole]);
}

bool leftlower(point *p1, point *p2) {
    if ((*p1)[X] < (*p2)[X]) return (-1);
    if ((*p1)[X] > (*p2)[X]) return (1);

    if ((*p1)[Y] < (*p2)[Y]) return (-1);
    if ((*p1)[Y] > (*p2)[Y]) return (1);

    return(0);
}

```

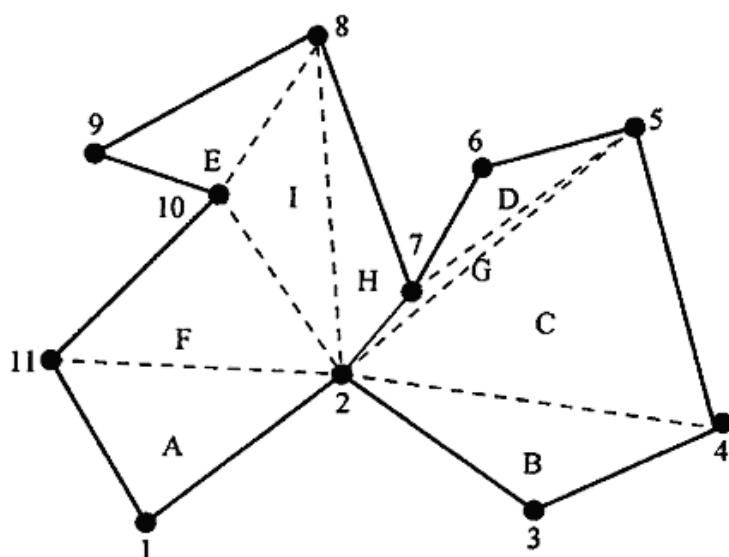


图 14.2 用 Van Gogh 算法 (也称切耳算法) 给一个多边形进行三角剖分。  
得到的三角形按照插入顺序编号为 A-I

最后，我们还想对 `convex_hull` 作点说明。代码巧妙地用到了哨兵来简化代码：把原始点拷贝到插入链的尾部，以避免显式的检测回环条件 (wrap-around condition)。最后在程序

结束之前正确设置凸包上的点数，以隐式的删除这个人工重复点。

最后，在按极角排序时并不需要计算角度的具体值。`ccw` 谓词完全可以胜任这项工作。

## 14.4 三角剖分：算法与相关问题

求多边形的周长很容易：用距离公式算出所有边的长度，加在一起就可以了。相比之下，计算不规则图形的面积就要困难一些了。最容易想到的方法也许是把多边形分成不相交的三角形，然后把这些三角形的面积加起来。把多边形划分成三角形的操作称为三角剖分 (*triangulation*)。

给凸多边形进行三角剖分很容易，因为我们只需用其中一个顶点  $v$  连接所有其他  $n - 1$  个顶点，得到一个类似于风扇的图形。但是，这个方法并不适用于一般多边形，因为这些连接线可能会位于多边形的外部。我们必须用完全位于多边形  $P$  内部的不相交弦 (chords) 把  $P$  切成三角形。

为了表示三角剖分，我们既可以列举出所用的弦，也可以像下面的代码一样显式的给出每个三角形的三个顶点编号。

```
typedef struct {
    int n;           /* number of triangles in triangulation */
    int t[MAXPOLY][3]; /* indices of vertices in triangulation */
} triangulation;
```

### 14.4.1 Van Gogh 算法

多边形的三角剖分有很多算法，其中最高效的算法时间复杂度和顶点数同阶。但其中最容易实现的算法基于是“切耳 (ear-cutting)”的。多边形  $P$  的耳 (*ear*) 是指由一个顶点  $v$  和它的左右邻居 ( $l$  和  $r$ ) 组成的三角形  $(v, l, r)$ ，它完全在  $P$  的内部。

由于  $\vec{lv}$  和  $\vec{vr}$  都是  $P$  边界上的线段，这个耳可以由弦  $\vec{rl}$  唯一确定。在什么样的情况下，这条弦属于一个三角剖分？首先， $\vec{rl}$  必须完全位于  $P$  的内部，这意味着内角  $lvr$  必须小于 180 度。其次，这条弦不得穿过多边形边界上的其他线段，否则这个“耳”就会像是被咬掉了一口似的不完整。

有一条重要事实：每个 多边形都有耳。事实上，当  $n \geq 4$  时至少有两个。据此可以得到以下算法：测试多边形的每个顶点，直到发现一个耳。把相应的弦加入三角剖分中，然后用它把耳切掉，从而让多边形的顶点数减少 1。对剩下的多边形重复上述切耳过程，直到只剩三个点，组成一个三角形。

需要两个步骤来测试一个顶点是否定义了一个耳。首先判断内角是否小于 180 度。这用我们的 `ccw/cw` 谓词很容易做到。这里需要注意多边形上顶点的排列顺序。我们假设多边形

上的顶点按照逆时针顺序标号，如图 14.2 所示。如果多边形上的顶点序相反，则角度测试中的符号也得反过来。

```
bool ear_Q(int i, int j, int k, polygon *p) {
    triangle t;                      /* coordinates for points i,j,k */
    int m;                            /* counter */
    bool cw();

    copy_point(p->p[i],t[0]);
    copy_point(p->p[j],t[1]);
    copy_point(p->p[k],t[2]);

    if (cw(t[0],t[1],t[2])) return(FALSE);

    for (m=0; m<p->n; m++) {
        if ((m!=i) && (m!=j) && (m!=k))
            if (point_in_triangle(p->p[m],t)) return(FALSE);
    }

    return(TRUE);
}
```

接下来判断这条弦是否穿过多边形上的其他线段，这只需检查是否有顶点位于这个耳对应的三角形内。如果这个耳内部没有顶点，则这条弦没有穿越其他线段，因为  $P$  本身是不自交的。判断给定点是否在三角形内将在 14.4.3 小节中讨论。

这样，三角剖分主算法就很简单了：对所有顶点进行测试，一旦发现耳就把它切掉。在数组表示法中，顶点  $i$  的邻居很容易找到：就是顶点  $(i - 1)$  和顶点  $(i + 1)$ 。不过，在数组中删除顶点就不是很方便了。为了解决这一问题，我们定义两个辅助数组  $l$  和  $r$ ，表示每个顶点在当前多边形中的左邻居和右邻居：

```
triangulate(polygon *p, triangulation *t) {
    int l[MAXPOLY], r[MAXPOLY];      /* left/right neighbor indices */
    int i;                            /* counter */

    for (i=0; i<p->n; i++) {        /* initialization */
        l[i] = ((i-1) + p->n) % p->n;
        r[i] = ((i+1) + p->n) % p->n;
    }
```

```
}

t->n = 0;
i = p->n-1;
while (t->n < (p->n-2)) {
    i = r[i];
    if (ear_Q(l[i], i, r[i], p)) {
        add_triangle(t, l[i], i, r[i], p);
        l[ r[i] ] = l[i];
        r[ l[i] ] = r[i];
    }
}
}
```

#### 14.4.2 面积计算

经过三角剖分后，多边形的面积就可以由各个小三角形的面积加起来得到了。有了前面介绍的过程后，这一步是很容易实现的。

其实还有一个更巧妙的算法，它充分挖掘了有向面积的性质。三角形的有向面积也是我们的 ccw 例程的基础。在平面上任取一点  $p$ ，连接它和  $P$  边界上的每条线段，我们将得到若干个三角形。这些三角形的有向面积之和便是多边形的面积，因为“负面积”正好把这些三角形露在  $P$  外面的部分抵消掉。这样，面积计算公式可以简化成：

$$A(P) = \frac{1}{2} \sum_{i=0}^{n-1} (x_i \cdot y_{i+1} - x_{i+1} \cdot y_i)$$

其中所有下标运算均在模  $n(P$  的顶点数) 意义下进行。这个公式甚至不需要用到我们的 signed\_area 函数！[O'R00] 中解释了这样做的道理，但不管是否理解其中的原由，代码毫无疑问是简单的：

```
double area(polygon *p) {
    double total = 0.0;                      /* total area so far */
    int i, j;                                /* counters */

    for (i=0; i<p->n; i++) {
        j = (i+1) % p->n;
        total += (p->p[i][X]*p->p[j][Y]) - (p->p[j][X]*p->p[i][Y]);
    }
}
```

```
    return(total / 2.0);
}
```

#### 14.4.3 点定位

我们的三角剖分算法要求耳的内部不能有多边形的顶点。因此，耳的判定需要判断一个给定点  $p$  是否位于三角形  $t$  的内部。

三角形总是凸多边形，因为无法形成凹点。一个点位于一个凸多边形内当且仅当它在所有有向线段  $\overrightarrow{p_i p_{i+1}}$  的左侧（这要求多边形的顶点按照逆时针顺序排列）。借助 `ccw` 谓词很容易判断这一点：

```
bool point_in_triangle(point p, triangle t) {
    int i; /* counter */
    bool cw();

    for (i=0; i<3; i++)
        if (cw(t[i],t[(i+1)%3],p)) return(FALSE);

    return(TRUE);
}
```

上述算法解决的是凸多边形的点定位 (*point location*) 问题（即判断点在  $P$  的内部、外部还是边界上），但不适用于任意多边形——想象一下，用这个算法判断点是否位于一个复杂的螺旋状多边形内部会怎样。尽管如此，任意多边形的情形仍然可以借助前面的代码轻松解决：用 `triangulate` 把多边形剖分成若干三角形，然后依次判断每个三角形是否包含给定点。如果其中一个三角形包含该点，则点位于多边形内<sup>5</sup>。

不过，和面积计算一样，用三角剖分来解决点定位问题可谓杀鸡用牛刀。有一个基于 *Jordan 曲线定理* (*Jordan curve theorem*) 的算法要简单许多。该定理说的是：每个多边形或其他闭合图形都有一个内部和一个外部。只要不穿过边界，你就无法从内部走到外部，也无法从外部走到内部。

利用该结论可以得到如下算法，如图 14.3 所示：从给定点  $q$  出发画一条射线  $l$ ，如果该射线穿越边界的次数为偶数，则  $q$  在  $P$  的外部。原因在于，射线在“足够远处”位于多边形外部。从这个“足够远处”出发逆着射线的方向朝着  $q$  走，每两次穿越边界都会重新回到多边形外部。同理，若穿越次数为奇数，则  $q$  在  $P$  的内部。

这个算法存在几个重要的退化情形。当射线穿过多边形  $P$  的一个顶点时，只有当射线穿进  $P$  的内部，而不是与它“擦肩而过”时才算穿越边界。换句话说，射线在顶点处穿越边

<sup>5</sup> 译者注：但如果点在某些三角形的边界上，则并不代表该点一定在多边形的边界上。这一点需要特别注意。

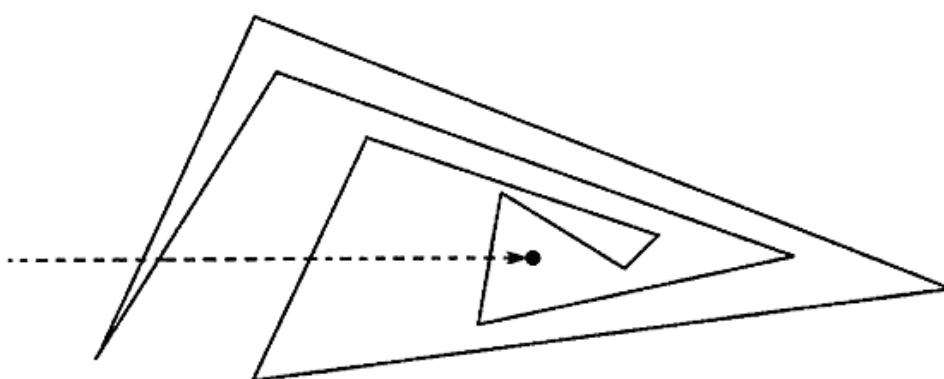


图 14.3 穿越边界次数的奇偶性决定了点是否在多边形内

界当且仅当该顶点在边界上的左右相邻点位于射线  $l$  的异侧。当射线和多边形的某条边部分重合时不算穿越边界，但边界上的点算在  $P$  的内部还是外部取决于具体应用。

## 14.5 网格上的算法

矩形网格或六边形网格上绘制的多边形都可以很自然地分解成单元格，这使得一些计算问题可以利用单元格来解决：

- 面积 — 矩形面积公式为长  $\times$  宽，三角形面积公式为  $1/2 \times$  底  $\times$  高。边长为  $r$  的等边三角形面积为  $\sqrt{3}r^2/4$ ，因此半径为  $r$  的正六边形面积为  $3\sqrt{3}r^2/2$ 。
- 周长 — 矩形的周长公式为  $2 \times (\text{长} + \text{宽})$ ，三角形周长公式为  $a + b + c$ ，对于边长为  $r$  的等边三角形来说可以化简为  $3r$ 。边长为  $r$  的正六边形周长为  $6r$ 。注意，这个值已经比较接近于圆的周长  $2\pi r \approx 6.28r$  了。
- 凸包 — 正方形、等边三角形和正六边形都是凸的，因此它们的凸包都是自身。
- 三角剖分 — 在正方形中随便挑一个对角线以及在正六边形中选择从任意顶点出发的所有三条对角线都能把相应图形三角剖分。这样做的依据是：这些图形都是凸的。如果有凹进去的部分，则三角剖分会复杂很多。
- 点定位 — 我们已经看到，点在边平行于坐标轴 (*axis-oriented*) 的矩形内当且仅当  $x_{max} > x > x_{min}$ ，并且  $y_{max} > y > y_{min}$ 。三角形和六边形的测试要稍微复杂一些，但可以先检查点是否在它们的包围盒中，以减少复杂测试的次数。

我们用网格上的两个有趣的几何计算问题来结束本节。尽管它们主要针对的是矩形网格，但如有需要，也能扩展到其他网格中。

### 14.5.1 范围查询

正交范围查询 (*orthogonal range queries*) 是  $n \times m$  矩形网格中的一种常见操作。我们需要一种数据结构快速、方便地回答如下形式的询问：“给定矩阵中的一个子矩形，矩形内元素的和是多少？”

任何边平行于坐标轴的矩形都可以由两个点来确定：左上角  $(x_l, y_l)$  和右下角  $(x_r, y_r)$ 。最简单的算法是用嵌套循环把满足  $x_l \leq i \leq x_r, y_r \leq j \leq y_l$  的所有  $m[i][j]$  加起来，但这样做效率太低，特别是当你需要反复回答这样的询问，以寻找元素和最大或最小的子矩形时。

另一种方法是构造一个新的矩阵，元素  $m1[x][y]$  表示所有  $m[i][j]$  之和，其中  $i \leq x, j \leq y$ 。这个所谓的优势矩阵 (*dominance matrix*)  $m1$  让任意矩阵内元素和的计算变得容易了：

$$S(x_l, y_l, x_r, y_r) = m1[x_r, y_l] - m1[x_l - 1, y_l] - m1[x_r, y_r - 1] + m1[x_l - 1, y_r - 1]$$

这样做当然就快多了，因为只需要读取四个元素，然后做三次加减法。但这个方法为什么是对的呢？因为  $m1[x_r, y_l]$  除了包含需要累加的所有元素外，还多加了一些。上式中接下来的两项把多余元素减去了，但左上角的小矩形减去了两次，必须重新加回来。这正好是我们在组合数学中讨论过的容斥原理。矩阵  $m1$  可以在  $O(mn)$  时间内构造完。具体来说，只需按照同样的思路，用行优先顺序填充这个矩阵。

### 14.5.2 网格多边形与 Pick 定理

单位间隔 (unit-spaced) 的点 (也称格点 (lattice points)) 组成的正规网格是网格坐标系的核心。一般来说，网格中的单位面积区域中约包含一个格点，因为每个格点都是一个不同的  $1 \times 1$  单位矩形的右上角。因此，给定图形内部的格点数是该图形面积的良好近似。

*Pick 定理* 叙述了格点多边形 (顶点全是格点的不自交图形)  $P$  的面积与其边界/内部格点数之间的关系。假设  $P$  的内部有  $I(P)$  个格点，边界上有  $B(P)$  个格点，则  $P$  的面积  $A(P)$  为：

$$A(P) = I(P) + B(P)/2 - 1$$

如图 14.4 所示。

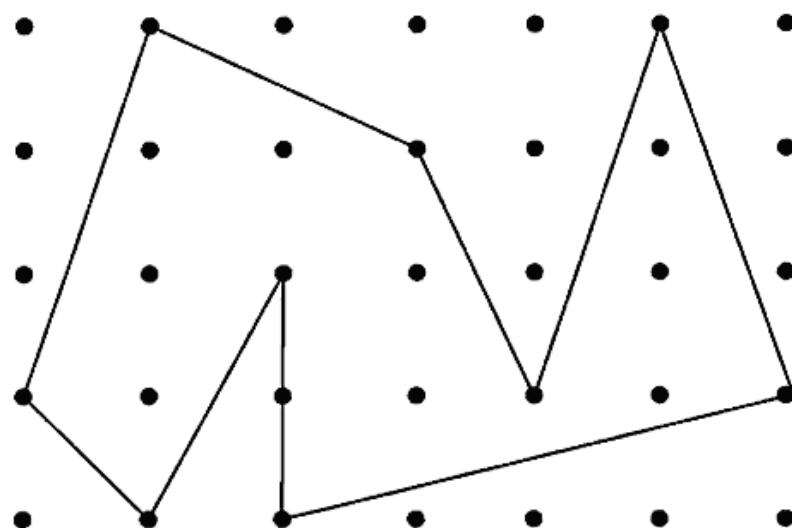


图 14.4 边界上有 10 个格点，内部有 9 个格点的格点多边形。根据 Pick 定理，面积为 13

例如，考虑一个三角形，它的三个顶点坐标分别为  $(x, 1), (y, 2)$  和  $(y + k, 2)$ 。不管  $x, y$

和  $k$  的值是多少，三角形内部一定没有格点，因为三个顶点位于网格上的连续两行。格点  $(x, 1)$  是三角形的顶点，而底边上有  $k + 1$  个格点。因此， $I(P) = 0$ ,  $B(P) = k + 2$ ，从而面积为  $k/2$ ，和直接套用三角形面积公式算出的结果完全一致。

再例如，考虑对角线上的两个顶点为  $(x_1, y_1)$  和  $(x_2, y_2)$  的矩形。边界上的格点数为

$$B(P) = 2|y_2 - y_1 + 1| + 2|x_2 - x_1 + 1| - 4 = 2(\Delta_y + \Delta_x)$$

其中最后减去 4 是为了避免四个角被计算两次。内部格点数等于边界和内部的总格点数减去边界格点数，即

$$I(P) = (\Delta_x + 1)(\Delta_y + 1) - 2(\Delta_y + \Delta_x)$$

Pick 定理正确地计算出矩形面积  $\Delta_x \Delta_y$ 。

使用 Pick 定理需要准确地统计格点数。如果多边形的面积较小，理论上可以用两类测试依次判断所有格点：(1) 点是否在线段上 (2) 点在多边形内部还是外部。更加巧妙的扫描线算法让我们不必测试所有点，而只是测试边界上的格点，从而大大提高效率。有兴趣的读者可以参考 [GS93]，以获取关于 Pick 定理的有趣讨论和相关课题。

## 14.6 几何函数库

Java 的 `java.awt.geom` 包定义了一些二维几何对象和它们的基本操作。`Polygon` 类提供了我们刚才讨论过的很多算法，包括解决点定位问题的 `contains`。更一般的 `Area` 类允许我们将多边形和其他图形、曲线做交和并。`Line2D` 类提供了我们讨论过的很多线段操作，包括相交判定和 `ccw` 谓词。

## 14.7 习 题

### 14.7.1 新生集会 (Herding Frosh)

PC/UVa 题号: 111401/10135, 流行度: C, 通过率: average 难度: 2

某一天，学校草坪上聚集了大批的大一新生。为了美化校园，一位学长决定用一条丝带把他们圈起来。你的任务是计算这位学长需要多长的丝带。

这位学长将丝带的一头绑在一个公用电话柱子上，然后沿着站有新生区域的边界走，拉住丝带来围住所有人。最后他回到公用电话柱旁。这位学长只使用最短需要的丝带长度来围住所有人，包括两端分别多出一米用来绑在柱子上。

你可以假设公用电话柱的坐标为  $(0, 0)$ ，其中第一个坐标为南北方向，第二个坐标为东西方向。所有新生的坐标按照与电话柱的相对位置给出。新生不超过 1000 名。

最新書  
坊  
www.ninsi.com

输入

输入数据开头单独一行一个正整数表示输入数据的组数，接下来是一个空行。每组数据的第一行为新生的人数，接下来每行两个实数，描述一个新生的坐标。

相邻两组输入数据间用一个空行分隔。

输出

对于每组数据，输出一行，包含一个实数，即丝带的长度（四舍五入保留小数点后两位）。相邻两组数据的输出间用一个空行分隔。

样例输入	样例输出
1	10.83
4	
1.0 1.0	
-1.0 1.0	
-1.0 -1.0	
1.0 -1.0	

#### 14.7.2 最近点对问题 (The Closest Pair Problem)

PC/UVa 题号: 111402/10245, 流行度: A, 通过率: low 难度: 2

有一家很没有效率的电话公司想要宣称他们能为顾客提供高速宽带的线路。在营销上，他们只要能够做出一条这样的线路直接连接两个地方，就达到目的了。由于安装这样的线路的成本与所连接两地的距离是成比例的，他们想要找到一对最近的地点，以最小的代价来实践他们的营销策略。

更精确的说，给出平面上一个点集，你需要找出最近点对之间的距离。但如果连最近点对之间的距离也超过了某个限制，营销部会考虑其他更合算的策略。

输入

输入包含多组数据。每组数据以一个整数  $N$  开头 ( $0 \leq N \leq 10\,000$ )，表示平面点集中有多少个点。接下来的  $N$  行为  $N$  个点的坐标。每行的两个数分别为一个点的  $x$  坐标和  $y$  坐标。 $N = 0$  时输入结束。所有坐标为小于  $40\,000$  的非负数。

输出

对于每组数据，输出单独一行，包含一个实数，四舍五入保留小数点后四位，即表示最近点对的距离。如果所有点两两间距离均大于 10 000，输出“INFINITY”。

www.n1n5i.com

样例输入	样例输出
3	INFINITY
0 0	36.2215
10000 10000	
20000 20000	
5	
0 2	
6 67	
43 71	
39 107	
189 140	
0	

### 14.7.3 电锯惊魂 (Chainsaw Massacre)

PC/UVa 题号: 111403/10043, 流行度: B, 通过率: low 难度: 3

加拿大的伐木工人团正在举行他们一年一度的伐木竞赛，蒙特利尔和温哥华间的国家森林遭了殃。接下来是派对时间！为了给晚间聚会布置出一块足够大的舞场，组委会想找出一大块没有树的矩形区域。所有的伐木工们都喝醉了，没有人想要冒险去操作电锯再砍倒一棵树。

组委会请你找到最大的一片矩形空地作为舞场。你需要找到一块矩形区域，整个舞场将被布置在那里。矩形的边要与整个地区的边沿平行。舞场可以座落在整个地区的边缘，并且舞场的边界上可以有树。

## 输入

输入数据第一行一个数说明测试数据的组数。对于每组数据，第一行为两个整数，即这片区域的长  $l$  和宽  $w(0 < l, w \leq 10\,000)$ (单位：米)。接下来每行按如下格式描述单独一棵树或是描述一列树：

- $1 \ x \ y$ , 其中 “1” 表明后面描述的是一棵树,  $x$  与  $y$  为这棵树相对于左上角的坐标, 以米为单位。
  - $k \ x \ y \ dx \ dy$ , 其中  $k > 1$  给出这一列树的数目, 且这些树的坐标为:  $(x, y), (x + dx, y + dy), \dots, (x + (k - 1)dx, y + (k - 1)dy)$ 。
  - $0$  表示数据结束。

坐标  $x, y, dx, dy$  均为整数。所有树都在这片区域内，即坐标在  $[0, l] \times [0, w]$  内。最多有 1000 棵树。

输出

对于每组数据，输出一行，包含一个整数，即最大舞场的面积（单位：平方米）。

样例输入	样例输出
2	6
2 3	80
0	
10 10	
2 1 1 8 0	
2 1 9 8 0	
0	

#### 14.7.4 冷热游戏 (Hotter Colder)

PC/UVa 题号: 111404/10084, 流行度: C, 通过率: low 难度: 3

小朋友们在玩一种叫*Hotter Colder*的游戏。游戏开始时，游戏者A离开房间，游戏者B把一个物品藏在房间的某个位置。然后A重新进入房间，站在(0, 0)这个位置，然后走到房间中的不同位置。游戏者A每到一个不同的位置，游戏者B就会说一句话来提示A：如果A离物品更近了，B会说“Hotter”；如果A离物品更远了，B会说“Colder”；如果距离相同，B会说“Same”。

输入

输入数据最多包含 50 行，每行包含一个  $(x, y)$  坐标，后面紧跟一个“Hotter”，“Colder”或“Same”。

每一个坐标表示房间中的一个位置，假设房间的对角为  $(0, 0)$  和  $(10, 10)$ 。

输出

对于输入数据的每行，输出一行，表示物品可能存在的区域的面积，四舍五入保留小数点后两位。如果不存在这样的区域，输出“0.00”。

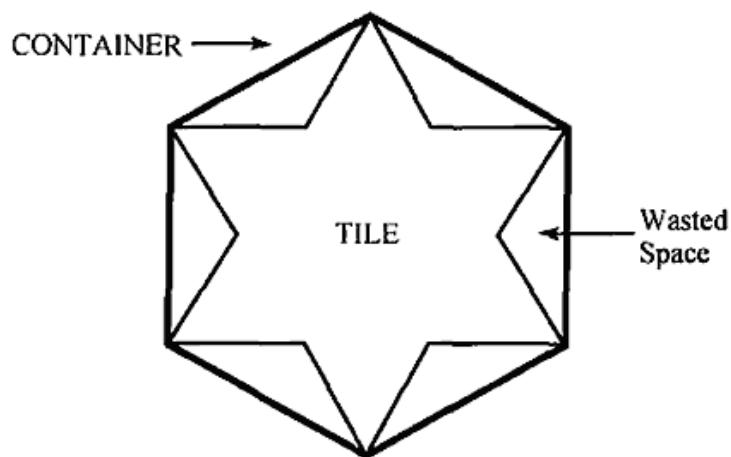
样例输入	样例输出
10.0 10.0 Colder	50.00
10.0 0.0 Hotter	37.50
0.0 0.0 Colder	12.50
10.0 10.0 Hotter	0.00

### 14.7.5 没用的瓷砖打包公司 (Useless Tile Packers)

PC/UVA 题号: 111405/10065, 流行度: C, 通过率: average 难度: 3

“没用的瓷砖打包公司”以自己的效率而自豪。公司的名字就是取的“比其他公司用更少的空间”的寓意。营销部门曾经试图建议管理层改掉这个名字，因为他们认为这个名字有其他的理解方式，但目前为止未被采纳<sup>6</sup>。

需要被打包的瓷砖厚度均匀，且轮廓为简单多边形。每一种瓷砖都由专门为其实制的容器打包。这种容器的底部总是能够容纳该种瓷砖的最小凸多边形。



这样的打包方法使得容器中部分空间被浪费了。你的任务就是计算对于某种特定的瓷砖被浪费的空间的百分比。

#### 输入

输入包含多组数据。每组数据描述一种瓷砖。数据第一行包含一个整数  $N$  ( $3 \leq N \leq 100$ )，表示瓷砖的顶点数目。接下来  $N$  行每行两个整数  $(x, y)$ ，表示顶点坐标（基于一个合适的原点及坐标系），且满足  $0 \leq x, y \leq 1000$ 。输入顺序与顶点在瓷砖边缘的排列顺序相同。不会出现连续三点共线的情况。

当输入  $N$  为 0 时表示输入数据结束。

#### 输出

对于输入的每个瓷砖，输出浪费空间的百分数，四舍五入保留小数点后两位。每组数据单独输出一行。相邻两组数据的输出之间用一个空行分隔。

#### 样例输入

```
5
0 0
```

#### 样例输出

```
Tile #1
Wasted Space = 25.00 %
```

<sup>6</sup> 译者注：“没用的”英文是 useless，而“用更少”英文是 use less。本想夸耀自己的公司，不仅错误地把名字取成“没用的公司”，而且还固执地不肯采纳营销部门的意见——这是原文中设置的幽默。

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

Tile #2  
Wasted Space = 0.00 %

#### 14.7.6 雷达追踪 (Radar Tracking)

PC/UVa 题号: 111406/849, 流行度: C, 通过率: low 难度: 2

一个地对空雷达系统使用一个以两秒为周期顺时针旋转的天线。当天线朝向某物体，该物体与天线的距离即被测算出，并在一个圆形屏幕上显示一个白点。白点与屏幕中心的距离与天线和物体的距离成正比，而屏幕上白点与中心所在直线表示物体相对天线的方向。在屏幕中心正上方的白点代表天线正北方向的物体，在屏幕中心右方的白点代表天线正东方向的物体，以此类推。

空中有若干物体，各自以一个恒定的速度运动，因而每次天线探测到这些物体在屏幕上显示出的位置不同。你的任务是，给出前两次天线探测的结果，推算天线下一次探测到该物体在屏幕上的位置。如果有多种可能情况，你必须全部输出。

输入

输入数据包含若干行，每行四个实数  $a_1, d_1, a_2, d_2$ 。第一对  $a_1, d_1$  表示第一次探测到时的极角（单位：度）和距离（已规定好单位距离），第二对  $a_2, d_2$  表示第二次探测到该物体时的极角和距离。

注意天线是顺时针旋转的。若  $t = 0$  时天线面向北方，则在  $t = 0.5$  时将面向东方， $t = 1.0$  时将面向南方， $t = 1.5$  时将面向西方， $t = 2$  时将面向北方。如果一个物体在天线的正上方，天线无法探测到它。极角的表示与罗盘相同，正北为  $0^\circ$  或  $360^\circ$ ，正东为  $90^\circ$ ，正南为  $90^\circ$ ，正西为  $270^\circ$ 。

输出

每组数据的输出包含一行，为所有可能解。每个解用两个实数（四舍五入保留小数点后两位）表示下一次探测到的角度  $a_3$  和距离  $d_3$ 。角度按照升序排序，当角度相同时按距离降序排序。

**样例输入**

```
90.0 100.0 90.0 110.0
90.0 100.0 270.0 10.0
90.0 100.0 180.0 50.0
```

**样例输出**

```
90.00 120.00
270.00 230.00
199.93 64.96 223.39 130.49
```

**14.7.7 岛上的树 (Trees on My Island)**

PC/UVa 题号: 111407/10088, 流行度: C, 通过率: average 难度: 3

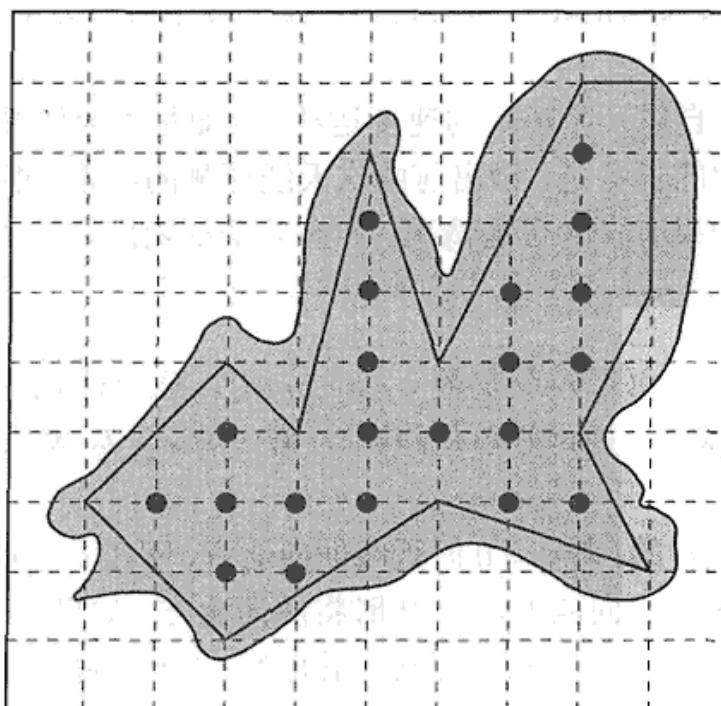
我买下了一个小岛，并想在上面整齐地种上树。所种下的树形成方形网格，因此只要选取恰当的点作为原点，每一棵树的位置均可以用整数坐标表示。

然而我的小岛并不是方形的。我在小岛内确定了一个顶点全部为整点的多边形，并决定在这个多边形内部（不包含边界）的整点种树。

请你帮我计算可以种多少棵树。

**输入**

输入可能包含多组数据。每组数据的第一行包含一个整数  $N(3 \leq N \leq 1000)$ ，表示多边形的顶点数。接下来  $N$  行以顺时针或逆时针的顺序给出多边形的所有顶点。每行为一个顶点的  $x$  坐标和  $y$  坐标。坐标的绝对值不超过 1 000 000。



我所拥有的小岛

输入数据以  $N = 0$  结束。

**输出**

对于每组数据，输出一行，仅包含一个数，即多边形内可以种树的棵数。

## 样例输出

21  
25990001

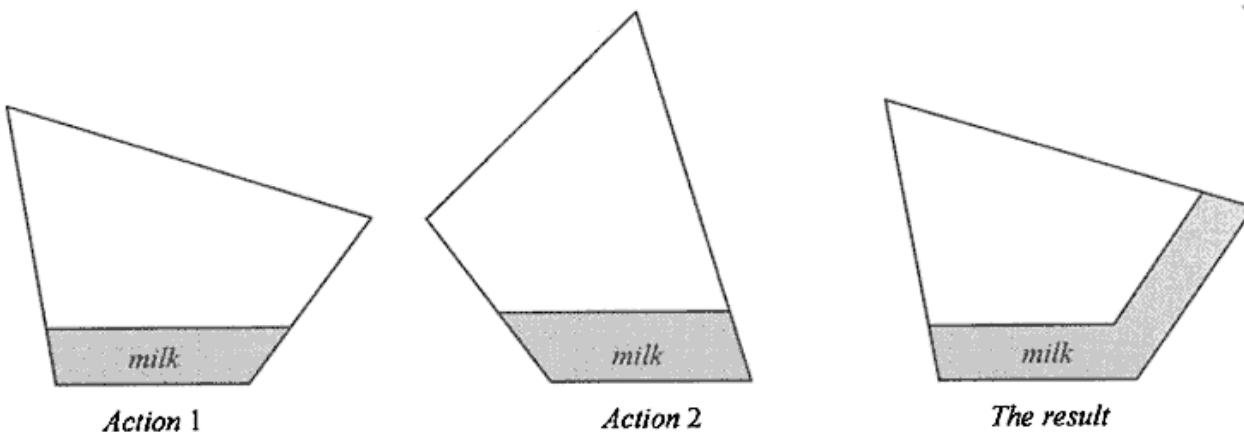
## 样例输入

12  
3 1  
6 3  
9 2  
8 4  
9 6  
9 9  
8 9  
6 5  
5 8  
4 4  
3 5  
1 3  
12  
1000 1000  
2000 1000  
4000 2000  
6000 1000  
8000 3000  
8000 8000  
7000 8000  
5000 4000  
4000 5000  
3000 4000  
3000 5000  
1000 3000  
0

## 14.7.8 美味的牛奶 (Nice Milk)

PC/UVa 题号: 111408/10117, 流行度: C, 通过率: low 难度: 4

Tomy 吃面包的时候喜欢蘸牛奶。在蘸的时候，他总是把面包浸入杯中直到触到杯子的底部，如下图：



由于杯中的牛奶有限，只有从牛奶液面到面包底部的这部分可以蘸到牛奶。这里认为牛奶的深度总是  $h$ ，而不会因蘸的次数改变。Tomy 想要把面包尽可能大的面积蘸上牛奶，但是不想蘸  $k$  次以上。你可以帮他吗？假定杯口足够大，面包任意一面都可以直接伸入杯底。

输入

每组数据以一行三个整数  $n, k, h$  开头 ( $3 \leq n \leq 20, 0 \leq k \leq 8, 0 \leq h \leq 10$ )。面包的形状保证为凸  $n$  边形。接下来  $n$  行每行两个整数  $x_i$  和  $y_i$  ( $0 \leq x_i, y_i \leq 1\,000$ )，表示第  $i$  个顶点的笛卡尔坐标。顶点按逆时针顺序编号。 $n = k = h = 0$  表示输入结束。

输出

对于每组数据，输出一行，包含面包蘸  $k$  次时，可以蘸上牛奶的最大可能面积，四舍五入保留小数点后两位。

### 样例输入

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

## 样例输出

7.46

## 14.1 如何处理公用电话柱?

14.2 两两比较每对点太慢。是否可以利用“我们只关心离得比较近的点”来减小比较次数？

14.3 数据应该被显式的填到一个  $l \times w$  的矩阵中，还是保留输入中的压缩形式？

14.4 如何表示所有可能位置组成的区域？它一定是一个凸多边形吗？

14.5 是分别算出两个面积再相减比较容易，还是分别计算那些“被浪费的多边形”的面积然后加起来比较容易？

### 14.6 什么情况下会有多解?

14.7 Pick 定理是否适用？或者有更好的方法？

14.8 是否存在一种保证最优解的贪心法？或者，必须穷举所有蘸牛奶的方案？

## 附录 A

要想在程序设计竞赛或者其他比赛中脱颖而出，光靠天份是不够的。你需要了解比赛、科学训练，并为比赛制定合理的战略战术。

在本章中，我们将介绍三个最重要的程序设计比赛，即大学生的ACM 国际大学生程序设计竞赛 (ACM ICPC)、中学生的国际信息学奥林匹克 (IOI) 和没有参赛限制的 TopCoder 挑战赛，包括它们各自的历史、比赛形式和参赛条件。另外，我们还采访了顶尖选手和教练，把他们训练和比赛策略中的秘诀与您分享。

### A.1 ACM 国际大学生程序设计竞赛

ACM 国际大学生程序设计竞赛 (ACM ICPC) 是学习计算机科学的学生向全世界展示才华的盛会。ICPC 自 1976 年首次举办以来，其规模、影响力和声望逐年增加。2002 年的比赛吸引了 3 082 个三人团队参加，这些队伍代表了来自 67 个国家的超过 1300 所学校。这些数字还不包括那些只参加过相关的内部比赛或网上竞赛的选手，而这部分学生不计其数。

比赛形式如下：每支队伍由三个学生组成，比赛任务是解决五到十个编程题目。一支队伍中的三个学生共用一台电脑，因此团队合作与协调能力是至关重要的。

获胜者是在规定时间（一般是 5 小时）内正确地解决<sup>1</sup> 了最多题目的队伍。题目没有部分分，因此只有完全正确的程序才算成绩。当解题数相同时，正确的解题所花总时间较小者排名在前。因此，最快的程序员（而不是最快的程序）获胜。程序风格和效率对成绩毫无影响，只要程序能在裁判允许的时间范围内运行完毕。每次错误的提交均会给总成绩带来 20 分钟的罚时，这就要求选手在提交前仔细检查。

我们采访了 2002 年 ACM ICPC 顶尖队伍的选手与教练，向他们请教训练和比赛的秘诀。下面便是我们所学到的 …

#### A.1.1 准备

##### A.1.1.1 队伍选择与培训

从在校学生里选拔参赛队伍是教练的职责。Cornell 大学和清华大学等学校聘用毕业生担任教练，而 Duke 和 Waterloo 等大学的教练由学校里有声望的高水平教师出任。不管采取哪种方法，不变的道理是：优秀的队伍需要层层选拔和有效的领导。上述两种教练都可以做得很成功，就像刚才提到的四所大学一样。

<sup>1</sup> 至少裁判认为是对的。二者存在差异，且在这一点上，至今仍有争议。

很多顶尖学校通过内部比赛为区域赛选拔队伍。与团队赛相比, Waterloo 大学的教练 Gordon Cormack 更倾向于组织校内个人赛, 以照顾独立选手, 降低参赛门槛, 在组队之前先选拔优秀的个人选手。借助于 Valladolid 在线评测系统, 这样的比赛很容易组织和管理。

只有刻苦训练才能打造出最优秀的队伍。例如, Waterloo 大学的队伍每周都会以团队方式训练两到三次, 训练环境和总决赛的比赛环境类似。

### A.1.1.2 资源

ACM ICPC 规则允许参赛队伍在比赛时携带纸质材料, 但不能访问在线资源。书里不能夹带配套光盘, 而网络连接要么不可用, 要么处于监控之下, 以避免参赛队伍访问因特网。

除了本书之外, 还有哪些优秀书籍值得学习? 作为一般性的算法参考, 我们推荐 Skiena 所著的算法设计手册 (*The Algorithm Design Manual*) [Ski97]。除了我们自己“王婆卖瓜”之外, 很多选手和教练同样推荐这本书。包含真实编程语言实现的书籍往往比较畅销, 包括 Sedgewick [Sed01] 的图算法书籍和 O'Rourke [O'R00] 的计算几何书籍。但是, 如果并未真正理解别人的代码就直接敲到电脑里用, 一旦出了错, 调试过程将变得异常艰难。Cormack 提醒到: “代码本身并没有你想象得那么有用, 除非是你亲自写的, 或者至少敲到电脑里去试过一段时间。”带一本语言参考和库函数手册也是有益的<sup>2</sup>。

做好了充分准备的队伍都会打印一些平时解决过的问题的代码。如果比赛中遇到了类似的题目, 这些代码将会很有用。Oldenburg 大学(德国)的 Christian Ohler 这样强调几何代码的重要性: “如果没事先准备并且充分测试好一个几何代码库, 碰到几何题你就完了。”

这样的模板对于 Java 来说尤其重要: 编写鲁棒的 I/O 以及解析常用数据类型等任务都有一些复杂, 但又是必不可少的。一个常见的技巧是在比赛开始的时候由一位队员把这些代码敲进电脑, 剩下的两位队员阅读题目。

### A.1.1.3 训练

最好的训练资源是 Valladolid 在线评测系统。去年总决赛的参赛队伍中, 至少有 80% 的选手用该系统训练。该系统不定期举办在线比赛, 在区域赛和总决赛来临之前, 在线比赛的频率还会有所提高。读者可登录网站 <http://uva.onlinejudge.org/> 查询比赛安排和其他信息。

Ural 州立大学在线评测系统 (<http://acm.timus.ru/>) 和因特网问题求解比赛 (IPSC) (<http://ipsc.ksp.sk/>) 都包含了不少好题。美国信息学奥林匹克网站 <http://www.usaco.org> 也有很多有趣的题目和相关资料。很多学生喜欢思考往年竞赛题, 但不一定打算花时间编程实现。这些题目可以在 ACM ICPC 官方网站 <http://www.acmicpc.org> 上找到。清华大学的 Rujia Liu(刘汝佳) 注意到不同国家的比赛题目在类型和风格上各不相同。他发现亚洲题目“又难又偏”, 因此很适合深入思考和分析; 北美题目相对更适合作为赛前的编程练习, 但很少用得上较深的算法。

<sup>2</sup> 译者注: 多数比赛都提供编程语言自带的电子版语言和库函数参考。

### A.1.2 战略战术

#### A.1.2.1 团队合作

在 ICPC 中，每支队伍由三个队员组成。由于三人共用一台电脑，团队合作显得格外重要，不停争夺键盘的队伍是不会有出息的。上海交通大学（2002 年世界冠军）的 Zhou Jian(周健) 如是说：“比赛中所做的一切都应该是为了让队伍的成绩更好，而不是让你自己发挥得最出色。”

最成功的队伍总是根据选手自身情况给不同队员分配不同角色。在一支典型的队伍中，一个学生担任代码员，因其卓越的语言功底和打字速度在比赛中长时间操作电脑；另一个学生担任算法员，以施展其优秀的题目分析和算法设计能力。第三个学生担任调试员，独自研究打印出的程序清单和调试输出，以找出和修复 bug，同时代码员可以用电脑编写其他程序。

当然，随着比赛进程，上述角色可能会发生转变，而且不同队伍之间的差异会很大。一些队伍会由一个常任队长来给队员分配题目和机器。

一些队伍使用特殊的读题技巧。最有效的方法似乎是把题目均分给每个人同时阅读，因为最简单的题目也许藏在试题的最后一页。当有人发现简单题后，要么自己做，要么交给一个更合适的队友。在一些非英语母语国家的队伍中，英文最棒的队员负责浏览题目并分配给队员。

#### A.1.2.2 比赛技巧

- 了解你的弱点 — 因为只有完全正确的程序才算成绩，应该先做简单题。很多看上去简单的题目有一些恶心的陷阱或者模糊的题目描述，引诱你反复提交，但每次都得到恼人的“答案错”。孟加拉工程科技大学 (BUET) 的 Shahriar Manzoor 的建议如下：如果连最简单的题的程序都被退回来了，而且原因未知，那么最好是找另外一个人重做一遍，因为你很可能已经落入了思维陷阱。
- 关注比赛进程 — 有空时多关注比赛的即时排名，找出通过队伍数最多的题目。如果其中有的题目你们还没有做过，赶紧去做吧！它们往往相对简单。
- 避免错误的提交 — 清华大学的 Rujia Liu(刘汝佳) 认为正确性比速度要重要得多。“正因为‘答案错’导致的大量罚时，我们和世界前三失之交臂。”要想减少这种罚时，需要在读题后和队友讨论，以确保正确理解了题目，然后提交之前充分测试你的程序。
- 停机问题 — 超时错 并不一定表示你的算法太低效。这可能是因为读取输入时陷入了无限循环 [Man01]。例如，你正在等待从标准输入读取数据，但是裁判却要求你从文件中读取数据。又例如，你假定了一种错误的输入格式，比如你以为输入会以 0 符号结尾，但实际上真实数据以文件结束符结尾。
- 了解编译器 — 某些编程环境提供了一些选项能简化特定的任务。例如，限制程序可使用的内存大小，当比赛规则中有内存限制时很有用。“消除惊讶的最好方式乃是熟悉环

境。”Gordon Cormack 如是说。

- 充分利用机器 — Cormack 要求他的队伍“尽量多敲键盘，即使是编写一个只能读取输入的程序框架。”
- 主流调试技巧 — 在反馈信息很少甚至完全没有的情况下，如何调试程序？裁判仅仅告诉你程序错了，而不会让你看到那个出错的数据。尽管裁判也有犯错的可能，但在多数情况下，程序没有通过就表明有 bug。清晰的程序清单和冷静的头脑是你最好的伙伴。

仔细检查你用来测试程序的数据。今年总决赛有一支队伍花了两个小时来调试一个正确的程序，原因是他们把样例输入敲错了。不过，即使你的程序是错的，也不要轻易放弃。Oldenburg 大学的 Jens Zumbrugel 这样警告：“离比赛结束只有 90 分钟时，如果手头有题目正在调试，就千万不要做新题了。”

- 非主流调试技巧 — 如果实在找不到错，有一个不那么正大光明的技巧可能会用得上：在你觉得程序可能会出错的地方加一个超时循环或者故意除零。这样，你可以用 20 分钟罚时来换取一点新的信息。这招不要用得太频繁，否则当程序最终通过时，巨大的罚时会让队友想灭了你。
- 利用异常处理机制 — Stanford 大学的 Daniel Wright 推荐下面的技巧：如果你的编程语言支持异常处理，当程序发生异常时还能返回一个猜测的结果，而不是直接崩溃。例如，当整个程序异常时输出无解，或者输入非法。
- 困惑时保持冷静 — 尽量别太有压力，也不要和你的队友发生冲突。“玩得开心，不要本末倒置，”Gordon Cormack 这样要求自己的学生。“只要把简单题找出来并且努力解决，你们就是好样的！”

## A.2 国际信息学奥林匹克

国际信息学奥林匹克 (IOI) 一年一度，是专为中学生举办的计算机科学比赛。自 1989 年创立以来，它已经迅速成长为了五大国际科学奥林匹克<sup>3</sup> 中的第二大比赛，仅次于历史悠久的数学。2002 年，来自 78 个国家的 276 位精英选手云集韩国参加最后的决赛，而这些参赛者本身又是从千千万万名各国内外选手中层层选拔出的。

IOI 的目标和 ACM ICPC 不太一样。由于参赛者往往还没有开始选择他们的职业道路，IOI 致力于激发选手们对信息学（计算机科学）的兴趣。IOI 把各国的天才少年集中到一起，让他们可以方便地分享和交流科学与文化。

### A.2.1 如何参加

IOI 的主办国家轮流坐庄：2003 年在美国的 Wisconsin 州，2004 年在希腊，2005 年在波兰。每个参赛国家将派一个由四个学生和两个陪同教练所组成的代表团。学生以个体形式参

<sup>3</sup> 译者注：联合国教科文组织发起的五大学科竞赛分别是数学 (IMO)、物理 (IPhO)、化学 (IChO)、生物 (IBO)、信息学 (IOI)。

赛，需要在两天比赛<sup>4</sup>中解决两套编程难题，目标是让两试总分尽可能大。一般来说，每天比赛都是 5 小时解决 3 道题目。

各国都用自己的方式选拔国家代表队。一些国家（例如中国和伊朗）会先从成千上万的学生中“海选”出最有潜力的学生，然后通过正规的比赛遴选出 20 个左右最出色的学生<sup>5</sup>。这些学生将会接受国家队教练精心准备的高强度严格训练，然后从中选拔出四名选手组成最后的国家队。

美国计算机奥林匹克 (USACO) 维护着一个很棒的训练项目：<http://train.usaco.org>，并举行一系列任何人都能参加的网上编程比赛。要想进入美国国家队，你必须参加全美公开赛。为此，你需要在比赛前从你所在的高中找一个老师监考。在公开赛中取得佳绩的学生会被邀请参加美国训练营，以接受正规训练，并参加最后的选拔。加拿大通过资格赛从约 1000 名选手中选拔出 22 名选手，然后训练一周。

### A.2.2 比赛形式

和 ACM ICPC 不同，IOI 允许一道题目得到部分分数。每道题目通常有十个测试用例，每个 10 分<sup>6</sup>。一般会有两天比赛，每天三题，总分 600 分。

所有题目（用 IOI 的术语来说是任务 (*tasks*)）着重考察算法。如果算法的效率很重要，测试数据往往能保证效率再低的程序也能获得一定的分数。

但是 IOI 科学委员会成员 Ian Munro 解释道：“很难设计一道题目，既能让大多数选手得分，也能在高分段保持良好的区分度。”

有一种 IOI 特有的题型交互式题目 (*reactive tasks*)，它的输入并不是实现确定的 [HV02]。你的程序通过函数调用而非数据文件<sup>7</sup>来获取这种“现场输入”。例如，你可能需要探索一个迷宫，通过函数调用来移动你自己，返回值表明你是否碰到了墙壁。或者你需要写一个博弈程序，和一个对手程序交互。

在 IOI2002，选手可以选择用 Linux 或者 Windows 操作系统，可选的编程语言包括 Pascal 和 C/C++。选手不得把纸质材料带进赛场，比赛时也不能访问在线资源。

IOI 在比赛结束后统一评分，而不是像 ACM ICPC 那样即时评测和排名。像通常的课程考试一样，在名次公布之前，学生们都不知道自己的成绩。

IOI 是三大编程竞赛中最不商业化的一个。对于这三项赛事的决赛选手 Daniel Wright 来说，IOI 偏重的是学术。这一点在食宿上也有所体现：IOI 选手通常住在大学宿舍里，而 ICPC/TopCoder 决赛选手则能享受到奢华的星级酒店。

<sup>4</sup> 译者注：为了避免脑力消耗过度，两天比赛中间有一天休息。

<sup>5</sup> 译者注：例如在中国，NOI 前 20 名学生组成国家集训队。

<sup>6</sup> 译者注：这是 2002 年以前的情况。现在通常有 10 到 25 个测试数据不等，各数据的分数也不一定相同，但每题一般都是 100 分。

<sup>7</sup> 译者注：并不总是如此。也有的系统要求选手程序从标准输入输出来交互，如 IOI2003。

### A.2.3 准备

美国的 IOI 教练 Rob Kolstad 鼓励所有有兴趣的学生积极进行在线训练，并在美国公开赛前参加一定数量的其他比赛。他还教给学生如何在比赛中合理安排时间。同时还担任加拿大 IOI 教练的 Gordon Cormack 鼓励他的学生改掉“调得看上去差不多了就行”这样的坏习惯，努力在有限的时间内写出能解决所有用例的完美程序。他的最终目标是让学生彻底脱离对调试器的依赖，而把注意力聚焦在算法上。

人们普遍认为 IOI 题目和 ACM ICPC 题目不太一样。根据 Kolstad 的说法，IOI 题目是“纯算法性的”，而且叙述更加清晰，避免在输入中设置陷阱。而 ACM ICPC 题目常常在输入检查和输出格式化上做文章。

IOI 题目“和 ACM 题目难度相当”，Cormack 解释道。它们的参考程序有时会比 ACM 题目短：毕竟这是个人赛，而不是组队赛。对于这些题目，“简单粗暴”的程序只能通过小部分数据。要想得到满分，聪明才智是必不可少的。历届赛题可以在官方 IOI 网站 <http://olympiads.win.tue.nl/ioi/> 中找到<sup>8</sup>。该网站同时还有两本官方推荐书目的链接，Kernighan 和 Pike 的程序设计实践 [KP99]<sup>9</sup> 以及 Skiena 的算法设计手册 [Ski97]。

深厚的数学功底似乎是一流选手的必备素质。IOI2001 的第一名，美国的 Reid Barton 同时也获得了国际数学奥林匹克的金牌。

参加 IOI 系列活动将给 ACM ICPC 打下良好的基础。第一年的 IOI 选手与第二年的 ACM ICPC 总决赛选手往往存在很大的交集。例如，2002 年 ACM ICPC 的清华队伍<sup>10</sup>(获得全球第四名)全部来自于中国的 IOI2001 国家集训队。类似的，几乎半数的 TopCoder “全明星”是往年 IOI 或 ACM ICPC 中的佼佼者。

### A.3 Topcoder.com

参加编程竞赛有很多不错的理由 —— 不仅乐趣多多，提高了你的编程水平，还让你的就业前景更加美好。本书中的编程挑战题目对于解决很多大公司的面试题有着很好的启发性。

一个称为 TopCoder 的公司悄然兴起。该公司利用编程竞赛来发现潜在的 IT 人才，并把它们的信息提供给客户。参加 TopCoder 比赛的一大收获就是金钱。2002 年 TopCoder 大学生挑战赛 (TCCC) 由 Sun 公司赞助，总奖金高达 \$150 000。Stanford 大学的 Daniel Wright 一举夺冠，获得 \$100 000 的巨额奖金，并慷慨地和我们分享他成功的秘密。

<sup>8</sup> 译者注：官方网站已移至 <http://ioinformatics.org/>。

<sup>9</sup> 译者注：该书的影印版、中文版和双语版均由机械工业出版社出版，出版年份分别为 2002、2004 和 2007。中文版和双语版均由凌宗燕教授翻译。

<sup>10</sup> 译者注：李益明、毛子青、刘汝佳。

TopCoder 有一个漂亮的网站 ([www.topcoder.com](http://www.topcoder.com))，上面有关于近期比赛的新闻文章，看上去和体育网站没什么两样。你可以在网站上自由练习，帮助你更好地参加约每周一次的常规赛（每次比赛有三道题目）。常规赛自 2001 年开始以来，已有超过 20 000 名来自世界各地的编程爱好者注册参加。到目前为止，TopCoder 公司和赛事赞助商所支付的总金额已达到一百万美元。

随着公司对商业模式的探索，TopCoder 比赛的形式不断更新。前几轮比赛都在网上完成，而重大赛事的决赛都在现场举行。

各轮比赛的基本形式相同。选手被分成若干个“房间”，每个房间内的选手可以相互挑战（见后）。每场比赛以 75 分钟的编码阶段 (*coding phase*) 开始，在这期间选手需要编程解决三道难度不同的题目。每道题目的得分是从“开题”到“提交”所经过的时间的减函数。接下来是 5 分钟的休息时间，然后是 15 分钟挑战阶段 (*challenge phase*)，在这期间每位选手都可以阅读同房间内的其他选手的程序，尝试找出这些程序的 bug。如果能构造出一组测试数据让另一位选手的程序出错，可以得到额外的 50 分奖励<sup>11</sup>。不正确的程序<sup>12</sup>，哪怕只有一组测试数据是错的，也无法得到任何分数。

多数选手习惯于按照从易到难的顺序做题，但 Wright 喜欢先做高分题目，如果他觉得时间不够做出所有题目。TopCoder 允许重新提交，但会影响得分，因此提交和测试的时机是有技巧的。在 TopCoder 比赛中，时间压力是一个关键因素。为了提高速度，Wright 鼓励选手学会有效地使用库函数。

编码阶段往往比挑战阶段更加重要，因为挑战成功所得的分数往往无法弥补做错一题所带来的损失<sup>13</sup>。在挑战阶段中，Wright 首先浏览程序，看看算法是否有问题。如果碰巧遇到了他曾经考虑过但已确认不正确的算法，他会毫不留情地挑战这个程序。更多的时候，他会寻找拼写粗心或者“只差一点点”的 bug。

## A.4 念研究生吧！

如果你觉得本书介绍的编程挑战很有趣，你应当考虑念一个研究生。计算机科学的研究生学习以本科为基础，但往往能涉及比较高级的课题。更重要的是，你可以选择自己喜欢的领域来进行全新的、原创性的研究工作。在美国，所有合理的博士项目都将为博士生提供学费、生活费和补贴，足以让他们过得很快乐（如果算不上奢侈的话）。

进入 ACM ICPC 总决赛，参加 IOI，甚至包括区域赛的前几名，都算得上是一个巨大的成就。显然，这清楚地表明你有足够的能力深入学习。在此，我强烈推荐你继续深造，最好能来 Stony Brook 大学和我（Steven Skiena）一起工作！我们组广泛涉猎算法和离散数学界

<sup>11</sup> 译者注：但另一方面，如果你提供的数据没能让你挑战的程序出错，你将被扣掉 25 分。

<sup>12</sup> 译者注：包括被挑战成功的程序以及没有通过挑战阶段之后的系统测试的程序。

<sup>13</sup> 译者注：但在包括重大赛事在内的很多比赛，在挑战阶段的得分足以保证晋级或者取得不错的名次。

的众多有趣的课题，更多的信息详见：<http://www.algorist.com/gradstudy>。

因此，我希望你能加入我们。另外，官方的 ACM ICPC 规则允许每支队伍有一名一年级研究生<sup>14</sup>，说不定你还能帮我们打入明年的决赛！

## A.5 题目致谢

作者列表中的第一个名字是命题者，他的委托人，或者整理题目的人，他们授权本书中使用这些题目。后面的名字（如果有的话）是题目的其他作者或相关人士。我们感谢以下所有人对本项目的贡献。

PC 题目 ID	UVa 题目 ID	题目名称	赞助/作者
1.1	110101	100 The 3n + 1 problem	Owen Astrakan
1.2	110102	10189 Minesweeper	Pedro Demasi
1.3	110103	10137 The Trip	Gordon Cormack
1.4	110104	706 LCD Display	Miguel Revilla, Immanuel Herrman
1.5	110105	10267 Graphical Editor	Alexander Denisjuk
1.6	110106	10033 Interpreter	Gordon Cormack
1.7	110107	10196 Check the Check	Pedro Demasi
1.8	110108	10142 Australian Voting	Gordon Cormack
2.1	110201	10038 Jolly Jumpers	Gordon Cormack, Wim Nuij
2.2	110202	10315 Poker Hands	Gordon Cormack
2.3	110203	10050 Hartals	Shahriar Manzoor, Rezaul Alam Chowdhury
2.4	110204	843 Crypt Kicker	Gordon Cormack
2.5	110205	10205 Stack 'em Up	Gordon Cormack
2.6	110206	10044 Erdős Numbers	Miguel Revilla, Felix Gaertner
2.7	110207	10258 Contest Scoreboard	Gordon Cormack, Michael Van Biesbrouck
2.8	110208	10149 Yahtzee	Gordon Cormack
3.1	110301	10082 WERTYU	Gordon Cormack
3.2	110302	10010 Where's Waldorf?	Gordon Cormack, University of Waterloo Computer Science Club (UWCSC)
3.3	110303	10252 Common Permutation	Shahriar Manzoor
3.4	110304	850 Crypt Kicker II	Gordon Cormack
3.5	110305	10188 Automated Judge Script	Pedro Demasi
3.6	110306	10132 File Fragmentation	Gordon Cormack, Charles Clarke
3.7	110307	10150 Doublets	Gordon Cormack
3.8	110308	848 Fmt	Gordon Cormack
4.1	110401	10041 Vito's Family	Miguel Revilla, Pablo Puente
4.2	110402	120 Stacks of Flapjacks	Owen Astrakan
4.3	110403	10037 Bridge	Gordon Cormack
4.4	110404	10191 Longest Nap	Pedro Demasi
4.5	110405	10026 Shoemaker's Problem	Alex Gevak, Antonio Sánchez
4.6	110406	10138 CDVII	Gordon Cormack
4.7	110407	10152 ShellSort	Gordon Cormack, Charles Clarke
4.8	110408	10194 Football (aka Soccer)	Pedro Demasi
5.1	110501	10035 Primary Arithmetic	Gordon Cormack
5.2	110502	10018 Reverse and Add	Enrique Moreno
5.3	110503	701 The Archeologist's Dilemma	Miguel Revilla

<sup>14</sup> 后来规则有所变动。从该研究生本科入学算起，最多只能有五年时间。

www.pin51.com  
站 拼 吾 爱  
新 闻 信 息 互 联 网

PC 题目 ID	UVa 题目 ID	题目名称	赞助/作者
5.4	110504	10127 Ones	Gordon Cormack, Piotr Rudnicki
5.5	110505	847 A Multiplication Game	Gordon Cormack, Piotr Rudnicki
5.6	110506	10105 Polynomial Coefficients	Alexander Denisjuk
5.7	110507	10077 Stern-Brocot Number System	Shahriar Manzoor, Rezaul Alam Chowdhury
5.8	110508	10202 Pairsumonious Numbers	Gordon Cormack, Piotr Rudnicki
6.1	110601	10183 How Many Fibs?	Rujia Liu, Walter Guttmann
6.2	110602	10213 How Many Pieces of Land?	Shahriar Manzoor
6.3	110603	10198 Counting	Pedro Demasi
6.4	110604	10157 Expressions	Petko Minkov
6.5	110605	10247 Complete Tree Labeling	Shahriar Manzoor
6.6	110606	10254 The Priest Mathematician	Shahriar Manzoor, Miguel Revilla
6.7	110607	10049 Self-describing Sequence	Shahriar Manzoor, Rezaul Alam Chowdhury
6.8	110608	846 Steps	Gordon Cormack, Piotr Rudnicki
7.1	110701	10110 Light, More Light	S. M. Mahbub Mrushed (Udvranto Pathik), Sadi Khan
7.2	110702	10006 Carmichael Numbers	Manuel Carro, César Sánchez
7.3	110703	10104 Euclid Problem	Alexander Denisjuk
7.4	110704	10139 Factoisors	Gordon Cormack
7.5	110705	10168 Summation of Four Primes	Shahriar Manzoor
7.6	110706	10042 Smith Numbers	Miguel Revilla, Felix Gaertner
7.7	110707	10090 Marbles	Shahriar Manzoor, Rezaul Alam Chowdhury
7.8	110708	10089 Repackaging	Shahriar Manzoor, Rezaul Alam Chowdhury
8.1	110801	861 Little Bishops	Shahriar Manzoor, Rezaul Alam Chowdhury
8.2	110802	10181 15-Puzzle Problem	Shahriar Manzoor, Rezaul Alam Chowdhury
8.3	110803	10128 Queue	Marcin Wojciechowski
8.4	110804	10160 Servicing Stations	Petko Minkov
8.5	110805	10032 Tug of War	Gordon Cormack
8.6	110806	10001 Garden of Eden	Manuel Carro, Manuel J. Petit de Gabriel
8.7	110807	704 Color Hash	Miguel Revilla, Pablo Puente
8.8	110808	10270 Bigger Square Please...	Rujia Liu
9.1	110901	10004 Bicoloring	Manuel Carro, Álvaro Martínez Echevarría
9.2	110902	10067 Playing With Wheels	Shahriar Manzoor, Rezaul Alam Chowdhury
9.3	110903	10099 The Tourist Guide	Shahriar Manzoor, Rezaul Alam Chowdhury
9.4	110904	705 Slash Maze	Miguel Revilla, Immanuel Herrman
9.5	110905	10029 Edit Step Ladders	Gordon Cormack
9.6	110906	10051 Tower of Cubes	Shahriar Manzoor, Rezaul Alam Chowdhury
9.7	110907	10187 From Dusk Till Dawn	Rujia Liu, Ralf Engels
9.8	110908	10276 Hanoi Tower Troubles Again!	Rujia Liu
10.1	111001	10034 Freckles	Gordon Cormack
10.2	111002	10054 The Necklace	Shahriar Manzoor, Rezaul Alam Chowdhury
10.3	111003	10278 Fire Station	Gordon Cormack
10.4	111004	10039 Railroads	Miguel Revilla, Philipp Hahn
10.5	111005	10158 War	Petko Minkov
10.6	111006	10199 Tourist Guide	Pedro Demasi
10.7	111007	10249 The Grand Dinner	Shahriar Manzoor, Rezaul Alam Chowdhury
10.8	111008	10092 Problem With Problem Setter	Shahriar Manzoor, Rezaul Alam Chowdhury
11.1	111101	10131 Is Bigger Smarter?	Gordon Cormack, Charles Rackoff
11.2	111102	10069 Distinct Subsequences	Shahriar Manzoor, Rezaul Alam Chowdhury
11.3	111103	10154 Weights and Measures	Gordon Cormack
11.4	111104	116 Unidirectional TSP	Owen Astrakan

PC 题目 ID	UVa 题目 ID	题目名称	赞助/作者
11.5	111105	10003 Cutting Sticks	Manuel Carro, Julio Mario
11.6	111106	10261 Ferry Loading	Gordon Cormack
11.7	111107	10271 Chopsticks	Rujia Liu
11.8	111108	10201 Adventures in Moving: Part IV	Gordon Cormack, Ondrej Lhotak
12.1	111201	10161 Ant on a Chessboard	Long Chong
12.2	111202	10047 The Monocycle	Shahriar Manzoor, Rezaul Alam Chowdhury
12.3	111203	10159 Star	Martins Opmanis
12.4	111204	10182 Bee Maja	Rujia Liu, Ralf Engels
12.5	111205	707 Robbery	Miguel Revilla, Immanuel Herrman
12.6	111206	10177 (2/3/4)-D Sqr/Rects/Cubes?	Shahriar Manzoor
12.7	111207	10233 Dermuba Triangle	Arun Kishore
12.8	111208	10075 Airlines	Shahriar Manzoor, Rezaul Alam Chowdhury
13.1	111301	10310 Dog and Gopher	Gordon Cormack
13.2	111302	10180 Rope Crisis in Ropeland!	Shahriar Manzoor, Rezaul Alam Chowdhury
13.3	111303	10195 Knights of the Round Table	Pedro Demasi
13.4	111304	10136 Chocolate Chip Cookies	Gordon Cormack
13.5	111305	10167 Birthday Cake	Long Chong
13.6	111306	10215 The Largest/Smallest Box ...	Shahriar Manzoor
13.7	111307	10209 Is This Integration?	Shahriar Manzoor
13.8	111308	10012 How Big Is It?	Gordon Cormack, University of Waterloo Computer Science Club (UWCSC)
14.1	111401	10135 Herding Frosh	Gordon Cormack
14.2	111402	10245 The Closest Pair Problem	Shahriar Manzoor
14.3	111403	10043 Chainsaw Massacre	Miguel Revilla, Christoph Mueller
14.4	111404	10084 Hotter Colder	Gordon Cormack
14.5	111405	10065 Useless Tile Packers	Shahriar Manzoor, Rezaul Alam Chowdhury
14.6	111406	849 Radar Tracking	Gordon Cormack
14.7	111407	10088 Trees on My Island	Shahriar Manzoor, Rezaul Alam Chowdhury
14.8	111408	10117 Nice Milk	Rujia Liu

## 参 考 文 献

- [AMO93] R. Ahuja, T. Magnanti and J. Orlin, *Network Flows*, Prentice Hall, Englewood Cliffs NJ, 1993.
- [Ber01] A. Bergeron. A Very elementary presentation of the Hannenhalli-Pevzner theory. In *Proc. 12th Symp. Combinatorial Pattern Matching (CPM)*, volume 2089, pages 106-117. Springer-Verlag Lecture Notes in Computer Science, 2001.
- [CC97] W. Cook and W. Cunningham. *Combinatorial Optimization*. Wiley, 1997.
- [COM94] COMAP. *For All Practical Purposes*, W. H. Freeman, New York, third edition, 1994.
- [dBvKOS00] M de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry: Algorithms and Applications*, Springer-Verlag, Berlin, second edition, 2000.
- [DGK83] P. Diaconis, R.L. Graham, and W.M. Kantor. The mathematics of perfect shuffles. *Advances in Applied Mathematics*, 4: 175, 1983.
- [Dij76] E. W. Dijkstra. A discipline of programming. 1976.
- [Gal01] J. Gallian, Graph labeling: A dynamic survey. *Electronic Journal of Combinatorics*, DS6, www.combinatorics.org, 2001.
- [GJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A guide to the theory of NP-completeness*, W. H. Freeman, San Francisco, 1979.
- [GKP89] R. Graham, D. Knuth, and O. Patashnik. *Concrete Mathematics*, Addison-Wesley, Reading MA, 1989.
- [GP79] B. Gates and C. Papadimitriou, Bounds for sorting by prefix reversals. *Discrete Mathematics*, 27: 47-57, 1979.
- [GS87] D. Gries and I. Stojmenović. A note on Graham's convex hull algorithm, *Information Processing Letters*, 25(5): 323-327, 10 July 1987.
- [GS93] B. Grunbaum and G. Shephard. Pick's theorem. *Amer. Math. Monthly*, 100: 150-161, 1993.
- [Gus97] D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.
- [Hof99] P. Hoffman. *The Man Who Loved Only Numbers: The Story of Paul Erdős and the Search for Mathematical Truth*. Little Brown, 1999
- [HV02] G. Horvath and T. Verhoeff. Finding the Median under IOI conditions. *Informatics in Education*, 1: 73-92, also at [http://www.vtex.lt/informatics\\_in\\_education/2002](http://www.vtex.lt/informatics_in_education/2002).
- [HW79] G. H. Hardy and E. M. Wright. *An Introduction to the Theory of Numbers*. Oxford University Press, fifth edition, 1979.

- [Kay00] R. Kaye. Minesweeper is NP-complete. *Mathematical Intelligencer*, 22(2): 9-15, 2000.
- [Knu73a] D. E. Knuth. *The Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley, Reading MA, second edition, 1973.
- [Knu73b] D. E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison-Wesley, Reading MA, 1973.
- [Knu81] D. E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, Reading MA, second edition, 1981.
- [KP99] B. Kernighan and R. Pike. *The Practice of Programming*. Addison Wesley, Reading MA, 1999.
- [Lag85] J. Lagarias. The  $3x+1$  problem and its generalizations. *American Mathematical Monthly*, 92: 3-23, 1985.
- [LR76] E. Luczak and A. Rosenfeld. Distance on a hexagonal grid. *IEEE Transactions on Computers*, 25(5): 532-533, 1976.
- [Man01] S. Manzoor. Common mistakes in online and real-time contests. ACM Crossroads Student Magazine, <http://www.acm.org/crossroads/xrds7-5/contests.html>, 2001.
- [McD87] W. McDaniel. The existance of infinitely many  $k$ -Smith numbers, *Fibonacci Quarterly*, 25: 76-80, 1987.
- [MDS01] D. Musser, G. Derge, and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison-Wesley, Boston MA, second edition, 2001.
- [Mor98] S. Morris. *Magic Tricks, Card Shuffling, and Dynamic Computer Memories: The Mathematics of the Perfect Shuffle*. Mathematical Association of America, Washington, D.C., 1998.
- [New96] M. Newborn. *Kasparov Versus Deep Blue: Computer Chess Comes of Age*. Springer-Verlag, 1996.
- [O'R00] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, New York, second edition, 2000.
- [PS03] S. Pemmaraju and S. Skiena. *Computational Discrete Mathematics: Combinatorics and Graph Theory with Mathematica*, Cambridge University Press, New York, 2003.
- [Sch94] B. Schneier. *Applied Cryptography*, Wiley, New York, 1994.
- [Sch97] J. Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag, 1997.
- [Sch00] B. Schechter. *My Brain Is Open: The Mathematical Journeys of Paul Erdős*, Touchstone Books, 2000.
- [Sed01] R. Sedgewick, *Algorithms in C++: Graph Algorithms*. Addison-Wesley, third edition, 2001.
- [Seu58] Dr. Seuss. *Yertle the Turtle*. Random House, 1958.
- [Seu63] Dr. Seuss. *Hop on Pop*. Random House, 1963.

- [Ski97] S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, New York, 1997.
- [Sti02] D. Stinson. *Cryptography: Theory and Practice*. Chapman and Hall, second edition, 2002.
- [Wes00] D. West. *Introduction to Graph Theory*. Prentice-Hall, Englewood Cliffs NJ, second edition, 2000.
- [Wil82] A. Wilansky. Smith numbers. *Two-Year College Math. J.*, 13: 21, 1982.
- [Wol02] S. Wolfram, *A New Kind of Science*. Wolfram Media, 2002.
- [ZMNN91] H. Zuckerman, H. Montgomery, I. Niven, and A. Niven. *An Introduction to the Theory of Numbers*. Wiley, New York, fifth edition, 1991.