

Programming Challenges with C++
Advanced Trainning Guide for Programming Contest (Second Edition)

C++，挑战编程

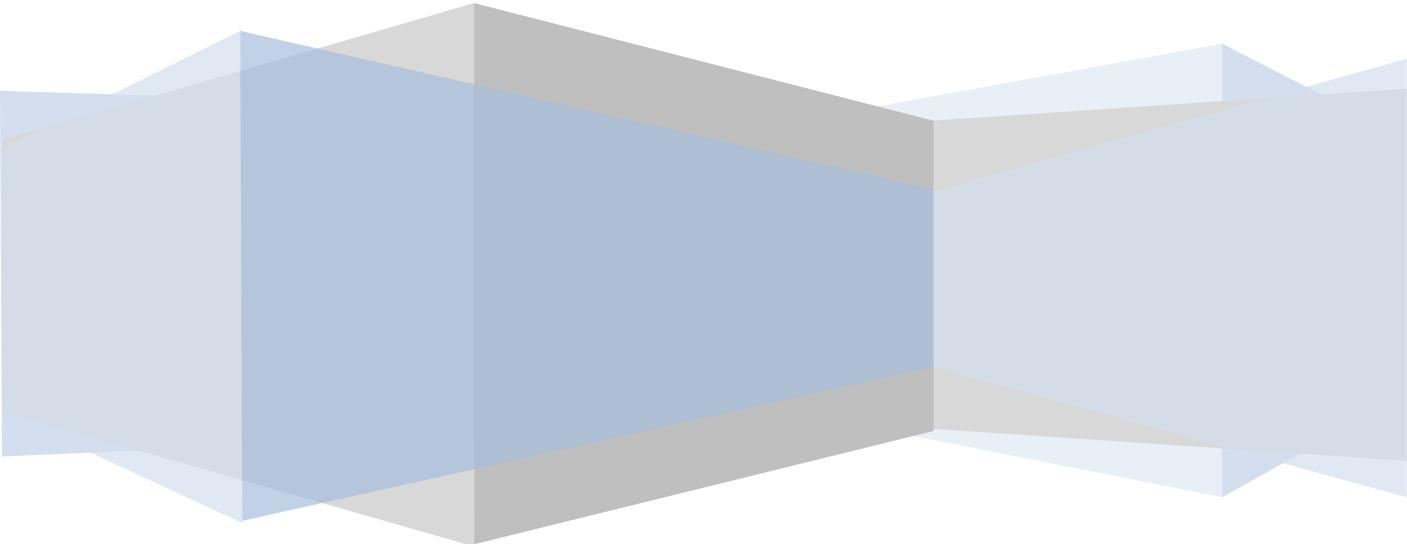
程序设计竞赛进阶训练指南（第二版）

邱秋 编著

```
#include <bits/stdc++.h>

using namespace std;

int main(int argc, char const *argv[])
{
    cout << "Hello World!" << endl;
    return 0;
}
```



内容提要

美国计算机协会（Association for Computing Machinery, ACM）主办的国际大学生程序设计竞赛（International Collegiate Programming Contest, ICPC），是国际上具有较大影响的计算机专业竞赛，目前全球有 110 多个国家和地区约 3000 所大学的近 50000 名大学生参与，加上其他的业余爱好者，参与人数达 20 万。该项竞赛自从 1977 年第一次举办世界总决赛以来，到 2020 年 1 月，已经连续举办了 43 届。

本书是针对 ACM-ICPC 编写的进阶训练指南。以 C++ 进行解题，读者对象是已经具备一定的 C 或者 C++ 基础的编程爱好者，或者是准备参加程序竞赛正在进行训练的高中生，或者是希望通过学习算法和练习以获得进一步提高的大学生。代码采用 GCC 5.3.0 进行编译，使用 C++11 语言标准（需要启用编译符号：`-std=c++11`）。例题和练习以 University of Valladolid Online Judge (UVa OJ) 题库中题号 100—1099 的题目、Halim 的 *Competitive Programming* 所介绍的习题以及作者在写作过程中解决的题目为基础，涵盖了常见的基本算法。

本书适用于有意愿参加 ACM-ICPC 的本科生和研究生，对参加国际信息学奥林匹克竞赛（International Olympiad in Informatics, IOI）的中学生具有指导价值。此外，作为数据结构、算法、程序设计等计算机专业相关课程的拓展与提升，本书也是较好的辅助读物。

自序

兴趣是最好的老师，它可以激发人的创造热情、好奇心和求知欲。
——佚名^I

子曰：“知之者不如好之者，好之者不如乐之者。”
——《论语·雍也》

1998 年，还是在高一的时候，我就对计算机产生了浓厚的兴趣。那时候计算机尚未全面普及，自己家庭条件也一般，根本无力购买“昂贵”的个人电脑，平时只能对着《电脑爱好者》^{II}杂志上的广告，想象自己拥有一台个人电脑的情景。高中期间，每逢周末放假，我都要去县城的新华书店逛一逛，看看是否有新书可“免费”阅读。一次偶然的机会，看到书架上有一本《C 语言教程》，我便不假思索就买了下来并开始自学。那时候学校的微机室建立不久，上面只有最简单的 QBasic，自己经常在上面尝试用 QBasic 语句编写一些小程序。有一次，在学习了高中物理的核裂变反应后，禁不住想使用 QBasic 编写一个程序来演示原子核的裂变反应。具体来说就是模拟中子撞击原子核，原子核分裂，释放出更多中子，这些中子继续撞击其他原子核……最终形成链式反应的过程。我用 QBasic 中的绘图函数绘制一个小点表示中子，用较大的圆圈表示原子核，当中子碰到原子时，表示中子的小点消失，原子分裂为两个，释放一个中子，继续撞击其他原子。当程序最终调试运行成功，看着链式反应的图像逐渐展现的时候，自己的内心非常具有成就感。我想，作为一个编程爱好者，当看到自己的“作品”能够良好地运行或者解决某个编程难题时，那便是最开心和最自豪的“高光”时刻。

不过阴差阳错，自己并未如愿选择感兴趣的计算机专业，而是进入了医学院校，于是编程便成了我最大的业余爱好。在 2011 年的时候，自己用了半年多的时间，完成了由 Skiena 和 Revilla^{III}合著的《挑战编程：程序设计竞赛训练手册》^[1]一书的习题。在全部完成后，感觉书中每一章讲解部分的内容较为简略，使得中低水平的编程爱好者在读完各章节的内容后，难以获得足够的知识来解决相应章节的问题，因此打算写一本用 C++ 来进行解题的参考书籍，以弥补上述不足。但是由于种种原因，一直没有下定决心来做，成为一个“心结”。一方面是已经有很多关于算法和编程竞赛的书籍，例如 Skiena 的 *The Algorithm Design Manual*^[2]，Sedgewick 的 *Algorithms*^[3]，刘汝佳的《算法竞赛入门经典(算法艺术与信息学竞赛)》，Halim 的 *Competitive*

^I 大多数人可能认为这句话是爱因斯坦的名言，万维网上也有人指出这段话出自《爱因斯坦文集》（商务印书馆，1979 年第 1 版）第 3 卷第 144 页。经查阅此版译作，发现是一篇名为《论教育》的演说稿（第 144 页是此文中间部分，文章从第 142 页开始），但该演说稿通篇并未出现这样的字句。更为准确地说，该演说稿仅有一句论及兴趣，原文：

“The same work may owe its origin to fear and compulsion, ambitious desire for authority and distinction, or loving interest in the object and a desire for truth and understanding, and thus to that divine curiosity which every healthy child possesses, but which so often is weakened early.”。“同样工作的动力，可以是恐怖和强制，追求威信荣誉的好胜心，也可以是对于对象的诚挚兴趣，和追求真理与理解的愿望，因而也可以是每个健康儿童都具有的天赋和好奇心，只是这种好奇心很早就衰退了。”望知道确切出处的读者不吝赐教。

^{II} 由中国科学院主管，北京《电脑爱好者》杂志社出版的一本日常计算机应用相关的杂志。

^{III} 2018 年 7 月，在与 uDebug 网站管理员 Vinit Shah 就 UVa 12348 Fun Coloring 的评测问题进行电子邮件交流的过程中，遗憾得知 Miguel Ángel Revilla 教授已于 2018 年 4 月去世。

Programming^[4]等等；另一方面自己也没有足够的时间和精力去进一步深入学习算法，缺乏知识积累和写书的资料。不过非常幸运，从 2015 年 11 月开始，自己终于有许多时间可以做这件事，于是本书逐渐写成。

本书以 C++ 进行解题，读者对象是已经具备一定的 C 或者 C++ 基础的编程爱好者，或者是准备参加程序竞赛正在进行训练的高中生，或者是希望通过学习算法和练习以获得进一步提高的大学生。代码采用 GCC 5.3.0 进行编译，使用 C++11 语言标准（需要启用编译符号：`-std=c++11`）。例题和练习以 University of Valladolid Online Judge (UVA OJ) 题库中题号 100–1099 的题目、Halim 的 *Competitive Programming* 所介绍的习题以及本人在写作过程中解决的题目为基础，涵盖了绝大部分的基本算法。

正如武术宗师的练成，如果不学习各种武术套路，建宗立派就如无根之木、无水之鱼，但是如果将武术套路学“死”，那就容易形成惯性思维，失去自己的创造性，更别谈创立新的武术流派。学习算法的最高境界，我认为和武术宗师的练成类似，既对各派的武功招数、优缺点了如指掌（就如同对基本算法的原理及实现非常熟悉），但又不囿于各派的武功路数（就像深刻理解了算法原理，掌握了算法的精髓所在），能够根据具体情况变通。不过本书并不是一本算法大全，并未将算法的所有细节介绍得面面俱到，只是摘录了要点，列出了关键所在，正所谓“师傅领进门，修行在自身”，需要读者在阅读本书的时候主动查阅相关资料并加以练习，以便进一步加深理解。古人云“授人以鱼，不如授之以渔”¹，我认为学习过程中最重要的是掌握学习的方法，而不是仅仅满足于某种具体算法的掌握，同时也不要被已经学习过的算法束缚了自己的想象力。

不言而喻，对于任何一道题目来说，“理解问题的解决过程”比“记住问题的解决过程”更为重要。解题的途径绝非只有书中所示例的一种。在理解问题的基础上，重新对问题进行定义，从某个新的角度再对原问题进行思考，激发解题的动力和突破既往解题思路的束缚，将已有的算法和数据结构知识予以重新组合，通过语言和编程技术使头脑中的想法变成计算机的实现代码，这样才能够在试题和编程解题间架设一座真正属于自己的桥梁^[5]。我认为这是学习编程的过程中应该努力达到的一种更高境界。

感谢父亲^{II}、母亲、妻子照顾家庭的辛劳付出以及女儿、儿子对于我未能给予更多时间陪伴的理解，因为她（他）们，我能够有时间专心思考，本书才得以完成。阙元伟通读了本书的初稿，对行文上不连贯或描述有歧义的地方提出了修改意见，在此表示衷心的感谢。在编写本书的过程中，参考了许多互联网上的资料和编程爱好者的博客文章，从他（她）们的解题思路中得到了诸多启发，由于篇幅所限，不能在此一一列出致谢。正如散文家陈之藩在《谢天》中所写道的：“……，即是无论什么事，得之于人者太多，出之于己者太少。因为需要感谢的人太多了，就感谢天罢。……”

由于本人水平有限，编写本书的过程，实际上也是一个不断学习和进步的过程，书中的谬误不当之处在所难免，敬请读者不吝指出，以便本书有机会再版时予以改进。如有任何意见或建议，请发送邮件到我的邮箱：metaphysis@yeah.net。

衷心希望读者在阅读本书的过程中能够独立思考，勤加练习，融会贯通，学有所得。祝“切”题愉快！

^I 《淮南子·说林训》中有“临河而羡鱼，不如归家织网”，“授人以鱼，不如授之以渔”是后人引申的说法，现常作“授人以鱼不如授人以渔”。

^{II} 在编著本书的过程中，由于工作的原因，父亲于 2018 年 3 月 17 日来海南帮忙照顾家庭。父亲有多年的高血压病史。海南天气炎热，使得父亲血压控制不理想，加之我对父亲的血压变化关注不够，且父亲自身对高血压可能导致脑出血、偏瘫等意外的风险也未能引起重视，长期服用降血压药物但未能规范检测血压变化……多种不利因素影响，使得父亲于 2018 年 7 月 27 日突发脑血管意外不幸去世，这让我心中深感愧疚，抱憾终生。

邱秋（寂静山林）
二〇二〇年一月一日于海南琼海

前 言

尽信《书》，则不如无《书》。
——《孟子·尽心上》

纸上得来终觉浅，绝知此事要躬行。
——陆游，《冬夜读书示子聿》

一、读者对象

本书的读者对象为计算机专业（或对 ACM-ICPC 竞赛感兴趣的其他专业）学生以及编程爱好者，可以作为 ACM-ICPC 竞赛训练的辅助参考书。本书不是面向初学者的 C++ 语言教程，要求读者已经具备一定的 C 或 C++ 编程基础，有一定的英语阅读能力，了解基本的数据结构，已经掌握了初步的程序设计思想和方法，具备一定程度的算法分析和设计能力。本书的目标是引导读者进一步深入学习算法，同时结合习题来提高分析和解决算法问题的能力。

二、章节安排

本书既是训练指南，又兼有读书笔记的性质。为了表达对 Skiena 和 Revilla 合著的《挑战编程：程序设计竞赛训练手册》一书的敬意（它激发了我对算法的兴趣，促使我编写这本书，可以说是让我对编程竞赛产生兴趣的“启蒙老师”），本书的章节名称及顺序与其完全一致，但叙述方式和具体内容已“面目全非”。每章均以“知识点”为单元进行介绍，每个“知识点”基本上都会有一份解题报告（题目源于 UVa OJ），之后再列出若干题目作为强化练习或者扩展练习。强化练习所给出的题目，一般只需要掌握当前所介绍的知识点就能予以解决。扩展练习所给出的题目，一般需要综合运用其他章节所介绍的知识点，甚至需要自行查询相关资料，对题目所涉及的相关背景知识及算法进行理解、消化、吸收后才能予以解决，其难度相对较高。**题目序号有下划线的练习题目是强烈建议读者尝试并完成的。题目序号后有星标 (*) 的练习题属于较难的题目，在附录中有简要的解题提示。**第 2 章至第 4 章的最后一节内容对一些在解题中常用的算法库函数给出了简要的解析和示例。

三、凡例

一、程序示例：代码使用 Courier New 字体，9pt；注释使用楷体，9pt。伪代码及程序的输入和输出使用 Consolas（或 Palatino Linotype）字体，9pt。脚注中有关历史人物或典籍的注释取自必应网典或 Wikipedia。英语姓名的汉译名参考李学军主编的《英语姓名译名手册，第 5 版》^[6]。

二、《挑战编程：程序设计竞赛训练手册》各章习题的解题代码有两个版本，最初的版本于 2011 年上传至 CSDN^I。2016 年，对部分解题代码进行了修改完善，并对编码风格进行了若干调整，将其与已解决题目的代码合并，上传至 GitHub^{II}。由于 UVa OJ 上的评测数据可能已经发生了变化，个别涉及浮点数精度的代码在提交时可能会得到“Wrong Answer”的评判，但解题的基本思路是正确的，请读者酌情参考使用。

三、本书着重于算法的思想介绍和实现，一般不对算法的正确性给予证明。对正确性证明感兴趣的读者

^I <http://blog.csdn.net/metaphysis>, 2020。

^{II} <https://github.com/metaphysis/Code>, 2020。

可参考标注的文献资料或者相关的专著，或者查阅“算法圣经”——Knuth 的《计算机程序设计艺术》^[7]或 Cormen 等人的《算法导论》^[8]。参考文献或资料仅在第一次引用的时候给出标注，对于后续引用不再予以标注。

四、本书的所有题目中，除个别题目以外，其他题目均选自 UVa OJ，因此不在每道题目前附加 UVa 以区分题目的来源。为了练习选择的便利，题目的右上角使用 A 至 E 的字母来标识此题的“相对难度”。以 2023 年 1 月 1 日为截止日期，按“解决该题目的不同用户数”(Distinct Accepted User, DACU) 进行分级：A (容易): $DACU \geq 2000$; B (偏易): $1999 \geq DACU \geq 1000$; C (中等难度): $999 \geq DACU \geq 500$; D (偏难): $499 \geq DACU \geq 100$; E (较难): $99 \geq DACU$ 。难度等级为 A—C 的题目建议全部完成，难度等级为 D—E 的题目尽自己最大努力去完成（对于某些尝试人数较少的题目，根据上述难度分级原则得到的难度值可能并不能准确反映题目的实际难度，本书适当进行了调整）。如果某道题的 DACU 较少，原因有多种，或者是该题所牵涉到的算法不太常见，具有一定难度；或者是输入的陷阱较多，不容易获得通过；或者是题目本身描述不够清楚导致读题时容易产生歧义；或者是题目的描述过于冗长，愿意尝试的人较少^I；或者是在线评测系统没有提供评测数据，导致无法对提交的代码进行评判^{II}。不管是什么原因，你都应该尝试去解题。对于学有余力的读者来说，研读文献资料并亲自实现算法，然后用习题来检验实现代码的正确性，这是提升能力素质的较好途径。

五、由于本书包含较多代码，为了尽量减少篇幅，每份代码均省略了头文件和默认的命名空间声明。为了代码能够正常运行，请读者直接下载 GitHub 上的代码，或者在手工输入的代码前增加如下的头文件和命名空间声明^{III}：

```
#include <algorithm>
#include <bitset>
#include <cmath>
#include <cstring>
#include <iomanip>
#include <iostream>
#include <limits>
#include <list>
#include <map>
#include <numeric>
#include <queue>
#include <set>
#include <sstream>
#include <stack>
#include <string>
#include <unordered_map>
#include <unordered_set>
#include <vector>

using namespace std;
```

对于使用 GCC 5.3.0（或以上）版本编译器的读者，可以使用下述更为简洁的方式来包含所有头文件：

^I 例如 199 Partial Differential Equations。

^{II} 例如 510 Optimal Routing。

^{III} 头文件<cassert>和<regex>极少使用，未加入列表，不过它们在某些特定题目的示例代码中有应用。

```
#include <bits/stdc++.h>
```

六、本书中的算法实现主旨在于帮助读者理解算法，较多借助 C++ 的标准模板库（Standard Template Library，STL），在运行效率和简洁性上可能并不是最佳的，建议读者在掌握算法后，自行尝试编写更为高效和简洁的算法实现作为自己的标准代码库（Standard Code Library，SCL）。

七、所有示例代码和参考代码均可从本书的配套资源获得。每个章节内属于同一节的示例代码按顺序排列，位于以章节编号命名的文件夹内。例如，第一章第一节第一小节的内容为“整数的表示”，假设该小节包含两份示例代码，则按出现的先后顺序依次命名为 1.1.1.cpp，1.1.2.cpp，如果只包含一份示例代码，则命名为 1.1.1.cpp，均位于文件夹“Books/PCC2/01”内，文件命名在示例代码的第一行给出。文件命名样式：

```
//-----1.1.1.cpp-----//  
// 示例代码。  
//-----1.1.1.cpp-----//
```

表示该示例代码是连续的，位于同一个文件中。文件命名样式：

```
//++++++1.1.1.cpp++++++//  
// 示例代码。  
//++++++1.1.1.cpp++++++//
```

表示该示例代码“跨越”正文的多个段落，即多个示例代码片段构成了整个示例代码，中间有正文分隔。

十一、当参考资料为统一资源标识符（Uniform Resource Identifier，URI）时，其后的四位数字为编写本书时最后一次访问该资源的年份。

目 录

第0章 准备	1
0.1 什么是程序设计竞赛	1
0.1.1 ACM-ICPC	1
0.1.2 Google Code Jam (GCJ)	1
0.1.3 TopCoder	2
0.1.4 CodeForces	2
0.1.5 IOI	3
0.2 如何使用 UVA OJ	3
0.2.1 注册	3
0.2.2 提交	4
0.3 如何选择编程语言	8
0.4 辅助工具	8
0.3.1 UVa Arena	8
0.3.2 uHunt	8
0.3.3 uDebug	8
0.3.4 VirtualBox	8
0.3.5 Virtual Judge	9
0.3.6 洛谷	9
第1章 入门	10
1.1 基本数据类型	10
1.1.1 整数的表示	10
1.1.2 浮点数的表示及精度	11
1.1.3 数据类型的取值范围	14
1.2 格式化输入	15
1.2.1 概述	15
1.2.2 标准输入	16
1.2.3 字符串输入	17
1.3 格式化输出	20
1.3.1 概述	20
1.3.2 输出对齐	23
1.3.3 整数输出	24
1.3.4 实数输出	24
1.3.5 缓冲区与输入输出同步	27
1.4 小结	29
第2章 数据结构	31
2.1 内置数组	31
2.1.1 顺序记录	31
2.1.2 游戏模拟	33
2.1.3 矩阵变换	34
2.1.4 约瑟夫问题	35
2.2 向量	38
2.3 栈	42
2.4 队列及优先队列	46
2.4.1 队列	46
2.4.2 优先队列	49
2.5 双端队列	52
2.6 映射	54
2.7 集合	58
2.8 位集	61
2.9 链表	65
2.10 二叉树	66
2.11 线段树	72
2.11.1 线段树的应用	76
2.11.2 二维线段树	84
2.11.3 可持久化线段树	90
2.11.4 区间树	93
2.11.5 动态开点线段树	错误! 未定义书签。
2.12 树状数组	94
2.13 稀疏表	98
2.14 根号分块	100
2.14.1 根号分块的操作	100
2.14.2 根号分块的应用	102
2.15 块状链表	114
2.15.1 块状链表的操作	114
2.15.2 块状链表的应用	117
2.16 莫队算法	122
2.16.1 莫队算法的原理	123
2.16.2 莫队算法的应用	123
2.17 并查集	123
2.18 二叉排序树	125
2.18.1 二叉排序树的操作	125
2.18.2 二叉排序树的应用	127
2.19 二叉堆	127
2.19.1 二叉堆的操作	127
2.19.2 二叉堆的应用	128
2.20 树堆	128
2.20.1 有旋转树堆的操作	129
2.20.2 无旋转树堆的操作	133
2.20.2 可持久化树堆	135
2.20.3 树堆的应用	136
2.21 左偏树	136
2.21.1 左偏树的操作	137
2.21.2 可持久化左偏树	138

2.21.3 左偏树的应用	139	3.6.6 正则表达式类	203
2.22 伸展树	144	3.7 算法库函数	204
2.22.1 伸展树的操作	144	3.7.1 <code>lexicographical_compare</code>	204
2.22.2 伸展树的应用	144	3.7.2 <code>next_permutation</code> 和 <code>prev_permutation</code>	205
2.23 动态树	145	3.7.3 <code>replace</code>	209
2.24 树链剖分	145	3.7.4 <code>reverse</code>	209
2.25 算法库函数	145	3.7.5 <code>transform</code>	210
2.25.1 <code>accumulate</code> 和 <code>count</code> 、 <code>count_if</code>	145	3.8 小结	210
2.25.2 <code>copy</code> 和 <code>reverse_copy</code>	145		
2.25.3 <code>fill</code>	146		
2.25.4 <code>iota^{c++11}</code>	147		
2.25.5 <code>max</code> 和 <code>min</code>	147		
2.25.6 <code>max_element</code> 和 <code>min_element</code>	148		
2.25.7 <code>memcpy</code> 和 <code>memset</code>	148		
2.26 小结	150		
第3章 字符串	152		
3.1 编码	152	4.1 交换排序	211
3.2 字符串类	153	4.1.1 冒泡排序	211
3.2.1 声明	154	4.1.2 快速排序	212
3.2.2 赋值	155	4.1.3 中位数	213
3.2.3 遍历	156	4.2 插入排序	214
3.2.4 连接与删除	157	4.2.1 直接插入排序	215
3.2.5 查找与替换	158	4.2.2 希尔排序	215
3.2.6 其他操作	159	4.3 选择排序	215
3.3 字符串库函数	159	4.3.1 直接选择排序	216
3.4 字符串类应用	161	4.3.2 堆排序	216
3.4.1 文本解析	161	4.4 归并排序	217
3.4.2 语法分析	165	4.4.1 逆序对数	217
3.4.3 KMP 匹配算法	169	4.5 计数排序	219
3.4.4 扩展 KMP 匹配算法	174	4.6 基数排序	219
3.4.5 Z 算法	177	4.7 桶排序	221
3.4.6 字符串的最小表示	178	4.8 查找	221
3.5 字符串数据结构及应用	179	4.8.1 顺序查找	221
3.5.1 Trie	179	4.8.2 二分查找	222
3.5.2 Aho-Corasick 算法	181	4.8.3 方程求近似解	224
3.5.3 后缀数组	187	4.8.4 最大值最小化问题	225
3.5.4 最长公共子串	196	4.8.5 三分搜索	226
3.5.5 最长重复子串	199	4.8.6 分散层叠	229
3.5.6 Burrows-Wheeler 变换	199	4.9 算法库函数	230
3.6 正则表达式	201	4.9.1 <code>binary_search</code>	230
3.6.1 元字符	201	4.9.2 <code>find</code>	230
3.6.2 转义字符	202	4.9.3 <code>lower_bound</code> 和 <code>upper_bound</code>	231
3.6.3 数量匹配符和分组	202	4.9.4 <code>nth_element</code>	236
3.6.4 字符类和可选模式	202	4.9.5 <code>partial_sort</code>	236
3.6.5 断言	203	4.9.6 <code>sort</code>	237
		4.9.7 <code>stable_sort</code>	240
		4.9.8 <code>unique</code>	240
		4.10 小结	241
第5章 算术与代数	243		
5.1 割鸡焉用牛刀乎	243		
5.2 他山之石，可以攻玉	249		

5.3 高精度整数类的实现	252	6.8 概率论	325
5.4 进制及其转换	261	6.8.1 基本概念	325
5.4.1 R 进制数转换为十进制数	261	6.8.2 条件概率和独立事件	328
5.4.2 十进制数转换为 R 进制数	261	6.8.3 全概率公式与贝叶斯公式	330
5.4.3 任意进制数之间的相互转换	262	6.8.4 随机变量	334
5.4.4 罗马计数法	264	6.8.5 期望	335
5.5 实数	266	6.9 博弈论	343
5.5.1 分数	266	6.9.1 Nim 游戏	344
5.5.2 连续分数	269	6.9.2 Sprague-Grundy 定理	347
5.5.3 分数转换为小数	269	6.9.3 Nim 游戏和 Sprague-Grundy 定理扩展 ..	348
5.5.4 小数转换为分数	270	6.9.4 PN 态分析	354
5.5.5 实数大小的比较	271	6.10 小结	357
5.6 代数	272	第 7 章 数论	359
5.6.1 多项式运算	272	7.1 素数	359
5.6.2 高斯消元法	273	7.1.1 素数判定	361
5.7 幂与对数	280	7.1.2 米勒-拉宾素性测试	361
5.8 实数函数库	283	7.1.3 高斯素数	363
5.9 小结	284	7.1.4 生成素数序列	364
第 6 章 组合数学	285	7.1.5 素因子分解	367
6.1 计数原理	285	7.1.6 完全数	369
6.1.1 加法原理	285	7.2 整除性	370
6.1.2 乘法原理	286	7.2.1 最大公约数	370
6.2 排列与组合	288	7.2.2 扩展欧几里得算法	373
6.2.1 康托展开和康托逆展开	289	7.2.3 线性同余方程	374
6.2.2 方程的整数解个数	295	7.2.4 最小公倍数	376
6.3 PÓLYA 计数定理	295	7.2.5 欧拉函数	377
6.3.1 基本概念	296	7.2.6 莫比乌斯函数	382
6.3.2 Burnside 引理	301	7.3 模算术	384
6.3.3 Pólya 计数定理	304	7.3.1 整数拆分	384
6.4 鸽笼原理	309	7.3.2 可乐兑换	384
6.4.1 拉姆齐理论	311	7.3.3 模运算规则	385
6.5 容斥原理	312	7.3.4 模的逆元	386
6.5.1 错排问题	313	7.3.5 离散对数	387
6.6 初等数列	314	7.3.6 中国剩余定理	389
6.6.1 等差数列	314	7.3.7 波拉德 ρ 启发式因子分解算法	390
6.6.2 等比数列	314	7.4 日期和时间转换	392
6.6.3 其他数列	314	7.4.1 日期转换	392
6.7 计数序列	314	7.4.2 时间转换	397
6.7.1 斐波那契数	315	7.5 小结	399
6.7.2 卡特兰数	319	第 8 章 回溯法	400
6.7.3 欧拉数	323	8.1 八皇后问题	400
6.7.4 斯特林数	323	8.2 搜索	412
6.7.5 调和级数	324	8.2.1 单向搜索	412
6.7.6 其他序列	325	8.2.2 双向搜索	418

8.3 剪枝	419	10.3.2 Kruskal 算法	529
8.3.1 正方形剖分	429	10.3.3 最小生成树的扩展问题	530
8.3.2 关灯问题	431	10.3.4 度限制最小生成树	531
8.4 舞蹈链 X 算法	432	10.3.5 次最优最小生成树	534
8.5 15 数码问题	432	10.4 最短路径问题	538
8.5 小结	442	10.4.1 Moore-Dijkstra 算法	538
第 9 章 图遍历	444	10.4.2 Bellman-Ford 算法	546
9.1 基本概念	444	10.4.3 Floyd-Warshall 算法	551
9.1.1 图的属性	444	10.4.4 传递闭包	553
9.1.2 欧拉公式	445	10.4.5 最小化的最大距离	556
9.1.3 路与连通	445	10.4.6 差分约束系统	556
9.2 图的表示	446	10.4.7 第 K 短路径问题	559
9.2.1 邻接矩阵	446	10.5 网络流问题	562
9.2.2 边列表和前向星	446	10.5.1 基本概念	563
9.2.3 邻接表	447	10.5.2 Ford-Fulkerson 方法	564
9.2.4 链式前向星	448	10.5.3 Edmonds-Karp 算法	567
9.3 图遍历	450	10.5.4 Dinic 算法	573
9.3.1 广度优先遍历	450	10.5.5 ISAP 算法	577
9.3.2 深度优先遍历	456	10.5.6 最小截问题	582
9.4 图遍历的应用	459	10.5.7 最小费用最大流问题	583
9.4.1 图的连通性	459	10.6 边独立集与二部图匹配	586
9.4.2 最短路径	461	10.6.1 网络流解法	587
9.4.3 最长简单路径	463	10.6.2 Hungarian 算法	590
9.4.4 图的着色	463	10.6.3 Hopcroft-Karp 算法	596
9.4.5 最近公共祖先	464	10.6.4 Gale-Shapley 算法	601
9.4.6 割顶	475	10.6.5 Edmonds 算法	603
9.4.7 割边	478	10.7 二部图加权完备匹配	608
9.4.8 强连通分支	481	10.7.1 网络流解法	609
9.4.9 半连通分支	491	10.7.2 Kuhn-Munkres 算法	609
9.4.10 2-SAT	491	10.8 点支配集、点覆盖集、点独立集	617
9.4.11 图的直径	497	10.8.1 点支配集	617
9.4.12 树的重心	499	10.8.2 点覆盖集	617
9.5 拓扑排序	501	10.8.3 点独立集与最大团	620
9.6 小结	504	10.9 路径覆盖和边覆盖	621
第 10 章 图算法	505	10.9.1 最小路径覆盖	621
10.1 基本概念	505	10.9.2 最小边覆盖	622
10.1.1 顶点度	505	10.10 树的相关问题求解	622
10.2 图的回路	506	10.10.1 最小点支配	622
10.2.1 欧拉回	506	10.10.2 最小点覆盖	623
10.2.2 中国投递员问题	523	10.10.3 最大点独立	624
10.2.3 哈密顿回	525	10.11 小结	624
10.2.4 旅行商问题	526		
10.3 最小生成树	527	第 11 章 动态规划	626
10.3.1 Prim 算法	527	11.1 背包问题	626
		11.1.1 01 背包问题	626
		11.1.2 完全背包问题	630

11.1.3 多重背包问题	631	第 12 章 网格	741
11.1.4 背包问题扩展	631	12.1 矩形网格	741
11.2 备忘	634	12.1.1 网格行走	741
11.2.1 $3n+1$ 问题	634	12.1.2 Flood-Fill 算法	743
11.2.2 正交范围查询	636	12.1.3 国际象棋棋盘	746
11.2.3 最大正方形 (长方形)	638	12.1.4 骑士周游问题	749
11.2.4 整数划分	641	12.2 三角形网格	755
11.2.5 博弈树	642	12.3 六边形网格	758
11.2.6 备忘与递推	646	12.4 经度与纬度	758
11.3 松弛	653	12.5 小结	759
11.3.1 Moore-Dijkstra 算法	653	第 13 章 几何	760
11.3.2 Bellman-Ford 算法	656	13.1 点	760
11.3.3 Floyd-Warshall 算法	657	13.2 直线	761
11.4 集合型动态规划	659	13.2.1 直线的表示	761
11.5 区间型动态规划	666	13.2.2 直线间关系	762
11.5.1 矩阵链乘法	666	13.2.3 相互垂直的两条直线交点	763
11.5.2 石子合并问题	668	13.3 坐标和坐标系变换	763
11.6 图论型动态规划	676	13.3.1 平移	764
11.6.1 路径计数	680	13.3.2 旋转	764
11.6.2 树形动态规划	682	13.3.3 缩放	766
11.6.3 旅行商问题	686	13.4 三角形	771
11.6.4 双调欧几里得旅行商问题	687	13.4.1 勾股定理	771
11.7 概率型动态规划	690	13.4.2 三角函数	773
11.8 非典型动态规划	694	13.4.3 正弦定理	774
11.9 动态规划的优化	697	13.4.4 余弦定理	774
11.9.1 空间优化	697	13.4.5 三角形面积	778
11.9.2 状态优化	697	13.4.6 三角函数库	779
11.9.3 二进制优化	701	13.4.7 桌球碰撞问题	780
11.9.4 单调队列优化	701	13.5 多边形	782
11.9.5 斜率优化	705	13.5.1 矩形	782
11.9.6 四边形不等式优化	709	13.5.2 四边形和正多边形	784
11.10 子序列和子串问题	712	13.6 圆	785
11.10.1 最短编辑距离	712	13.6.1 圆的周长和面积	785
11.10.2 最长公共子序列	716	13.6.2 圆的切线	787
11.10.3 最长公共子串	717	13.6.3 三角形的内切圆与外接圆	790
11.10.4 最长递增子序列	718	13.6.4 圆与圆的位置关系	791
11.10.5 最长不重复子串	722	13.6.5 最小圆覆盖	796
11.10.6 最长回文子串	723	13.7 小结	798
11.10.7 最大连续子序列和 (积)	727	第 14 章 计算几何	799
11.11 贪心算法	729	14.1 基本概念	799
11.11.1 部分背包问题	731	14.1.1 线段	799
11.11.2 纸币找零问题	731	14.1.2 多边形	799
11.11.3 硬币兑换问题	735	14.2 几何对象间的关系	800
11.11.4 霍夫曼编码	736		
11.11.5 最优策略选择	738		
11.12 小结	739		

14.2.1 向量、内积和外积	800	14.6.1 Pick 定理	843
14.2.2 点和直线的关系	803	14.6.2 多边形面积	844
14.2.3 确定线段转动方向	804	14.6.3 多边形重心	845
14.2.4 确定线段是否相交	805	10.6.4 三维几何体的表面积和体积	847
14.2.5 点的投影	809	14.7 半平面交问题	848
14.2.6 点的映像	811	14.7.1 凸多边形切分	848
14.2.7 点和直线间距离	811	14.7.2 多边形内核	854
14.2.8 点和线段间距离	812	14.8 最近点对问题	854
14.2.9 线段和线段间距离	813	14.9 最远点对问题	858
14.2.10 点和多边形的关系	813	14.10 三维空间计算几何	863
14.2.11 直线和圆的交点	817	14.10.1 点	864
14.2.12 圆和圆的交点	818	14.10.2 直线	865
14.2.13 圆的切点	819	14.10.3 平面	868
14.3 扫描线算法	820	14.10.2 三维凸包	874
14.4 坐标离散化	822	14.11 小结	877
14.4.1 最大化矩形问题	825	附录	879
14.4.2 矩形并的面积	826	1 ASCII 表	879
14.4.3 矩形并的周长	829	2 C++运算符优先级	880
14.5 凸包	833	3 解题提示	881
14.5.1 Graham 扫描法	833	4 习题索引	931
14.5.2 Jarvis 步进法	837	参考资料	958
14.5.3 Andrew 合并法	840		
14.5.4 Melkman 算法	842		
14.6 公式及定理应用	843		

第 0 章 准备

凡事预则立，不预则废。
——《礼记·中庸》

本章旨在帮助读者了解什么是程序设计竞赛以及如何开始自己的第一次挑战。如果读者已经熟悉相关内容，可以跳过此章，直接从第 1 章开始阅读。

0.1 什么是程序设计竞赛

本书所指的程序设计竞赛是解题竞赛，是参赛者利用自己所学的计算机相关知识，在限定的时间内解决若干道具有一定难度的编程题目。这些题目一般都与某种算法有关，如果读者没有相关的训练，难以在限定时间内予以解决。除了解题竞赛以外，还有很多其他类型的程序设计竞赛，例如以提高程序运行效率为目标的性能竞赛，以完成某个具有特定功能的软件为目标的创意竞赛等。以下介绍一些具有较大影响的解题程序设计竞赛。

0.1.1 ACM-ICPC

美国计算机协会（Association for Computing Machinery, ACM）主办的国际大学生程序设计竞赛（International Collegiate Programming Contest, ICPC），是历史最悠久的国际大学生程序设计竞赛，其目的在于使大学生运用计算机来充分展示自己分析问题和解决问题的能力^I。

该项竞赛自从 1977 年第一次举办世界总决赛以来，截至 2020 年 1 月，已经连续举办了 43 届。该项竞赛一直受到国际各知名大学的重视，全世界各大 IT 企业也给予了高度关注，有的（例如 IBM、Oracle、惠普、微软等公司）还经常出资赞助比赛的进行。

ACM-ICPC 以团队的形式代表各学校参赛，每队由至多 3 名队员组成。每位队员必须是在校学生，有一定的年龄限制，并且每年最多可以参加两站区域选拔赛。比赛期间，每队使用 1 台电脑需要在 5 个小时内使用 C/C++、Java 和 Python 中的一种编写程序解决 7 到 13 个问题。程序完成之后提交裁判运行，运行的结果会判定为正确或错误两种并及时通知参赛队。最后的获胜者为正确解答题目最多且总用时最少的队伍。

与其它计算机程序竞赛相比，ACM-ICPC 的特点在于其题量大，每队需要在 5 小时内完成 7 道或以上的题目。另外，一支队伍 3 名队员却只有 1 台电脑，使得时间显得更为紧张。因此除了扎实的专业水平，良好的团队协作和心理素质同样是获胜的关键。

0.1.2 Google Code Jam (GCJ)

Google Code Jam 谷歌全球编程挑战赛是 Google 举行的一项国际编程竞赛，目标是为 Google 选拔顶尖的工程人才^{II}。

比赛的每道题目均由 Google 的工程师仔细设计，既有趣也极具挑战性，选手不仅能测试自己的编程水平，更能在比赛环境中快速提升实践技能，丰富自身履历。该项赛事始于 2003 年，竞赛内容包括在限定时

^I ACM 的官方网站：<https://www.acm.org>，ICPC 的官方网站：<https://icpc.global/>。

^{II} Google Code Jam 的官方网站：<https://codingcompetitions.withgoogle.com/codejam/>。

间内解决一系列特定的算法问题，编程语言和环境的选择不受限制。每年竞赛中所有参赛者在经过 4 轮线上比赛后，将会诞生 25 位选手参加在不同 Google Offices 地点举办的 The World Finals 全球总决赛，竞争现金大奖及奖杯。

0.1.3 TopCoder

TopCoder 是一个面向平面设计师和程序员的网站，它采用比赛、评分、支酬等方式吸引众多平面设计师和程序员业余工作^I。

该网站每个月都有两到三次在线比赛，根据比赛的结果对参赛者进行新的排名。参赛者可根据自己的爱好选用 Java, C++, C#, VB 或 Python 进行编程。参赛者须在 1 小时 15 分钟的时间内完成三道不同难度的题目，每道题完成的时间决定该题在编程部分所得的分数。比赛可分为三部分：Coding Phase, Challenge Phase 和 System Test Phase，比 ACM-ICPC 多了 Challenge Phase，这部分是让参赛者浏览分配在同一房间的其他参赛者的源代码，然后设法找出其中错误，并构造一组数据使其不能通过测试。如果某参赛者的程序不能通过别人或系统的测试，则该参赛者在此题目的得分将为零。

0.1.4 CodeForces

CodeForces 是一个提供在线评测系统的俄罗斯网站^{II}，该网站由一群来自俄罗斯萨拉托夫国立大学的程序员创建并维护，其中主要的领导者为 Mike Mirzayanov。

CodeForces 的每位用户在参加比赛后都会有一个得分，系统根据得分以及在以往比赛中的表现赋予用户一个 Rating 并冠以不同的头衔，名字也会以不同的颜色显示。参加比赛的用户按 Rating 以某个分值为界划分为 Div1 和 Div2 两类。相应地，CodeForces 上的比赛也会指明是 Div1 还是 Div2，抑或同时进行。Div1 的比赛较难，Div2 的比赛较简单。如果同时进行，Div1 的 A、B、C 三题会和 Div2 的 C、D、E 三题相同。每次比赛结束后 Rating 都会依据此前各个选手的 Rating 和公式重新计算。

比赛中，选手有两个小时的时间去解决五道题目，这里的“解决某道题目”是指预测试通过，即通过了一次仅含部分测试点的测评，而最终决定是否得到这道题的分数，要看比赛结束后的统一测评。某道题的分数随着时间线性减少，但不会低于初始分值的 30%，也就是说，你解决问题的速度越快，得分也相应越高。而且，对于参数者的每次提交都会扣除一定的分数。例如，某道题目的初始分为 1000 分，每过 1 分钟，该题的分数减少 4 分，如果你在 10 分钟内尝试提交了 3 次并在第 3 次最终通过了预测试，每次错误提交扣除 50 分，那么该题的得分为 $1000 - 4 \times 10 - 2 \times 50 = 860$ 分。

同一个 Div 的选手将被划分到若干个房间里，每个房间约 20 位参赛者。当某道题的预测试通过之后，参赛者可以选择锁定该题代码，锁定该代码后，你之后将无法就该道题目进行再次提交（即使发现代码中包含错误）。之后就可以查看同一个房间内其他参赛者的代码并试图找出其中的漏洞。你可以自己构造一个测试（可以是数据，也可以是数据生成器）使得该代码不能通过，称之为 Hack（有时也称 Challenge）。一次成功的 Hack 可以得 100 分，而如果没有成功，将会被扣 50 分。最后，所有通过预测试的代码将提交进行最终测试，参赛者的最终得分为通过最终测试的代码分数和 Hack 其他参赛者所获得（扣除）的分数。

CodeForces 的题目偏向于考察解题思路，一般较少涉及复杂的算法，标程的代码一般都比较简短而精巧。

^I TopCoder 的官方网站：<https://www.topcoder.com/>。

^{II} CodeForces 的官方网站：<https://codeforces.com/>。

0.1.5 IOI

国际信息学奥林匹克竞赛 (International Olympiad in Informatics, IOI)，是面向中学生的一年一度的信息学科竞赛^I。

这项竞赛包含两天的计算机程序设计，解决算法问题。选手以个人为单位，每个国家最多可选派四名选手参加。参赛选手从各国相应计算机竞赛中选拔。国际信息学奥林匹克竞赛属于智力与应用计算机解题能力的比赛，题目有相当的难度，解好这类题目，需要具备很强的综合能力。首先是观察和分析问题的能力；第二是将实际问题转化为数学模型的能力；第三是灵活地运用各种算法的能力；第四是熟练编写程序并将其调试通过的能力；第五是根据题目的要求，自己设计测试数据，检查自己的解法是否正确，是否完备的能力。能够参加 IOI 的选手应该具有很强的自学能力和动手能力，需要学习有关组合数学、图论、基本算法、数据结构、人工智能搜索算法及数学建模等知识，还要学会高级语言和编程技巧，要具备很强的上机操作能力。国际信息学奥林匹克竞赛鼓励创造性，在评分的标准上给予倾斜，创造性强的解题方法可以拿到高分。

0.2 如何使用 UVa OJ

随着 ACM-ICPC 程序设计竞赛的推广，各种在线评测 (Online Judge, OJ) 网站及工具应运而生。其中历史最悠久的当数 University of Valladolid Online Judge (简称 UVa OJ 或 UVa)^{II}。UVa OJ 的特点是题目丰富、题型多样，比较适合中等水平的 ACM-ICPC 选手进行训练。

0.2.1 注册

要想在 UVa OJ 上解题，必须先注册一个账号。进入 UVa OJ 的官方主页：<https://onlinejudge.org/>，点击页面左侧的“Register”，跳转到账户注册页面，填写必要的信息之后，UVa OJ 会向你填写的邮箱地址发送一封验证邮件，通过邮件验证后，账户即激活。

Registration

Name:

Email:

Username:

Password:

Verify Password:

Former UVa ID:

Results email:

Virtual Judge:

I'm not a robot reCAPTCHA Privacy - Terms

Your IP address is: 112.67.215.111

^I IOI 的官方网站：<https://ioinformatics.org/>。

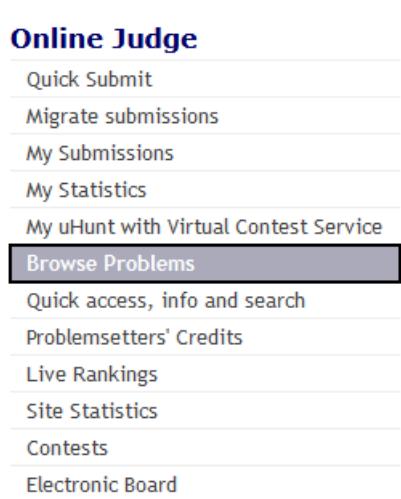
^{II} UVa OJ 的官方网站：<https://onlinejudge.org/>。2018 年 4 月，创建 UVa OJ 的 Miguel Ángel Revilla 教授不幸去世，而 Valladolid 大学校方宣布不再继续提供资金以维持 UVa OJ 的运行，因此 UVa OJ 有停止运行的可能。不过好消息是，Revilla 教授的儿子 Miguel Revilla Rodríguez 表示愿意继续为 UVa OJ 的运行做出努力。目前他正在使用 Wt 框架 (<https://www.webtoolkit.eu/wt>) 开发新一版的在线评测系统 (<https://github.com/TheOnlineJudge/ojudge>)。乐观地估计，有望在 2022 年正式上线运行。

图 0-1 注册 UVa OJ 账号

0.2.2 提交

在注册并登陆账号之后，您就可以选择题库中的题目开始解题并提交答案。提交有两种方式，一种是先浏览到具体的题目描述界面，点击“Submit”进行提交，另外一种是“Quick Submit”。

先介绍第一种方法。以提交 UVa 100 The 3n+1 Problem 为例，首先点击左侧功能栏中的“Browser Problems”，



之后选择“Problem Set Volumes (100...1999)”，

Root		
	Title	Total Solved
📁	Problem Set Volumes (100...1999)	
📁	Contest Volumes (10000...)	
📁	Interactive Problems	
📁	Programming Challenges (Skiena & Revilla)	
📁	ACM-ICPC World Finals	
📁	ACM-ICPC Dhaka Site Regional Contests	
📁	Western and Southwestern European Regionals	
📁	Prominent Problemsetters	
📁	Rujia Liu's Presents	

然后选择“Volume 1 (100-199)”，

Root :: Problem Set Volumes (100...199)

Title	Total Sol
Volume 1 (100-199)	
Volume 2 (200-299)	
Volume 3 (300-399)	
Volume 4 (400-499)	
Volume 5 (500-599)	
Volume 6 (600-699)	
Volume 7 (700-799)	
Volume 8 (800-899)	
Volume 9 (900-999)	
Volume 10 (1000-1099)	
Volume 11 (1100-1199)	

然后点击“100 - The $3n+1$ Problem”：

Root :: Problem Set Volumes (100...199)

Title	Total Subm
100 - The $3n+1$ problem	886396
101 - The Blocks Problem	117273
102 - Ecological Bin Packing	107998
103 - Stacking Boxes	45075
104 - Arbitrage	32616
105 - The Skyline Problem	61644
106 - Fermat vs. Pythagoras	29499
107 - The Cat in the Hat	52507
108 - Maximum Sum	69044

进入题目描述页面，点击“Submit”，

Root

100 - The $3n+1$ problem

Time limit: 3.000 seconds

 Submit  Statistics
 Debug  PDF

将解题代码粘贴到输入框中（或者点击“Browser”选择本机上的代码文件上传），单击“Submit”即

可。

100 - The 3n + 1 problem

Language

ANSI C 5.3.0 - GNU C Compiler with options: -lm -lcrypt -O2 -pipe -ansi -DONLINE_JUDGE

JAVA 1.8.0 - OpenJDK Java

C++ 5.3.0 - GNU C++ Compiler with options: -lm -lcrypt -O2 -pipe -DONLINE_JUDGE

PASCAL 3.0.0 - Free Pascal Compiler

C++11 5.3.0 - GNU C++ Compiler with options: -lm -lcrypt -O2 -std=c++11 -pipe -DONLINE_JUDGE

PYTH3 3.5.1 - Python 3

Paste your code...

```
// The 3n+1 Problem
// UVa ID: 100
// Verdict: Accepted
// Submission Date: 2011-05-22
// UVa Run Time: 0.032s
//
// 版权所有 (C) 2011, 邱秋。metaphysis # yeah dot net.
```

...or upload it No file selected.

提交成功后，会显示该提交的编号：

Submission received with ID 26243427

第二种提交方法适用于你已经熟悉了题目描述而且已经编写了题目的解题代码的情形，**用户只需选择浏览器左侧功能栏中的“Quick Submit”选项：**

Online Judge

Quick Submit

[Migrate submissions](#)

[My Submissions](#)

[My Statistics](#)

[My uHunt with Virtual Contest Service](#)

[Browse Problems](#)

[Quick access, info and search](#)

[Problemsetters' Credits](#)

[Live Rankings](#)

[Site Statistics](#)

[Contests](#)

[Electronic Board](#)

然后再额外填写问题的编号，其他项目与第一种方法的相同。

Quick Submit

Problem ID

Language ANSI C 5.3.0 - GNU C Compiler with options: -lm -lcrypt -O2 -pipe -ansi -DONLINE_JUDGE
 JAVA 1.8.0 - OpenJDK Java
 C++ 5.3.0 - GNU C++ Compiler with options: -lm -lcrypt -O2 -pipe -DONLINE_JUDGE
 PASCAL 3.0.0 - Free Pascal Compiler
 C++11 5.3.0 - GNU C++ Compiler with options: -lm -lcrypt -O2 -std=c++11 -pipe -DONLINE_JUDGE
 PYTH3 3.5.1 - Python 3

Paste your code...

...or upload it No file selected.

在提交以后，就可以通过点击浏览器左侧功能栏中的“*My Submissions*”查看提交结果。

26243427	100 The 3n + 1 problem	Accepted	C++11	0.000	2021-03-29 12:32:10
----------	------------------------	----------	-------	-------	------------------------

在 UVa OJ 上提交，可能会有以下可能的结果：

Accepted (AC): 通过。你的程序在限定的时间和内存下产生了正确的输出，恭喜该题获得通过！

Wrong Answer (WA): 错误提交。你的程序产生的输出与参考输出不匹配。

Presentation Error (PE): 格式错误。你的代码所产生的输出内容是正确的，但是格式有错误。检查输出是否有多余的空格，或者对齐、换行符是否正确。

Compile Error (CE): 编译错误。你的代码存在语法或其他错误而无法被正确编译。

Runtime Error (RE): 运行时错误。你的代码在运行过程中出现错误被强制退出。例如，引用了声明范围以外的数组元素，除数为零错误等。

Time Limit Exceeded (TLE): 时间超出限制。你的代码所采用的算法时间效率不高或者出现无限循环，导致程序运行时间超出限制。

Memory Limit Exceeded (MLE): 内存超出限制。你的代码内存使用效率不高或者出现无限循环，导致程序使用的内存超出限制。

Output Limit Exceeded (OLE): 输出超出限制。你的代码产生的输出太长，超出了输出限制。一般是因为存在无限循环而导致输出过长。

Submission Error (SE): 代码提交未成功。在代码提交处理的过程中由于某些错误或提交的数据损坏而导致提交失败。

Restricted Function (RF): 限制使用的函数。某些系统函数在评测中限制使用，你的代码中包含这些限制使用的函数时会出现该错误。

Can't Be Judge (CJ): 无法评测。所选择的题目可能不存在测试输入或者测试输出，因此无法进行评测。

In Queue (QU): 评测机繁忙未能对你的提交进行评判，当评测机空闲时将尽快对你的提交进行评判。

0.3 如何选择编程语言

一般来说，在编程竞赛中比较的是谁能够在有限的时间内解决最多的问题，因此程序只要能够在给定的时限内通过即可，语言的效率并不是第一位的考虑因素。由于 C++ 具有丰富的库函数以及字符串类，因此推荐使用 C++ 作为编程竞赛的首选语言。如果某些竞赛对语言的运行效率要求非常高，又或者不允许使用 C++ 的库函数，抑或需要自行实现某些基本的数据结构（例如堆），那么可以考虑使用 C 语言。

值得一提的是，其他语言由于一些非常便利的特性，如果学有余力，也建议适当掌握。例如 Java 中的高精度整数类 BigInteger 在解决有关大数的问题时就非常有用。同样的，掌握 Python 也会在某些场景下带来便利，例如，Python 本身就直接支持高精度整数的运算。

0.4 辅助工具

本节介绍若干有助于提高学习效率的工具，熟练掌握和应用这些工具可以使得你的学习过程事半功倍。

0.3.1 UVa Arena

UVa Arena 是一款用于 UVa OJ 解题的辅助软件^I，可以通过该软件下载整个 UVa OJ 题库，并且可以从网上数据库更新你的解题完成情况信息。除此之外，还包括一些诸如保存解题代码、调用编译器编译、代码提交等实用功能。

0.3.2 uHunt

uHunt 是一个跟踪 UVa OJ 解题情况的网站^{II}，输入你在 UVa OJ 上的用户名即可查看你目前题目的完成情况，还可根据题目的通过率、通过提交数等对题目排序，便于寻找符合自己需求的题目进行训练。

0.3.3 uDebug

uDebug 是专门针对网上各大题库建立的测试数据网站^{III}，它包括绝大多数的 UVa OJ 上题目的测试数据。用户可以先下载测试数据（一般通过复制粘贴的方式），使用自己的解决方案生成输出，然后与网站上已经获得过“Accepted”的程序的输出进行比对，检查是否匹配以发现自己代码的问题。

0.3.4 VirtualBox

VirtualBox 是一款虚拟机软件^{IV}，通过该软件可以在 Windows 操作系统上安装一个“真正”的 Linux 系统，在此虚拟系统中配置 GCC 编译器搭建编程环境即可开始 UVa OJ 解题，便于使用 Windows 系统但

^I UVa Arena 下载链接: <https://github.com/dipu-bd/UVA-Arena>。

^{II} uHunt 网站链接: <http://uhunt.felix-halim.net>。

^{III} uDebug 网站链接: <https://www.udebug.com/>

^{IV} VirtualBox 网站链接: <https://www.virtualbox.org/>。

又不愿意安装双系统的用户使用。

0.3.5 Virtual Judge

Virtual Judge 是一个虚拟评判网站^I，通过该网站可以完成对各大主要在线评测网站的问题进行间接提交，因为国内访问 UVa OJ 经常出现速度较慢甚至无法打开网页的情况，可以通过此网站进行提交，能够较为快速地查看提交结果。

0.3.6 洛谷

洛谷是国内较为知名的**信息学奥林匹克**网站^{II}，在该网站的题库中，包含了 UVa OJ 的几乎全部题目，其中有一部分给出了中文翻译，对于英语水平不是很好的解题者较为友好。解题者可以通过洛谷将 UVa OJ 上注册的账号予以绑定，之后使用洛谷的相应接口进行代码远程提交并获得评判结果。

^I Virtual Judge 网站链接：<https://vjudge.net/>。

^{II} 洛谷网站链接：<https://www.luogu.com.cn/>。

第1章 入门

故不积跬步，无以至千里；不积小流，无以成江海。
——《荀子·劝学篇》

UVa OJ 上的题目偏向于在输入和输出上设置一些“陷阱”(trick)，而不只是单纯地让你解决算法问题。完全理解题意和严格遵循输出格式非常重要，你需要仔细研读题目的输入和输出部分，否则很有可能会因为一些细小的错误导致多次提交却得不到通过。另外，需要特别注意边界情况的处理，很多时候，算法或者解题思路本身是正确的，但由于对边界情况考虑不周全而无法获得“Accepted”的提交结果。

1.1 基本数据类型

1.1.1 整数的表示

在计算机的内部实现中，一般使用二进制来表示整数。二进制整数分无符号整数和有符号整数。在有符号二进制数中，其首位指定为符号位，符号位为 1 表示负数，为 0 表示正数。

给定一组二进制位 $[x_{w-1}, x_{w-2}, \dots, x_0]$ ，如果它表示的是无符号整数 u ，有

$$u = \sum_{i=0}^{w-1} x_i 2^i$$

例如，若二进制数 1100101101_2 表示的是无符号整数，它所对应的十进制数为

$$1100101101_2 = 1 \times 2^9 + 1 \times 2^8 + \dots + 0 \times 2^1 + 1 \times 2^0 = 813_{10}$$

如果二进制位表示的是一个有符号整数 s ，因为计算机内部一般使用 2 的补码 (Two's complement) 来表示有符号整数^I，有

$$s = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

若 1100101101_2 表示的是一个有符号整数，它所对应的十进制数为

$$1100101101_2 = -1 \times 2^9 + 1 \times 2^8 + \dots + 0 \times 2^1 + 1 \times 2^0 = -211_{10}$$

在存储整数时，一般按字节为逻辑单位进行存储，有“小端序”和“大端序”之分。小端序 (little-endian) 是指将表示整数的低位字节存储在内存地址的低位，高位字节存储在内存地址的高位。如果将整数 19820624_{10} 存储至内存，由于

$$19820624_{10} = ([00000001] [00101110] [01110000] [01010000])_2$$

使用小端序存储时，如果内存存储起始地址为 $0x100$ ，从低位地址到高位地址存储的内容依次为

$$[01010000]_{0x100} [01110000]_{0x101} [00101110]_{0x102} [00000001]_{0x103}$$

而使用大端序 (big-endian) 方式存储时，从低位地址到高位地址存储的内容依次为

$$[00000001]_{0x100} [00101110]_{0x101} [01110000]_{0x102} [01010000]_{0x103}$$

^I 使用 2 的补码来表示有符号整数便于计算机对二进制的加法和减法进行统一处理。

可以看到，存储顺序正好相反。

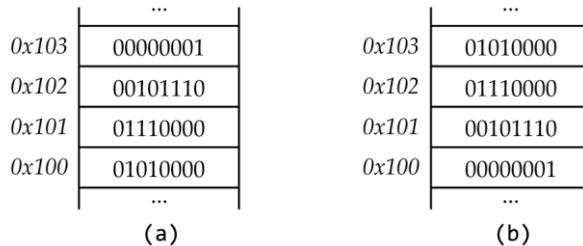


图 1-1 (a) 小端序存储方式内存中各字节的内容。(b) 大端序存储方式内存中各字节的内容

不过计算机具体使用哪种端序对程序员来说是透明的，而且两种端序之间并没有什么优劣之分。之所以会出现两种端序，原因是计算机出现的初期，各个硬件厂商在具体实现时所做选择的不同^[9]。

利用 C++ 中的 union 数据结构（或者指针）可以很容易确定计算机使用的是何种端序。

```
-----1.1.1.cpp-----
union {
    unsigned int bytes;
    unsigned char lowerByte;
} block;
int main(int argc, char *argv[]) {
    // 第一种方式：利用 union 的特性，将 block 的第一个成员赋值为 1，然后获取内存
    // 低位字节的值。如果是小端序，低位字节存储的值为 1；若为大端序，则值为 0。
    block.bytes = 1;
    cout << (block.lowerByte ? "little-endian" : "big-endian") << endl;
    // 第二种方式：利用指针直接获取低位字节的值。如果是小端序，低位字节存储的值为 1；
    // 若为大端序，则值为 0。
    unsigned int bytes = 1;
    cout << (*((char *)(&bytes))) ? "little-endian" : "big-endian" << endl;
    return 0;
}
-----1.1.1.cpp-----
```

强化练习：[594 One Little Two Little Three Little Endians^A](#)。

扩展练习：[12720 Algorithm of Phil^D](#)。

1.1.2 浮点数的表示及精度

根据 IEEE 754-2019 标准^{[10][11]}，任意一个浮点数 f 可以表示为

$$f = (-1)^s \times m \times b^e$$

其中 s 为符号（sign），使用一个二进制位表示，0 表示正数，1 表示负数。 m 为尾数（significand），尾数实际上是有效数字（significant digits）的一种表示，通过选择不同的阶码，可以得到不同的尾数，为了统一，规定 $1 \leq m < 2$ 。 b 为浮点数所使用进制的基数（radix），一般使用二进制或十进制，对于二进制浮点数而言， $b=2$ 。 e 为阶码（exponent），类似于科学计数法中 10 的幂次。例如，将十进制浮点数 -12.5_{10} 表示成上述格式时，由于 12.5_{10} 对应的二进制带小数格式为 1100.1_2 ，有

$$-12.5_{10} = (-1)^1 \times 1.1001_2 \times 2^3$$

对应的参数为： $s=1$ ， $b=2$ ， $m=1.1001_2$ ， $e=3$ 。



图 1-2 二进制可交换浮点数格式。S 为符号 (sign)，使用一个二进制位表示；E 为移码 (biased exponent)，使用实际指数加上偏移表示；T 为尾数。默认的 d_0 已经编码到指数位中，故存储的位数从 d_1 开始。32 位浮点数的移码 E 为 8 位，尾数 T 为 23 位；64 位浮点数的移码 E 为 11 位，尾数 T 为 52 位

IEEE 754 标准规定，在计算机内部保存尾数 m 时，默认该数的第一位总是 1，这样的二进制浮点数称为规范浮点数 (normalized float number)，简称规范数。由于规范数的尾数 m 的第一位总是 1，实现时可以舍去，只保存后面的小数部分，因此规范数的尾数 m 满足 $0.5 \leq m < 1$ 。例如，在保存 1.01101_2 的时候，可以只保存 01101_2 ，在读取时，把舍去的第一位上的 1 再恢复，这样可以节省一位有效数字存储空间。以 32 位浮点数为例，尾数 m 有 23 位，将第一位的 1 舍去以后，等效于可以保存 24 位有效数字。

阶码 e 的表示稍复杂。标准规定 e 为一个无符号整数，如果是 32 位浮点数， e 为 8 位，它的取值范围是 $[0, 255]$ ；如果是 64 位浮点数， e 为 11 位，它的取值范围是 $[0, 2047]$ 。但是，科学计数法中的 e 可以为负数，所以 IEEE 754 又附加规定： e 的实际值必须再加上一个偏移 (bias) 以得到移码 E 。对于 8 位的 e ，这个偏移是 127；对于 11 位的 e ，这个偏移是 1023。例如， 2^3 的 e 是 3，保存为 32 位浮点数时， $E = 3 + 127 = 130$ ，即 10000010_2 。最终 -12.5_{10} 的 32 位二进制可交换浮点数格式编码为

$$-12.5_{10} = (-1)^1 \times 1.1001_2 \times 2^3 = 1_S 10000010_E 10010000000000000000000000000000_T$$

可以通过以下代码对上述结果进行验证。

```
-----1.1.2.1.cpp-----
void toReadable(string s) {
    cout << "S = " << s.substr(0, 1);
    cout << " E = " << s.substr(1, 8);
    cout << " T = " << s.substr(9);
}
int main(int argc, char *argv[]) {
    float f = -12.5;
    // 将存储浮点数的四个字节解释为一个无符号整数。
    unsigned int *ui = (unsigned int *)(&f);
    // 将无符号整数表示成二进制形式并输出，该二进制编码即为浮点数在内存中的编码。
    bitset<32> uis(*ui);
    toReadable(uis.to_string());
    cout << endl;
    return 0;
}
-----1.1.2.1.cpp-----
```

其输出为：

```
S = 1 E = 10000010 T = 10010000000000000000000000000000
```

扩展练习：11809 Floating-Point Numbers^C。

根据 IEEE 754 标准，`float` 数据类型有 23 位（二进制）尾数，如果保存的是规范数，即小数点的左侧始终为 1，那么可以节省 1 位存储空间，相当于使用 23 位的存储空间保存 24 位的尾数。由于 24 位二进制数能够保存的最大整数为

$$2^{24} - 1 = 16777215$$

当使用单精度浮点数来保存整数时，如果整数的有效数字个数超过 8 位，很可能无法精确保存。注意，有效数字个数是指数值的全部数字的个数，并不是仅仅指小数点后面的数字个数。例如，123456789.0，整数部分有效数字共 9 位，虽然小数点后只有一位，但是无法用 float 数据类型精确表示到小数点后一位，而最多能精确到百位，即只能保证误差不大于 10；而对于 1234567.0，其整数部分有效数字为 7 位，可以使用 float 数据类型精确表示^I。一般来说，使用 float 数据类型，其精度能够保存 6 到 7 位有效数字^{II}。

```
-----1.1.2.2.cpp-----
void toReadable(string s) {
    cout << "S = " << s.substr(0, 1);
    cout << " E = " << s.substr(1, 8);
    cout << " T = " << s.substr(9);
}
int main(int argc, char *argv[]) {
    float f1 = 123456789.0, f2 = 1234567.0;
    // 将浮点数 f1 以二进制编码形式输出。
    unsigned int *ui1 = (unsigned int *)(&f1);
    bitset<32> uis1(*ui1);
    cout << "original value: 123456789.0" << '\n';
    cout << "stored value: " << fixed << f1 << '\n';
    toReadable(uis1.to_string());
    cout << "\n\n";
    // 将浮点数 f2 以二进制编码形式输出。
    unsigned int *ui2 = (unsigned int *)(&f2);
    bitset<32> uis2(*ui2);
    cout << "original value: 1234567.0" << endl;
    cout << "stored value: " << fixed << f2 << '\n';
    toReadable(uis2.to_string());
    cout << '\n';
    return 0;
}
-----1.1.2.2.cpp-----
```

其输出为：

```
original value: 123456789.0
stored value: 123456792.000000
S = 0 E = 10011001 T = 110101101111001101100011

original value: 1234567.0
stored value: 1234567.000000
S = 0 E = 10010011 T = 00101101011010000111000
```

^I $123456789.0 = 1.11010110111100110100010101_2 \times 2^{26}$ ，由于单精度浮点数只能保存 23 位尾数，因此计算机内部将其四舍五入表示为： $123456789.0 \approx 1.11010110111100110100011_2 \times 2^{26} = 123456792$ ，能够精确表示到百位，绝对误差为 3，其移码 $E = 26 + 127 = 153 = 10011001_2$ 。 $1234567.0 = 1.00101101011010000111_2 \times 2^{20}$ ，由于单精度浮点数能够保存 23 位尾数，因此能够精确表示，其移码 $E = 20 + 127 = 147 = 10010011_2$ 。单精度浮点数表示的误差和要表示的数的大小有关。一般来说，在其可表示范围内，单精度浮点数表示越大的数，其绝对误差越大，但相对误差变化不大。

^{II} 当使用 float 数据类型保存小数时，由于其尾数最大为 23 位，而 $2^{-23} = 0.00000011920928955078125$ ，则对于小于 0.0000001 的小数部分，float 数据类型无法精确表示，因此 float 数据类型在表示小数时的精度最多为 7 位。

对于 `double` 数据类型来说, 由于其尾数为 52 位, 若保存规范浮点数, 可以表示 53 位的二进制数, 能够表示的最大整数为

$$2^{53} - 1 = 9007199254740991$$

当用来表示浮点数的整数部分时, 能够表示的有效数字不超过 16 位。根据最大尾数取对数的结果, `float` 数据类型的实际表示精度为 $\log_{10}(2^{24}-1)+1 \approx 8$ 位十进制数, `double` 数据类型为 $\log_{10}(2^{53}-1)+1 \approx 16$ 位十进制数。在解题时, 如果题目涉及浮点数的相加或相乘操作, 那么就需要注意是选择单精度还是双精度浮点数, 以免计算结果超出所能表示的精度范围。

强化练习: [10114 Loansome Car Buyer^A](#)。

1.1.3 数据类型的取值范围

对于解题常用的编程语言来说, 其提供的各种数据类型均有特定的取值范围。熟悉数据类型的取值范围非常重要, 这对解题时确定应该使用何种数据类型, 使得中间运算结果不会溢出, 从而保证最终结果的正确性起关键作用。在 C++ 中, 头文件 `<limits>` 中定义了各种数据类型的取值范围。例如, `int` 类型的取值上限为 `numeric_limits<int>::max()`, 下限为 `numeric_limits<int>::min()`。可以利用这些定义, 使用如下的代码来生成一个 C++ 源代码文件, 编译运行后可以获取各种数据类型存储时使用的字节数及其表示范围^I。

```
//-----1.1.3.cpp-----
int main(int argc, char *argv[]) {
    // 为程序生成所需的头文件。
    cout << "#include <iostream>\n"
        << "#include <iomanip>\n"
        << "#include <limits>\n"
        << "using namespace std;\n"
        << "int main(int argc, char *argv[])\n"
        << "{\n";
    // 枚举数据类型。
    vector<string> dataTypes = {
        "bool", "char", "unsigned char",
        "short int", "unsigned short int",
        "int", "unsigned int", "long int", "unsigned long int",
        "long long int", "unsigned long long int",
        "float", "double", "long double"
    };
    // 定义输出格式。
    string literal =
        "    cout << [$: ] << sizeof($) << [B, ] << #numeric_limits<$>::min()"
        " << [ ~ ] << #numeric_limits<$>::max() << endl;";
    // 为每种数据类型生成一行输出。
    for (auto t : dataTypes) {
        for (auto c : literal) {
            if (c == '[' || c == ']') cout << '\'';
            else if (c == '$') cout << t;
            else if (c == '#') {
                if (t.front() == 'b' || t.back() == 'r') cout << "(int)";
            }
        }
    }
}
```

^I 此源代码文件经过编译运行所生成的输出是另外一个源代码文件, 需要再次进行编译运行才能得到预期的结果。

```

        else cout << c;
    }
    cout << endl;
}
cout << "    return 0;\n"
    << "}\n";
return 0;
}
//-----1.1.3.cpp-----//

```

将以上代码保存为一个 C++ 源文件（例如命名为 first.cpp），使用 GCC 编译器，按照以下命令进行编译运行^I：

```

qiuqiu@qiuqiu-VirtualBox:~$ g++ -std=c++11 first.cpp -o first.exe
qiuqiu@qiuqiu-VirtualBox:~$ first.exe >second.cpp
qiuqiu@qiuqiu-VirtualBox:~$ g++ -std=c++11 second.cpp -o second.exe
qiuqiu@qiuqiu-VirtualBox:~$ second.exe >third.txt

```

最后得到的文件 third.txt 中的内容为^{II}：

```

bool: 1B, 0 ~ 1
char: 1B, -128 ~ 127
unsigned char: 1B, 0 ~ 255
short int: 2B, -32768 ~ 32767
unsigned short int: 2B, 0 ~ 65535
int: 4B, -2147483648 ~ 2147483647
unsigned int: 4B, 0 ~ 4294967295
long int: 4B, -2147483648 ~ 2147483647
unsigned long int: 4B, 0 ~ 4294967295
long long int: 8B, -9223372036854775808 ~ 9223372036854775807
unsigned long long int: 8B, 0 ~ 18446744073709551615
float: 4B, 1.17549e-38 ~ 3.40282e+38
double: 8B, 2.22507e-308 ~ 1.79769e+308
long double: 12B, 3.3621e-4932 ~ 1.18973e+4932

```

强化练习：465 Overflow^A，913 Joana and the Odd Numbers^A。

1.2 格式化输入

1.2.1 概述

在使用 C++ 进行解题时，需要将数据按照指定的格式读入之后再进行处理，这个过程称为格式化输入。在 C 中，一般使用 `scanf` 函数，指定相应的参数进行读入。而在 C++ 中，只要定义了变量的类型，可以直接使用重载的运算符“`>>`”配合 `cin` 和（或）`istringstream` 对象进行输入处理。C++ 会根据变量类型以空白字符（空格、制表符）和换行符作为默认分隔符进行数据的读入。其中 `cin` 是 `istream` 类的一个实例，可以直接使用，`istringstream` 则是继承自 `istream` 的一个类，需要实例化后才能使用。

^I 以 GCC 5.3.0 为例，假设源文件保存在用户 Home 目录。如果未将当前目录添加到 PATH 变量中，需要在运行生成的可执行文件前添加相对路径“`./`”。

^{II} 这是在 Ubuntu 14.04 i686 32 位系统上使用 GCC 5.3.0 编译然后执行得到的结果。每行输出的内容依次为：数据类型名称，数据类型占用的内存空间（单位：字节），数据类型表示值的下限，数据类型表示值的上限。

cin 包含了下列用于辅助输入的成员函数：

get

从输入流中读取一个字符。

`istream& get(char& c);`

getline

从输入流中读取一行，直到已经读取指定数量的字符或遇到特定的结束字符（默认为换行符 ‘\n’）。

`istream& getline(char* s, streamsize n);`

`istream& getline(char* s, streamsize n, char delim);`

unget

将输入流的位置计数回退一个位置，使得上一个已经读入的字符能够再次被读入。

`istream& unget();`

putback

将输入流的位置计数回退一个位置，将指定字符放入输入流的当前位置使得可以读入该字符。

`istream& putback(char c);`

ignore

忽略从输入流当前位置开始的指定个数字符，直到满足下列条件之一：已经忽略的字符个数达到指定的数值（默认为 1）；遇到指定的结束标记（默认为文件结束符）；发生输入流读错误。

`istream& ignore(streamsize n = 1, int delim = EOF);`

需要注意，如果混合使用输入重载符“>>”和 `cin.getline` 读取输入，需要在每次使用 `cin.getline` 前通过 `cin.ignore` 忽略掉换行符，否则 `cin.getline` 读取的将是上一次输入未读尽的换行符。

另外一个常用的输入处理函数是头文件`<string>`中的 `getline` 函数，它和 `cin` 中的成员函数 `getline` 名称相同，功能也类似^I。`string` 类的 `getline` 函数也有两个版本，其函数原型如下：

getline

从输入流中读入一行，直到满足下列条件之一：遇到指定的结束符；遇到文件结束符 EOF；

发生输入流读错误。

`istream& getline(istream& is, string& str, char delim);`

`istream& getline(istream& is, string& str);`

`getline` 的作用是从输入流中读入字符，将其存储到 `string` 类变量中，直到遇到指定的结束符，如果未指定结束符，则使用默认的结束符 ‘\n’（回车换行符）。如果在读入过程中遇到文件结束标记（end of file, EOF）或者发生输入流读错误，也会结束输入。

强化练习：391 Mark-Up^C。

1.2.2 标准输入

如果输入中每行给出的是整数、实数或不包含空格的字符串，则可以声明相应的变量，直接使用 `cin` 进行读入。例如，如果每行均包含两个整数，最后以文件结束符作为输入终结的标志，则可以按以下方式处理输入：

^I 实际上，`string` 类是模板类 `basic_string` 的实例化，在 SGI 的实现中，有如下的定义：“`typedef basic_string<char> string;`”。

```
int i, j;
while (cin >> i >> j) {
    // 进一步处理。
}
```

强化练习: 10055 Hashmat the Brave Warrior^A, [10071](#) Back to High School Physics^A, 10970 Big Chocolate^A, [11559](#) Event Planning^A, [12279](#) Emoogle Balance^A, [12650](#) Dangerous Dive^B, [12709](#) Falling Ants^C, [12917](#) Prop Hunt^C, [12952](#) Tri-Du^A, [12996](#) Ultimate Mango Challenge^D, [13007](#) D as in Daedalus^D。

如果输入最后以整数对“0 0”而不是以文件结束符作为输入终结标志,可以在 while 循环中增加条件进行判断。

```
int i, j;
while (cin >> i >> j, i || j) {
    // 进一步处理。
}
```

强化练习: [573](#) The Snail^A, [591](#) Box of Bricks^A, [11679](#) Sub-Prime^A, [12468](#) Zapping^A。

如果输入给定了测试数据的组数,则可以先读入组数,然后使用 for (或者 while) 循环逐组读入数据。

```
int cases;
cin >> cases;
for (int cs = 1; cs <= cases; cs++) {
    // 进一步处理。
}
while (cases--) {
    // 进一步处理。
}
```

强化练习: [10300](#) Ecological Premium^A, [10783](#) Odd Sum^A, [10812](#) Beat the Spread^A, [11172](#) Relational Operator^A, [11547](#) Automatic Answer^A, [11727](#) Cost Cutting^A, [11764](#) Jumping Mario^A, [11777](#) Automate the Grades^A, [11799](#) Horror Dash^A, [11942](#) Lumberjack Sequencing^A, [12157](#) Tariff Plan^A, [12750](#) Keep Rafa at Chelsea^C, [12798](#) Handball^B, [12854](#) Automated Checking Machine^B, [12893](#) Count It^C, [12992](#) Huatuo's Medicine^C, [13012](#) Identifying Tea^B, [13034](#) Solve Everything^B。

1.2.3 字符串输入

不包含空白字符的字符串输入

如果每行只有一个单词,可采用如下的方式进行读入:

```
string word;
while (cin >> word) {
    // 进一步处理。
}
```

如果每行输入中包含空格,而空格必须作为输入的一部分,那么可以采用:

```
string line;
// getline 读入一行输入中除行末换行符 (\n) 之外的所有字符。
while (getline(cin, line)) {
    // 进一步处理。
}
```

注意: 如果使用 `cin >> line`, 获取的是该行的第一个字符串, 而不是整行字符。假设输入是:

```
this is a fox.
```

`cin >> line` 的结果是:

```
this
```

`getline(cin, line)` 的结果是:

```
this is a fox.
```

强化练习: [1124 Celebrity Jeopardy^A](#), [10530 Guessing Game^A](#), [11687 Digits^B](#), [12289 One-Two-Three^A](#), [12545* Bits Equalizer^B](#)。

如果混合使用上述两种字符串读入方法, 在使用 `getline` 函数之前需要将输入缓冲区的换行符“读尽”, 否则会导致 `getline` 函数从 `cin` 尚未“读尽”的换行符开始读取, 最终得到一个空行, 不符合原来的预期。可以使用 `cin.ignore` 来完成“读尽”的工作。

```
string word;
while (cin >> word, word != "#") {
    // 进一步处理。
}
// 忽略行末的回车换行符。
cin.ignore(1024, '\n');
while (getline(cin, puzzle), puzzle != "#") {
    // 进一步处理。
}
```

强化练习: [895 Word Problem^B](#), [10141 Request for Proposal^A](#)。

包含空白字符的字符串输入

默认情况下, `cin` 在进行输入时会忽略空白字符和回车换行符。如果需要将这些字符读入, 则可以通过 `unsetf` 函数设置输入标志来达到目的^I。

```
-----1.2.3.1.cpp-----//
int main(int argc, char *argv[]) {
    char c;
    // 默认选项是忽略空白字符。
    while (cin >> c && c != '*') cout << c;
    // 设置读入时不跳过空白字符和回车换行符, 即将输入原样输出, 包括空白字符和回车换行符。
    cin.unsetf(ios::skipws);
    while (cin >> c && c != '*') cout << c;
    cout << '\n';
    return 0;
}
-----1.2.3.1.cpp-----//
```

对于以下输入:

```
The quick red fox jumps over a lazy dog.
```

^I 参见本章第 1.3 节“格式化输出”中关于 `unsetf` 函数功能介绍的内容。

```
The quick brown fox jumps over a lazy dog.*
```

```
The quick red fox jumps over a lazy dog.
The quick brown fox jumps over a lazy dog.*
```

其输出为：

```
Thequickredfoxjumpsoveralazydog.Thequickbrownfoxjumpsoveralazydog.
```

```
The quick red fox jumps over a lazy dog.
The quick brown fox jumps over a lazy dog.
```

强化练习：272 TEX Quotes^A, [492 Pig-Latin^A](#)。

如果需要将一行输入拆分为多个单词，而输入中的各个单词之间以空白字符（空格或制表符）分隔，可使用 `istringstream` 类进行处理。

```
-----1.2.3.2.cpp-----
int main(int argc, char *argv[]) {
    string line, word;
    line = "The quick brown fox jumps over a lazy dog";
    // 将输入以空格作为分隔符拆分成单词。
    istringstream iss(line);
    while (iss >> word) cout << word << "-";
    cout << endl;
    // 如果需要连续使用 istringstream 实例，注意将输入状态标志重置，否则会发生错误。
    // 读者可以尝试将 iss.clear() 注释掉后观察相应的输出以理解其作用。
    iss.clear();
    line.assign("The quick black dog jumps over a lazy fox");
    iss.str(line);
    while (iss >> word) cout << word << "-";
    cout << endl;
    return 0;
}
-----1.2.3.2.cpp-----
```

以上代码将指定字符串拆分为多个单词，并在每个单词后面附加一个连字符，其输出为：

```
The-quick-brown-fox-jumps-over-a-lazy-dog-
The-quick-black-dog-jumps-over-a-lazy-fox-
```

强化练习：[1593 Alignment of Code^B](#), [12243 Flowers Flourish from France^B](#), [13093 Acronyms^D](#)。

以特定字符作为分隔符

如果输入为一行字符串，但是不以空白字符作为分隔符（例如逗号、分号等），在 C 中，处理思路可能与以下代码类似。

```
-----1.2.3.3.cpp-----
// 寻找分隔符，获取分隔符之间的字符串。
void parse(string line) {
    size_t start = 0, next = line.find(',', start);
    while (next != linenpos) {
        string block = line.substr(start, next - start);
        cout << block << endl;
        start = next + 1;
        next = line.find(',', start);
```

```
    }
    if (start < line.length()) cout << line.substr(start) << endl;
}
```

使用上述方式对字符串进行拆分，需要注意拆分结束的条件。如果分隔符不是字符串的末尾字符，会将最后一个分隔符到字符串末尾之间的字符作为一个“分组”输出。也就是说，此种情况下会将字符串末尾视为一个“隐形”的分隔符。

强化练习: 277 Cabinets^E, 450 Little Black Book^A, 576 Haiku Review^A, 12195 Jingle Composing^B。

以特定字符串或字符作为结束标志

若输入的最后一行以一个特定的字符串或字符作为结束标志，例如以符号‘#’作为结束行的标记，则可以采用如下的读取方法。

```
string line;
while (getline(cin, line), line.length() > 0 && line[0] != '#') {
    // 进一步处理。
}
```

需要注意，某些情况下不能只采用判断条件：`line != "#"`，因为最后一行以 '#' 开始，并不表示这一行就只有一个 '#' 字符，有可能最后一行为“#this is a empty line”，题目的输入可能在此设置陷阱。

强化练习: 12250 Language Detection^A, 12403 Save Setu^A, 12577 Hajj-e-Akbar^A。

1.3 格式化输出

1.3.1 概述

在 C++ 中，一般使用标准类库提供的 `cout` 进行输出操作。`cout` 是 `ostream` 的一个实例，认为标准输出，拥有 `ios_base` 基类的全部函数和成员数据。`cout` 提供了两个常用的格式化操作：`setf` 函数和 `unsetf` 函数，用以在当前的格式状态下追加或删除指定的格式。

格式参数

以下列出了在使用 `setf` 函数和 `unsetf` 函数时可用的格式参数。

<code>ios::dec</code>	在读入和输出整数时使用十进制格式。
<code>ios::hex</code>	在读入和输出整数时使用十六进制格式。
<code>ios::oct</code>	在读入和输出整数时使用八进制格式。
<code>ios::showbase</code>	在输出整数时添加一个表示其进制的前缀，八进制前加 <code>0</code> ，十进制原样输出，十六进制前加 <code>0x</code> 。
<code>ios::internal</code>	两端对齐。当输出长度不足时，在符号位和数值的中间插入填充字符使得输出的两端对齐。

<code>ios::left</code>	左对齐。当输出宽度不足时，在输出的末端插入填充字符以使得输出左对齐。
<code>ios::right</code>	右对齐。当输出长度不足时，在输出的前端插入填充字符以使得输出右对齐。
<code>ios::fixed</code>	使用定点小数方式输出浮点数。
<code>ios::scientific</code>	使用科学计数法方式输出浮点数。
<code>ios::boolalpha</code>	输出布尔值时，若布尔值为 1，则输出 ‘ <code>true</code> ’，否则输出 ‘ <code>false</code> ’。
<code>ios::showpoint</code>	在输出浮点数时强制显示小数点。
<code>ios::showpos</code>	在输出非负数值时，在数值前附加符号 ‘ <code>+</code> ’。
<code>ios::skipws</code>	在进行读入时，忽略输入流中的前导空白字符，直到遇到一个非空白字符。
<code>ios::unitbuf</code>	在每次输出操作后清空缓存。
<code>ios::uppercase</code>	输出十六进制数时表示数位的字母强制大写。

使用的方法是将其作为参数调用 `setf` 或 `unsetf` 函数。例如，如果输出十六进制数时，要求字母字符以大写形式显示，可使用：`cout.setf(ios::uppercase)`，那么在输出时，小写字母 `a-f` 将被转换为大写字母 `A-F` 进行输出。需要注意的是，`ios::uppercase` 仅使输出流相关产生的小写字母转换为大写字母输出，不对非输出流相关产生的输出内容起作用，仅用于输出十六进制数或数制前缀の場合。对于字符串变量，欲通过使用 `ios::uppercase` 参数将小写字母自动转换为大写字母进行输出，将无法达到预期目的。

```
//-----1.3.1.cpp-----//
int main(int argc, char *argv[]) {
    string line = "the quick brown fox jumps over the lazy dog.";
    cout.setf(ios::uppercase);
    cout << line << endl;
    int number = 0x1af;
    cout.setf(ios::hex, ios::basefield);
    cout.setf(ios::showbase);
    cout << number << endl;
    return 0;
}
//-----1.3.1.cpp-----//
```

输出为：

```
the quick brown fox jumps over the lazy dog.
0X1AF
```

格式参数的连接

可以使用“或”(|)运算符对多个格式参数进行连接，以达到一次性设置多个格式的目的。例如，在输出十六进制数时，需要显示进制提示并要求其大写，则可以使用：

```
cout.setf(ios::showbase | ios::uppercase);
```

为了更为方便地应用，标准库在定义时已经将某些参数进行了分组，即将类似的格式参数使用“或”位运算事先合并在一起，称之为域（field）。

<code>ios::basefield = ios::dec ios::oct ios::hex</code>	// 基数域
<code>ios::adjustfield = ios::left ios::right ios::internal</code>	// 对齐域
<code>ios::floatfield = ios::fixed ios::scientific</code>	// 浮点数域

通过域来进行设置，可以一次清除多个格式位。例如要将基数位复位，并将输出整数的基数调整为十六进制，可以使用如下语句：

```
cout.setf(ios::hex, ios::basefield);
```

需要注意的是, 如果基数已经设置为其他值, 使用“或”位运算将格式标记 `ios:: dec`、`ios::oct`、`ios:: hex` 和其他格式标志同时使用, 不会产生预期的效果。例如, 当前需要以十六进制输出整数, 且将数位字母大写, 拟使用如下语句:

```
cout.setf(ios::showbase | ios::uppercase | ios::hex);
```

如果之前的基数为十进制, 使用该语句后输出仍然为十进制, 无法达到预期效果, 为了能够使输出更改为十六进制, 可以使用:

```
cout.setf(ios::showbase | ios::uppercase);
cout.setf(ios::hex, ios::basefield);
```

格式控制内联函数

为了设置格式的方便, 标准类库中还定义了一系列的内联函数, 这些内联函数和相应的格式参数名称相似, 效果相同。以下列举了部分内联函数以及相对应的 `setf/unsetf` 函数调用。

<code>boolalpha/noboolalpha</code>	<code><=> cout.setf/unsetf (ios::boolalpha)</code>
<code>dec</code>	<code><=> cout.setf(ios::dec)</code>
<code>fixed</code>	<code><=> cout.setf(ios::fixed)</code>
<code>hex</code>	<code><=> cout.setf(ios::hex)</code>
<code>internal</code>	<code><=> cout.setf(ios::internal)</code>
<code>left</code>	<code><=> cout.setf(ios::left)</code>
<code>oct</code>	<code><=> cout.setf(ios::oct)</code>
<code>right</code>	<code><=> cout.setf(ios::right)</code>
<code>scientific</code>	<code><=> cout.setf(ios::scientific)</code>
<code>showbase/noshowbase</code>	<code><=> cout.setf/unsetf(ios::showbase)</code>
<code>showpoint/noshowpoint</code>	<code><=> cout.setf/unsetf(ios::showpoint)</code>
<code>showpos/noshowpos</code>	<code><=> cout.setf/unsetf(ios::showpos)</code>
<code>skipws/noskipws</code>	<code><=> cout.setf/unsetf(ios::skipws)</code>
<code>uppercase/nouppercase</code>	<code><=> cout.setf/unsetf(ios::uppercase)</code>

使用这些格式控制内联函数的方式很简单, 直接在输出重载操作符后指定即可。例如, 需要将整数输出更改为十六进制且左对齐输出, 可以使用:

```
cout << hex << setw(8) << left << 1024 << endl;
```

输入输出操纵器

若需要进行更多的输出控制, 可以包含头文件`<iomanip>`, 从而使用其中的输入输出操纵器。在输入输出操纵器中, `io` 是 `input and output` 的缩写, 表示输入输出; `manip` 是 `manipulator` 的缩写, 表示操纵器 (`manipulator`)。以下列出了若干常用的操纵器。

<code>setiosflags</code>	设置输出标记。在头文件 <code><iostream></code> 中定义了多个标记, 这些标记的组合可以控制输出的格式, 例如 <code>setiosflags(showbase uppercase)</code> 表示在输出整数时显示基数且基数以大写方式显示。
--------------------------	---

<code>setbase</code>	设置输出的基数, 其中 <code>dec</code> 为十进制, <code>hex</code> 为十六进制, <code>oct</code> 为八进制。例如 <code>setbase(hex)</code> 表示将基数设置为十六进制。
----------------------	---

<code>setfill</code>	设置填充字符。例如 <code>setfill('#')</code> 表示当输出宽度不足时以字符 ‘#’
----------------------	---

进行填充。

setprecision 设置输出精度。例如 `setprecision(6)` 表示将输出的数值四舍五入到小数点后 6 位，如果原有数字小数点后不足 6 位则补 0。

setw 设置输出宽度。例如 `setw(10)` 表示将输出宽度设置为 10 个字符宽度。

操纵器可以在输出时连续使用。例如，以下代码片段将实数 3.1415926 按照特定格式予以输出。

```
// 输出宽度为 6, 右对齐, 宽度不足时以字符 '#' 填充, 保留小数点后两位并显示小数点。
cout << setw(6) << right << setfill('#') << setprecision(2) << fixed << 3.1415926;
```

最终输出为：

##3.14

1.3.2 输出对齐

`cout` 提供了三种方式调整输出对齐。

left: 若输出内容宽度小于设置的输出宽度，以指定填充字符在输出的右侧进行填充，直到达到输出宽度，效果相当于将输出内容左对齐 (left aligned)。

right: 与 **left** 的作用相反。若输出内容宽度小于设置的输出宽度，以指定的填充字符在输出的左侧进行填充，直到达到输出宽度，效果相当于将输出内容右对齐 (right aligned)。

internal: 若输出内容的宽度小于输出宽度，在输出内容内部的特定位置插入填充符，直到达到输出宽度。效果类似于 Microsoft Office Word 排版中的两端对齐 (justified)。对于数值输出来说，**internal** 的效果是在正负符号和数值之间插入填充字符，对于非数值变量的输出，其效果等同于使用 **right**。

```
-----1.3.2.cpp-----
int main(int argc, char *argv[]) {
    string line = "the quick brown fox jumps over the lazy dog.";
    cout << setfill('#') << setw(60) << left << line << endl;
    cout << setw(60) << right << line << endl;
    cout << setw(60) << internal << line << endl;
    int number = -1234567890;
    cout << setw(30) << left << number << endl;
    cout << setw(30) << internal << number << endl;
    cout << setw(30) << right << number << endl;
    return 0;
}
-----1.3.2.cpp-----
```

输出为：

```
the quick brown fox jumps over the lazy dog.#####
#####the quick brown fox jumps over the lazy dog.
#####the quick brown fox jumps over the lazy dog.
-1234567890#####
-1234567890
##### -1234567890
```

强化练习：[706 LC-Display^A](#)。

1.3.3 整数输出

在输出整数时，主要需要控制的是显示整数时所用的基数。可以使用内联函数 `showbase`、`dec`、`oct`、`hex`、`uppercase` 或者 `setf` 函数对整数输出格式进行调整。

```
//-----1.3.3.cpp-----
int main(int argc, char *argv[]) {
    int n = 60;
    cout.setf(ios::showbase);
    cout.setf(ios::dec, ios::basefield);
    cout << "dec: " << n << endl;
    cout << oct << "oct: " << n << endl;
    cout << hex << "hex: " << n << endl;
    cout.setf(ios::uppercase);
    cout << "hex | uppercase: " << n << endl;
    return 0;
}
//-----1.3.3.cpp-----
```

输出为：

```
dec: 60
oct: 074
hex: 0x3c
hex | uppercase: 0X3C
```

强化练习：10550 Combination Lock^A, [11044](#) Searching for Nessy^A, 11956 Brainfuck^B, 12342 Tax Calculator^B。

1.3.4 实数输出

在输出实数时，主要关注的是输出的精度以及小数的形式——是以定点小数（fixed point number）还是以科学计数法（scientific notation）输出。

```
//-----1.3.4.1.cpp-----
int main(int argc, char *argv[]) {
    double number = 123456789.0123456789;
    cout << number << endl;
    cout << fixed << setprecision(4) << number << endl;
    cout << setprecision(8) << number << endl;
    cout << scientific << number << endl;
    return 0;
}
//-----1.3.4.1.cpp-----
```

输出为：

```
1.23457e+08
123456789.0123
123456789.01234567
1.23456789e+08
```

在实数的输出中，经常要做的是对结果进行四舍五入（rounding）——保留指定位数的小数进行输出。一般结合 `fixed` 和 `setprecision` 操纵器对输出精度进行控制。传入操纵器 `setprecision` 的整数决定保留的小数点位数，`fixed` 的作用是指明以定点小数形式输出实数。有时因为浮点数表示精度的问题，

`setprecision` 的四舍五入结果与预期的结果有差异，此时可以将结果加上一个很小的常数（比如， 10^{-7} ）后再进行输出，可以达到预期的效果。如下例所示，由于无法通过浮点数精确表示 1.005，导致四舍五入的结果与预期有差异，而加上一个很小的常数后可以得到预期结果。

```
-----1.3.4.2.cpp-----
const double EPSILON = 1e-7;
int main(int argc, char *argv[]) {
    double number = 1.005;
    cout << fixed << setprecision(2) << number << endl;
    cout << fixed << setprecision(2) << (number + EPSILON) << endl;
    return 0;
}
-----1.3.4.2.cpp-----//
```

输出为：

```
1.00
1.01
```

如果输出部分的所有实数均需要按要求进行四舍五入，那么可以先统一设定输出格式后再予以输出。

```
-----1.3.4.3.cpp-----
int main(int argc, char *argv[]) {
    vector<double> datas = {1.111, 2.222, 3.333, 4.444, 5.555, 6.666};
    cout.setf(ios::fixed);
    cout.precision(2);
    for (int i = 0; i < datas.size(); i++) {
        if (i > 1) cout << ' ';
        cout << datas[i];
    }
    cout << endl;
    return 0;
}
-----1.3.4.3.cpp-----//
```

输出为：

```
1.11 2.22 3.33 4.44 5.55 6.67
```

在进行与角度相关的输出时，如果要求输出的角度值必须位于左闭右开区间[0, 360.0)中，且要求对输出进行四舍五入处理，则角度值 359.9956 按照精确到小数点后两位输出将为 360.00，最终应该输出 0.00。使用常规的方式不便处理此种情况，可结合 `stringstream` 类进行处理。

```
-----1.3.4.4.cpp-----
string roundAngle(double angle) {
    stringstream ss;
    string roundedAngle;
    ss << fixed << setprecision(2) << (angle + 1e-7);
    ss >> roundedAngle;
    if (roundedAngle == "360.00") roundedAngle = "0.00";
    return roundedAngle;
}
int main(int argc, char *argv[]) {
    double angle1 = 355.8762, angle2 = 359.9985;
    cout << roundAngle(angle1) << endl;
    cout << roundAngle(angle2) << endl;
}
```

```

        return 0;
}
//-----1.3.4.4.cpp-----//

```

输出为：

```

355.88
0.00

```

10137 The Trip^A (旅行)

有一群学生去旅行，在旅行途中进行消费时，有的学生会先垫付一部分钱，在旅行结束后，学生们根据预先垫付的情况相互之间还钱。给出学生的支付清单，要求你计算一个最小总“交易”金额，使得学生能够平摊所有费用，而且每个人的支出差距在 1 分钱以内。

输入

输入包含若干组数据。每组数据的第一行为一个正整数 n ，表示此次旅行中的学生人数。接下来的 n 行每行包含一个学生的支出，精确到分。学生人数不超过 1000，而且每个学生的支出不超过 100000 美元，输入以只包含 0 的一行结束。

输出

对于每组测试数据输出一行，包含一个数值，表示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

```

4
15.00
15.01
3.00
3.01
0

```

样例输出

```

$11.99

```

分析

此题关键在于对“平均费用”的理解。“相差一分钱”的含义是一一各个成员所交钱的数量彼此相差在一分钱以内，而不是指所有钱之和与最后支出之和相差一分钱，也不是指相邻两个成员之间所交的钱相差在一分钱以内。例如，有 3 个人，交的钱分别为（单位为美元）：10.01，10.02，10.01，则可以称其为“相差一分钱”，但是如果交的钱为：10.01，10.02，10.03，虽然前后相差在一分钱以内，但是第三个和第一个相差为两分钱，不符合题意。

令 s 为总的花费（将每位学生的花费转换为美分，表示为整数以方便讨论）， t 为总人数， a 为平均费用， r 为余数，则有

$$a = \left\lfloor \frac{s}{t} \right\rfloor$$

$$s = a \times t + r, r < t$$

将 t 个学生按花费进行分组，比平均数 a 大的人为 X 组，共有 x 个人，小于等于平均数 a 的人为 Y 组，共有 y 个人。交换的钱就是从 Y 组收取，还给 X 组。欲使得互相交换的钱尽可能地少， Y 组就要尽量少还钱给 X 组。

现在，根据余数 r 的情况分别进行讨论。（1）如果余数 r 为 0，则以花费的平均数 a 为基准向花费少的

Y组收取的钱刚好可以归还给X组的人，不多不少，大家的花费均为 a 。如果向Y组的某个人多收1分钱，则归还时，X组的某个人可以少花费1分钱，则Y组有一个人花费为 $a+1$ 分钱，X组有一个人花费为 $a-1$ 分钱，最终导致相差钱数不在1分钱以内，不满足题意。（2）当 $0 < r \leq x$ 时，表明按平均花费 a 向花费少的Y组收取的钱尚不够偿还X组的人多花的钱，还差 r 分钱，为了使得交易的钱数最少，在向多花费的X组退钱时，每个人可以少退一分钱，这样这些少退钱的人所花的钱为 $a+1$ 分钱，剩余其他人花的钱均为 a 分钱，满足题意。（3）当 $r > x$ 时，即使X组的人每个人都少退一分钱，钱数仍然有“缺口”，少的钱数为 $r-x$ 分钱，少的钱只能向Y组的人收取，每人比平均花费 a 多收一分钱，直到填满“缺口”，由于 $r < t = x + y$ ，则有 $r - x < y$ ，即最多向Y组中的 $y-1$ 个人再收取一分钱，就能填满“缺口”。这样总共有 r 个人的花费为 $a+1$ 分钱， $t-r$ 个人花费为 a 分钱。

尽管上述各种情况的讨论看起来似乎比较复杂，但最终实现为代码却很简单。

关键代码

```
// 数组 money 存放每个学生的花费（单位：美分），n 为学生的总数。
int findChange(int *money, int n) {
    long long int sum = 0, average = 0, remain = 0, debt = 0;
    for (int i = 0; i < n; i++) sum += money[i];
    average = sum / n, remain = sum % n;
    for (int i = 0; i < n; i++)
        if (money[i] > average) {
            debt += (money[i] - average);
            if (remain-- > 0) debt--;
        }
    return debt;
}
```

强化练习：[570 Stats^D](#), [10281 Average Speed^A](#), [10370 Above Average^A](#), [11945 Financial Management^C](#), [11984 A Change in Thermal Unit^A](#), [12725 Fat and Orial^D](#)。

1.3.5 缓冲区与输入输出同步

`endl`

`endl` (end of line) 是一个输出控制符号，表示将换行符放入输出流并立即将输出流缓冲区内的内容输出，其本身是一个函数模板。`cout` 是以流的形式操纵输出，一般情况下，输出一个字符实际上是将此字符放入输出缓冲区内，不会将其立即输出到屏幕或文件中，而是在某个不确定的时机将缓冲区的内容输出到相应输出。但是使用 `endl` 后，`cout` 会首先将一个换行符放入缓冲区内，并立即将缓冲区的内容输出，然后清空缓冲区。对于存在大量输出操作的程序来说，建议在最后才使用 `endl`，而不是每当有输出产生时就使用 `endl`，否则调用的代价较高，会导致程序运行时间延长。更为合理的方法是在需要换行的地方使用转义符“\n”，待程序运行结束时一并输出。

tie 与 sync_with_stdio

C++以流的方式处理输入输出，默认情况下，`cin` 和 `cout` 绑定，即在进行任何输入操作之前，保证刷新输出缓冲区。为了提高输入速度，避免每次输入时都刷新缓冲区，可以将输入输出解绑定，该操作可通过 `tie` 函数来实现。

对于输入流来说，`tie` 有两种应用形式：

```
// 不带参数，返回与当前 cin 绑定的输出对象的指针。
```

```

cin.tie();
// 带参数, 将当前输入与指定的输出对象绑定。
cin.tie(ostream* out);

```

如果将 0 或 NULL 作为参数传入 tie 函数, 则会将输入与输出解绑定。注意, 在解绑定后, 不保证在进行输入操作前刷新 cout 的缓冲区, 也不保证不刷新 cout 的缓冲区。一般计算机在可用资源允许的情况下会刷新 cout 的缓冲区, 实际情况是绝大部分时候都会刷新 cout 的缓冲区。

为了保持与 C 输入输出函数 scanf 及 printf 的兼容性, C++ 提供了 sync_with_stdio 函数。默认 C++ 的 cin 和 cout 与 C 的 scanf 和 printf 之间保持同步, 可以混合使用, 但是这样做有一个缺点, 当输入中有大量数据时, C++ 使用 cin 和 cout 进行输入输出会比 C 使用 scanf 和 printf 慢, 其真正原因是 C++ 为了同步输入输出而牺牲了效率, 并不是 C++ 底层的执行效率比 C 低。可以使用 sync_with_stdio(false) 来关闭“同步”这个特性。关闭“同步”特性后, 当进行大数据量的输入输出时, 使用 cin 和 cout 进行输入和输出的效率与 C 使用 scanf 和 printf 进行输入和输出的效率相差不大。注意 sync_with_stdio(false) 需要在进行任何输入输出操作之前调用, 使用该语句后, 不能再将 cin、cout 与 scanf、printf 进行混用, 否则会发生输入输出混乱的情况。sync_with_stdio 是一个静态函数, 调用后对所有的输入输出流对象 (如 cin、cout、cerr、clog 等) 均产生影响。具体使用时可以参考如下主函数代码框架:

```

int main(int argc, char *argv[]) {
    // 需要在进行任何输入输出前使用。
    cin.tie(0); cout.tie(0); ios::sync_with_stdio(false);
    // 或者使用:
    // cin.tie(NULL); cout.tie(NULL); ios_base::sync_with_stdio(false);
    // 后续不能将 scanf, printf 与 cin, cout 混和使用。输出换行符时, 避免使用 endl,
    // 否则会导致每次输出时刷新缓冲区, 增加时间消耗。
    // 后续代码...
    return 0;
}

```

在解题时, 如果输入或输出数据量非常大, 对程序的运行时间具有显著的影响, 可以考虑使用 tie 和 sync_with_stdio, 但应该首先检查算法的效率是否符合要求。因为在绝大多数情况下, 使用了正确而且高效的算法, 运行时间应该都会在时间限制内。如果是想在 UVa OJ 的解题时间排行榜占有一席之地, 推荐使用 cin.tie(0) 和 cout.tie(0) 及 ios::sync_with_stdio(false)。

如果需要更为快速地读取输入, 可以使用文件读函数 fread, 结合缓冲区、静态变量等技巧以获得更高的读取效率。以下给出读取字符和整数的快速输入实现^{[12][13]}, 读者可以根据类似思路实现其他数据类型的快速读取输入模块。

```

//-----1.3.5.cpp-----//
// 输入缓冲区长度。
const int LENGTH = (1 << 20);
// 从输入中读取一个字符。
inline int nextChar() {
    static char buffer[LENGTH], *p = buffer, *end = buffer;
    // 当前缓冲区已经读尽, 需要从输入中再读取一块数据到缓冲区。
    if (p == end) {
        // fread 的返回值为成功读取的字节数。
        if ((end = buffer + fread(buffer, 1, LENGTH, stdin)) == buffer) return EOF;
        p = buffer;
    }
    return *p;
}

```

```

}

// 未读尽，直接取当前一个字符返回。
return *p++;
}

// 从输入中读入一个整数。
inline bool nextInt(int &x) {
    static char negative = 0, c = nextChar();
    negative = 0, x = 0;
    // 读入字符直到遇到数字字符或负号。
    while ((c < '0' || c > '9') && c != '-') {
        if (c == EOF) return false;
        c = nextChar();
    }
    if (c == '-') { negative = 1; c = nextChar(); }
    // 继续读入数字字符，使用位运算代替乘法运算。
    do x = (x << 3) + (x << 1) + c - '0'; while ((c = nextChar()) >= '0');
    // 考虑数值的符号。
    if (negative) x = -x;
    return true;
}
//-----1.3.5.cpp-----//

```

需要注意，在使用上述快速读取方式进行读取时，不能再混合使用 `cin` 或 `scanf` 进行输入，否则会导致缓冲区状态不一致，使得最终获取的输入结果产生混乱。

强化练习：[11717 Energy Saving Microcontroller^D](#), [12356 Army Buddies^A](#), [12608 Garbage Collection^D](#)。

1.4 小结

本章旨在帮助读者复习 C++ 语言的一些基本知识，熟练掌握这些基本知识对编程竞赛来说是一项基本功。例如，熟悉各种数据类型的取值范围，在阅读题目时就能迅速确定应该使用何种数据类型，使得计算结果不会产生溢出错误。

一般来说，在编程竞赛中都是以文本文件的形式提供输入，在输出时需要按照指定的格式（例如，输出对齐、小数点后保留指定的位数等）进行输出。如果输入解析错误，会导致后续的计算环节产生错误。如果输出不符合要求，即使你的解题思路和算法实现都是正确的，也不会通过评测。因此熟悉你使用语言的全部输入输出处理功能非常重要。要想快速掌握一种语言的输入输出，最好的方法就是先阅读相应语言的参考手册，在了解输入输出方面该语言提供了哪些函数接口之后，按照语言参考手册的指示逐个上机进行试验，以了解每种函数接口的功能和意义，只有多加练习才能熟能生巧。

在 C 语言中，我们通常会使用 `scanf` 和 `printf` 来对数据进行输入输出操作。在 C++ 语言中，C 语言的这一套输入输出库我们仍然能使用，但是 C++ 又增加了一套新的、更容易使用的输入输出库。C++ 中的输入与输出可以看做是一连串的数据流，输入即可视为从文件或键盘中输入程序中的一串数据流，而输出则可以视为从程序中输出一连串的数据流到显示屏或文件中。C++ 的输入和输出使用了运算符重载，使用 `cin` 进行输入时需要紧跟“`>>`”运算符，使用 `cout` 进行输出时需要紧跟“`<<`”运算符，这两个运算符可以自行分析所处理的数据类型，因此无需像使用 `scanf` 和 `printf` 那样给出格式控制字符串。`cout`、`cin` 的用法非常强大，它们比 C 语言中的 `scanf`、`printf` 更加灵活易用。

在各类编程竞赛中，常常出现输入数据量非常大的情形，如果不采用效率较高的输入方法，在读入数据这一步就可能耗费较多的时间，从而使得程序的运行时间较长，与其他使用相同算法但采用较快输入方法的解题方案相比，在运行效率上就会落后，因此掌握快速输入技巧对建立比赛的用时优势有一定的作用。

第 2 章 数据结构

工欲善其事，必先利其器。
——《论语·卫灵公》

C++提供了丰富的数据结构来提高编程效率。为了更快地解题，需要熟悉各种数据结构所擅长处理的问题类型。熟练掌握数据结构的应用并没有特别的捷径可走，只有通过反复地查阅文档和不断地练习来了解它们的使用方法和特性。在解题中常用的基本数据结构有内置数组、向量、栈、队列、映射、集合等，以下逐一介绍了它们常用的属性和方法。除此此外，本章还介绍了若干应用较多的高级数据结构，例如，线段树、区间树、树状数组、稀疏表、并查集、二叉堆（binary heap）、左偏树（leftist tree）、树堆（treap）、伸展树（splay tree）、动态树（dynamic tree）、块状链表、树链剖分等。

2.1 内置数组

C++提供了内置数组（built-in array），它是静态的，一旦声明后其大小将不能改变。当数据数量有固定上限，且不需要对数据进行大量地增加或删除操作时，使用内置数组较为适宜。

2.1.1 顺序记录

在解题应用中，常见的是一维、二维、三维数组。一维数组主要用来表示构成序列的一组元素。在内部表示中，一维数组一般声明为一块连续的内存区域，下标相邻的元素在内存区域中是相邻的。二维数组有时也可使用一维数组进行替代以便于解题。使用一维数组表示二维数组时，令二维数组 A 为 m 行 n 列，当从 0 开始对下标计数时，二维数组的元素 $A[i][j]$ 对应的一维数组形式的元素为 $A[i * n + j]$ 。反之，一维数组形式的元素 $A[x]$ 对应的二维数组元素为 $A[x / n][x \% n]$ 。二维数组主要用于表示类似于网格的数据结构，如游戏棋盘。三维数组一般用于表示立方体网格结构形式的数据。

457 Linear Cellular Automata^A（线性细胞自动机）

某个生物学家正在使用 DNA^I修饰技术对菌群的生长现象进行实验，实验在排成一列的若干个培养皿中进行。通过改变细菌的 DNA，他能够对细菌进行“编程”操作，使得细菌对相邻培养皿的菌群密度产生响应。每个培养皿的菌群密度使用四个等级进行描述（评分从 0 到 3），而 DNA 的信息则表示成一个数组 dna ，编号从 0 到 9，菌群密度按以下规则进行计算：

(1) 对于任意给定的培养皿，令其左侧的培养皿、自身、右侧的培养皿三者的菌群密度之和为 K ，则一天之后，给定的培养皿的菌群密度为 $dna[K]$ ；

(2) 位于最左侧的培养皿，设定其有一个假想的位于左侧的培养皿，其菌群密度为 0；

(3) 对于最右侧的培养皿，设定其有一个假想的位于右侧的培养皿，其菌群密度为 0；

根据上述规则，显然，某些初始的 DNA 程序将导致所有细菌死亡（例如，[0, 0, 0, 0, 0, 0, 0, 0,

^I DNA (deoxyribonucleic acid)，脱氧核糖核酸，动植物的细胞中带有基因信息的化学物质。

0, 0]), 而某些则可能立即导致密度爆炸 (例如, [3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3])。生物学家感兴趣的是那些演化趋势看起来不那么明显的 DNA 程序是如何演化的。

编写程序, 模拟排成一列共 40 个培养皿中的菌群生长情况。你可以假定: 起始时第 20 个培养皿的菌群密度为 1, 其他培养皿的菌群密度均为 0。

输入

输入的第一行是一个正整数, 表示测试数据的组数, 后面跟着一个空行。每组测试数据一行, 包含 10 个整数, 表示 DNA 程序, 两组测试数据之间有一个空行。

输出

对于每组测试数据, 按照后续指定的格式进行输出。相邻两组测试数据的输出之间包含一个空行。每组测试数据的输出由 50 行输出构成, 每行包含 40 个字符, 每个培养皿由该行中的一个字符予以表示。如果培养皿菌群密度为 0 则输出 ‘ ’ (空格), 菌群密度为 1 则输出 ‘.’ (点), 菌群密度为 2 则输出 ‘x’ (小写字母 x), 菌群密度为 3 则输出 ‘W’ (大写字母 W)。

注意: 在样例输出中, 只给出了样例输入所对应输出的前 10 行 (总共应该是 50 行输出), 为了输出的可读性, 使用小写字母 ‘b’ 来表示空格, 但在实际输出中, 需要使用真正的空格字符而不是小写字母 ‘b’。

样例输入

```
1
0 1 2 0 1 3 3 2 3 0
```

样例输出

```
bbbbbbbbbbbbbbbbbbbb.bbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb..bbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb.xbx.bbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbb.bb.bb.bbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbb.....bbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbb.xbxbbbbbbx.bbbbbbbbbbb
bbbbbbbbbbbbbbbbb.bbxbbbbbbxb.bbbbbbbbbbb
bbbbbbbbbbbbbb...xxxbbx...bbbbbbbbbb
bbbbbbbbbbbbb...xb.WW.xbx.WW.bx.bbbbbbb
bbbbbbbbbbbbb.bbb.xxWb.bWxx.bbb.bbbbbbb
```

分析

由于培养皿一天后的菌群密度与自身及左右两个相邻的培养皿的菌群密度有关, 因此不能在原数组上直接进行“写”操作, 这样会使得计算所需的菌群密度值被“覆盖”, 导致后续计算无法进行, 应该设置一个“影子”数组, 该数组为当前数组元素的一个副本, 从“影子”数组中获取值, 将计算值写入原数组, 在下次计算前再将原数组的结果复制到“影子”数组中。

强化练习: 447 Population Explosion^C, 482 Permutation Arrays^A, 541 Error Correction^A, 626 Ecosystem^B, 703 Triple Ties: The Organizer’s Nightmare^C, 815 Flooded^B, 946 A Pile of Boxes^D, 1260 Sales^A, 1315 Crazy Tea Party^D, 10038 Jolly Jumper^A, 10260 Soundex^A, 10730 Antiarithmetic^B, 11222 Only I Did It^B, 11461 Square Numbers^A, 11511 Frieze Patterns^{[14][15]}, 11608 No Problem^A, 11760 Brother Arif Please Feed Us^C, 12150 Pole Position^C, 12485 Perfect Choir^D, 12583 Memory Overflow^B, 12658 Character Recognition^C, 12662 Good Teacher^C, 12667 Last Blood^C, 12959 Strategy Game^D, 13130 Cacho^C。

扩展练习: 199 Partial Differential Equations^D, 244 Train Time^E, 279* Spin^F, 903 Spiral of Numbers^D。

在一维数组应用中, 经常需要将其表示成“环状数组”, 即将一维数组首尾相连, 当枚举到数组的最末一个元素时, 再枚举下一个元素将回到一维数组的第一个元素, 类似于“咬尾蛇”。使用一个指示当前位置的序号, 结合偏移值, 应用简单的模运算即可得到下一个位置的序号。

```

// 将一维数组模拟为环状数组使用。
int number[256], n = 256, idx, offset;

// 从环状数组第一个位置往后移动 20 个元素位置。
idx = 0, offset = 20;
idx = (idx + offset) % n;

// 从环状数组第一个位置往前移动 50 个元素位置。
idx = 0, offset = 50;
idx = (idx - offset + n) % n;

```

强化练习: [10978 Let's Play Magic^C](#), [11093 Just Finish It Up^B](#), [11496 Musical Loop^B](#)。

2.1.2 游戏模拟

在有关二维数组的题目中,一个常见的主题是游戏的模拟。一维的游戏一般都较为简单,可玩性不高,在现实生活中并不常见。三维的游戏又不便于制作成实物在现实中展示,一般在计算机游戏中多见,因此二维的游戏是人们最为常见的游戏形式,例如国际象棋、数独、拼图等。表示二维游戏最为简便的方式是二维数组,数组的每个元素对应着游戏中的一个方格,在游戏进行过程中,对方格中的物体移动或数量的增减可以直接映射到二维数组中对应元素位置的变化或数量上的更改。

10363 Tic Tac Toe^A (井字游戏)

Tic Tac Toe 是孩子们在 3×3 的网格上进行的游戏。两个玩家轮流在棋盘上放置 ‘X’ 和 ‘O’, 执 ‘X’ 棋子为先手。如果某个玩家使用同样的棋子将棋盘上任意一条横线、竖线、斜线填充则取胜。使用 9 个点表示初始的棋盘状态,当玩家轮流在棋盘上放置棋子后,使用下列的方式来表示棋盘的状态,直到某个玩家获胜:

```

... \X.. \X.O \X.O \X.O \X.O \X.O \X.O !
... \.... \.... \.O. \.O. \OO. \OO. !
... \.... \.... \..X\ ..X\ X.X\ X.X\ XXX!

```

给定一个棋盘状态,确实该棋盘状态是否是合法的,即确定是否能够通过正常的游戏过程生成这个棋盘状态。

输入

输入第一行包含一个整数 N ,表示测试数据的组数。接着是 $4N-1$ 行测试数据,指定了由空行分开的 N 个棋盘状态。

输出

对于每组测试数据,输出 ‘yes’ 或者 ‘no’,表示给定的棋盘状态是否可以通过正常的游戏过程获得。

样例输入

```

2
X.O
OO.
XXX

```

```

O.X
XX.
OOO

```

样例输出

```

yes
no

```

分析

根据题意, 可以得到以下约束条件:

- (1) 因为执 ‘X’ 棋子的玩家为先手, 那么最终某个玩家获胜时, ‘X’ 棋子和 ‘O’ 棋子要么一样多, 要么 ‘X’ 棋子比 ‘O’ 棋子多一枚。
- (2) 棋盘最多能放置 9 枚棋子且执 ‘X’ 棋子的玩家为先手, 则 ‘X’ 棋子最多为 5 枚, ‘O’ 棋子最多为 4 枚。
- (3) 执 ‘X’ 棋子的玩家和执 ‘O’ 棋子的玩家不能同时获胜。
- (4) 因为有先后手的约束, 当 ‘X’ 棋子和 ‘O’ 棋子数量相同时, 执 ‘X’ 棋子的玩家应该尚未获胜; 类似的, 当 ‘X’ 比 ‘O’ 棋子多一枚时, 执 ‘O’ 棋子的玩家应该尚未获胜。

强化练习: 220 Othello^C, 232 Crossword Answers^B, 285 Crosswords^E, 339 SameGame Simulation^D, 379 Hi-Q^C, 395 Board Silly^D, 647 Chutes and Ladders^C, 844 Pousse^D, 10016 Flip-Flop the Squarelotron^C, 10443 Rock Scissors Paper^B, 10703 Free Spots^A, 10813 Traditional BINGO^B, 11140 Little Ali's Little Brother^D, 11221 Magic Square Palindromes^A, 11459 Snakes and Ladders^A, 11520 Fill the Square^B, 11835 Formula 1^D, 12291* Polyomino Composer^D, 12366* King's Poker^D, 12398 NumPuzz I^C, 13115 Sudoku^D。

2.1.3 矩阵变换

在二维数组的应用中, 另外一个常见的主题是对二维数组表示的矩阵进行变换。例如, 旋转或以特定的对称轴进行翻转。

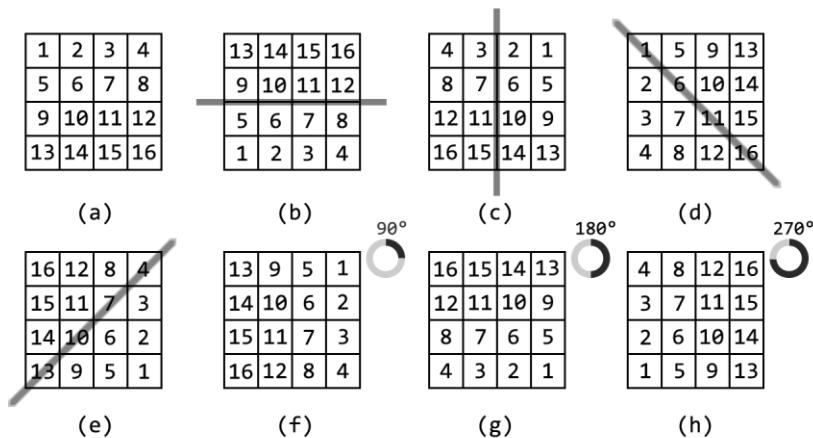


图 2-1 矩阵变换的几种常见形式。(a) 原矩阵; (b) 以水平线为对称轴进行变换; (c) 以垂直线为对称轴进行变换; (d) 以主对角线为对称轴进行变换; (e) 以副对角线为对称轴进行变换; (f) 将原矩阵顺时针旋转 90 度; (g) 将原矩阵顺时针旋转 180 度; (h) 将原矩阵顺时针旋转 270 度

矩阵的变换都具有特定的规律, 根据规律可以容易地将其实现为代码。关键在于处理时要小心谨慎, 特别是要注意边界情况的处理。

强化练习: 466 Mirror Mirror^B, 10855 Rotated Square^A, 10895 Matrix Transpose^B, 10920 Spiral Tap^B, 11040 Add Bricks in the Wall^B, 11349 Symmetric Matrix^A, 11360 Have Fun with Matrices^B, 11470 Square Sums^A, 11581 Grid Successors^B, 12981 Secret Master Plan^D。

扩展练习: 250* Pattern Matching Prelims^D, 11055* Homogeneous Squares^D。

2.1.4 约瑟夫问题

130 Roman Roulette^A (罗马轮盘赌)

历史学家弗拉维斯·约瑟夫^I曾经讲述过这样的一个故事：在公元 67 年的罗马—犹太冲突中，罗马人占领了他所掌权的城镇——乔塔帕塔^{II}。在逃跑过程中，他和另外四十位同伴被困在某个山洞中。罗马人发现了他的藏身之处并劝他投降，但他的同伴不允许他这么做。在这种情况下，他提议按照站位顺序逐个处死对方。也就是说，所有人围成一圈，从某个人开始计数，每数到第三个人就把他处死。他们按此执行，直到剩下最后一个幸存者，而此人恰好是约瑟夫，他随后向罗马人投降了。那么问题来了：是不是约瑟夫暗地里用 41 块石头在一旁偷偷练习过，还是他经过了数学计算，使得他知道需要站在第 31 个位置才能够成为最后的幸存者？

编写程序，当给定一个起始位置后，按照前述类似的方法进行计数，程序能够给出相应的位置序号，从而能够保证你是唯一的幸存者。更确切地说，你的程序应该能够处理如下的约瑟夫问题变形：给定包括你在内的 n 个人，面朝内顺时针围成一圈并依次编号，你的编号为 1，你右手边的人编号为 n ，从编号 i 开始，按顺时针方向数 k 个人，此人将被处死，然后从死者所在位置的左手边开始，顺时针方向数 k 个人，选取此人将死者的尸体埋掉，之后埋尸者调换到死者所在的位置，接着从埋尸者当前所在的位置的左手边开始，顺时针方向数 k 个人，重复以上过程，直到只剩下最后一个幸存者。

例如当 $n=5$, $k=2$, $i=1$ 时，被处死者的编号依次为 2、5、3、1，幸存者的编号为 4。

输入输出

输入包含多行，每行由一组 n 和 k 组成。对于每一行输入，输出 i 为多少，序号为 1 的人才能成为幸存者。例如，在以上给出的例子中，当 $i=3$ 可以保证序号为 1 的人是幸存者。输入以“0 0”结束。你可以假定 n 不大于 100。

样例输入

```
1 1
1 5
0 0
```

样例输出

```
1
1
```

分析

约瑟夫问题，又称约瑟夫环问题，具体描述在题目的前半部分已经给出。原始问题的提法是每数到第 3 个人，就将此人处死。可以将该问题一般化，从编号为 1 的人开始计数，每计数到第 k 个人，将此人移出环形队列，求最后剩下的人的编号。

最直接的方法是模拟计数的过程。设总人数为 n ，以一个内置数组来表示计数的参与者，初始时，数组中所有元素均为 `true`，每当一个参与者被移出，就将该参与者所对应的数组元素置为 `false`，并将剩余的参与者人数减少 1，则可得到如下的代码。

```
//-----2.1.4.1.cpp-----
int n, k;
const int MAXN = 10000;
```

^I 弗拉维斯·约瑟夫 (Flavius Josephus, 37 年—约 100 年)，犹太历史学家。

^{II} 乔塔帕塔，原名 Jotapata，现名 Yodfat，位于现在的以色列境内。

```

bool circle[MAXN + 10];
int findLast() {
    fill(circle, circle + MAXN, true);
    // 从编号为 1 的参与者开始计数。
    int index = 1, nn = n;
    while (nn > 1) {
        // 因为是用数组来模拟环，首先需要从上一次计数结束的地方开始计数到数组末尾。
        int kk = k;
        for (; index <= n; index++)
            if (circle[index] && (--kk == 0))
                break;
        // 若计数未达到设定的计数值，从数组起始位置继续计数。
        while (kk) {
            for (index = 1; index <= n; index++)
                if (circle[index] && (--kk == 0))
                    break;
        }
        // 已计数到 k，剩余参与者人数减少 1。
        circle[index] = false;
        nn--;
    }
    // 查找并返回幸存者编号。
    for (int i = 1; i <= n; i++)
        if (circle[i])
            return i;
}
int main(int argc, char *argv[]) {
    while (cin >> n >> k, n && k) cout << findLast() << endl;
    return 0;
}
//-----2.1.4.1.cpp-----//
```

当 n 和 k 都比较小时，此方法能够快速得到结果，但是随着 n 和 k 的逐渐增大，效率越来越低。因为在模拟过程中，需要不断地遍历整个数组，消耗时间增多¹，此时可考虑使用数学方法来得到结果。为方便数学方法的讨论，将编号方法改成从 0 开始编号（此改动不影响问题的解决，只需要将最后求出的幸存者编号加 1 即可求得从 1 开始编号时幸存者的编号），那么问题转换为：参与者的编号从 0 到 $n-1$ ，顺时针从 0 计数到 $k-1$ ，将第 $k-1$ 个参与者移除，求最后剩下的参与者编号。第一轮计数，设将编号为 x 的人移出队列，则有

$$x = (k - 1) \% n \quad (2.1)$$

此时剩余参与者的编号为

$$0, 1, \dots, x-2, x-1, x+1, x+2, \dots, n-2, n-1 \quad (2.2)$$

因为下一轮计数将从编号为 $x+1$ 的参与者开始，不妨将编号为 $x+1$ 的参与者放在首位，并将编号重新排列为

$$x+1, x+2, \dots, n-2, n-1, 0, 1, \dots, x-2, x-1 \quad (2.3)$$

¹ 使用后续介绍的 `vector` 代替内置数组，通过不断地移除参与者可以使得 `vector` 的大小不断减小，同时结合使用模运算得到下一个被移除的参与者序号，可以在一定程度上提高效率。但对于较大的 n ，仍然存在效率较低的问题。

因为每次移除都对应一次删除操作，频繁地删除将会导致 `vector` 不断调整存储空间，使得耗时增加。

接着从 0 开始为剩余的参与者重新赋予编号，即编号为 $x+1$ 的参与者当前编号为 0，编号为 $x+2$ 的参与者当前编号为 1，依此类推。那么序列 (2.3) 变成

$$0, 1, \dots, n-8, n-7, n-6, n-5, \dots, n-3, n-2 \quad (2.4)$$

可以看到，原有的约瑟夫问题转换为以下问题：给定 $n-1$ 个参与者，编号从 0 到 $n-2$ ，每次从 0 计数到 $k-1$ ，将第 $k-1$ 个参与者移除，求最后幸存者的编号问题。很明显，这是一个递归的过程。若能找出前后两轮计数编号间的关系，利用反向递推可以很容易地由人数为 $n-1$ 时的结果得到人数为 n 时的结果。观察编号序列 (2.3) 和 (2.4)，由于是模 n 的结果，可以看成是连续的，而且有 0 对应 $x+1$ ，1 对应 $x+2$ ， \dots ， $n-2$ 对应 $x-1$ 的关系，令 y' 是序列 (2.3) 中的编号， y 是序列 (2.4) 的编号，那么可以得到序号对应关系为

$$y' = (y + x + 1) \% n \quad (2.5)$$

将 (2.1) 式代入 (2.5) 式可得

$$y' = (y + ((k-1)\%n) + 1)\%n = (y + 1 + (k-1))\%n = (y + k)\%n \quad (2.6)$$

即对于第二轮计数来说，如果最后剩下的参与者编号为 y ，则可通过 (2.6) 式将编号转换为第一轮计数时该参与者的编号 y' 。根据递推关系，只要求出当人数为 1 时的情形，可以反推得到人数为 n 时的结果，而人数为 1 时为最简单的情形，此时剩余参与者的编号为 0。根据上述讨论，利用数学方法求人数为 n 时的幸存者编号变得相当简单。

```
-----2.1.4.2.cpp-----
int n, k;
int findLast() {
    int last = 0;
    // 只需逆向递推 n-1 次。
    for (int i = 2; i <= n; i++) last = (last + k) % i;
    // 返回从 1 开始计数时幸存者的编号。
    return (last + 1);
}
int main(int argc, char *argv[]) {
    while (cin >> n >> k, n && k) cout << findLast() << endl;
    return 0;
}
-----2.1.4.2.cpp-----
```

数学方法对于 n 和 k 比较大的情况也可以很快处理¹。如果需要确定逆时针方向计数的结果，根据计数过程的对称性，假设从 1 开始编号，顺时针进行计数时得到的幸存者编号为 s ，逆时针进行计数时幸存者的编号为 s' ，则两者之间的关系为

$$s' = (n + 2 - s)\%n \quad (2.7)$$

由于此题并不是典型的约瑟夫问题，不便于利用数学方法，使用模拟方法解题更为简单。可以使用一个内置数组，将参与者的序号作为数组元素，每当某人被处死，则将其对应的数组元素置为零。模拟计数过程找到对应的“埋尸者”，进行替换然后继续计数，直到剩下一个人，最后根据环形对称得到结果。

参考代码

¹ 参见 Graham、Knuth、Patashnik 合著的 *Concrete Mathematics*，在此书中，对 $k=2$ 的情形进行了详细地分析，感兴趣的读者可以进一步查阅。

```

const int MAX_NUMBER = 100;
int n, k;
int circle[MAX_NUMBER + 1];
// 按顺时针方向模拟计数的过程。
int findCW(int start, int count) {
    // 从上一次计数的结束位置开始计数到末尾，完成一次循环，以便后续每次从起始位置开始计数。
    for (int i = start; i < n; i++) {
        if (circle[i] > 0 && ((--count) == 0))
            return i;
    }
    while (true) {
        for (int i = 0; i < n; i++)
            if (circle[i] > 0 && ((--count) == 0))
                return i;
    }
}
// 找到幸存者的编号。
int findSurvivor() {
    for (int i = 0; i < n; i++)
        if (circle[i] > 0)
            return circle[i];
}
int main(int argc, char *argv[]) {
    int counter;
    while (cin >> n >> k, n || k) {
        // 赋予编号。
        counter = n;
        for (int i = 1; i <= n; i++) circle[i - 1] = i;
        // 按照题意，每次减少一名参与者。
        int killed = 0, burier;
        while (counter > 1) {
            killed = findCW(killed, k);
            circle[killed] = 0;
            // 找到埋尸者，交换位置。
            burier = findCW(killed, k);
            circle[killed] = circle[burier];
            circle[burier] = 0;
            // 根据题意，往顺时针方向移动一个位置开始下一轮计数。
            killed = (killed + 1) % n;
            counter--;
        }
        // 根据环形对称，假设幸存者的编号为 s，那么从编号为 1 的人往逆时针方向数 s 个人，然后从此位置开始计数，则幸存者为原编号为 1 的人。
        cout << (n - findSurvivor() + 1) % n + 1 << endl;
    }
    return 0;
}

```

强化练习: 133 The Dole Queue^A, 151 Power Crisis^A, 180 Eeny Meeny^D, 305 Joseph^A, 402 M*A*S*H^B, 440 Eeny Meeny Moo^A, 1176 A Benevolent Josephus^D, 10771 Barbarian Tribes^C, 10774 Repeated Josephus^C, 11351 Last Man Standing^C。

扩展练习: 432 Modern Art^D, 10940 Throwing Cards Away II^A, 11053 Flavius Josephus Reloaded^C。

2.2 向量

在标准模板库（Standard Template Library, STL）中，提供了向量（vector）模板类来实现动态数组的功能。向量使用内存中的连续地址空间来存储数组元素，并能够根据需要，对数组的大小自动进行调整。当数组需要存储的元素数量不定时，使用向量来替代C++的内置数组将会给解题带来便利。由于向量是一个模板类，需要为其指定所包含元素的类型。向量的元素类型既可以是整数、浮点数、字符等基本类型，也可以是字符串、结构体、类。

```
//-----2.2.cpp-----
int main(int argc, char *argv[]) {
    vector<int> circle;
    for (int i = 1; i <= 10; i++) circle.push_back(i);
    // 使用传统的下标访问形式。
    for (int i = (circle.size() - 1); i >= 0; i--)
        cout << setw(2) << circle[i] << " ";
    cout << endl;
    // 迭代器访问形式。
    for (auto it = circle.begin(); it != circle.end(); it++)
        cout << setw(2) << *it << " ";
    return 0;
}
//-----2.2.cpp-----
```

输出为：

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

使用向量需要包含头文件<vector>，以下列出了向量的若干常用属性和方法^[16]。

begin()	返回指向向量首个元素的迭代器。
end()	返回指向向量最末元素之后一个位置的迭代器。
rbegin()	返回指向向量最末元素的逆序迭代器。
rend()	返回指向向量首个元素之前一个位置的逆序迭代器。
size()	返回向量的大小。
empty()	测试向量是否为空，为空返回 <code>true</code> ，否则返回 <code>false</code> 。
[index]	获取序号为 <code>index</code> 的元素，不进行范围检查。
at(index)^{c++11}	获取序号为 <code>index</code> 的元素，进行范围检查。
front()	获取向量的首个元素。
back()	获取向量的最末元素。
clear()	清空向量。

assign

将当前向量初始化为 `n` 个特定的值，同时更改向量的大小。

`void assign(size_type n, const value_type& val);`

将其他容器指定范围内的元素值赋值给当前向量。

`template <class InputIterator>`
`void assign(InputIterator first, InputIterator last);`

push_back

将指定元素添加到向量末尾。

¹ 如果函数名称右上角具有 `c++11` 标记，表示只有支持 `c++11` 标准的编译器才支持此函数。

```
void push_back(const value_type& value);
```

pop_back()

删除向量的末尾元素。

```
void pop_back();
```

insert

在指定位置之前插入单个或一组元素，并返回插入元素后的迭代器。需要使用迭代器指定位置参数。

```
iterator insert(iterator position, const value_type& value);
```

```
template <class InputIterator>
```

```
void insert(iterator position, InputIterator first, InputIterator last);
```

erase

移除指定位置的元素，并返回删除元素后的迭代器。需要使用迭代器指定位置参数。

```
iterator erase(iterator position);
```

127 Accordion Patience^A (手风琴纸牌)

本题要求你对手风琴纸牌游戏进行模拟。该游戏的规则如下：

将一叠牌从左至右一张一张摆开，摆开时牌与牌不能重叠。如果某张牌与其左手边的第一张牌“匹配”，或者与左手边的第三张牌“匹配”，那么可以将这张牌移动到左边的相应牌上面。“匹配”是指牌的花色或点数相同。在移动“匹配”的牌后，还要查看是否可以进行更多的“匹配—移动”操作。注意：只有每叠牌最顶端的那张牌可以移动。在牌移动后，如果两叠牌之间出现空隙，则将右侧的牌堆整体往左移动以消除空隙。当最后所有牌都叠在一起时游戏结束，玩家获胜。

当有多张牌均可移动时，总是先移动最左边的牌；当某张牌有多个移动可选择时，总是将牌移动到最靠左的牌堆上。

输入

输入包含多组数据。每组数据包含两行，每行由表示 26 张牌的字母和数字组成，每张牌间隔一个空格。输入最后一行以字符 ‘#’ 作为结束标记。每张牌以两个字符表示，第一个字符表示点数 (A=Ace, 2—9, T=10, J=Jack, Q=Queen, K=King)，第二个字符表示花色 (C=梅花, D=方块, H=红桃, S=黑桃)。每组数据的第一张牌为发牌时最左边的那张牌，依此类推。

输出

对于每组数据输出一行，给出游戏进行到最后时，剩下的牌堆数以及从左至右每个牌堆的牌数。

样例输入

```
QD AD 8H 5S 3H 5H TC 4D JH KS 6H 8S JS AC AS 8D 2H QS TS 3S AH 4H TH TD 3C 6S
8C 7D 4C 4S 7S 9H 7C 5D 2S KD 2D QH JD 6D 9D JC 2C KH 3D QC 6C 9S KC 7H 9C 5C
#
```

样例输出

```
6 piles remaining: 40 8 1 1 1 1
```

分析

按照给定的规则，模拟游戏中牌的移动即可。使用 `vector` 进行模拟较为方便。在下述实现中，使用两个字符表示一张牌，第一个字符为牌的点数，第二个字符为牌的花色。每堆牌位于顶部的牌存放于 `vector`

的末尾。

关键代码

```

// 每一叠牌都使用一个 vector 来表示。
vector<vector<string>> piles;
// 检查是否可以将牌移动到左手边的堆牌上。
bool canMoveToLeft(int index) {
    return (index >= 1 && (piles[index].back() [0] == piles[index - 1].back() [0] ||
        piles[index].back() [1] == piles[index - 1].back() [1]));
}
// 检查是否可将牌移动到左手边第三堆牌上。
bool canMoveToThirdLeft(int index) {
    return (index >= 3 && (piles[index].back() [0] == piles[index - 3].back() [0] ||
        piles[index].back() [1] == piles[index - 3].back() [1]));
}
// 模拟游戏过程。
void play() {
    while (true) {
        // 移除空的牌堆。
        for (int i = piles.size() - 1; i >= 0; i--)
            if (piles[i].size() == 0)
                piles.erase(piles.begin() + i);
        // 从左至右逐个牌堆检查，确定是否可进行相应的移动。
        int index = 0;
        bool moved = false;
        while (index >= 0 && index < piles.size()) {
            if (piles[index].size() == 0)
                break;
            // 检查是否可将牌移动到位于左手侧的第三堆牌上。
            if (canMoveToThirdLeft(index)) {
                piles[index - 3].push_back(piles[index].back());
                piles[index].erase(piles[index].end());
                index -= 3;
                moved = true;
                continue;
            }
            // 检查是否可将牌移动到位于左手侧的堆牌上。
            if (canMoveToLeft(index)) {
                piles[index - 1].push_back(piles[index].back());
                piles[index].erase(piles[index].end());
                index -= 1;
                moved = true;
                continue;
            }
            index++;
        }
        // 当无法继续进行移动时，退出模拟过程。
        if (!moved) break;
    }
}

```

强化练习：162 Beggar My Neighbour^C，170 Clock Patience^A，178 Shuffling Patience^D，451 Poker Solitaire Evaluator^D，10205 Stack'em Up^A，10315 Poker Hands^A，10646* What is the Card^A，10700 Camel Trading^A。

扩展练习：131 The Psychic Poker Player^B，603 Parking Lot^D。

2.3 栈

栈 (stack) 是一种具有先进后出 (last-in-first-out, LIFO) 性质的数据结构，适用于嵌套式结构的处理，如图的深度优先遍历、递归程序的调用。现实生活中栈的例子很多，例如在轮渡过程中，如果船的一端是封闭的，车辆只从船的一端进出，那么在下船时先进入船舱的车辆总是要等到后进入的车辆离开后才能从船舱开出，而且每次只有最靠近出口的车辆可以开出，这时船舱就可以看做是一个栈。

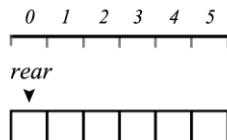
使用栈需要包含头文件<stack>，以下列出了栈的若干常用属性和方法。

empty()	测试栈是否为空，为空返回 <code>true</code> ，否则返回 <code>false</code> 。
size()	返回栈的大小。
top()	获取栈顶元素。
pop()	移除栈顶元素。

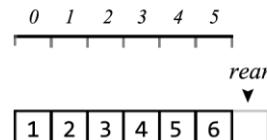
push

将指定的元素压入栈顶。

```
void push(const value_type& value);
```



(a)



(b)

图 2-2 使用数组实现栈，栈容量为 6 个元素，栈顶指针为 `rear`。(a) 栈为空的情形；(b) 栈满的情形

在某些场合，可以使用数组来实现栈的功能，有时候对于解题来说更为方便。方法是使用数组来保存栈的元素，使用一个整数来保存栈元素个数并将其作为栈顶元素的“指针”。当压入元素时，栈顶“指针”递增，出栈时，栈顶“指针”递减。

```
-----2.3.1.cpp-----//
// 常量 ERROR 表示取栈顶元素发生错误，需要根据具体应用进行设置。
const int ERROR = -0x3f3f3f3f, CAPACITY = 1010;
// rear 为栈顶的指针。
int memory[CAPACITY], rear = 0;
bool empty() { return rear == 0; }
int size() { return rear; }
int top() {
    if (rear > 0) return memory[rear - 1];
    return ERROR;
}
void pop() { if (rear > 0) rear--; }
void reset() { rear = 0; }
bool push(int x) {
    if (rear < CAPACITY) { memory[rear++] = x; return true; }
    return false;
}
-----2.3.1.cpp-----//
```

727 Equation^A (等式)

编写程序将中缀表达式转换为对应的后缀表达式。

输入

- (1) 需要转换的中缀表达式在输入文件中给出，每行一个表达式，每个表达式最多 50 个字符^I。
- (2) 输入文件的第一行为一个整数，表示测试数据的组数，后面接着一个空行，之后是测试数据。每组测试数据表示一个中缀表达式，后面接着一个空行。
- (3) 程序只需处理四则运算符：+，-，*，/。
- (4) 所有运算数均为个位数。
- (5) 运算符 '*' 和 '/' 拥有最高优先级，运算符 '+' 和 '-' 拥有最低优先级。具有相同优先级的运算符按照从左至右的顺序进行运算。括号可以改变运算符的优先级顺序。
- (6) 每组测试数据给出的中缀表达式均为合法的中缀表达式。

输出

对于每组测试数据，输出其对应的后缀表达式，要求将后缀表达式在一行上输出。在相邻两组测试数据的输出之间打印一个空行。

样例输入

1

(3+2)*5

样例输出

32+5*

分析

为了便于处理表达式，首先需要了解一下表达式的表示方式。表达式中有运算符和操作数。运算符是对操作数进行某种运算的符号标记，例如，‘+’，‘-’，‘×’，‘÷’ 符号。操作数即为表达式中出现的数值。括号较为特殊，它在中缀表达式中的作用是改变运算的优先顺序。一般在日常生活中，人们都将操作数写在运算符的两边构成一个表达式，如

$$5 * (9 - 1) / 4 + 7 \quad (2.8)$$

在数学上，这样的表示方式称为中缀表达式 (infix notation)，即运算符位于操作数的中间。虽然中缀表达式符合人类的思维习惯，人们处理起来很便利，但是对于计算机来说，直接处理中缀表达式却相对困难。1920 年左右，波兰数学家 Jan Lukasiewicz^{II}提出了表达式的另外一种表示方式——将运算符放在操作数的前面，这样中缀表达式 (2.8) 变成了

$$+/* 5 - 9 1 4 7 \quad (2.9)$$

称之为波兰表达式 (Polish notation)，又称前缀表达式 (prefix notation)。根据波兰表达式，人们又提出了另外一种表示方式——将运算符放在操作数的后面，这样中缀表达式 (2.8) 变成了

$$5 9 1 - * 4 / 7 + \quad (2.10)$$

称之为逆波兰表达式 (reverse Polish notation)，又称后缀表达式 (postfix notation)。波兰表达式和逆波

^I 此处对输入格式进行了适当改变，原题目描述为“每行一个字符，每个表达式最多 50 行”。

^{II} Jan Lukasiewicz 的生平介绍：<http://www.calculator.org/Lukasiewicz.aspx>, 2020。

兰表达式有两个显著的优点：一是不需要使用括号来指定运算顺序（运算顺序已经隐含在表达式中）；二是计算机处理起来非常方便。

对于表达式的逆波兰表示形式，计算机可按照如下方式计算其值：设立一个栈，从左至右扫描逆波兰表达式，当遇到操作数时，将其压入栈中，当遇到运算符时，顺序弹出栈顶的两个操作数进行运算符所指定的运算，然后将结果作为操作数入栈，重复此过程，直到表达式处理完毕，栈顶值即为表达式的值。对于波兰表达式，只需从右往左扫描表达式，按计算逆波兰表达式的方式操作即可。

```
//++++++2.3.2.cpp+++++++
// 使用栈来计算后缀表达式的值。从左至右扫描后缀表达式，如果是数字，则压入栈中，  

// 如果是运算符，则取出栈顶的两个元素进行运算符所指定的运算，然后将运算结果压入栈中，  

// 直到后缀表达式处理完毕，栈顶元素即为表达式的值。  

int calculate(string postfix) {  

    stack<int> result;  

    for (auto c : postfix) {  

        // 如果为数字，将其压入结果栈。  

        if (isdigit(c)) result.push(c - '0');  

        else {  

            // 计算并将结果入栈。注意出栈时运算数的先后顺序。  

            int second = result.top(); result.pop();  

            int first = result.top(); result.pop();  

            if (c == '+') result.push(first + second);  

            if (c == '-') result.push(first - second);  

            if (c == '*') result.push(first * second);  

            if (c == '/') result.push(first / second);  

        }  

    }  

    return result.top();  

}
```

由上可知，只要将中缀表达式转换为逆波兰表达式，使用栈来计算表达式的值将变得非常简便。如何将中缀表达式转换为后缀表达式呢？可以采用如下的算法：

- 1) 从左至右扫描中缀表达式；
- 2) 若读取的是操作数，则判断该操作数的类型，并将该操作数存入操作数栈；
- 3) 若读取的是运算符：
 - 3a) 该运算符为左括号，则直接存入运算符栈；
 - 3b) 该运算符为右括号，则输出运算符栈中的运算符到操作数栈，直到遇到左括号为止；
 - 3c) 该运算符为非括号运算符：
 - 3c1) 若运算符栈为空，则直接存入运算符栈；
 - 3c2) 若运算符栈顶的运算符为左括号，则直接存入运算符栈；
 - 3c3) 若比运算符栈顶的运算符优先级高，则直接存入运算符栈；
 - 3c4) 若比运算符栈顶的运算符优先级低或相等，则输出栈顶运算符到操作数栈，继续比较当前运算符和栈顶运算符的优先级，如果低或相等则输出栈顶运算符，直到优先级比栈顶运算符高或者遇到左括号或栈运算符为空，然后将当前运算符压入运算符栈。
- 4) 若表达式读取完毕，运算符栈中尚有运算符，则依次取出运算符到操作数栈，直到运算符栈为空。
- 5) 操作数栈中存储的即为后缀表达式。

上述算法的核心在于：如果当前运算符是非括号运算符，且优先级不高于运算符栈顶运算符的优先级，表明可以安全地以该运算符作为分割点将表达式分成两个部分，这样的分割不会影响表达式计算结果的正确性。

参考代码

```

// 定义运算符在栈中的优先级顺序，值越高，优先级越高。注意，括号在栈中的优先级最小。
map<char, int> priority = {
    {'+', 1}, {'-', 1}, {'*', 2}, {'/', 2}, {'(', 0}, {')', 0}
};

// 比较运算符在栈中的优先级顺序。
bool lessPriority(char previous, char next) {
    return priority[previous] <= priority[next];
}

// 将中缀表达式转换为后缀表达式。
string toPostfix(string infix) {
    // 操作数栈和运算符栈。
    stack<char> operands, operators;
    for (auto c : infix) {
        // 如果是数字，直接压入操作数栈中。
        if (isdigit(c)) {
            operands.push(c);
            continue;
        }
        // 如果是左括号，直接压入运算符栈中。
        if (c == '(') {
            operators.push(c);
            continue;
        }
        // 如果是右括号。
        if (c == ')') {
            // 弹出运算符栈顶元素，直到遇到左括号。
            while (!operators.empty() && operators.top() != '(') {
                operands.push(operators.top());
                operators.pop();
            }
            // 运算符栈不为空，继续弹出匹配的左括号。
            if (!operators.empty()) operators.pop();
            continue;
        }
        // 如果是非括号运算符，当运算符栈为空，或者运算符栈顶元素为
        // 左括号，或者比运算符栈顶运算符的优先级高，将当前运算符压入
        // 运算符栈。
        if (operators.empty() || operators.top() == '(' ||
            !lessPriority(c, operators.top())) {
            operators.push(c);
        }
        else {
            // 当运算符的优先级比运算符栈顶元素的优先级低或相等时，
            // 弹出运算符栈顶元素，直到运算符栈为空，或者遇到比
            // 当前运算符优先级低的运算符时结束。
            while (!operators.empty() && lessPriority(c, operators.top())) {
                operands.push(operators.top());
                operators.pop();
            }
            // 将当前运算符压入运算符栈。
            operators.push(c);
        }
    }
}

```

```

    }
    // 当中缀表达式处理完毕, 运算符栈不为空时, 逐个弹出压入到操作数栈中。
    while (!operators.empty()) {
        operands.push(operators.top());
        operators.pop();
    }
    // 获取操作数栈中保存的后缀表达式。注意, 栈中保存的表达式是从左至右的顺序,
    // 但从栈中弹出时是从右至左的顺序, 需要适当调整。
    string postfix;
    while (!operands.empty()) {
        postfix = operands.top() + postfix;
        operands.pop();
    }
    // 返回结果。
    return postfix;
}

int main(int argc, char *argv[]) {
    int cases = 0;
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        if (c > 1) cout << '\n';
        string infix;
        cin >> infix;
        cout << toPostfix(infix) << '\n';
    }
    return 0;
}
//+++++++++++++++++++++++++++++++++++++++++++++++++2.3.2.cpp+++++++++++++++++/

```

强化练习: 172 Calculator Language^C, 198 Peter's Calculator^D, 327 Evaluating Simple C Expressions^B, 533 Equation Solver^D, [551](#) Nesting a Bunch of Brackets^C, [622](#) Grammar Evaluation^C, [673](#) Parentheses Balance^A, [11111](#) Generalized Matrioshkas^B, [13055](#) Inception^D。

扩展练习: 214 Code Generation^D, [514](#) Rails^A, 586 Instant Complexity^C, [732](#) Anagrams by Stack^B, [12392](#) Guess the Numbers^D。

2.4 队列及优先队列

2.4.1 队列

队列 (queue) 是一种具有先进先出 (first-in-first-out, FIFO) 性质的数据结构, 在编程竞赛中应用较多, 特别是图算法中。例如, 它常被用于实现图的广度优先遍历。队列在应用形式上类似于日常生活中银行的排队, 排在队首的人先获得柜台人员的服务先离开, 排在队尾的人后获得服务后离开。

使用队列需要包含头文件<queue>, 以下列出了队列的若干常用属性和方法。

empty()	测试队列是否为空, 为空返回 <code>true</code> , 否则返回 <code>false</code> 。
size()	返回队列的大小。
front()	获取队首元素。
back()	获取队尾元素。

push

将指定的元素追加到队尾。

```
void push(const value_type& value);

pop
移除队首元素。
void pop();
```

需要注意的是，队列不支持以下标形式访问元素的操作，亦不支持查找操作，即不能通过队列本身提供的属性或方法得知某个元素是否存在于队列中。一般需要通过与其他数据结构配合使用来进行查询操作，例如配合使用映射或集合。在取用队列的元素之前，一定要注意检查队列是否为空，如果为空仍然进行取用队列元素的操作，会产生错误或者埋下不易排查的 Bug。

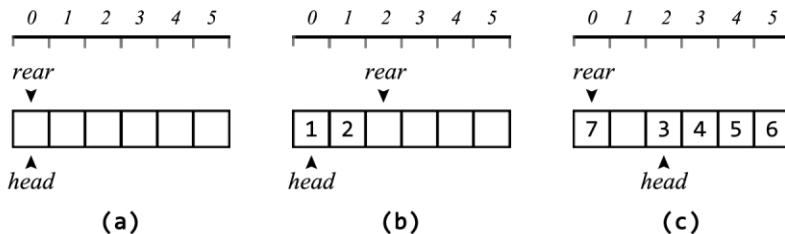


图 2-3 使用数组实现队列，队列容量为 6 个元素，队首指针为 *head*，队尾指针为 *rear*。(a) 初始时，队首和队尾重合，队列为空；(b) 插入两个元素后，队首位置不变，队尾向后移动；(c) 插入和删除若干元素后，队尾已经在队首之前

类似于使用数组来实现栈的功能，同样也可以使用数组来实现队列的功能。方法是设立两个“指针”来指示队列的首位和末尾，将数组作为一个环形数组看待，指针到达数组的末端时可以绕回数组的起始。只要数组的大小设置得比题目中可能的应用大，就不会发生“指针”首尾交叉的情况，从而保证功能的正确性。

```
-----2.4.1.cpp-----
const int CAPACITY = 1010;
// head 为队首的指针, rear 为队尾的指针。
int memory[CAPACITY], head = 0, rear = 0;
// 属性。
bool empty() { return head == rear; }
int size() { return rear >= head ? (rear - head) : (rear + CAPACITY - head); }
int front() { return memory[head]; }
int back() { return memory[(rear - 1 + CAPACITY) % CAPACITY]; }
// 注意, 使用 push() 和 pop() 之前要检查队列的大小是否符合要求。
void push(int x) {
    memory[rear] = x;
    rear = (rear + 1) % CAPACITY;
}
void pop() { head = (head + 1) % CAPACITY; }
void reset() { head = rear = 0; }
-----2.4.1.cpp-----
```

强化练习：[10172 The Lonesome Cargo Distributor^C](#)，[10935 Throwing Cards Away I^A](#)。

144 Student Grants^A（助学金）

积贫症者（Impecunia）联合政府决定使大专学生助学金的领取过程变得繁琐而费时，以期打消学生参加大专教育的热情。政府为每位学生注册了一张背面有磁条的 ID 卡，用来记录学生的助学金发放情况。卡

的余额最初为 0 美元，每年的上限是 40 美元，学生可以在他们的生日临近的时候于工作日支取（积贫症者所处的社会环境类似于中世纪，只有男性才有接受高等教育的权利）。因此在工作日，总是可以看到有多达 25 名的学生出现在学生助学金部附近，他们的目的就是支取助学金。

助学金可在自动取款机(Automated-Teller Machine, ATM)上支取，此种 ATM 由可重编程的 8085¹芯片控制。ATM 由国有企业制造并带有防暴设计。结构上，它由内外两部分组成，内部为保险箱，存储了大量的 1 美元硬币，外部为储存箱，供取款者支取现金。为了减少 ATM 被盗时的损失，只有当储存箱中的硬币被支取完毕后，保险箱才向其输出硬币。每天早上 ATM 开机时，储存箱为空，保险箱向储存箱输出一枚硬币，当此枚硬币被取走后，输出两枚硬币，然后是三枚硬币，每次增加一枚硬币，直到达到预设的上限—— k 枚硬币，之后输出到储存箱的硬币数量重置为一枚，然后是两枚，按此循环。

每名学生依次排队，插入他的 ID 卡在 ATM 上取款。他可以取走 ATM 储存箱中的所有硬币，ATM 会把该学生已经支取助学金的总额写到 ID 卡上，如果学生的支取总额尚未达到每年设定的 40 美元上限值，那么他可以在取卡后重新排队，继续等待取款。如果储存箱中的现金加上学生已经支取的现金数量超过 40 美元，学生只能支取与 40 美元之间的差额，剩余在储存箱中的现金留给下一位学生支取。

编写程序读入一系列的 N ($1 \leq N \leq 25$) 和 k ($1 \leq k \leq 40$)，确定学生离开排队序列的先后顺序。

输入与输出

输入的每一行包括两个整数，分别表示 N 和 k 。输入以两个空格分隔的 0 表示结束。

对于每行输入均输出一行。输出由完成取款依次离开队列的学生的序号组成。学生的序号按最开始排队时的先后顺序确定，第一个学生的序号为 1。每个序号按右对齐、输出宽度 3 进行输出。

样例输入

```
5 3
0 0
```

样例输出

```
1 3 5 2 4
```

分析

此题可通过模拟助学金支取的过程予以解决——构造一个学生排队的队列，学生逐个取款，直到达到取款的上限后离开。使用 queue 数据结构对取款队列进行模拟。

参考代码

```
// 表示学生取款的结构体，id 为学生的序号，withdraw 为学生已经支取的奖学金数额。
struct student { int id, withdraw; };
int main(int argc, char *argv[]) {
    int n, k;
    while (cin >> n >> k, n || k) {
        // 构建学生取款队列。
        queue<student> students;
        for (int i = 1; i <= n; i++) students.push((student){i, 0});
        // coin 表示每次向储存箱输出的硬币数量，remain 表示储存箱中剩余的硬币数量。
        int coins = 0, remain = 0;
        while (true) {
            // 确定当前储存箱的硬币数量。
            int current = (remain > 0) ? remain : ((++coins) > k ? (coins = 1) : coins);
            // 如果学生队列为空，表明处理完毕，可以退出循环。
            if (students.empty()) break;
            // 模拟学生取款。
            student s = students.front();
            s.withdraw += current;
            if (s.withdraw > 40) s.withdraw = 40;
            students.pop();
            if (s.withdraw == 40) cout << s.id << " ";
        }
    }
}
```

```

student s = students.front(); students.pop();
// 额度达到上限值, 完成取款。
if ((s.withdraw + current) >= 40) {
    remain = s.withdraw + current - 40;
    cout << setw(3) << right << s.id;
}
// 额度未达到上限值, 继续排队取款。
else {
    s.total += current;
    students.push(s);
    remain = 0;
}
cout << '\n';
}
return 0;
}

```

强化练习: 417 Word Index^A, 10901 Ferry Loading III^B, 11034 Ferry Loading IV^B, 11797 Drutojan Express^D。

扩展练习: 540 Team Queue^A。

2.4.2 优先队列

优先队列 (priority queue) 具有队列的性质, 而且可以根据指定的条件自动对队列中的元素进行位置调整, 使得队首元素按照给定的比较规则总是最大 (或最小) 的元素。

使用优先队列需要包含头文件<queue>, 以下列出了优先队列的若干常用属性和方法。

empty()	测试队列是否为空, 为空返回 <code>true</code> , 否则返回 <code>false</code> 。
size()	返回队列的大小。
top()	获取队首元素。
pop()	移除队首元素。

push

将指定的元素追加到队尾。

void push(const value_type& value)

优先队列只能获取队首元素且其操作名称与普通的队列有差异——优先队列获取队首元素的是 `top` 操作, 无普通队列的 `front` 和 `back` 操作。

在 C++ 的 SGI^I 库实现中, 优先队列的元素有序采用堆排序实现, 故而优先队列的构造函数较为特殊。对于内置数据类型来说, 一般情况下只需要声明元素的数据类型即可。例如要声明一个整数优先队列, 可以使用:

```

// 默认优先级为最大值优先, 即队首为最大元素。
priority_queue<int> queue;

```

若是特定解题需要, 需要更改优先级顺序, 则可使用以下的几种示例形式。需要特别注意的是, 在使用结构模板更改优先级时, 应用 `greater<T>` 得到的是最小值优先队列, 而应用 `less<T>` 得到的是最大值优

^I Silicon Graphics International Corporation, 硅图国际公司, 成立于 1982 年, 总部设在美国加州旧金山硅谷。

先队列，有违直观感觉。其原因是在 SGI 的库实现中，优先队列是采用最大堆完成的，具有“最大值”的元素会位于堆顶（二叉树的根结点）。因此，为了排序过程中能够使得较小的元素位于堆顶，在重载小于运算符时要对应地修改元素大小需要满足的关系。

```
-----2.4.2.cpp-----
// 默认为最大值优先队列。
priority_queue<int> greaterDefault;

// 使用结构模板 greater<T>得到最小值优先队列。
priority_queue<int, vector<int>, greater<int>> lessInt;

// 使用结构模板 less<T>得到最大值优先队列。
priority_queue<int, vector<int>, less<int>> greaterInt;

// 通过重载括号运算符得到最小值优先队列。
struct A {
    bool operator() (int x, int y) const { return x > y; }
};
priority_queue<int, vector<int>, A> lessA;

// 通过重载括号运算符得到最大值优先队列。
struct B {
    bool operator() (int x, int y) const { return x < y; }
};
priority_queue<int, vector<int>, B> greaterB;

// 通过重载小于运算符得到最小值优先队列。
struct C {
    int index;
    bool operator<(C x) const { return index > x.index; }
};
priority_queue<C> lessC;

// 通过重载小于运算符得到最大值优先队列。
struct D {
    int index;
    bool operator<(D x) const { return index < x.index; }
};
priority_queue<D> greaterD;

// 通过重载小于运算符得到最小值优先队列。
class E {
public:
    int index;
    bool operator<(E x) const { return index > x.index; }
};
priority_queue<E> lessE;
-----2.4.2.cpp-----
```

136 Ugly Numbers^A（丑数）

丑数是指那些素因子只包括 2、3、5 的数。序列：1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, …列出了丑数序列的前 11 个数，按照惯例，1 也包括在丑数中。编写程序找到并输出第 1500 个丑数。

输入输出

此题没有输入。输出由一行组成，将样例输出中的< number >替换为第 1500 个丑数。

样例输出

```
The 1500'th ugly number is < number >.
```

分析

朴素的方法是从 1 开始枚举整数，逐个数判断是否为丑数，直到找到第 1500 个丑数。虽然此方法运行时间较长，但由于输出特殊（只要求输出一个特定序号的丑数），完全可以先运行程序得到指定丑数后再提交，以免超过运行时间限制。

参考代码

```
int main(int argc, char *argv[]) {
    long long int start = 1;
    int counter = 1;
    while (counter < 1500) {
        start++;
        long long int number = start;
        while (number % 2 == 0) number /= 2;
        while (number % 3 == 0) number /= 3;
        while (number % 5 == 0) number /= 5;
        if (number == 1) counter++;
    }
    cout << "The 1500'th ugly number is " << start << "." << endl;
    return 0;
}
```

另外一种更为巧妙的方法是利用优先队列解题。由于丑数乘以 2、3、5 之后仍然为丑数，那么可将生成的丑数继续乘以这三个素因子以得到后续丑数。具体来说，就是将生成的丑数放入最小优先队列中，每次从队首取最小的一个丑数进行乘的操作，这样就能够保证得到丑数序列。由于优先队列本身并不支持查找，需要另外一种数据结构来保存已经得到的丑数，以防止生成重复的丑数而导致计数错误（例如丑数 6 可以通过 2 乘以 3 得到，也可以通过 3 乘以 2 得到）。

参考代码

```
typedef long long int bigNumber;
int main(int argc, char *argv[]) {
    int factors[3] = {2, 3, 5};
    set<bigNumber> uglyNumbers;
    priority_queue<bigNumber, vector<bigNumber>, greater<bigNumber>> candidates;
    candidates.push(1);
    for (int i = 1; i <= 1500; i++) {
        bigNumber top;
        do {
            top = candidates.top(); candidates.pop();
        } while (uglyNumbers.count(top) > 0);
        uglyNumbers.insert(top);
        for (int j = 0; j < 3; j++) {
            bigNumber next = top * factors[j];
            if (uglyNumbers.count(next) == 0) candidates.push(next);
        }
    }
    if (i == 1500) {
        cout << "The 1500'th ugly number is " << top << "." << endl;
    }
}
```

```

        break;
    }
    return 0;
}

```

强化练习: [161 Traffic Lights^A](#), [443 Humble Numbers^A](#), [467 Synching Signals^C](#), [978 Lemmings Battle^B](#), [1064 Network^D](#), [1203 Argus^A](#), [11621 Small Factors^C](#), [12100 Printer Queue^B](#), [13190 Rockabye Tobby^D](#)。

扩展练习: [212 Use of Hospital Facilities^E](#), [304 Department^E](#), [11995 I Can Guess the Data Structure^A](#), [12207 That is Your Queue^C](#)。

2.5 双端队列

标准的队列只能在队尾进行压入操作，在队首进行弹出操作，而双端队列（double-ended queue）在队列的两端都能进行增加和删除元素的操作。为了区分双端队列在队首和队尾的操作，按照惯例，一般将队首进行的操作称之为压入（push）和弹出（pop），队尾进行的操作称之为插入（insert）和移除（remove）。

使用双端队列需要包含头文件`<deque>`，以下列出了双端队列的若干常用属性和方法。

<code>empty()</code>	测试队列是否为空，为空返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>size()</code>	返回队列的大小。
<code>front()</code>	获取队首元素。
<code>back()</code>	获取队尾元素。
<code>pop_back()</code>	移除队尾元素。
<code>pop_front()</code>	移除队首元素。
<code>clear()</code>	清空整个队列。

`push_back`

将指定的元素追加到队尾。

```
void push_back(const value_type& value);
```

`push_front`

将指定的元素插入到队首。

```
void push_front(const value_type& value);
```

`insert`

将元素插入到指定位置，必须使用迭代器指定位置。

```
iterator insert(iterator position, const value_type& value);
```

`erase`

移除指定位置的元素，必须使用迭代器指定位置。

```
iterator erase(iterator position);
```

957 Popes^B（教皇）

在教皇 John Paul 二世去世的时候，美国《时代》周刊做了一个统计，得到了以下结果：从公元 867 年（Adrian 二世）到公元 965 年（John 十三世），这近一百年间共选出了 28 位教皇，同时这也是以 100 年为时间跨度选出最多教皇的年份区间。编写程序，在给定教皇当选年份 Y （为正整数）的列表后，计算在给定的时间跨度内，具有最大当选教皇数量的年份区间以及当选的教皇数量。注意，当给定年份 N 时， Y 年内的含义是指从第 N 年的第一天到第 $N+Y-1$ 年的最后一天这个时间段。如果有多个年份区间均具有最大当选教皇数量，取最先出现的年份区间。

输入

输入包含多组测试数据，相邻两组测试数据之间以一个空行分隔，每组测试数据的格式描述如下。

每组测试数据的第一行是一个正整数 Y ，表示我们感兴趣的时间跨度。第二行包含另外一个正整数 P ，表示教皇的数量。接下来的 P 个正整数，表示这 P 位教皇各自当选的年份，年份按照时间顺序给出。表示教皇数量的整数 $P \leq 100000$ ，表示教皇当选年份的整数 $L \leq 1000000$ ， $Y \leq L$ 。

输出

对于每组测试数据输出一行，每行包含三个整数，以空格分隔。第一个整数表示在 Y 年内当选的最大教皇数量，第二个整数表示区间的起始年份，第三个整数表示区间的结束年份。

样例输入

```
5
20
1 2 3 6 8 12 13 13 15 16 17 18 19 20 20 21 25 26 30 31
```

样例输出

```
6 16 20
```

分析

本题的实质是在给定的年份序列中寻找这样的一个连续子序列：该子序列的起始年份和结尾年份的差值在 Y 年内且子序列的长度最大。假想有一种数据结构符合解题的需要，当顺序读取输入数据时，先将某个年份 A 压入该数据结构，对于后续的年份 B ，如果它和该数据结构中的第一个元素——即起始年份 A 的差值小于 Y ，则将后续年份 B 压入，直到不满足条件，此时数据结构中元素的个数即为当前 Y 年内的最大教皇当选数量，将此数量与已经得到的最大当选数量 M 进行比较，如果比已经得到的最大数量 M 还要大，则更新最大数量 M 的值和相应的起始和结束年份。接下来，从该数据结构的第一个元素开始进行删除操作，直到第一个元素所代表的年份 A 和尚未进入该数据结构的年份 B 之间的差值小于 Y ，然后将年份 B 压入数据结构，继续读入数据以构造满足年份跨度的子序列。重复以上过程，即可找到符合题意的最大值及区间。

回顾队列的性质，可以发现它正好符合上述假想数据结构的要求，因此使用它来解决本题非常合适。对于本题来说，既可以使用双端队列，也可以使用普通的队列进行解题。使用此种队列的方式进行解题的过程类似于一个“窗口”在待扫描序列上移动的过程，因此有一个特别的名称——“滑动窗口”(sliding window) 技巧¹。

参考代码

```
int main(int argc, char *argv[]) {
    int period, popes, year;
    int maxCount, maxStart, maxEnd;
    while (cin >> period) {
        cin >> popes >> year;
        // 将第一位教皇的年份置入双端队列并设置相应的变量值。
        deque<int> years;
        years.push_back(year);
```

¹ 亦称“尺取法”，即类似一把可以伸缩的“尺”在数组上移动，“尺”所覆盖的范围是符合要求的序列。

```

maxCount = 1, maxStart = year, maxEnd = year;
// 依次处理余下的教皇年份。
for (int i = 2; i <= popes; i++) {
    cin >> year;
    if (year - years.front() < period)
        years.push_back(year);
    else {
        if (years.size() > maxCount) {
            maxCount = years.size();
            maxStart = years.front();
            maxEnd = years.back();
        }
        while (!years.empty()) {
            if (year - years.front() < period) {
                years.push_back(year);
                break;
            }
            years.pop_front();
        }
    }
}
if (years.size() > maxCount) {
    maxCount = years.size();
    maxStart = years.front();
    maxEnd = years.back();
}
cout << maxCount << ' ' << maxStart << ' ' << maxEnd << '\n';
}
return 0;
}

```

强化练习: 210 Concurrent Simulator^D, 999 Book Signatures^E, [1121 Subsequence^B](#), [11536 Smallest Sub-Array^C](#)。

2.6 映射

映射 (map) 是一种关联容器, 它提供了键 (key) 和值 (value) 的一一对应, 与日常生活中邮政编码和地名的对应关系类似。标准库中存在两种形式的映射——普通映射和多重映射。在普通映射中, 键必须唯一, 而在多重映射 (multimap) 中, 键可以不唯一。

使用映射需要包括头文件`<map>`或`<multimap>`, 以下列出了映射的若干常用属性和方法。

begin()	返回指向映射首个元素的迭代器。
end()	返回指向映射最末元素之后一个位置的迭代器。
rbegin()	返回指向映射最末元素的逆序迭代器。
rend()	返回指向映射首个元素之前一个位置的逆序迭代器。
size()	返回映射的大小。
empty()	测试映射是否为空, 为空返回 <code>true</code> , 否则返回 <code>false</code> 。
clear()	清空映射。
[key]	使用下标的形式来获取键 <code>key</code> 对应的值 <code>value</code> 。
insert	
插入元素。	
iterator insert(const value_type& value);	

erase

删除元素。

```
void erase(iterator position)
size_type erase(const key_type& key);
```

swap

交换两个映射的内容，要求两个映射所包含的数据类型必须相同。

```
void swap(map& x);
void swap(multimap& x);
```

find

查找元素。

```
iterator find(const key_type& key);
```

count

获取指定键 key 在映射中出现的次数。

```
size_type count(const key_type& key) const;
```

lower_bound

返回一个迭代器，指向映射中第一个不小于指定参数的元素的位置。

```
iterator lower_bound(const key_type& key);
```

upper_bound

返回一个迭代器，指向映射中第一个大于指定参数的元素的位置。

```
iterator upper_bound(const key_type& key);
```

equal_range

返回一对迭代器，在此迭代器范围内的元素的键等于给定的参数。

```
pair<iterator, iterator> equal_range(const key_type& key);
```

使用 map 需要注意以下几点：

(1) 以方括号加 key 的形式访问 map 时，如果 key 存在，则直接返回其对应的 value，否则以 key 为键新建一个 pair 插入到 map 中，key 对应的 value 为默认值，最终会导致 map 的大小增加 1。

(2) map 中的元素默认是按照 key 值升序排列。

(3) map 中的元素是按照有序方式存储的，如果不需要元素有序存储而只是关注查询的效率，可以使用更为高效的 unordered_map。使用 unordered_map 需要包含头文件<unordered_map>。

(4) map 不支持通过下标形式对容器中的元素进行访问，而只支持迭代器的访问方式，有时候会为获取指定元素带来不便，需要采取一些变通的方法。例如，需要获取容器中最后一个元素，可以采用以下方法：

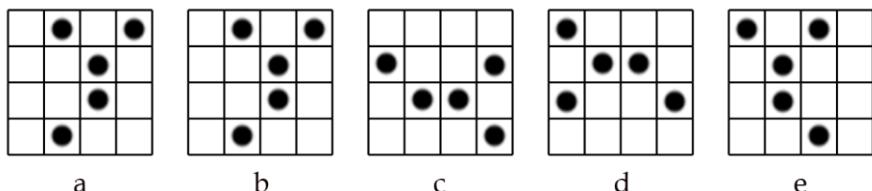
```
map<int, int> cnt = {{1, 10}};
auto it = --cnt.end();
cout << it->first << ' ' << it->second << endl;
```

141 The Spot Game^A (斑点游戏)

斑点游戏在 $N \times N$ 的方格棋盘上进行，下面以 $N=4$ 为例解释其游戏规则。在游戏中，双方选手轮流操作，可以选择在棋盘上的空白方格内放入一个棋子（棋子为黑色），或从棋盘上移除一个棋子，在此过程中，产生了各种棋盘模式。假如一种棋盘模式（包括其旋转 90 度或 180 度后形成的棋盘模式）在后续游戏中被另外一名选手重复，则最初生成此种棋盘模式的选手判定为负，如果在游戏过程中没有任何一位选手产生重

复的棋盘模式，那么在 $2 \times N$ 步后，游戏以平局结束。

考虑如下的棋盘模式



假如棋盘模式 a 在游戏的较早时间产生，那么在后续游戏中如果出现棋盘模式 b、c、d（还有一种重复的棋盘模式未绘出），则产生棋盘模式 a 的选手判定为负，如果出现棋盘模式 e，则不认为是重复的棋盘模式。

输入输出

输入由一系列游戏组成。每次游戏以表示棋盘大小的整数 N ($2 \leq N \leq 50$) 开始，紧接着每行一步操作，总是给出 $2 \times N$ 步操作，不论在所有操作之前游戏是否已经结束。一个操作由以下要素表示：方格位置的坐标（横纵坐标值为 1 至 N 之间的整数），一个空格，一个“+”或一个“-”（提示是增加棋子还是移除棋子）。你可以假定所有操作都是合法的，不会出现向已有棋子的方格放入棋子或者从没有棋子的方格移除棋子的情形。输入以一个 0 作为结束标记。

每次游戏都应输出一行，确定最终是哪位选手在哪一次操作获胜还是以平局结束游戏。

样例输入

```
2
1 1 +
2 2 +
2 2 -
1 2 +
2
1 1 +
2 2 +
1 2 +
2 2 -
0
```

样例输出

```
Player 2 wins on move 3
Draw
```

分析

总的解题思路是模拟游戏的进行，同时记录已经生成的棋盘模式，检查是否有重复的棋盘模式产生。解题的关键是找到一种恰当的棋盘模式表示方式。为了判断是否有重复，在每次操作后，需要生成当前棋盘模式旋转 90 度或 180 度后的棋盘模式，同时记录产生此棋盘模式的游戏选手编号，存放于某种数据结构中备查以判断有无重复。

可以将棋盘有棋子的方格使用字符 ‘1’ 来表示，没有棋子的方格使用字符 ‘0’ 来表示，那么可以将整个棋盘的状态表示成一个只包含 0 和 1 字符的 `string` 类实例（使用二维数组表示棋盘亦可，使用 `string` 类这种一维数组的形式来表示棋盘更为简洁），由于将棋盘旋转 90 度和 180 度后，原下标和旋转后的下标有相应的规律可循，可以根据这种规律获得旋转后的棋盘模式表示，然后将这些表示棋盘模式的 `string` 类实例储存在 `map` 关联容器中备查。

将 $N=4$ 的棋盘从左至右、从上至下标记 1~16 的整数（图 a），其顺时针旋转 90 度（图 b）、逆时针旋

转 90 度 (图 c)、旋转 180 度 (顺时针和逆时针旋转 180 度的棋盘模式相同) 后的棋盘模式 (图 d) 为

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

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

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

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

根据旋转后的数字位置的规律, 可以容易地得到旋转后的棋盘模式。

二维数组下标转换为一维数组下标 (下标从 0 开始计数): 二维数组中元素的下标为 i 行 j 列, 那么在相应的一维数组表示中, 其下标为 $i \times n + j$, 其中 n 为一行元素的数量。

参考代码

```

// 获取顺时针旋转 90 度后的棋盘模式。
string rotateCW90(string matrix, int n) {
    string newMatrix;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            newMatrix += matrix[(n - 1) * n + (i - 1) - (j - 1) * n];
    return newMatrix;
}

// 获取逆时针旋转 90 度后的棋盘模式。
string rotateCCW90(string matrix, int n) {
    string newMatrix;
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            newMatrix += matrix[(n - 1) - (i - 1) + (j - 1) * n];
    return newMatrix;
}

// 获取旋转 180 度后的棋盘模式。
string rotate180(string matrix) {
    reverse(matrix.begin(), matrix.end());
    return matrix;
}

int main(int argc, char *argv[]) {
    int n, x, y;
    string line;
    map<string, int> steps;
    while (cin >> n, n) {
        int winner = 0, move = 0;
        string matrix = string(n * n, '0');
        steps.clear();
        for (int i = 1; i <= 2 * n; i++) {
            cin >> x >> y;
            getline(cin, line);
            // 如果已经确定赢家, 后续输入不需处理。
            if (winner > 0) continue;
            // 根据操作确定单元格所对应的字符。
            matrix[(x - 1) * n + (y - 1)] =
                (line.find('+') != linenpos) ? '1' : '0';
        }
    }
}

```

```

// 检查当前矩阵的字符串表示是否已经存在。
if (steps.find(matrix) != steps.end()) {
    winner = (steps[matrix] % 2 == 1) ? 2 : 1;
    move = i;
}
// 获得初始矩阵的四种变换的字符串表示。
string newMatrix[4] = {
    matrix,
    rotateCW90(matrix, n), rotateCCW90(matrix, n),
    rotate180(matrix)
};
// 检查变换是否已经存在。
for (int j = 0; j < 4; j++)
    if (steps.find(newMatrix[j]) == steps.end())
        steps.insert(make_pair(newMatrix[j], i));
}
// 根据结果输出。
if (winner == 0) cout << "Draw\n";
else cout << "Player " << winner << " wins on move " << move << '\n';
}
return 0;
}

```

强化练习: 119 Greedy Gift Givers^A, 196 Spreadsheet^B, 340 Master-Mind Hints^A, 350 Pseudo-Random Number^A, 380 Call Forwarding^C, 394 Mapmaker^A, 405 Message Routing^D, 484 The Department of Redundancy Department^A, 962 Taxicab Numbers^D, 1727 Counting Weekend Days^C, 10145 Lock Manager^D, 10226 Hardwood Species^A, 10282 Babelfish^A, 10295 Hay Points^A, 11239 Open Source^B, 11286 Conformity^A, 11308 Bankrupt Baker^B, 11428 Cubes^A, 11572 Unique Snowflakes^A, 11629 Ballot Evaluation^B, 11917 Do Your Own Homework^A, 11991 Easy Problem from Ruija Liu^A, 12504 Updating a Dictionary^C, 12592 Slogan Learning of Princess^B, 12820 Cool Word^C。

扩展练习: 335 Processing MX Records^D, 506 System Dependencies^D, 698 Index^D, 11860* Document Analyzer^D。

2.7 集合

集合 (set) 是一种存储元素的关联容器, 集合关联容器类分为两种子类型——普通集合和多重集合 (multiset)。普通集合的元素唯一, 而多重集合中的元素可以重复。集合的作用是保存一组元素, 可以在 $O(\log n)$ 的时间复杂度内获取该元素。

使用集合需要包含头文件`<set>`, 以下列出了集合的若干常用属性和方法。

begin()	返回指向集合首个元素的迭代器。
end()	返回指向集合最末元素之后一个位置的迭代器。
rbegin()	返回指向集合最末元素的逆序迭代器。
rend()	返回指向首个元素之前一个位置的逆序迭代器。
empty()	测试集合是否为空, 为空返回 <code>true</code> , 否则返回 <code>false</code> 。
clear()	清空集合。
 insert	
插入元素。	
iterator insert(const value_type& value);	

erase

删除元素。

```
void erase(iterator position)
size_type erase(const value_type& value);
```

swap

交换两个集合的内容，要求两个集合所包含的数据类型必须相同。

```
void swap(set& x);
void swap(multiset& x);
```

find

查找指定值在集合中的位置，使用迭代器表示结果。

```
iterator find(const value_type& value) const;
```

count

获取指定值在集合中出现的次数。

```
size_type count(const value_type& value) const;
```

lower_bound

返回指向集合中第一个不小于指定参数的元素的位置迭代器。

```
iterator lower_bound(const value_type& value) const;
```

upper_bound

返回指向集合中第一个大于指定参数的元素的位置迭代器。

```
iterator upper_bound(const value_type& value) const;
```

equal_range

返回一对迭代器，在此迭代器范围内的元素的键等于指定值。

```
pair<iterator, iterator> equal_range(const value_type& value) const;
```

集合一般应用于需要反复查询某些值是否存在的场合。在 STL 的 SGI 实现中，集合使用的是红黑树，可以在 $O(\log n)$ 的时间复杂度内确定元素是否在集合中，效率较高。

在 `set` 中，元素是有序排列的，如果不需要元素有序排列，而只是关注查询效率，可以使用 `unordered_set`，效率较 `set` 要高。使用 `unordered_set` 需要包含头文件 `<unordered_set>`。

`set` 在初始化时可以有以下几种方式：

(1) 使用类似于内置数组的初始化方式，将数值直接赋予 `set` 实例，例如：

```
set<int> s1 = {0, 11, 24, 39, 416, 525, 636, 749, 864, 981};
```

(2) 使用其他容器类的一部分元素初始化 `set` 实例，例如：

```
vector<int> s1 = {0, 11, 24, 39, 416, 525, 636, 749, 864, 981};
set<int> s2(s1.begin(), s1.end());
```

156 Ananagrams^A (非变位词)

许多字谜游戏爱好者都熟悉变位词——一组单词的构成字母相同但顺序不同——例如，OPTS，SPOT，STOP，POTS 和 POST。然而有些单词不管你如何变换字母的位置，都无法构成另外一个单词，这些词称为非变位词，例如 QUIZ。

当然, 以上非变位词的定义和人们的职业性质有关, 你可能认为 ATHENE^I是一个非变位词, 但是化学家可以很快举出其变位词 ETHANE^{II}。可以把英语中的所有词汇看成在同一个范畴内, 但是这会导致一些问题。如果将范畴予以限制, 比如说音乐范畴里, SCALE 是一个相对非变位词(因为 SCALE 的变位词 LACES 不在音乐范畴词汇内), 而 NOTE 却不是, 因为它能产生变位词 TONE。

编写程序读入属于特定范畴的词汇字典并确定其中的相对非变位词。注意: 由单个字母构成的单词是相对非变位词, 因为它们根本就无法进行“变位”操作。输入的字典中包括的单词数量不超过 1000 个。

输入

输入包含多行, 每行不超过 80 个字符, 但包含的单词数量不定。每个单词最多由 20 个字母构成, 字母可为大写或小写, 不存在单词跨行的情形。单词间的空格数量不定, 但在同一行的单词之间至少有一个空格。注意, 包含相同字母但大小写不同的单词互为变位词, 比如 tHeD 和 EdiT。输入以只包含 '#' 字符的一行作为结束标记。

输出

输出包含多行, 每行包含字典中一个相对非变位词。这些单词必须按照字典序(大小写敏感)输出。输入保证存在至少一个相对非变位词。

样例输入

```
ladder came tape soon leader acme RIDE lone Dreis peat
ScALE orb eye Rides dealer NotE derail LaCeS drIed
noel dire Disk mace Rob dries
#
```

样例输出

```
Disk
NotE
derail
drIed
eye
ladder
soon
```

分析

由变位词的定义, 其构成字母相同(可能大小写不一致)但顺序不同, 那么可以将单词中的字母转换为小写再进行排序后判断。设立两个存储结构, 一个为 `vector`, 另一个为 `multiset`, 逐个读入单词, `vector` 保存单词的原始形式, `multiset` 保存单词转换为小写排序后的形式。当输入读取完毕时, 将原始的单词形式逐个取出, 令原始的单词为 *A*, 将其转换为小写后排序成为 *B*, 在 `multiset` 中查找元素 *B* 是否唯一, 如果唯一表明单词 *A* 为相对非变位词, 将其添加到结果 `vector` 中, 最后对结果 `vector` 排序输出即可。

参考代码

```
int main(int argc, char *argv[]) {
```

^I 雅典娜, 古希腊神话中的智慧女神。

^{II} 乙烷, 一种无色无味的可燃气体, 分子式为 C₂H₆。

```

vector<string> allWords;
multiset<string> lowerCase;
// 读入数据。
string line;
while (cin >> line, line != "#") {
    allWords.push_back(line);
    // 利用库函数 transform 将单词中的大写字母转换为小写字母。
    transform(line.begin(), line.end(), line.begin(), ::tolower);
    sort(line.begin(), line.end());
    lowerCase.insert(line);
}
// 查找是否为相对非变位词。
vector<string> ananagrams;
for (int i = 0; i < allWords.size(); i++) {
    string word = allWords[i];
    transform(word.begin(), word.end(), word.begin(), ::tolower);
    sort(word.begin(), word.end());
    if (lowerCase.count(word) == 1) ananagrams.push_back(allWords[i]);
}
// 排序输出。
sort(ananagrams.begin(), ananagrams.end());
for (int i = 0; i < ananagrams.size(); i++) cout << ananagrams[i] << endl;
return 0;
}

```

强化练习: 246 10-20-30^C, [255](#) Correct Move^B, [261](#) The Window Property^D, 310 L-System^D, 409 Excuses Excuses^A, [489](#) Hangman Judge^A, [496](#) Simply Subsets^A, 665 False Coin^C, 1594 Ducci Sequence^B, 10391 Compound Words^A, [10393](#) The One-Handed Typist^C, 10415 Eb Alto Saxophone Player^B, 10686* SQF Problems^D, [10887](#) Concatenation of Languages^B, 10919 Prerequisites^A, 11063 B2-Sequence^A, [11136](#) Hoax or What^A, 11549 Calculator Conundrum^B, [11634](#) Generate Random Numbers^C, [11849](#) CD^A, 12049 Just Prune The List^B, 12394 Peer Review^D, [12641](#) Reodrnreig Lteetrs in Wrds^D, [12770](#) Palinagram^D, [13037](#) Chocolate^D, [13148](#) A Giveaway^B。

扩展练习: 215 Spreadsheet Calculator^D, 789 Indexing^D, [11348](#) Exhibition^C, 12096* The SetStack Computer^B。

2.8 位集

在解题中, 经常会遇到需要使用位运算的场合。C++中提供了多种位运算符, 可以实现不同的位操作。在使用二进制数时, 掌握以下的位操作技巧往往可以事半功倍。

```

int x = 19820624, b = 0, i = 6;
b = (x & (0x1 << i)) >> i; // 获取 x 的二进制表示的第 i 位 (最右侧为第 0 位, 下同)
x = x | (0x1 << i); // 将 x 的第 i 位设置为 1
x = x & (~0x1 << i); // 将 x 的第 i 位设置为 0
x = x ^ (0x1 << i); // 将 x 的第 i 位取反
b = x & (-x); // 获取 x 的二进制表示中从最右侧的 1 开始到末尾的所有二进制位

```

强化练习: 10469 To Carry or Not to Carry^A, 12614 Earn For Future^B, [12571](#) Brother & Sisters!^D, [12643](#)*

Tennis Rounds^C。

关于位运算，有三道有趣的题目，在此一并介绍^{I[17]}。

Single Number I

给定一个整数数组，其中除了一个元素只出现一次外，其他元素都出现两次，找出这个数字。要求算法具有 $O(n)$ 的时间复杂度且只使用常数量的额外内存空间。

分析

朴素的方法是使用 map 记录各个元素出现的次数，最后累计次数为 1 的数即为所求，但是此种方法需要额外的内存空间，而且需要先统计次数然后再找出次数为 1 的数，不够高效。使用位运算中的异或运算可以得到更为高效的解决方法。根据异或运算的定义，任意一个数和 0 进行异或运算的结果仍为自身，即

$$x = (x \wedge 0)$$

而任意一个数和自身进行异或运算的结果为 0，即

$$(x \wedge x) = 0$$

又由于异或运算满足结合律以及交换律，即

$$x \wedge (y \wedge z) = (x \wedge y) \wedge z, (x \wedge y) = (y \wedge x)$$

那么不妨设数组的元素为

$$A_1, C_2, B_1, A_2, X_1, B_2, C_1, \dots$$

其中 X_1 是只出现一次的元素，其他均为出现两次的元素，由于异或运算满足交换律，对所有元素进行异或操作，可得

$$A_1 \wedge C_2 \wedge B_1 \wedge A_2 \wedge X_1 \wedge B_2 \wedge C_1 \wedge \dots = A_1 \wedge A_2 \wedge B_1 \wedge B_2 \wedge C_1 \wedge C_2 \wedge \dots \wedge X_1 = 0 \wedge X_1 = X_1$$

也就是说，从数组的第一个元素一直异或下去，最后得到的结果恰为只出现一次的数字。

Single Number II

给定一个整数数组，除了一个元素只出现一次外，其他元素都出现三次，找出这个数字。要求算法具有 $O(n)$ 的时间复杂度且只使用常数量的额外内存空间。

分析

沿用前述解决 Single Number I 的思路似乎行不通，需要转换一下思维角度。因为其他数是出现三次的，也就是说，对于每一个二进制位，如果只出现一次的数在该二进制位为 1，那么这个二进制位在全部数字中所出现的次数就无法被 3 整除。因此，将每一个数字按位累加，然后将最后结果每一位上的 1 出现的次数对 3 取模，剩下的就是结果。

Single Number III

给定一个整数数组，其中有两个元素只出现一次，其他元素都出现两次，找出只出现一次的两个数。要求算法具有 $O(n)$ 的时间复杂度且只使用常数量的额外内存空间。

分析

可以沿用解决 Single Number I 的基本思路进行解题。如果能将数组分成两个部分，每个部分里只有一个元素出现一次，其余元素都出现两次，那么使用解决 Single Number I 的方法就可以找出这两个元素。不

^I 三道题目均源自 LeetCode (<https://leetcode.com/>)，相应的题目编号依次为 136、137、260。

妨假设只出现一次的两个元素是 X 和 Y ，那么最终所有的元素异或的结果就是 $R=X\wedge Y$ ，并且 $R\neq 0$ 。那么我们可以找出 R 的二进制表示中为 1 的某一位，对于原来的数组，可以根据某个数此二进制位是否为 1 将其划分到两个子数组中。最后， X 和 Y 必定在不同的两个子数组中。而且对于其他成对出现的元素，要么在 X 所在的子数组，要么在 Y 所在的子数组。到此为止，继续使用解决 Single Number I 的方法即可找出两个只出现一次的数字。

强化练习：1241 Jollybee Tournament^C，[10264 The Most Potent Corner^B](#)，11173 Grey Codes^B，[11933 Splitting Numbers^A](#)。

扩展练习：11567 Moliu Number Generator^C。

位集（bitset）是一种序列容器，不过与 `vector` 等序列容器不同，它专门用来存储一连串的位（bit）数据。

使用位集需要包含头文件`<bitset>`，以下列出了位集的若干常用属性和方法。

[index]	使用数组下标的方式获取序号为 <code>index</code> 的位的值或引用。
<code>count()</code>	返回位集中被设置为 1 的位的数量。
<code>size()</code>	返回位集的大小。
<code>any()</code>	测试位集中是否至少有一个位为 1，是则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>none()</code>	测试位集中是否所有位均为 0，是则返回 <code>true</code> ，否则返回 <code>false</code> 。
<code>all()^{c++11}</code>	测试是否所有位均被设为 1。
<code>to_string()</code>	将位集转换为 <code>string</code> 类实例。
<code>to_ulong()</code>	将位集所表示的值转换为 <code>unsigned long</code> 整数值。
<code>to_ullong()</code>	将位集所表示的值转换为 <code>unsigned long long</code> 整数值。
 set	
设置位集所有位或者指定位的值，默认将位设为 1。序号从 0 开始计数。	
<code>bitset& set();</code>	
<code>bitset& set(size_t pos, bool val = true);</code>	
 test	
测试位集指定位是否设为 1，是则返回 <code>true</code> ，否则返回 <code>false</code> 。	
<code>bool test(size_t pos) const;</code>	
 reset	
重设位集所有位或者指定位的值为 0。序号从 0 开始计数。	
<code>bitset& reset();</code>	
<code>bitset& reset(size_t pos);</code>	
 flip	
反转位集所有位或者指定位的值。值为 1 的变为 0，值为 0 的变为 1。	
<code>bitset& flip();</code>	
<code>bitset& flip(size_t pos);</code>	

位集的大小使用放在一对尖括号内的整数值进行指定。可以在声明的同时使用整数或者只包含‘0’或‘1’的字符串进行初始化，如果使用包含非‘0’和‘1’字符的字符串进行初始化会发生运行时错误。

```
//-----2.8.cpp-----//
int main(int argc, char *argv[]) {
    bitset<16> empty, some(64), integer(0x64);
    bitset<16> other(string("1010101010101010"));
    cout << empty << endl;
```

```

    cout << some << endl;
    cout << some.to_string() << endl;
    cout << integer << endl;
    cout << other.to_ulong() << endl;
    return 0;
}
//-----2.8.cpp-----//

```

输出为：

```

0000000000000000
0000000010000000
0000000010000000
000000001100100
43690

```

10718 Bit Mask^B (位掩码)

给定一个 32 位无符号整数 N ，寻找整数 M 使得 $L \leq M \leq U$ ，且 $N \text{ OR } M$ 的值最大，**OR** 表示“二进制或”运算。如果有多个 M 满足条件，输出最小的 M 。

输入

每行输入包含 3 个整数， N ， L ， U ， $L \leq U$ ，输入以文件结束符作为结束标志。

输出

对于每组输入，输出最小的 M ，使得 $L \leq M \leq U$ 且 $N \text{ OR } M$ 的值最大。

样例输入

```
100 50 60
```

样例输出

```
59
```

分析

直观地，将 N 表示为二进制后，从高位开始，如果位为 0，将其置为 1 后所获得的值也越大，因此掩码 M 应该尽可能让 N 的二进制表示中高位为 0 的位反转为 1。那么可以从 N 的二进制表示中最高有效位开始考虑 M 的二进制位值。以样例输入为例，此时 $N=100$ ，因其表示为 32 位二进制数后前 3 个字节全为 0，若将其中任意一个位反转为 1，则对应的 M 将大于 60，不符合约束条件，故取 N 的二进制表示末尾的 8 个二进制位—— 01100100_2 ——进行分析。从最高位 0 开始，如果将此位反转为 1，则要求 M 至少为 $10000000_2 = 128 > 60$ ，不满足要求，故此位不能进行反转。接着看第二位 1，此位已经为 1，不需反转，但是 M 中此二进制位是否也可为 0 呢？如果 M 此二进制位设置为 1 后所得到的值不大于 L ，那么应该将此二进制位设置为 1，否则即使将 M 后续的所有二进制位全部设置为 1 也无法得到大于等于 L 的数。继续沿上述思路确定后续 M 各二进制位的取值，最终可得 M 的最小二进制表示为 $00111011_2 = 59$ 。

参考代码

```

int main(int argc, char *argv[]) {
    unsigned N, L, U;
    while (cin >> N >> L >> U) {
        bitset<32> NN(N), M(0);
        for (int i = 31; i >= 0; i--) {
            M.set(i, 1);
            if (NN.test(i)) {
                if (M.to_ulong() > L) M.set(i, 0);
            }
        }
    }
}

```

```

    }
    else {
        if (M.to_ulong() > U) M.set(i, 0);
    }
}
cout << M.to_ulong() << '\n';
}
return 0;
}

```

强化练习：[213 Message Decoding^B](#)，[446 Kibbles "n" Bits "n" Bits "n" Bits^A](#)，[565 Pizza Anyone^C](#)，[740 Baudot Data Communication Code^A](#)，[10019 Funny Encryption Method^A](#)，[10227* Forests^B](#)，[10931 Parity^A](#)，[11744 Parallel Carry Adder^D](#)，[11926 Multitasking^B](#)。

扩展练习：[10666* The Eurocup is Here^D](#)，[11532 Simple Adjacency Maximization^C](#)。

2.9 链表

链表（list）也是一种容器，但与向量有所不同，链表不能使用下标形式对容器中的元素进行随机访问，而只能通过迭代器的方式进行访问。链表最大的优点是可以高效地完成元素的插入和删除操作，其时间复杂度为 $O(1)$ ，相对于向量要高，适用于需要频繁进行插入删除操作的应用场合，其缺点是占用的内存较大。

使用链表需要包含头文件`<list>`，以下列出了链表的若干常用属性和方法。

begin()	返回指向链表首个元素的迭代器。
end()	返回指向链表最末元素之后一个位置的迭代器。
rbegin()	返回指向链表最末元素的逆序迭代器。
rend()	返回指向链表首个元素之前一个位置的逆序迭代器。
size()	返回链表的大小。
empty()	测试链表是否为空，为空返回 <code>true</code> ，否则返回 <code>false</code> 。
front()	获取链表的首个元素。
back()	获取链表的最末元素。
clear()	清空链表。
pop_back()	删除链表末尾元素。
pop_front()	删除链表首个元素。
 push_back	
将指定元素添加到链表末尾。	
void push_back(const value_type& value);	
 push_front	
将指定元素添加到链表起始。	
void push_front(const value_type& value);	
 insert	
在指定位置之前插入单个或一组元素。需要使用迭代器指定位置参数。返回插入元素位置的迭代器。	
iterator insert(const_iterator position, const value_type& value);	
 erase	
删除指定位置处或指定范围内的元素，并返回删除元素后的迭代器。需要使用迭代器指定位置参数。	
iterator erase(const_iterator position);	
 sort	

对链表中的元素进行排序。

```
void sort();
template <class Compare> void sort(Compare cmp);
```

unique

移除链表中的重复元素。

```
void unique();
template <class BinaryPredicate> void unique(BinaryPredicate binary_pred);
```

除了双向链表外, C++标准库还提供了单向链表 `forward_list`。双向链表可以进行双向遍历, 但 `forward_list` 只能进行前向遍历, 可以应用于一些只需顺序遍历的操作场合。使用 `forward_list` 需要包含头文件`<forward_list>`。

强化练习: [245 Uncompress^C](#), [289 A Very Nasty Text Formatter^E](#), [520 Append^D](#), [11988 Broken Keyboard \(a.k.a. Beiju Text\)^A](#), [12657 Boxes in a Line^B](#)。

扩展练习: [12108 Extraordinarily Tired Students^C](#)。

2.10 二叉树

二叉树 (binary tree) 是一种常见的数据结构, 不过标准模板库中并未有对应的表示^[18]。可能是因为二叉树的应用较为灵活, 不太容易使用一个固定的功能集对其进行表示, 而且利用标准库所提供的其他数据结构可以轻松构建应用所需的二叉树, 因此并不需要在“基础”的标准库中予以实现。

每棵非空二叉树都有一个根结点 (root node)^I, 非根结点则属于某个其他结点的子结点 (child node), 二叉树中的结点最多包含两个子结点, 分别称为左子结点 (left child node) 和右子结点 (right child node), 子结点数为零的结点称为叶结点 (leaf node)。

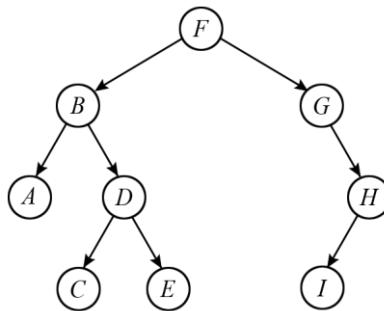


图 2-4 一棵二叉树。结点 F 为根结点, B 为 F 的左子结点, G 为 F 的右子结点, A、C、E、I 均为叶结点

值得一提的是, 对于二叉树的某些概念, 国内和国际通行的定义并不完全一致。例如满二叉树 (full binary tree) 的概念。满二叉树, 国内某些教材的定义是“一棵深度为 k 且有 $2^k - 1$ 个结点的二叉树”^[19], 即除了树的最后一层全部为叶子结点以外, 其他层的结点均有左右子结点的二叉树。但国际通行的满二叉树

^I 结点所对应的英文单词为“node”, 有些书籍将其翻译为“节点”, 似有不妥。“节”表示分段之间连接的部分, 例如“竹节”, 而“结”表示交结于一处, 例如“绳结”, 对于树这种数据结构, 将“node”翻译为“结点”比“节点”更为恰当。查询“全国科学技术名词审定委员会”官方网站 (<http://www.cnctst.cn/>, 2020) 上的“术语在线” (<http://www.termonline.cn/index.htm>, 2020), 在“计算机科学技术”学科分类中, “node”的审定翻译亦为“结点”。

(strict binary tree, 又称严格二叉树) 定义是“任意结点的子结点数要么为 0, 要么为 2 的二叉树”, 如图 2-5 所示。

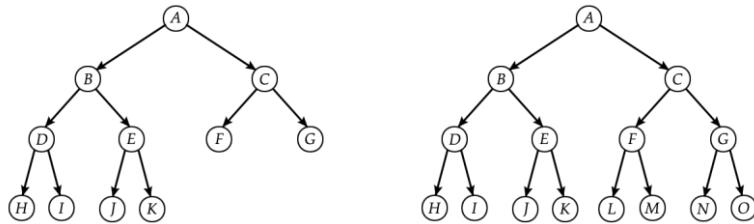


图 2-5 左侧图例为国际通行定义的满二叉树 (严格二叉树); 右侧图例为国内定义的满二叉树, 对应国际通行定义的完美二叉树

国内定义的满二叉树实际上对应着国际通行定义的完美二叉树 (perfect binary tree) —— “一棵高度为 h 的二叉树, 所有叶结点深度均为 h 且其他非叶结点均有左右子结点”, 但国内教材一般很少提到这个概念。在本书中, 当涉及到二叉树的有关概念时均采用国际通行的定义。

强化练习: 615 Is It a Tree^A。

在实际应用中, 一般都是采用结构体来表示树结点, 通过指针将二叉树中结点的相互关系使用类似于链表的形式予以表示。

```
//++++++++++++++2.10.1.cpp+++++++++++++++
struct TreeNode {
    // 结点的标记和权值。
    int id, weight;
    // 父结点、左右子树的指针。
    TreeNode *parent, *leftChild, *rightChild;
};
```

对二叉树进行遍历操作, 常用的有两种方式, 一种是深度优先遍历 (depth-first order traversal), 另外一种是广度优先遍历 (breadth-first order traversal)。深度优先遍历又分为三种, 分别称为前序遍历 (preorder traversal)、中序遍历 (inorder traversal)、后序遍历 (postorder traversal)。

- (1) 前序遍历: 先访问根结点, 然后前序遍历左子树, 最后前序遍历右子树;
- (2) 中序遍历: 先中序遍历左子树, 然后访问根, 最后中序遍历右子树;
- (3) 后序遍历: 先后序遍历左子树, 然后后序遍历右子树, 最后访问根。

可以看到, 三种遍历顺序的差别仅在于访问根结点的次序, 且遍历的过程均包含递归。还有一种遍历是按照结点深度进行遍历, 称之为层序遍历 (level-order traversal), 此种遍历是将广度优先遍历应用在二叉树上的结果。需要注意, 在大多数解题应用中, 遍历的顺序选择和具体的题目要求有关。

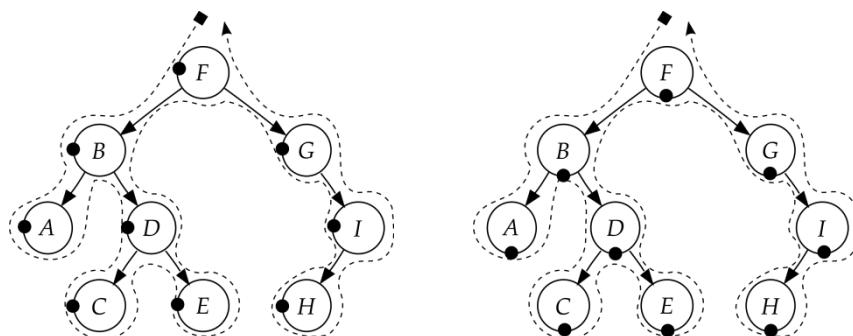


图 2-6 前序遍历 (FBADCEGIH) 和中序遍历 (ABCDEFGHI)

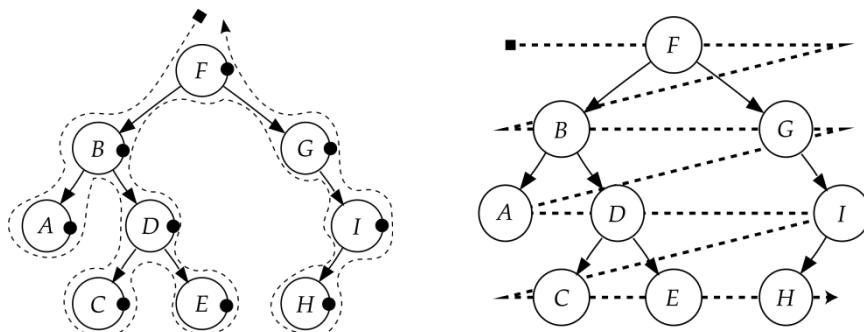


图 2-7 后序遍历 (ACEDBHGIF) 和层序遍历 (FBGADICEH)

```

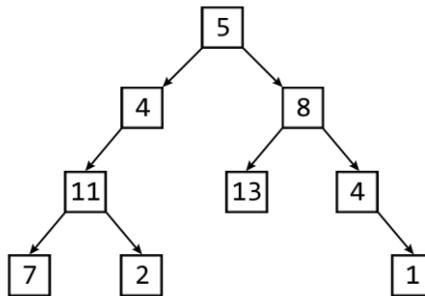
// 使用指向根的指针进行前序遍历。
void preorderTraversal(TreeNode* root) {
    if (root == NULL) return;
    cout << ' ' << root->id;
    preorderTraversal(root->leftChild);
    preorderTraversal(root->rightChild);
}

// 使用指向根的指针进行中序遍历。
void inorderTraversal(TreeNode* root) {
    if (root == NULL) return;
    inorderTraversal(root->leftChild);
    cout << ' ' << root->id;
    inorderTraversal(root->rightChild);
}

// 使用指向根的指针进行后序遍历。
void postorderTraversal(TreeNode* root) {
    if (root == NULL) return;
    postorderTraversal(root->leftChild);
    postorderTraversal(root->rightChild);
    cout << ' ' << root->id;
}
//+++++++++++++2.10.1.cpp+++++++++++++
  
```

112 Tree Summing^A (路径求和)

给定一棵二叉树，树结点中存储有一个整数，编写程序确定从树根到叶结点的路径中，路径上各结点的“整数和”是否与某个指定的整数相等。例如，在下面的树中，共有 4 条从树根到叶结点的路径，各路径上结点的整数和分别为 27, 22, 26, 18。



在输入中给出的二叉树以 LISP 语言的 S 表达式予以表示，它具有以下形式：

空树 ::= ()
树 ::= 空树 (整数 树 树)

上图所给出的树可使用 S 表达式表示为：

(5(4(11(7())())(2())())())(8(13())())(4()(1())()))))
--

注意，所有的叶结点均具有以下形式：

(整数())()

由于空树不存在任何从树根到叶结点的路径，因此对空树进行上述查询时，应该输出否定的答案。

输入

输入包含多组测试数据，每组测试数据包括一个整数，后跟一个或多个空格，接着是用 S 表达式表示的二叉树结构。给定二叉树所对应的 S 表达式均是合法的，但表达式可能跨行且包含数量不定的空格。每个输入文件可能包含一组或多组测试数据，输入以文件结束符表示结束。

输出

为输入文件中的每一个测试用例（整数/树）输出一行。对于每组“整数/树”中的数值 I 和 T (I 表示整数， T 表示树)，如果存在从树根到叶结点的“路径和”等于 I ，输出“yes”，否则输出“no”。

样例输入

22 (5(4(11(7())())(2())())()) (8(13())())(4()(1())()))))
20 (5(4(11(7())())(2())())()) (8(13())())(4()(1())()))))
10 (3
(2 (4 () ()))
(8 () () ()))
(1 (6 () ()))
(4 () () ())))
5 ()

样例输出

yes
no
yes
no

分析

使用结构体和指针来表示树，若使用数组，当树的深度较大时，会导致数组过大而无法存储。解题的一

一个关键是如何将输入解析为树，可以借助 `cin.putback()` 并结合递归完成输入的解析。将输入解析成二叉树后，剩下的问题就是如何通过树遍历求“路径和”，可以通过在遍历时累加所经过的结点值并将“路径和”保存在叶结点中来实现。

关键代码

```
// 利用递归和 cin.putback()，将输入解析为链表形式的树。
void parse(TreeNode *node) {
    bool isLeaf = false;
    // 忽略空白字符。
    char c;
    while (cin >> c, c != '(') { }
    cin >> c;
    // 需要考虑输入中的整数为负数的情形。
    if (isdigit(c) || c == '-') {
        int sign = (c == '-' ? (-1) : 1), number = 0;
        if (isdigit(c)) number = c - '0';
        while (cin >> c, isdigit(c)) number *= 10, number += (c - '0');
        cin.putback(c);
        node->weight = number * sign;
    } else {
        // 当前字符是括号，需要将其送回输入流，以保证后续能够正确解析。
        cin.putback(c);
        // 若当前结点为空，则将父结点的相应子结点设置为空。
        if (node->parent != NULL) {
            if (node == node->parent->leftChild) node->parent->leftChild = NULL;
            else node->parent->rightChild = NULL;
        } else empty = true;
        // 表示当前结点是一个叶结点。
        isLeaf = true;
    }
    // 如果当前结点为非叶结点则继续递归解析。
    if (!isLeaf) {
        // 解析左子树。
        TreeNode *left = new TreeNode;
        node->leftChild = left;
        left->parent = node;
        parse(left);
        // 解析右子树。
        TreeNode *right = new TreeNode;
        node->rightChild = right;
        right->parent = node;
        parse(right);
    }
    // 忽略空白字符。
    while (cin >> c, c != ')') { }
}
```

强化练习：115 Climbing Trees^B，122 Trees on the Level^A，297 Quadtrees^A，699 The Falling Leaves^A，[1738* Ceiling Function^D](#)，11108 Tautology^C，[12347 Binary Search Tree^B](#)。

扩展练习：839 Not so Mobile^A，939 Genes^D，[11147 KuPellaKeS BST^D](#)，11234 Expressions^B。

如果给定的是一棵满二叉树，而且树的深度不大（例如，深度小于 20），那么可以使用数组来表示整棵树，其效率也很高。方法是设立一个一维数组 `tree`，以元素 `tree[i]` 表示父结点，元素 `tree[2i+1]` 和 `tree[2i+2]`

作为其左右子结点¹，整棵树的根结点为 $tree[0]$ 。这样深度为 d 的满二叉树可以使用大小为 $2^d - 1$ 的一维数组进行表示。

强化练习：[679 Dropping Balls^A](#)，[712 S-Trees^B](#)，[11615 Family Tree^D](#)。

在某些情况下，如果遍历结果中不包含重复的结点标记，给定三种遍历方式结果中的两种，可以根据遍历的特点来确定第三种遍历方式的结果。常见的是给定前序遍历和中序遍历结果要求确定后序遍历结果，或者给定后序遍历和中序遍历结果要求确定前序遍历结果。如果给定前序遍历和后序遍历结果，则无法唯一确定中序遍历结果，因为无法唯一地确定左右子树。

如图 2-8 所示的二叉树，其前序遍历结果为： $FBADCEGHI$ ，中序遍历结果为： $ABCDEFGIH$ ，后序遍历结果为： $ACEDBIHGF$ 。

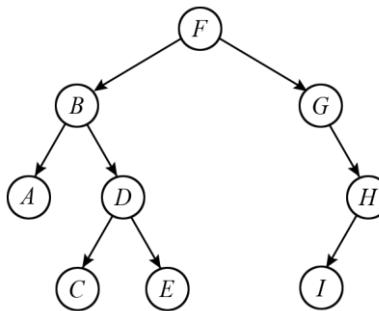


图 2-8 具有 9 个结点的二叉树

下面我们来看看，如何根据前序遍历和中序遍历的结果推导出树的结构，进而得到后序遍历的结果。已知前序遍历结果为 $FBADCEGHI$ ，根据前序遍历的特点——“第一个访问的结点为整棵树的根”，可以知道整棵树的根结点为 F ，结合中序遍历结果 $ABCDEFGIH$ ，可以推导出 F 左侧的 $ABCDE$ 必然是根的左子树， F 右侧的 GHI 必然是根的右子树：

$$[F] BADCEGHI \rightarrow (ABCDE) [F] (GHI)$$

由于不管是前序遍历还是中序遍历，左右子树遍历结果的长度是不变的，当前已经知道中序遍历时左子树为 $ABCDE$ ，长度为 5，右子树为 GHI ，长度为 3，那么可以根据长度确定前序遍历中 $BADCE$ 是左子树的遍历结果，而 GHI 为右子树的遍历结果：

$$(ABCDE) [F] (GHI) \rightarrow [F] (BADCE) (GHI)$$

对于左子树来说，其根结点是 F 的左儿子，而在前序遍历中， F 的左子结点在遍历结果中一定是紧随 F 之后的，由遍历结果 $FBADCEGHI$ 可知左子树的根结点为 B ，结合中序遍历左子树的结果 $ABCDE$ ，可以推导出 A 是 B 的左子树中序遍历结果， CDE 是 B 的右子树中序遍历结果……

$$[F] (BADCE) (GHI) \rightarrow ((A) [B] (CDE)) [F] (GHI)$$

同样的，可以按照类似方法推导出 F 的右子树结构。在前序遍历中，先是遍历完左子树，然后才开始遍历右子树，则开始遍历右子树时的第一个结点即为右子树的根，那么由前序遍历右子树的结果 GHI 可知， G 是右子树的根结点，进而由中序遍历结果 GHI 可知， G 无左子树， G 的右子树中序遍历结果为 HI ……

¹ 此处以序号为 0 的数组元素作为满二叉树的根结点，左右儿子结点所对应的数组元素序号分别为 $2i+1$ 和 $2i+2$ 。如果以序号为 1 的数组元素作为满二叉树的根结点，则左右儿子结点所对应的数组元素序号分别为 $2i$ 和 $2i+1$ 。

$$[F] ([B](A)(DCE)) (GIH) \rightarrow ((A) [B] (CDE)) [F] ([G] (HI))$$

根据以上叙述不难发现推导的过程是递归的，即先找到当前树的根结点，然后划分为左、右子树，此为一个求解步骤，接着先进入左子树重复求解步骤，之后对右子树重复求解步骤，最后就可以还原整棵二叉树。求解过程，可以从逻辑上划分为四个步骤：(1) 确定根结点、左子树、右子树；(2) 在左子树中递归；(3) 在右子树中递归；(4) 输出根结点。

```
-----2.10.2.cpp-----
// 根据前序遍历和中序遍历结果输出后序遍历结果。
void postorder(string preorder, string inorder) {
    // 递归出口：前序遍历结果为空。
    if (preorder.length() == 0) return;
    // 找到根结点。
    int root = 0;
    for (; root < inorder.length(); root++)
        if (inorder[root] == preorder.front())
            break;
    // 由根结点确定左子树，在左子树中递归。
    postorder(preorder.substr(1, root), inorder.substr(0, root));
    // 由根结点确定右子树，在右子树中递归。
    postorder(preorder.substr(root + 1), inorder.substr(root + 1));
    // 输出根。
    cout << preorder.front();
}
-----2.10.2.cpp-----
```

类似的，也可以由后序遍历和中序遍历结果确定前序遍历结果，只不过在确定根结点时需要从后序遍历结果的末尾开始划分左右子树。

```
-----2.10.3.cpp-----
// 根据后序遍历和中序遍历结果输出前序遍历结果。
void preorder(string postorder, string inorder) {
    // 递归出口：后序遍历结果为空。
    if (postorder.length() == 0) return;
    // 找到根结点。
    int root = 0;
    for (; root < inorder.length(); root++)
        if (inorder[root] == postorder.back())
            break;
    // 输出根。
    cout << postorder.back();
    // 由根结点确定左子树，在左子树中递归。
    preorder(postorder.substr(0, root), inorder.substr(0, root));
    // 由根结点确定右子树，在右子树中递归。
    preorder(postorder.substr(root, postorder.length() - root - 1),
            inorder.substr(root + 1));
}
-----2.10.3.cpp-----
```

强化练习：372 WhatFix Notation^E，536 Tree Recovery^A，548 Tree^B，10701 Pre In and Post^A。

2.11 线段树

给定一个无序数组 A ，要求找出在给定序号区间 $[i, j]$ 内具有最小值的元素序号。朴素的做法是顺序扫

描区间 $[i, j]$ 内的所有元素，通过反复比较以确定具有最小值的元素序号。当区间范围很大或者查询非常频繁时，这一做法显然效率不高。虽然可以在 $O(n^2)$ 的时间内，通过为每种可能的区间生成具有最小值的元素序号的方式对数组进行预处理以提高查询速度，但是无法满足后续更新数组元素的要求，因为每次更新数组元素后，原先预处理的结果会失效，需要耗费 $O(n^2)$ 的时间再次进行预处理。类似于这种查询范围内元素最大（小）值（Range Maximum/Minimum Query, RMQ）或者查询范围内元素和（Range Sum Query, RSQ）的问题，它们都有一个共同的特点——最后区间的結果可以由多个相邻的连续区间合并的結果来获得。针对这个特点，可以使用多种巧妙的数据结构来解决 RMQ/RSQ 问题^[20]。

线段树（segment tree）是一种以二叉树为基础的数据结构，可以用于进行高效的范围最大（小）值查询、范围和查询等。利用二叉树的特点，可以将初始的查询范围逐次二分，最后由一系列的相邻的区间合并起来构成初始的查询范围。由于初始查询范围的結果可以由相邻的区间的結果合并而来，故只需反复合并相邻两个区间的查询結果即可得到最后所需要的结果，这样时间复杂度可以降低到 $O(\log n)$ 。下面以范围最大值查询为例，介绍线段树的使用。

创建

为了简便，可以使用一维数组来表示线段树。二叉树的根结点对应序号为 0 的数组元素，如果某个非叶结点对应序号为 p 的数组元素，则其左右子结点所对应的数组元素序号分别为 $2p+1$ 和 $2p+2$ 。由于结点所记录的信息各式各样，对应的数组元素既可以是内置数据类型，也可以是结构体。在声明结构体时，为了记录待查询的信息，需要设置必要的域来存储相应的值。由于每个非叶子结点都具有左右两个子结点（尽管这两个子结点在具体应用中可能并不会使用），而区间内的每个元素都需要有一个叶结点对应，则应该以完美二叉树的形式来创建线段树，因此在声明结点的数量时要考虑到内部结点的数量。令区间长度为 L ，整数 N 是满足 $2^N \geq L$ 的最小整数，则数组的大小至少应为 $2^{N+1}-1$ 。例如，如果需要查询的区间为 $[0, 1000]$ ，则 N 为 10，即完美二叉树的叶子结点至少应该有 1024 个，加上根结点和内部结点数量，总共的结点数为 2047 个。在声明空间时，一般以查询区间长度的 4 倍来申请存储空间较为“安全”。

```
//+++++++
// 2.11.0.1.cpp
const int MAXN = 1000010, INF = 0x7f7f7f7f;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

int data[MAXN];
struct node { int field; } st[4 * MAXN];

// 在创建线段树的同时进行预处理，获取区间的最大值。
void pushUp(int p) {
    st[p].field = max(st[LC(p)].field, st[RC(p)].field);
}

// 递归创建线段树。设数组长度为 n，则调用方式为 build(0, 0, n - 1)。
void build(int p, int left, int right) {
    if (left == right) st[p].field = data[left];
    else {
        // 将给定区间二分，递归创建。
        int middle = (left + right) >> 1;
        build(LC(p), left, middle);
        build(RC(p), middle + 1, right);
        pushUp(p);
    }
}
```

}

在上述代码中，`pushUp` 方法的作用是将父结点更新的操作抽取出，便于适应各种应用场景的需要。例如，求最大值时可使用 `max`，求最小值可用 `min`。需要注意的是，在创建线段树时，叶结点所表示的区间是可以调整的。一般情况下，叶结点表示的区间长度为 0，即数轴上的一个点，可以进行适当更改以使其表示一个长度大于 1 的区间，这样在某些解题应用中更为方便。

查询

线段树创建完毕后即可对其进行查询。查询一般是给出一个区间，要求确定此区间内的最大（小）值或者此区间内的元素和。由于在创建线段树时，已经进行了预处理，得到了结点所表示的区间最大值，故只需不断将查询区间进行二分，反复将各个子区间的最大值结果合并即可获得最终结果。

```

int query(int p, int left, int right, int qleft, int qright) {
    // 当查询区间未落在结点所表示的区间范围内时, 返回一个“哨兵”值。
    if (left > qright || right < qleft) return -INF;
    if (left >= qleft && right <= qright) return st[p].field;
    int middle = (left + right) >> 1;
    int q1 = query(LC(p), left, middle, qleft, qright);
    int q2 = query(RC(p), middle + 1, right, qleft, qright);
    return max(q1, q2);
}

```

此处需要注意的是，当查询区间与结点所表示的区间不重叠时，返回值的处理应该根据具体应用相应调整。若查询的是范围和，则 `INF` 应为 0；若查询的是最大值，则 `INF` 应该设置为一个应用中不会小于的负数值，即此应用中的一个“无限小”值；若查询的是最小值，则返回的 `INF` 所代表的是一个应用中不会超过的值，即此应用中的一个“无限大”值。更为稳妥的方法是返回具有最大（小）值元素在原数组中的序号，当区间不重叠时返回特殊标记值 -1，这样可以避免 `INF` 值设置不合理可能导致的问题。

更新

如果更新的是单个数组元素，即只涉及线段树中的一个叶结点，称之为单结点更新，如果更新范围为一个区间，即涉及多个叶结点，则称之为区间更新。单结点更新是在线段树中找到原始数据序号为指定值的叶结点，将其值进行更新，同时更新该叶结点的所有祖先结点的值。

```
void update(int p, int left, int right, int index, int value) {
    if (left == right) st[p].field = value;
    else {
        int middle = (left + right) >> 1;
        if (index <= middle)
            update(LC(p), left, middle, index, value);
        else
            update(RC(p), middle + 1, right, index, value);
        pushUp(p);
    }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++2.11.0.1.cpp+++++++++++++++++++++++++++++++++//
```

延迟更新

在对线段树进行更新后，内部结点的值一般都需要做相应的改变，但是每次在更新后都立即对内部结点进行一次更新，这样会导致更新效率退化为 $O(n)$ 。如果在必须更新时才对内部结点进行更新，可以使效率

仍保持为 $O(\log n)$ 。为了实现这种效果, 可以采用延迟标记 (lazy tag) 技巧。应用延迟标记进行更新的方法称为延迟更新 (lazy propagation)。延迟标记是在表示结点信息的结构体中增加一个域, 表示当前结点累积的更新量, 当某个查询需要访问此结点的左右子结点时, 将此累积更新量应用到当前结点上, 并向其左右子结点传递, 最后将结点的累积更新量“清零”。这样做, 可以尽量减少结点的总更新数量, 获得更高的效率。在进行区间更新时, 可能每次都需要对较多的结点进行更新, 应用延迟更新技巧较为合适。在具体实现时, 为了记录累积更新量, 需要对结构体进行适当更改, 同时相应的查询和更新过程也要做适当的修改, 并在创建线段树的时候对初始累积更新量进行设置。需要注意, 应用延迟更新的条件是——更新可以叠加, 例如将区间内的元素加上或减去一个数值。如果更新操作是不可叠加的, 则每次更新必须到达区间的所有叶结点, 在这种情况下, 应用延迟标记无助于效率的提高。

```
-----2.11.0.2.cpp-----
struct node { int field, tag; } st[4 * MAXN];

// 应用延迟标记的线段树创建。
void build(int p, int left, int right) {
    // 在创建线段树时设置初始更新累积量为 0, 表示此结点不需向其左右子结点传递更新量。
    if (left == right) st[p].field = data[left], st[p].tag = 0;
    // 其他创建线段树的代码与前述相同。
}

// 根据延迟标记对线段树的结点做相应的更改。
void commit(int p, int ctag) {
    // 可能结点上存在多次的延迟更新, 所以要叠加。
    st[p].tag += ctag;
    st[p].field += st[p].tag;
}

// 将延迟标记向左右子结点传递。
void pushDown(int p) {
    if (st[p].tag) {
        commit(LC(p), st[p].tag);
        commit(RC(p), st[p].tag);
        st[p].tag = 0;
    }
}

// 应用延迟标记的查询。
int query(int p, int left, int right, int qleft, int qright) {
    if (left > qright || right < qleft) return -INF;
    if (left >= qleft && right <= qright) return st[p].field;
    // 在查询左右子结点之前需要将延迟标记向下传递。
    pushDown(p);
    int middle = (left + right) >> 1;
    int q1 = query(LC(p), left, middle, qleft, qright);
    int q2 = query(RC(p), middle + 1, right, qleft, qright);
    return max(q1, q2);
}

// 应用延迟标记的更新, 将区间内的所有元素改变 utag 所指定的值。
void update(int p, int left, int right, intuleft, inturight, intutag) {
    if (left > uright || right <uleft) return;
    if (left >=uleft && right <=uright) commit(p, utag);
```

```

    else {
        // 在更新左右子结点之前需要将延迟标记向下传递。
        pushDown(p);
        int middle = (left + right) >> 1;
        update(LC(p), left, middle,uleft, uright, utag);
        update(RC(p), middle + 1, right,uleft, uright, utag);
        pushUp(p);
    }
}
//-----2.11.0.2.cpp-----

```

从上述区间更新的延迟标记实现可以看出，单结点更新实际上可以作为区间更新的一个特例来看待。

2.11.1 线段树的应用

由于可以在线段树的结点中记录各种信息，线段树除了用于高效地进行 RMQ/RSQ 操作外，还可以对其实灵活改变加以巧妙地运用¹。

(1) 给定一个正整数数组 A ，查询给定区间 $[l, r]$ 内整数的最大公约数或最小公倍数。在求最大公约数时，可以将数组分成若干部分，分别求出这若干部分的最大公约数，然后再对这若干个最大公约数再求一次最大公约数，所得的结果即为整个数组的最大公约数。类似的，求最小公倍数也可以这样操作。上述过程符合线段树应用的条件，即可以将若干个子区间的解进行合并以得到原区间的解。因此，类似于 RMQ，可以先建立线段树，结点中存储的是左右子结点的最大公约数或最小公倍数，时间复杂度为 $O(n \log n)$ 。当进行查询时，采用类似于区间最大值查询的方法即可，时间复杂度为 $O(\log n)$ 。

(2) 给定一个整数数组 A ，查询给定区间 $[l, r]$ 内值为 0 的元素的个数以及查询数组 A 中第 k 个为 0 的元素的序号。查询给定区间内 0 的个数，可以采用类似于区间和查询的做法，线段树结点中存储的是区间内 0 的个数。查询第 k 个 0 元素的序号，可以从根结点开始，如果根结点 0 元素的个数小于 k ，则不存在第 k 个为 0 的元素，否则比较根结点的左右儿子结点，如果左儿子结点的 0 元素个数大于等于 k ，则第 k 个为 0 的元素在根结点的左侧子树，否则在右侧子树，然后递归查询即可。注意，如果第 k 个为 0 的元素在右侧子树，需要将 k 减去左侧子树为 0 的元素个数，然后再继续查询。

```

//-----2.11.1.2.cpp-----
// st[p].zeros 存储的是结点 p 所对应的区间内 0 元素的个数。
int findKthZero(int p, int left, int right, int k) {
    if (k > st[p].zeros) return -1;
    if (left == right) return left;
    int middle = (left + right) >> 1;
    if (st[LC[p]].zeros >= k)
        return findKthZero(LC(p), left, middle, k);
    else
        return findKthZero(RC(p), middle + 1, right, k - st[LC(p)].zeros);
}
//-----2.11.1.2.cpp-----

```

(3) 给定一个非负整数数组 A 及整数 x ，要求确定一个最小的序号 i ，使得数组 A 中的前 i 个元素之和不小于 x 。由于给定的是非负整数数组，则可以先求得数组的前缀和数组，由于前缀和数组是不递减的，因此可以使用二分查找来确定所求的最小序号 i ，此方法的查询时间复杂度为 $O(\log n)$ ，但如果涉及数组元素

¹ 参阅：https://cp-algorithms.com/data_structures/segment_tree.html, 2020。

的更新操作，每次均需要 $O(n)$ 的时间复杂度重新构建前缀和数组。为了使数组元素更新操作的时间复杂度降低到 $O(\log n)$ ，可以使用类似于区间和查询的方法，先建立线段树，线段树的结点存储的是对应区间内元素的和，之后利用二分查找来确定序号 i ，使得区间 $[1, i]$ 内的元素和不小于 x 。查询区间 $[1, i]$ 的元素和可以利用线段树使得时间复杂度为 $O(\log n)$ ，二分查找确定序号 i 的时间复杂度亦为 $O(\log n)$ ，因此该方法的查询时间复杂度为 $O(\log^2 n)$ 。更为高效的方法是使用类似于前述的查询数组中的第 k 个为 0 的元素的做法，借助于线段树中结点保存的区间和，从根结点开始，如果根结点的和小于 x ，则不存在序号 i 满足要求。若左子树的和大于等于 x ，表明满足要求的序号 i 在根结点的左子树，否则就在右子树。类似的，如果满足要求的序号 i 在右子树，则在递归查询时需要将当前的 x 减去左子树的和。容易知道，最后一种方法的查询时间复杂度为 $O(\log n)$ ，更新数组元素的时间复杂度也为 $O(\log n)$ 。

```
-----2.11.1.3.cpp-----
// st[p].sum 存储的是结点 p 所对应的区间内元素的和。
int findMinIndex(int p, int left, int right, int x) {
    if (x > st[p].sum) return -1;
    if (left == right) return left;
    int middle = (left + right) >> 1;
    if (st[LC[p]].sum >= x)
        return findMinIndex(LC(p), left, middle, x);
    else
        return findMinIndex(RC(p), middle + 1, right, x - st[LC(p)].sum);
}
-----2.11.1.3.cpp-----
```

(4) 给定一个整数数组 A 及整数 x ，查询给定区间 $[l, r]$ 内第一个大于 x 的元素的序号。建立线段树，线段树的结点存储的是对应区间内元素的最大值，然后从根结点开始进行查询，根据左右子树最大值的情况递归查找，时间复杂度为 $O(\log n)$ 。

```
-----2.11.1.4.cpp-----
// st[p].max 存储的是结点 p 所对应的区间内元素的最大值。
int findFirst(int p, int left, int right, int qleft, int qright, int x) {
    if (right < qleft || left > qright) return -1;
    if (left <= qleft && qright <= right) {
        if (st[p].max < x) return -1;
        // 满足要求的元素在此区间内，使用二分查找确定其最终位置。
        while (left != right) {
            int middle = (left + right) >> 1;
            if (st[LC(p)].max > x) { p = LC(p); right = middle; }
            else { p = RC(p); left = middle + 1; }
        }
        return left;
    }
    // 递归查询。
    int middle = (left + right) >> 1;
    int idx = findFirst(LC(p), left, middle, qleft, qright, x);
    if (idx != -1) return idx;
    return findFirst(RC(p), middle + 1, right, qleft, qright, x);
}
-----2.11.1.4.cpp-----
```

(5) 寻找区间最大值以及该最大值出现的次数。由于不仅需要记录区间的最大值，还需要记录最大值出现的次数，需要对结点所保存的信息域进行修改。在此种情形下，使用配对 (pair) 模板数据类型较为合

适，配对的第一个成员表示区间的最大值，第二个成员表示该最大值在此区间中出现的次数。在获取子区间的结果后，可以使用适当的方法将子区间的结果进行合并。

```
//++++++2.11.1.5.cpp+++++++
// 合并子区间的结果。
pair<int, int> combine(pair<int, int> a, pair<int, int> b) {
    if (a.first > b.first) return a;
    if (b.first > a.first) return b;
    return make_pair(a.first, a.second + b.second);
}

创建、查询、更新线段树与前述介绍的线段树基本操作类似。

const int MAXN = 1000010, INF = 0x7f7f7f7f;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

int data[MAXN];
pair<int, int> st[4 * MAXN];

void pushUp(int p) {
    st[p] = combine(st[LC(p)], st[RC(P)]);
}

// 创建线段树。
void build(int data[], int p, int left, int right) {
    if (left == right) st[p] = make_pair(data[left], 1);
    else {
        int middle = (left + right) >> 1;
        build(data, LC(p), left, middle);
        build(data, RC(P), middle + 1, right);
        pushUp(p);
    }
}

// 查询线段树。
pair<int, int> query(int p, int left, int right, int qleft, int qright) {
    if (left > qright || right < qleft) return make_pair(-INF, 0);
    if (left >= qleft && right <= qright) return st[p];
    int middle = (left + right) >> 1;
    pair<int, int> q1 = query(LC(p), left, middle, qleft, qright);
    pair<int, int> q2 = query(RC(P), middle + 1, right, qleft, qright);
    return combine(q1, q2);
}

// 更新线段树。
void update(int p, int left, int right, int index, int value) {
    if (left == right) st[p] = make_pair(value, 1);
    else {
        int middle = (left + right) >> 1;
        if (index <= middle)
            update(LC(p), left, middle, index, value);
        else
            update(RC(P), middle + 1, right, index, value);
        pushUp(p);
    }
}
```

```

    }
}

//++++++2.11.1.5.cpp+++++++

```

(6) 查询具有最大和的子区间。给定整数数组 A 和区间 $[l, r]$, 要求在此区间中寻找一个子区间 $[l', r']$, 其中 $l \leq l'$, $r' \leq r$, 使得在区间 $[l', r']$ 中的元素具有最大的和。为了确定给定区间的最大和子区间, 需要记录四个信息: 区间的“和” sum , 区间的“最大前缀和” $prefix$, 区间的“最大后缀和” $suffix$, 区间的“最大子区间和” sub 。因此定义以下的结构体来表示结点需要记录的信息。

```

//++++++2.11.1.6.cpp+++++++
// 定义结点所包含的信息。
struct node { int sum, prefix, suffix, sub; };

```

对于给定的区间, 其值由其左右儿子区间的值所确定, 可能存在以下三种情况:

- (a) 具有最大和的子区间位于左子结点;
- (b) 具有最大和的子区间位于右子结点;
- (c) 具有最大和的子区间“跨越”左右子结点, 即最大和子区间一部分位于左子结点所代表的区间内, 另外一部分位于右子结点所代表的区间内。

```

// 合并子区间的结果。
node combine(node a, node b) {
    if (a.sum == -INF) return b;
    if (b.sum == -INF) return a;
    node nd;
    nd.sum = a.sum + b.sum;
    nd.prefix = max(a.prefix, a.sum + b.prefix);
    nd.suffix = max(b.suffix, b.sum + a.suffix);
    nd.sub = max(max(a.sub, b.sub), a.suffix + b.prefix);
    return nd;
}

```

创建、查询、更新线段树与基本的线段树操作类似。此处定义了一个 `getData` 方法将给定的值转换为结点所记录的数据类型。

```

const int MAXN = 1000010, INF = 0x7f7f7f7f;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

node st[4 * MAXN];

// 将给定的值转换为结点。
node getData(int value) {
    node nd;
    nd.sum = nd.prefix = nd.suffix = nd.sub = value;
    return nd;
}

void pushUp(int p) {
    st[p] = combine(st[LC(p)], st[RC(p)]);
}

// 创建线段树。
void build(int data[], int p, int left, int right) {

```

```

    if (left == right) st[p] = getData(data[left]);
    else {
        int middle = (left + right) >> 1;
        build(data, LC(p), left, middle);
        build(data, RC(p), middle + 1, right);
        pushUp(p);
    }
}

// 查询线段树。
node query(int p, int left, int right, int qleft, int qright) {
    if (left > qright || right < qleft) return getData(-INF);
    if (left >= qleft && right <= qright) return st[p];
    int middle = (left + right) >> 1;
    node q1 = query(LC(p), left, middle, qleft, qright);
    node q2 = query(RC(p), middle + 1, right, qleft, qright);
    return combine(q1, q2);
}

// 更新线段树。
void update(int p, int left, int right, int index, int value) {
    if (left == right) st[p] = getData(value);
    else {
        int middle = (left + right) >> 1;
        if (index <= middle)
            update(LC(p), left, middle, index, value);
        else
            update(RC(p), middle + 1, right, index, value);
        pushUp(p);
    }
}
//+++++++++++++++++2.11.1.6.cpp+++++++++++++++++/

```

(7) 给定一个整数数组 A 及整数 x , 查询给定区间 $[l, r]$ 内不小于 x 的最小元素的值。先来考虑不带修改的查询。如果给定的区间内的数是有序的, 使用二分查找可以在 $O(\log n)$ 的时间内得到结果, 基于此, 我们可以在线段树的结点保存一个列表, 该列表保存的是该结点所对应的区间内的数组元素, 而且该列表是有序的。根据这些有序的列表, 我们可以从线段树的根结点出发, 沿着线段树的向下分解待查询的区间, 利用二分查找得到每个分解区间内的查询结果, 然后再取这些分解区间查询结果的最小值即可。使用这样的方法, 在每个线段树的结点保存的数组元素列表总的空间消耗为 $O(n \log n)$, 这似乎有违直觉, 但是由于线段树的高度是 $O(\log n)$ 的, 而线段树的每一层所存储的列表合并起来就是整个数组, 即空间消耗为 $O(n)$, 因此总的空间消耗是 $O(n \log n)$ 。那么如何高效地建立这样一棵线段树呢? 我们可以使用类似于归并排序的方法¹, 自底向上构建该线段树, 称之为“归并排序树”(merge sort tree)。

```

//+++++++++++++++++2.11.1.7.cpp+++++++++++++++++/
#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

const int MAXN = 1000010;

```

¹ 参见第4章“排序与查找”第4.4节“归并排序”的内容。

```

int n, A[MAXN];
vector<int> st[MAXN << 2];

void build(int p, int left, int right) {
    if (left == right) st[p] = vector<int>(1, A[left]);
    else {
        int mid = (left + right) >> 1;
        // 递归构建线段树。
        build(LC(p), left, mid);
        build(RC(p), mid + 1, right);
        // 利用算法库函数 merge 合并两个有序列表。
        merge(st[LC(p)].begin(), st[LC(p)].end(),
              st[RC(p)].begin(), st[RC(p)].end(), back_inserter(st[p]));
    }
}

```

线段树本身的构造时间复杂度为 $O(n\log n)$ ，而归并排序的时间复杂度亦为 $O(n\log n)$ ，因此总的时间复杂度仍为 $O(n\log n)$ 。

接下来，我们可以从线段树的根结点出发，沿着左右儿子结点往下，将待查询区间分解为一系列的子区间，这些子区间最多有 $O(\log n)$ 个，对这些子区间使用二分查找确定各自的结果，最后将结果进行合并即可得到最后的查询结果。

```

// 定义一个“无穷大值”，当待查询区间内不存在符合要求的元素值时，返回该值。
const int INF = 0x7f7f7f7f;

int query(int p, int left, int right, int ql, int qr, int x) {
    if (ql > qr) return INF;
    // 对子区间内的有序列表使用二分查找获得符合要求的元素值。
    if (ql == left && qr == right) {
        vector<int>::iterator idx = lower_bound(st[p].begin(), st[p].end(), x);
        if (idx != st[p].end())
            return *idx;
        return INF;
    }
    // 递归查询，取较小的元素值。
    int mid = (left + right) >> 1;
    return min(query(LC(p), left, mid, ql, min(qr, mid), x),
               query(RC(p), mid + 1, right, max(ql, mid + 1), qr, x));
}

```

从修改过程不难看出，修改单个元素的时间复杂度为 $O(\log^2 n)$ 。现在我们将查询条件放宽，使得查询支持修改，即可以对数组中的元素进行修改，修改后查询时间复杂度不变。由于需要修改数组元素的值，仍然使用向量来保存有序列表将使得修改元素的代价较高。由于数组元素可能包含重复的数组元素，我们可以使用 STL 中的多重集合 (multiset) 来保存有序列表，这样修改数组元素将非常便利，而且在初始构建线段树的时间复杂度不变。

```

multiset<int> st[MAXN << 2];

void update(int p, int left, int right, int idx, int value) {
    st[p].erase(st[p].find(A[idx]));
    st[p].insert(value);
    if (left != right) {
        int mid = (left + right) >> 1;

```

```

        if (idx <= mid) update(LC(p), left, mid, idx, value);
        else update(RC(p), mid + 1, right, idx, value);
    } else A[idx] = value;
}

```

由于使用了多重集合，该数据结构不支持随机访问迭代器，使用算法库函数 `lower_bound` 进行二分查找无法保证 $O(\log n)$ 的时间复杂度，因此在进行二分查找时需要使用多重集合自带的二分查找方法，因此对构建和查询需要进行相应修改。

```

void build(int p, int left, int right) {
    if (left == right) st[p].insert(A[left]);
    else {
        int mid = (left + right) >> 1;
        build(LC(p), left, mid);
        build(RC(p), mid + 1, right);
        // 利用算法库函数 set_union 合并两个 multiset。
        set_union(st[LC(p)].begin(), st[LC(p)].end(),
                  st[RC(p)].begin(), st[RC(p)].end(), inserter(st[p], st[p].begin())));
    }
}

int query(int p, int left, int right, int ql, int qr, int x) {
    if (ql > qr) return INF;
    if (ql == left && qr == right) {
        multiset<int>::iterator idx = st[p].lower_bound(x);
        if (idx != st[p].end())
            return *idx;
        return INF;
    }
    int mid = (left + right) >> 1;
    return min(query(LC(p), left, mid, ql, min(qr, mid), x),
               query(RC(p), mid + 1, right, max(ql, mid + 1), qr, x)));
}

```

现在我们来考虑进一步优化不带修改时的查询效率。使用分散层叠（fractional cascading）技巧¹，通过适当的预处理为线段树的每个结点生成一个辅助列表，只需在根结点对应的辅助列表中进行一次二分查找，然后利用查找所得到的信息，沿着根结点往下，通过常数次的比对，即可获得查询结果。分散层叠的本质是在一系列有序表之间建立若干“桥”，利用这些桥，能够使得只进行一次二分查找，然后根据查找结果通过桥跳转到另外一个有序列表继续进行查询。在前述生成归并排序树的过程中，令当前所在结点所对应的数组有序列表为 L_p ，其左侧子树结点所对应的有序列表为 $L_{LC(p)}$ ，右侧子树结点所对应的有序列表为 $L_{RC(p)}$ ，构建归并排序树的目的是将 $L_{LC(p)}$ 与 $L_{RC(p)}$ 合并得到 L_p 。在此过程中，我们额外维护以下信息：令 y 为列表 L_p 中的某个元素，序号 i 为列表中 $L_{LC(p)}$ 中大于或等于 y 的最小数组元素的序号，如果不存在这样的数组元素，则 i 为 -1，序号 j 为列表 $L_{RC(p)}$ 中大于或等于 y 的最小数组元素的序号，如果不存在这样的数组元素，则 j 为 -1。上述信息可以在构建归并排序树时方便地得到。当构建完辅助列表和相应的额外信息后，给定整数 x ，我们可以在线段树的根节点所对应的有序列表中进行一次二分查询，得到大于等于 x 的最小数组元素，令其为 y ，根据 y 所附带的额外信息，我们可以立即得到在根结点左侧子树所对应的有序列表中大于等于 y 的最小数组

¹ 参见第 4 章“排序与查找”第 4.8.6 小节“分散层叠”的内容。

元素的序号 i , 如果 i 不为-1, 则由于序号 i 所对应的数组元素和 y 之间并不存在具有其他值的数组元素, 因此在根结点的左子树中, 序号 i 所对应的数组元素就是根结点的左子树对应有序列表中大于等于 x 的最小整数, 同理, 如果 y 所附带的额外信息 j 不为-1, 则序号 j 所对应的数组元素就是根结点的右子树对应有序列表中大于等于 x 的最小整数。继续沿着线段树往下, 直到将待查询区间分解为若干个子区间, 这若干个子区间内的结点所对应的有序列表中大于等于 x 的最小整数都可以利用前述的形式予以获取, 最后只需常数次比较取其中的最小值即为结果。

```

// 存储有序列表中元素的额外信息。
vector<pair<int, int>> b[MAXN << 2];

void build(int p, int left, int right) {
    if (left == right) st[p] = vector<int>(1, A[left]);
    else {
        int mid = (left + right) >> 1;
        build(LC(p), left, mid);
        build(RC(p), mid + 1, right);
        // 在构建归并排序树的过程中同时维护有序列表元素的额外信息。
        int i = 0, j = 0;
        int nlc = st[LC(p)].size(), nrc = st[RC(p)].size();
        while (i < nlc && j < nrc) {
            if (st[LC(p)][i] <= st[RC(p)][j]) {
                st[p].push_back(st[LC(p)][i]);
                b[p].push_back(make_pair(i, j));
                i++;
            } else {
                st[p].push_back(st[RC(p)][j]);
                b[p].push_back(make_pair(i, j));
                j++;
            }
        }
        while (i < nlc) {
            st[p].push_back(st[LC(p)][i]);
            b[p].push_back(make_pair(i, -1));
            i++;
        }
        while (j < nrc) {
            st[p].push_back(st[RC(p)][j]);
            b[p].push_back(make_pair(-1, j));
            j++;
        }
    }
}

int query(int p, int left, int right, int ql, int qr, int idx) {
    if (ql > qr) return INF;
    if (ql == left && qr == right) {
        if (idx == -1) return INF;
        return st[p][idx];
    }
    int mid = (left + right) >> 1;
    return min(query(LC(p), left, mid, ql, min(qr, mid), b[p][idx].first),
               query(RC(p), mid + 1, right, max(ql, mid + 1), qr, b[p][idx].second));
}

```

```

int main(int argc, char *argv[]) {
    cin >> n;
    for (int i = 0; i < n; i++) cin >> A[i];
    int rt = 0;
    build(rt, 0, n - 1);
    int l, r, x;
    while (cin >> l >> r >> x) {
        int idx = lower_bound(st[rt].begin(), st[rt].end(), x) - st[rt].begin();
        if (idx == st[rt].size()) { cout << "Not exists!\n"; continue; }
        int y = query(rt, 0, n - 1, l, r, idx);
        if (y != INF) cout << y << '\n';
        else cout << "Not exists!\n" << '\n';
    }
    return 0;
}
//+++++2.11.1.7.cpp+++++

```

2.11.2 二维线段树

在实际应用中，除了使用二叉树来表示一维线段树外，还可以使用四叉树（quadtree）来实现二维线段树（2D segment tree），进行矩形范围查询，或者更复杂的，使用八叉树（octree）来实现三维线段树，进行三维空间范围查询，不过由于实现较为繁琐，实际解题中极少运用。二维线段树有两种常见的实现方法，一种是沿用一维线段树的思路，使用四叉树来实现，即采用矩形分割的方法来二分查询范围，单个结点表示的是一个子矩形所对应的范围；另外一种实现方法是“树套树”，即一维线段树中包含的不再仅仅是一个结体，而是另外一维的线段树。

二维线段树的四叉树实现

给定一个矩形范围(x_1, y_1, x_2, y_2)，可分别确定行和列的中点后将其分解为四个子矩形范围（假设坐标的 X 轴正向水平向右，Y 轴正向垂直向下），即：

```

int mx = (x1 + x2) / 2, my = (y1 + y2) / 2;
左上角子矩形范围: (x1, y1, mx, my);
左下角子矩形范围: (x1, my + 1, mx, y2);
右上角子矩形范围: (mx + 1, y1, x2, my);
右下角子矩形范围: (mx + 1, my + 1, x2, y2)

```

在四叉树中，这四个子矩形所对应的树结点是“母矩形”对应树结点的四个子结点。在获取子矩形的过程中，由于初始给定的矩形可能只有一行或者一列，当进行分割时，导致后续得到的子矩形是一个“非法”的矩形，需要对其进行验证，否则在创建、查询、更新时会产生错误。

```

//-----2.11.2.1.cpp-----//
const int MAXN = 512, INF = 0x7f7f7f7f;

int data[MAXN][MAXN];
int st[4 * MAXN * MAXN];

void pushUp(int p) {
    int high = -INF;
    for (int i = 1; i <= 4; i++) high = max(high, st[4 * p + i]);
    st[p] = high;
}

void build(int p, int x1, int y1, int x2, int y2) {

```

```

// 当范围无效时, 需要正确设置子结点的值, 否则在进行 pushUp 操作时会得到错误的结果。
if (x1 > x2 || y1 > y2) {
    st[p] = -INF;
    return;
}
if (x1 == x2 && y1 == y2) {
    st[p] = data[x1][y1];
    return;
}
int mx = (x1 + x2) >> 1, my = (y1 + y2) >> 1;
build(4 * p + 1, x1, y1, mx, my);
build(4 * p + 2, x1, my + 1, mx, y2);
build(4 * p + 3, mx + 1, y1, x2, my);
build(4 * p + 4, mx + 1, my + 1, x2, y2);
pushUp(p);
}

// 查询指定矩形范围内的最大值。矩形的左上角坐标为 (qx1, qy1), 右下角坐标为 (qx2, qy2)。
int query
(int p, int x1, int y1, int x2, int y2, int qx1, int qy1, int qx2, int qy2) {
    if (x1 > x2 || y1 > y2) return -INF;
    if (x2 < qx1 || y2 < qy1 || x1 > qx2 || y1 > qy2) return -INF;
    if (qx1 <= x1 && x2 <= qx2 && qy1 <= y1 && y2 <= qy2) return st[p];
    int mx = (x1 + x2) >> 1, my = (y1 + y2) >> 1;
    int q1 = query(4 * p + 1, x1, y1, mx, my, qx1, qy1, qx2, qy2);
    int q2 = query(4 * p + 2, x1, my + 1, mx, y2, qx1, qy1, qx2, qy2);
    int q3 = query(4 * p + 3, mx + 1, y1, x2, my, qx1, qy1, qx2, qy2);
    int q4 = query(4 * p + 4, mx + 1, my + 1, x2, y2, qx1, qy1, qx2, qy2);
    return max(max(q1, q2), max(q3, q4));
}

// 单点更新。将元素 data[ux][uy] 的值更新为 v。
void update(int p, int x1, int y1, int x2, int y2, int ux, int uy, int v) {
    if (x1 > x2 || y1 > y2) return;
    if (x2 < ux || y2 < uy || x1 > ux || y1 > uy) return;
    if (x1 == x2 && y1 == y2 && x1 == ux && y1 == uy) { st[p] = v; return; }
    int mx = (x1 + x2) >> 1, my = (y1 + y2) >> 1;
    update(4 * p + 1, x1, y1, mx, my, ux, uy, v);
    update(4 * p + 2, x1, my + 1, mx, y2, ux, uy, v);
    update(4 * p + 3, mx + 1, y1, x2, my, ux, uy, v);
    update(4 * p + 4, mx + 1, my + 1, x2, y2, ux, uy, v);
    pushUp(p);
}
//-----2.11.2.1.cpp-----//

```

在上述实现中, 使用数组来表示整个二维线段树, 在某些情况下, 可能矩形的某一维度较大, 而另外一个维度较小, 如果仍然使用数组的表示方法会导致较多的空间被浪费, 甚至有可能超出内存限制, 此时可以采用动态分配内存的方式来建立结点, 从而尽可能地节省空间。但由于“随用随建”, 程序的运行效率要稍低。另外, 可以将操作“封装”到一个表示矩形的结构体中, 这样能够使得实现更为“井然有序”, 也便于理解, 但代码的运行效率可能不够高。

```

//-----2.11.2.2.cpp-----//
const int MAXN = 512, INF = 0x7f7f7f7f;

struct rectangle {

```

```

// (x1, y1) 表示矩形的左上角坐标, (x2, y2) 表示矩形的右下角坐标。
int x1, y1, x2, y2;

rectangle(int x1 = 0, int y1 = 0, int x2 = 0, int y2 = 0):
x1(x1), y1(y1), x2(x2), y2(y2) {}

// 测试是否为有效矩形。
bool isBad() { return x1 > x2 || y1 > y2; }

// 测试是否已经为单位矩形。
bool isCell() { return x1 == x2 && y1 == y2; }

// 测试是否包含指定的单位矩形。
bool contains(int x, int y)
{ return x1 <= x && x <= x2 && y1 <= y && y <= y2; }

// 测试是否包含矩形 q。
bool contains(rectangle q)
{ return x1 <= q.x1 && q.x2 <= x2 && y1 <= q.y1 && q.y2 <= y2; }

// 测试是否与矩形 q 相交。
bool intersects(rectangle q)
{ return !(x1 > q.x2 || y1 > q.y2 || x2 < q.x1 || y2 < q.y1); }

// 返回位于左上角的子矩形。
rectangle getLU()
{ return rectangle(x1, y1, (x1 + x2) >> 1, (y1 + y2) >> 1); }

// 返回位于右上角的子矩形。
rectangle getRU()
{ return rectangle(x1, ((y1 + y2) >> 1) + 1, (x1 + x2) >> 1, y2); }

// 返回位于左下角的子矩形。
rectangle getLB()
{ return rectangle(((x1 + x2) >> 1) + 1, y1, x2, (y1 + y2) >> 1); }

// 返回位于右下角的子矩形。
rectangle getRB()
{ return rectangle(((x1 + x2) >> 1) + 1, ((y1 + y2) >> 1) + 1, x2, y2); }

};

struct node {
    int high;
    node* children[4];
};

int data[MAXN][MAXN];

// 创建结点。
node* getNode() {
    node *nd = new node;
    nd->high = -INF;
    for (int i = 0; i < 4; i++) nd->children[i] = NULL;
    return nd;
}

```

```

// 更新结点的值。
void pushUp(node *nd) {
    int high = -INF;
    for (int i = 0; i < 4; i++) {
        if (nd->children[i] == NULL) continue;
        high = max(high, nd->children[i]->high);
    }
    nd->high = high;
}

// 创建线段树。
node* build(rectangle r) {
    if (r.isBad()) return NULL;
    if (r.isCell()) {
        node *nd = getNode();
        nd->high = data[r.x1][r.y1];
        return nd;
    }
    node *nd = getNode();
    nd->children[0] = build(r.getLU());
    nd->children[1] = build(r.getRU());
    nd->children[2] = build(r.getLB());
    nd->children[3] = build(r.getRB());
    pushUp(nd);
    return nd;
}

// 查询线段树。
int query(node *nd, rectangle r, rectangle qr) {
    if (r.isBad()) return -INF;
    if (!r.intersects(qr)) return -INF;
    if (qr.contains(r)) return nd->high;
    int q1 = query(nd->children[0], r.getLU(), qr);
    int q2 = query(nd->children[1], r.getRU(), qr);
    int q3 = query(nd->children[2], r.getLB(), qr);
    int q4 = query(nd->children[3], r.getRB(), qr);
    return max(max(q1, q2), max(q3, q4));
}

// 单点更新。
void update(node *nd, rectangle r, int ux, int uy, int v) {
    if (r.isBad()) return;
    if (!r.contains(ux, uy)) return;
    if (r.isCell()) { nd->high = v; return; }
    update(nd->children[0], r.getLU(), ux, uy, v);
    update(nd->children[1], r.getRU(), ux, uy, v);
    update(nd->children[2], r.getLB(), ux, uy, v);
    update(nd->children[3], r.getRB(), ux, uy, v);
    pushUp(nd);
}
//-----2.11.2.2.cpp-----//

```

二维线段树的嵌套实现

二维线段树的嵌套实现又称“树套树”实现。使用“树套树”的实现方法，先对横坐标进行分割，当横

坐标分割为基本单元后，再对纵坐标进分割。可以这样理解：对于一维线段树来说，每个结点就是单一个结点，里面保存的是若干个数据，但是二维线段树中，相当于将一维线段树中每个结点进行扩展，每个结点都是一棵线段树。

```

//-----2.11.2.3.cpp-----
const int MAXN = 512, INF = 0x7f7f7f7f;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

int n, m, data[MAXN][MAXN];

int st[4 * MAXN][4 * MAXN];

void buildY(int px, int lx, int rx, int py, int ly, int ry) {
    if (ly == ry) {
        if (lx == rx) st[px][py] = data[lx][ly];
        else st[px][py] = max(st[LC(px)][py], st[RC(px)][py]);
    } else {
        int my = (ly + ry) >> 1;
        buildY(px, lx, rx, LC(py), ly, my);
        buildY(px, lx, rx, RC(py), my + 1, ry);
        st[px][py] = max(st[px][LC(py)], st[px][RC(py)]);
    }
}

void buildX(int px, int lx, int rx) {
    if (lx != rx) {
        int mx = (lx + rx) >> 1;
        buildX(LC(px), lx, mx);
        buildX(RC(px), mx + 1, rx);
    }
    buildY(px, lx, rx, 0, 0, m - 1);
}

int queryY(int px, int py, int ly, int ry, int qly, int qry) {
    if (ly > qry || ry < qry) return -INF;
    if (qly <= ly && ry <= qry) return st[px][py];
    int my = (ly + ry) >> 1;
    int q1 = queryY(px, LC(py), ly, my, qly, qry);
    int q2 = queryY(px, RC(py), my + 1, ry, qly, qry);
    return max(q1, q2);
}

int queryX(int px, int lx, int rx, int qlx, int qly, int qrx, int qry) {
    if (lx > qrx || rx < qlx) return -INF;
    if (qlx <= lx && rx <= qrx) return queryY(px, 0, 0, m - 1, qly, qry);
    int mx = (lx + rx) >> 1;
    int q1 = queryX(LC(px), lx, mx, qlx, qly, qrx, qry);
    int q2 = queryX(RC(px), mx + 1, rx, qlx, qly, qrx, qry);
    return max(q1, q2);
}

void updateY(
int px, int lx, int rx, int py, int ly, int ry, int x, int y, int value) {
    if (ly == ry) {

```

```

        if (lx == rx) st[px][py] = value;
        else st[px][py] = max(st[LC(px)][py], st[RC(px)][py]);
    } else {
        int my = (ly + ry) >> 1;
        if (y <= my)
            updateY(px, lx, rx, LC(py), ly, my, x, y, value);
        else
            updateY(px, lx, rx, RC(py), my + 1, ry, x, y, value);
        st[px][py] = max(st[px][LC(py)], st[px][RC(py)]);
    }
}

void updateX(int px, int lx, int rx, int x, int y, int value) {
    if (lx != rx) {
        int mx = (lx + rx) >> 1;
        if (x <= mx)
            updateX(LC(px), lx, mx, x, y, value);
        else
            updateX(RC(px), mx + 1, rx, x, y, value);
    }
    updateY(px, lx, rx, 0, 0, m - 1, x, y, value);
}
//-----2.11.2.3.cpp-----//

```

二维线段树的两种实现方法各有优劣。使用四叉树的方式实现，可以便于应用延迟更新，但在查询效率上较“树套树”的实现慢，不过空间利用率较高，因为结点是“随用随建”；使用“树套树”的实现方法，空间利用率较前者低，不便于应用延迟更新，但是查询效率较高。

动态创建结点的线段树

通常来说，线段树所占用空间是区间总长度 n 的常数倍，即空间复杂度为 $O(n)$ ，但在某些应用中，由于 n 较大，且不需使用所有的结点，则可以选择动态建立线段树而不是一次性建立整棵线段树，也就是一边查询、修改，一边按需建立新的线段树结点。尤其对于二维线段树来说，有时给定的 X 轴坐标和 Y 轴坐标范围相当大，无法使用一个二维数组来予以表示，此时可以对线段树所需要的结点随用随建。由于对于 n 个结点的线段树来说，在 X 轴坐标这一维度创建一条线段树的路径需要 $\log n$ 个结点，在 Y 轴坐标这一维度创建一条线段树的路径又需要 $\log n$ 个结点，则与二维线段树一个结点相关联的空间需求是 $\log^2 n$ 级别，总的空间需求是 $n \log^2 n$ 级别。

12086 Potentiometers^A (电位计)

电位计是用来测量电位差的一种仪器，其内部的电阻值是可调节的。将若干个电位计依次串联起来（不构成环，即第一个电位计的左端和最后一个电位计的右端是未连接的，其他电位计的左右两端均和相邻电位计的对应端连接），给定初始时各个电位计的电阻值，要求你进行以下两种操作：

- (1) 将指定序号的电位计的电阻值设置为某个值；
- (2) 计算指定序号范围内电位计的电阻值之和。

输入

输入包含的测试数据组数少于 3 组。每组测试数据以 N 开始，表示电位计的数量， N 最多可达 200000，接下来的 N 行每行包含一个 0 至 1000 之间的整数，表示序号从 1 到 N 的电位计的初始电阻值。后续是一

系列的操作，这些操作的数量最多可达 200000 个，操作分三种：

- (1) “S $x r$ ”，表示将序号为 x 的电位计的电阻值设置为 r ，该操作即时生效，会影响后续的电阻测量；
- (2)“M $x y$ ”，表示测量从序号 x 到序号 y 的电位计的电阻值之和，输入保证 x 和 y 都在序号范围之内，且 x 小于 y ；
- (3) “END”，表示此组测试数据结束。

当 $N=0$ 时，表示输入文件结束。

输出

对于每组测试数据，先输出测试数据的组数序号，从 1 开始计数，输出形式为 “Case n:”，对于测试数据中每次测量，输出测量得到的电阻值，在相邻两组测试数据的输出之间打印一个空行。

样例输出

```
3
100
100
100
M 1 1
END
0
```

样例输出

```
Case 1:
100
```

分析

题目给定的数据量较大，如果使用朴素的“线性累加”方法进行解题会导致超时。由于查询的是电位计的电阻之和，符合“目标区间的信息可以由相邻两个连续区间合并后的信息表示”的特点，可使用线段树予以解决。构建线段树时，树的叶结点保存的是各个电位计的电阻值，内部结点保存的是左右儿子区间的电阻值之和。每次重新设置电位计的电阻，相当于单结点更新。

强化练习：1232 SKYLINE^D, 11235 Frequent Values^A, 11402 Ahoy Pirates^B, 12299 RMQ with Shifts^D, 12532 Interval Product^B。

扩展练习：1400 Ray Pass Me the Dishes^D, 11297* Census^D, 11992* Fast Matrix Operations^D。

2.11.3 可持久化线段树

可持久化数据结构（*persistent data structure*）是指能够存储数据结构的历史版本的数据结构，它能够允许对若干次修改前状态的数据结构进行查询。线段树可以方便地可持久化，而且可持久化所需消耗的时间和空间复杂度均是 $O(\log n)$ 。

回顾线段树的单点更新过程，我们从根结点出发，沿着线段树向下走，至多需要 $\lfloor \log n \rfloor + 1$ 步就可以到达需要更新的叶结点，在更新叶结点后，需要沿着原来的路径返回根结点，逐次更新路径上的相应结点的信息，以适应此次单结点更新后的变化。容易看出，在上述过程中，线段树中受影响的结点至多为 $\lfloor \log n \rfloor + 1$ 个，而其他结点所保存的信息是未发生变化的。因此，我们可以使用指针的方式来创建线段树的结点，每进行一次单点更新，只需将更新路径上的结点使用新的结点替换即可，最后会得到一个新的根结点。如果进行多次单点更新，就会得到多个新的根结点，每个根结点对应了一次单点更新后线段树的版本，使用对应版本的线段树根结点进行查询，所得到的查询结果就是对应历史版本的线段树查询结果。

以下是可持久化线段树的一种实现，使用指针存储结点，完成区间和查询，支持单点更新和历史版本数据的查询。

```

//-----2.11.3.1.cpp-----
const int MAXN = 1000010;

struct node {
    int sum;
    node *lc, *rc;
    node(int v): lc(nullptr), rc(nullptr), sum(v) {}
    node(node *lc, node *rc): lc(lc), rc(rc), sum(0) {
        if (lc) sum += lc->sum;
        if (rc) sum += rc->sum;
    }
};

int n, A[MAXN];

node* build(int left, int right) {
    if (left == right) return new node(A[left]);
    int mid = (left + right) >> 1;
    return new node(build(left, mid), build(mid + 1, right));
}

int query(node* p, int left, int right, int ql, int qr) {
    if (ql > qr) return 0;
    if (left == ql && right == qr) return p->sum;
    int mid = (left + right) >> 1;
    return query(p->lc, left, mid, ql, min(qr, mid)) +
        query(p->rc, mid + 1, right, max(left, mid + 1), right);
}

node* update(node* p, int left, int right, int idx, int v) {
    if (left == right) return new node(v);
    int mid = (left + right) >> 1;
    if (idx <= mid) return new node(update(p->lc, left, mid, idx, v), p->rc);
    else return new node(p->lc, update(p->rc, mid + 1, right, idx, v));
}
//-----2.11.3.1.cpp-----

```

现在我们来考虑以下问题：给定有 n 个元素的数组 A 和区间 $[l, r]$ ，要求确定区间内的第 k 小元素。如果利用二分查找和前述介绍的归并排序树，可以在 $O(\log^3 n)$ 的时间复杂度内回答上述查询，而利用可持久化线段树，可以将查询时间复杂度优化到 $O(\log n)$ 。

为了便于说明，我们先来考虑一个更简单一些的问题，在解决这个更简单问题的基础上，一步一步向目标问题进行转化，直到解决最终的问题。考虑以下问题：给定数组 A ，数组 A 中的元素满足 $0 \leq A[i] \leq n$ ，要求查询区间 $[1, r]$ 内的第 k 小元素（为了便于后续的说明，数组元素的序号从 1 开始计数）。类似于计数排序，我们可以使用线段树来计数出现过的数的数量，也就是说，在线段树的结点中存储的是值为 i 的元素出现的次数，然后使用类似于前述介绍的查询区间内 0 的元素个数和查询第 k 个为 0 的元素的方法，利用“范围和”查询来确定第 k 小元素的值。朴素的做法是为每个查询区间 $[1, 1], [1, 2], [1, 3], \dots, [1, n]$ 都创建一棵线段树用于查询，但更为巧妙的方法是利用可持久化线段树。在初始时，建立一棵空线段树，线段树中每个结点的计数都为 0，令此时线段树的根结点为 $root_0$ ，接着将 $A[1], A[2], A[3], \dots, A[n]$ 依次添加到可持久化线段树中，对于每次更新，我们都能够得到一个新的根结点 $root$ ——在插入数组 A 的前 i 个元素后得到的线段树的根结点。根结点为 $root_i$ 的线段树包含数组元素 $A[1 \dots i]$ 的计数信息，利用这些计数信息，我们就可以在 $O(\log n)$ 的时间内查询任意给定区间 $[1, r]$ 内的第 k 小元素。

现在我们考虑将问题的限制进一步放宽。相对于只能查询区间 $[1, r]$ 内的第 k 小元素，我们需要查询任意给定区间 $[l, r]$ 内的第 k 小元素，那么我们需要得到区间 $[l, r]$ 内数组元素的计数信息，容易知道，该计数信息恰恰就是根结点为 $root_r$ 的线段树所保存的数组元素计数信息与根结点为 $root_{l-1}$ 的线段树所保存的数组元素计数信息的差。根据上述思路，我们有以下的实现来完成查询区间 $[l, r]$ 内的第 k 小元素的任务。

```
//+++++2.11.3.2.cpp+++++  
node* build(int left, int right) {  
    if (left == right) return new node(0);  
    int mid = (left + right) / 2;  
    return new node(build(left, mid), build(mid + 1, right));  
}  
  
node* update(node* p, int left, int right, int idx) {  
    if (left == right) return new node(p->sum + 1);  
    int mid = (left + right) / 2;  
    if (idx <= mid) return new node(update(p->lc, left, mid, idx), p->rc);  
    else return new node(p->lc, update(p->rc, mid + 1, right, idx));  
}  
  
int findKth(node* pl, node *pr, int left, int right, int k) {  
    if (left == right) return left;  
    int mid = (left + right) / 2, remain = pr->lc->sum - pl->lc->sum;  
    if (remain >= k) return findKth(pl->lc, pr->lc, left, mid, k);  
    return findKth(pl->rc, pr->rc, mid + 1, right, k - remain);  
}
```

由于需要保存每次插入一个数组元素后的线段树根结点，可以使用以下方法来构建所需的可持久化线段树根结点数组。

```
int left = 0, right = B.size() + 1;  
vector<node*> roots;  
roots.push_back(build(left, right));  
for (int i = 0; i < n; i++)  
    roots.push_back(update(roots.back(), left, right, A[i]));
```

那么查询区间 $[l, r]$ 内的第 k 小元素，只需调用：

```
// 查询区间 [4, 32] 内的第 8 小元素。  
int l = 4, r = 32, k = 8;  
int result = findKth(roots[4], roots[32], left, right, 8);
```

最后，我们来考虑将数组元素的值放宽到任意值的问题。利用离散化技巧，我们可以将给定的数进行排序，然后根据大小对其值重设，将原数组中最小的数赋值为0，次小的数赋值为1……以便将原来的数映射到 $[0, n-1]$ 的范围，之后再使用前面所介绍的方法进行查询，最后只需通过查表来输出原数组元素的值即可，通过使用类似于STL中map的数据结构，将数组元素和赋值进行转换的时间可以做到 $O(\log n)$ 。

```
// 离散化，对数组元素的序号进行压缩。  
vector<int> B;  
for (int i = 0; i < n; i++) B.push_back(A[i]);  
sort(B.begin(), B.end());  
B.erase(unique(B.begin(), B.end()), B.end());  
map<int, int> A2B, B2A;  
for (int i = 0; i < n; i++) {  
    A2B[A[i]] = lower_bound(B.begin(), B.end(), A[i]) - B.begin();  
    B2A[A2B[A[i]]] = A[i];
```

```
B2A[A2B[A[i]]] = A[i];
}

// 对数组元素的序号进行压缩处理后, 前述的代码进行相应更改:
// 建立线段树:
// roots.push_back(update(roots.back(), left, right, A2B[A[i]]));
// 输出查询结果:
// int idx = findKth(roots[4], roots[32], left, right, 8);
// cout << B2A[idx] << '\n';
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++2.11.3.2.cpp++++++++++++++++++++++++++++++++++/
```

2.11.4 区间树

在解题应用中,有时需要对区间进行高效的查询和插入操作——查询是否存在与给定的区间相重叠的区间,若不存在,则将此区间插入到数据结构中。在这种应用场景下,可以使用区间树(interval tree)数据结构。区间树能够在 $O(\log n)$ 的时间内完成区间的查询和插入操作。实际上,前述的线段树就是一种特殊的区间树,其叶结点的区间长度均为0。

区间树的实现以二叉树为基础，二叉树的结点存储了区间信息以及在此区间上我们感兴趣的信息。

```
-----2.11.4.cpp-----
struct interval { int low, high; };

struct node
{
    interval i;
    int max;
    node *leftNode, *rightNode;
};

node* getNode(interval i)
{
    node *nd = new node;
    nd->i = i;
    nd->max = i.high;
    nd->leftNode = nd->rightNode = NULL;
    return nd;
}

node* insert(node *root, interval i)
{
    if (root == NULL) return getNode(i);
    if (i.low < root->i.low) root->leftNode = insert(root->leftNode, i);
    else root->rightNode = insert(root->rightNode, i);
    if (root->max < i.high) root->max = i.high;
    return root;
}

bool isOverlapped(interval i1, interval i2)
{
    if (i1.low <= i2.high && i2.low <= i1.high) return true;
    return false;
}

bool query(node *root, interval i)
{
```

```

if (root == NULL) return false;
if (isOverlapped(root->i, i)) return true;
if (root->leftNode != NULL && root->leftNode->max >= i.low)
    return query(root->leftNode, i);
return query(root->rightNode, i);
}
//-----2.11.4.cpp-----/

```

强化练习：11601* Avoiding Overlaps^D。

2.12 树状数组

树状数组，又称二叉索引树（binary index tree）、Fenwick 树，由 Fenwick 于 1994 年首先提出^[21]。树状数组最初被设计用于数据压缩，而现在主要用于存储频次信息或者用于计算累计频次表。

给定一个长度为 n 的整数数组 A ，要求确定指定范围内元素的和。朴素的做法是计算数组的“前缀和” $P[i] = A[1] + \dots + A[i]$ ， $i \geq 1$ ，得到“前缀和”数组 P 后，令 $P[0] = 0$ ，则区间 $[L, R]$ 的“范围和” $S[L, R] = P[R] - P[L-1]$ 。此种计算方式的时间复杂度和空间复杂度均为 $O(n)$ ，缺点是每当数组元素 $A[i]$ 的值发生改变后，都需要重新计算 $P[i]$ 之后的“前缀和”，平均需要 $O(n)$ 的时间，效率不够高。而应用树状数组，可以在 $O(n \log n)$ 的时间内构建一个效果类似于数组 P 的数组 T ，之后可以在 $O(\log n)$ 的时间内对数组 T 的元素进行更新，同时可以在 $O(\log n)$ 的时间内查询数组 A 中指定范围的元素和。

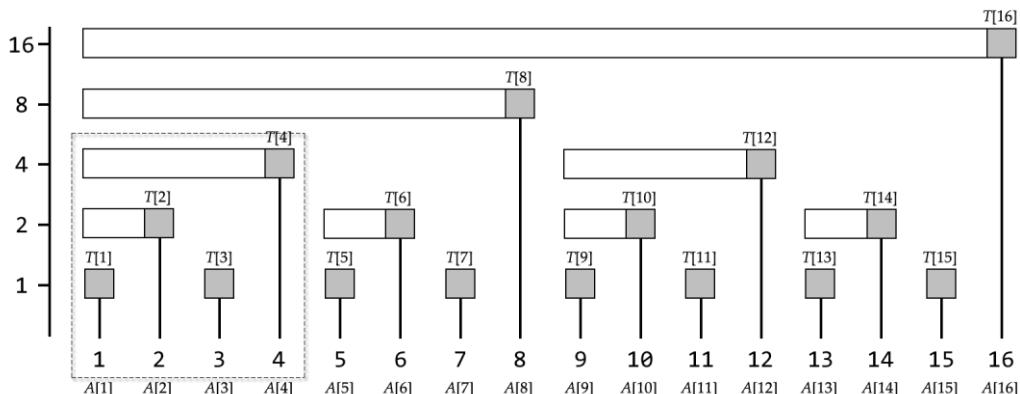


图 2-9 树状数组的简易表示，横坐标为数组 A 元素的序号 x ，纵坐标为 $\text{lowbit}(x)$ 。“长条矩形”下侧所覆盖的结点被“长条矩形”右端结点所累加（包括右端结点本身，即阴影方块）。例如，标记有虚线外框的树状数组元素 $T[4]$ ，其相应的“长条矩形”覆盖数组 A 的元素 $A[1], A[2], A[3], A[4]$ 所对应的结点，表示 $T[4]$ 累加了数组元素 $A[1], A[2], A[3], A[4]$ 。 $T[3]$ 只覆盖了数组元素 $A[3]$ ，表示 $T[3]$ 只累加了数组元素 $A[3]$ 。

如图 2-9 所示，这是一个长度为 16 的整数数组的树状数组表示，其中横坐标为数组 A 元素的序号 x （从 1 开始计数），纵坐标为 $\text{lowbit}(x)$ 。 $\text{lowbit}(x)$ 是一个函数，表示 x 的二进制表示中位于最右侧为 1 的位的权值。例如，十进制数 40_{10} 的二进制表示为 101000_2 ，其位于最右侧的 1 所在位的权值为 8，则 $\text{lowbit}(40_{10}) = 8$ 。应用位运算，可以巧妙地计算 $\text{lowbit}(x)$ 。

```

//+++++2.12.cpp+++++
inline int lowbit(int x) { return x & (-x); }

```

在图 2-9 中，“长条矩形”下方所覆盖的范围对应树状数组元素 $T[i]$ 所累加的原数组元素范围。例如，

$T[4]=A[1]+A[2]+A[3]+A[4]$, $T[6]=A[5]+A[6]$, $T[7]=A[7]$ 。

初始时构建（或者当 $A[i]$ 改变后需要更新）数组 T 的过程，可以看做是从数组 T 中序号为 x 的元素开始，不断向右向上“爬树”的过程，在“爬树”过程中更新中途所遇到结点的值。

```
const int MAXN = (1 << 10);
int T[MAXN + 16] = {};
// 将元素值的改变量累加到树状数组对应元素。
// x 表示元素的序号，delta 表示元素值的改变量。
void add(int x, int delta) {
    for (int i = x; i <= MAXN; i += lowbit(i))
        T[i] += delta;
}
```

对于 $A[1]$ 来说，在构建树状数组的过程中，其值被累加到树状数组元素 $T[1]$ 、 $T[2]$ 、 $T[4]$ 、 $T[8]$ 、 $T[16]$ 中，共累加了 5 次。可以证明，若原数组的长度为 n ，对于单个数组元素来说，将其累加到树状数组中的次数不会超过 $1+\log n$ 次，故构建整个树状数组的时间复杂度为 $O(n \log n)$ 。

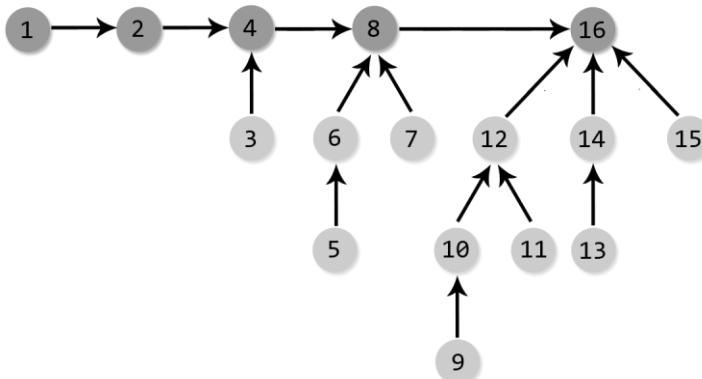


图 2-10 构建（更新）树状数组时，数组 A 中各个元素的“累加路径”。箭头所指方向表示数组 A 中元素所需要累加到的下一个树结点。例如， $A[3]$ 在累加时，经过 $T[3]$ 、 $T[4]$ 、 $T[8]$ 、 $T[16]$ 四个树结点； $A[5]$ 在累加时，经过 $T[5]$ 、 $T[6]$ 、 $T[8]$ 、 $T[16]$ 四个树结点

当数组 T 构建完毕后，给定区间 $[L, R]$ ，可以通过数组 T 计算“前缀和” $P[R]$ 和 $P[L-1]$ ，进而求出“范围和” $S[L, R] = P[R] - P[L-1]$ 。使用 **数组 T 构建数组 P** 的过程，可以看做是从数组 T 中序号为 x 的元素开始，不断向左向上“爬树”的过程，在“爬树”过程中累加中途所遇到数组 T 中结点的值。

```
// 通过累计相应的树状数组元素来确定“前缀和” P[x]。
int get(int x) {
    int sum = 0;
    for (int i = x; i; i -= lowbit(i))
        sum += T[i];
    return sum;
}
// 确定区间 [L, R] 的范围和。
int sum(int L, int R) { return get(R) - get(L - 1); }
```

以 $x=11$ 为例，在求和过程中依次累加了 $T[11]+T[10]+T[8]$ ，而 $T[11]=A[11]$, $T[10]=A[9]+A[10]$, $T[8]=A[1]+A[2]+\cdots+A[8]$ ，等效于累加了 $A[1]$ 到 $A[11]$ 的所有元素。

可以使用下述的代码对实现的正确性进行验证。

```
int main(int argc, char *argv[]) {
    // 将数组 A 赋值为从 1 开始的自然数序列。
    int A[MAXN + 16];
    for (int i = 1; i <= MAXN; i++) A[i] = i;
    // 初始化树状数组 T。
    for (int i = 1; i <= MAXN; i++) add(i, A[i]);
    // 验证结果是否正确。
    srand(time(NULL));
    for (int cases = 1; cases <= 100; cases++) {
        // 随机生成区间。
        int L = rand() % MAXN + 1, R = rand() % MAXN + 1;
        if (L > R) swap(L, R);
        cout << "S[" << setw(4) << right << L << ", ";
        cout << setw(4) << right << R << "] => ";
        // 使用树状数组 T 求范围和。
        cout << sum(L, R);
        // 由于是等差数列，其结果可利用等差数列的求和公式求得。
        cout << " = " << (R + L) * (R - L + 1) / 2 << '\n';
    }
    return 0;
}
```

需要注意，前述实现要求数组的元素从序号 1 开始存储，否则计算会发生错误，因为在更新数组 T 的过程中使用了 $i += \text{lowbit}(i)$ ，如果 i 为 0 则会陷入无限循环，但这并不表示树状数组就无法支持从 0 开始计数，可用使用下述实现来支持从 0 开始为数组 T 计数^[22]。

```
struct FenwickTree {
    int MAXN;
    vector<int> T;
    void initialize(int n) {
        this->MAXN = n;
        T.assign(n, 0);
    }
    int get(int x) {
        int sum = 0;
        for (; x >= 0; x = (x & (x + 1)) - 1)
            sum += T[x];
        return sum;
    }
    void add(int x, int delta) {
        for (; x < MAXN; x = x | (x + 1))
            T[x] += delta;
    }
    int sum(int L, int R) { return get(R) - get(L - 1); }
    void prepare(vector<int> A) {
        initialize(A.size());
        for (size_t i = 0; i < A.size(); i++)
            add(i, A[i]);
    }
};
//+++++++++++++++++++++++++++++++++++++++++++++++++2.12.cpp+++++++++++++++++//
```

在理解一维树状数组的基础上，可以很容易地将一维数组所对应的树状数组拓展到二维（或者多维）。

```

// 二维树状数组。
struct FenwickTree2D {
    int MAXN, MAXM;
    vector<vector<int>> T;

    // initialize(...) { ... }

    int get(int x, int y) {
        int sum = 0;
        for (int i = x; i >= 0; i = (i & (i + 1)) - 1)
            for (int j = y; j >= 0; j = (j & (j + 1)) - 1)
                sum += T[i][j];
        return sum;
    }

    void add(int x, int y, int delta) {
        for (int i = x; i < MAXN; i = i | (i + 1))
            for (int j = y; j < MAXM; j = j | (j + 1))
                T[i][j] += delta;
    }
};

```

1513 Movie Collection^D (电影收藏)

给定 n 盘电影录像带，编号从 1 到 n ，将其堆成一摞，编号为 n 的电影录像带在最下方，编号为 1 的电影录像带在最上方，每次从这一摞录像带中定位编号为 i 的电影录像带，将其抽出然后放置到顶端，要求你输出在抽出录像带之前，在编号为 i 的录像带上方的录像带的数量。

输入

在输入的第一行是一个正整数，表示测试数据的组数，最多有 100 组测试数据。每组测试数据的格式如下：第一行是两个整数 n 和 m ， $1 \leq n, m \leq 100000$ ，其中 n 表示电影录像带的数量， m 表示定位录像带的次数。接下来有 m 个整数 a_1, \dots, a_m ($1 \leq a_i \leq n$)，表示依次定位编号为 a_i 的录像带并将其放置到录像带堆的顶端。

输出

对于每组测试数据，输出 m 个整数，其中第 i 个整数表示在定位编号为 a_i 的录像带后，在将其放置到录像带堆的顶端之前，在编号为 a_i 的录像带的上方的录像带的数量。

样例输入

```

2
3 3
3 1 1
5 3
4 4 5

```

样例输出

```

2 1 0
3 0 4

```

分析

将问题转化为可以应用树状数组或者线段树进行解决的问题。设立数组 f ，将编号从 n 到 1 的电影带对应数组 f 序号从 1 到 n 的元素，即录像带 n 对应 $f[1]$ ，录像带 $n-1$ 对应 $f[2]$ ， \dots ，录像带 1 对应 $f[n]$ ，初始时， $f[1]$ 到 $f[n]$ 的值均设置为 1。令 $p[a]$ 表示编号为 a 的录像带所对应的数组元素序号，则有 $p[a] = n - a + 1$ ， $1 \leq a \leq n$ 。

给定编号为 a_i 的录像带，要求输出在其上方的录像带数量，可以转换为以下操作：编号为 a_i 的录像带对应的数组元素的序号为 $p[a_i]$ ，令数组 f 中从 $f[1]$ 到 $f[p[a_i]]$ 的元素中值为 1 的元素个数为 s ，则输出编号为 a_i 的录像带的上方的录像带的数量，等价于输出 $n - s$ 。

将编号为 a_i 的录像带放置到录像带堆的顶端，可以转换为以下操作：将编号为 a_i 的录像带在数组 f 中对应元素 $f[p[a_i]]$ 的值置为 0，然后从数组 f 的第 $n+1$ 个元素开始，寻找一个元素 $f[x]$ ，该元素尚未有录像带与其对应，将编号为 a_i 的录像带与 $f[x]$ 对应并将 $f[x]$ 的值置为 1，同时更新编号为 a_i 的录像带所对应的数组元素序号 $p[a_i]$ 为 x 。

参考代码

```
const int MAXN = 200010;

int n, m, f[MAXN], p[MAXN >> 1];

inline int lowbit(int x) { return x & (-x); }

void add(int x, int delta) {
    for (int i = x; i <= MAXN; i += lowbit(i)) f[i] += delta;
}

int get(int x) {
    int s = 0;
    for (int i = x; i -= lowbit(i)) s += f[i];
    return s;
}

int main(int argc, char *argv[]) {
    int T;
    cin >> T;
    for (int cs = 1; cs <= T; cs++) {
        cin >> n >> m;
        memset(f, 0, sizeof f);
        for (int i = 1; i <= n; i++) {
            p[i] = n - i + 1;
            add(i, 1);
        }
        int x = n + 1;
        for (int i = 0, ai; i < m; i++) {
            if (i) cout << ' ';
            cin >> ai;
            int s = get(p[ai]);
            cout << n - s;
            add(p[ai], -1);
            p[ai] = x++;
            add(p[ai], 1);
        }
        cout << '\n';
    }
    return 0;
}
```

强化练习：[11032 Function Overloading^D](#)，[11423* Cache Simulator^D](#)，[13095* Tobby and Query^D](#)。

扩展练习：[10909* Lucky Number^D](#)。

2.13 稀疏表

稀疏表(sparse table)也是一种应用于RMQ的数据结构,它只需要经过 $O(n\log n)$ 的时间进行预处理,然后就能够在 $O(1)$ 的时间内回答每个查询,其实现应用了倍增法的思想。

给定任意一个正整数 n ,可以将其唯一地表示成一个二进制数,例如,

$$25_{10} = 11001_2$$

因为二进制数中每个位的权值均为2的幂,也就是说,可以将 n 表示为一个递减序列的和,序列中每个元素均为2的幂,即

$$25 = 2^4 + 2^3 + 2^0 = 16 + 8 + 1$$

类似的,给定一个闭区间 $[L, R]$,可以将其唯一地表示为长度递减的若干子区间的并集,其中每个子区间的长度均为2的幂,例如,

$$[3, 27] = [3, 18] \cup [19, 26] \cup [27, 27]$$

区间总长度为25,其中每个子区间的长度依次为16,8,1。根据2的幂性质,给定长度为 n 的区间,至多包含 $\lceil \log_2 n \rceil$ 个这样的子区间。

根据区间划分的性质,稀疏表先进行预处理操作,预处理完成后会得到一张表,之后就可以基于这张表高效地完成各种查询。在预处理过程中,使用一个二维数组 st 来存储计算结果, $st[i][j]$ 存储的是长度为 2^j 的子区间 $[i, i+2^j-1]$ 的查询结果, st 的第一维大小为需要应用RMQ查询的数组长度 n ,第二维的大小为 K ,满足

$$K \geq \lceil \log_2 n \rceil + 1$$

对于一般的应用来说,若 $n=10^7$,则可取 $k=24$ 。根据2的幂性质,给定区间 $[i, i+2^j-1]$,可以将其等分为两个长度均为 2^{j-1} 的子区间 $[i, i+2^{j-1}-1]$ 和 $[i+2^{j-1}, i+2^j-1]$,假设当前需要查询数组 A 的区间最小值,应用动态规划思维,可以得到递推关系式

$$st[i][j] = \min(st[i][j-1], st[1 + (1 \ll (j-1))][j-1]), st[i][0] = A[i]$$

应用上述递归关系式,可以在 $O(n\log n)$ 的时间内完成 st 数组的构建。对于给定需要查询最小值的区间 $[L, R]$,由于

$$[L, R] = [L, R - 2^j] \cup [R - 2^j + 1, R], j = \log_2(R - L + 1)$$

则

$$\min(A[L, R]) = \min(st[L][j], st[R - 2^j + 1][j]), j = \log_2(R - L + 1)$$

由于求最小值时需要先确定区间长度的对数值,相较于每次求值时现场计算,可以预先计算区间长度的对数表以备用。

```
//+++++++
// 2.13.cpp
const int MAXN = (1 << 20), K = 24;
int N, A[MAXN], log2t[MAXN + 1] = {}, st[MAXN][K] = {};

// 构建稀疏表。
void prepare() {
    // 预先计算对数值。
    log2t[1] = 0;
    for (int i = 2; i <= N; i++) log2t[i] = log2t[i >> 1] + 1;
    // 为边界赋值。
    for (int i = 0; i < N; i++) st[i][0] = A[i];
    // 根据递推关系式求区间最值。
}
```

```

    for (int j = 1; j < K; j++)
        for (int i = 0; i + (1 << j) <= N; i++)
            st[i][j] = min(st[i][j - 1], st[i + (1 << (j - 1))][j - 1]);
}

// 查询, L 为区间的起始位置, R 为区间的结束位置。
int query(int L, int R) {
    int j = log2t[R - L + 1];
    return min(st[L][j], st[R - (1 << j) + 1][j]);
}

```

可以将稀疏表用于求“范围和”，只需在预处理时将取最小值操作更换为求和操作，在求取给定区间 $[L, R]$ 的范围和时，从大到小遍历 2 的幂，当发现 2 的 j 次幂满足

$$2^j \leq (R - L + 1)$$

则先将区间 $[L, L + 2^j - 1]$ 内的值加入总和中，继续对区间 $[L + 2^j, R]$ 进行求和。

```

int query(int L, int R) {
    int sum = 0;
    for (int j = K; j >= 0; j--) {
        if ((1 << j) <= R - L + 1) {
            sum += st[L][j];
            L += 1 << j;
        }
    }
    return sum;
}
//++++++2.13.cpp+++++++

```

强化练习：[11491 Erasing and Winning](#)。

2.14 根号分块

根号分块 (sqrt decomposition)，类似于稀疏表，也是一种通过预处理来提高查询效率的数据结构（或者说“技巧”），它能够以 $O(\sqrt{n})$ 的时间复杂度完成诸如区间最大/小值查询、区间求和的操作。其核心思想是将待处理序列分割为大小相同的“块”，在块中进行指定操作，便于提高速度。

2.14.1 根号分块的操作

给定长度为 n 的数组，数组元素为 $a[1], a[2], \dots, a[n]$ ，给定区间 $[L, R]$ ， $L \leq R$ ，要求确定元素 $a[L]$ 至 $a[R]$ 的和。朴素的方法是逐个累加区间 $[L, R]$ 内的所有元素，很明显，当查询较多时效率很低。令 $s = \lceil \sqrt{n} \rceil$ ，将数组 a 按照每 s 个元素一组分成若干块，那么每个元素都会被划分到某个块内。由于 n 不一定是 s 的整数倍，因此有可能最后一个块不足 s 个元素。在完成块的划分后，累加每个块内元素的和，可以得到一个块内元素和数组 sum ， $sum[i]$ 表示第 i 个块内所有元素的和。当对区间 $[L, R]$ 内的元素求和时，与普通的逐个累加不同，可以将区间表示为若干个完整块和至多两个不完整块的并。例如，令 $n = 200$ ，则 $s = 15$ ，于是数组被划分为 14 个块，第 1 块至第 13 块的大小均为 15，第 14 块的大小为 5。当求区间 $[24, 110]$ 的元素和时，可以将其分解为第 2 块的后 7 个元素，整个第 3 块、第 4 块、第 5 块、第 6 块、第 7 块，第 8 块的前 5 个元素，则区间 $[24, 110]$ 的元素和 S 可以表示为

$$S = \left(\sum_{i=24}^{30} a[i] \right) + \left(\sum_{i=3}^7 sum[i] \right) + \left(\sum_{i=106}^{110} a[i] \right)$$

由于第 3 块至第 7 块的元素和已经在预处理阶段求得，因此只需 $O(1)$ 的时间复杂度获取，而前后两个不完

整块的元素和计算，至多需要 $O(\sqrt{n})$ 的时间，因此总的时间复杂度可以优化为 $O(\sqrt{n})$ 。如果在后续的过程中对元素 $a[i]$ 进行了更改，只需将第 i 个元素所属的“块内和” $sum[j]$ 做相应更改即可用于后续的求和，而这个更改很容易做到。

```

//++++++++++++++++++++++++++2.14.1.cpp+++++++++++++++++++++++++
const int MAXN = 10000010, MAXB = 10010;

int n, s;
int a[MAXN], link[MAXN], sum[MAXB];

// 查询区间 [L, R] 的元素和。
int query(int L, int R) {
    int p = link[L], q = link[R];
    int r = 0;
    for (int i = L; i <= min(R, p * s); i++) r += a[i];
    for (int i = p + 1; i < q; i++) r += sum[i];
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) r += a[i];
    }
    return r;
}

int main(int argc, char *argv[]) {
    cin >> n;
    // 确定分块的大小。
    s = sqrt(n) + 1;
    for (int i = 1; i <= s; i++) sum[i] = 0;
    // 读入数据，数组元素的序号从 1 开始计数。
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        // 确定第 i 个元素所属的分块。
        link[i] = (i - 1) / s + 1;
        // 将第 i 个元素累加到对应的块内和。
        sum[link[i]] += a[i];
    }
    int m, L, R;
    cin >> m;
    for (int i = 0; i < m; i++) {
        cin >> L >> R;
        // 查询区间元素和。
        cout << query(L, R) << '\n';
    }
    return 0;
}

```

如果对数组的元素进行更新，则不仅需要更新数组元素的值，还需要更新数组元素所在分块的块内和，以便后续在统计时提高效率。

```
// 将序号为 L 的数组元素值更新为 x。
void update(int L, int x) {
    sum[link[L]] -= a[L];
    a[L] = x;
    sum[link[L]] += a[L];
}
//+++++++++++++++++2.14.1.cpp+++++++++++++++++
```

从上述根号分块的实现不难看出, 该数据结构实际上可以看做是在普通的暴力计算的基础上的一种优化, 它尽可能地减少了对单个元素进行累加的次数, 从而在一定程度上提高了效率。由于分块过大或者过小都对效率有不利影响, 为了保证效率, 在一般情况下, 取分块的大小 $s = \lceil \sqrt{n} \rceil$, 可以使得操作的平摊时间复杂度保持在 $O(\sqrt{n})$ 。

2.14.2 根号分块的应用

根号分块作为分块算法中的一种, 不仅可以查询区间和、区间最大值/最小值, 经过适当拓展能够处理一些线段树不便于处理的问题。由于根号分块的核心在于分块, 因此在进行处理时需要将已经处理过的块加上适当的标记, 以便后续处理时直接获得结果而不需要再次进行处理。下面, 我们结合实例来介绍若干根号分块的应用¹。

(1) 给定由 n 个整数构成的数组 A , 要求完成两种操作: 第一种是将区间 $[l, r]$ 内的所有元素都加上 x , 第二种是查询数组中序号为 x 的元素的值。对数组进行分块处理后, 为第 i 个分块设置标记 $added[i]$, 表示第 i 个分块的累加值, 在进行第一种操作时, 如果是对整个第 i 分块内的所有元素进行累加, 则只需将分块所对应的标记 $added[i]$ 加上 x 。否则, 所操作的是边角分块的元素, 则直接将对应的元素加上 x 即可。在查询时, 将指定序号元素的当前值加上其所在分块的累加值输出即可。

```
//-----2.14.2.1.cpp-----//
const int MAXN = 50010, MAXB = 256;

// n 为数组的小, s 为分块的大小, a 存储数组 A 的元素, link 记录序号为 i 的元素所对应的块序号。
int n, s, a[MAXN], link[MAXN];
// added 记录块内累加值。
int added[MAXB];

// 区间更新。
void update(int L, int R, int x) {
    int p = link[L], q = link[R];
    // 边角块内的元素直接更新值。
    for (int i = L; i <= min(R, p * s); i++) a[i] += x;
    // 块内更新累加值。
    for (int i = p + 1; i < q; i++) added[i] += x;
    // 边角块内的元素直接更新值。
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) a[i] += x;
    }
}

int main(int argc, char *argv[]) {
    cin >> n;
    // 确定分块的大小。
    s = sqrt(n) + 1;
    // 读入数据, 数组元素的序号从 1 开始计数。
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
    }
}
```

¹ 此处介绍的九种应用依次对应 LibreOJ 上题号 6277 至 6285 的九道题目。LibreOJ 官方网站: <https://loj.ac/>。

```

}
int c, L, R, x;
for (int i = 0; i < n; i++) {
    cin >> c >> L >> R >> x;
    if (c == 0) update(L, R, x);
    else cout << a[R] + added[(R - 1) / s + 1] << '\n';
}
return 0;
}
//-----2.14.2.1.cpp-----//

```

(2) 给定由 n 个整数构成的数组 A , 要求完成两种操作: 第一种是将区间 $[l, r]$ 内的所有元素都加上 x , 第二种是查询给定区间 $[l, r]$ 内小于 x 的元素个数。由于需要查询小于 x 的元素个数, 如果使用朴素的线性扫描, 时间复杂度为 $O(n^2)$ 。若块内的元素有序, 则可以使用二分查找来确定块内第一个大于或等于 x 的元素值的序号, 通过该序号即可确定该块内小于 x 的元素值个数, 因此我们可以额外维护一个数据结构, 该数据结构所存储的是对应块内元素的有序列表。当更新发生在整个块时, 只需更新累加值, 对应块的有序列表不需改动, 其相对顺序不会发生改变。如果更新发生在边角块上, 直接对单个元素进行更新值的操作, 然后需要对此边角块内的所有元素进行重新排序, 以便后续能够通过二分查找来确定小于 x 的元素值个数。在上述参考实现中, 令每个分块的大小为 B , 则查询的时间复杂度为 $O(n\sqrt{B}\log n)$, 如果将分块 B 设置为 \sqrt{n} , 则时间复杂度 $O(n\sqrt{n}\log n)$ 。可以进一步优化, 将分块 B 设置为 $\sqrt{\frac{n}{\log n}}$, 则时间复杂度为 $O(n\sqrt{n\log n})$ 。根据题目的特定情况适当设置分块的大小以提高效率是根号分块应用中一个值得注意的技巧。

```

//-----2.14.2.2.cpp-----//
const int MAXN = 50010, MAXB = 256;

int n, s, a[MAXN], link[MAXN];
// sorted 记录块内元素的排序列表, cnt 记录块内元素的个数。
int sorted[MAXB][MAXB], cnt[MAXB];
// added 记录块内累加值。
int added[MAXB];

int query(int L, int R, int x) {
    int p = link[L], q = link[R];
    int r = 0;
    for (int i = L; i <= min(R, p * s); i++) r += (a[i] + added[p] < x);
    // 利用算法库函数 lower_bound 查找符合要求的元素值的个数。
    for (int i = p + 1; i < q; i++)
        r += lower_bound(sorted[i], sorted[i] + cnt[i], x - added[i]) - sorted[i];
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) r += (a[i] + added[q] < x);
    }
    return r;
}

// 当块内元素的值发生改变后, 需要对整个块内的元素进行重排序, 以便后续的查询。
void resort(int block) {
    cnt[block] = 0;
    for (int i = (block - 1) * s + 1; i <= min(block * s, n); i++)
        sorted[block][cnt[block]++] = a[i];
    sort(sorted[block], sorted[block] + cnt[block]);
}

```

```

// 更新块内元素。整块更新只需更新累加值，边角块更新直接更新元素值。
void update(int L, int R, int x) {
    int p = link[L], q = link[R];
    for (int i = L; i <= min(R, p * s); i++) a[i] += x;
    resort(p);
    for (int i = p + 1; i < q; i++) added[i] += x;
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) a[i] += x;
        resort(q);
    }
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n / log2(n)) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
        sorted[link[i]][cnt[link[i]]++] = a[i];
    }
    int blocks = (n - 1) / s + 1;
    for (int i = 1; i <= blocks; i++) sort(sorted[i], sorted[i] + cnt[i]);
    int c, L, R, x;
    for (int i = 0; i < n; i++) {
        cin >> c >> L >> R >> x;
        if (c == 0) update(L, R, x);
        else { x *= x; cout << query(L, R, x) << '\n'; }
    }
    return 0;
}
//-----2.14.2.2.cpp-----

```

(3) 给定由 n 个非负整数构成的数组 A ，要求完成两种操作：第一种是将区间 $[l, r]$ 内的所有元素都加上 x ，第二种是查询给定区间 $[l, r]$ 内小于 x 的最大元素的值。与前述查询给定区间内小于 x 的元素类似，仍然保持块内元素有序，然后使用库函数 `lower_bound` 辅助进行查询即可，只需对查询的实现进行适当修改即可。

```

//-----2.14.2.3.cpp-----
int query(int L, int R, int x) {
    int p = link[L], q = link[R];
    int r = -1;
    for (int i = L; i <= min(R, p * s); i++) {
        if (a[i] + added[p] < x)
            r = max(r, a[i] + added[p]);
    }
    // 确定块内第一个大于等于 x 的元素位置。
    for (int i = p + 1; i < q; i++) {
        int idx =
            lower_bound(sorted[i], sorted[i] + cnt[i], x - added[i]) - sorted[i];
        if (idx) r = max(r, sorted[i][idx - 1] + added[i]);
    }
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) {
            if (a[i] + added[q] < x)
                r = max(r, a[i] + added[q]);
        }
    }
}

```

```

    return r;
}
//-----2.14.2.3.cpp-----//

```

(4) 给定由 n 个整数构成的数组 A , 要求完成两种操作: 第一种是将区间 $[l, r]$ 内的所有元素都加上 x , 第二种是查询给定区间 $[l, r]$ 内所有元素值之和模 x 之后的值。维护第 i 个分块内和标记 $sum[i]$ 、块内累加值标记 $added[i]$, 按照前述所介绍的方法进行区间更新, 在输出结果时取模即可, 只需对查询函数进行适当修改即可。

```

//-----2.14.2.4.cpp-----//
const int MAXN = 50010, MAXB = 256;

int n, s, a[MAXN], link[MAXN];
// added 记录块内元素的累加值, sum 记录块内元素的和。
long long added[MAXB], sum[MAXB];
// 区间查询。
int query(int L, int R, int x) {
    int p = link[L], q = link[R];
    long long r = 0;
    // 边角块内元素, 将当前值和块内累加值加入到总和中。
    for (int i = L; i <= min(R, p * s); i++) r += a[i] + added[p], r %= x;
    // 整块元素, 直接将块内和加入总和即可。
    for (int i = p + 1; i < q; i++) r += sum[i], r %= x;
    // 边角块内元素, 将当前值和块内累加值加入到总和中。
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) r += a[i] + added[q], r %= x;
    }
    return r;
}

// 当单个元素更新时, 相应更新块内和。
void modify(int block, int idx, int x) {
    sum[block] -= a[idx];
    a[idx] += x;
    sum[block] += a[idx];
}

// 区间更新。
void update(int L, int R, int x) {
    int p = link[L], q = link[R];
    // 对于边角块内的元素, 直接更新元素的值和块内和。
    for (int i = L; i <= min(R, p * s); i++) modify(p, i, x);
    // 整块元素, 更新累加值, 更新块内和。
    for (int i = p + 1; i < q; i++) added[i] += x, sum[i] += x * s;
    // 对于边角块内的元素, 直接更新元素的值和块内和。
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) modify(q, i, x);
    }
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n) + 1;
    for (int i = 1; i <= n; i++) {

```

```

    cin >> a[i];
    link[i] = (i - 1) / s + 1;
    sum[link[i]] += a[i];
}
int c, L, R, x;
for (int i = 0; i < n; i++) {
    cin >> c >> L >> R >> x;
    if (c == 0) update(L, R, x);
    else { x += 1; cout << query(L, R, x) << '\n'; }
}
return 0;
}
//-----2.14.2.4.cpp-----//

```

(5) 给定由 n 个非负整数构成的数组 A ，要求完成两种操作：第一种是将区间 $[l, r]$ 内的所有元素值开方后向下取整，第二种是查询给定区间 $[l, r]$ 内所有元素值之和。将给定区间内的元素开根号并向下取整，若该元素的值大于 1，则元素在进行操作后的值为原值的一半左右，易知在若干次操作后该区间的元素都将变为 1（或 0），继续开根号向下取整不会改变元素的值。因此只要某个块内的元素均为 1（或 0）之后，就可以给这个块加上标记，表示不需要再对此块进行处理，从而能够提高更新的效率。因此除了维护块内和标记外，再维护一个块内值是否不再改变的标记即可。

```

//-----2.14.2.5.cpp-----//
const int MAXN = 50010, MAXB = 256;

// flag 标记某个块内的元素是否已经“稳定”，即所有元素进行操作后值不会发生改变。
int flag[MAXB];
int n, s, a[MAXN], link[MAXN], sum[MAXB];

// 查询区间  $[L, R]$  内元素的和。
int query(int L, int R) {
    int p = link[L], q = link[R];
    int r = 0;
    for (int i = L; i <= min(R, p * s); i++) r += a[i];
    for (int i = p + 1; i < q; i++) r += sum[i];
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) r += a[i];
    }
    return r;
}

// 设置某个块内单个元素的值并更新块内和。
void modify(int block, int idx) {
    sum[block] -= a[idx];
    a[idx] = (int)sqrt(a[idx]);
    sum[block] += a[idx];
}

// 更新区间  $[L, R]$  内元素的值，需要更新对应的块内和。
void update(int L, int R) {
    int p = link[L], q = link[R];
    for (int i = L; i <= min(R, p * s); i++) modify(p, i);
    for (int i = p + 1; i < q; i++) {
        // 当块内元素已经“稳定”，不再处理。
        if (flag[i]) continue;
        int r = query(i, i + s - 1);
        if (r == a[i]) {
            flag[i] = 1;
            for (int j = i; j <= i + s - 1; j++) a[j] = 1;
        } else {
            a[i] = r;
            sum[i] = r;
        }
    }
}

```

```

flag[i] = 1;
for (int j = (i - 1) * s + 1; j <= i * s; j++) {
    // 当元素的值大于1时, 块不“稳定”。
    if (a[j] > 1) flag[i] = 0;
    modify(i, j);
}
if (p != q) {
    for (int i = (q - 1) * s + 1; i <= R; i++) modify(q, i);
}
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
        sum[link[i]] += a[i];
    }
    int c, L, R, x;
    for (int i = 0; i < n; i++) {
        cin >> c >> L >> R >> x;
        if (c == 0) update(L, R);
        else cout << query(L, R) << '\n';
    }
    return 0;
}
//-----2.14.2.5.cpp-----//

```

(6) 给定由 n 个整数构成的数组 A , 要求完成两种操作: 第一种是在序号为 x 的数组元素前插入数 y , 第二种是查询序号为 x 的数组元素值。由于插入数之后会导致插入位置之后的数的序号发生改变, 为了适应这种改变, 块内存储数据的结构最好能够支持动态增加和删除, 因此使用 STL 的向量容器类 (vector) 是一种合适的选择。当存储块内数据的向量大小因为多次插入数据而变得较大时, 为了保证查询的效率, 需要对所有数组元素重新分块。

```

//-----2.14.2.6.cpp-----//
const int MAXN = 100010, MAXB = 10010;

int n, m, s, a[MAXN], link[MAXN];
// 每个分块内的数组元素使用向量容器类予以存在, 便于增加和删除。
vector<int> block[MAXB];

// 查询当前序号为 L 的数组元素值。
int query(int L) {
    int i = 1;
    while (i <= s && L > (int)block[i].size()) L -= block[i++].size();
    return block[i][L - 1];
}

// 当某个块因为多次的插入操作变得较大时, 为了保证查询效率, 需要进行重新分块的操作。
void rebuild() {
    vector<int> tmp;
    for (int i = 1; i <= s; i++) {
        tmp.insert(tmp.end(), block[i].begin(), block[i].end());
    }
    block = tmp;
    s = tmp.size();
}
```

```

        block[i].clear();
    }
    s = sqrt(m) + 1;
    for (int i = 1; i <= m; i++) block[(i - 1) / s + 1].push_back(tmp[i - 1]);
}

// 在序号为 L 的元素之前插入数值 x。
void update(int L, int x) {
    int i = 1;
    while (i <= s && L > (int)block[i].size()) L -= block[i++].size();
    block[i].insert(block[i].begin() + L - 1, x);
    if ((int)block[i].size() > 10 * s) rebuild();
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
        block[link[i]].push_back(a[i]);
    }
    m = n;
    int c, L, R, x;
    for (int i = 0; i < n; i++) {
        cin >> c >> L >> R >> x;
        if (c == 0) { update(L, R); m++; }
        else cout << query(R) << '\n';
    }
    return 0;
}
//-----2.14.2.6.cpp-----//

```

(7) 给定由 n 个整数构成的数组 A , 要求完成三种操作: 第一种是将区间 $[l, r]$ 内的所有元素都加上 x , 第二种是将区间 $[l, r]$ 内的所有元素都乘以 x , 第三种是查询序号为 x 的数组元素模 10007 后的值。由于更新操作涉及到加和乘两种操作, 因此需要维护两个标记, 一个标记是第 i 个分块的累加值 $added[i]$, 另外一个标记是第 i 个分块的累乘值 $multiplied[i]$ 。当查询序号为 x 的某个数组元素的值时, 假定其位于第 j 个分块内, 需要将其当前值 $A[x]$ 乘以所在块内的累乘值 $multiplied[j]$ 然后再加上累加值 $added[j]$ 后才是数组元素 $A[x]$ 的真实值。类似的, 在更新时, 如果是将整个块都乘以 x , 则不仅块内的累乘值 $multiplied[i]$ 要乘以 x , 块内的累加值 $added[i]$ 也要乘以 x 。对于边角块内的元素, 由于更新的两种操作无法直接进行叠加, 需要计算所在块的所有元素的真实值之后重新设置边角块内的累加值和累乘值标记。

```

//-----2.14.2.7.cpp-----//
const int MAXN = 100010, MAXB = 320, MOD = 10007;

int n, s, a[MAXN], link[MAXN];
// added 记录块内元素的累加值, multiplied 记录块内元素的累乘值。
long long added[MAXB], multiplied[MAXB];

// 查询序号为 L 的元素值。
int query(int L) {
    int p = link[L];
    return (a[L] * multiplied[p] % MOD + added[p]) % MOD;
}

```

```

}

// 将区间 [L, R] 内的元素加上 x。
void add(int L, int R, int x) {
    int p = link[L], q = link[R];
    // 边角块内的元素单独处理后重置块内标记。
    for (int i = (p - 1) * s + 1; i <= min(n, p * s); i++)
        a[i] = (a[i] * multiplied[p] % MOD + added[p]) % MOD;
    for (int i = L; i <= min(R, p * s); i++) a[i] += x, a[i] %= MOD;
    added[p] = 0, multiplied[p] = 1;
    // 整块统一处理。
    for (int i = p + 1; i < q; i++) added[i] += x, added[i] %= MOD;
    // 边角块内的元素单独处理后重置块内标记。
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= min(n, q * s); i++)
            a[i] = (a[i] * multiplied[q] % MOD + added[q]) % MOD;
        for (int i = (q - 1) * s + 1; i <= R; i++) a[i] += x, a[i] %= MOD;
        added[q] = 0, multiplied[q] = 1;
    }
}

// 将区间 [L, R] 内的元素乘以 x。
void multiply(int L, int R, int x) {
    int p = link[L], q = link[R];
    // 边角块内的元素单独处理后重置块内标记。
    for (int i = (p - 1) * s + 1; i <= min(n, p * s); i++)
        a[i] = (a[i] * multiplied[p] % MOD + added[p]) % MOD;
    for (int i = L; i <= min(R, p * s); i++) a[i] *= x, a[i] %= MOD;
    added[p] = 0, multiplied[p] = 1;
    // 整块统一处理。
    for (int i = p + 1; i < q; i++) {
        added[i] *= x, added[i] %= MOD;
        multiplied[i] *= x, multiplied[i] %= MOD;
    }
    // 边角块内的元素单独处理后重置块内标记。
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= min(n, q * s); i++)
            a[i] = (a[i] * multiplied[q] % MOD + added[q]) % MOD;
        for (int i = (q - 1) * s + 1; i <= R; i++) a[i] *= x, a[i] %= MOD;
        added[q] = 0, multiplied[q] = 1;
    }
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
    }
    for (int i = 1; i <= s; i++) multiplied[i] = 1;
    int c, L, R, x;
    for (int i = 0; i < n; i++) {
        cin >> c >> L >> R >> x;
        if (c == 0) add(L, R, x);
        else if (c == 1) multiply(L, R, x);
    }
}

```

```

        else cout << query(R) << '\n';
    }
    return 0;
}
//-----2.14.2.7.cpp-----//

```

(8) 给定由 n 个整数构成的数组 A ，要求完成以下操作：查询区间 $[l, r]$ 内等于 x 的数组元素的个数，然后将区间 $[l, r]$ 内的元素值均设置为 x 。为了提高效率，可以为第 i 个分块设置标记 $setted[i]$ 和 $value[i]$ ，其中 $setted[i]$ 表示第 i 个分块是否已经设置为同一个值， $value[i]$ 表示在已经将第 i 个分块设置为同一个值的情况下该值的大小。在统计边角块时，先根据分块的标记更新分块内数组元素的值，然后再统计符合要求的数组元素个数。在统计整个分块时，如果分块内的数组元素已经被设置为同一个值，则可以统一更改标记以设置块内元素的值，而不需要逐个数组元素进行设置，从而提高效率。

```

//-----2.14.2.8.cpp-----//
const int MAXN = 100010, MAXB = 320;

int n, s, a[MAXN], link[MAXN], setted[MAXB], value[MAXB];

// 根据标记更新块内元素值，重置块内标记。
void update(int b) {
    if (!setted[b]) return;
    for (int i = (b - 1) * s + 1; i <= min(n, b * s); i++)
        a[i] = value[b];
    setted[b] = 0;
}

int query(int L, int R, int x) {
    int r = 0;
    int p = link[L], q = link[R];
    // 对于边角块，先更新块内元素然后再统计。
    update(p);
    for (int i = L; i <= min(R, p * s); i++) {
        if (a[i] == x) r++;
        a[i] = x;
    }
    // 整块更新。
    for (int i = p + 1; i < q; i++) {
        if (setted[i]) {
            if (value[i] == x) r += s;
            value[i] = x;
        } else {
            for (int j = (i - 1) * s + 1; j <= i * s; j++)
                if (a[j] == x) r++;
            setted[i] = 1;
            value[i] = x;
        }
    }
    // 对于边角块，先更新块内元素然后再统计。
    if (p != q) {
        update(q);
        for (int i = (q - 1) * s + 1; i <= R; i++) {
            if (a[i] == x) r++;
            a[i] = x;
        }
    }
}
```

```

    }
    return r;
}

int main(int argc, char *argv[]) {
    cin >> n;
    s = sqrt(n) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        link[i] = (i - 1) / s + 1;
    }
    int L, R, x;
    for (int i = 0; i < n; i++) {
        cin >> L >> R >> x;
        cout << query(L, R, x) << '\n';
    }
    return 0;
}
//-----2.14.2.8.cpp-----//

```

(9) 给定由 n 个数整数构成的数列 A ，查询区间 $[l, r]$ 内的最小众数。众数（mode）是指一组数中出现次数最多的数。给定一组数，可能存在多个众数。由于需要统计每个数出现的频次，如果每次查询时全部统计将超时，需要进行预处理，即预处理得到第 i 个分块到第 j 个分块之间的最小众数，当查询给定区间的众数时，边角块内的元素单独统计频次，整块内的元素直接利用预处理结果得到最小众数，然后再与边角块内的元素频次进行比较即可。

```

//-----2.14.2.9.cpp-----//
const int MAXN = 100010, MAXB = 1300;

// mode 记录第 i 个分块到第 j 个分块的最小众数。
int n, nn, s, a[MAXN], b[MAXN], link[MAXN], mode[MAXB][MAXB], cnt[MAXN];
// id 保存 A 中每个元素在数组中出现的序号。
vector<int> idx[MAXN];
int tmp[MAXB * 2], tot;

int query(int L, int R) {
    tot = 0;
    int p = link[L], q = link[R];
    // 记录边角块内出现的数组元素。
    for (int i = L; i <= min(R, p * s); i++) tmp[tot++] = a[i];
    if (p != q) {
        for (int i = (q - 1) * s + 1; i <= R; i++) tmp[tot++] = a[i];
    }
    // 如果有连续的整块，则将该连续整块的最小众数予以记录，以便统一比较。
    if (p + 2 <= q) tmp[tot++] = mode[p + 1][q - 1];
    // 根据保存的每个元素的序号数组查询区间内相应元素的数量，进而比较得到最小众数。
    int m = -1, c = 0;
    for (int i = 0; i < tot; i++) {
        int x = tmp[i];
        // 利用二分查找确定区间 [L, R] 内元素 x 出现的频次。
        int tot = upper_bound(idx[x].begin(), idx[x].end(), R) -
            lower_bound(idx[x].begin(), idx[x].end(), L);
        if (tot > c || (tot == c && x < m)) {
            c = tot;
            m = x;
        }
    }
}

```

```

    }
}

return b[m];
}

int main(int argc, char *argv[]) {
    cin >> n;
    // 适当设置分块的大小以提高查询效率。
    s = sqrt(n / log2(n)) + 1;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        b[i - 1] = a[i];
        link[i] = (i - 1) / s + 1;
    }
    // 利用离散化技巧，对原数组的元素进行序号压缩以便后续处理。
    sort(b, b + n);
    nn = unique(b, b + n) - b;
    for (int i = 1; i <= n; i++) {
        a[i] = lower_bound(b, b + nn, a[i]) - b;
        idx[a[i]].push_back(i);
    }
    // 预处理得到第 i 个分块到第 j 个分块之间数组元素的最小众数。
    int blocks = (n - 1) / s + 1;
    for (int i = 1; i <= blocks; i++) {
        int m = -1, c = 0;
        for (int j = 0; j < nn; j++) cnt[j] = 0;
        for (int j = i; j <= blocks; j++) {
            for (int k = (j - 1) * s + 1; k <= min(n, j * s); k++) {
                cnt[a[k]]++;
                if (cnt[a[k]] > c || (cnt[a[k]] == c && a[k] < m)) {
                    c = cnt[a[k]];
                    m = a[k];
                }
            }
        }
        mode[i][j] = m;
    }
}
int L, R;
for (int i = 0; i < n; i++) {
    cin >> L >> R;
    cout << query(L, R) << '\n';
}
return 0;
}
//-----2.14.2.9.cpp-----//

```

12003 Array Transformer^D (数组变换器)

编写程序，根据 m 条指令对数组 $A[1], A[2], \dots, A[n]$ 进行变换。每条指令 (L, R, v, p) 的含义为：首先确定从数组元素 $A[L]$ 到 $A[R]$ (包括 $A[L]$ 和 $A[R]$) 中严格小于 v 的数组元素个数 k ，然后将数组元素 $A[p]$ 的值更改为 $u \times k / (R - L + 1)$ 。这里的除法使用整除，即忽略小数部分。

输入

输入的第一行包含三个整数 n, m, u ($1 \leq n \leq 300000, 1 \leq m \leq 50000, 1 \leq u \leq 1000000000$)。接下来的 n 行，每行包含一个整数 $A[i]$ ($1 \leq A[i] \leq u$)。再接下来的 m 行，每行包含一条指令，每条指令包含四个整

数 L, R, v, p ($1 \leq L \leq R \leq n, 1 \leq v \leq u, 1 \leq p \leq n$)。

输出

输出包含 n 行，每行一个整数，表示最终数组的值。

对于样例输入来说，总共只有一条指令： $L=2, R=10, v=6, p=10$ ，总共有 4 个数组元素 (2, 3, 4,

5) 小于 6，故 $k=4$ ，则数组元素 $A[10]$ 的新值为 $11 \times 4 / (8 - 2 + 1) = 44 / 7 = 6$ 。

样例输入

```
10 1 11
1
2
3
4
5
6
7
8
9
10
2 8 6 10
```

样例输出

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

分析

由于查询数量较多，如果使用朴素的暴力统计，肯定会超时。考虑分块处理，每个块内对元素进行排序，以便使用库函数 `lower_bound` 查找小于 v 的元素个数。由于对块内元素进行排序后会影响元素在原数组中的序号，为了便于得到原序号所对应的数组元素，可将原数组做一个备份，以避免后续保持块内有序对原数组元素位置带来的影响。

参考代码

```
const int MAXN = 300010, MAXM = 1010;

int n, m, u, s, link[MAXN], A[MAXN], B[MAXN];
int head[MAXM], tail[MAXM];

// 分块。确定各块边界，对块内元素进行排序以便查找。
void build() {
    s = sqrt(n);
    int block = n / s;
    if (block * s < n) block++;
    for (int i = 1; i <= block; i++) head[i] = (i - 1) * s + 1, tail[i] = i * s;
    tail[block] = n;
    for (int i = 1; i <= block; i++) {
        for (int j = head[i]; j <= tail[i]; j++) link[j] = i;
        sort(A + head[i], A + tail[i] + 1);
    }
}

// 查询。使用库函数 lower_bound 提高查找效率。
int query(int L, int R, int v) {
    int k = 0;
    for (int i = L; i <= min(R, tail[link[L]]); i++) k += (B[i] < v);
    for (int i = link[L] + 1; i < link[R]; i++)
        for (int j = head[i]; j <= tail[i]; j++) k += (link[j] < v);
    return k;
}
```

```

        k += lower_bound(A + head[i], A + tail[i] + 1, v) - A - head[i];
    if (link[L] != link[R])
        for (int i = head[link[R]]; i <= R; i++) k += (B[i] < v);
    return k;
}

void update(int L, int R, int p, int k) {
    int belong = link[p];
    // 查找已排序数组中具有对应元素值的元素位置。
    int pp = lower_bound(A + head[belong], A + tail[belong] + 1, B[p]) - A;
    B[p] = (long long)(u) * k / (R - L + 1);
    // 使用插入排序，保持块内有序以便后续进行查找。
    while (pp > head[belong] && B[p] < A[pp - 1]) { swap(A[pp - 1], A[pp]); pp--; }
    while (pp < tail[belong] && B[p] > A[pp + 1]) { swap(A[pp + 1], A[pp]); pp++; }
    A[pp] = B[p];
}

int main(int argc, char *argv[]) {
    cin >> n >> m >> u;
    // A 为已排序数组, B 为未排序数组。
    for (int i = 1; i <= n; i++) {
        cin >> A[i]; B[i] = A[i];
    }
    build();
    for (int i = 1, L, R, v, p; i <= m; i++) {
        cin >> L >> R >> v >> p;
        update(L, R, p, query(L, R, v));
    }
    for (int i = 1; i <= n; i++) cout << B[i] << '\n';
    return 0;
}

```

扩展练习：11990* Dynamic Inversion^D。

2.15 块状链表

在前述的根号分块数据结构中，我们使用数组来记录各个分块，使用数组来记录分块的好处是寻找某个元素的位置只需 $O(1)$ 的时间，但是增加或者删除数据时，由于需要移动后续的数据，因此时间复杂度是 $O(n)$ 。块状链表是另外一种使用分块技巧的数据结构，它通过指针将各个分块“链接”起来，从而使得插入或者删除数据只需 $O(1)$ 的时间，但是在寻找某个元素时则需要 $O(n)$ 的时间。在块状链表中，某个块可以存储一段数据，存储这段数据的数据结构需要支持动态插入和删除，一般采用向量或者集合来进行存储。

2.15.1 块状链表的操作

首先定义块状链表中保存信息的数据结构。为了示例的方便，假设分块中存储的是字符。

```

//++++++++++++++++++2.15.1.cpp++++++++++++++++++/
// 分块的大小。
const int BLOCK_SIZE = 1024;

// 存储信息的分块结构体。
typedef struct _BLOCK {
    // size 记录该块的大小，next 记录分块所指向的下一个分块的编号。
    int size, next;

```

```

// s 记录分块所存储的字符。
char s[BLOCK_SIZE << 1];
} block;

// 定义链表所能使用的全部分块。
block blk[BLOCK_SIZE << 2];

为了操作的便利，定义若干辅助操作的功能。

// cache 为分块缓存池，用于分配和回收分块，cp 为缓存池的指针，指向第一个可用的分块。
int cache[BLOCK_SIZE << 2], cp = 0;

// 定位序号为 idx 的字符所在的分块编号。
int locate(int &idx) {
    int now = 0;
    while (idx > blk[now].size && blk[now].next != -1) {
        idx -= blk[now].size;
        now = blk[now].next;
    }
    return now;
}

// 从缓存池中分配一个分块。
int allocate() { return cache[cp++]; }
// 回收编号为 x 的分块。
void recycle(int x) { cache[--cp] = x; }

// 初始化。
void initialize() {
    // 将所有可用分块放入缓存池。
    for (int i = 0; i < (BLOCK_SIZE << 2); i++) cache[i] = i;
    // 为了便于操作，将编号为 0 的分块分配给链表的起始分块。
    blk[0].size = 0, blk[0].next = -1;
    // 将缓存池可用分块指针向后移动一个位置。
    cp++;
}

// 附加。
// 在编号为 x 的分块后面附加一个编号为 y 的分块并将字符数组 s 中的前 n 个字符复制到分块 y 中。
void append(int x, int y, char s[], int n) {
    if (y != -1) {
        blk[y].next = blk[x].next, blk[y].size = n;
        memcpy(blk[y].s, s, n);
    }
    blk[x].next = y;
}

// 合并。
// 合并编号为 x 和编号为 y 的两个分块，若合并成功则回收分块 y。
void merge(int x, int y) {
    if (x == -1 || y == -1) return;
    if (blk[x].size + blk[y].size >= BLOCK_SIZE) return;
    memcpy(blk[x].s + blk[x].size, blk[y].s, blk[y].size);
    blk[x].size += blk[y].size, blk[x].next = blk[y].next;
    recycle(y);
}

```

```

}

// 分割。
// 将编号为 x 的分块从该分块所存储的序号为 idx 的字符处分割为两个分块。
void split(int x, int idx) {
    if (x == -1 || idx == blk[x].size) return;
    append(x, allocate(), blk[x].s + idx, blk[x].size - idx);
    blk[x].size = idx;
}

```

插入

在序号为 idx 的字符处，插入一个长度为 n 的字符串 s 。

```

void insert(int idx, char s[], int n) {
    int now = locate(idx);
    if (idx > blk[now].size && blk[now].next == -1) return;
    split(now, idx);
    int inserted = 0, start = now;
    while (inserted + BLOCK_SIZE < n) {
        append(now, allocate(), s + inserted, BLOCK_SIZE);
        inserted += BLOCK_SIZE;
        now = blk[now].next;
    }
    if (inserted < n) append(now, allocate(), s + inserted, n - inserted);
    merge(now, blk[now].next);
    merge(start, blk[start].next);
}

```

删除

从序号为 idx 的字符开始，删除 n 个字符。

```

bool erase(int idx, int n)
{
    int now = locate(idx);
    if (idx > blk[now].size && blk[now].next == -1) return;
    split(now, idx);
    int end = blk[now].next;
    while (end != -1 && n > blk[end].size) {
        n -= blk[end].size;
        end = blk[end].next;
    }
    split(end, n);
    end = blk[end].next;
    for (int i = blk[now].next; i != end; i = blk[now].next) {
        blk[now].next = blk[i].next;
        recycle(i);
    }
    merge(now, end);
}

```

获取

从序号为 idx 的字符开始，获取 n 个字符并将其复制到指定的字符数组 s 中。

```

void get(int idx, char s[], int n) {
    int now = locate(idx);

```

```

if (idx > blk[now].size && blk[now].next == -1) {
    s[0] = '\0';
    return;
}
int copied = min(n, blk[now].size - idx);
memcpy(s, blk[now].s + idx, copied);
now = blk[now].next;
while (now != -1 && n > copied + blk[now].size) {
    memcpy(s + copied, blk[now].s, blk[now].size);
    copied += blk[now].size;
    now = blk[now].next;
}
if (copied < n && now != -1) memcpy(s + copied, blk[now].s, n - copied);
s[n] = '\0';
}
//+++++++++++++++++2.15.1.cpp+++++++++++++++++

```

2.15.2 块状链表的应用

使用数组实现的分块数据结构，适用于数据数量固定不发生变化的情形，如果数据的数量会发生动态改变，使用块状链表就较为合适。以下通过一道例题的分析来示例块状链表的应用^I。

NOI 2003 文本编辑器

文本编辑器由一段文本和该文本中的一个光标组成。文本是由 0 个或多个 ASCII 码值在闭区间[32, 126]内的字符构成的序列。光标是在一段文本中用于指示位置的标记，可以位于文本首部，文本尾部或文本的某两个字符之间。如果文本编辑器所包含的文本为空，我们就说这个文本编辑器是空。文本编辑器支持以下六种操作。

操作	输入格式	功能
移动	Move <i>k</i>	将光标移动到第 <i>k</i> 个字符之后，如果 <i>k</i> =0，将光标移动到文本起始处
插入	Insert <i>n</i> <i>s</i>	在光标处插入长度为 <i>n</i> 的字符串 <i>s</i> ，光标位置不变， <i>n</i> ≥1
删除	Delete <i>n</i>	删除光标后的 <i>n</i> 个字符，光标位置不变， <i>n</i> ≥1
输出	Get <i>n</i>	输出光标后的 <i>n</i> 个字符，光标位置不变， <i>n</i> ≥1
后退	Prev	光标前移一个字符
前进	Next	光标后移一个字符

你的任务是：(1) 建立一个空的文本编辑器。(2) 从输入文件中读入一些操作并执行。(3) 对所有执行过的 **Get** 操作，将指定的内容输出。

输入

^I 此题目对应洛谷主题库中编号为 P4008 的题目。

输入的第一行是一个整数 t , 表示需要执行的指令数目。接下来是 t 个操作。为了使输入文件便于阅读, **Insert** 操作的字符串中可能会插入一些回车符, 请忽略掉它们(如果难以理解这句话, 可以参照样例输入)。除了回车符之外, 输入文件的所有字符的 ASCII 码值都在闭区间[32, 126]内。且行尾没有空格。

输出

对于每条 **Get** 指令, 输出一行, 对应该条指令的输出。

样例输入

```
15
Insert 26
abcdefghijklmnopqrstuvwxyz
qrstuvwxyz wxy
Move 15
Delete 11
Move 5
Insert 1
^
Next
Insert 1

-
Next
Next
Insert 4
.\/.
Get 4
Prev
Insert 1
^
Move 0
Get 22
```

样例输出

```
.\/.
abcdefghijklmnopqrstuvwxyz^f.\/.qrstuvwxyz
```

分析

本题需要动态插入和删除一段字符, 使用普通的数组予以处理效率较低, 而使用块状链表可以将操作集中在局部范围以提高效率。由于是字符相关的处理, 可以使用字符串类实例来存储数据。处理的方式和之前介绍的块状链表数组实现类似。

```
//-----2.15.2.cpp-----//
const int BLOCK_SIZE = 2000;

typedef struct _BLOCK {
    string s;
    _BLOCK *next;
} block;

block *head = new block; // 定义链表的头。
int idx; // 用于获取光标位置所在的分块
int cursor = 0; // 记录文本编辑器光标所在位置

// 获取当前光标位置所处的分块。
block* locate() {
    idx = cursor;
    block *now = head;
```

```

while (idx > now->s.size() && now->next != nullptr) {
    idx -= now->s.size();
    now = now->next;
}
return now;
}

// 创建。
// 根据给定的字符串创建一个块状链表以便将其插入原有的块状链表中。
pair<block*, block*> build(string s) {
    int cnt = 0, n = s.size();
    block *first = nullptr, *last = nullptr;
    while (cnt < n) {
        block *tmp = new block;
        tmp->next = nullptr;
        tmp->s.insert(
            tmp->s.end(), s.begin() + cnt, s.begin() + min(n, cnt + BLOCK_SIZE));
        if (first == nullptr) first = tmp;
        else last->next = tmp;
        last = tmp;
        cnt += BLOCK_SIZE;
    }
    return make_pair(first, last);
}

// 合并。
bool merge(block *b1, block *b2) {
    if (b1 == b2) return false;
    if (b1 == nullptr || b2 == nullptr) return false;
    if (b1->s.size() + b2->s.size() >= 2 * BLOCK_SIZE) return false;
    b1->s.insert(b1->s.end(), b2->s.begin(), b2->s.end());
    b1->next = b2->next;
    delete b2;
    return true;
}

// 分割。
pair<block*, block*> split(block *now, int idx) {
    block *tmp = new block;
    tmp->next = nullptr;
    tmp->s.insert(tmp->s.end(), now->s.begin() + idx, now->s.end());
    now->s.erase(now->s.begin() + idx, now->s.end());
    tmp->next = now->next;
    now->next = tmp;
    return make_pair(now, tmp);
}

// 插入。
void insert() {
    int n;
    cin >> n;
    // 读入 Insert 指令所插入的字符串。
    string s, line;
    cin.ignore(256, '\n');
    while (n > 0) {
        getline(cin, line);

```

```

        while (line.back() < 32 || line.back() > 126) line.pop_back();
        s += line;
        n -= line.length();
    }
    // 如果插入的字符串较小, 直接将其插入到某个分块中。
    block *now = locate();
    if (now->s.size() + s.size() < 2 * BLOCK_SIZE)
        now->s.insert(now->s.begin() + idx, s.begin(), s.end());
    // 如果插入的字符串较大, 则将其分割为若干分块后再予以插入。
    else {
        pair<block*, block*> p1 = split(now, idx);
        pair<block*, block*> p2 = build(s);
        p1.first->next = p2.first;
        p2.second->next = p1.second;
        // 尝试将插入的分块与之前和之后的分块融合以保证复杂度。
        if (p2.first == p2.second) {
            if (!merge(p1.first, p2.first))
                merge(p2.first, p1.second);
        } else {
            merge(p1.first, p2.first);
            merge(p2.second, p1.second);
        }
    }
}

// 删除。
void erase() {
    int n;
    cin >> n;
    block *now = locate();
    split(now, idx);
    block *end = now->next;
    while (end != nullptr && n > end->s.size()) {
        n -= end->s.size();
        end = end->next;
    }
    split(end, n);
    end = end->next;
    for (block *i = now->next; i != end; i = now->next) {
        now->next = i->next;
        delete i;
    }
    merge(now, end);
}

// 输出。
void get() {
    int n;
    cin >> n;
    block *now = locate();
    while (n) {
        if (idx + n < now->s.size()) {
            cout << now->s.substr(idx, n);
            n = 0;
        } else {
            if (idx) {

```

```

        cout << now->s.substr(idx);
        n -= now->s.size() - idx;
        idx = 0;
    } else {
        cout << now->s;
        n -= now->s.size();
    }
}
now = now->next;
}
cout << '\n';
}

// 释放内存。
void destroy(block *now) {
    if (now != nullptr) {
        if (now->next != nullptr) destroy(now->next);
        delete now;
    }
}

int main(int argc, char *argv[]) {
    int t;
    cin >> t;
    string cmd;
    for (int i = 1; i <= t; i++) {
        cin >> cmd;
        if (cmd == "Insert") insert();
        else if (cmd == "Delete") erase();
        else if (cmd == "Get") get();
        else if (cmd == "Move") cin >> cursor;
        else if (cmd == "Prev") cursor--;
        else if (cmd == "Next") cursor++;
    }
    destroy(head);
    return 0;
}
//-----2.15.2.1.cpp-----//

```

在 STL 的 SGI 实现中，提供了一个容器类 `rope`，它实现的是类似于块状链表的功能，当题目时间限制不是很紧时，可以使用 `rope` 来作为块状链表的替代。`rope` 容器类提供了以下的操作^[23]：

操作	功能
<code>insert(i, x)</code>	在序号为 i 的元素前插入元素（或元素序列） x
<code>erase(i, n)</code>	移除从序号 i 开始的 n 个元素
<code>substr(i, n)</code>	获取从序号 i 开始的 n 元素
<code>replace(i, x)</code>	从序号为 i 的元素开始，将后续的元素替换为元素（或元素序列） x
<code>push_back(x)</code>	在容器末尾添加元素 x

<code>at(i)/[i]</code>	访问序号为 i 的元素
------------------------	---------------

注意，使用 `rope` 需要包含扩展库`<ext/rope>`并增加命名空间`__gnu_cxx`。

```
-----2.15.2.2.cpp-----//
#include <bits/stdc++.h>
#include <ext/rope>

// 使用 rope 容器类需要引用命名空间 __gnu_cxx。
using namespace std;
using namespace __gnu_cxx;

// 声明 rope 容器类的一个实例，容器中存储的是字符。
// 因为头文件定义：
// typedef rope<char> __gnu_cxx::crope;
// 因此可以使用另外一种声明方式：crope r;
rope<char> r;

// 记录光标的位置。
int cursor = 0;

// 插入字符。
void insert() {
    int n;
    cin >> n;
    // 读入需要插入的字符。
    string s, line;
    cin.ignore(256, '\n');
    while (n > 0) {
        getline(cin, line);
        while (line.back() < 32 || line.back() > 126) line.pop_back();
        s += line;
        n -= line.length();
    }
    // 需要将字符串类实例转换为 C 风格的字符串才能插入。
    r.insert(cursor, s.c_str());
}

int main(int argc, char *argv[]) {
    int t, n;
    cin >> t;
    string cmd;
    for (int i = 1; i <= t; i++) {
        cin >> cmd;
        if (cmd == "Insert") insert();
        else if (cmd == "Delete") { cin >> n; r.erase(cursor, n); }
        else if (cmd == "Get") { cin >> n; cout << r.substr(cursor, n) << '\n'; }
        else if (cmd == "Move") cin >> cursor;
        else if (cmd == "Prev") cursor--;
        else if (cmd == "Next") cursor++;
    }
    return 0;
}
-----2.15.2.2.cpp-----//
```

强化练习：。

2.16 莫队算法

2.16.1 莫队算法的原理

2.16.2 莫队算法的应用

2.17 并查集

并查集（union-find set）是一种表示不相交集合（disjoint set）的数据结构，用于处理不相交集合的合并与查询问题。在不相交集合中，每个集合通过代表（represent）来区分，代表是集合中的某个成员，能够起到唯一标识该集合的作用。一般来说，选择哪一个元素作为代表是无关紧要的，关键是在进行查找操作时，得到的答案是一致的。

在不相交集合上，需要经常进行如下操作：

find(x)：查找元素 x 属于哪个集合，如果 x 属于某一集合，则返回该集合的代表。

union(x, y)：如果元素 x 和元素 y 分别属于不同的集合，则将两个集合合并，否则不做操作。

一种简单的实现方法是使用链表来表示并查集。每个集合用一个链表来表示，链表的第一个元素作为它所在集合的代表。链表中的元素具有 *head* 指针和 *tail* 指针，*head* 指向链表的第一个元素，即代表元素，*tail* 指向链表中最后一个元素。那么 *find* 操作所需时间复杂度为 $O(1)$ 。合并操作可以采用每次将 x 所在链表拼接到 y 所在链表末尾的方法来完成，也可以在每次合并时将长度较短的链表合并到长度较长的链表末尾，这样的话可以减少合并时的操作次数，提高效率，这种合并策略被称为加权合并启发式策略（weighted-union heuristic）。

更为高效地实现并查集的方法是使用有根树来表示集合——树中的每个结点都表示集合的一个元素，每棵树表示一个集合。在此基础上，可以应用以下两种改进运行时间的启发式策略以得到表示不相交集合的更快实现。

路径压缩

在查找过程中，如果找到了祖先结点 p ，则将查找路径上的所有子孙结点的根结点均设置为该祖先结点 p ，这样在后续查找时能够节省时间。否则有可能导致树的深度过大，降低查询速度。

按秩合并

在合并过程中，将元素少的集合合并到元素多的集合中，这样合并之后树的高度将会减少，从而提高查询速度。具体实现时，为每个集合设置一个“秩”，合并时如果两个集合的“秩”相同，则任意选择一个集合将其“秩”值增加一，作为另一集合的祖先。

```
//-----2.17.cpp-----//
const int MAXV = 1000000;

int parent[MAXV], ranks[MAXV];

// 不带参数的初始化版本。
void makeSet() {
```

```

    for (int i = 0; i < MAXV; i++) parent[i] = i, ranks[i] = 0;
}

// 带参数的初始化版本，指定了元素的个数，可以避免每次都对整个数组进行初始化，节省时间。
void makeSet(int n) {
    for (int i = 0; i < n; i++) parent[i] = i, ranks[i] = 0;
}

// 带路径压缩的查找，使用递归实现。
int findSet(int x) {
    return (x == parent[x] ? x : parent[x] = findSet(parent[x]));
}

// 带路径压缩的查找，使用迭代实现。
int findSet(int x) {
    // 迭代寻找根。
    int ancestor = x;
    while (ancestor != parent[ancestor]) ancestor = parent[ancestor];
    // 路径压缩。
    while (x != ancestor) {
        int temp = parent[x];
        parent[x] = ancestor;
        x = temp;
    }
    return x;
}

// 集合按秩合并。
bool unionSet(int x, int y) {
    x = findSet(x), y = findSet(y);
    if (x != y) {
        if (ranks[x] > ranks[y]) parent[y] = x;
        else {
            parent[x] = y,
            if (ranks[x] == ranks[y]) ranks[y]++;
        }
        return true;
    }
    return false;
}
//-----2.17.cpp-----//

```

强化练习: [459 Graph Connectivity^A](#), [599 The Forrest for the Trees^B](#), [793 Network Connections^A](#), [1197 The Suspects^B](#), [1329 Corporative Network^C](#), [11690 Money Matters^B](#)。

扩展练习: [10158 War^B](#)。

在合并过程中，每成功合并一次，不同的集合数量就减少一，利用这个特点可以容易地求得最终的不同集合数量。另外，如果启用按秩合并，将秩小的集合总是合并到秩大的集合中，只要在初始化及合并时对“秩”的更新做适当调整，就可以使得“秩”能够表示以某个元素为代表的集合中元素的总个数，这在统计具有最大元素个数的集合时非常有用。

```

// 集合的初始化。
void makeSet(int n) {
    // 初始时，每个集合中有一个元素，秩的大小即为以此元素为代表的集合中的元素个数。

```

```

    for (int i = 0; i <= n; i++) parent[i] = i, ranks[i] = 1;
}

// 集合按秩合并，秩表示以某元素为代表的集合中的元素个数。
bool unionSet(int x, int y) {
    x = findSet(x), y = findSet(y);
    if (x != y) {
        // 更改合并过程中秩改变的规则，使得秩表示集合中元素的个数的性质不变。
        if (ranks[x] > ranks[y]) parent[y] = x, ranks[x] += ranks[y];
        else parent[x] = y, ranks[y] += ranks[x];
        return true;
    }
    return false;
}

```

强化练习：10583 Ubiquitous Religions^A, 10608 Friends^A, 10685 Nature^A, 11503 Virtual Friends^A。

扩展练习：1160 X-Plosives^C, 11987 Almost Union-Find^C。

2.18 二叉排序树

二叉排序树 (binary sort tree, 又称二叉搜索树, binary search tree, BST), 是一棵空树, 或者是具有以下性质的二叉树:

- (1) 若左子树不为空, 则左子树上的所有结点的键值均小于它的根结点的键值;
- (2) 若右子树不为空, 则右子树上的所有结点的键值均大于它的根结点的键值;
- (3) 左子树、右子树均为二叉排序树。

一般来说, 二叉搜索树中不存在键值相同的结点。在实践中, 常见的方式是使用一个结构来表示二叉排序树中的结点。

```

//+++++++++++++++++2.18.cpp+++++++++++++++++
struct TreeNode {
    int key;
    TreeNode *leftChild, *rightChild;
};

```

2.18.1 二叉排序树的操作

二叉搜索树的常见操作包括: 查找、插入、删除、遍历。由于二叉搜索树本身是一棵树, 其遍历操作和普通的二叉树并无二致, 故不再赘述, 主要介绍前面三种操作。

查找

查找操作的逻辑:

- (1) 从二叉搜索树的根结点开始搜索, 如果当前结点为空的二叉搜索树, 表明欲查询的键值不存在, 返回一个空指针;
- (2) 将当前结点的键值与待查询键值相比较, 如果待查询键值等于当前结点的键值, 则表明查询成功, 返回该结点即可; 如果待查询键值小于当前结点的键值, 则在该结点的左子树中继续查询, 否则在该结点的右子树中继续查询;
- (3) 重复上述过程, 直到查询结束。

```

// 查找
TreeNode* find(TreeNode* node, int key) {

```

```

    if (node == nullptr) return nullptr;
    if (node->key == key) return node;
    if (node->key > key) return find(node->leftChild, key);
    else return find(node->rightChild, key);
}

```

插入

插入操作首选需要确定在二叉搜索树中插入的位置，其逻辑为：

(1) 从二叉搜索树的根结点开始搜索，当插入过程到达了一个空的二叉搜索树，则表明找到了待插入的位置，新建一个结点，将新建的结点挂接在原二叉搜索树上即可。

(2) 比较当前结点的键值和待插入结点的键值，如果当前结点的键值大于待插入结点的键值，则在当前结点的左子树中继续执行插入操作；如果当前结点的键值小于待插入结点的键值，则在当前结点的右子树中继续执行插入操作；如果当前结点的键值等于待插入结点的键值，因为结点已经存在，直接返回该结点即可，无须再重新创建结点。

(3) 重复上述过程，直到插入过程结束。

```

// 插入
void insert(TreeNode* &node, int key) {
    if (node == nullptr) {
        node = new TreeNode;
        node->key = key;
        node->leftChild = node->rightChild = nullptr;
    }
    if (node->key == key) return;
    if (node->key > key) insert(node->leftChild, key);
    else insert(node->rightChild, key);
}

```

删除

删除操作相对于查找和插入来说复杂一些，需要分几种情况分别予以处理。首先，根据二叉搜索树的特点，从根结点开始查找，直到找到欲删除的结点 s ，如果该结点为叶结点，则直接删除即可；如果该结点只有一棵左子树或者只有一棵右子树，则直接将子树的根结点将欲删除的结点替换即可；如果欲删除的结点具有左右两棵子树，根据二叉搜索树的特性，则需要从左子树中找到键值的最大的结点（或者从右子树中找到键值最小的结点） d ，将欲删除的结点 s 替换掉，然后再将 d 从原二叉搜索树中删除，即可达到删除指定键值结点的目的。

```

// 删除
TreeNode* remove(TreeNode* &node, int key) {
    if (node == nullptr) return node;
    if (key < node->key) return remove(node->leftChild, key);
    if (key > node->key) return remove(node->rightChild, key);
    if (node->leftChild == nullptr && node->rightChild == nullptr) {
        node = nullptr;
        return node;
    }
    if (node->leftChild != nullptr && node->rightChild == nullptr) {
        node = node->leftChild;
        return node;
    }
    if (node->leftChild == nullptr && node->rightChild != nullptr) {

```

```

        node = node->rightChild;
        return node;
    }
    // 查找左子树中的最大值
    TreeNode* tmp = node->leftChild;
    int maxInLeftTree = 0;
    while (tmp != nullptr) {
        maxInLeftTree = tmp->key;
        tmp = tmp->rightChild;
    }
    node->key = maxInLeftTree;
    node->leftChild = remove(node->leftChild, maxInLeftTree);
    return node;
}
//++++++2.18.cpp+++++

```

2.18.2 二叉排序树的应用

2.19 二叉堆

堆 (heap) 是一类数据结构, 它们具有树状结构, 而且能够保证父结点比子结点大 (或小)。当根结点保存堆中最大值时, 称为大根堆; 反之, 则称为小根堆。

二叉堆 (binary heap) 是最简单、常用的堆, 它是一棵符合堆性质的完全二叉树, 可以在 $O(\log n)$ 的时间复杂度内完成插入或删除某个值的操作, 并且以 $O(1)$ 的时间复杂度查询最大 (或最小) 值。由于二叉堆是一棵完全二叉树, 我们可以考虑使用一维数组来存储二叉树的结点: 若结点对应的数组元素序号为 p , 则序号为 $2p$ 的数组元素对应结点的左儿子, 序号为 $2p+1$ 的数组元素对应结点的右儿子。

2.19.1 二叉堆的操作

当在二叉堆中插入一个结点时, 可能会破坏二叉树的堆性质, 因此需要通过相应的操作来进行维护。下面以小根堆为例, 说明如何通过上浮和下沉操作来维护二叉树的堆性质。

当一个结点比它的父结点的键值要小时, 就需要将其与父结点进行交换, 直到其键值不小于父结点或者成为根结点为止, 这个过程就称之为“上浮”。

```

//++++++2.19.cpp+++++
const int MAXN = 1000010;
// cnt 记录二叉堆的大小, heap 记录二叉堆的结点。
int cnt, heap[MAXN];
// 上浮操作。
void swim(int n) {
    for (int i = n; i > 1 && heap[i] > heap[i >> 1]; i >>= 1)
        swap(heap[i], heap[i / 2]);
}

```

类似的, 当一个结点比它的具有较大键值的子结点的键值要大, 就需要将其与该子结点进行交换, 直到其键值不大于任何子结点或者成为叶结点为止, 这个过程就称之为“下沉”。

```

// 获取具有较大值的儿子结点的序号。
int child(int n) {
    return (n << 1) + ((n << 1) + 1 <= cnt && heap[(n << 1) + 1] > heap[(n << 1)]);
}

```

```
// 下沉操作。
void sink(int n) {
    for (int i = n, c = child(i); c <= cnt && heap[c] > heap[i]; i = c, c = child(i))
        swap(heap[i], heap[c]);
}
```

依靠上浮和下沉这两种基本操作，我们可以很容易完成堆的其他操作。当要在堆中插入值时，直接在堆的底部插入值，然后再上浮即可。

```
void push(int x) {  
    heap[cnt++] = x;  
    swim(cnt - 1);  
}
```

当需要删除堆顶元素时，可以将堆顶元素与堆的最后一个结点交换，接着将堆的大小减 1，然后再对堆顶元素进行下沉操作以维护堆的性质。

```
void pop() {  
    swap(heap[0], heap[--cnt]);  
    sink(0);  
}
```

如果需要查询堆顶元素，直接返回根结点即可。

```
int top() { return heap[0]; }
```

如果给定一个由 n 个整数构成的数组 A ，可以很容易将其转换为堆，只需将其复制到堆数据结构中，然后按照从堆的底部到顶部的顺序依次进行下沉操作即可。由于复制到堆数据结构中的一半数据属于叶结点，这些结点并不需要进行下沉操作，因此可以从数组的二分之一处开始遍历。

```
void build(int A[], int n) {
    memcpy(heap, a, sizeof(int) * n);
    for (int i = n >> 1; i >= 0; i--)
        sink(i);
}
```

根据堆的元素个数计数，可以确定堆的大小以及堆是否为空。

```
// 返回二叉堆的大小。  
int size() { return cnt; }  
// 返回二叉堆是否为空。  
bool empty() { return !cnt; }  
//++++++2.19.cpp++++++
```

2.19.2 二叉堆的应用

二叉堆最常见的应用是实现优先队列。在 STL 中包含有优先队列，在 STL 的 SGI 实现中，使用向量来存储数据，并使用堆操作来调整向量中存储的数据次序，使之构成优先队列。

2.20 树堆

二叉排序树在特殊情况下可能退化为一条链，其操作效率会退化为 $O(n)$ ，为了避免这种情况的出现，人们发明了一种近似平衡树的数据结构——树堆。

树堆 (treap) 是二叉排序树 (binary sort tree) 与堆 (heap) 相结合产生的一种具有堆性质的二叉排序树。

序树。类似于红黑树或 AVL 树，树堆是一种平衡化的二叉搜索树。树堆中结点的键值（key）在满足二叉排序树要求的前提下，增加了随机附加的优先级（priority）以满足维护堆序的要求。可以证明，如果优先级是随机的，那么树堆的期望深度是 $O(\log n)$ 的，因此树堆的查询、插入、删除操作可以在 $O(\log n)$ 的期望时间内完成。

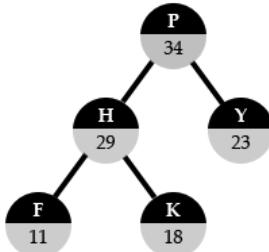


图 2-13 树堆（大根堆）。该树堆共包含 5 个结点，每个结点包含键值（结点上方的白色字母）和优先级（结点下方的黑色数字），容易看出，按照 ASCII 值的顺序，结点的键值符合二叉排序树的要求（任选一个结点 x ，结点 x 的左侧子树所包含的结点的键值均小于结点 x 的键值，结点 x 的右侧子树所包含的结点的键值均大于结点 x 的键值），结点的优先级则满足最大堆的要求（任选一个结点 x ，结点 x 的左右子树所包含的结点的优先级小于结点 x 的优先级）

树堆有两种常见的实现，一种是有旋转树堆，另外一种是无旋转树堆，先介绍有旋转树堆的操作实现，再介绍无旋转树堆的操作实现。

2.20.1 有旋转树堆的操作

旋转

在有旋转树堆中，旋转是一个基本操作，类似于左偏树，合并是左偏树的核心操作。由于插入或者删除结点后，二叉排序树可能已经不再符合堆的性质，因此需要通过特定的操作予以维护。以大根堆为例，当插入一个结点后，如果结点随机给定的优先级比其父结点的优先级要大，那么就需要将其和父结点交换以满足堆的性质。如图 2-14 和图 2-15 所示，旋转分为两种，如果不符合堆性质的结点是其父结点的左子树根结点，则进行右旋操作，否则进行左旋操作。

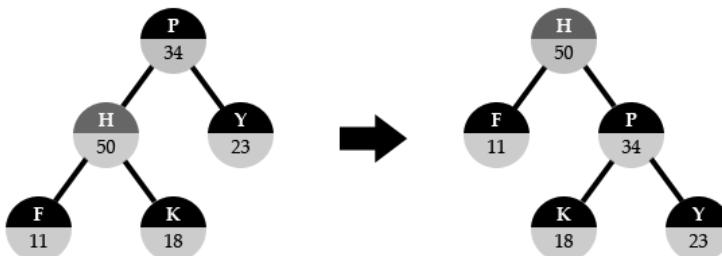


图 2-14 右旋操作。约定以结点的键值来表示结点，结点 H 的优先级为 50，大于其父结点 P 的优先级 34，不符合大根堆的性质，需要进行调整。将结点 H 作为根结点，结点 F 及其子树仍作为 H 的左子树，将结点 P 及其右子树作为 H 的右子树，然后将结点 K 及其子树作为 P 的左子树，即完成右旋操作。可以看到，完成右旋操作后，树堆符合二叉排序树及大根堆的性质。同时，旋转前后树堆的中序遍历顺序不变：F-H-K-P-Y

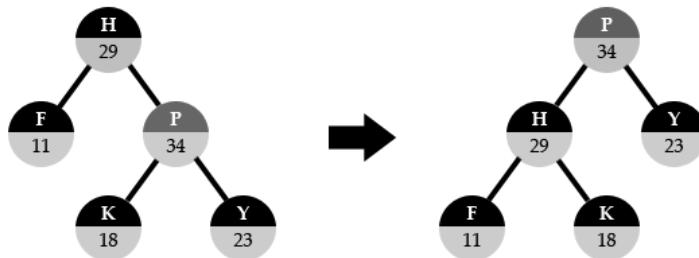


图 2-15 左旋操作。约定以结点的键值来表示结点，结点 P 的优先级为 34，大于其父结点 H 的优先级 29，不符合大根堆的性质，需要进行调整。将结点 P 作为根结点，结点 Y 及其子树仍作为 P 的右子树，将结点 H 及其左子树作为 P 的左子树，然后将结点 K 及其子树作为 H 的右子树，即完成左旋操作。可以看到，完成左旋操作后，树堆符合二叉排序树及大根堆的性质。同时，旋转前后树堆的中序遍历顺序不变：F-H-K-P-Y

根据旋转的定义，可以容易地写出其实现代码^I。

```
//+++++2.20.1.cpp+++++  
typedef struct Node {  
    // key 表示结点的键值，priority 表示结点的优先级。  
    int key, priority;  
    // cnt 表示具有与当前结点相同键值的结点数量，即允许键值存在重复。  
    // size 表示以该结点为根的子树中结点的数量。  
    int cnt, size;  
    Node *leftChild, *rightChild;  
    Node (int k) {
```

^I 将树堆各种操作的参考实现与以下主程序合并，即可通过洛谷主题库中编号为 P3369 的题目：

```
int main(int argc, char *argv[]) {  
    Tree root = new Node(-INF);  
    int n, cmd, x;  
    cin >> n;  
    for (int i = 1; i <= n; i++) {  
        cin >> cmd >> x;  
        if (cmd == 1) insert(root, x);  
        if (cmd == 2) remove(root, x);  
        if (cmd == 3) cout << getRank(root, x) - 1 << '\n';  
        if (cmd == 4) cout << getKth(root, x + 1) << '\n';  
        if (cmd == 5) cout << getPrevious(root, x) << '\n';  
        if (cmd == 6) cout << getNext(root, x) << '\n';  
    }  
    return 0;  
}
```

```

        key = k; size = cnt = 1; priority = rand();
        leftChild = rightChild = nullptr;
    }
} * Tree;

// 更新结点的额外信息。
void pushUp(Tree t) {
    if (t == nullptr) return;
    t->size = t->cnt;
    if (t->leftChild != nullptr) t->size += t->leftChild->size;
    if (t->rightChild != nullptr) t->size += t->rightChild->size;
}

// 右旋操作。
// 为了便于实现，传入的参数 t 是不符合堆性质的结点 lt 的父结点。
void rightRotate(Tree &t) {
    // 结点 lt 不符合堆性质，且是其父结点 t 的左子树根结点。
    Tree lt = t->leftChild;
    t->leftChild = lt->rightChild;
    lt->rightChild = t;
    t = lt;
    pushUp(t->rightChild);
    pushUp(t);
}

// 左旋操作。
// 为了便于实现，传入的参数 t 是不符合堆性质的结点 rt 的父结点。
void leftRotate(Tree &t) {
    // 结点 rt 不符合堆性质，且是其父结点 t 的右子树根结点。
    Tree rt = t->rightChild;
    t->rightChild = rt->leftChild;
    rt->leftChild = t;
    t = rt;
    pushUp(t->leftChild);
    pushUp(t);
}

```

插入

在树堆插入结点时，先按照键值将结点插入到二叉树的对应位置，一般为了简便，会按照二叉排序树的性质选择将其作为一个叶结点插入二叉树，然后随机给该叶结点设定一个权值作为优先级，接着维护堆的性质，即每次比较该结点与其父结点，如果不满足堆的要求就进行旋转操作。由于旋转的时间复杂度为 $O(1)$ ，最多进行 h 次旋转 (h 为树堆的高度)，因此插入操作的时间复杂度为 $O(h)$ ，而树堆的期望高度 h 为 $O(\log n)$ 级别，因此插入操作的期望时间复杂度为 $O(\log n)$ 。

```

void insert(Tree &t, int k) {
    // 将新的结点作为叶结点插入树堆。
    if (t == nullptr) {
        t = new Node(k);
    } else if (k == t->key) {
        t->cnt++;
        pushUp(t);
    } else if (k < t->key) {
        insert(t->leftChild, k);
    } else {
        insert(t->rightChild, k);
    }
}

```

```

        // 当左子结点不满足堆性质时进行右旋。
        if (t->priority < t->leftChild->priority) rightRotate(t);
    } else {
        insert(t->rightChild, k);
        // 当右子结点不满足堆性质时进行左旋。
        if (t->priority < t->rightChild->priority) leftRotate(t);
    }
    pushUp(t);
}

```

删除

如果需要删除的结点是叶结点，直接删除即可。若不是叶结点，则通过旋转操作，将待删除结点转变为叶结点进行删除。

```

void remove(Tree &t, int k) {
    if (t == nullptr) return;
    if (k == t->key) {
        // 结点副本数大于 1，将副本数减少 1 即可。
        if (t->cnt > 1) t->cnt--;
        // 欲删除的结点不是叶结点，则通过旋转将结点转变为叶结点。
        else if (t->leftChild != nullptr || t->rightChild != nullptr) {
            if (t->rightChild == nullptr ||
                (t->leftChild != nullptr &&
                t->leftChild->priority > t->rightChild->priority)) {
                rightRotate(t);
                remove(t->rightChild, k);
            } else {
                leftRotate(t);
                remove(t->leftChild, k);
            }
        } else {
            delete t;
            t = nullptr;
        }
    } else if (k < t->key) remove(t->leftChild, k);
    else remove(t->rightChild, k);
    pushUp(t);
}

```

查找

树堆的查找有多种形式，其操作与二叉排序树的相应操作类似。此处介绍三种常用的查询：查询树堆中某个键值的排名、查询树堆中第 k 大的键值、查询给定值 x 在树堆的键值中的前驱（定义 x 在树堆中的前驱为小于 x 的键值中的最大值）或后继（定义 x 在树堆中的后继为大于 x 的键值中的最小值）。

```

const int INF = 0x7f7f7f7f;

// 查询树堆中某个键值的排名（从 1 开始计数）。
int getRank(Tree t, int x) {
    // 在树堆中未找到该键值时返回一个标记值。
    if (t == nullptr) return -INF + 1;
    int s = t->leftChild == nullptr ? 0 : t->leftChild->size;
    // 找到该键值，返回从 1 开始计数的排名。
    if (x == t->key) return s + 1;
}

```

```

if (x < t->key) return getRank(t->leftChild, x);
return s + t->cnt + getRank(t->rightChild, x);
}

// 查询树堆中第 k 大的键值 (从 1 开始计数)。
int getKth(Tree t, int k) {
    // 在树堆中未找到第 k 大的键值时返回一个标记值。
    if (t == nullptr) return INF;
    int s = t->leftChild == nullptr ? 0 : t->leftChild->size;
    if (k <= s) return getKth(t->leftChild, k);
    if (k <= s + t->cnt) return t->key;
    return getKth(t->rightChild, k - s - t->cnt);
}

// 查询给定值 x 在树堆中键值的前驱。
int getPrevious(Tree t, int x) {
    // 默认返回标记值。
    int r = INF;
    // 根据前驱的定义, 从根结点开始, 不断地在子树中搜索符合要求的键值。
    while (t != nullptr) {
        if (t->key < x) {
            r = t->key;
            t = t->rightChild;
        } else t = t->leftChild;
    }
    return r;
}

// 查询给定值 x 在树堆中键值的后继。
int getNext(Tree t, int x) {
    // 默认返回标记值。
    int r = INF;
    // 根据后继的定义, 从根结点开始, 不断地在子树中搜索符合要求的键值。
    while (t != nullptr) {
        if (t->key > x) {
            r = t->key;
            t = t->leftChild;
        } else t = t->rightChild;
    }
    return r;
}
//+++++++++++++++++2.20.1.cpp+++++++++++++++++

```

2.20.2 无旋转树堆的操作

无旋转树堆是树堆的另外一种实现, 相对于有旋转树堆, 其原生支持可持久化, 但是在理解难度上比有旋转树堆要大。在无旋转树堆的实现中, 堆性质的维护通过与平衡树类似的拆分、合并操作来进行。

拆分

拆分 (split) 是指将以 t 为根结点的树堆, 按照给定的值 k , 分割为两个树堆, 一个以 x 为根结点, 另一个以 y 为根节点, 以 x 为根结点的树堆中的键值均小于等于 k , 以 y 为根结点的树堆中的键值均大于 k 。

拆分的实现思路: 从原树堆根结点 t 开始, 将根结点的键值与给定的键值 k 进行比较, 若根结点的键值小于等于 k , 根据树堆的性质, t 的左子树中的键值均小于 k , 则 t 以及 t 的左子树属于树堆 x , 否则 t 及其

右子树属于树堆 y 。接着，以递归的形式继续将 t 的相应子树以 k 为界划分到树堆 x 和 y 中。

```
//+++++2.20.2.cpp+++++/
void split(Tree t, Tree &x, Tree &y, int k) {
    if (t == nullptr) {
        x = y = nullptr;
        return;
    }
    if (t->key <= k) {
        x = t;
        split(t->rightChild, x->rightChild, y, k);
    } else {
        y = t;
        split(t->leftChild, x, y->leftChild, k);
    }
    pushUp(t);
}
```

合并

合并 (merge) 操作是将给定的两个树堆合并为一个树堆。给定以结点 x 为根结点的树堆 X ，以及以结点 y 为根结点的树堆 Y ，满足树堆 X 中的任意键值均小于树堆 Y 中的任意键值（回顾拆分操作，分割得到的两个树堆 X 和 Y ，其中一个树堆 X 中任意结点的键值均小于另外一个树堆 Y 中任意结点的键值），那么在合并时只需考虑结点优先级的影响。以大根堆为例，若根结点 x 的优先级大于结点 y 的优先级，那么结点 y 及其子树应该作为结点 x 的右子树“挂接”到合并后的树堆 T 上，结点 x 的左子树“原封不动”地作为合并后树堆 T 的左子树，而结点 x 则作为合并后树堆 T 的根结点。反之，结点 x 及其子树应该作为结点 y 的左子树“挂接”到合并后的树堆 T 上， y 的右子树“原封不动”地作为合并后树堆 T 的右子树，而结点 y 则作为合并后树堆 T 的根结点。递归的继续上述过程，则最终可以将两个树堆 X 和 Y “参差”地合并为一个树堆 T 。

```
void merge(Tree &t, Tree x, Tree y) {
    if (x == nullptr || y == nullptr) {
        t = x == nullptr ? y : x;
        return;
    }
    if (x->priority > y->priority) {
        t = x;
        merge(t->rightChild, x->rightChild, y);
    } else {
        t = y;
        merge(t->leftChild, x, y->leftChild);
    }
    pushUp(t);
}
```

插入

插入操作的实现思路是将树堆分为三个子树堆：所有结点键值均小于待插入值 k 的树堆 X ，所有结点键值均等于待插入值 k 的树堆 Z ，所有结点键值均大于待插入值 k 的树堆 Y 。接着检查树堆 Z 是否为空，如果为空，表明键值为 k 的结点不存在，否则将键值为 k 的结点计数增加 1 即可。后续按照顺序将三个树堆合并为一个树堆即完成插入操作。

```

void insert(Tree &t, int k) {
    Tree x, y, z;
    split(t, x, y, k);
    split(x, x, z, k - 1);
    if (z == nullptr) z = new Node(k);
    else { z->cnt++; z->size = z->cnt; }
    merge(x, x, z);
    merge(t, x, y);
}

```

删除

删除操作的实现思路与插入操作的思路是类似的，只不过具体执行的是相反的操作：如果键值为 k 的结点计数为 1，则直接删除，否则将键值为 k 的结点计数减少 1 即可。

```

void remove(Tree &t, int k) {
    Tree x, y, z;
    split(t, x, y, k);
    split(x, x, z, k - 1);
    if (z->cnt > 1) { z->cnt--; z->size = z->cnt; }
    else {
        delete z;
        z = nullptr;
    }
    merge(x, x, z);
    merge(t, x, y);
}
//++++++2.20.2.cpp+++++++

```

查找

由于上述操作后所得到的二叉树仍然满足树堆的性质，因此查找的操作实现与有旋转树堆的查找实现相同，在此不再赘述。

2.20.2 可持久化树堆

在树堆的两种实现中，有旋转树堆由于旋转操作可能会更改结点的相对位置关系，因此不便于可持久化，而无旋转树堆，则由于拆分和合并操作均保留了结点的相对位置不变，因此可以方便地可持久化。回顾线段树的可持久化，通过在更改结点的同时，复制一条从叶结点到根结点的路径，以新的根结点作为新版本线段树的根结点。与此类似，树堆在拆分和合并过程中，亦可对涉及的结点复制一个副本，这些副本会构成一条路径，从而尽可能地复用了原树堆的数据，减少了可持久化所需的空间消耗。

在与可持久化树堆有关的应用中，为了保证拆分和合并后的数据符合前后顺序不变的要求，一般在拆分时是按照位置进行拆分而不是按键值进行拆分。也就是说，给定树堆 T 和数值 k ，拆分的目的是将树堆中的前 k 个结点划分为树堆 X ，将剩余的结点划分为树堆 Y 。另外，为了避免一个结点存储键值的多个副本可能带来的问题，一般是多个相同的键值使用不同的结点分别予以表示。

```

//++++++2.20.3.cpp+++++++
// 获取子树大小。
inline int getSize(Tree &t) {
    return t == nullptr ? 0 : t->size;
}

// 复制结点。

```

```

inline void copy(Tree &x, Tree &y) {
    if (y == nullptr) x = y;
    else {
        x = new Node(-1);
        *x = *y;
    }
}

// 拆分。
// 将以 t 为根结点的树堆，按照中序遍历的顺序，将前 k 个结点划分到以 x 为根结点的树堆 X 中，
// 剩余的结点划分到以 y 为根结点的树堆 Y 中。
void split(Tree t, Tree &x, Tree &y, int k) {
    // 需要划分到树堆 X 的结点数为 0，则剩余的结点全部划分到树堆 Y 中。
    if (!k) {
        copy(y, t);
        x = nullptr;
    }
    // 当树堆 t 的结点总数不大于 k 时，全部划分到树堆 X 中。
    else if (getSize(t) <= k) {
        copy(x, t);
        y = nullptr;
    }
    // 当树堆 t 的左子树大小不小于 k，则将 t 及其右子树划分到树堆 Y 中，
    // 继续在 t 的左子树中进行拆分。
    else if (getSize(t->leftChild) >= k) {
        copy(y, t);
        split(t->leftChild, x, y->leftChild, k);
        pushUp(y);
    }
    // 否则，将 t 及其左子树划分到树堆 X 中，继续在 t 的右子树中进行拆分。
    else {
        copy(x, t);
        split(t->rightChild, x->rightChild, y, k - getSize(t->leftChild) - 1);
        pushUp(x);
    }
}

```

强化练习：[12538 Version Controlled IDE^D](#)。

2.20.3 树堆的应用

由于树堆借用随机优先级的方式来动态调整树的高度，使得树堆支持在 $O(\log n)$ 的期望时间内完成插入一个元素、删除一个元素以及查找第 k 大元素的任务，因此可以近似认为是一棵平衡树。回顾树堆的两种实现，各有优缺点。有旋转树堆，因为旋转改变了结点的相对位置关系，难以有效地可持久化，编码较为复杂但实现理解。无旋转树堆，容易支持可持久化，编码较为简单但理解难度相对较大。

2.21 左偏树

左偏树 (leftist tree)，又称左偏堆 (leftist heap)，它是一棵二叉树，又具有堆的性质，而且支持堆的合并，属于可并堆 (mergeable heap)，是优先队列的一种实现形式，在各类编程竞赛中考察较多。

左偏树中的每个结点具有两个属性，一个是键值，另外一个是距离。键值就是结点所保存的数据，而距离是指该结点与外结点的最短距离。在树中，如果某个结点的左儿子或者右儿子为空，将空的左儿子或者右

儿子视为一棵空树，将这些空树定义为外结点（external node）。给定左偏树中的某个结点 x ，如果 x 是外结点，则将 x 的距离 $d(x)$ 定义为 0，如果 x 不是外结点，即 x 是内结点（internal node），则定义其距离 $d(x)$ 为 x 与其所在子树中的外结点的最短距离¹。根据二叉树的性质，一棵具有 n 个结点的左偏树，其根结点的距离不超过 $\lfloor \log n \rfloor + 1$ 。之所以将其称为左偏树，是因为该二叉树保持“左偏”的性质：每个结点的左儿子的距离都大于等于右儿子的距离。因此可以得出左偏树中每个结点的距离等于其右儿子距离加 1。

2.21.1 左偏树的操作

合并

合并是左偏树的核心操作。为了便于说明，下面以小根堆为例来说明左偏树的合并。在合并两棵左偏树时，既要保持堆的性质，又要保持左偏树的“左偏”性质。令合并的两棵左偏树为 T_1 和 T_2 ，合并后的左偏树为 T ，由于需要满足堆的性质，需要取两棵左偏树中较小的根结点作为合并后左偏树的根结点。如果 T_1 的根结点小于 T_2 的根结点，则取 T_1 的根结点作为 T 的根结点，将 T_1 的左子树作为 T 的左子树，然后递归地将 T_1 的右子树与 T_2 合并（若 T_2 的根结点小于 T_1 的根结点，合并过程相反）。为了满足“左偏”的性质，在合并之后，如果发现左儿子结点的距离大于右儿子结点的距离，则予以交换，最后将根结点的距离设置为右儿子结点的距离加 1。

```
//++++++2.21.1.cpp+++++++
const int MAXN = 1000010;

// 结点。
struct NODE {
    // lc 记录左儿子，rc 记录右儿子，key 记录键值，d 记录距离。
    int lc, rc, key, d;
} lt[MAXN];

// 左偏树的合并操作。
// 为了便于编码，当左偏树的根结点为 0 时表示左偏树为空。
int merge(int x, int y) {
    // 若一棵左偏树为空则返回另外一棵左偏树。
    if (!x || !y) return x | y;
    // 取两棵左偏树中具有较小键值的根结点作为合并后左偏树的根结点。
    if (lt[x].key > lt[y].key) swap(x, y);
    // 递归合并根结点的右子树与另外一棵左偏树，将其作为根结点的右子树。
    lt[x].rc = merge(lt[x].rc, y);
    // 若不满足“左偏”性质则交换左右儿子结点。
    if (lt[lt[x].rc].d > lt[lt[x].lc].d) swap(lt[x].lc, lt[x].rc);
    // 调整根结点的距离。
    lt[x].d = lt[lt[x].rc].d + 1;
    // 返回左偏树根结点的序号。
    return x;
}
```

由于“左偏”的性质，每递归一次，其中一个堆的根结点距离就会减小 1，而一棵有 n 个结点的二叉树，

¹ 对于外结点的定义，不同的文献可能定义不一致，但不影响左偏树的性质。有的文献按照如下方式定义外结点：如果某个结点的左儿子或者右儿子为空，则将该结点称之为外结点，此时外结点的距离定义为 1。

根的距离不超过 $O(\log n)$, 所以合并两个大小分别为 n 和 m 的左偏树的时间复杂度是 $O(\log n + \log m)$ 。

获取

获取堆顶元素的操作非常简单, 由于堆顶元素对应左偏树的根结点, 直接返回根结点的键值即可。

```
// 获取根结点为 x 的左偏树的堆顶值。
int top(int x) { return lt[x].key; }
```

删除

删除堆顶元素相当于将左偏树根结点的两棵子树合并。

```
// 删除根结点为 x 的左偏树的堆顶元素。
void pop(int x) { return merge(lt[x].lc, lt[x].rc); }
```

插入

向左偏树中插入一个结点相当于将原来的左偏树与只有一个结点的左偏树进行合并。

```
// root 用于记录新建的左偏树的根结点编号。
int root = 0;
// 向根结点为 x 的左偏树中插入一个键值为 y 的结点。
void push(int x, int y) {
    ++root;
    lt[root].key = y;
    lt[root].lc = lt[root].rc = lt[root].d = 1;
    return merge(x, root);
}
//-----2.21.1.cpp-----//
```

如果需要删除左偏树中的任意结点, 令待删除的结点为 x , 则先合并结点 x 的左右子树, 之后从 x 开始, 自底向上递归更新其祖先结点的距离, 当祖先结点的距离不符合“左偏”的性质时, 交换左右儿子, 一直到距离无需更新时结束递归过程。

生成

如果需要从一个数组开始生成一棵左偏树, 朴素的做法是逐个元素执行插入操作。采用下述方法可以提高效率: 设立一个队列, 先将数组中的元素两个为一组合并为一棵左偏树, 依次放入队列中, 接着从队列中依次取出前两棵左偏树予以合并, 然后将合并后的左偏树再次放入队列的末端, 重复此步骤, 直到所有左偏树合并为一棵左偏树。使用优化的左偏树生成方法, 对于长度为 n 的数组来说, 可以通过 $O(\log n)$ 次合并生成其对应的左偏树, 相较于朴素方法的 $O(n)$ 次合并, 效率有明显的提升。

2.21.2 可持久化左偏树

左偏树支持可持久化, 只需在合并左偏树的结点时新建结点来保存信息即可, 原有的结点不需改动就可以很方便地实现持久化。仍以小根堆为例, 以下是可持久化左偏树的合并操作参考实现。

```
//-----2.21.2.cpp-----//
int root = 0;
int merge(int x, int y) {
    if (!x || !y) return x | y;
    if (lt[x].key > lt[y].key) swap(x, y);
    int p = ++root;
    lt[p] = lt[x];
    lt[p].rc = merge(lt[x].rc, y);
}
```

```

if (lt[lt[p].rc].d > lt[lt[p].lc].d) swap(lt[p].lc, lt[p].rc);
lt[p].d = lt[lt[p].rc].d + 1;
return p;
}
//-----2.21.2.cpp-----//

```

2.21.3 左偏树的应用

X Cell (X 细胞)¹

X 细胞是来自于仙女座星系 Gamma 行星的一种古老生命形式。

X 细胞通过直接分裂来产生后代 X 细胞。对于某个 X 细胞 i 来说，如果它产生了一个直接后代 X 细胞 j ，则将细胞 i 称为细胞 j 的“母细胞”，将细胞 j 称为 i 的“子细胞”。注意，母细胞、子细胞的定义不具有传递性。假设细胞 i 产生了一个直接后代细胞 j ，细胞 j 又产生了一个直接后代细胞 k ，则将 j 称为 i 的子细胞， k 称为 j 的子细胞，但 k 不是 i 的子细胞。

每个 X 细胞具有活力值 h_x 和体积 v_x ，为了研究的方便，人们为 X 细胞定义了“变异指数”：

$$d_x = \frac{h_x}{v_x}$$

该指数用于衡量 X 细胞对环境的适应性，变异指数越低，细胞存活的概率越高。

人们发现，当 X 细胞受到特定的外界刺激后，它会激活并开始一种被人们称为“同化”的过程来转变为一个 Z 细胞。在同化过程开始前，激活的 X 细胞会改变自身状态成为一个 Y 细胞，Y 细胞会不断吸收它的子细胞并进行融合，使得该子细胞成为 Y 细胞的一部分。在融合后，Y 细胞的活力值和体积为融合前的细胞活力值和体积的加和。也就是说，假设有 n 个细胞经过融合成为一个 Y 细胞，这 n 个细胞的活力值和体积分别为 h_1, h_2, \dots, h_n 和 v_1, v_2, \dots, v_n ，则融合完成后，该 Y 细胞的活力值 $h_y = \sum_{i=1}^n h_i$ ，体积 $v_y = \sum_{i=1}^n v_i$ ，变异指数 $d_y = \frac{h_y}{v_y}$ 。

在同化过程中，Y 细胞会遵循以下原则：(1) 如果某个子细胞 j 的母细胞 i 尚未被同化，则该子细胞 j 不会被同化。(2) 能够使得 Y 细胞变异指数尽可能地小且同化尽可能多的细胞。当 Y 细胞无法再同化更多的细胞时，它会停止同化过程并转变为一个 Z 细胞并释放信息素（状态转变前后，细胞的活力值和体积不变）。该信息素会产生以下作用：令生成的 Z 细胞的变异指数为 d_z ，如果某个尚未被同化的子细胞 j 的母细胞 i 被该 Z 细胞同化，则该子细胞 j 的活力值 h_j 增加 $\lceil d_z \rceil$ ($\lceil x \rceil$ 表示不小于 x 的最小整数)。

研究人员需要通过一种专用设备来产生激活 X 细胞的特定外界刺激，每次使用该设备都会消耗一定数量的激活剂，消耗的激活剂数量 c_t 使用以下公式进行计算：

$$c_t = t \times \lceil d_z \rceil$$

其中 t 表示使用该设备的次数序号(初始时从 1 开始计数)， d_z 表示该次激活最终生成的 Z 细胞的变异指数。

由于母细胞会分泌信息素使得子细胞无法被激活，只能选择不存在母细胞或者母细胞已经被同化的 X 细胞作为特定外界刺激的对象，以使其激活并开始同化过程。给定所有 X 细胞之间的相互关系及其活力值和体积，鉴于激活剂非常难以制造，现在需要你制定一个最优的 X 细胞激活顺序方案，使得所有的 X 细胞均转变为 Z 细胞且消耗的激活剂数量最少。你只需要输出该最少值即可。

输入

¹ 这是本书作者拟制的题目，收录在洛谷主题库中，题号为 P8581 (<https://www.luogu.com.cn/problem/P8581>)。

输入的第一行是一个整数 n , 表示 X 细胞的数量。接下来的一行, 包含 $n-1$ 个整数, 依次表示编号为 $2 \sim n$ 的 X 细胞的母细胞的编号。接下来的一行, 包含 n 个整数, 依次表示编号为 i 的 X 细胞的活力值 h_i 。再接下来的一行, 包含 n 个整数, 依次表示编号为 i 的 X 细胞的体积 v_i , 初始的 X 细胞的编号为 1。

输出

输出一个整数, 表示使得所有 X 细胞均转变为 Z 细胞所需消耗的激活剂数量的最少值。

样例输入 1

```
3
1 2
5 7 12
1 1 10
```

样例输出 1

```
2
```

样例输入 2

```
3
1 1
2 2 12
2 3 4
```

样例输出 2

```
9
```

样例输入 3

```
5
1 2 2 1
1 7 10 20 9
1 1 1 1 1
```

样例输出 3

```
223
```

样例输入 4

```
12
1 2 3 1 5 6 7 1 9 10 11
4 10 2 1 50 1 20 9 40 2 15 10
1 1 1 1 1 1 1 1 1 1 1 1
```

样例输出 4

```
124
```

分析

可以将题目转换为以下等价形式:

有根树为一个有向图, 该有向图有一个特殊的顶点, 称之为根, 从根出发, 存在唯一的有向路径到达图中的任意其他顶点。按照习惯, 一般将有根树中的顶点称为结点。子树为有根树 T 的一个子图且该子图是一棵以 T 中某个结点为根的有根树。在有根树中, 如果有一条边从结点 i 出发, 到达结点 j , 则将结点 i 称为结点 j 的父结点, 将结点 j 称为结点 i 的子结点。将有根树中不存在子结点的结点称为叶结点。

给定有根树 T , 第 i 个结点具有权值 $a_i \in \mathbb{Z}^+$ 和 $b_i \in \mathbb{Z}^+$ 。令 T' 为 T 的一棵子树, F_i 为 T 中所有以结点 i 为根的子树的集合。定义:

$$(1) \quad r(T') = \frac{a(T')}{b(T')}, \text{ 其中 } a(T') = \sum_{j \in T'} a_j, \quad b(T') = \sum_{j \in T'} b_j.$$

(2) T_i 为一棵以结点 i 为根的子树, 令 $r_i = r(T_i)$, T_i 满足 $r_i = \min_{T' \in F_i} r_{T'}$ 且具有最多的结点数量。

(3) $S(T')$ 为 T' 的叶结点在 T 中的子结点构成的集合, 即对于 T 中的某个结点 j , 令其父结点为 i , 则 $j \in S(T')$ 当且仅当 $i \in T'$ 但 $j \notin T'$ 。

给定一棵具有 n 个结点的有根树 T , 令根结点为 i , 对其执行以下操作:

- (1) 将根结点 i 放入结点集合 Q , 即初始时置 $Q \leftarrow \{i\}$;
- (2) 任取 Q 中的一个元素, 令其为 j , 确定 T_j , 对于结点 $k \in S(T_j)$, 置 $a_k \leftarrow a_k + [r(T_j)]$;
- (3) 从集合 Q 中删除元素 j , 并置 $Q \leftarrow Q \cup S(T_j)$;
- (4) 若集合 $Q = \emptyset$, 结束操作, 否则转步骤 (2)。

每执行一次步骤 (2) 就会确定一棵 T 的子树, 假设在结束操作时一共执行了 m 次步骤 (2), 令第 i 次执行步骤 (2) 所确定的子树为 K_i , 最小化以下 W 值:

$$W = 1 \times [r(K_1)] + 2 \times [r(K_2)] + \cdots + m \times [r(K_m)] = \sum_{i=1}^m i \times [r(K_i)]$$

为了便于说明, 首先证明以下若干命题。

命题 1: 给定四个正整数 a, b, c, d , 若有 $\frac{a}{b} \leq \frac{c}{d}$ 则有 $\frac{a+c}{b+d} \leq \frac{c}{d}$ 。

证明: 由于 $\frac{a}{b} \leq \frac{c}{d}$, 则有 $ad - bc \leq 0$, 而 $\frac{a+c}{b+d} - \frac{c}{d} = \frac{(a+c)d - (b+d)c}{(b+d)d} = \frac{ad - bc}{(b+d)d} \leq 0$, 故 $\frac{a+c}{b+d} \leq \frac{c}{d}$ 。 ■

根据**命题 1**, 可以容易知道, 如果子结点的 r 值小于或者等于父结点的 r 值, 那么将子结点和父结点合并后父结点的 r 值不会变大。反之, 如果子结点的 r 值大于或者等于父结点的 r 值, 将子结点和父结点合并后父结点的 r 值不会变小。

命题 2: 若结点 $j \in S(T_i)$, 则 $r_j > r_i$ 。

证明: 假设 $r_j \leq r_i$, 根据**命题 1**, 将 T_i 与 T_j 合并后, r_i 不会变大且 T_i 具有更多的结点数量, 得出 T_i 应该包含 T_j , 因此 $j \in T_i$, 而给定 $j \in S(T_i)$, 得知 $j \notin T_i$, 产生矛盾, 故原假设不成立。 ■

命题 3: 若结点 $j \in T_i$, 则 T_i 包含 T_j , 即 $\forall p \in T_j, p \in T_i$ 。

证明: 假设 $\exists p \in T_j, p \notin T_i$, 根据子树的定义, T_i 包含的是 T_j 的一棵以结点 j 为根的子树 T'_j , 而对于任意 T_i 的子树 T'_j 来说, 均有 $r(T'_j) \geq r(T_j)$ (若不然, T_j 的 r_j 不是最优), 则包含整个 T_i 比包含 T'_j 不会使得 r_i 变大, 但是能够具有更多的结点, 因此原来的 T_i 不是最优的, 产生矛盾, 故假设不成立。 ■

命题 4: T_i 唯一。

证明: 假设 T_i 不唯一, 对于结点 i 来说, 存在两棵以结点 i 为根的子树 T_i 和 T'_i , 满足 $r(T_i) = r(T'_i) = \min_{T' \in F_i} T'$, 令 $A = T_i \cup T'_i$, 由于 T_i 和 T'_i 均是以结点 i 为根的子树, 故 $A \neq \emptyset$, 而由于 T_i 和 T'_i 不同但具有相同的结点数量, 肯定存在这样结点 p 和 q , 满足 p 是 q 的父结点, 且 $p \in A, q \in T'_i, q \notin T_i$ (若不存在, 则表明 T_i 包含 T'_i , 但由于 T_i 和 T'_i 不同, 则 T'_i 的所有结点均在 T_i 内, 且 T_i 的结点数量比 T'_i 多, 与假设矛盾), 进而可以得到 $q \in S(T_i)$, 根据**命题 2**, 有 $r_q > r(T_i)$, 根据**命题 3**, T'_i 包含 T_q , 由于 $r_q > r(T_i) = r(T'_i)$, 根据**命题 1**, T'_i 在合并 T_q 之前应该具有更小的 r 值, 表明 $r(T'_i)$ 不是最优的, 产生矛盾, 故原假设不成立, T_i 唯一。 ■

先不考虑结点权值的改变, 思考如何确定给定结点 i 的 r_i 和 T_i 。

对于某个结点 i 来说, 我们在第一时间无法确定应该添加哪些结点来使得 r_i 更小, 因为在添加顺序上有限制, 必须先添加父结点才能添加子结点, 而有时会出现这样的一种情形: 添加某个结点 j 后, r_i 是暂时变大的, 而再继续添加结点 j 的子结点 k 后, r_i 会变小。根据**命题 2** 和**命题 3**, 如果反过来, 假设已经确定了结点 i 的所有子结点 i_1, i_2, \dots, i_k 对应的 $r_{i1}, r_{i2}, \dots, r_{ik}$ 和 $T_{i1}, T_{i2}, \dots, T_{ik}$, 那么对于某个子结点 j 来说, 如果 r_j 是所有子结点中最小的, 若 r_j 大于 $\frac{a_i}{b_i}$, 显然不应将结点 i 与 T_j 合并, 因为这将使得 r_i 变大, 若 r_j 小于或等于 $\frac{a_i}{b_i}$, 那么应该将结点 i 与 T_j 合并, 这样可以得到一棵以结点 i 为根且具有更小 r_i 值的子树。由于合并之后, r_i 的最优值会发生改变且 T_j 的后继结点 $S(T_j)$ 也可能被合并以优化 r_i 和 T_i , 因此需要将其与原来未合

并的子结点一并考虑。这提示我们应该考虑一种自底向上的方法来确定 r_i 和 T_i ，因此有以下的初步算法：

算法 1

- (1) 如果结点 i 是叶结点，则 T_i 为结点 i 本身， $r_i = \frac{a_i}{b_i}$ ；
- (2) 如果结点 i 不是叶结点，令其子结点为 i_1, i_2, \dots, i_k ，假设已经确定了所有子结点的 $r_{i1}, r_{i2}, \dots, r_{ik}$ 和 $T_{i1}, T_{i2}, \dots, T_{ik}$ ，置 $T_i \leftarrow \{i\}$ ， $r_i \leftarrow \frac{a_i}{b_i}$ ，将结点 i 的所有子结点加入结点集合 P ，即置 $P \leftarrow \{i_1, i_2, \dots, i_k\}$ ；
- (2.1) 从 P 中选择一个结点 j ，使得 $r_j = \min_{k \in P} r_k$ ；
- (2.2) 如果 $r_j > r_i$ ，算法结束；
- (2.3) 将 T_j 与 T_i 合并，即置 $T_i \leftarrow T_i \cup T_j$ ；
- (2.4) 将 j 从 P 中删除，并将 $S(T_j)$ 加入 P ，即置 $P \leftarrow P \cup S(T_j) - \{j\}$ ；
- (2.5) 若 $P = \emptyset$ ，算法结束，否则转步骤 (2.1)。

可以使用深度优先遍历来实现**算法 1**，如果在寻找最小值时使用朴素的线性扫描方法，则可以在 $O(n^2)$ 的时间内确定某个结点 i 的 r_i 和 T_i 。

现在来尝试改进**算法 1** 的时间复杂度。注意到在**算法 1** 中，每次均需要从集合 P 中选择一个具有最小 r 值的结点，如果使用朴素的线性扫描，时间复杂度为 $O(n)$ ，如果使用可并堆 (mergeable heap)，则可以将该步骤的时间复杂度降低为 $O(\log n)$ ，从而使得**算法 1** 的时间复杂度降低。

算法 2

$$\text{令 } \hat{a}_i = \sum_{j \in T_i} a_j, \quad \hat{b}_i = \sum_{j \in T_i} b_j.$$

- (1) 如果结点 i 是叶结点，则 T_i 为结点 i 本身，置 $r_i \leftarrow \frac{a_i}{b_i}$ ， $\hat{a}_i \leftarrow a_i$ ， $\hat{b}_i \leftarrow b_i$ ， $MH_i \leftarrow \emptyset$ ；
- (2) 如果结点 i 不是叶结点，令其子结点为 i_1, i_2, \dots, i_k ，假设已经确定了所有子结点的 $r_{i1}, r_{i2}, \dots, r_{ik}$ 和 $T_{i1}, T_{i2}, \dots, T_{ik}$ ，置 $T_i \leftarrow \{i\}$ ， $r_i \leftarrow \frac{a_i}{b_i}$ ， $\hat{a}_i \leftarrow a_i$ ， $\hat{b}_i \leftarrow b_i$ ， $MH_i \leftarrow \{(i_1, r_{i1}), (i_2, r_{i2}), \dots, (i_k, r_{ik})\}$ ；
- (2.1) 若 $MH_i = \emptyset$ ，算法结束，否则选择 MH_i 中的一个配对 (j, r_j) ，该配对的第二个分量的值最小；
- (2.2) 若 $r_i \geq r_j$ ，置 $\hat{a}_i \leftarrow \hat{a}_i + \hat{a}_j$ ， $\hat{b}_i \leftarrow \hat{b}_i + \hat{b}_j$ ， $r_i = \frac{\hat{a}_i}{\hat{b}_i}$ ；
- (2.3) 置 $MH_i \leftarrow MH_i \cup MH_j - \{(j, r_j)\}$ ，转步骤 (2.2)；
- (2.4) 否则，若 $r_i < r_j$ ，算法结束。

现在来分析**算法 2** 的时间复杂度。将配对插入可并堆的操作最多执行 n 次，从可并堆中获取第二个分量值最小的操作执行的次数最多是可并堆合并次数的两倍，即 $2n$ 次，从可并堆中删除配对操作执行的次数是可并堆合并的次数，即 n 次。由于可并堆的合并是在不相交集合之间的合并（因为 $MH_i \cap MH_j = \emptyset$ ），因此最多的合并次数为 n 次。故在可并堆上的操作最多为 $5n$ 次，而每次可并堆的操作时间可以做到 $O(\log n)$ ，因此总的时间复杂度为 $O(n \log n)$ 。

由于题目要求在确定某个结点 i 的 r_i 和 T_i 后，需要对所有结点 $j \in S(T_i)$ 执行操作： $a_j \leftarrow a_j + \lceil r_i \rceil$ ，可行的方法是反复调用**算法 1**，时间复杂度为 $O(n^3)$ ，如果反复调用**算法 2**，时间复杂度为 $O(n^2 \log n)$ 。

通过进一步的思考，可以发现以下事实：令根结点为 i ，当使用**算法 2** 确定 r_i 和 T_i 后，由于是使用自底向上的方式来确定的，此时对于树中的所有结点 j ， r_j 和 T_j 均已确定，接下来需要对在 $S(T_i)$ 中的结点进行权值更新的操作，考虑某个结点 $k \in S(T_i)$ ，当更新 $a_k \leftarrow a_k + \lceil r_i \rceil$ 后，可能需要重新确定 r_k 和 T_k ，但由于 k 的子结

点和后代结点 l 的 r_l 和 T_l 并未改变, 此时并不需要再次调用 **算法 2** 从头开始计算 r_k 和 T_k , 而只需要利用在确定 r_i 和 T_i 时所得到的结果。因此有以下优化的算法^I (文献引用 1):

算法 3

- (1) 设立队列 Q , 初始时将根结点加入 Q ;
- (2) 从队列 Q 中取队首元素, 令其为 i , 使用 **算法 2** 确定结点 i 的 r_i 和 T_i ;
- (3) 对于结点 $j \in S(T_i)$, 置 $a_j \leftarrow a_j + [r_i]$, $r_j \leftarrow \frac{a_j}{b_j}$, 并将 j 加入 Q ;
- (4) 从 Q 中删除 i , 若队列为空, 算法停止, 否则转步骤 (2)。

现在来分析 **算法 3** 的时间复杂度。与 **算法 2** 类似, 除了在寻找具有最小第二分量值的配对时的操作次数最多为 $3n$ 次之外, 其他的操作次数没有变化, 而每次在可并堆上的操作时间复杂度可以做到 $O(\log n)$, 因此总的时间复杂度仍为 $O(n \log n)$ 。由于 **算法 3** 中可并堆 MH_i 在合并后在后续仍可能需要使用, 如果不使用的话就无法保证 $O(n \log n)$ 的时间复杂度, 因此必须使用可持久化的可并堆 (persistent mergeable heap)。

在 **算法 3** 中, 每执行一次步骤 (2) 就会生成一棵有根树, 令第 i 次生成的有根树为 K_i , 将每棵有根树使用一个结点 i 来表示, 其权值为 $w_i = [r(K_i)]$, 则可以得到一棵新的有根树 T_w , 最小化 W 值的问题可以转化为以下问题: 从根结点开始, 逐次删除结点, 只有在删除父结点后才能删除子结点, 第 i 次删除结点的代价为结点权值与删除次数序号的乘积, 如何最小化删除所有结点的代价之和。

根据 **命题 4**, T_i 是唯一的, 因此生成的这棵新的有根树也是唯一的, 根据 **命题 2**, 可知这棵新的有根树, 从根结点出发, 向着叶结点的任意一条路径走, 所经过的结点的权值必定是递增的。根据这两个性质, 似乎可以采用这样一种贪心的策略: 在当前可删除的结点中选择一个权值最大的结点进行删除。依照这种思路, 对于输入#3 来说, 激活细胞的最优顺序应该是: 5、1、2、3、4, 最终得到的最小删除代价为 247, 并不是最优的。什么地方出问题了? 关键还是在于删除具有先后顺序, 如果删除不具有先后顺序, 直接选择权值最大的结点进行删除肯定可以得到最小的删除代价, 但是一旦具有先后顺序, 只选取当前可删除的结点中的具有权值最大的结点可能无法做到最优。举个例子, 假设有三个结点 a , b , c , 结点 b 是结点 c 的父结点, 从第 t 次开始依次删除这三个结点, 结点 b 和 c 要求连续删除, 则有两种删除顺序, 第一种顺序是 a , b , c , 第二种顺序是 b , c , a , 第一种顺序的删除代价为:

$$W_1 = t \times w_a + (t+1) \times w_b + (t+2) \times w_c$$

第二种删除顺序的代价是:

$$W_2 = t \times w_b + (t+1) \times w_c + (t+2) \times w_a$$

我们来比较两种顺序的删除代价:

$$W_1 - W_2 = w_b + w_c - 2w_a$$

需要判断 $w_b + w_c - 2w_a$ 的大小, 将其转换一下, 实际上就是要判断 $(w_b + w_c)/2$ 和 w_a 的大小关系, 也就是说, 如果结点 b 和 c 的平均权值大于结点 a 的权值, 则应该选择第二种删除顺序, 即先删除 b 和 c , 然后再删除 a , 否则就应该先删除 a , 在删除 b 和 c 。这提示我们, 删除的优先顺序不是与单个结点的权值相关, 而是与多个结点的平均权值有关。可以证明, 由于删除的代价与删除的次序序号成线性关系, 当结点数增多时,

^I 参考文献: Galil Z. Applications of efficient mergeable heaps for optimization problems on trees [J]. Acta Informatica, 1980, 13: 53-58.

上述性质仍然成立（文献引用 2）^I。那么问题转化为以下的问题：对于有根树中的每个结点 i ，确定以结点 i 为根的具有最多结点的子树 T_i ，该子树 T_i 具有最大的平均权值。按照平均权值进行贪心选择来删除结点，能够获得最小化的删除代价。而这个问题，实际上就是将第一次调用算法 3 得到的新的有根树 T_w 的结点权值设置为 $a_i=w_i=\lceil r(K_i) \rceil$, $b_i=1$ ，从根结点开始再调用一次算法 2，只不过将算法 2 中的取最小值改成取最大值。

在确定了新的有根树 T_w 中所有结点 i 的 r_i 和 T_i 后，令 W 表示总代价， t 为当前的操作序号，则可以使以下算法来确定 W 的最小值：

算法 4

- (1) 初始时，置 $W \leftarrow 0$, $t \leftarrow 1$, 将根结点的配对 (i, r_i) 送入堆 H ;
- (2) 若 $H = \emptyset$, 算法停止, 否则取 H 中的一个配对 (j, r_j) , 该配对的第二个分量值最大;
- (3) 置 $W \leftarrow W + t \times w_j$, $t \leftarrow t + 1$;
- (4) 从 H 中删除配对 (j, r_j) , 对于 $k \in S(T_j)$, 将配对 (k, r_k) 送入 H , 转步骤 (2)。

最后，算法 4 所得到的值 W 即为解。如果使用二叉堆来实现算法 4，易知其时间复杂度为 $O(n \log n)$ 。

2.22 伸展树

2.22.1 伸展树的操作

2.22.2 伸展树的应用

以下通过一道例题的分析来示例伸展树的应用^{II}。

NOI 2005 维护数列

给定一个数列，要求完成以下六种操作。

操作	输入格式	功能
插入	INSERT $posi\ tot\ c_1\ c_2\ c_3\ \dots\ c_{tot}$	在当前数列的第 $posi$ 个数字后插入 tot 个数字： c_1, c_2, \dots, c_{tot} ；若在数列首插入，则 $posi$ 为 0
删除	DELETE $posi\ tot$	从当前数列的第 $posi$ 个数字开始连续删除 tot 个数字
修改	MAKE-SAME $posi\ tot\ c$	将当前数列的第 $posi$ 个数字开始的连续 tot 个数字统一修改为 c
翻转	REVERSE $posi\ tot$	取出从当前数列的第 $posi$ 个数字开始的 tot 个数字，翻转后放入原来的位置

^I 参考文献：Horn W A. Single-machine job sequencing with treelike precedence ordering and linear delay penalties [J]. SIAM Journal on Applied Mathematics, 1972, 23(2): 189-202.

^{II} 此题目对应洛谷主题库中编号为 P2042 的题目。

求和	GET-SUM <i>posi tot</i>	计算从当前数列第 <i>posi</i> 个数字开始的 <i>tot</i> 个数字的和并输出
最大连续子序列和	MAX-SUM	输出当前数列中的最大连续子序列和

分析

2.23 动态树

2.24 树链剖分

2.25 算法库函数

2.25.1 accumulate 和 count、count_if

accumulate 返回指定范围内各元素的累加。count 函数返回指定范围内的某个值出现的次数。count_if 返回指定范围内满足特定条件的值出现的次数。注意 accumulate 函数包含在头文件<numeric>中，而不是常见的<algorithm>中。

```
#include <numeric>
template <class InputIterator, class T>
T accumulate(InputIterator first, InputIterator last, T init);

#include <algorithm>
template <class InputIterator, class T>
typename iterator_traits<InputIterator>::difference_type
count(InputIterator first, InputIterator last, const T& val);

template <class InputIterator, class UnaryPredicate>
typename iterator_traits<InputIterator>::difference_type
count_if(InputIterator first, InputIterator last, UnaryPredicate pred);
```

在使用 accumulate 进行求和时，需要指定元素的类型，可以使用如下的方式进行指定：

```
// 求 1 至 100 的累加和。
vector<int> numbers(100);
iota(numbers.begin(), numbers.end(), 1);
int sum = accumulate(numbers.begin(), numbers.end(), int(0));
```

强化练习：414 Machined Surfaces^A。

2.25.2 copy 和 reverse_copy

copy 函数将数组或序列指定范围内的元素复制到目标数组或序列指定起始位置的内存单元中，而 reverse_copy 在复制时将原始的数据反向复制到目标地址，其函数声明为：

```
template <class InputIterator, class OutputIterator>
```

```
OutputIterator copy(InputIterator first, InputIterator last, OutputIterator result);

template <class BidirectionalIterator, class OutputIterator>
OutputIterator reverse_copy(BidirectionalIterator first, BidirectionalIterator
last, OutputIterator result);
```

如下例所示，将第一个 `vector` 中的元素复制到第二个 `vector` 中。

```
vector<int> v1, v2;
for (int i = 1; i < 100; i++) v1.push_back(i);
// 注意，需要为向量 v2 预先分配存储空间。
v2.resize(100);
copy(v1.begin(), v1.end(), v2.begin());
```

类似的算法库函数还有：

(1) `copy_backward`: 将源序列指定范围内的元素逆序复制到目标序列以指定位置作为结束的范围内。函数声明为：

```
template <class BidirectionalIterator1, class BidirectionalIterator2>
BidirectionalIterator2 copy_backward(BidirectionalIterator1 first,
BidirectionalIterator1 last, BidirectionalIterator2 result);
```

(2) `copy_if`^{C++11}, 将源序列中指定范围内满足指定条件的元素复制到目标序列指定位置开始的范围内。函数声明为：

```
template <class InputIterator, class OutputIterator, class UnaryPredicate>
OutputIterator copy_if(InputIterator first, InputIterator last, OutputIterator
result, UnaryPredicate pred);
```

(3) `copy_n`^{C++11}, 将源序列指定位置开始的 n 个元素复制到目标序列以指定位置开始的范围内。函数声明为：

```
template <class InputIterator, class Size, class OutputIterator>
OutputIterator copy_n(InputIterator first, Size n, OutputIterator result);
```

2.25.3 fill

`fill` 是一个模板函数，其作用是将指定地址范围内的所有存储单元设置为指定的值。

```
#include <algorithm>
template <class ForwardIterator, class T>
void fill(ForwardIterator first, ForwardIterator last, const T& item);
```

通过 `memset` 可能无法达到为数组统一赋值的目的，而通过 `fill` 却可以。当数组较大时，使用 `for` 循环和使用 `fill` 函数为数组赋予初值，其效率相差不大，不过均比 `memset` 的效率要低。

```
//-----2.25.3.cpp-----//
int main(int argc, char *argv[]) {
    int number[10];
    for (int i = 0; i < 10; i++) {
        number[i] = i + 1;
        cout << setw(3) << right << number[i];
    }
    cout << endl;
```

```

        fill(number + 5, number + 10, 5);
        for (int i = 0; i < 10; i++)
            cout << setw(3) << right << number[i];
        cout << endl;
        return 0;
    }
//-----2.25.3.cpp-----//

```

输出为：

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

2.25.4 iota^{C++11}

`iota` 是随 C++11 标准新增的一个函数，该函数包含在头文件`<numeric>`中，函数声明为：

```

#include <numeric>
template <class ForwardIterator, class T>
void iota(ForwardIterator first, ForwardIterator last, T val);

```

`iota` 最初来自于艾弗森^I于 1972 年开发的编程语言——APL^{II}。`iota` 在 APL 中的作用是获得一个从 1 到 N 的序列，用于模拟 `for` 循环。在 C++11 标准中，`iota` 的作用是给序列容器中指定范围内的元素赋值，从指定的初值开始，每给一个元素赋值，其值增加 1，最后的效果是构成一个从初值开始项差为 1 的递增序列。

```

//-----2.25.4.cpp-----//
int main(int argc, char *argv[]) {
    // 声明一个大小为 10 的 vector，初始值全为 0。
    vector<int> numbers(10, 0);

    // 将第一个至第五个元素顺序赋值为 1, 2, 3, 4, 5，后五个元素的值仍为 0。
    iota(numbers.begin(), numbers.begin() + 5, 1);
    for (auto number : numbers)
        cout << setw(2) << right << number;
    cout << '\n';
    return 0;
}
//-----2.25.4.cpp-----//

```

输出为：

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

2.25.5 max 和 min

`max` 和 `min` 函数的功能很简单，返回给定两个值的较大值和较小值，当两者相等时，返回前一个参数。函数声明为：

^I 肯尼斯·艾弗森 (Kenneth E. Iverson, 1920—2004)，加拿大数学家、计算机学家，编程语言 APL 的发明者，1979 年第 14 届图灵奖 (Turing Award) 获得者。

^{II} A Programming Language 或 Array Processing Language 的缩写，其名称取自希腊字母表的第九个字母 ‘ι’，读音为 ‘Iota’。

```
template <class T> const T& max(const T& a, const T& b);
template <class T> const T& min(const T& a, const T& b);
```

一般使用函数的第一个版本，即不带比较函数的版本，此时将使用默认的小于比较运算符对给定的两个参数进行比较，然后返回合适的值。当需要更改比较规则时，可以使用带自定义比较函数的第二种版本，增加了灵活性。需要注意，传入的函数的参数类型需要一致。例如，以下的代码在进行编译时编译器可能会报错：

```
string s1 = "abcdefg";
int maxLength = 0;
// 编译器报错。
maxLength = max(maxLength, s1.size());
```

因为 `string` 类的 `size` 属性其数据类型为 `size_t`，需要显式的将其转换为 `int` 数据类型方能正确编译。

强化练习：[12372](#) Packing for Holiday^A。

2.25.6 `max_element` 和 `min_element`

返回指定范围元素的最大值或最小值。函数声明为：

```
// 返回指定范围内指向具有最大值元素的迭代器，如果指定范围内有多个最大值，则返回第一个。
// 由于返回的是一个迭代器或地址，如果需要获取具体的数值，需要使用解引用操作。
template <class ForwardIterator>
ForwardIterator max_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator>
ForwardIterator min_element(ForwardIterator first, ForwardIterator last);

template <class ForwardIterator>
pair<ForwardIterator, ForwardIterator>
minmax_element(ForwardIterator first, ForwardIterator last);
```

以上函数均有默认版本和自定义比较器的版本，使用默认版本将使用小于运算符对元素的大小进行比较。当获取范围不大的数组或序列容器中数据的最大值或最小值时，使用这两个函数很方便，不再需要通过循环去逐一比较以获取最大值或最小值，较为简便。

强化练习：[499](#) What's The Frequency Kenneth^A，[11364](#) Optimal Parking^A。

2.25.7 `memcpy` 和 `memset`

`memcpy` 的作用是通过内存直接复制的方法，将指定长度的若干字节从源起始地址复制到目标起始地址。

```
// 使用 memcpy 函数，需要包含头文件 cstring。
#include <cstring>
void* memcpy(void* destination, const void* source, size_t num);
```

注意，`memcpy` 函数的第一个参数为目标起始地址，第二个参数为源起始地址，第三个参数为复制的字节数目。

```
const int MAXN = 110;
int n, grid[MAXN][MAXN], temp[MAXN][MAXN];
// 对 grid 和 temp 进行特定操作后再将 temp 复制回 grid 数组。
memcpy(grid, temp, sizeof(temp));
```

强化练习：[12187 Brothers](#)。

`memset` 函数的作用是将指定起始地址的若干个字节填充为特定的值^[24]。

```
// 使用 memset 函数需要包含头文件<cstring>。
#include <cstring>
void* memset(void* ptr, int value, size_t num);
```

函数的参数 `ptr` 表示需要填充的起始地址, `value` 表示需要填充的值 (只取该值的低位字节), `num` 表示需要填充的字节数量。如下例所示, 使用 `memset` 将指定字符数组的某个连续范围的字节设置为某一个特定值。

```
//-----2.25.7.1.cpp-----
int main(int argc, char *argv[]) {
    char sentence[] = "the quick brown fox jumps over the lazy dog.";
    // 将给定字符串的前四个字符修改为连字符 '-'。
    memset(sentence, '-', 4);
    cout << sentence << endl;
    return 0;
}
//-----2.25.7.1.cpp-----
```

输出为:

```
----quick brown fox jumps over the lazy dog.
```

可以使用 `memset` 函数将整数数组、布尔值数组或者结构体中的元素置为 0。

```
//-----2.25.7.2.cpp-----
// 定义一个结构体。
struct tag {
    int age;
    bool gender;
    char name[20];
};

int main(int argc, char *argv[]) {
    // 定义一个含有 10 个元素的整数数组, 赋值并输出。
    int data[10];
    for (int i = 0; i < 10; i++) {
        data[i] = i + 1;
        cout << data[i] << " ";
    }
    cout << endl;

    // 将 data 数组的第 2 个至第 6 个元素设置为 0 然后输出。
    memset(data + 1, 0, sizeof(int) * 5);
    for (int i = 0; i < 10; i++) cout << data[i] << " ";
    cout << endl;

    // 声明一个结构体, 赋初始值然后输出。
    tag t = tag{1, true, "the brown fox"};
    cout << t.age << " " << t.gender << " " << t.name << endl;

    // 将结构体的元素值置为 0 然后输出。
    memset(&tag, 0, sizeof(tag));
```

```

    cout << tag.age << " " << tag.gender << " " << tag.name << endl;
    return 0;
}
//-----2.25.7.2.cpp-----//

```

输出为：

```

1 2 3 4 5 6 7 8 9 10
1 0 0 0 0 7 8 9 10
1 1 the brown fox
0 0

```

`memset` 函数是以字节为单位对指定存储区间进行填充，如果忽略了这一点，在使用时会出现与编程预期不一致的赋值结果。

```

//-----2.25.7.3.cpp-----//
int main(int argc, char *argv[]) {
    int data[6];
    memset(data, 1, sizeof(data));
    for (int i = 0; i < 6; i++) cout << data[i] << " ";
    cout << endl;
    return 0;
}
//-----2.25.7.3.cpp-----//

```

以上代码的预期是将数组 `data` 的所有元素都设置为 1，但是输出却为：

```

16843009 16843009 16843009 16843009 16843009 16843009

```

为什么会出现这种结果呢？原因就是 `memset` 按字节进行填充所致。一个整数占四个字节的存储空间，`memset` 将每个字节均填充为数值 1，那么四个字节连起来对应的二进制数及其对应的十进制数为

$$00000001000000010000000100000001_2 = 16843009_{10}$$

一般情况下，通过 `memset` 函数将整数数组统一赋值为同一个非 0 值是无法实现的，除非要赋予的值转化为二进制数时四个字节具有相同的值，那么使用 `memset` 可以达到目的，例如语句 `memset(data, -1, sizeof(data))` 可以实现将 `data` 数组全部赋值为 -1 的效果。因为 -1 为有符号整数，在计算机中一般使用 2 的补码表示，其对应的二进制数为 $11111111111111111111111111111111_2$ ，在赋值时取其低位字节为 11111111_2 ，最后总的效果是将数组所占内存空间的所有字节均设置为 11111111_2 ，由于是有符号整数，故计算机将整数数组中的元素全部解释为 -1，达到了预期效果。

2.26 小结

Pascal 之父——Nicklaus Wirth 曾提出：“算法 + 数据结构 = 程序”，可见数据结构的重要性。在编程竞赛中，理清数据的内在联系，合理地组织数据，才能对他们进行有效地处理，设计出高效的算法。使用正确的数据结构，往往能够使程序的效率得到提高，甚至使问题迎刃而解。学习数据结构，不仅是学习其使用方法，学习其思想同样非常重要，因为很多题目都是围绕数据结构的核心思想而设置，需要解题者在理解数据结构的基础上加以适当变换思维方式进行解决。数据结构是后续章节内容的基础，例如，最小生成树、哈夫曼树、搜索树等都是以树为基础的概念，在图遍历中，使用向量以邻接表的形式来表示图是常见的操作，链式前向星则是一种应用结构体和数组来高效表示图的数据结构。

数据结构的内容非常丰富，在既往的竞赛中也常常是重点内容，要想熟练的掌握不是一件容易的事，需

要从基础的数据结构学起，逐个予以掌握。对于每种数据结构，不仅需要了解其善于处理的问题，也要了解其弱点和不足，在解题时才能做到“扬长避短”。本章只是介绍了较为常见的数据结构，对于一些高级的数据结构（例如可持久化线段树、k-d 树、动态树、最小树形图、左偏树等）并未予以介绍，需要读者在已经掌握基本数据结构的基础上，自行对这些高级的数据结构进行理解和学习，在这个过程中，需要查阅网上的资料，锻炼自己的自学能力。

对于较难理解的数据结构，关键在于理解其核心思想，在理解核心思想的基础上，结合一些简单的样例能够直观地理解数据结构的具体应用，能够帮助进一步加深对数据结构的理解。之后通过练习题，可以积累数据结构应用的经验。

第 3 章 字符串

Unicode 给每个字符提供了一个唯一的数字，不论是什么平台、不论是什么程序、不论是什么语言。

——Unicode 官方网站^I

据美国互联网研究机构 Netcraft 于 2019 年 8 月发布的调查报告显示，全世界活跃网站数量已经超过 12.7 亿个^{II}；人类基因组中大约有 30 亿个碱基对^{III}；在线销售网站的数据库动辄 100TB……在这些大量数据中，字符串（text strings）是一种非常重要的基础数据结构，因此人们对字符串的处理非常重视，也因此提出了很多有效的方法。在 UVa 的习题中，很多题目与字符串操作有关，熟悉 C++ 中的字符串表示以及操作方法对解题很有帮助。

3.1 编码

从本质上讲，计算机处理的字符可以看成是单个的数字，而字符编码就是一个特定字符集中的符号和数字之间的映射。美国标准信息交换码（American Standard Code for Information Interchange，ASCII）是一个包含 128 个字符的单字节编码，它于 1963 年发布第一个版本^{IV}。ASCII 作为计算机处理信息的一个基础标准，在计算机发展史中的地位非常重要。由于在设计时将表示数字的 10 个字符和表示大小写字母的字符都安排在连续的位置，编程实践中可以利用这个特点来进行特定转换。比如，将数字字符和对应的数值进行互相转换，将大写字母和小写字母进行互相转换。在编程中需要记忆的几个范围：数字从 0 到 9 对应的 ASCII 码值为 48 至 57，小写字母 ‘a’ 至 ‘z’ 对应的 ASCII 码值为 97 至 122，大写字母 ‘A’ 至 ‘Z’ 对应的 ASCII 码值为 65 至 90。

随着计算机技术的发展，各个国家和组织都制定了不同的字符编码方案，导致了一种“混乱”的情况出现——“不同编码对应同一个字符，不同字符又具有同一个编码”，这对信息的交换和统一处理带来了很大的障碍。为了消除这种障碍，迫切需要一个国际统一表示的字符集，因此出现了 Unicode，即统一码。截止 2020 年 1 月 1 日，该标准发布了 12.0.0 版本。正因为有了统一的编码，编写国际化的程序、在不同平台之间进行程序移植较以往更为便利。该标准被世界上的绝大多数 IT 企业所支持，如 Apple（苹果），HP（惠普），IBM（国际商业机器公司），JustSystem，Microsoft（微软），Oracle（甲骨文），SAP，Sun，Sybase 等。

Unicode 有三种编码方案，分别是 UTF - 8，UTF - 16，UTF - 32，其中 UTF 是 Unicode 字符集转换格式（Unicode Transformation Format，UTF）的缩写，它定义了如何将 Unicode 中字符对应的数字转换成程序中使用的数据形式。三种编码方案分别使用 8 位、16 位、32 位二进制数来编码字符。

为了使用上的便利，Unicode 将字符按照语义和功能进行了分类，将 0 至 $10FFFF_{16}$ 的编码空间划分成

^I <http://www.unicode.org/>，2020。

^{II} <https://news.netcraft.com/>，2020。

^{III} https://en.wikipedia.org/wiki/Human_genome，2020。

^{IV} 本书附录包含了一份 ASCII 表以备参考。

17个平面 (plane)，每个平面包含 64K (2^{16}) 个编码点，以下为各平面的划分。

平面	编码范围	命名	备注
0	U+00000 至 U+0FFFF	基本多文种平面 (Basic Multilingual Plane, BMP)	包含了几乎所有的常见字符
1	U+10000 至 U+1FFFF	多文种补充平面 (Supplementary Multilingual Plane, SMP)	包含了不常用的字符
2	U+20000 至 U+2FFFF	表意文字补充平面 (Supplementary Ideographic Plane, SIP)	
3至13	U+30000 至 U+DFFFF	意向命名为第三表意平面 (Tertiary Ideographic Plane, TIP)	12.0 版本中，这些平面均尚未使用
14	U+E0000 至 U+EFFFF	特别用途补充平面 (Supplementary Special-purpose Plane, SSP)	12.0 版本中，该平面均尚未使用
15	U+F0000 至 U+FFFFF	保留作为私人使用区 (Private Use Area, PUA)	
16	U+100000 至 U+10FFFF	保留作为私人使用区 (Private Use Area)	

常用的简体中文字符包含在平面 0 的“中日韩统一表意文字”区间 (U+4E00 至 U+9FFF)。需要注意，常用的汉字（基本等同于 GBK 字符集¹，大约有 21000 个汉字）使用 UTF-8 进行编码时，占用的是 3 个字节，而对于不常用的汉字，其编码可能为 4 个字节。

强化练习: 458 The Decoder^A, 10082 WERTYU^A, 10222 Decode the Mad Man^A, 10851 2D Hieroglyphs Decoder^B, 10878 Decode the Tape^A, 11483 Code Creator^B, 13049 Combination Lock^D。

扩展练习: 12555 Baby Me^C。

3.2 字符串类

若需要在 C++ 中表示一个字符串，有以下几种方法：

(1) 使用字符内置数组。内置数组在处理字符串时，插入和删除字符会较为繁琐，因为内置数组是固定的。

(2) 使用 `vector` 容器类，以 `char` 类型来进行实例化。此种方法便于字符串的处理，缺点是占用空间较大，处理效率稍慢。

(3) 使用字符链表。此种方法空间利用率较低，但是对于需要频繁进行插入和删除操作的情形，效率一般较 `vector` 要高。

使用过 C 的人都知道，处理字符串在很多情况下让人非常头痛，常常为了一个简单的功能而需要在代码上大动干戈。到了 C++，情况变得乐观了许多，因为标准库提供了 `string` 类，大大提高了处理字符串时的编程效率。`string` 类在标准库的源文件中是这样定义的：

¹ GBK 是一个汉字编码标准，全称《汉字内码扩展规范》(Chinese Internal Code Specification)，GB 即“国标”，K 是“扩展”汉语拼音的第一个字母。

```
typedef basic_string<char> string;
```

即 `string` 是一个用内置 `char` 类型进行实例化的 `basic_string` 模板类，属于容器类的范畴。在表达功能上，可以把 `string` 类实例视为一个字符数组，尽管效率上可能不会总是好于字符数组，但是 `string` 类所提供的操作功能是字符数组望尘莫及的，所以使用 C++ 语言参加编程竞赛时，能用 `string` 类的地方可以考虑优先使用。

强化练习：[12896 Mobile SMS^B](#)。

3.2.1 声明

`string` 类的标准头文件是`<string>`，在 GCC 的 SGI 库实现中，该头文件已经被头文件`<iostream>`所包含，因此只需要包含头文件`<iostream>`，不再包含头文件`<string>`。如果需要使用兼容 C 的字符串功能，需要包含头文件`<cstring>`。

`string` 类的声明有多种方式，可以根据具体情况灵活选用以提高效率。

```
string
声明一个字符串实例。
string();
string(const string& str);
string(const string& str, size_t pos, size_t len = npos);
string(const char* s);
string(const char* s, size_t n);
string(size_t n, char c);
template <class InputIterator> string(InputIterator first, InputIterator last);
```

以下示例各种声明方式的使用方法。

```
-----3.2.1.cpp-----
int main(int argc, char *argv[]) {
    // string()
    // 默认初始化方法，空字符串。
    string s0;
    // s0 = ""

    // string(const char* s)
    // 使用字符串常量或字符数组进行初始化。
    string s1("the quick brown fox jumps over the lazy dog.");
    // s1 = "the quick brown fox jumps over the lazy dog."

    // string(const string& str)
    // 使用另外一个 string 类实例来初始化。
    string s2(s1);
    // s2 = "the quick brown fox jumps over the lazy dog."

    // string(const string& str, size_t pos, size_t len = npos)
    // 使用另外一个 string 类实例，从指定位置 pos 开始，取 len 个字符，第三个
    // 参数可以省略，若省略第三个参数，则从指定位置 pos 开始取到字符串末尾。
    string s3(s1, 4, 5);
    string s4(s1, 10);
    // s3 = "quick"
    // s4 = "brown fox jumps over the lazy dog."

    // string(const char* s, size_t n)
```

```

// 从字符串常量或者字符数组的指定位置开始取字符进行初始化。
string s5("the quick brown dog jumps over the lazy fox.", 9);
char data[64] = "the quick brown dog jumps over the lazy fox.";
string s6(data, 9);
// s5 = "the quick"
// s6 = "the quick"

// string (size_t n, char c)
// 以指定数量的特定字符来填充字符串, 数值 35 是字符 '#' 的 ASCII 值。
string s7(10, 'A');
string s8(10, 35);
// s7 = "AAAAAAAAAA"
// s8 = "#####"

// template <class InputIterator>
// string(InputIterator first, InputIterator lsst)
// 使用迭代器来指定 string 类实例的特定范围来进行新 string 类的初始化。
string s9(s1.begin(), s1.begin() + 9);
// s9 = "the quick"

return 0;
}
//-----3.2.1.cpp-----//

```

强化练习: [488 Triangle Wave^A](#)。

3.2.2 赋值

如果已经声明了一个 string 类实例, 当前需要更改其内容, 可以使用 string 类的 assign 方法。

assign

为 string 类实例赋值, 更改其内容。

```

string& assign(const string& str, size_t subpos, size_t sublen);
string& assign(size_t n, char c);

```

以下为 assign 使用方法的示例。

```

//-----3.2.2.cpp-----//
int main(int argc, char* argv[]) {
    char s1[] = "the quick brown fox jumps over the lazy dog.";
    string s2(s1), s3;

    s3.assign(s2);
    // s3 = "the quick brown fox jumps over the lazy dog."

    s3.assign(s2, 10, 5);
    // s3 = "brown"

    s3.assign(s1);
    // s3 = "the quick brown fox jumps over the lazy dog."

    s3.assign(s1, 9);
    // s3 = "the quick"

    s3.assign(10, 'A');
    // s3 = "AAAAAAAAAA"

```

```

s3.assign(s2.begin(), s2.begin() + 9);
// s3 = "the quick"

return 0;
}
//-----3.2.2.cpp-----//

```

3.2.3 遍历

在字符串的操作中, 对字符串中字符逐个进行处理的方式很常见, 需要遍历操作的支持。`string` 类支持两种方式的遍历, 一种是传统的基于下标的遍历, 另外一种是使用容器类的遍历接口, 即迭代器(iterator)。在程序竞赛的场景中, 不需要考虑程序的健壮性(robustness)、可维护性(maintainability), 为了节省源代码的输入时间, 一般均直接采用基于下标的访问方式。以下列出了与遍历操作相关的属性和方法。

<code>begin()</code>	返回指向字符串第一个字符的迭代器。
<code>end()</code>	返回指向字符串最后一个字符之后一个位置的迭代器。
<code>rbegin()</code>	返回指向字符串最后一个字符的逆序迭代器。
<code>rend()</code>	返回指向字符串第一个字符之前一个位置的逆序迭代器。
<code>front()^{c++11}</code>	获取字符串的第一个字符。
<code>back()^{c++11}</code>	获取字符串的最后一个字符。
<code>length()</code>	获取字符串的长度, 与属性 <code>size()</code> 作用相同。
<code>size()</code>	获取字符串的长度(而不是其使用的存储空间大小), 与属性 <code>length()</code> 作用相同。
<code>empty()</code>	测试字符串是否为空字符串, 为空则为 <code>true</code> , 否则为 <code>false</code> 。
<code>[index]</code>	使用下标的方式访问字符串在位置 <code>index</code> 处的字符, 不进行范围检查。
 <code>at^{c++11}</code>	
<code>at</code>	获取字符串在指定位置处的字符, 进行范围检查。
<code>char& at(size_t pos);</code>	

注意 `string` 类的 `length` 和 `size` 属性, 它们的数据类型为 `size_t`, 是专为表示字符串的长度定义的一个数据类型(与机器相关, 内部一般使用 `unsigned int` 数据类型表示), 在与其他 `int` 型数据进行大小的比较时, 建议首先将其显式转换为 `int` 类型以避免出现难以预料的副作用。

强化练习: 508 Morse Mismatches^D。

访问字符串在某个特定位置处的字符, 可以使用使用常规的下标访问方式, 或者使用 `at` 方法访问, 区别是下标访问不检查范围, `at` 方法访问会对序号的范围进行检查。获取字符串的第一个字符和最后一个字符, C++11 标准中新增了对应的 `back` 和 `front` 属性, 以增加便利性。对于使用 C++98 标准的编译器, 可以使用替代的方法实现。例如, 获取 `string` 类实例 `s` 的第一个字符, 可以通过使用 `s[0]` 得到, 获取 `s` 的最末一个字符, 可以通过使用 `s[s.length() - 1]` 来得到。

```

//-----3.2.3.cpp-----//
int main(int argc, char *argv[]) {
    string s = "the quick brown fox jumps over the lazy dog.";
    // 下标访问形式。
    for (int i = 0; i < s.length(); i++)
        cout << s[i];
    cout << endl;
    // 迭代器访问形式。
    for (string::iterator it = s.begin(); it != s.end(); it++)
        cout << *it;
    cout << endl;
}

```

```

    return 0;
}
//-----3.2.3.cpp-----//

```

输出为：

```

the quick brown fox jumps over the lazy dog.
the quick brown fox jumps over the lazy dog.

```

强化练习：129 Krypton Factor^B, 159 Word Crosses^B, 282 Rename^E, 490 Rotating Sentences^A, 553 Simply Proportion^D, 621 Secret Research^A, 865 Substitution Cypher^C, 892 Finding Words^C, 11548 Blackboard Bonanza^D, 11713 Abstract Names^A, 11830 Contract Revision^B, 13048 Burger Stand^D, 13181 Sleeping in Hostels^D。

3.2.4 连接与删除

string类实例的连接与删除可以通过以下方法实现：

clear() 清空整个字符串的内容。
pop_back()^{c++11} 删除字符串末尾的单个字符。

append
将字符或字符串添加到原字符串末尾。

```

string& append(const string& str);
string& append(size_t n, char c);

```

erase
删除指定位置开始的单个或多个字符。

```

iterator erase(iterator p);
string& erase(size_t pos = 0, size_t len = npos);

```

insert
在指定位置插入字符或字符串。

```

iterator insert(iterator p, char c);
string& insert(size_t pos, const string& str);

```

+=
重载运算符，将字符串或字符附加到原字符串末尾。

```

string& operator+=(char c);
string& operator+=(const string& str);

```

push_back
将单个字符添加到字符串末尾。

```

void push_back(char c)

```

强化练习：625 Compression^D, 739 Soundex Indexing^A, 10388* Snap^D, 11734 Big Number of Teams Will Solve This^A, 12646 Zero or One^A, 13026 Search the Khoj^C。

3.2.5 查找与替换

`string` 类实例的查找与替换可以通过以下方法实现^I:

find

从左至右在字符串中查找参数所指定的字符串（或字符）的第一次出现位置。

```
size_t find(const string& str, size_t pos = 0) const;
size_t find(char c, size_t pos = 0) const;
```

rfind

从右至左在字符串中查找参数所指定的字符串（或字符）的第一次出现位置。

```
size_t rfind(const string& str, size_t pos = npos) const;
size_t rfind(char c, size_t pos = npos) const;
```

find_first_of

从左至右在字符串中查找，如果某个字符能够匹配参数中的任意一个字符，则返回该字符所处的位置。

```
size_t find_first_of(const string& str, size_t pos = 0) const;
size_t find_first_of(char c, size_t pos = npos) const;
```

find_last_of

从右至左在字符串中查找，如果某个字符能够匹配参数中的任意一个字符，则返回该字符所处的位置。

```
size_t find_last_of(const string& str, size_t pos = npos) const;
size_t find_last_of(char c, size_t pos = npos) const;
```

replace

将字符串中指定范围的字符替换为目标字符串指定范围的字符。

```
string& replace(size_t pos, size_t len, const string& str);
```

swap

交换两个字符串的内容。

```
void swap(string& str);
```

需要注意 `rind` 方法和 `find_last_of` 方法的差异，当两者的参数都是字符串类实例时，对于 `rfind` 来说，是从右至左在目标字符串中查找参数第一次出现的位置，`find_last_of` 也是从右至左在目标字符串中查找，但是只要目标字符串中某个字符和给定的参数中任意一个字符相匹配（即不需要与整个参数相匹配），则返回目标字符串中此字符的位置，可以通过以下示例代码进一步理解。

```
-----3.2.5.cpp-----
int main(int argc, char *argv[]) {
    string s = "The quick brown fox jumps over a lazy dog";
    size_t pos1 = s.rfind("fox"), pos2 = s.find_last_of("fox");
    cout << "pos1 = " << pos1 << " pos2 = " << pos2 << '\n';
    return 0;
}
-----3.2.5.cpp-----
```

输出为：

```
pos1 = 16 pos2 = 39
```

^I 列表中只列出了解题中最为常用的一些方法，完整的方法列表请读者进一步查阅标准库手册。

另外需要注意的是 `string` 类所提供的 `replace` 方法与直观感觉上的功能有出入。与其他语言提供的字符串替换方法不同，它并不是完成“查找并替换”的功能，而只是完成将指定位置处的字符串“替换”这一功能。在 C# 中，若需要将特定字符或字符串用指定的值予以替换，可使用

```
string.Replace(char oldChar, char newChar);
```

或

```
string.Replace(string oldValue, string newValue);
```

在 Java 中可使用

```
String.replace(char oldChar, char newChar);
```

而在 C++ 中，则需要首先查找到字符位置，然后使用 `replace` 方法完成替换。较为便利的方法是要么不使用 `replace`，直接采用赋值的方法改变字符的值，要么是使用算法库函数 `replace` 来完成替换，关于算法库函数 `replace` 的具体使用参见本章的“算法库函数”一节。

强化练习：[455 Periodic Strings^A](#)，[644 Immediate Decodability^A](#)，[10115 Automatic Editing^A](#)。

3.2.6 其他操作

`string` 类还提供了若干操作，以便于对字符串进行处理。

<code>c_str()</code>	返回 C 风格的 <code>string</code> 类实例所包含的字符串，保证以 ‘\0’ 为结束符。
<code>data()</code>	返回 <code>string</code> 类实例所包含的字符串数据，不一定以 ‘\0’ 为结束符。

compare

与另一个 `string` 类实例或字符序列按字典序比较大小。

```
int compare(const string& str);
```

copy

将当前字符串的指定范围的字符复制到字符数组中。

```
size_t copy(char* s, size_t len, size_t pos = 0) const;
```

substr

如果指定参数，获取指定开始位置指定长度的字符，若字符串的长度不足，则获取尽可能多的字符。

若未指定起始位置参数，默认从位置 0 开始。当不指定长度参数时，一直获取到字符串末尾。

```
string substr(size_t pos = 0, size_t len = npos) const;
```

强化练习：[10361 Automatic Poetry^A](#)，[11233 Deli Deli^A](#)。

3.3 字符串库函数

为了更为方便地对字符串进行操作，C++不仅向后兼容了原有的 C 字符函数，还提供了许多有用的方法。

```
#include <cctype>

// 字符分类函数
int isalnum(int c);           // 检查是否为字母或数字字符
int isalpha(int c);           // 检查是否为字母字符
int isblank(int c)C++11;    // 检查是否为分隔字符串的空白字符 (' ', '\t')
int iscntrl(int c);           // 检查是否为控制字符
int isdigit(int c);           // 检查是否为数字字符
```

```

int isgraph(int c);           // 检查是否为具有图形输出的字符
int islower(int c);          // 检查是否为小写字符
int isprint(int c);          // 检查是否为可打印字符
int ispunct(int c);          // 检查是否为标点符号字符
int isspace(int c);          // 检查是否为空白字符 (' ', '\t', '\v', '\f', '\r', '\n')
int isupper(int c);          // 检查是否为大写字母
int isxdigit(int c);         // 检查是否为十六进制中的数字字符

// 字符转换函数
int tolower(int c);          // 将大写字母转换为小写字母
int toupper(int c);          // 将小写字母转换为大写字母

```

注意以上函数的参数及返回值均为整数类型，在传入参数时，使用 `char` 类型不会发生问题，因为在编译时会自动将 `char` 类型转换为 `int` 类型，但是对于返回值，需要自行进行类型转换，例如：

```
char c = 'a';
cout << (char)(toupper(c)) << endl;
```

强化练习：139 Telephone Tangles^C，445 Marvelous Mazes^A。

头文件`<string>`包含了若干用于在字符串与数值之间进行相互转换的函数，但它们是 C++11 标准的，在编译时需要使用支持此标准的编译器。

stoi^{C++11}

默认以十进制形式将字符串解析为 `int` 类型的整数，如果指定 `base` 为 0，而且待转换的字符串前附加了有效的数制前缀（八进制为 0，十六进制为 0x），则按照前缀指定的进制转换为十进制数。

```
int stoi(const string& str, size_t* idx = 0, int base = 10);
```

stol^{C++11}

默认以十进制形式将字符串解析为 `long int` 类型的整数。

```
long stol(const string& str, size_t* idx = 0, int base = 10);
```

stoul^{C++11}

默认以十进制形式将字符串解析为 `unsigned long int` 类型的整数。

```
unsigned long stoul(const string& str, size_t* idx = 0, int base = 10);
```

stoll^{C++11}

默认以十进制形式将字符串解析为 `long long int` 类型的整数。

```
long long stoll(const string& str, size_t* idx = 0, int base = 10);
```

stoull^{C++11}

默认以十进制形式将字符串解析为 `unsigned long long int` 类型的整数。

```
unsigned long long stoull(const string& str, size_t* idx = 0, int base = 10);
```

stof^{C++11}

默认以十进制形式将字符串解析为 `float` 类型的浮点数。

```
float stof(const string& str, size_t* idx = 0);
```

stod^{C++11}

默认以十进制形式将字符串解析为 `double` 类型的浮点数。

```
double stod(const string& str, size_t* idx = 0);
```

stold^{C++11}

默认以十进制形式将字符串解析为 long double 类型的浮点数。

```
long double stold(const string& str, size_t* idx = 0);
```

to_string^{c++11}

将数值转换为对应的字符串表示。

```
string to_string(int val);
```

以下示例 stoi 函数的使用，其他函数的使用读者可自行类推。

```
-----3.3.cpp-----//
int main(int argc, char *argv[]) {
    string numberText = "100";
    int iDecNumber = stoi(numberText);
    int iOctNumber = stoi(numberText, 0, 8);
    int iHexNumber = stoi(numberText, 0, 16);
    cout << iDecNumber << endl;
    cout << oct << showbase << iDecNumber << endl;
    cout << hex << iHexNumber << endl;
    return 0;
}
-----3.3.cpp-----//
```

输出为：

```
100
0144
0x100
```

强化练习: 123 Searching Quickly^A, 263 Number Chains^A, 509 RAID^D, 10473 Simple Base Conversion^A, 11736 Debugging RAM^D, 12085 Mobile Casanova^C。

3.4 字符串类应用

3.4.1 文本解析

在有关字符串的应用中，大多数题目以模拟（ad hoc）的形式出现，一般是先将输入中的字符串进行相应的拆分，然后按照要求对拆分得到的输入单元进行某种操作，需要应用本章介绍的字符串相关操作以及结合标准输入来完成。大致可以分为以下几种类型。

（1）计分或统计。对字符串完成解析，对得到的记录按规则进行统计，然后格式化输出。需要注意的是，string 类本身存储的是 char 类型的字符，底层是以 int 存储，而输入中可能出现 unsigned char 类型的字符，如果使用 int 存储，可能为负值，如果将字符元素作为数组下标引用会导致错误或者错误的结果，应将 char 类型的字符值加上偏移值 128 后转换为非负值再使用。

强化练习：145 Gondwanaland Telecom^B, 154 Recycling^A, 187 Transaction Processing^C, 381 Making the Grade^C, 462 Bridge Hand Evaluator^B, 538 Balancing Bank Accounts^C, 584 Bowling^B, 655 Scrabble^E, 661 Blowing Fuses^A, 933 Water Flow^E, 1368 DNA Consensus String^B, 1585 Score^A, 1586 Molar Mass^A, 10008 What's Cryptanalysis^A, 10062 Tell Me the Frequencies^A, 10126 Zipf's Law^C, 10293 Word Length and Frequency^B, 10420 List of Conquests^A, 10554 Calories from Fat^C, 10789 Prime Frequency^A, 10815 Andy's First Dictionary^A, 11062 Andy's Second Dictionary^B, 11225 Tarot Scores^C, 11279* Keyboard Comparison^D, 11340 Newspaper^A, 11530 SMS Typing^A, 11577 Letter Frequency^A, 11743 Credit Check^A, 11786 Global

Raining at Bididibus^C, 11839 Optical Reader^B, 11878 Homework Checker^A, 12412* A Typical Homework^C, 12543 Longest Word^B, 12626 I Love Pizza^A, 12696 Cabin Baggage^B, 12700 Banglawash^B, 13047 Arrows^D, 13091 No Ball^D, 13151* Rational Grading^D。

扩展练习: 207 PGA Tour Prize Money^E, 293 Bits^E, 365 Welfare Reform^E, 635 Clock Solitaire^D, 1215 String Cutting^D, 10625* GNU = GNU'sNotUnix^C。

(2) 选举计票。题目给定选举规则及各个候选人的选票, 要求按照指定的规则统计选票数量, 在统计过程中将选票最少的候选人剔除, 再次计数选票, 直到确定获胜的候选人。此类题目考察的是代码实现的细致程度和考虑问题的周密性。评判测试数据中可能会包含很多边界情形的数据, 稍不注意就会得到错误结果。

强化练习: 262 Transferable Voting^E, 349 Transferable Voting (II)^D, 10142 Australian Voting^A, 10374 Election^B。

(3) 格式化输出。此类题目有几种类型: 1) 给定若干行字符串, 要求按指定的格式输出 (或者先根据指定的条件计算最小的输出宽度或高度, 之后再予以输出)。2) 给定输出尺寸, 要求按指定规则输出字符或图案。一般来说, 此类题目所要求的输出格式都比较复杂。由于在终端上输出是一个线性的过程, 即只能按照从左至右, 从上到下的顺序进行, 不能任意选择输出位置, 这给输出带来了一定困难。一个技巧是将输出先映射到一个二维字符数组中, 然后再将其输出, 因为可以对二维字符数组进行非线性操作, 这样可以降低输出的难度。3) 在某些题目类型中, 还会涉及字符的识别, 即给定特定字符的二维字符数组表示, 在二维字符数组表示的输入中要求识别给定的特定字符。对于此种题型, 只需逐个位置比对相应的字符是否相同即可。

强化练习: 177 Paper Folding^C, 338 Long Multiplication^D, 362 18000 Seconds Remaining^B, 392 Polynomial Showdown^A, 400 Unix ls^A, 403 Postscript^C, 428 Swamp County Roofs^D, 500 Table^D, 637 Booklet Printing^A, 848 Fmt^D, 1605* Building for UNC^C, 10146 Dictionary^D, 10197 Learning Portuguese^C, 10659 Fitting Text into Slides^D, 10800 Not That Kind of Graph^B, 11074 Draw Grid^B, 11403 Binary Multiplication^D, 11965 Extra Spaces^B, 12155 ASCII Diamond^D, 12280 A Digital Satire of Digital Age^D, 12364 In Braille^D, 12482 Short Story Competition^D。

扩展练习: 370 Bingo^E, 373 Romulan Spelling^D, 396 Top Dog^E, 397 Equation Elation^C, 398 18-Wheeler Caravans (aka Semigroups)^D, 426 Fifth Bank of Swamp County^D, 645* File Mapping^D, 890 Maze (II)^E, 10017 The Never Ending Towers of Hanoi^C, 10333 The Tower of ASCII^D, 10761 Broken Keyboard^D, 10875 Big Math^D, 10894 Save Hridoy^C, 11482 Building a Triangular Museum^D。

(4) 记录排序。将输入中的字符串解析成结构体, 按照指定规则对结构体排序然后输出。

强化练习: 169 Xenosemantics^D, 230 Borrowers^C, 642 Word Amalgamation^A, 790* Head Judge Headache^D, 857 Quantiser^D, 10138 CDVII^C, 10194 Football (aka Soccer)^A, 10508 Word Morphing^B, 10698* Football Sort^D, 11056 Formula 1^B, 13293 All-Star Three-Point Contest^E。

(5) 编码或解码。此类题目要求按照指定的编码或解码规则对字符进行加密或解密, 一般结合 map 来进行操作较为方便, 因为 map 可以方便的提供查找功能而不必寻求库函数 find 所提供的顺序查找。需要注意的是要对题目描述中的编码规则和解码规则理解透彻, 严格按照规则进行编码, 同时注意边界情形的处理。

强化练习: 183 Bit Maps^C, 333 Recognizing Good ISBNs^B, 385 DNA Translation^D, 444 Encoder and Decoder^A, 449 Majoring in Scales^D, 468 Key to Success^C, 486 English-Number Translator^B, 517 Word^C,

[554 Caesar Cypher^D](#), [1339 Ancient Cipher^A](#), [10896 Known Plaintext Attack^C](#), [10921 Find the Telephone^A](#), [11220 Decoding the Message^B](#), [11223 O: Dah Dah Dah^B](#), [11278 One-Handed Typist^B](#), [11541 Decoding^A](#), [11716 Digital Fortress^A](#), [11787 Numeral Hieroglyphs^C](#), [11946 Code Number^B](#), [12515 Movie Police^D](#), [13107 Royale With Cheese^D](#), [13145 Wuymul Wixcha^D](#)。

扩展练习: 346 Getting Chorded^D, 425* Enigmatic Encryption^D, 433 Bank (Not Quite O.C.R.)^D, 613 Numbers That Count^D, 726 Decode^D, 795* Sandorf's Cipher^D, 828 Deciphering Messages^D, 856 The Vigenère Cipher^D, 1091* Barcodes^D, 11697 Playfair Cipher^D, 12134 Find the Format String^D。

(6) 指令解析。题目给定一组规则和相应的一系列指令, 要求按照指令进行模拟操作, 最后输出结果。主要考察解题者读题以及实现的细致程度。

强化练习: 101 The Blocks Problem^A, 337 Interpreting Control Sequences^B, 448 OOPS^B, 512 Spreadsheet Tracking^C, 537 Artificial Intelligence^A, 964 Custom Language^E, 10033 Interpreter^A, 10134 AutoFish^D, [12503 Robot Instructions^A](#)。

扩展练习: 328 The Finite State Text-Processing Machine^D, 330 Inventory Maintenance^D, 577 WIMP^D。

11103 WFF 'N PROOF^D (WFF 'N PROOF 游戏)

WFF 'N PROOF 是一种使用多个骰子进行的逻辑游戏, 每个骰子有六个面, 每个面有一个字母, 分别为 K, A, N, C, E, p, q, r, s, t 中的某一个。一个合式公式 (Well-Formed Formul, WFF) 是满足下列规则的字符串:

- (1) p, q, r, s 和 t 是 WFF;
- (2) 如果 w 是 WFF, Nw 也是 WFF;
- (3) 如果 w 和 x 是 WFF, Kwx , Awx , Cwx 和 Ewx 是 WFF。

其中 p, q, r, s, t 是逻辑变量, 其值可以为 0 (表示 false) 或 1 (表示 true); K, A, N, C, E 的含义为 *and*, *or*, *not*, *implies*, *equals*, 与二进制位运算的含义类似但又不完全相同。

给定一组符号的集合, 确定从中选取若干字符所能组成的最长 WFF。

输入

输入包含多组测试数据, 每组测试数据一行, 每行由 1 至 100 个字符组成, 输入以包含 “0” 的一行结束。

输出

对于输入中的每组测试数据, 输出使用输入中的字符的子集能够得到的最长 WFF。如果存在多个满足要求的 WFF, 输出其中任意一个即可, 如果不存在合法的 WFF, 则输出 “no WFF possible”。

样例输入

```
qKpNq
KKN
0
```

样例输出

```
KqNq
no WFF possible
```

分析

题目需要确定从输入字符串中取出若干字符所能构建的最长的 WFF, 并未要求取出字符的顺序必须按照输入字符串中所限定的字符顺序, 因此可以先将输入中的逻辑运算符和变量分离出来, 然后使用贪心算法构建尽可能长的 WFF。逻辑运算符分两种, 一种是一元逻辑运算符, 一种是二元逻辑运算符, 此处只有 ‘N’

是一元逻辑运算符，其他的都是二元逻辑运算符。在使用贪心法构造 WFF 时，因为一元逻辑运算符可以附加在任何合法的 WFF 之前构成新的 WFF，因此目标是尽可能多地使用二元逻辑运算符来构造 WFF，由于需要使得 WFF 尽可能地长，因此之前构造的 WFF 应该作为一个“变量”来参与新 WFF 的构造，这样可以更为有效地利用原有的变量，从而使得构造得到的 WFF 尽可能地长。给定 n 个二元运算符，最多能够结合 $n+1$ 个变量，从而构成长度最多为 $2n+1$ 的 WFF，因为将二元运算符书写在变量之前，类似于在计算四则运算表达式时将中缀表达式转换为前缀表达式，而在中缀表达式中， n 个运算符至多能够连接 $n+1$ 个运算数。

参考代码

```

bool isLogical(char c) {
    return c == 'k' || c == 'a' || c == 'n' || c == 'c' || c == 'e';
}

bool isVariable(char c) { return 'p' <= c && c <= 't'; }

int main(int argc, char *argv[]) {
    string symbol;
    while (cin >> symbol, symbol.front() != '0') {
        // 分离逻辑运算符和变量。
        vector<char> nots, logicals, variables;
        for (int i = 0; i < symbol.length(); i++) {
            char c = tolower(symbol[i]);
            if (isLogical(c)) {
                if (c == 'n') nots.push_back('N');
                else logicals.push_back(toupper(c));
            }
            if (isVariable(c)) variables.push_back(c);
        }
        // 贪心算法构建 WFF。
        string wff;
        while (variables.size() > 0) {
            // 尽可能多地使用二元逻辑运算符。
            if (wff.length()) {
                wff.insert(wff.begin(), logicals.back());
                logicals.pop_back();
            }
            wff.push_back(variables.back());
            variables.pop_back();
            if (!logicals.size()) break;
        }
        // 当 WFF 不为空时，所有一元逻辑运算符均可使用。
        if (wff.length() > 0) {
            while (nots.size() > 0) {
                wff.insert(wff.begin(), 'N');
                nots.pop_back();
            }
        }
        if (wff.length() == 0) cout << "no WFF possible\n";
        else cout << wff << '\n';
    }
    return 0;
}

```

强化练习: 175 Keywords^D, 189 Pascal Program Lengths^D, 309 FORCAL^D, 502 DEL Command^D, 912 Live From Mars^D, 1061 Consanguine Calculations^D, 1200 A DP Problem^D, 10602 Editor Nottoobad^B, 10848 Make Palindrome Checker^D, 10903 Rock-Paper-Scissors Tournament^B, 11357 Ensuring Truth^D, 12414 Calculating Yuan Fen^D。

3.4.2 语法分析

134 Loglan-A Logical Language^C (Loglan 逻辑语言)

Loglan 是一种人工设计的可发音语言, 主要用来对语法学的一些基础问题进行验证。在 Loglan 中, 句子由一系列的词和名称构成, 之间用空格分开, 以符号 ‘.’ 作为句子的结束符。Loglan 中的词都以元音字母结尾; 名称 (names) 以辅音字母结尾, 由其他语言衍生而来。词分为两类, 一种是小词, 用来指定句子的结构, 另一种是谓词 (predicates), 具有形如 CCVCV 或 CVCCV 的结构, 其中 C 代表某个辅音字母 (consonant), V 代表某个元音字母 (vowel)。句法规则如下:

```

A          => a | e | i | o | u
MOD        => ga | ge | gi | go | gu
BA         => ba | be | bi | bo | bu
DA         => da | de | di | do | du
LA         => la | le | li | lo | lu
NAM        => {all names}
PREDA      => {all predicates}
<sentence> => <statement> | <predclaim>
<predclaim> => <predname> BA <preds> | DA <preds>
<preds>    => <predstring> | <preds> A <predstring>
<predname>  => LA <predstring> | NAM
<predstring> => PREDA | <predstring> PREDA
<statement>  => <predname> <verbpred> <predname> | <predname> <verbpred>
<verbpred>   => MOD <predstring>

```

输入与输出

输入中给出了一些 Loglan 句子, 每个句子均从新的一行开始, 以 ‘.’ 结尾。句子的单词可能不全在一行上, 单词之间的空格也可能不止一个, 输入最后以一个 ‘#’ 结束。你可以假设所有单词的格式都是正确的。

判断给定的句子是否符合 Loglan 的句法规则, 如果符合则输出 “Good”, 否则输出 “Bad!”。

样例输入

```

la mutce bunbo mrenu bi dicta.
la fumna bi le mrenu.
djan ga vedma le negro kepti.
#

```

样例输出

```

Good
Bad!
Good

```

分析

本题实质上是要求编写一个简化版的语法解析器, 需要了解基本的编译原理知识才能顺利解决。语法分析 (syntax analysis) 是根据语言所指定的语法定义 (syntax definition) 进行输入的解析, 语法定义一般使用下列形式给出:

$$MOD \rightarrow ga | ge | gi | go | gu$$

箭头本身读作 “可以具有形式”, 整个式子称为产生式 (production), 箭头右侧的 ga、ge、gi、go、gu 称

为标记 (token)，箭头左侧为非终结符 (nonterminals)，可以认为非终结符代表了一个标记序列 (可能包含其他的非终结符)。非终结符为产生式的左端 (left side)，箭头、标记和/或非终结符序列称为产生式的右端 (right side)。语法解析的最终结果就是要根据产生式得到一棵语法解析树，如果能够得到不止一棵语法解析树，表明语法定义存在二义性，进行解析时会产生矛盾。语法定义不能循环定义，如两个非终结符定义为

$$S \rightarrow L, L \rightarrow S$$

即构成循环定义，进行解析时会陷入无限循环。但是语法规则可以存在递归定义，如本题中的语法定义

$$<\text{predstring}> \rightarrow \text{PREDA} | <\text{predstring}> \text{PREDA}$$

不同于循环定义，递归定义有出口，而循环定义无出口，因此递归定义可以正确解析。

常用的语法解析方法有自顶向下解析 (top-down parsing)、自底向上解析 (bottom-up parsing) 等^[25]。自顶向下的解析可以视为从根结点开始将输入字符串构建为一棵解析树，其中最为常见的形式是递归下降解析 (recursive-descent parsing)。对于本题来说，要求判断给定的字符串能否解析为`<sentence>`，根据给定的语法定义，如果使用递归下降解析，相当于使用前序遍历的方式构造类似于图 3-1 的语法解析树。

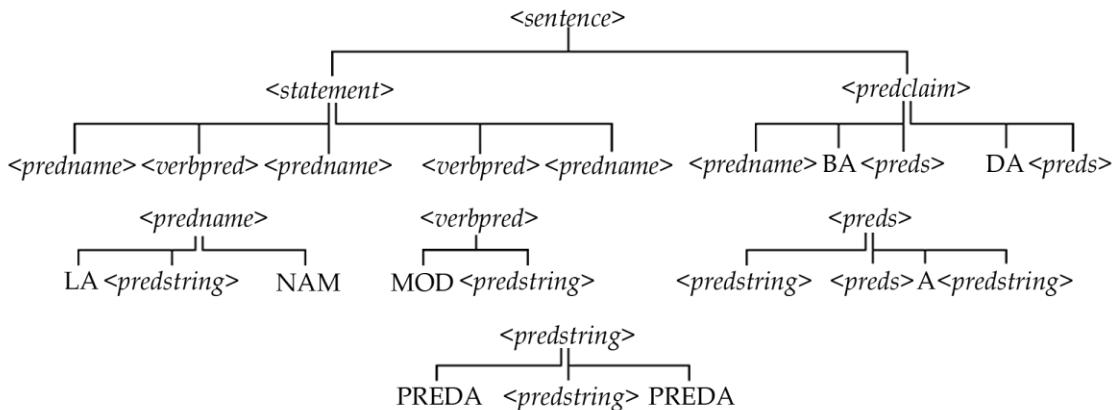


图 3-1 语法解析树（未将所有语法成分表示在一棵树中，而是表示为森林形式）

以样例输入 “`la mutce bunbo mrenu bi dicta.`” 为例，使用递归下降的解析方法，需要设立一个指针 (索引)，表示当前所扫描的位置，接着从根结点`<sentence>`开始，检查其是否符合`<statement>`或`<predclaim>`的语法，而`<statement>`可由两种形式构成，先解析第一种形式，即`<predname><verbpred><predname>`的形式，而`<predname>`可由`LA<predstring>`或`NAM` 构成，将指针移动到第一个单词进行检查，可以发现其匹配`LA`，那么需要判断后续单词是否匹配`<predstring>`的定义，检查可知后续三个单词“`mutce`”、“`bunbo`”、“`mrenu`” 均符合`PREDA` 的语法定义，这三个单词构成`<predstring>`，与`LA` 共同构成`<predname>`，此时指针指向第五个单词，但是第五个单词“`bi`”不匹配`<verbpred>`的定义，因此不能将语句解析为`<statement>`的第一种形式，此时需要将指针回退到第一个单词，转而使用`<verbpred><predname>`的形式进行匹配，可以发现输入同样不符合。使用类似的方式继续匹配，最终可知输入符合`<sentence>`的第二种形式`<predclaim>`，那么可以判定样例输入是符合要求的`<sentence>`。在具体实现时，需要为每个非终结符编写对应的方法进行检查，如果一个

非终结符有多种产生式，则需要判断多种情形^I。

在自底向上解析中，常见的形式是 LR(k) 解析，即从左至右扫描和最右推导 (left-to-right scan and rightmost derivation, LR) 方式解析，其中 k 表示向前查看 (lookahead) 的符号 (symbol) 个数，用以作出解析选择。LR 解析按照移进一归约 (shift and reduce) 的方式进行。简单地说，就是将各个语法成分按照产生式予以归约合并，如果是一个合法的 Loglan 语句，最后会归约为唯一的一个语法成分。对于本题来说，处理步骤是先将输入处理成基本的语法成分，根据 FIRST 和 FOLLOW 集合制定合并规则，例如，BA 加上前缀 $<predname>$ 及后缀 $<preds>$ 可将其归约为 $<predclaim>$ 。使用 LR 解析会使得编码更为简洁，其关键是分析语法成分，构建语法成分合并的法则，如果规则制定不当，会导致解析过程陷入无限循环或者得出错误的结果。

需要注意的是，在 UVa OJ 上的评测数据包含的单词并不是如题目描述所说都是正确的，而是包含不符合格式的单词，所以进行单词的格式检查是必需的。

参考代码

```
const int GROUPS = 14;
const int NONE = -1, A = 0, MOD = 1, BA = 2, DA = 3, LA = 4, NAM = 5,
PREDA = 6, PREDSTRING = 7, PREDNAME = 8, PREDs = 9, VERBPRED = 10,
PREDVERB = 11, PREDCLAIM = 12, STATEMENT = 13, SENTENCE = 14;

vector<int> S; vector<string> W;

int T[GROUPS][4] = {
    {PREDA, NONE, PREDA, PREDA}, {PREDA, NONE, NONE, PREDSTRING},
    {NAM, NONE, NONE, PREDNAME}, {LA, NONE, PREDSTRING, PREDNAME},
    {MOD, NONE, PREDSTRING, VERBPRED}, {A, PREDSTRING, PREDSTRING, PREDSTRING},
    {PREDSTRING, NONE, NONE, PREDs}, {DA, NONE, PREDs, PREDCLAIM},
    {BA, PREDNAME, PREDs, PREDCLAIM}, {VERBPRED, PREDNAME, NONE, PREDVERB},
    {PREDVERB, NONE, PREDNAME, STATEMENT}, {PREDVERB, NONE, NONE, STATEMENT},
    {STATEMENT, NONE, NONE, SENTENCE}, {PREDCLAIM, NONE, NONE, SENTENCE}
};

// 将输入解析成基本语法成分。
int getSymbol(string word) {
    string vowel = "aeiou";
    if (word.length() == 1 && vowel.find(word[0]) != wordnpos) return A;
    if (word.length() == 2 && vowel.find(word[1]) != wordnpos) {
        if (word[0] == 'g') return MOD;
        if (word[0] == 'b') return BA;
        if (word[0] == 'd') return DA;
        if (word[0] == 'l') return LA;
        return NONE;
    }
    if (vowel.find(word.back()) == wordnpos) return NAM;
    if (word.length() == 5) {
        int bitOr = 0;
```

^I 由于本书篇幅所限，关于使用递归下降的解析方法进行解题的代码，读者可参阅：

<https://github.com/metaphysis/Code/blob/master/UVa%20Online%20Judge/volume001/134%20Loglan-A%20Logical%20Language/program.cpp>。

```

        for (int i = 0; i < 5; i++)
            bitOr |= ((vowel.find(word[i]) != wordnpos ? 1 : 0) << (4 - i));
        if (bitOr == 5 || bitOr == 9) return PREDA;
    }
    return NONE;
}

// 将语法成分转换为符号。
bool parse() {
    S.clear();
    for (int i = 0; i < W.size(); i++) {
        int symbol = getSymbol(W[i]);
        if (symbol == NONE) return false;
        else S.push_back(symbol);
    }
    return true;
}

// 根据语法规则不断合并语法成分，检查最后是否可将给定输入合并为单一的语法成分。
bool good() {
    if (!parse()) return false;
    for (int i = 0; i < GROUPS; i++) {
        for (int j = 0; j < S.size(); ) {
            // 检查是否符合前缀和后缀限定。
            if ((S[j] != T[i][0]) || (~T[i][1] && (!j || S[j - 1] != T[i][1])) ||
                (~T[i][2] && (j == (S.size() - 1) || S[j + 1] != T[i][2]))) {
                j++;
                continue;
            }
            // 合并。
            j = ~T[i][1] ? S.erase(S.begin() + j - 1) - S.begin() : j;
            j = ~T[i][2] ? S.erase(S.begin() + j + 1) - S.begin() - 1 : j;
            S[j] = T[i][3];
        }
        return (S.size() == 1 && S.front() == SENTENCE);
    }
}

int main(int argc, char *argv[]) {
    string line, word;
    while (getline(cin, line), line != "#") {
        string temp(line);
        if (line.find('.') != string::npos) temp = temp.substr(0, temp.find('.'));
        istringstream iss(temp);
        while (iss >> word) W.push_back(word);
        if (line.find('.') != string::npos) {
            cout << (good() ? "Good" : "Bad!") << '\n';
            W.clear();
        }
    }
    return 0;
}

```

强化练习: 171 Car Trialling^D, 271 Simply Syntax^A, 342 HTML Syntax Checking^D, 384 Slurpys^B, 620 Cellular Structure^B, 743* The MTM Machine^C, 10058 Jimmy's Riddles^C, 11203 Can You Decide It for ME^C。

扩展练习: 174 Strategy^D, 10854 Number of Paths^D, 10906 Strange Integration^D, 11070 The Good Old

Times^D。

3.4.3 KMP 匹配算法

给定非空字符串 s 和 p , 其长度分别为 n 和 m , 为了便于讨论, 将 s 称为主串, p 称为模式串。若需要查找 p 是否在 s 中存在, 朴素的方法是将 p 的第一个字符与 s 的某个字符对齐, 检查对应的字符是否相同, 若从主串的某个位置开始, 两者字符全部相同, 则发现了匹配。

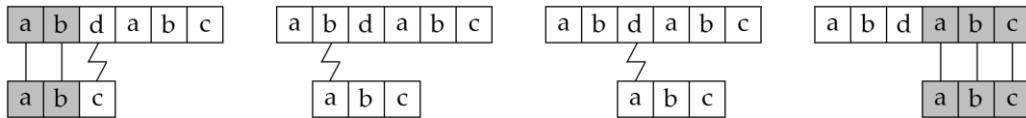


图 3-2 朴素的字符串匹配过程。 $s = "abdabc"$, $p = "abc"$ 。阴影覆盖的方格为匹配成功的字符, 中间使用直线连接。使用折线连接的方格为最先匹配不成功的字符。观察朴素字符串的匹配过程, 不难发现如下规律: 匹配每失败一次, 模式串就向右“滑动”一个字符, 直到匹配成功或到达主串的末尾

```
-----3.4.3.1.cpp-----//  
// 使用暴力匹配的方法检查字符串 p 是否为字符串 s 的子串。  
bool match(const string &s, const string &p) {  
    for (int si = 0; si < s.length(); si++) {  
        int i = si, j = 0;  
        while (i < s.length() && j < p.length() && s[i] == p[j]) i++, j++;  
        if (j >= p.length()) return true;  
    }  
    return false;  
}  
-----3.4.3.1.cpp-----//
```

在 C++ 的 `string` 类中提供了 `find` 方法，该方法的内部实现中所使用的即是朴素匹配算法。当题目中要求的字符串匹配数量规模较小且为“线性”字符串时（即给定的字符串是连续的而不是位于矩阵中），使用 `find` 方法进行匹配较为简便。

强化练习: [422 Word-Search Wonder^B](#), [736* Lost in Space^D](#), [850 Crypt Kicker II^A](#), [886* Named Extension Dialing^D](#), [1588 Kickdown^B](#), [10010 Where's Waldorf^A](#), [10132 File Fragmentation^B](#), [10188 Automated Judge Script^A](#), [10252 Common Permutation^A](#), [10298 Power Strings^A](#), [11048 Automatic Correction of Misspellings^C](#), [11362 Phone List^A](#), [11452 Dancing the Cheeky-Cheeky^C](#), [11576 Scrolling Sign^C](#), [12478 Hardest Problem Ever \(Easy\)^A](#)。

扩展练习: 292 Presentation Error^E, 475* Wild Thing^D, 581* Word Search Wonder^E。

朴素的匹配算法之所以在某些情况下效率较低，原因在于每次“失配”后都从模式串的起始处开始重新进行匹配，将之前通过匹配所得到的“额外信息”完全予以丢弃，而这些“额外信息”是能够供后续匹配使用的。如果善加利用这些“额外信息”，能够有效地提高后续匹配的效率。

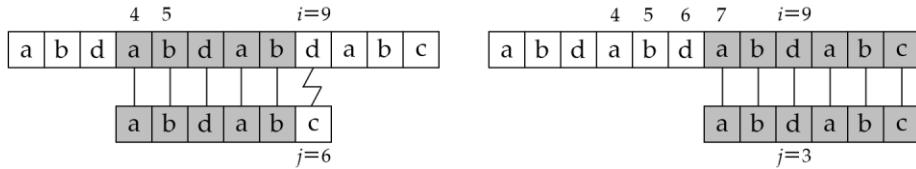


图 3-3 失配时“额外信息”的利用。主串 $s = "abdabdbabdabc"$, 模式串 $p = "abdabc"$, 在 $i=9, j=6$ 处失配。

如果是朴素的匹配算法, 下一次应该进行 $i=5, j=1$ 的匹配, 但是观察模式串 p , 在失配处 $j=6$ 的字符 ‘c’ 之前有两个字符 “ab” 与模式串的起始两个字符相同, 而且已经匹配, 那么可以将模式串一次性向右“滑动” 3 个字符, 跳过 $i=5, j=1, i=6, j=1, i=7, j=1$ 的匹配, 直接开始 $i=9, j=3$ 的匹配。也就是说, 当 $j=6$ 失配时, 可以将模式串位于 $j=3$ 的字符与主串的失配字符对齐, 继续进行匹配。因此 $j=3$ 是 $j=6$ 失配时的“跳转”位置, 亦即 $j=6$ 失配时, 模式串应该向右“滑动” 3 个字符, 从失配处继续进行匹配

KMP 匹配算法由 Knuth、Morris 和 Pratt 各自独立发现, 三者合作公布了工作成果^[26]。此算法基于字符串匹配自动机, 但是不需要计算变迁函数, 只是用到了“失配”辅助函数 $fail[1, m]$, 它可以在 $\Theta(m)$ 的时间复杂度内根据模式预先计算得到, 算法总的匹配时间复杂度为 $\Theta(n)$ 。

假设主串为 $s_1s_2\cdots s_n$, 模式串为 $p_1p_2\cdots p_m$, 为了提高匹配的效率, 需要解决以下问题: 当匹配过程中产生“失配”时 (即 $s_i \neq p_j$), 模式串向右“滑动”的最远距离。也就是说, 当主串中第 i 个字符与模式串中第 j 个字符“失配”时, 主串中第 i 个字符应与模式串中哪个字符进行再次比较。假设此时应与模式串中第 k ($k < j$) 个字符继续比较, 此时模式串的前 $(k-1)$ 个字符与主串第 i 个字符之前的 $(k-1)$ 个字符相同, 即模式串中前 $k-1$ 个字符构成的子串必须满足

$$p_1p_2\cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1} \quad (3.1)$$

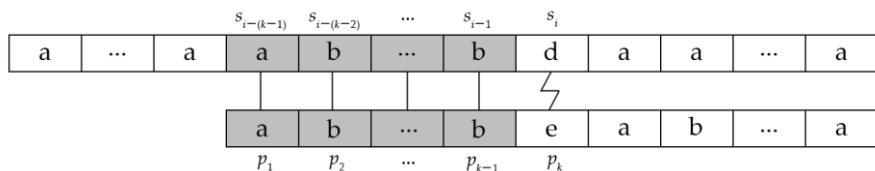
且对于任意的 k' ($k' < k < j$), k' 均不满足关系式 (3.1), 亦即 k 是满足关系式 (3.1) 的最大字符序号。而当前已经得到的“部分匹配”结果是

$$p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1} \quad (3.2)$$

由式 (3.1) 和式 (3.2) 可以得到

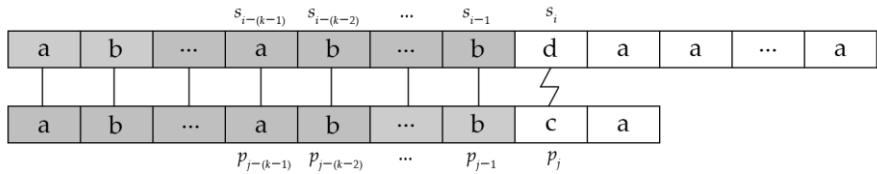
$$p_1p_2\cdots p_{k-1} = p_{j-(k-1)}p_{j-(k-2)}\cdots p_{j-1} \quad (3.3)$$

那么, 假设模式串中存在满足式 (3.3) 的两个子串, 则在匹配过程中, 当主串的第 i 个字符与模式串的第 j 个字符“失配”时, 仅需将模式串向右滑动至模式串的第 k 个字符与主串的第 i 个字符对齐, 此时模式串中初始的 $(k-1)$ 个字符所构成的子串 $p_1p_2\cdots p_{k-1}$ 必定与主串中第 i 个字符之前长度为 $(k-1)$ 的子串 $s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$ 相等, 由此可知, 匹配仅需从模式串的第 k 个字符与主串的第 i 个字符开始, 继续进行比较即可。

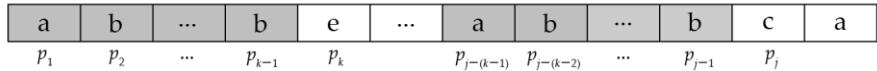


(a) 在失配后, 将模式串向右“滑动”, 假设主串的第 i 个字符与模式串的第 k 个字符开始匹配,

$$\text{易知有: } p_1p_2\cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)}\cdots s_{i-1}$$



(b) 在失配进行“滑动”之前, 根据部分匹配的结果, 有: $p_{j-(k-1)}p_{j-(k-2)} \cdots p_{j-1} = s_{i-(k-1)}s_{i-(k-2)} \cdots s_{i-1}$



(c) 结合失配后向右“滑动”进行匹配的情况及部分匹配的结果, 有: $p_1 p_2 \cdots p_{k-1} = p_{i-(k-1)} p_{i-(k-2)} \cdots p_{i-1}$

图 3—4 模式串中子串的相互关系

若令 $fail[j]=k$ ，则 $fail[j]$ 表示当模式串的第 j 个字符与主串中相应字符“失配”时，在模式串中重新和主串中该字符进行比较的字符的位置，由前述讨论可得到模式串 $fail$ 函数的定义

$$fail[j] = \begin{cases} 0, & \text{当 } j = 1 \text{ 时} \\ \max\{k \mid 1 < k < j, p_1 p_2 \cdots p_{k-1} = p_{j-(k-1)} p_{j-(k-2)} \cdots p_{j-1}\}, & \text{当此集合不为空时} \\ 1, & \text{其他情况} \end{cases}$$

由上述定义可以得到模式串 $p = \text{"abcabcdabcde"}$ 的 fail 函数值：

j	1	2	3	4	5	6	7	8	9	10	11	12
p_j	a	b	c	a	b	c	d	a	b	c	d	e
$fail[j]$	0	1	1	1	2	3	4	1	2	3	4	1

在确定模式串 p 的 $fail$ 函数之后，匹配可按照以下步骤进行：假设以指针 i 和 j 分别指示主串 s 和模式串 p 中当前比较的字符，令 i 和 j 的初值均为 1。若在匹配过程中 $s_i=p_j$ ，则 i 和 j 分别自增 1；否则， i 不变，而 j “回退”到 $fail[j]$ 的位置再比较。若相等，则指针各自增 1，否则 j 再“回退”到下一个 $fail$ 值的位置。依此类推，直到出现以下两种情形之一：一种是 j 回退到某个 $fail$ 值 ($fail[fail[\cdots fail[j] \cdots]]$) 时字符比较相等，则指针各自增 1，继续进行匹配；另一种是 j 回退到值为零（即模式的第一个字符“失配”），则此时需将模式串向右滑动一个位置，即从主串的下一个字符 s_{i+1} 开始，与模式串重新开始匹配。

```
//++++++++++++++++++++++++++++++++++++++3.4.3.2.cpp+++++++
const int MAXN = 1010;

int fail[MAXN] = {0};

// 根据失配函数进行匹配，找到第一个匹配即返回。
bool kmp(string &s, string &p) {
    int i = 1, j = 1;
    while (i < s.length()) {
        // j 为 0 表示失配指针已经跳转到模式串的起始位置；s[i] == s[j] 表示主串和模式串
        // 的相应字符匹配。在此情形下，均需将主串中下一个字符与模式串中的相应字符进行比较。
        if (j == 0 || s[i] == p[j]) i++, j++;
        // 失配指针未跳转到模式串的起始位置或者当前的字符不匹配，失配指针发生跳转。
    }
}
```

```

        else j = fail[j];
        // 如果从 1 开始计数, j 要大于模式串的长度才表明找到一个匹配。
        if (j > p.length()) return true;
    }
    return false;
}

// 根据失配函数寻找所有匹配并输出匹配的起始位置。
void kmp(string &s, string &p) {
    int i = 1, j = 1;
    while (i < s.length()) {
        if (j == 0 || s[i] == p[j]) i++, j++;
        else j = fail[j];
        if (j > p.length()) {
            cout << i - j + 1 << '\n';
            j = fail[j];
        }
    }
}

```

KMP 算法是在已知模式串的 *fail* 函数值的基础上执行的, 那么, 如何求得模式串的 *fail* 函数值呢? 由于 *fail* 函数值仅取决于模式串本身而与需要匹配的主串无关, 因此可以从 *fail* 函数的定义出发, 使用递推的方法求得其值。

由定义可知, 当 $j=1$ 时, 有

$$fail[1] = 0$$

当 $j > 1$ 时, 不妨令 $fail[j] = k$, 这表明在模式串中

$$p_1 p_2 \cdots p_{k-1} = p_{j-(k-1)} p_{j-(k-2)} \cdots p_{j-1} \quad (3.4)$$

其中 k 为满足 $1 < k < j$ 的某个值, 并且不存在 $k' > k$ 满足等式 (3.4)。此时可能有两种情形:

(1) 若 $p_k = p_j$, 则表明在模式串中, 存在

$$p_1 p_2 \cdots p_k = p_{j-(k-1)} p_{j-(k-2)} \cdots p_j \quad (3.5)$$

并且不存在 $k' > k$ 满足等式 (3.5), 即有

$$fail[j+1] = k+1 = fail[j] + 1$$

a	b	a	b	a	f	a	b	a	b	a	g
p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}

图 3—5 第 (1) 种情形的示例。模式串 $p = "ababafababag"$, 因为 $p_1 p_2 p_3 p_4 = p_7 p_8 p_9 p_{10}$, 易知 $fail[11] = 5$, 当计算 $fail[12]$ 时, 由于 $p_5 = p_{11}$, 故有 $fail[12] = 5 + 1 = fail[11] + 1 = 6$

(2) 若 $p_k \neq p_j$, 则表明在模式串中

$$p_1 p_2 \cdots p_k \neq p_{j-(k-1)} p_{j-(k-2)} \cdots p_j \quad (3.6)$$

此时可把求 *fail* 函数值的问题看成是一个模式匹配的问题——整个模式串既是主串又是模式串。在当前的匹配过程中, 已有 $p_{j-(k-1)} = p_1$, $p_{j-(k-2)} = p_2$, ..., $p_{j-1} = p_{k-1}$, 则当 $p_k \neq p_j$ 时, 应将模式串向右滑动至以模式串中的第 $fail[k]$ 个字符与主串中的第 j 个字符相比较。令 $fail[k] = k'$, 若有 $p_k = p_{j'}$, 表明在模式串中第 $(j+1)$ 个字符之前存在一个长度为 k' (即 $fail[k]$) 的最长子串, 它和模式串中从首字符起长度为 k' 的子串相等, 即

$$p_1 p_2 \cdots p_{k'} = p_{j-(k'-1)} p_{j-(k'-2)} \cdots p_{j'}, \quad 1 < k' < k < j \quad (3.7)$$

可得

$$fail[j+1] = k' + 1 = fail[fail[j]] + 1$$

a	b	a	b	b	f	a	b	a	b	a	g
p_1	p_2	p_3	p_4	p_5	p_6	p_7	p_8	p_9	p_{10}	p_{11}	p_{12}

图3-6 第(2)种情形的示例。模式串 $p = "ababbfababag"$, 由于 $p_1p_2 = p_3p_4$, 易知 $fail[5] = 3$, 由于 $p_1p_2p_3p_4 = p_7p_8p_9p_{10}$, 易知 $fail[11] = 5$ 。当计算 $fail[12]$ 时, 由于 $p_5 \neq p_{11}$, 但 $p_1p_2p_3 = p_9p_{10}p_{11}$, $p_1p_2 = p_3p_4$, 故有 $fail[12] = fail[fail[11]] + 1 = fail[5] + 1 = 3 + 1 = 4$

同理, 若 $p_k \neq p_j$, 则将模式串本身继续向右滑动, 使得模式串中第 $fail[k']$ 个字符与 p_j 对齐, 反复进行此操作, 直至 p_i 和模式串中某个字符匹配成功或者不存在任何 k' ($1 < k' < j$) 满足关系式 (3.7), 若为后者, 则有

$$fail[j+1] = 1$$

// 根据定义得到 fail 函数值, 可以看成模式串和自身进行匹配。

```
void getFail(string &p) {
    fail[1] = 0;
    int i = 1, j = 0;
    while (i < p.length()) {
        if (j == 0 || p[i] == p[j]) i++, j++, fail[i] = j;
        else j = fail[j];
    }
}
```

还可以对上述求 $fail$ 函数值的过程予以进一步的优化。例如, 给定主串 $s = "aaabaaaab"$ 和模式串 $p = "aaaab"$, 对两者进行匹配, 当 $i=4, j=4$ 时, $s[4] \neq p[4]$ ($s[4] = 'b'$, $p[4] = 'a'$), 根据 $fail[j]$ 的指示还需要进行 $i=4, j=3$, $i=4, j=2$, $i=4, j=1$ 这三次比较, 而模式串中第 1、2、3 个字符和第 4 个字符都相等, 实际上已不需要再和主串中第 4 个字符相比较, 而是可以将模式串一次性向右滑动 4 个字符的位置, 直接进行 $i=5, j=1$ 时的比较。这就是说, 若按上述定义得到 $fail[j] = k$, 而模式中 $p_i = p_k$, 则当主串中字符 s_i 和 p_j 比较不等时, 不需要再和 p_j 进行比较而直接与 $p_{fail[k]}$ 进行比较, 亦即此时 $fail[j]$ 与 $fail[k]$ 应该相同^I。

^I 还可以进行更为“激进”的优化。如果后续发现 $p_i = p_{fail[k]}$, 则应将 p_i 与 $p_{fail[fail[k]]}$ 进行比较, 依此类推, 直到出现以下两种情形之一: 令 $k' = fail[fail[\dots fail[j]]]$, 第一种情形是 $fail$ 函数“回退”到某个值使得 $p_i \neq p_{k'}$, 此时 $fail[j] = k'$; 第二种情形是 $fail$ 函数不断“回退”且 $p_i = p_{k'}$, 最终使得 $k' = 0$, 此时 s_i 应与模式串的首字符进行比较。一般情况下, 给定的模式串可能最多需要一次“回退”即可达到第一种情形的状态, 使得上述“激进”的优化效果不明显。以下是使用“激进”优化从而消除了所有无效跳转的失配函数:

```
void getFail(string &p) {
    fail[1] = 0;
    int i = 1, j = 0;
    while (i < p.length()) {
        if (j == 0 || p[i] == p[j]) {
            i++, j++;
            int k = j;
            while (k && p[i] == p[k]) k = fail[k];
            fail[i] = k;
        } else j = fail[j];
    }
}
```

需要注意, 经过优化后的失配函数值所对应的意义可能已经改变, 虽然可以应用于匹配, 但是应用于其他方面可能不再正确, 下面举例说明。定义字符串的“镶边”(border) 为原字符串的一个子串, 该子串既是原字符串的前缀, (转到下一页)

```

// 优化的 fail 函数值生成。
void getFail(string &p) {
    fail[1] = 0;
    int i = 1, j = 0;
    while (i < p.length()) {
        if (j == 0 || p[i] == p[j]) {
            i++, j++;
            if (p[i] != p[j]) fail[i] = j;
            else fail[i] = fail[j];
        } else j = fail[j];
    }
}

```

经过上述优化后，模式串“abcabcdabcde”的 fail' 函数值为：

j	1	2	3	4	5	6	7	8	9	10	11	12
p_j	a	b	c	a	b	c	d	a	b	c	d	e
$fail[j]$	0	1	1	1	2	3	4	1	2	3	4	1
$fail'[j]$	0	1	1	0	1	1	4	0	1	1	4	1

在上述讨论中，为了方便，字符串从 1 开始计数，而在使用 C++ 实现时，字符串一般都是从 0 开始计数。虽然可以为字符串在起始位置增加一个“哨兵”字符以使得原始字符串能够从 1 开始计数，但是这样的做法并不必要，可以调整实现使得无需进行此操作。以下给出一种参考实现，在这种实现中，主串和模式均从 0 开始计数。

```

const int MAXN = 1010;

int fail[MAXN] = {};

void getFail(string &p) {
    fail[0] = -1;
    int i = 0, j = -1;
    while (i < p.length() - 1) {
        if (j == -1 || p[i] == p[j]) {
            i++, j++;
            int k = j;
            while (k != -1 && p[i] == p[k]) k = fail[k];
            fail[i] = k;
        } else j = fail[j];
    }
}

bool kmp(string &s, string &p) {

```

同时也是其后缀，即镶边是字符串的公共前后缀。例如，给定字符串 $s = “abcdeabcd”$ ，则子串“abcd”是 s 的镶边，因为“abcd”既是 s 的前缀，也是 s 的后缀。需要注意，一般不将 s 本身视为 s 的一个镶边。进一步定义“真镶边”，真镶边为异于原字符串的镶边，即字符串本身不是自身的真镶边。将字符串的所有真镶边罗列出来，其中具有最大长度的真镶边称为最长真镶边。可以验证，对于原字符串的任意一个真前缀 $s[0..i]$ 来说， $s[0..i]$ 的最长真镶边长度恰为第 $(i+1)$ 个字符的未经优化的失配函数值（实际上，KMP 算法正是通过最长真镶边来定义失配函数的）。如果对失配函数进行优化，则失配函数值不再与最长真镶边的长度对应。

```

int i = 0, j = 0;
while (i < s.length()) {
    if (j == -1 || s[i] == p[j]) i++, j++;
    else j = fail[j];
    // 当 j 从 0 开始计数时, 只要 j 等于模式串的长度就表明找到了匹配。
    if (j == p.length()) return true;
}
return false;
}
//+++++++++3.4.3.2.cpp+++++

```

强化练习: 10679* I Love Strings^A。

3.4.4 扩展 KMP 匹配算法

给定源字符串 S 和目标字符串 T , S 的长度为 n , T 的长度为 m , 要求确定 S 的所有后缀与 T 的最长公共前缀。朴素的方法是将 S 的每个后缀逐一与 T 进行匹配以确定最长公共前缀, 此种方法的时间复杂度为 $O(nm)$, 显然效率较低。更为高效的是应用一种称之为扩展 KMP 匹配算法的方法来解决这个问题, 该算法的时间复杂度为 $O(n+m)$ 。

从 0 开始计数字符, 令 $extend[i]$ 表示 $S[i, n-1]$ 与 T 的最长公共前缀长度 ($i \leq n-1$), $next[j]$ 表示 $T[j, m-1]$ 与 T 的最长公共前缀长度 ($j \leq m-1$)。假设当前已经确定了 $extend[0], extend[1], \dots, extend[k-1]$ 的值, 现在需要确定 $extend[k]$ 的值, 利用动态规划的思维, 我们来看看如何利用 $extend[0]$ 至 $extend[k-1]$ 的值来提高计算 $extend[k]$ 的值的效率。

如图 3-7 所示, 由于在从左至右计算的过程中, $extend[0]$ 至 $extend[k-1]$ 的值已经确定, 假设在这个匹配过程中所能到达 S 的最右侧字符的位置为 p_r , 即定义

$$p_r = \max\{i + extend[i] - 1, 0 \leq i \leq k-1\}$$

并假设对应 p_r 的匹配起始位置为 p_l , 根据 $extend$ 数组的定义, 可以得到

$$S[p_l, p_r] = T[0, p_r - p_l]$$

从而有

$$S[k, p_r] = T[k - p_l, p_r - p_l]$$

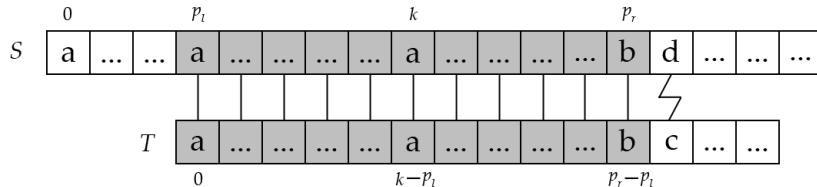


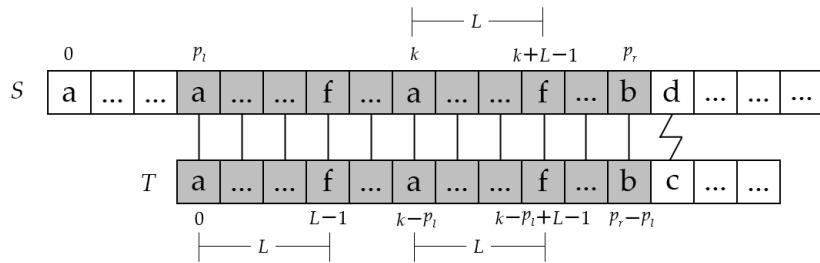
图 3-7 p_r 为之前确定 $extend[0] \sim extend[k-1]$ 的值进行匹配的过程中能够到达的最右匹配位置, p_l 为 p_r 对应的起始位置, 不难得出: $0 \leq p_l \leq k \leq p_r$

令 $L = next[k - p_l]$, 即 L 定义为字符串 T 从位置 $k - p_l$ 开始的后缀与 T 的最长公共前缀长度。区分以下三种情况分别处理:

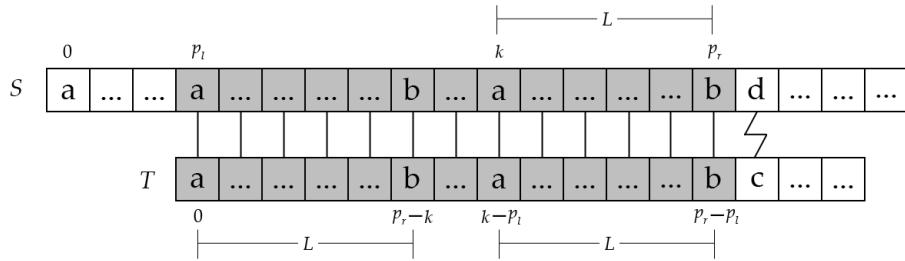
(1) $k + L < p_r$ 。如图 3-8 所示, 有

$$S[k, k + L - 1] = T[k - p_l, k - p_l + L - 1] = T[0, L - 1]$$

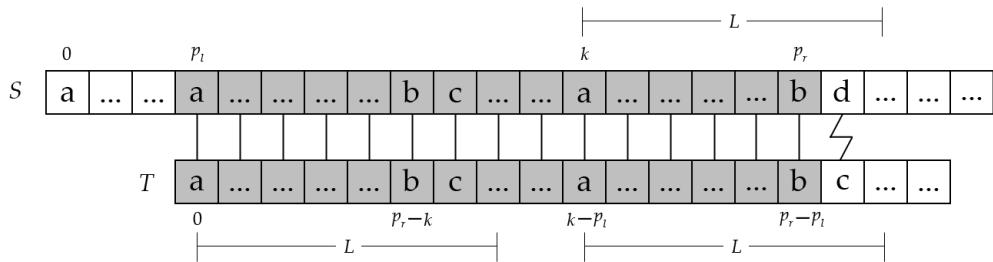
则此时根据 $extend$ 数组的定义即可得 $extend[k] = L = next[k - p_l]$ 。

图 3-8 令 $L = \text{next}[k-p_l]$, $k+L < p_r$ 的情形, 此时 $\text{extend}[k] = L = \text{next}[k-p_l]$

(2) $k+L=p_r$ 。如图 3-9 所示, 此时有 $S[p_r+1] \neq T[p_r-p_l+1]$ 且 $T[p_r-p_l+1] \neq T[p_r-k+1]$, 但是 $S[p_r+1]$ 有可能等于 $T[p_r-k+1]$, 因此可以“跳过”已匹配的部分, 直接从 $S[p_r+1]$ 和 $T[p_r-k+1]$ 开始往后逐个字符进行匹配, 直到发生失配为止, 当匹配完成后, 如果得到的 $k+\text{extend}[k]$ 大于原有的 p_r , 则需要更新 p_r 和 p_l 。

图 3-9 令 $L = \text{next}[k-p_l]$, $k+L = p_r$ 的情形, 此时可以从 $S[p_r+1]$ 和 $T[p_r-k+1]$ 开始往后匹配

(3) $k+L > p_r$ 。如果 3-10 所示, 此时 $S[k+1, p_r]$ 与 $T[k-p_l, p_r-p_l]$ 相同, 注意到 $S[p_r+1] \neq T[p_r-p_l+1]$ 且 $T[p_r-p_l+1] = T[p_r-k+1]$, 则 $S[p_r+1] \neq T[p_r-k+1]$, 所以不再需要继续对后续的字符进行匹配, 而是可以直接断定 $\text{extend}[k] = p_r - k + 1$ 。

图 3-10 令 $L = \text{next}[k-p_l]$, $k+L > p_r$ 的情形, 此时 $\text{extend}[k] = p_r - k + 1$

根据 extend 数组和 next 数组的定义, 不难看出, 计算 $\text{next}[i]$ 的过程和计算 $\text{extend}[i]$ 的过程实际上是相同的——此时源字符串为 T , 目标字符串也为 T 。根据前述介绍的计算 $\text{extend}[i]$ 的方式计算 $\text{next}[i]$, 等同于对源字符串为 T 目标字符串也为 T 的情形执行一次扩展 KMP 匹配算法。

通过前述的算法介绍可知, 对于第一种和第三种情形, 无需进行任何匹配即可计算得到 $\text{extend}[i]$; 对于

第二种情形，是从尚未被匹配的位置开始匹配，匹配过的位置不再匹配，也就是说对于源字符串的每一个位置，都只匹配了一次，所以总体时间复杂度为 $O(n)$ ，为了计算辅助数组 $next[i]$ 需要先对字符串 T 进行一次扩展 KMP 算法处理，所以算法总的时间复杂度为 $O(n+m)$ ，其中 n 为源字符串的长度， m 为匹配字符串的长度。

以下给出扩展 KMP 匹配算法的一种参考实现。

```
//-----3.4.4.cpp-----
const int MAXN = 1024;

void getNext(string &T, int next[]) {
    int pl = 0, pr = 0, m = T.size();
    next[0] = m;
    for (int k = 1; k < m; k++) {
        if (k >= pr || k + next[k - pl] >= pr) {
            if (k >= pr) pr = k;
            while (pr < m && T[pr] == T[pr - k]) pr++;
            next[k] = pr - k, pl = k;
        } else next[k] = next[k - pl];
    }
}

void getExtend(string &S, string &T, int extend[], int next[]) {
    int pl = 0, pr = 0;
    int n = S.size(), m = T.size();
    getNext(T, next);
    for (int k = 0; k < n; k++) {
        if (k >= pr || k + next[k - pl] >= pr) {
            if (k >= pr) pr = k;
            while (pr < n && pr - k < m && S[pr] == T[pr - k]) pr++;
            extend[k] = pr - k, pl = k;
        } else extend[k] = next[k - pl];
    }
}
//-----3.4.4.cpp-----
```

3.4.5 Z 算法

Z 算法可以在 $O(n)$ 的时间复杂度内完成字符串匹配。给定字符串 S ，其前缀定义为从字符串第一个字符开始的任意连续字符，其后缀定义为从字符串中任意一个字符开始到最末一个字符所构成的字符串。通过 Z 算法可以在线性时间内确定 S 的任意后缀与 S 本身的最长公共前缀。约定字符串 S 的下标从 0 开始， $S[i, j]$ 表示字符串 $S_iS_{i+1}\cdots S_j$ ，算法步骤如下：

- (1) 初始时令 $i=1, j=1$ ；
- (2) 若 $j < i$ ，则 $j=i$ ，比较 S_j 与 S_{j-i} ，若相等则自增继续比较，否则停止，此时有 $z[i]=j-i$ 且 $S_j \neq S_{j-i}$ ；
- (3) 考虑利用 $S[0, j-i-1]$ 与 $S[i, j-1]$ 相等这个性质优化 $z[i+1, j-1]$ 的计算，令指针 k 从 $i+1$ 开始向后遍历，若 $k+z[k-i] < j$ ，则 $z[k]=z[k-i]$ ，否则停止遍历，令 $i=k$ ，转步骤 (2)。

```
//-----3.4.5.cpp-----
int z[1 << 20];

int Z(string &s) {
```

```

z[0] = s.length();
for (int i = 1, j = 1, k; i < s.length(); i = k) {
    if (j < i) j = i;
    while (j < s.length() && s[j] == s[j - i]) j++;
    z[i] = j - i, k = i + 1;
    while (k + z[k - i] < j) z[k] = z[k - i], k++;
}
//-----3.4.5.cpp-----

```

利用 z 数组, 可以高效地解决下述字符串匹配问题: 给定一个模板串 P 和文本串 T , 确定 P 在 T 的哪些位置出现。令字符串 $S=P+\$+T$, 对 S 执行 Z 算法, 检查 $z[P.length()+1]$ 至 $z[P.length() + T.length()]$, 若 $z[i]=P.length()$, 表明从 $T_{i-P.length()-1}$ 处开始的字符与 P 匹配¹。

强化练习: [12467 Secret Word](#)。

3.4.6 字符串的最小表示

给定一个环形的字符串 s , 求字符串 t , 使得 t 是所有与 s 长度相同的子串里字典序最小的字符串。

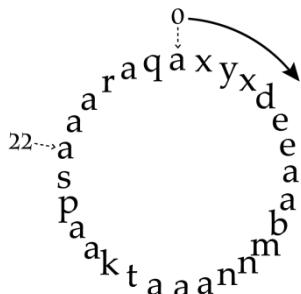


图 3-11 令 $s=$ “axyxdeeaabmnnaatkaapsaaraq”, 从 0 开始顺时针方向计数, 则 s 的最小表示从序号 22 开始, 为 $t=$ “aaaraqaxyxdeeaabmnnaatkaaps”

朴素的方法是从每个字符的起始位置进行比较, 效率为 $O(n^2)$, 当字符串长度较大时, 显然会超时。此问题可以通过后缀数组解决, 或者转化为模式匹配进而使用 KMP 算法解决, 时间复杂度为 $O(n)$ 。此处介绍一种更为简洁的方法, 称为字符串最小表示^[27]。其算法如下: 首先将字符串复制一遍衔接在源串后, 将环转化为链。使用两个指针 i 和 j 维护最优起始位置和待比较起始位置, 设 $k=\{\text{最小的 } x \mid s[i+x] \neq s[j+x]\}$, 如果 $k \geq n$, 那么 i 已经是最优起始位置。否则, 当 $s[j+k] > s[i+k]$ 时, 直接将 j 向后滑动 $k+1$ 个位置, 若此时 $s[j+k] < s[i+k]$, 则更新 $j=\max(j, i+k)+1$, 并同时更新最优位置 i , 重复上述步骤, 直到 $j \geq n$ 时算法结束。

```

//-----3.4.6.cpp-----
int minimumIdx(string &s) {
    int i = 0, j = 1, k, n = s.length();
    while (i < n && j < n) {
        k = 0;
        while (k < n && s[(i + k) % n] == s[(j + k) % n]) k++;

```

¹ 此处使用字符 ‘\$’ 作为分隔符, 基于字符串 P 和 T 中均不存在字符 ‘\$’ 的假设, 如果 P 和 T 中存在字符 ‘\$’, 可以选取某个其他的字符作为分隔符, 只要该字符在 P 和 T 中均不存在即可。

```

if (k == n) break;
if (s[(i + k) % n] > s[(j + k) % n]) {
    i = max(j, i + k + 1);
    j = i + 1;
}
else j += k + 1;
}
return i;
}
//-----3.4.6.cpp-----

```

强化练习：[719 Glass Beads^B](#)，[1584 Circular Sequence^A](#)。

3.5 字符串数据结构及应用

3.5.1 Trie

Trie^I又称字典树，它主要支持两种操作：插入一个字符串以及查询一个字符串是否存在。使用 C++ 的 set 或 map 数据结构也可以完成这项任务，但是 set 或 map 数据结构不能完成 Trie 的其他功能，例如寻找最长公共前缀，而且熟悉 Trie 的思想及实现对拓展解题思维具有帮助。

简单来说，Trie 所表示的是一种树形结构，每条边都和一个字符相关联，结点在树中的位置决定了它代表的字符串。例如，给定字符串：tom、tomy、tim、andy、andrew、mary，其对应的 Trie 数据结构如图 3-12 所示。

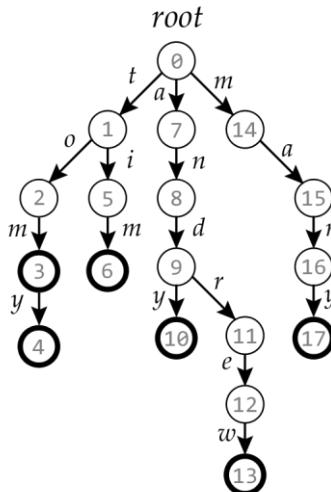


图 3-12 Trie 数据结构示意。从根结点 *root* 到某一结点，将路径上经过的字符连接起来，为该结点所对应的字符串。

根结点表示空字符串。加粗圆圈表示当前结点是结尾字符，非加粗圆圈表示结点只是某个字符串的前缀，结点内的数字为结点的编号

以下是 Trie 数据结构的一种参考实现，读者可以结合注释进行理解。

```

//-----3.5.1.cpp-----

```

^I 名称 Trie 来源于 information reTRIEval。

```

// CHILDREN 表示每个结点最多可能的子结点数量，因为只考虑记录小写字母表示的字符串，
// CHILDREN 取值为 26。OFFSET 表示偏移量，将小写字母调整为从 0 开始计数。
const int MAXN = 102400, CHILDREN = 26, OFFSET = 'a';

// Trie 数据结构。
struct Trie {
    int cnt;      // 结点计数
    int root;     // 根结点序号
    int child[MAXN][CHILDREN]; // 结点
    int ending[MAXN]; // 标识某个结点是否为字符串的结尾位置
    // Trie 数据结构的初始化。
    Trie() {
        memset(child[0], 0, sizeof(child[0]));
        root = cnt = ending[0] = 0;
    }

    // 将字符串 s 插入 Trie 中。
    // 具体方法是沿着 Trie 的根往下，逐个比对结点存储的字符是否与字符串中的字符相同，
    // 如果相同，则沿着该分支继续向下，否则，新建分支，继续比较，一直到字符串的结尾。
    void insert(const string s) {
        int *current = &root;
        for (auto c : s) {
            current = &child[*current][c - OFFSET];
            if (!(*current)) {
                *current = ++cnt;
                memset(child[cnt], 0, sizeof(child[cnt]));
                ending[cnt] = 0;
            }
        }
        ending[*current] = 1;
    }

    // 查询字符串 s 是否于 Trie 中存在。
    // 具体方法是沿着 Trie 的根向下，逐个比对结点存储的字符是否与字符串中的字符相同，
    // 如果比较到字符串的末尾而且当前结点也被标记为结尾位置则表明 Trie 中存在此字符串。
    bool query(const string s) {
        int *current = &root;
        for (auto c : s) {
            if (!(*current)) break;
            current = &child[*current][c - OFFSET];
        }
        return (*current && ending[*current]);
    }
};

//-----3.5.1.cpp-----//

```

从 Trie 的实现可以看出，这是一种以空间换取时间效率的数据结构。如果给定的字符串存在较多重复，则在存储时不会浪费太多空间，因为公共前缀只存储一次。如果给定的字符串重叠较少或者字符集较大（需要为每个可能出现的字符分配一个存储位来表示此字符是否出现），则需要较多的空间花销。

Trie 主要有以下几种用途：

(1) 字符串排序。在将字符串插入 Trie 后，使用前序遍历（有向边上具有较小的 ASCII 码值的字符先遍历）即可获得排序字符串。

(2) 词频统计。由于 Trie 的结点上可以存储额外的信息，可以在结点上设置一个域，每次插入字符串时，在表示字符串结尾的结点将该域所代表的值加 1，最后此域的值即表示该结点结尾的字符串所出现的次数。

(3) 以 Trie 为基础构建后缀树以查找两个字符串的最长公共前缀。

给定一个字符串 $s=a_0a_1a_2\cdots a_{n-1}$ ，其后缀 s_i 定义为从位置 i 开始到结尾的字符串， $0 \leq i \leq n-1$ 。例如，字符串“mississippi”的所有后缀为：

mississippi, ississippi, ssissippi, sissippi, issippi, ssippi, sippi, ippi, ppi, pi, i。

将给定字符串的所有后缀构建成 Trie 的形式，称为后缀 Trie 或后缀树 (suffix tree)。可以在 $O(n)$ 的时间复杂度内完成后缀树的构造。利用后缀树可以完成诸如后缀间的最长公共前缀、两个字符串的最长公共子串、最长重复子串查询。不过构建后缀树在竞赛环境中相对来说代码较多，在具体实现时容易出错，可以使用后续介绍的后缀数组来替代后缀树完成相应功能。

3.5.2 Aho-Corasick 算法

Aho-Corasick 算法是由 Aho 和 Corasick 共同提出的一种多模式串匹配算法^[28]。回顾此前介绍的 KMP 算法，它是一种单模式串匹配算法，也就是说，在同一时间内，KMP 算法只对一个模式串进行匹配。如果存在多个模式串要与主串进行匹配，则需要分开逐个执行一次 KMP 算法，而使用 Aho-Corasick 算法，能够一次性将所有模式串构建为转移图 (goto graph)，利用一趟比较确定主串中所有模式串可能的匹配，从而提高了效率。本质上，可以将 Aho-Corasick 算法视为 KMP 算法的“并行匹配”版本。

定义字符串 (string) 为一个有限的符号 (symbol) 序列，令 $K=\{y_1, y_2, \dots, y_k\}$ 为关键字 (keyword) 集合， x 为任意的文本字符串 (text string)，Aho-Corasick 算法能够在 $O(n)$ 的时间复杂度内确定 K 中所有关键字在 x 中出现的位置。Aho-Corasick 算法的工作过程和使用有限自动机 (finite automata) 进行匹配的过程类似，即可以将 Aho-Corasick 算法看做是一种模式匹配机 (pattern matching machine)。模式匹配机由一系列状态 (state) 构成，每个状态由一个数字予以表示，匹配机不断读入 x 中的字符，通过状态转移 (state transition) 确定 x 中是否包含 K 中的关键字，如果发现有特定的关键字就予以输出。匹配机主要由三个函数构成：转移函数 (goto function)，失配函数 (failure function)，输出函数 (output function)。以关键字集合 $K=\{\text{he, she, his, hers}\}$ 为例，其相应的模式匹配机如图 3-13 所示。

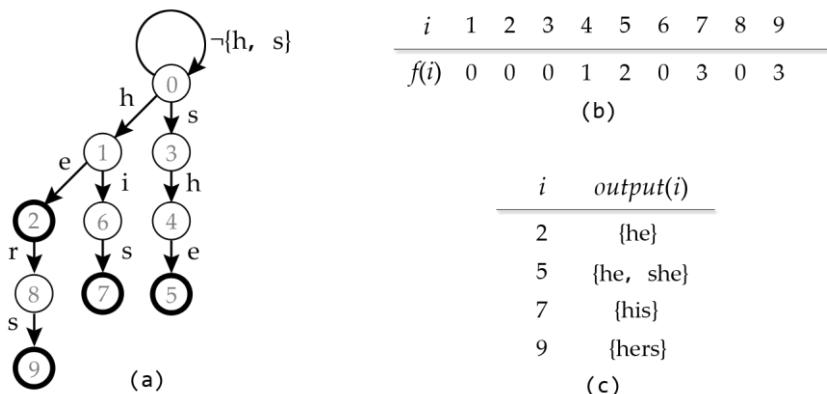


图 3-13 关键字集合 $K=\{\text{he, she, his, hers}\}$ 所对应的模式匹配机。(a) 转移函数, 使用有向有根树予以表示。

结点中的数字表示状态的编号。状态 0 为初始状态, 如果当前位于状态 0 且输入字符不属于集合 {h, s}, 则后续转移到状态 0, 相当于失配函数中的 $f(0)=0$ 。(b) 失配函数, 失配函数指示当匹配失败时, 当前状态需要跳转到哪个状态。例如 $f(1)=0$, 表示当位于状态 1 时, 如果读入的字符不属于集合 {e, i}, 则当前状态应该跳转到状态 0。失配函数的作用是“补全”转移图, 也就是说, 当状态 1 发生失配时, 需要跳转到状态 0, 与之对应的应该有一条从状态 1 出发到达状态 0 的有向边。(c) 输出函数, 将状态映射到输出。具有加粗外圈的状态表示该状态关联有相应的输出

在模式匹配机中, 转移函数使用有向图表示, 故又称转移图, 使用 g 予以表示。 g 由两个域构成, 一个是当前的状态 s , 另外一个是输入的字符 a , 转移函数的作用是根据当前状态 s 和输入字符 a , 将其映射到另外一个状态 s' 或者报告匹配失败 (fail), 即 $g(s, a)=s'$ 或 $g(s, a)=\text{fail}$ 。例如 $g(1, e)=2$, 表示在状态 1 时, 如果输入字符为 ‘e’, 则转移到状态 2; 而 $g(1, k)=\text{fail}$, 即位于状态 1 时, 如果输入字符为 ‘k’, 则报告匹配失败。状态 0 被设定为起始状态 (start state), 它较为特殊, 对于任意的字符 σ 来说, $g(0, \sigma) \neq \text{fail}$, 即从状态 0 开始, 如果能够匹配则跳转到相应的其他状态, 否则会回到状态 0, 而对于非 0 状态的其他状态, 如果未发现匹配, 则会报告失败。注意, 图 3-13 中并未将所有转移关系绘出。例如, 当位于状态 1 时, 如果输入字符不为 ‘e’, 应该跳转到状态 0。这些未予绘出的转移关系由失配函数来指定。

如果转移函数报告匹配失败, 则查看失配函数 f , 根据失配函数指定的当前状态 s 需要跳转到的后续状态 $f(s)$ 进行跳转。例如 $f(4)=1$, 表示当位于状态 4 时, 如果输入字符不为 ‘e’, 则从状态 4 不能到达状态 5, 需要跳转, 此时已经匹配了字符 ‘s’ 和 ‘h’, 而字符 ‘h’ 是关键字 “he”的第一个字符, 因此可以跳转到状态 1, 继续沿着转移图进行匹配。

如果在转移过程中, 某个状态和输出函数 $output$ 有关联, 即 $output(s)$ 不为空, 则表明在此处发现了匹配, 可以进行输出。例如 $output(2)=\{\text{he}\}$, 表示到达状态 2 时, 已经匹配了关键字 “he”, 可以输出。

有了模式串对应的转移函数、失配函数、输出函数, 就可以进行匹配。具体方法是从初始状态 0 开始, 每次读入一个字符, 根据转移函数和读入的字符从当前状态转移到下一个状态 (可能发生失配, 如果发生失配则进行状态的跳转), 如果某个状态与输出函数有关联则表明产生了匹配, 予以输出。可以将模式匹配机的工作过程使用以下伪代码予以描述。

```
// x=a1a2…an 为文本字符串, g 为转移函数, f 为失配函数, output 为输出函数。
Aho-Corasick-Algorithm(x, g, f, output)
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, ai) = fail do state ← f(state)
            state ← g(state, ai)
            if output(state) ≠ empty then
                print i
                print output(state)
            end
        end
    end
end
```

那么如何构建转移函数、失配函数以及输出函数呢? 转移函数的构建可以使用类似于建立 Trie 的过程

来完成。给定一个关键字 y ，从 Trie 的根开始，逐个检查关键字的每个字符 a ，如果字符 a 在 Trie 中存在，则沿着有向树继续向下，如果 a 不存在，则新建一个结点。最终关键字 y 的每个字符 a 都会关联到 Trie 中的某条边，如果某个结点恰好位于关键字的结尾位置，则将此结点与输出函数 $output$ 相关联。

可以将构建转移函数的过程使用以下伪代码予以表示。

```

// 构建转移函数。K 为关键字集合  $\{y_1, y_2, \dots, y_k\}$ 。
Construction-of-the-Goto-Function(K)
begin
    // 初始状态为 0，对应 Trie 的根结点。
    newstate  $\leftarrow 0$ 
    // 将所有关键字逐个插入到 Trie 中。
    for  $i \leftarrow 1$  until  $k$  do Enter( $y_i$ )
    // 对初始状态 0 的无效转移进行特殊处理：如果从初始状态 0 出发，某个字符 a 未能匹配，
    // 则后续状态仍为初始状态 0。
    for all  $a$  such that  $g(0, a) = fail$  do  $g(0, a) \leftarrow 0$ 
end

// 将关键字  $a_1a_2\dots a_m$  插入到转移函数所对应的转移图中。
Enter( $a_1a_2\dots a_m$ )
begin
    state  $\leftarrow 0$ 
     $j \leftarrow 0$ 
    // 从转移图的根结点开始，逐个插入关键字的字符，直到产生失配。
    while  $g(state, a_j) \neq fail$  do
        begin
            state  $\leftarrow g(state, a_j)$ 
             $j \leftarrow j + 1$ 
        end
    // 从失配处开始创建新的状态，使用相应的边来存储关键字的字符。
    for  $p \leftarrow j$  until  $m$  do
        begin
            newstate  $\leftarrow state + 1$ 
             $g(state, a_p) \leftarrow newstate$ 
            state  $\leftarrow newstate$ 
        end
    // 将关键字关联到相应状态。
     $output(state) \leftarrow \{a_1a_2\dots a_m\}$ 
end

```

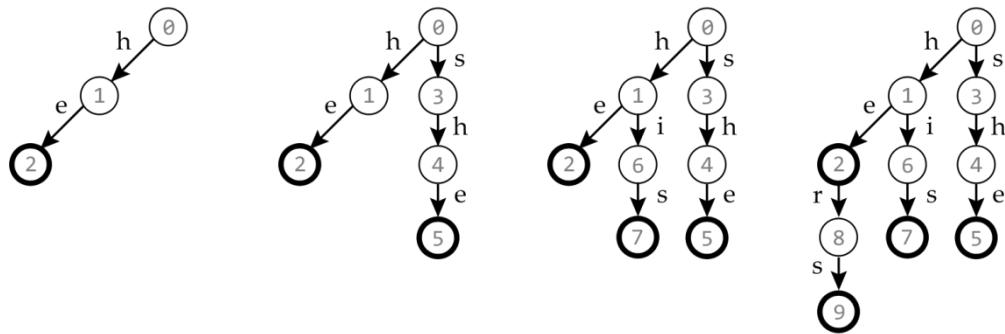


图 3-14 构建转移函数。每次将一个关键字添加到 Trie 中。第一次将关键字“he”添加到 Trie 中，新建 1 和 2 两个结点，结点 2 关联到输出函数；第二次将关键字“she”添加到 Trie 中，新建 3、4、5 三个结点，结点 5 关联到输出函数；第三次将关键字“his”添加到 Trie 中，新建 6 和 7 两个结点，结点 7 关联到输出函数；第四次将关键字“hers”添加到 Trie 中，新建 8 和 9 两个结点，结点 9 关联到输出函数。注意，在将关键字“his”添加到 Trie 中时，“his”和“he”具有最长公共前缀“h”，因此结点 1 为关键字“he”和“his”的公共结点；在将关键字“hers”添加到 Trie 中时，“hers”和“he”具有最长公共前缀“he”，因此 1 和 2 两个结点为关键字“he”和“hers”的公共结点

在转移图中，定义从初始状态 0 到达给定的某个状态 s 的最短路径长度为状态 s 的深度 (depth)。如图 3-14 所示，初始状态 0 的深度为 0，状态 1 和 3 的深度为 1，状态 2、6、4 的深度为 2，依此类推。观察由关键字集合所构建的转移图，容易知道，对于深度为 d ($d \geq 1$) 的某个状态 s' ，可以由深度为 $d-1$ 的某个状态 s 通过一步转移得到。进一步地，如果对于深度为 $d-1$ 的所有状态，其失配函数所映射的跳转状态均已确定，那么对于任意深度为 d 的状态来说，其失配函数所映射的跳转状态必定是深度小于 d 的某个状态，也就是说，可以使用广度优先遍历 (Breath First Search, BFS) 沿着转移图“逐层”构造失配函数。具体来说，对于所有深度为 1 的状态 s ，令 $f(s)=0$ ，这样深度为 1 的状态的失配函数均已计算，在此基础上，计算深度为 2 的状态的失配函数，接着在深度为 2 的状态的失配函数已经计算的基础上，计算深度为 3 的状态的失配函数，依此类推。

为了计算深度为 d 的状态的失配函数，考虑深度为 $d-1$ 的每个状态 r ，进行下述操作：

- (1) 如果对于所有的输入字符 a 来说，均有 $g(r, a) = \text{fail}$ ，则忽略；
 - (2) 否则，对于每个满足 $g(r, a) = s$ 的输入字符 a ，进行如下的操作：
 - (2.1) 置 $state = f(r)$ ；
 - (2.2) 执行 $state \leftarrow f(state)$ 若干次，直到出现某个状态 $state$ ，使得 $g(state, a) \neq \text{fail}$ （注意，当位于初始状态 0 时，对于任意输入字符 a 来说，均有 $g(0, a) \neq \text{fail}$ ，故使得 $g(state, a) \neq \text{fail}$ 的状态总是存在）；
 - (2.3) 置 $f(s) = g(state, a)$ 。

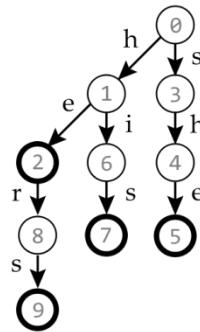


图3-15 转移函数所对应的转移图

如图3-15所示，由于状态1和3的深度均为1，故置 $f(1)=f(3)=0$ 。接着计算状态2、6、4的失配函数。为了计算 $f(2)$ ，首先置 $state=f(1)=0$ ，由于 $g(0, 'e')=0$ ，可以得知 $f(2)=0$ 。为了计算 $f(6)$ ，置 $state=f(1)=0$ ，由于 $g(0, 'i')=0$ ，得到 $f(6)=0$ 。为了计算 $f(4)$ ，置 $state=f(3)=0$ ，由于 $g(0, 'h')=1$ ，可得 $f(4)=1$ 。

以下是使用广度优先遍历构建失配函数过程的伪代码表示。

```
// 由转移函数和输出函数构建失配函数。g为转移函数，output为输出函数。
Construction-of-the-Failure-Function(g, output)
begin
    queue  $\leftarrow$  empty
    for each a such that  $g(0, a) = s \neq \theta$  do
        begin
            queue  $\leftarrow$  queue  $\cup \{s\}$ 
             $f(s) \leftarrow 0$ 
        end
    while queue  $\neq$  empty do
        begin
            let r be the next state in queue
            queue  $\leftarrow$  queue  $- \{r\}$ 
            for each a such that  $g(r, a) = s \neq fail$  do
                begin
                    queue  $\leftarrow$  queue  $\cup \{s\}$ 
                    state  $\leftarrow f(r)$ 
                    while  $g(state, a) = fail$  do state  $\leftarrow f(state)$ 
                     $f(s) \leftarrow g(state, a)$ 
                    output(s)  $\leftarrow$  output(s)  $\cup$  output(f(s))
                end
        end
    end

```

输出函数较为简单，如果某个结点是特定关键字的结尾字符，则可将此结点与关键字相关联，只要在转移过程中到达这些特殊的结点，就表明已经匹配了相应的关键字，予以输出即可。在构建转移函数时已经得到了初始的输出函数，不过此时每个关键字都独立关联到某个状态，在构建失配函数的过程中，可以将已有

的输出函数予以合并，即不同的关键字可能都关联到同一个状态。

以下是 Aho-Corasick 算法的参考实现。

```

//-----3.5.2.cpp-----//
const int MAXN = 10240, CHARSET = 128;

class AhoCorasick {
private:
    int cnt;      // 转移图的结点计数
    int root;     // 转移图的根结点
    int go[MAXN][CHARSET]; // 转移函数
    int fail[MAXN]; // 失配指针
    vector<string> keywords; // 关键字集合
    vector<int> output[MAXN]; // 输出函数

    // 构建转移函数。
    void buildGotoFunction() {
        for (int i = 0; i < keywords.size(); i++) {
            int *current = &root;
            for (auto c : keywords[i]) {
                current = &go[*current][c];
                if (!*current) {
                    *current = ++cnt;
                    memset(go[cnt], 0, sizeof(go[cnt]));
                    output[cnt].clear();
                }
            }
            output[*current].push_back(i);
        }
    }

    // 构建失配函数。
    void buildFailureFunction() {
        queue<int> q;
        for (int i = 0; i < CHARSET; i++)
            if (go[0][i])
                q.push(go[0][i]);
            fail[go[0][i]] = 0;
        }
        while (!q.empty()) {
            int r = q.front(); q.pop();
            for (int i = 0; i < CHARSET; i++)
                if (go[r][i]) {
                    int s = go[r][i], f = fail[r];
                    q.push(s);
                    while (f && !go[f][i]) f = fail[f];
                    fail[s] = go[f][i];
                    output[s].insert(output[s].end(),
                                     output[fail[s]].begin(),
                                     output[fail[s]].end());
                }
        }
    }

public:
    // 初始化。
}

```

```

void initialize() {
    root = cnt = 0;
    keywords.clear();
    memset(go[0], 0, sizeof(go[0]));
    for (int i = 0; i < MAXN; i++) output[i].clear();
}

// 新增关键字。
void add(string s) { keywords.push_back(s); }

// 匹配。
void match(string &s) {
    buildGotoFunction();
    buildFailureFunction();
    int current = root;
    for (auto c : s) {
        while (current && !go[current][c]) current = fail[current];
        current = go[current][c];
        if (output[current].size() > 0) {
            for (auto i : output[current])
                cout << keywords[i] << ' ';
            cout << '\n';
        }
    }
}
};

int main(int argc, char *argv[]) {
    int n;
    string key, line;
    AhoCorasick ac;
    while(cin >> n) {
        ac.initialize();
        for (int i = 0; i < n; i++) {
            cin >> key;
            ac.add(key);
        }
        cin.ignore(256, '\n');
        string text;
        while (getline(cin, line), line.length() > 0) text += line;
        ac.match(text);
    }
    return 0;
}
//-----3.5.2.cpp-----

```

对于以下输入：

```

4
he
she
his
hers
ushers

```

对应的输出为：

she	he
hers	

强化练习: [1449 Dominating Patterns^D](#), [11019* Matrix Matcher^C](#)。

3.5.3 后缀数组

后缀数组 (suffix array) 由 Manber 和 Myers 于 1990 年提出^[29]。相对于后缀树, 后缀数组空间花销较少, 代码实现较为简单, 作为后缀树的“替代品”, 在编程竞赛中应用较多。

简单来说, 后缀数组就是将给定字符串 s 的所有后缀按字典序升序排列后得到的一个次序数组 sa 。为了节省存储空间和便于操作, 一般情况下, sa 存储的是后缀在 s 中的起始位置而不是后缀本身。例如, 给定字符串 “abstract”, 其所有后缀依次为 (从 0 开始计数, 方括号内的数字为后缀在字符串中的起始位置)

[0]abstract, [1]bstract, [2]stract, [3]tract, [4]ract, [5]act, [6]ct, [7]t

对所有后缀按字典序升序排列, 可得

[0]abstract, [5]act, [1]bstract, [6]ct, [4]ract, [2]stract, [7]t, [3]tract

对于排序后的第一个后缀 “abstract”, 它在所有后缀中字典序是最小的, 在原字符串 s 中起始位置为 0; 第二个后缀 “act”, 在所有后缀中字典序是第二小的, 在原字符串 s 中起始位置为 5……将排序后的后缀所对应的起始位置按从左至右的顺序排成一列即为后缀数组 sa , 其元素为

0, 5, 1, 6, 4, 2, 7, 3

容易推知, 如果字符串 s 的长度为 n , 则数组 sa 实际上是闭区间 $[0, n-1]$ 内整数的一个排列。

令 s_i 表示字符串 s 从起始位置 i 开始的后缀, 由于字符串从不同位置起始的任意两个后缀不会相同 (后缀的长度不同, 因此不可能相同), 对于 sa 中的两个元素 $sa[i]$ 和 $sa[j]$, 如果 $i < j$, 根据后缀数组的定义, 必有 $s_{sa[i]} < s_{sa[j]}$ 。相应的, 可以定义名次数组 (rank array) $rank$, 它表示 s 的某个后缀在后缀数组 sa 中的排位。例如, 从字符串 “abstract”的位置 5 开始的后缀为 “act”, 如果从 0 开始计数, 其在后缀数组中排第 1 位, 因此 $rank[“act”] = rank[5] = 1$ 。不难推知, 后缀数组和名次数组互为逆运算——令后缀数组的第 i 个元素为 j , 则从字符串 s 的起始位置 j 开始的后缀是后缀数组的第 i 个元素, 亦即

$$sa[i] = j \Leftrightarrow rank[j] = i$$

构建后缀数组, 朴素的方法是利用时间复杂度为 $O(n \log n)$ 的算法库函数 $sort$ 对所有后缀进行一次排序, 这样就能够得到后缀数组, 但是此种方式的时间成本太高——对两个后缀进行比较的时间复杂度为 $O(n)$, 而需要进行 $O(n \log n)$ 数量级的类似比较, 从而使得总体的时间复杂度为 $O(n^2 \log n)$ 。显然, 在竞赛环境中不能有效应对规模较大的评测数据。虽然可以先在 $O(n)$ 的时间内构造后缀树, 然后再通过 $O(n)$ 的时间对后缀树进行深度优先遍历以得到后缀数组, 但是这样的做法失去了构建后缀数组的意义, 因为在完成后缀树的构建后就可以直接使用其进行问题的求解, 而不需再多此一举去构建后缀数组。

在 Manber 和 Mayer 的论文中, 介绍了如何通过倍增法在 $O(n \log n)$ 的时间复杂度内构造后缀数组的方法, 其核心思想如下: 给定长度为 n ($n > 0$) 的字符串 s , 令 $s[i, j]$ 表示从起始位置 i 开始的 j 个字符构成的子串, 在比较 s 的后缀时, 依次对所有的 $s[i, 1], s[i, 2], s[i, 4], s[i, 8], \dots, s[i, 2^k]$ 进行排序, 很明显, 当 $2^k \geq n$ 时, s 的所有后缀排序即已确定, 这个过程最多需要重复 $\lceil \log n \rceil + 1$ 次。如何使得每次排序的效率尽可能高, 从而使得总的时间复杂度尽可能低呢? 注意到当 $s[i, 1]$ 排序完毕后, 如果对 $s[i, 2]$ 进行排序, 则 $s[i, 2]$ 可以看成是由 $s[i, 1]$ 和 $s[i+1, 1]$ 连接而成, 由于 $s[i, 1]$ 和 $s[i+1, 1]$ 的相对顺序已经在前一次排序中获得, 利用这些信息可以使得对 $s[i, 2]$ 的排序能够在 $O(n)$ 的时间复杂度内完成, 其关键就是采用基数排

序。回顾基数排序，每个数字按照其数位从低到高进行排序，整个排序过程的时间复杂度为 $O(n)$ 。类似的，将 $s[i, 2^k]$ 的排序看成对二元组 $\{s[i, 2^{k-1}], s[i+2^{k-1}+1, 2^{k-1}]\}$ 进行排序，首先对第二关键字（将 $s[i+2^{k-1}+1, 2^{k-1}]$ 视为低位数字）进行排序，然后对第一关键字（将 $s[i, 2^{k-1}]$ 视为高位数字）进行排序，即可得到 $s[i, 2^k]$ 的排序，由于 $s[i, 2^{k-1}]$ 和 $s[i+2^{k-1}+1, 2^{k-1}]$ 在后缀数组中是用其起始位置来表示，其值不会超过字符串 s 的长度 n ，因此可以使用计数排序来分别对 $s[i, 2^{k-1}]$ 和 $s[i+2^{k-1}+1, 2^{k-1}]$ 进行排序，最后合并得到 $s[i, 2^k]$ 的排序，这样每趟排序的时间复杂度可以保持在 $O(n)$ ，由于总共需要最多 $\lceil \log n \rceil + 1$ 次排序过程，则总的时间复杂度为 $O(n \log n)$ 。

倍增法从思想上理解是不复杂的，但是如何将其具体实现为代码呢？需要注意以下两个细节：

- (1) 给定字符串的长度并不一定刚好是 2 的幂次长度，如何处理 $s[i, 2^k]$ 中超出字符串末尾的部分？
- (2) 如何合并 $s[i, 2^{k-1}]$ 和 $s[i+2^{k-1}+1, 2^{k-1}]$ 的排序结果来得到 $s[i, 2^k]$ 的排序？

下面先给出使用倍增法构造后缀数组的一种参考实现。

```
-----3.5.3.1.cpp-----
const int MAXN = 256;

// 计数排序。
void countSort(int *s, int *ranks, int *sa, int n, int m) {
    static int cnt[MAXN];
    memset(cnt, 0, sizeof(cnt));
    for (int i = 0; i < n; i++) cnt[s[ranks[i]]]++;
    for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];
    for (int i = n - 1; i >= 0; i--) sa[--cnt[s[ranks[i]]]] = ranks[i];
}

// 构建后缀数组。
// s 为字符数组， sa 为后缀数组， n 为字符数组的大小， m 为字符集的大小。
// s 中的字符使用其对应的 ASCII 码值表示。
void buildSA(int *s, int *sa, int n, int m) {
    static int ranks[MAXN] = {}, higher[MAXN] = {}, lower[MAXN] = {};
    // 对 s[i, 1] 进行排序。
    iota(ranks, ranks + MAXN, 0);
    countSort(s, ranks, sa, n, m);
    // 由后缀数组获得名次数组。
    ranks[sa[0]] = 0;
    for (int i = 1; i < n; i++) {
        ranks[sa[i]] = ranks[sa[i - 1]];
        ranks[sa[i]] += (s[sa[i]] != s[sa[i - 1]]);
    }
    // 比较的子串长度依次倍增。
    for (int i = 0; (1 << i) < n; i++) {
        for (int j = 0; j < n; j++) {
            higher[j] = ranks[j] + 1;
            lower[j] = (j + (1 << i) >= n) ? 0 : (ranks[j + (1 << i)] + 1);
            sa[j] = j;
        }
    }
    // 基数排序。
    // 因为只有两位“数字”，可以通过两次计数排序实现后缀的排序。先排序末位，后排序首位。
    // 数组 higher 存储的是首位“数字”，数组 lower 存储的是末位“数字”。
    countSort(lower, sa, ranks, n, n);
    countSort(higher, ranks, sa, n, n);
}
```

```

    // 因为可能存在两个后缀的名次相同的情况，需要通过比较字符串进一步确定名次。
    ranks[sa[0]] = 0;
    for (int j = 1; j < n; j++) {
        ranks[sa[j]] = ranks[sa[j - 1]];
        ranks[sa[j]] += (higher[sa[j - 1]] != higher[sa[j]] ||
            lower[sa[j - 1]] != lower[sa[j]]);
    }
}
//-----3.5.3.1.cpp-----/

```

侯捷在其《STL 源码剖析》一书中曾经说过：“源码之前，了无秘密。”但是面对一段实现较为精巧的代码，在没有他人从旁指引的情况下，要想达到快速理解的目的似乎并没有太好的办法。其中一种较为“笨拙”但有效的方法就是选择具有代表性的输入，观察代码的运行过程中给定的输入发生了何种变换，以此来理解代码的行为。通过这种方法对代码进行理解往往印象更为深刻，如果再进一步结合相应的代码原理性分析，理解的效果将会更佳。

下面以输入“edcbaabcde”为例来解析参考实现代码的行为^I。要构建“edcbaabcde”所对应的后缀数组，可以通过下述调用来实现。

```

int main(int argc, char *argv[]) {
    string S = "edcbaabcde";
    int s[32] = {}, sa[32] = {};
    for (int i = 0; i < S.length(); i++) s[i] = S[i];
    buildSA(s, sa, 10, 128);
    return 0;
}

```

即先将字符串转换为整数数组，每个字符使用其 ASCII 码值予以表示，以便于后续使用计数排序对字符数组进行排序，进而有利于后缀数组的构建。在进行转换后，整数数组 s 包含 10 个元素，其值依次为：

```
101 100 99 98 97 97 98 99 100 101
```

在调用 buildSA 时，字符集的大小设定为 128，这是因为竞赛环境下给定的字符串一般均为 ASCII 字符，不同字符的个数最多为 128 个，因此在进行计数排序时最多只需统计 128 种不同字符的个数^{II}。

在函数 buildSA 中，起始声明了三个静态数组：

```
static int ranks[MAXN] = {}, higher[MAXN] = {}, lower[MAXN] = {};
```

其中 ranks 为名次数组，其作用是记录各个后缀在后缀数组中的排位，higher 和 lower 为辅助数组，分别存储进行基数排序时的高位数字和低位数字，之所以将它们声明为静态的，其目的是为了在后续过程可以多次使用，不必再次申请内存空间。

构建后缀数组的第一步是对 s[i, 1] 进行排序，参考实现中通过以下两行代码予以实现：

^I 建议读者将代码输入到编译环境中并跟随本书的解析过程编写相应的调试语句查看输出加以印证，这样对倍增法的理解将会更为深刻。

^{II} 对于示例输入“edcbaabcde”，由于其只包含 5 个不同的字符且其 ASCII 码值连续（在 97—101 的范围内），如果以字符‘a’为“参考点”，将其视为‘1’，则“edcbaabcde”可以转换为“5432112345”，那么此字符串所对应的字符集大小为 5，在调用 buildSA 时，可将 128 使用 5 进行替换。

```

iota(ranks, ranks + MAXN, 0);
countSort(s, ranks, sa, n, m);

```

其中库函数 `iota` 的作用是为名次数组 `ranks` 赋初值，将该数组从第一个元素到最后一个元素依次填充从 0 开始的整数序列，接着调用计数排序子函数 `countSort` 实现 $s[i, 1]$ 的有序。

为什么 `countSort` 能够实现 $s[i, 1]$ 的排序呢？`countSort` 所使用的排序方法为计数排序，回顾计数排序，如果给定的序列其数据分布在一个有限且相对较小的范围内（例如，序列中最大和最小的元素之间差的绝对值小于 2^{16} ），那么可以先计数各个不同元素 x 出现的次数 $cnt[x]$ ，确定 $cnt[x]$ 后即可以知道在最终的有序数组中，值为 x 的元素必定会出现 $cnt[x]$ 次，而且其位置必定也是连续的。如果从 1 开始计数位置，只要确定小于 x 的元素 y 的数量 z ，就能够知道值为 x 的元素在最终的有序数组中的起始位置必定是 $z+1$ ，而 z 可以通过累加小于 x 的各个元素 y 的出现次数 $z = \sum_{y < x} cnt[y]$ 予以确定。换句话说，统计值为 x 的元素的出现次数 $cnt[x]$ ，相当于确定了值为 x 的各个元素在最终有序数组中的相对位置，而累加小于 x 的各个元素 y 的出现次数 $z = \sum_{y < x} cnt[y]$ 则确定了值为 x 的元素在最终有序数组中起始的绝对位置。这与编译器对多个源文件进行编译和链接以确定二进制代码的执行次序有些类似——编译器在编译由多个源文件构成的源代码时，先对单个的源文件进行编译，确定二进制代码在最终执行序列中的相对顺序，而后通过与库文件的链接，确定所有二进制代码在最终执行序列中的绝对顺序。

下面对 `countSort` 函数的结构进行“解剖”以理解其行为。该子函数的起始两行代码为：

```

static int cnt[MAXN];
memset(cnt, 0, sizeof(cnt));

```

根据前述分析，其作用是声明一个计数数组并将其全部置 0 以用于统计各个字符的出现次数，将其声明为静态数组是为了便于后续的重复使用，不需要反复申请内存空间以提高效率。

	0	1	2	3	4	5	6	7	8	9
S	e	d	c	b	a	a	b	c	d	e
s	101	100	99	98	97	97	98	99	100	101
cnt	0	0	0	0	0	0	0	0	0	·
ranks	0	1	2	3	4	5	6	7	8	9
sa	-	-	-	-	-	-	-	-	-	·

图 3-16 字符串 $S = \text{"edcbaabcde"}$ 转换为字符（整数）数组 s 后，调用 `countSort` 子函数进行排序时的初始状态。计数数组 cnt 的初始值均为 0，名次数组 $ranks$ 的初始值为从 0 开始的整数序列，后缀数组 sa 的初始值为未定义状态

接着两行代码：

```

for (int i = 0; i < n; i++) cnt[s[ranks[i]]]++;
for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];

```

其作用是先计数各个字符出现的次数以确定“相对位置”，之后是根据字符集的大小 m （此处 $m=128$ ），按照从小到大的顺序逐次累加各种字符的出现总次数以确定“绝对位置”。

	94	95	96	97	98	99	100	101	102	103	104
<i>cnt</i>	.	.	.	0	0	0	2	2	2	2	0
<i>cnt</i>	.	.	.	0	0	0	2	4	6	8	10

图 3-17 字符数组频次计数完毕以及累加完毕时计数数组 *cnt* 中各元素的值。在计数各字符的出现次数后，由于字符串“edcbaabcde”中从‘a’到‘e’的字符各出现 2 次，而字符‘a’到‘e’的 ASCII 码值依次为 97 到 101，故数组 *cnt* 中从序号 97 到 101 的元素值均为 2，其他元素值均为 0。累加各字符出现的次数后，数组 *cnt* 中从序号 0 到 96 的元素值均为 0，从序号 97 到 101 的元素值依次递增 2，从序号为 102 的元素开始，其值均为 10

接着的一行代码：

```
for (int i = n - 1; i >= 0; i--) sa[--cnt[s[ranks[i]]]] = ranks[i];
```

其作用是根据统计得到的“相对位置”和“绝对位置”来确定各个元素在最终的有序数组中的位置，之所以采用从后往前的顺序确定次序有两个原因：一是为了使得排序是“稳定的”，即两个相同的字符 x_i 和 x_j ，如果在原始数组中其位置满足 $i < j$ ，则在最终的有序数组中， x_i 和 x_j 的最终位置 i' 和 j' 仍然满足 $i' < j'$ ；二是在累加各种字符的出现频次时，使用的是 *cnt* 数组本身，如果严格按照前述的分析，需要另外一个数组来累加小于 x 的其他元素 y 出现的次数 $z = \sum_{y < x} cnt[y]$ ，而为了代码的简练，仍使用数组 *cnt* 进行累加后得到的是包含 x 自身在内的元素出现次数 $z' = \sum_{y \leq x} cnt[y]$ ，如果从 1 开始计数位置，则次数 $z' = \sum_{y \leq x} cnt[y]$ 对应 x 在最终有序数组中的最后一个位置的序号，因此需要通过逆序来确定次序。

	94	95	96	97	98	99	100	101	102	103	104
<i>cnt</i>	.	.	.	0	0	0	2	4	6	8	10
	0	1	2	3	4	5	6	7	8	9	
<i>S</i>	e	d	c	b	a	a	b	c	d	e	
<i>S</i>	101	100	99	98	97	97	98	99	100	101	
<i>sa</i>	4	5	3	6	2	7	1	8	0	9	

图 3-18 根据累加的字符频次确定长度为 1 的子串在后缀数组中的次序。以序号为 8 的字符‘d’为例，该字符的 ASCII 码值为 100，在计数数组 *cnt* 中序号为 100 的元素的值为 8，表示 ASCII 码值小于等于 100 的元素总共出现了 8 次。不难得知，若将 *S* 的所有长度为 1 的子串进行排序，从 1 开始计数，子串“d”的最后位置必定位于第 8 位。由于后缀数组从 0 开始计数，初始从 1 开始计数的位置需要相应减去 1，可得子串 $s[8, 1] = "d"$ 在名次数组中的值为 7（根据名次数组和后缀数组互为逆运算的关系，由 $rank[8] = 7$ 推出 $sa[7] = 8$ ），即在后缀数组中，位于第 7 位的字符串是从序号 8 开始的子串 $s[8, 1]$ 。依此类推，可以确定其他子串 $s[i, 1]$ 在后缀数组中的次序

在使用 `countSort` 函数获得 $s[i, 1]$ 的后缀数组后，需要确定各个后缀的名次，以便为后续使用基数排序确定 $s[i, 2]$ 的次序做准备。也就是说，需要确定基数排序中高位数字 $s[i, 1]$ 和低位数字 $s[i+1, 1]$ 的具体值，而该数位的具体值可以使用后缀在名次数组中的值来“代表”，即某个后缀的名次数组值越小，表明该后缀越小，那么就可以使用一个较小的数值来“代表”该后缀，该数值也就是高位数字或低位数字的值。在

参考实现中是通过下述代码予以实现：

```
ranks[sa[0]] = 0;
for (int i = 1; i < n; i++) {
    ranks[sa[i]] = ranks[sa[i - 1]];
    ranks[sa[i]] += (s[sa[i]] != s[sa[i - 1]]);
}
```

回顾名次数组的定义，它表示从位置 i 起始的后缀在后缀数组中的排位，其与后缀数组互为“逆反关系”，通过后缀数组可以直接得到名次数组，通过名次数组也可以得到后缀数组。但是此处并不是直接由后缀数组获得名次数组，这是为什么呢？因为在后续的基数排序中要考虑到两个数字的值相同的情况。以“edcbaabcde”为例，第一个字符和最后一个字符均为‘e’，在转换为数位之后，这两个字符应该具有相同的值，如果按照一一对应的关系将后缀数组转换为名次数组，那么相同的字符会具有不同的名次，最终转换得到的数位是不同的，这会导致后续的计数排序过程“失效”，从而无法获得正确的排序结果。

	0	1	2	3	4	5	6	7	8	9
S	e	d	c	b	a	a	b	c	d	e
s	101	100	99	98	97	97	98	99	100	101
sa	4	5	3	6	2	7	1	8	0	9
$ranks$	4	3	2	1	0	0	1	2	3	4

图 3-19 根据后缀数组 sa 获得名次数组 $ranks$ 。实际上将后缀 $s[i, 1]$ 表示成数位上的数字，由于基数排序时要求相同的字符其数位值必须相同，否则将无法得到正确的排序，因此序号为 4 和元素和序号为 5 的元素（字母‘a’）其对应的数位值均为 0，可以依此类推得到其他字符所对应的数位的数值

在获得 $s[i, 1]$ 的数位表示后即可准备为 $s[i, 2]$ 排序，此时需要确定 $s[i, 1]$ 对应的高位数位和 $s[i+1, 1]$ 所对应的低位数位。参考实现中使用下述代码确定高位数位和低位数位的具体值：

```
for (int j = 0; j < n; j++) {
    higher[j] = ranks[j] + 1;
    lower[j] = (j + (1 << i) >= n) ? 0 : (ranks[j + (1 << i)] + 1);
    sa[j] = j;
}
```

回顾之前的解析，基数排序时分为高位数位和低位数位，此处数组 $higher$ 存储的是高位数位，数组 $lower$ 存储的是低位数位。考虑到字符串的长度不一定是 2 的幂的整数倍，需要为超出字符串长度的对应数位考虑一个合适的表示，因此使用 0 来表示超出字符串长度的位置所对应的数位值。由于 0 已经被占用，将其他未超过字符串长度的位置所对应的数位值在其原始值上偏移 1 以示区别。

	0	1	2	3	4	5	6	7	8	9
$s[i, 2]$	e	d	c	b	a	a	b	c	d	e
	d	c	b	a	a	b	c	d	e	-
$s[i, 2]$	101	100	99	98	97	97	98	99	100	101
	100	99	98	97	97	98	99	100	101	-
$ranks$	4	3	2	1	0	0	1	2	3	4
$higher$	5	4	3	2	1	1	2	3	4	5
$lower$	4	3	2	1	1	2	3	4	5	0

图 3-20 根据名次数组 $ranks$ 获得 $s[i, 2]$ 所对应的高位数字数组 $higher$ 和低位数字数组 $lower$

在确定了高位数字和低位数字后，连续使用两次计数排序：

```
countSort(lower, sa, ranks, n, n);
countSort(higher, ranks, sa, n, n);
```

这样就能够完成基数排序。第一次计数排序将 $s[i, 2]$ 按照低位数字的值进行排序，接着在低位数字有序的情况下再按照高位数字排序，最终使得整个数字即 $s[i, 2]$ 达到有序的状态，此时的后缀数组 sa 中的值存储的就是 $s[i, 2]$ 排序后的次序。注意，参考实现中为了代码的简练，数组 $ranks$ 和 sa 进行了重复使用，但是两次使用其意义有所区别。在第一次使用时，数组 sa 存储的实际上是低位数字的次序， $ranks$ 存储的是按低位数字进行排序后 $s[i, 2]$ 的后缀数组值，此时的后缀数组是初步的，其值是中间结果。在第二次使用时，两者角色相反，此时 $ranks$ 存储的是在低位数字排序基础上 $s[i, 2]$ 的次序， sa 存储的是在低位数字排序基础上按高位数字进行排序后 $s[i, 2]$ 的后缀数组值，其值是最终结果。由于重复使用了 $ranks$ 和 sa ，在图 3-21 中调用计数排序子函数 `countSort` 时按照实际所使用的数组对代码进行了调整以便能够看出具体操作的数组。

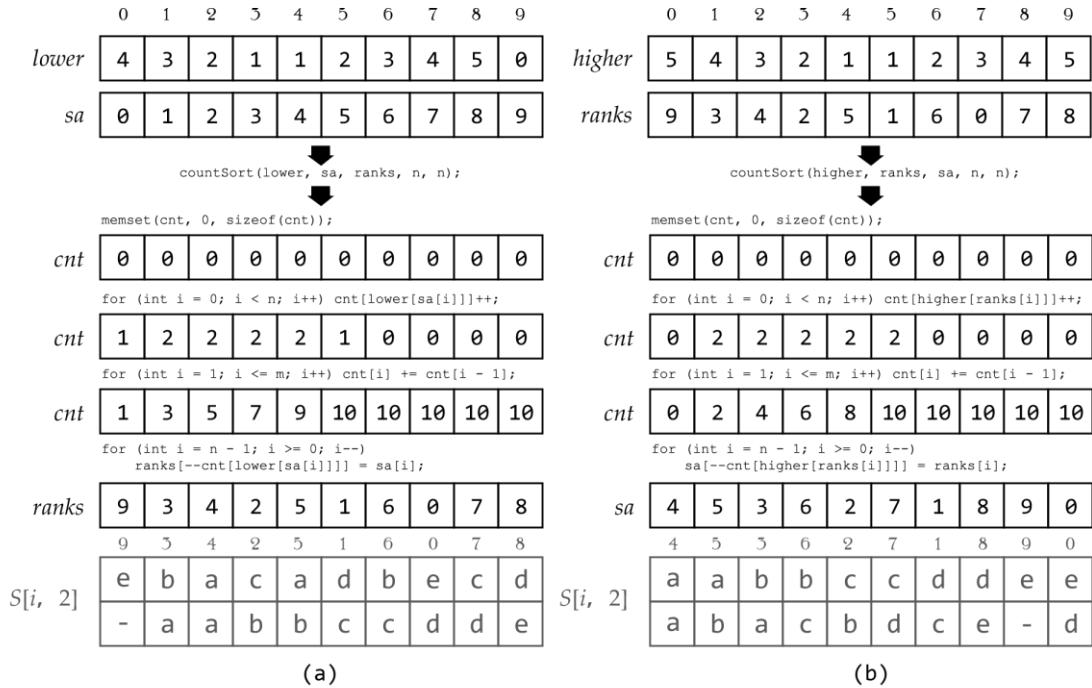


图 3-21 (a) 按照低位数字数组 *lower* 进行第一次计数排序后, 数组 *ranks* 中存储的元素值是 *s[i, 2]* 按低位数字进行排序后的后缀数组值, 在数组 *ranks* 下方的是按低位数字排序后对应的字符串 *S[i, 2]*。(b) 在低位数字排序的基础上, 根据高位数字数组 *higher* 进行第二次计数排序后, 数组 *sa* 中存储的元素值即为 *s[i, 2]* 所对应的后缀数组值, 在数组 *sa* 下方的是按高位数字排序后对应的字符串 *S[i, 2]*

继续使用同样的方法确定名次数组, 即确定下一次排序时后缀 *s[i, 2]* 所对应数位的具体值:

```

ranks[sa[0]] = 0;
for (int j = 1; j < n; j++) {
    ranks[sa[j]] = ranks[sa[j - 1]];
    ranks[sa[j]] += (higher[sa[j - 1]] != higher[sa[j]] ||
        lower[sa[j - 1]] != lower[sa[j]]));
}

```

此处和第一次确定数位值时有细微差别, 原因在于需要考虑高位数字和低位数字两个数位的值, 而最初的一次确定数位值不需考虑此因素。

最终, 通过外循环:

```

for (int i = 0; (1 << i) < n; i++) {
    // 排序。
}

```

每次排序时后缀的长度增加为原来的两倍, 那么至多需要 $\lceil \log_2 n \rceil + 1$ 次操作就能得到 *s* 的后缀数组。

下面再给出一种更为简练和巧妙的倍增法实现^[30], 由于实现代码包含多项“魔法”般的技巧和优化, 理解起来相对困难, 建议读者在充分理解前述倍增法基本实现的基础上, 通过跟踪代码的执行过程和查看其输出来理解代码的行为, 最后再结合标注给出的参考资料尝试进行理解和运用。

```
/*+++++3.5.3.2.cpp+++++*/
const int MAXN = 256;

int ta[MAXN], tb[MAXN], tv[MAXN], ts[MAXN];

int cmp(int *s, int a, int b, int offset) {
    return s[a] == s[b] && s[a + offset] == s[b + offset];
}

void da(int *s, int *sa, int n, int m) {
    int *x = ta, *y = tb, *t;
    // 使用计数排序对 s[i, 1] 进行排序。
    for (int i = 0; i < m; i++) ts[i] = 0;
    for (int i = 0; i < n; i++) ts[x[i]] = s[i]++;
    for (int i = 1; i < m; i++) ts[i] += ts[i - 1];
    for (int i = n - 1; i >= 0; i--) sa[--ts[x[i]]] = i;
}
```

当次序数组 sa 中不同的次序个数 p 达到 n 时表明排序完毕，可以提前退出以提高效率。字符集的大小 m 随着次序个数 p 改变，可以减少计数排序的工作量，提高程序效率。

```
for (int i, k = 1, p = 1; p < n; k *= 2, m = p) {
```

利用上一次对所有长度为 k 的 n 个子串 $s[i, k]$ 排序得到的次序数组 sa , 直接获得子串 $s[i, 2k]$ 低位数字的排序而不需要经过名次数组进行“中转”。在前述的参考实现中, 对 $s[i, 1]$ 使用计数排序后可以得到 $s[i, 1]$ 的次序数组 sa , 然后通过次序数组 sa 来获得 $s[i, 1]$ 的名次数组 $ranks$, 进而通过名次数组 $ranks$ 得到 $s[i, 2]$ 的高位数字数组 $higher$ 和低位数字数组 $lower$, 通过对低位数字数组 $lower$ 再进行一次计数排序后可以得到低位数字的次序, 从而确定 $s[i, 2]$ 的初步次序。而此处的实现却是直接通过次序数组 sa 来得到 $s[i, 2]$ 的初步次序, 其正确性基于以下结论: $s[i, 2k]$ 拆分为 $s[i, k]$ 和 $s[i+k+1, k]$, 必定有 k 个子串 $s[i+k+1, k]$ 的起始字符位置已经超出原字符串 s 的末尾位置, 从而使得这 k 个子串所对应的低位数字为 0。那么很明显, 由于计数排序的“稳定性”, 这 k 个为 0 的低位数字占据了按低位数字排序后的数组 y 的前 k 个位置, 而其他 $n-k$ 个不为 0 的低位数字在此前的排序中已经确定了相对顺序, 因此只需将其按照前一次的排序结果附加在第 k 个位置之后即可。

```
for (p = 0, i = n - k; i < n; i++) y[p++] = i;
```

如前所述, 需要将其他 $n-k$ 个不为 0 的低位数字按照此前排序中已经确定了的相对顺序附加在第 k 个位置之后, 而这 $n-k$ 个不为 0 的低位数字依次对应着子串 $s[k, k]$, $s[k+1, k]$, \dots , $s[n-1, k]$, 这 $n-k$ 个子串的相对顺序可以从次序数组 sa 直接予以“提取”。为什么? 因为 sa 保存的就是 $s[0, k]$, $s[1, k]$, $s[2, k]$, \dots , $s[n-1, k]$ 的次序。在进行“提取”时, 由于 $sa[i]$ 保存的是按字典序排在第 i 位的子串的起始位置 j , 因此只有当 $j \geq k$ (即 $sa[i] \geq k$) 时的子串位置 j 才是我们所需要的。最后, 由于数组 y 的前 k 个位置已经依次填充了 $n-k$, $n-k+1$, \dots , $n-1$, 那么剩下的 $n-k$ 个位置需要依次填充 $0, 1, \dots, n-k-1$, 所以需要将起始位置 j (即 $sa[i]$) 进行“偏移”操作, 其“偏移量”为 k 。

```

for (i = 0; i < n; i++) if (sa[i] >= k) y[p++] = sa[i] - k;
// 利用 s[i, 2k] 的低位数字的次序数组 y 得到 s[i, 2k] 的高位数字。
for (i = 0; i < n; i++) tv[i] = x[y[i]];
// 对高位数字进行计数排序。
for (i = 0; i < m; i++) ts[i] = 0;
for (i = 0; i < n; i++) ts[tv[i]]++;

```

```

for (i = 1; i < m; i++) ts[i] += ts[i - 1];
for (i = n - 1; i >= 0; i--) sa[--ts[tv[i]]] = y[i];
// 根据 s[i, 2k] 的次序数组 sa 得到 s[i, 4k] 的名次数组, 如果两个子串相同则名次相同。
for (t = x, x = y, y = t, p = 1, x[sa[0]] = 0, i = 1; i < n; i++)
    x[sa[i]] = cmp(y, sa[i - 1], sa[i], k) ? p - 1 : p++;
}
//+++++3.5.3.2.cpp+++++

```

此外, 后缀数组还可以通过 Kärkkäinen 和 Sanders 提出的 Skew/DC3 (Difference Cover modulo 3) 算法在 $O(n)$ 的时间复杂度内构造完毕^[31], 不过该种方法编码难度相对于倍增法要高, 若题目对运行时间要求较为宽松, 使用倍增法更为“经济”。

3.5.4 最长公共子串

子串 (substring) 定义为给定字符串中的任意非空连续字符, 如给定字符串“mississippi”, 则“m”、“iss”、“ppi”都是其子串, 但“mippi”不是其子串, 因为在原字符串中并不连续。两个字符串的最长公共子串 (Longest Common Substring, LCS) 是指同时属于两个字符串且长度最大的子串。使用朴素的穷尽算法可以在 $O(mn)$ 的时间内确定两个字符串的最长公共子串, 其中 m 和 n 分别为两个字符串的长度。显然, 这样做的效率不高。要高效地求解两个字符串的最长公共子串, 需要另辟蹊径——可以从单个字符串的所有后缀间的最长公共前缀着手, 解决这一问题。

前缀 (prefix) 定义为从字符串初始位置开始的子串。例如给定字符串“mississippi”, 其子串“mi”、“miss”都是其前缀, 因为它们都是从位置 0 开始的子串, 但是子串“ssi”不是前缀, 因为它并不是从位置 0 开始的子串, 而是从位置 2 (或 5) 开始的子串。给定两个字符串 S_1 和 S_2 , 它们的公共前缀 cp 是一个字符串, 且 cp 同时是 S_1 和 S_2 的前缀, 对于所有这样的公共前缀 cp , 其中长度最大的称为最长公共前缀 (Longest Common Prefix, LCP)。

利用后缀数组可以高效地计算两个字符串的最长公共子串。读者可能会产生疑问: 后缀数组得到的是单一字符串后缀的排序, 它是如何与两个字符串之间的最长公共子串发生关联的呢? 设 s_i 表示字符串 s 从起始位置 i 开始的后缀, 字符串 s 的长度为 n 。这里先定义一个“辅助”数组 $height$, $height[i]$ 表示的是后缀数组中相邻两个后缀 $s_{sa[i-1]}$ 和 $s_{sa[i]}$ 的最长公共前缀, $1 \leq i \leq n-1$, 对于不一定相邻的两个后缀 $s_{sa[x]}$ 和 $s_{sa[y]}$ (不失一般性, 设 $0 \leq x < y \leq n-1$, 按字典序有 $s_{sa[x]} < s_{sa[y]}$), 两者的最长公共前缀可以由连续相邻后缀间的最长公共前缀表示^I, 即

$$LCP[s_{sa[x]}, s_{sa[y]}] = \min_{k \in [x, y-1]} \{LCP[s_{sa[k]}, s_{sa[k+1]}]\} = \min_{k \in [x+1, y]} \{height[k]\} \quad (3.8)$$

也就是说, $s_{sa[x]}$ 与 $s_{sa[y]}$ 的最长公共前缀就是 $height[x+1], height[x+2], \dots, height[y]$ 中的最小值。根据上述 LCP 的有关结论, (3.8) 式还可以得到一个推论, 即

$$LCP[s_i, s_k] \leq LCP[s_j, s_k], \quad 0 < i \leq j < k < n \quad (3.9)$$

请读者注意这个推论, 该推论在后续证明有关 $height$ 数组的性质时会予以应用。

^I 此处给出的有关 LCP 的性质可通过数学归纳法予以证明。受篇幅所限, 相关证明过程从略, 感兴趣的读者可以查阅后缀数组的相关资料以进一步了解具体的证明过程。

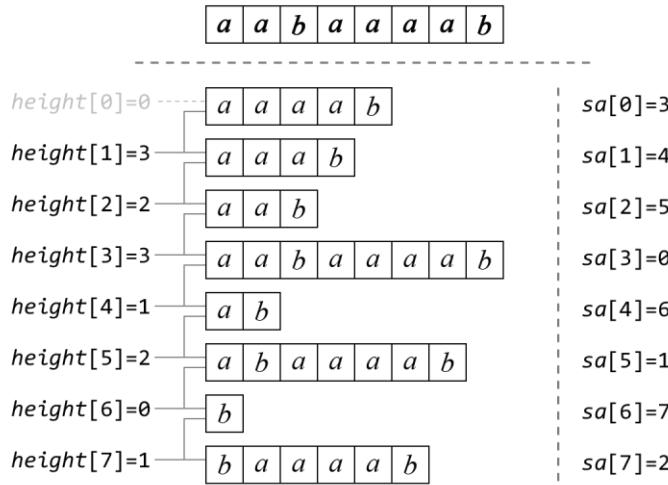


图 3-22 字符串 $s = "aabaaaab"$ 所对应的后缀数组 sa 和 $height$ 数组。示例: $s_{sa[0]} = "aaaab"$, $s_{sa[3]} = "aabaaaab"$, 则 $LCP[s_{sa[0]}, s_{sa[3]}] = \min\{height[1], height[2], height[3]\} = 2$

但是直接使用等式 (3.8) 进行后缀间的 LCP 计算显然费时较多, 能不能对其进行优化以提高效率呢? 答案是肯定的。这里需要应用一个结论: 令 $H[i] = height[rank[i]]$, 有

$$H[i] \geq H[i-1] - 1 \equiv height[rank[i]] \geq height[rank[i-1]] - 1, \quad i \geq 1 \quad (3.10)$$

不等式 (3.10) 是高效计算 $height$ 数组的基础, 大多数有关后缀数组的资料对于此不等式的证明, 不是一笔带过, 就是语焉不详, 没有完整的逻辑推理过程, 由于理解不等式 (3.10) 的证明会增进对 $height$ 数组性质的理解, 有助于理解算法的代码实现, 因此下面给出简要证明^[32]。

为了证明不等式 (3.10), 先给出有关最长公共前缀的两个结论。令字符串 s 的长度为 $n > 0$, $i < n$ 且 $j < n$, 若 s_i 和 s_j 满足 $LCP[s_i, s_j] \geq 1$, 则有以下两个结论:

(a) 若 $s_i < s_j$, 必定可推导出 $s_{i+1} < s_{j+1}$, 即 $s_i < s_j$ 等价于 $s_{i+1} < s_{j+1}$ 。证明: $LCP[s_i, s_j] \geq 1$ 说明 s_i 和 s_j 至少第一个字符是相同的, 不妨设其为 α , 则有 $s_i = \alpha s_{i+1}$, $s_j = \alpha s_{j+1}$, 当比较 s_i 和 s_j 时, 其首字符均为 α , 则后续的比较相当于比较 s_{i+1} 和 s_{j+1} , 因此结论 (a) 成立。

(b) $LCP[s_{i+1}, s_{j+1}] = LCP[s_i, s_j] - 1$ 。证明: 由于 $LCP[s_i, s_j] \geq 1$, 则说明 s_i 和 s_j 至少第一个字符是相同的, 当去掉 s_i 和 s_j 的首字符后, 两者的 LCP 即为 $LCP[s_{i+1}, s_{j+1}]$, 因此结论 (b) 成立。

根据结论 (a) 和 (b) 可以证明前述不等式。分两种情况, 若 $H[i-1] \leq 1$, 很显然不等式成立, 因为对于所有 i , 必定有 $H[i] \geq 0 \geq H[i-1] - 1$; 若 $H[i-1] > 1$, 亦即 $height[rank[i-1]] > 1$, 可知 $rank[i-1] > 0$ (因为按照 $height$ 数组的定义, $height[0] = 0$, 而 $height[rank[i-1]] \neq 0$, 得知 $rank[i-1] > 0$), 令 $k = sa[rank[i-1]-1]$, 根据后缀数组的定义有 $s_k < s_{i-1}$, 根据 $H[i-1] = LCP[s_k, s_{i-1}] > 1$ 和 $s_k < s_{i-1}$, 结合前述的结论 (b) 有 $LCP[s_{k+1}, s_i] = LCP[s_k, s_{i-1}] - 1 = H[i-1] - 1$, 又由结论 (a) 可得 $rank[k+1] < rank[i]$, 亦即 $rank[k+1] \leq rank[i] - 1$, 根据 LCP 所具有的性质不等式 (3.9) 有

$$\begin{aligned} H[i] &= LCP[s_{rank[i]-1}, s_{rank[i]}] \\ &\geq LCP[s_{rank[k+1]}, s_{rank[i]}] \\ &= LCP[s_{k+1}, s_i] \\ &= H[i-1] - 1 \end{aligned} \quad (3.11)$$

最终, 可以根据不等式 (3.11), 按照 $H[1], H[2], \dots, H[n-1]$ 的顺序计算 $height$ 数组 (需要注意, 在具体计算时, $height[1], height[2], \dots, height[n-1]$ 并不是按顺序获得的), 从而将时间复杂度控制在 $O(n)$ 。

```
//++++++3.5.4.cpp++++++//  
const int MAXN = 1000010;  
  
// 由后缀数组计算 height 数组。  
void getHeight(int *s, int *sa, int *height, int n) {  
    static int ranks[MAXN] = {};  
    height[0] = 0;  
    for (int i = 0; i < n; i++) ranks[sa[i]] = i;  
    for (int i = 0, k = 0; i < n; i++, (k ? k-- : 0)) {  
        // 由于需要顺次比较字符串, 为了能够正确得到结果, 需要在原始的字符串表示的末尾  
        // 添加一个原字符串中不存在的字符, 常见是添加 0 作为末尾元素, 这样可以使得比较  
        // 能够正确终止。  
        if (ranks[i]) while (s[i + k] == s[sa[ranks[i] - 1] + k]) k++;  
        height[ranks[i]] = k;  
    }  
}
```

在完成 $height$ 数组的构建后，可以使用稀疏表来实现 RMQ，从而快速完成两个后缀间的 LCP 查询^I。

```

int log2t[MAXN], st[20][MAXN];

// 构建 RMQ。
void prepare(int n) {
    log2t[1] = 0;
    for (int i = 2; i <= n; i++) log2t[i] = log2t[i >> 1] + 1;
    for (int i = 0; i < n; i++) st[i][0] = i;
    for (int j = 1; j <= log2t[n]; j++) {
        for (int i = 0; i + (1 << j) <= n; i++) {
            int L = st[i][j - 1], R = st[i + (1 << (j - 1))][j - 1];
            // 稀疏表中存储的是 height 数组中具有较小值元素的序号而不是数组元素的值。
            st[i][j] = (height[L] < height[R] ? L : R);
        }
    }
}

// 返回具有较小 height 值的元素序号。
int query(int L, int R) {
    int j = log2t[R - L + 1];
    L = st[L][j], R = st[R - (1 << j) + 1][j];
    return (height[L] < height[R] ? L : R);
}

// 查询后缀 s_L 和 s_R 的最长公共前缀。
int lcp(int L, int R) {
    L = rank[L], R = rank[R];
    if (L > R) swap(L, R);
    return height[query(L + 1, R)];
}
//+++++++++++++3.5.4.cpp+++++/

```

¹ 关于稀疏表, 请读者参阅本书第2章“数据结构”中第2.11.5小节“稀疏表”的内容。

最后, 两个字符串之间的最长公共子串问题可以按照如下方法转换为同一个字符串后缀之间的最长公共前缀查询问题: 令给定的两个字符串为 S_1 和 S_2 , 使用一个在 S_1 和 S_2 中不存在的字符将两个字符串连接 (通常情况下, 问题求解所涉及的为英文字母字符, 可选择 ASCII 中码值为 1 至 10 的非打印字符作为分隔符), 此处假设 S_1 和 S_2 中不存在字符 ‘\$’, 使用字符 ‘\$’ 连接 S_1 和 S_2 , 对 $S_1\$S_2$ 构建后缀数组, 并得到后缀间的最长公共前缀数组 $height$, 那么通过遍历 $height$ 数组即可获得两个字符串的最长公共前缀长度。为什么这样做可行呢? 下面以一个具体的例子予以说明。设 $S_1=\text{mississippi}$, $S_2=\text{sister}$, 将两者使用字符 ‘\$’ 进行连接得到 $S_3=\text{mississippi\$sister}$, 对 S_3 构建后缀数组并求得所有后缀间的最长公共前缀, 容易推知 S_1 和 S_2 的最长公共子串必定存在于 S_3 的 $height$ 数组中, 但不一定是 $height$ 数组中的最大值, 因为最大值可能是位于分隔符之前的同一个字符串中, 例如 S_3 的后缀 “ississippi\\$sister” 和 “issippi\\$sister” 具有最长的公共前缀 “issi”, 但是 “issi” 属于 S_1 , 不属于 S_2 , 因此不构成 S_1 和 S_2 的最长公共子串。因此, 还需满足这样一个条件: 两个后缀的起始位置必须是一个位于选取的分隔符之前, 一个位于分隔符之后。选取满足上述条件的 $height[i]$ 值, 取其最大值即为所求。

强化练习: [760 DNA Sequencing^B](#), [11107 Life Forms^C](#)。

扩展练习: [1254* Top 10^D](#), [10526* Intellectual Property^D](#)。

3.5.5 最长重复子串

利用后缀数组可以高效地查找字符串中的最长重复子串 (Longest Repeating Substring, LRS)。如果对重复子串没有限制, 即两个重复子串可以重叠, 则只需求原字符串的后缀数组, 然后计算 $height$ 数组, 取 $height[i]$ 的最大值即可。因为在此种情况下, 求最长重复子串等价于求两个后缀的最长公共前缀的最大值, 而任意两个后缀的最长公共前缀都可以通过 $height$ 数组获得, 其结果是 $height$ 数组某个区间内值的最小值, 因此, 最长重复子串的长度等价于 $height$ 数组的最大值。

如果加以限制, 规定两个重复的子串不能重叠, 应该如何处理呢? 可以使用二分搜索, 将问题转化为可行性判定问题, 即判定是否存在两个长度为 k 的相同子串且不重叠。可以利用 $height$ 数组将排序后的后缀分成若干组 (同组后缀在后缀数组中是连续的), 在每组的后缀中, 后缀之间的 $height$ 值都不小于 k , 然后对于每组后缀, 判断每个后缀在后缀数组 sa 中的值其最大值和最小值之间的差是否不小于 k , 只要有一组满足条件, 则说明存在两个后缀, 其最长公共前缀长度不小于 k , 且起始位置相差不小于 k , 这正好符合判定条件。

强化练习: [1223 Editor^D](#), [11512 GATTACA^B](#)。

3.5.6 Burrows-Wheeler 变换

Burrows-Wheeler 变换 (Burrows-Wheeler Transform, BWT), 又称为块排序压缩 (block-sorting compression), 是一种将字符串进行特定处理后以便更为高效地对数据进行压缩的编码算法, 由 Michael Burrows 和 David Wheeler 于 1994 年发明^[33]。

当使用该算法对字符串进行转换时, 算法只改变该字符串中字符的顺序而并不改变其内容。如果原字符串有多个重复的子串, 那么经过转换的字符串中就会包含若干连续重复的字符, 这对提高压缩效率很有帮助。该算法能够使得基于处理字符串中连续重复字符的技术 (如 MTF 变换和游程编码) 的编码更容易被压缩。例如, 当给定如下输入:

SIX.MIXED.PIXIES.SIFT.SIXTY.PIXIE.DUST.BOXES

经过 BWT 之后，可以得到如下输出：

```
TEXYDST.E.IXIXIXXSSMPPS.B..E.S.EUSFXDIIIOIIIT
```

可以看到，原字符串不包含任何连续的字符，而经过转换的字符串包含六个同等字符游程 (run): XX, SS, PP, …, II, III，因而更容易被压缩。

BWT 通过将原字符串 s 不断进行循环移位 (cyclic shift) 而得到原字符串 s 的全排列 P 的一个子集 P' ，对子集 P' 中包含的字符串按字典序 (lexicographic order) 进行排序，取排序得到的字符串的最后一个字符从而得到 BWT 编码。以字符串 “^BANANA|” 为例，选择通过循环左移得到所有字符串（通过循环右移具有相同效果）：

```
^BANANA|
BANANA|^
ANANA|^B
NANA|^BA
ANA|^BAN
NA|^BANA
A|^BANAN
|^BANANA
```

对其按字典序进行排序可得：

```
ANANA|^B
ANA|^BAN
A|^BANAN
BANANA|^
NANA|^BA
NA|^BANA
^BANANA|
|^BANANA
```

取排序后所有字符串的最后一个字符可得：

```
BNN^AA|A
```

如果从 0 开始计数，原始字符串 “^BANANA|” 在排序后字符串中的序号为 6，那么字符串 “^BANANA|” 的 BWT 编码可以表示为 [BNN^AA|A, 6]。

BWT 的“奇妙”之处在于给定编码后的字符串和原字符串在循环移位排序后的位置，可以通过一个简单的方法确定原始的字符串，这个过程称之为 Burrows-Wheeler 逆变换 (Inverse Burrows-Wheeler Transform, IBWT)。理解 IBWT 的关键在于：字符串循环移位得到的字符矩阵中，每一列均包含原字符串中的所有字符一次且仅一次。将 BWT 编码进行排序后即可得到循环移位后字符串矩阵的第一列，得到第一列之后，在其前附加 BWT 编码，再次进行排序，则可以得到第二列，如此重复，最终可以得到原字符串的所有循环移位，根据原始字符串在循环移位字符串矩阵中的序号即可确定原始字符串。

输入：[BNN^AA|A, 6]

附加 1:	排序 1	附加 2	排序 2
B	A	BA	AN
N	A	NA	AN
N	A	NA	A

\wedge	B	$\wedge B$	BA
A	N	$\wedge N$	NA
A	N	$\wedge N$	NA
	\wedge	\wedge	$\wedge B$
A		A	$\wedge $
附加 3 排序 3 附加 4 排序 4			
BAN	ANA	$\wedge BAN$	ANAN
NAN	ANA	$\wedge NAN$	ANA
NA	$\wedge $	$\wedge A$	A $\wedge B$
$\wedge BA$	BAN	$\wedge BAN$	BANA
ANA	NAN	$\wedge NAN$	NANA
ANA	NA	$\wedge NA $	NA $\wedge $
$\wedge B$	$\wedge BA$	$\wedge BA$	$\wedge BAN$
A $\wedge $	$\wedge B$	A $\wedge B$	$\wedge BA$
附加 5 排序 5 附加 6 排序 6			
BANAN	ANANA	$\wedge BANANA$	ANANA
NANA	$\wedge $	$\wedge NAN$	ANA $\wedge B$
NA $\wedge B$	A $\wedge BA$	$\wedge BA$	A $\wedge BAN$
$\wedge BANA$	BANAN	$\wedge BANAN$	BANANA
ANANA	NANA	$\wedge NAN$	NANA \wedge
ANA $\wedge $	$\wedge B$	$\wedge A$	NA $\wedge BA$
$\wedge BAN$	$\wedge BANA$	$\wedge BANA$	$\wedge BANAN$
A $\wedge BA$	$\wedge BAN$	A $\wedge BAN$	$\wedge BANA$
附加 7 排序 7 附加 8 排序 8			
BANANA	ANANA \wedge	$\wedge BANANA $	ANANA $\wedge B$
NANA $\wedge B$	$\wedge $	$\wedge NAN$	ANA $\wedge BAN$
NA $\wedge BAN$	A $\wedge BANA$	$\wedge BANA$	A $\wedge BANAN$
$\wedge BANANA$	BANANA	$\wedge BANANA $	BANANA \wedge
ANANA \wedge	$\wedge B$	$\wedge NAN$	NANA $\wedge BA$
ANA $\wedge BA$	$\wedge $	$\wedge A$	NA $\wedge BANA$
$\wedge BANAN$	$\wedge BANA$	$\wedge BANA$	$\wedge BANANA $
A $\wedge BANA$	$\wedge BAN$	A $\wedge BAN$	$\wedge BANANA$

输出: $\wedge BANANA|$

根据原始字符串的序号 6 即可确定原始字符串为 “ $\wedge BANANA|$ ”。

强化练习: 741 Burrows Wheeler Decoder^C。

3.6 正则表达式

正则表达式 (regular expression) 是为了方便字符串的处理而发展的工具, 它通过定义一系列的规则来匹配特定的目标字符串^[34]。在 GCC 5.3.0 版本的编译器中, 对正则表达式已经有了较为完善的支持。

3.6.1 元字符

在匹配过程中, 为了区分匹配规则和要匹配的字符, 定义了一些特殊的字符, 称为元字符, 它们的作用是规定特定的匹配模式。以下是常用的元字符及其含义。

.	匹配除行结束符 (LF, CR, LS, PS) 以外的任意字符
\t	匹配水平制表符

\n	匹配换行符
\v	垂直制表符
\f	换页符
\r	回车符
\d	数字字符
\D	非数字字符
\s	空白字符
\S	非空白字符
\w	字母字符
\W	非字母字符

3.6.2 转义字符

有时需要在匹配过程中匹配一些特殊字符，例如需要匹配字符 ‘\’，但是该字符已经作为定义匹配模式的字符使用，为了解决这样的问题，可以使用转义字符将匹配模式中使用的特殊字符“转回”它们的本义。例如 ‘\\’ 表示将定义元字符的 ‘\’ 解释成普通的右斜杠， ‘\\d’ 的意思是匹配一个右斜杠和一个小写的字母 d。

由于 C++ 中已经将字符 ‘\’ 作为转义字符使用，所以在正则表达式中，如果要转义一个右斜杠字符，使正则表达式引擎将其解释为一个右斜杠，可以使用以下方式：

```
string pattern = "\\\";
```

或者使用 C++11 支持的原生字符串标识 (raw string) 来表示正则表达式。在原生字符串标识中，右斜杠被解释为其原义，而不被当做转义字符看待，例如：

```
string pattern = R"(\\"");
```

表示匹配两个右斜杠字符。

3.6.3 数量匹配符和分组

如果需要匹配特定数量的字符，可以使用数量匹配符。

*	指定的模式匹配零次或多次
+	指定的模式匹配至少一次
?	指定的模式匹配零次或一次
{int}	指定的模式匹配特定的次数，次数由 int 指定
{int,}	指定的模式匹配次数至少为 int 所指定的次数
{min,max}	指定的模式匹配次数在 min 和 max 指定的范围之间（包括 min 和 max）

根据需要可以将整个匹配模式分解为“子模式”，这样方便在后续过程中进行引用，此种使用方法称为分组 (group)。分组使用符号 “()” 进行，比如，为了将多个 C 类 IP 地址的第三位 IP 统一进行更改，可以使用以下匹配模式：

```
string pattern = R"((\d+)\.(\d+)\.(\d+)\.(\d+))";
```

所得到的子匹配会按照顺序从 1 开始编号，这样在后续使用 `regex_replace` 进行替换时，引用 “\$1” 对应的是 IP 地址的首位数字，引用 “\$2” 对应的是 IP 地址的第二位，依此类推。

3.6.4 字符类和可选模式

为了便于表示需要匹配的字符，正则表达式提供了字符类，它可以将多个需要匹配的字符进行归类合并。

字符类使用一对方括号 ‘[]’ 来表示，例如：

[acopz]	匹配字符 a 或 c 或 o 或 p 或 z
[^abc]	使用符号^表示匹配除 a、b、c 的其他任意字符
[a-zA-Z0-9]	使用范围表示符号 ‘-’，表示匹配 a 至 z 的小写字母或 0 至 9 的数字字符

如果有多个匹配模式可选，可使用 ‘|’ 进行分隔，称为可选模式。默认可选模式下进行贪婪匹配，即按照可选模式从前到后的顺序进行匹配，若需要指定非贪婪模式，在匹配模式后增加数量匹配符 ‘?’ 即可。

需要注意，在匹配过程中，元字符 ‘\w’ 等价于 ‘[a-zA-Z0-9]+’，也就是说会将数字视为字母字符，如果需要匹配纯英文单词，应该使用 ‘[a-zA-Z]+’。

3.6.5 断言

在匹配过程中，为了表示满足特定条件的匹配表达式，例如需要匹配一个浮点数，该浮点数从字符串的第一个字符开始，到最后一个字符结束，前述的匹配模式定义只有关于数量和字符类型的表示，但是如何表示“从第一个字符开始到最后一个字符结束”这个条件呢？为了解决这一问题，正则表达式提供了断言 (assertions)，以下是常用的断言：

^	表示需要匹配的字符串的起始位置
\$	表示需要匹配的字符串的结束位置
\b	表示单词边界，即前一个字符为字母字符，当前字符为非字母字符
\B	表示非单词边界，前一个和当前字符均为字母字符或均不是字母字符

例如，匹配一个浮点数（不包括指数部分）的正则表达式为（可以具有前导 0）：

```
string pattern = R"(^[+-]?( [0-9]*\.\?[0-9]+) $)";
```

3.6.6 正则表达式类

在 GCC C++ 中使用正则表达式需要包含头文件 `<regex>`。使用正则表达式，一般是声明一个 `string` 类变量，将匹配模式以原生字符串表示，然后使用 `regex` 类提供的匹配函数 `regex_match` 进行匹配。`regex_match` 的功能是检查匹配模式能否与整个字符串相匹配。以下代码示例的是通过 `regex_match` 对浮点数进行匹配。注意：正则表达式只有在运行时才予以构造，而不是编译时构造，所以是一个费时的操作，如果需要进行多次匹配，应该尽量将正则表达式放在循环之外，否则每次循环都构造一次，对运行效率有较大影响。

```
//+++++3.6.6.cpp+++++
void match() {
    string float1 = " -5.236e-12 ", float2 = "6.e+12";
    string pattern = R"(^ \s* [\+|-] ? [1-9] \d* \. \d+ [\+|-] ? [1-9] \d* \s* $)";
    regex e(pattern, regex_constants::icase);
    cout << (regex_match(float1, e) ? "Matched." : "Unmatched.") << endl;
    cout << (regex_match(float2, e) ? "Matched." : "Unmatched.") << endl;
}
```

输出为：

```
Matched.
Unmatched.
```

如果需要匹配的是给定字符串中的子串，可以使用 `regex` 类中的 `regex_search`。`regex_search`

的功能是搜索给定字符序列中是否存在与正则表达式相匹配的子串，如果存在，则返回第一个匹配的子串。如果需要找出字符串中所有符合匹配模式的子串，可以使用 `regex_iterator` 迭代器类。

```
void search() {
    string ip = "192.168.1.100, 192.168.1.200, 192.168.1.300.";
    string pattern = R"((\d+)(\.\d+){3})";
    regex e(pattern);
    cout << "First matched:";
    smatch sm;
    if (regex_search(ip, sm, e)) cout << " " << sm[0].str();
    cout << endl;
    cout << "All matched:";
    regex_iterator<string::iterator> it(ip.begin(), ip.end(), e);
    regex_iterator<string::iterator> end;
    while (it != end) {
        cout << " " << it->str();
        it++;
    }
    cout << endl;
}
```

输出为：

```
First matched: 192.168.1.100
All matched: 192.168.1.100 192.168.1.200 192.168.1.300
```

若需要将字符串中指定模式的字符串替换成其他子串，可以使用 `regex` 类中的 `regex_replace`。
`regex_replace` 的功能是将给定字符序列中与正则表达式匹配的子串替换成指定的子串。

```
void replace() {
    string ip = "192.168.1.100, 192.168.1.200, 192.168.1.300.";
    string pattern = R"((\d+)\.(\d+)\.(\d+)\.(\d+))";
    regex e(pattern);
    cout << "Replaced: " << regex_replace(ip, e, "$1.$2.$2.$4") << endl;
}
//++++++++++++++3.6.6.cpp+++++++/
```

输出为：

```
Replaced: 192.168.2.100, 192.168.2.200, 192.168.2.300.
```

强化练习：[325 Identifying Legal Pascal Real Constants^B](#)，[494 Kindergarten Counting Game^A](#)，[11148 Moliu Fractions^D](#)。

3.7 算法库函数

3.7.1 lexicographical_compare

按字典序比较两个字符串的大小关系，常用于对字符串进行排序，当函数返回 `true` 时表示按字典序前一字符串小于后一字符串，返回 `false` 表示前一字符串大于或等于后一字符串。字典序比较是指按字符的 ASCII 大小进行比较，因此 ‘A’ (ASCII 为 65) 小于 ‘a’ (ASCII 为 97)。其函数声明为：

```
// 使用默认比较函数的版本。
template <class InputIterator1, class InputIterator2>
```

```

bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2);

// 使用自定义比较函数的版本。
template <class InputIterator1, class InputIterator2, class Compare>
bool lexicographical_compare(InputIterator1 first1, InputIterator1 last1,
InputIterator2 first2, InputIterator2 last2, Compare comp);

```

`lexicographical_compare` 在比较时，从前到后逐个字符进行比较，直到某个字符串的末尾。令比较的字符串为 x 和 y ，可能会出现以下几种情形：(1) 在逐个字符比较的过程中，如果 $x_i < y_i$ ，即 x 字符串的第 i 个字符的 ASCII 码小于 y 字符串第 i 个字符，则 $x < y$ ；(2) 在逐个字符比较的过程中，如果 $x_i > y_i$ ，即 x 字符串的第 i 个字符的 ASCII 码大于 y 字符串第 i 个字符，则 $x > y$ ；(3) 如果 x 和 y 的长度相等且所有对应字符相同，则 $x = y$ ；(4) 如果字符串 x 较字符串 y 短，且 x 为 y 的前缀，则 $x < y$ ；(4) 如果字符串 x 较字符串 y 长，且 y 为 x 的前缀，则 $x > y$ 。如果需要实现自定义版本的字典序比较，可以使用 `lexicographical_compare` 函数的第二种版本，并为其指定比较函数。例如以下代码进行字典序比较时，按字母表顺序进行，忽略大小写，即字符 ‘a’ 小于字符 ‘B’。

```

//-----3.7.1.cpp-----
bool cmp(const char &a, const char &b) { return toupper(a) < toupper(b); }

int main(int argc, char *argv[]) {
    string s1 = "aBcD", s2 = "BcDe";
    if (lexicographical_compare(s1.begin(), s1.end(), s2.begin(), s2.end(), cmp))
        cout << "s1 is less than s2.\n";
    else
        cout << "s1 is greater than s2.\n";
    return 0;
}
//-----3.7.1.cpp-----

```

输出为：

```
s1 is less than s2.
```

3.7.2 `next_permutation` 和 `prev_permutation`

在 UVa OJ 的题目中，经常需要处理字符串或其他类型序列的排列问题。可以通过使用 `next_permutation` 和 `prev_permutation` 这两个函数生成相应序列的所有排列。其函数声明为：

```

template <class BidirectionalIterator>
bool next_permutation(BidirectionalIterator first, BidirectionalIterator last);

template <class BidirectionalIterator>
bool prev_permutation(BidirectionalIterator first, BidirectionalIterator last);

```

`next_permutation` 的作用是生成按字典序的后一个排列，如果已经是最后一个排列，则会将其反转 (`reverse`) 并返回 `false`，否则返回 `true`；`prev_permutation` 的作用是生成按字典序的前一个排列，如果已经是按字典序的第一个排列，则会将其反转并返回 `false`，否则返回 `true`。

```

//-----3.7.2.cpp-----
int main(int argc, char *argv[]) {
    string something = "abc";

```

```

cout << something << endl;
while (next_permutation(something.begin(), something.end()))
    cout << something << endl;
cout << endl;
cout << "after next_permutation: " << something << endl;
cout << endl;
something.assign("cba");
while (prev_permutation(something.begin(), something.end()))
    cout << something << endl;
cout << something << endl;
cout << endl;
cout << "after prev_permutation: " << something << endl;
return 0;
}
//-----3.7.2.cpp-----//

```

输出为：

```

abc
acb
bac
bca
cab
cba

after next_permutation: abc

cab
bca
bac
acb
abc
cba

after prev_permutation: cba

```

如果使用带两个参数的函数形式，默认使用小于运算符(<)对序列中的两个元素进行大小的比较。如果需要改变排列的生成方式，可以使用带比较器的函数形式。

在SGI的库实现中，`next_permutation`的实现代码如下：

```

template<typename _BidirectionalIterator>
bool next_permutation
(_BidirectionalIterator __first, _BidirectionalIterator __last) {
    // 检查指定的元素范围是否为空。
    if (__first == __last) return false;
    // 检查指定的范围是否只有一个元素。
    _BidirectionalIterator __i = __first;
    ++__i;
    if (__i == __last) return false;
    // 从指定范围的前一个元素开始搜索。
    __i = __last;
    --__i;
    for (;;) {
        // 逐个和当前元素比较，直到找到比当前元素小的某个元素。
        _BidirectionalIterator __ii = __i;

```

```

-- __i;
if (*__i < *__ii) {
    BidirectionalIterator __j = __last;
    while (!(*__i < *__j)) {}
    std::iter_swap(__i, __j);
    std::reverse(__ii, __last);
    return true;
}
// 如果未找到满足条件的元素，则表明此排列已经是最后一个排列。
if (__i == __first) {
    std::reverse(__first, __last);
    return false;
}
}
}

```

算法从最尾端开始寻找最先出现的两个相邻的元素——第一个元素是 $*i$, 第二个元素是 $*ii$, 满足 $*i < *ii$ 。找到这样一对元素后, 再从尾端开始往前选择第一个大于 $*i$ 的元素 $*j$, 将元素 $*i$ 和 $*j$ 对调, 然后将 $*ii$ 之后的所有元素反转, 就得到了下一个组合^I。

SGI 标准库实现中的 `next_permutation` 算法很巧妙, 似乎像变魔术般生成了下一个排列, 为什么这样做可行呢?

算法首先从后往前寻找最先出现的一对相邻元素, 该对元素满足第一个元素小于第二个元素的性质。这个容易理解, 因为全排列就是把序列从完全顺序排列一步步转换到完全的逆序排列, 如果找到了这样的一对相邻元素, 那么这是目前从后往前首次出现的顺序元素对, 所以肯定在此处着手进行变换。关键是理解为何把第 i 个元素与从后往前第一个大于它的元素交换, 并且颠倒原序列区间 $[i+1, last)$ 内元素就可以得到下个组合。根据代码可以知道区间 $[first, i)$ 这部分元素与原来的序列相同, 把第 i 个元素与从后往前第一个大于它的第 j 个元素交换, 则不管后面如何排列, 由于 $*j > *i$, 得到的新排列肯定大于原来的排列, 下面需要说明的就是如何保证这个新排列恰是原排列的下一个, 而不是更后面的排列。

$*i$ 和 $*ii$ 是从后往前第一个顺序的相邻元素对, 这就可以肯定区间 $[ii, last)$ 这部分数据是完全逆序的, 有 $*j > *i \geq *j > *i + 1$, 而 $*j$ 是从后往前第一个大于 $*i$ 的元素, 故 $*j > *i \geq *j > *i + 1$, 所以 $*i$ 与 $*j$ 互换后区间 $[ii, last)$ 这部分数据仍然是完全逆序的, 既然如此, 把这部分序列反转一下, 就得到了完全顺序的序列, 在排列组合中, 这是最小的情形, 因此可以肯定得到的新排列是原序列的下一个排列。

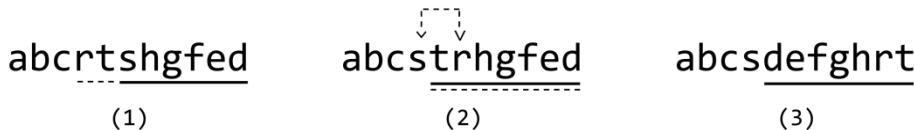


图 3-23 令当前字符串 $T = "abcrtshgfed"$, 需要找到其下一变换。(1) 从后往前找到如虚线所示的第一对相邻的顺序元素 “rt”, 易知实线标识的部分字符串为完全逆序; (2) 从字符串末尾往前寻找, 找到第一个大于字符 ‘r’ 的字符 ‘s’, 将两者位置互换, 然后将从字符 ‘t’ 开始的部分字符反转; (3) 易知字符串 $T' = "abc...defghrt"$ 为字符串 T 的下一个排列

^I 为表示的简便, 在讨论中略去了实现代码中变量前的两个连续的下划线字符, 即变量 i 对应代码中的变量 $__i$, 变量 j 对应代码中的变量 $__j$, 依此类推。

在理解了 `next_permutation` 实现的基础上，理解 `prev_permutation` 的实现就相对容易得多。以下给出其源代码，请读者自行“揣摩”并理解其实现。

```
template<typename _BidirectionalIterator>
bool prev_permutation
(_BidirectionalIterator __first, _BidirectionalIterator __last) {
    if (__first == __last) return false;
    _BidirectionalIterator __i = __first;
    ++__i;
    if (__i == __last) return false;
    __i = __last;
    --__i;
    for (;;) {
        // 逐个和当前元素比较，直到找到比当前元素大的某个元素。
        _BidirectionalIterator __ii = __i;
        --__i;
        if (*__ii < *__i) {
            _BidirectionalIterator __j = __last;
            while (!(*__j < *__i)) {}
            std::iter_swap(__i, __j);
            std::reverse(__ii, __last);
            return true;
        }
        // 如果未找到满足条件的元素，则表明此排列已经是最初一个排列，反转得到最大排列。
        if (__i == __first) {
            std::reverse(__first, __last);
            return false;
        }
    }
}
```

146 ID Codes^A（身份编码）

2084年，尽管是在一个世纪之后，老大哥^I——最终还是来了。为了彰显他对民众的强大掌控能力，同时也为了维持法律和秩序的长期稳定，政府决定采取一项激进的措施——在所有市民的左手腕植入一枚微型芯片。微型芯片包含此人的所有相关信息，并且能够通过发射器将携带者的位置信息发送给中央控制计算机（政府也乐见此举将会减少整形外科医生的失业人数）。

微型芯片的一个必要组件是唯一的标识码，它由小写字母组成，最长不超过50个字符。任意给定的标识码中的字母都是随机选择的。由于植入芯片复杂，为了方便，制造商先选择一组字母，在将这组字母所有可能的排列用完之前，不会选另外一组字母从头开始产生编码。

例如，如果编码中需要包含三个“a”，两个“b”，一个“c”，满足条件的编码共有60个，将它们按字典序从小到大排列，其中三个依次是：

```
abaabc
abaacb
ababac
```

写一个程序来帮助制造商生成这些编码。你的程序需要能够接受一个不超过50个字符的输入（输入中

^I Big Brother, 乔治·奥威尔的科幻小说《1984》中的人物，“老大哥”是小说中大洋国的统治者。

可能包含重复的字母), 如果其存在后继, 则输出其后继编码, 如果不存在, 则输出“**No Successor**”。

输入输出

输入的每一行包含一个表示编码的字符串。整个输入文件以只包含‘#’字符的输入行结束。

对于每一行输入, 要么输出其后继的编码, 要么输出“**No Successor**”。

样例输入

```
abaacb
cbaa
#
```

样例输出

```
ababac
No Successor
```

分析

直接应用 `next_permutation` 函数解题即可。

参考代码

```
int main(int argc, char *argv[]) {
    string line;
    while (getline(cin, line), line != "#") {
        if (next_permutation(line.begin(), line.end())) cout << line << endl;
        else cout << "No Successor" << endl;
    }
    return 0;
}
```

强化练习: 140 Bandwidth^A, 195 Anagram^A, 234 Switching Channels^D, 729 The Hamming Distance Problem^A, 1209* Wordfish^D, 10098 Generating Fast Sorted Permutation^A, 11553 Grid Game^A, 11742 Social Constraints^B, 12247 Jollo^B, 12249 Overlapping Scenes^D。

3.7.3 replace

将序列容器中指定范围内的特定值用新值予以替换。函数原型为:

```
template <class ForwardIterator, class T>
void replace(ForwardIterator first, ForwardIterator last, const T& old_value, const
T& new_value);
```

此函数可以方便的完成字符串替换功能, 但是无法通过该函数将特定字符从序列中删除。

3.7.4 reverse

`reverse` 是将序列中指定范围的元素顺序反转, 即将其逆序。其函数声明为:

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last);
```

在对字符串执行逆序输出时, 此函数非常方便。应用于容器类时参数需要使用迭代器, 对于数组则使用数组地址范围。

```
//-----3.7.4.cpp-----//
int main(int argc, char *argv[]) {
    int number[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    reverse(number, number + 10);
    // number[10] = { 9, 8, 7, 6, 5, 4, 3, 2, 1, 0 }
```

```

        string line = "0123456789";
        reverse(line.begin(), line.end());
        // line = "9876543210"
        return 0;
    }
//-----3.7.4.cpp-----//

```

强化练习：483 Word Scramble^A，11192 Group Reverse^A。

3.7.5 transform

transform 的作用是将序列指定范围内的元素进行某种变换，其函数声明为：

```

template <class InputIterator, class OutputIterator, class UnaryOperation>
OutputIterator transform(InputIterator first1, InputIterator last1, OutputIterator
result, UnaryOperation op);

```

经常用来将字符串的内容全部改写为大写或者小写。

```

//-----3.7.5.cpp-----//
int main(int argc, char *argv[]) {
    string s = "ThE qUiCk BrOwN fOx JuMpS oVeR a LaZy DoG.";
    // 输出原始字符串。
    cout << s << endl;
    // 将所有英文字符转换为大写。
    transform(s.begin(), s.end(), s.begin(), ::toupper);
    cout << s << endl;
    // 将所有英文字符转换为小写。
    transform(s.begin(), s.end(), s.begin(), ::tolower);
    cout << s << endl;
    return 0;
}
//-----3.7.5.cpp-----//

```

输出为：

```

ThE qUiCk BrOwN fOx JuMpS oVeR a LaZy DoG.
THE QUICK BROWN FOX JUMPS OVER A LAZY DOG.
the quick brown fox jumps over a lazy dog.

```

3.8 小结

字符串作为存储数据的一种手段，是一种重要的形式，在解题中，需要充分理解的一点是，字符串在本质上是数字和字符的映射，处理字符串就是在处理数字。掌握字符串分成两个部分，一个部分是掌握字符串的基本操作，例如连接、删除等。另外一个部分是关于字符串的算法，需要重点掌握的内容包括字符串匹配算法（KMP 匹配算法）、后缀数组、Aho-Corasick 算法，以上均是各类竞赛重点考察的内容。理解有关字符串的算法，要点仍然是理解其核心思想，因为在编程竞赛中，与字符串算法有关的题目一般不会出现非常直接的题目，一般都是将题目的底层模型加以适当隐藏，或者在算法基本思想的基础上加以扩展和变形，使得解题难度增加。由于字符串算法的特点是核心思想简单但理解困难，解题者在平时需要反复阅读、琢磨算法实现，在计算机上多次复习代码实现，做到信手拈来、胸有成竹。

第4章 排序与查找

天地浑沌如鸡子，盘古生其中。万八千岁，天地开辟，阳清为天，阴浊为地。
——徐整^I，《三五历记》

排序与查找在计算机科学中有着非常重要的地位。作为非底层的程序员，可能并不需要去实现某种具体的排序算法，因为常用的编程语言已经提供了相应的排序函数，只需要调用即可。但是不能因此对排序算法予以忽视，因为排序算法包含了很多“营养”，熟悉各种排序算法的基本思想和具体实现有助于提高自身的编程水平和思维层次。

排序是将元素序列按照指定的大小规则进行排列以使得元素有序的过程。排序有多种方法，每种方法的思想不尽相同。根据排序的性质，可以对其进行以下区分：

- **内部排序与外部排序。** 内部排序（internal sorting）是指待排序序列完全存放在内存中进行排序的过程，适合数据量较小的情形。外部排序（external sorting）指的是大文件的排序，即待排序的记录存储在外部存储器上，由于待排序文件无法一次性装入内存，需要在内存和外部存储器之间进行多次数据交换，以达到排序整个文件的目的。
- **稳定排序与不稳定排序。** 如果排序前后具有相同值的元素其相对位置关系不变，则称相应的排序为稳定排序（stable sorting），不满足这个性质的排序称为不稳定排序（unstable sorting）。注意排序算法的稳定性是相对的，有些算法根据其实现的不同可能具有不同的稳定性，如选择排序，如果采用普通的原地交换选择排序，不使用额外的存储空间，那么是不稳定的，如果使用额外的存储空间，采用直接选择排序则算法是稳定的。

4.1 交换排序

4.1.1 冒泡排序

交换排序（sorting by exchanging）是指通过不断交换未排序的“元素对”直到所有元素有序的过程，最简单易懂的是冒泡排序（bubble sort）。冒泡排序的基本思想是通过比较相邻两个元素的大小，将逆序的元素进行交换来完成排序。冒泡排序采用多趟比较来完成，如果相邻两个元素的大小为逆序关系，则予以交换，在此过程中，大的元素“下沉”，小的元素“上浮”，直到最大的元素“下沉”到最末尾的位置，这样便完成了第一趟“冒泡”，之后继续第二趟“冒泡”，将第二大的元素“下沉”到倒数第二个位置……继续此过程直到排序完毕。整个过程类似于水泡上升，因此形象地称之为冒泡排序。

当大部分数据已经有序时，可以使用一个“标记”来记录最内层循环是否有交换发生，如果某一趟没有交换发生，则表明数据已经排序完毕，不需要继续排序。

冒泡排序的时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，属于稳定排序。

```
//++++++4.1.1.cpp++++++  
void bubbleSort(int data[], int n) {
```

^I 徐整（生卒年月不详），字文操，豫章（今江西南昌）人，三国时期吴国的太常卿。据《隋书》记载撰有《毛诗谱》，注有《孝经默注》，另著有记载中国上古传说的《三五历记》及《五远历年纪》。

```

for (int i = 0; i < n; i++)
    for (int j = 0; j < (n - i - 1); j++)
        if (data[j] > data[j + 1])
            swap(data[j], data[j + 1]);
}

```

双向冒泡排序(bidirectional bubble sort)在冒泡排序的基础上进行了少量优化,其基本思想并未改变,只不过在每次“冒泡”进行到最后时,不是从头开始“冒泡”,而是从后往前将最小的元素“冒泡”到其正确位置,这样可以最大程度减少循环比较的次数,得到常数项的优化,但是总的时间复杂度仍然是 $O(n^2)$ 。

```

// 双向冒泡排序, 尽量减少循环比较的次数。
void bidirectionalBubbleSort(int data[], int n) {
    int left = 0, right = n - 1, shift;
    while(left < right) {
        // 将较大的值移到末尾。
        for(int i = left; i < right; i++)
            if(data[i] > data[i + 1]) {
                swap(data[i], data[i + 1]);
                shift = i;
            }
        right = shift;
        // 将较小的值移到开头。
        for(int i = right - 1; i >= left; i--)
            if(data[i] > data[i + 1]) {
                swap(data[i], data[i + 1]);
                shift = i + 1;
            }
        left = shift;
    }
}
//+++++4.1.1.cpp+++++

```

强化练习: 10152 ShellSort^A, [10327 Flip Sort^A](#)。

扩展练习: [120 Stacks of Flapjacks^A](#), [299 Train Swapping^A](#), 331 Mapping the Swaps^B。

4.1.2 快速排序

快速排序(quicksort)是对冒泡排序的一种改进,由霍尔^I在1962年提出的一种划分交换排序发展而来,它采用了一种分治的策略,通常称其为分治法(divide-and-conquer method)。算法基本思想是通过一趟排序将要排序的数据分割成独立的两部分,其中一部分的数据比另外一部分的数据都要小,然后再按此方法对这两部分数据分别进行快速排序。整个排序过程可以递归进行,以此达到整个数据有序的目的。

排序的第一步是要确定一个基准值(pivot),为了简便,一般选择位于区间中心的元素作为基准值,然后以此基准值将区间划分为两部分进行排序,之后递归调用。编写一个正确的快速排序并非想象中的那么容易,需要考虑许多边界情形。

快速排序的时间复杂度为 $O(n \log n)$,空间复杂度为 $O(\log n)$,属于不稳定排序。

```

//-----4.1.2.cpp-----//

```

^I 查尔斯·安东尼·理查德·霍尔(Charles Antony Richard Hoare, Tony Hoare 或 C.A.R. Hoare,, 1934—), 英国计算机学家,1980年图灵奖获得者。

```

void quickSort(int data[], int left, int right) {
    if (left < right) {
        int pivot = data[(left + right) >> 1];
        int i = left - 1, j = right + 1;
        while (i < j) {
            // 从前往后找到第一个不小于基准值的元素位置。
            do i++; while (data[i] < pivot);
            // 从后往前找到第一个不大于基准值的元素位置。
            do j--; while (data[j] > pivot);
            // 交换位置。
            if (i < j) swap(data[i], data[j]);
        }
        // 递归解决问题。
        quickSort(data, left, i - 1);
        quickSort(data, j + 1, right);
    }
}
//-----4.1.2.cpp-----//

```

在 C 和 C++ 中，可以使用库函数中的 `qsort` 来调用快速排序的功能。`qsort` 的声明为：

```

void qsort
(void* base, size_t num, size_t size, int (*compar)(const void*,const void*));

```

其中第一个参数为待排序数组起始地址，第二个参数为数组中待排序元素数量，第三个参数为单个数组元素占用空间的大小，第四个参数为指向比较函数的指针。比较函数以数组中的两个元素 a 和 b 作为参数，当函数返回大于 0 的值时，指示 `qsort` 在排序中将 a 放在 b 之后，即 $a > b$ ；如果返回值小于 0，则将 a 放在 b 之前，即 $a < b$ ；返回 0 值表示 a 和 b 相等。

```

int numbers[8] = {19, 82, 0, 6, 24, 31, 80, 2891};
int cmp(const void *a, const void *b) { return *(int *)a > *(int *)b ? 1 : -1; }
qsort(numbers, 8, sizeof(int), cmp);

```

强化练习：755 487-3279^A。

4.1.3 中位数

给定 n 个实数 a_1, a_2, \dots, a_n ，其均值（mean）为 n 个数的平均值，令其为 M ，则

$$M = \frac{a_1 + a_2 + \dots + a_n}{n}$$

中位数（median）是将这 n 个数按从小到大的顺序排列后位于“中间”位置的数。将 n 个数按序排列，取中间位置的数即为中位数，如果 n 为偶数，则取位于最中间的两个数的平均数作为中位数。中位数具有以下性质——它与 a_i 差的绝对值之和最小，即令

$$S = \sum_{i=1}^n |x - a_i|, \quad x \in \mathbb{R}$$

当 x 为这 n 个数的中位数时， S 取得最小值。在数轴上，中位数是与这 n 个数的距离之和最小的位置。如前所述，应用排序算法，将 n 个数按从小到大的顺序排序以后，若 n 为奇数，则中位数 m 为

$$m = a_{\lfloor n/2 \rfloor}$$

若 n 为偶数，则中位数 m 为

$$m = \frac{a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2 \rfloor + 1}}{2}$$

使用排序算法获得中位数的方法，其时间复杂度为 $O(n \log n)$ 。

存在时间复杂度为 $O(n)$ 的算法来确定数列的中位数。首先介绍如何在 $O(n)$ 的时间内获取数列的第 k 小的数。给定数列 A ，可以通过以下算法来获取该数列的第 k 小的数：从序列中取一个数 A_i ，然后把序列分为小于 A_i 和大于等于 A_i 的两部分，由两个部分的元素个数与 k 的大小关系可以确定第 k 小的数是在哪个部分。

```
-----4.1.3.cpp-----
int partition(int A[], int left, int right) {
    int pivot = A[left];
    while (true) {
        while (left < right && pivot <= A[right]) right--;
        if (left >= right) break;
        A[left++] = A[right];
        while (left < right && A[left] <= pivot) left++;
        if (left >= right) break;
        A[right--) = A[left];
    }
    A[right] = pivot;
    return right;
}

int kth_element(int A[], int left, int right, int k) {
    while (true) {
        int middle = partition(A, left, right);
        if (middle == k) return A[k];
        else {
            if (middle < k) left = middle + 1;
            else right = middle - 1;
        }
    }
}-----4.1.3.cpp-----
//-----
```

利用上述实现，数列 A 的中位数 m 可以通过以下调用获得：

```
double m = 0;
m += kth_element(A, 0, n - 1, (n - 1) / 2);
m += kth_element(A, 0, n - 1, n / 2);
m /= 2.0;
```

除了自行编写代码实现中位数的获取，还可以使用算法函数库提供的 `nth_element`，它所完成的功能正是使得数组的第 k 个位置的数恰为排序后的第 k 小元素¹。

强化练习：10041 Vito's Family^A。

扩展练习：11300* Spreading the Wealth^B。

4.2 插入排序

¹ 参见本章第 4.9.4 小节 “`nth_element`” 的内容。

4.2.1 直接插入排序

直接插入排序 (straight insertion sort) 的算法思想为：通过持续构建有序序列来达到整个序列有序的目的，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。插入排序在实现上通常采用本地排序，因此在从后向前扫描过程中，需要反复地把已排序元素逐步向后挪位，为最新元素提供插入空间。如果使用普通的插入排序，其时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，属于稳定排序。

```
-----4.2.1.cpp-----
void insertionSort(int data[], int n) {
    for (int i = 1; i < n; i++) {
        // 查找插入位置。若未找到，则将有序元素向后移动一个位置。
        int temp = data[i], j = i - 1;
        while (j >= 0 && data[j] > temp) {
            data[j + 1] = data[j];
            j--;
        }
        // 将元素写入找到的位置。
        data[j + 1] = temp;
    }
}
-----4.2.1.cpp-----
```

注意到数组的一部分已经有序，在后续插入过程中，可以使用二分查找来找到需要插入的位置，这样可以减少比较次数。

扩展练习：[110 Meta-Loopless Sorts^B](#)，[10107 What is the Median^A](#)，[12488 Start Grid^C](#)。

4.2.2 希尔排序

希尔排序 (Shell sort)，最初由希尔^I提出，因此得名。希尔排序基于插入排序，但是此排序算法采用了新的技巧，提高了插入排序的效率^[35]。其基本思想如下：将待排序数据按照一个逐渐递减的间隔 d 分成若干组，对每组数据采用插入排序，当最后间隔 d 变为 1 时，即为普通的插入排序。算法要求间隔 d 最后必须为 1 以保证能够使数据有序。对于间隔 d ，不同的选择导致稍有差异的实现。

希尔排序时间复杂度在最差的情况下为 $\Theta(n \log n) \sim \Theta(n^2)$ ，平均时间复杂度大致为 $\Theta(n \sqrt{n})$ ，与选择的间隔序列有关；空间复杂度为 $O(1)$ ，属于不稳定排序。

```
-----4.2.2.cpp-----
void shellSort(int data[], int n) {
    int gap, i, j, temp;
    for (gap = n / 2; gap > 0; gap /= 2)
        for (i = gap; i < n; i++)
            for (j = i - gap; j >= 0 && data[j] > data[j + gap]; j -= gap)
                swap(data[j], data[j + gap]);
}
-----4.2.2.cpp-----
```

强化练习：[855 Lunch in Grid City^B](#)。

4.3 选择排序

^I 唐纳德·希尔 (Donald L. Shell, 1924—2015)，美国计算机科学家。

4.3.1 直接选择排序

选择排序 (selection sort)，其算法思想为：首先在未排序序列中找到最小元素，将其交换到排序序列的起始位置，再从剩余未排序元素中继续找出最小元素，将其交换到已排序序列的末尾，重复此过程，直到所有元素均排序完毕。需要注意，选择排序和冒泡排序的算法思想看起来类似但实质上是不同的。

选择排序的时间复杂度为 $O(n^2)$ ，空间复杂度为 $O(1)$ ，如果使用原地交换的方法，属于不稳定排序，如果借助额外的存储空间可使算法成为稳定的。

```
-----4.3.1.cpp-----
void selectionSort(int data[], int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i + 1; j < n; j++)
            if (data[i] > data[j])
                swap(data[i], data[j]);
    }
}-----4.3.1.cpp-----
```

强化练习：[11875 Brick Game^A](#)。

4.3.2 堆排序

堆排序 (heap sort) 利用了堆的性质来进行排序。一个堆就是一棵二叉树，树中每一个结点的值都大于或者等于任意一个子结点的值。

堆排序时间复杂度为 $O(n \log n)$ ，空间复杂度为 $O(n)$ ，属于不稳定排序。

```
-----4.3.2.cpp-----
// 对数组进行调整，使之具有堆的性质。
void heapify(int data[], int parent, int n) {
    int left = 2 * parent + 1, right = 2 * parent + 2, max = parent;
    if (left < n && data[left] > data[max]) max = left;
    if (right < n && data[right] > data[max]) max = right;
    if (max != parent) {
        swap(data[parent], data[max]);
        heapify(data, max, n);
    }
}

// 构建堆。
void buildHeap(int data[], int n) {
    for (int parent = n / 2 - 1; parent >= 0; parent--)
        heapify(data, parent, n);
}

// 堆排序。先构建最大堆，然后每次将最大元素放到最后，继续调整剩下元素使之构成堆。
void heapSort(int data[], int n) {
    buildHeap(data, n);
    for (int i = n - 1; i > 0; i--) {
        swap(data[0], data[i]);
        heapify(data, 0, i);
    }
}-----4.3.2.cpp-----
```

强化练习：[13109 Elephants^A](#)。

4.4 归并排序

归并排序 (merge sort) 是建立在归并操作上的一种有效的排序算法, 该算法是分治法的一个非常典型的应用。归并排序是通过将已经有序的子序列予以合并来得到完全有序的序列, 即先使每个子序列有序, 再使各个子序列 “段间有序”, 最后达到整个序列有序。若在排序过程中, 每次合并操作中至多只将两个有序表合并成一个有序表, 称为二路归并排序, 如果在一次合并中将两个以上有序表合并为一个有序表, 称为多路合并排序。在 C++ 的 SGI 库实现中, 稳定排序函数 `stable_sort` 的实现即使用了归并排序作为子过程。

二路归并排序的时间复杂度为 $O(n \log n)$, 空间复杂度为 $O(n)$, 属于稳定排序。

```
//-----4.4.cpp-----
const int MAXN = 10000;

// 临时数组, 用于存储归并后的数据。
int tmp[MAXN];

// 将两个有序区间合并。
void merge(int data[], int left, int middle, int right) {
    int i = left, j = middle + 1, k = 0;
    // 逐个取元素直到一个有序区间被取完。
    while (i <= middle && j <= right)
        tmp[k++] = data[i] <= data[j] ? data[i++] : data[j++];
    // 取完另一个区间的元素。
    while (i <= middle) tmp[k++] = data[i++];
    while (j <= right) tmp[k++] = data[j++];
    // 将数据复制回原数组。
    for (int i = 0; i < k; i++) data[left + i] = tmp[i];
}

// 合并排序, 先排序左右两个区间, 然后合并左右两个有序的区间。
void mergeSort(int data[], int left, int right) {
    if (left < right) {
        // 以中间元素作为左右区间的分隔。
        int middle = (left + right) >> 1;
        mergeSort(data, left, middle);
        mergeSort(data, middle + 1, right);
        merge(data, left, middle, right);
    }
}
//-----4.4.cpp-----
```

强化练习: [12673 Footballc](#)。

4.4.1 逆序对数

给定一个长度为 n 的数列 a_1, a_2, \dots, a_n , 数列中的元素互不相同, 将满足 $1 \leq i < j \leq n$ 且 $a_i > a_j$ 的序号对 (i, j) 称为逆序 (inversion), 数列中所有逆序的数量称为逆序对数 (the number of inversions)。朴素的方法是枚举每一对序号 (i, j) 以检查其是否为逆序, 这将导致 $O(n^2)$ 的算法。令人感到惊奇的是, 归并排序竟然与逆序对数有关联。根据归并排序的特点, 可以设计一种时间复杂度为 $O(n \log n)$ 的算法来计数逆序对数, 其关键是应用分治法的思想来递归地解决这个问题。

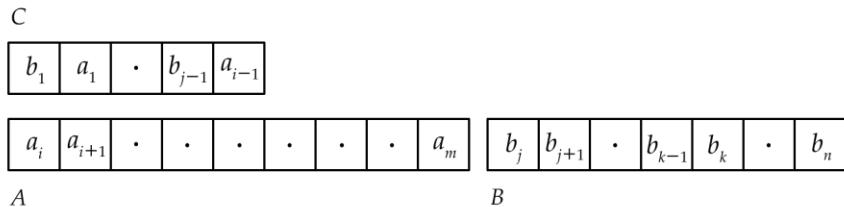


图 4-1 确定逆序对数。子序列 A 是归并前位于左侧的已排序子序列，子序列 B 是归并前位于右侧的已排序子序列，子序列 C 是归并后的结果序列，假设 $b_j < a_i$ (如果 $a_i < b_j$ 可以同理论证)，则将 b_j 附加到结果序列 C 中时， b_j 和子序列 A 中从 a_i 到 a_m 的元素构成逆序对，因此 b_j 对逆序对数的贡献为 $m - i + 1$

如图 4-1 所示，观察归并排序的过程，在将两个子序列排序好以后，令其为子序列 A 和子序列 B，假设在进行合并之前，已经得到了两个子序列各自的逆序对数 I_A 和 I_B ，那么现在需要做的就是统计子序列 A 相对于子序列 B 来说存在多少个逆序。令合并后的序列为 C，对于子序列 A 和 B 中的两个元素 a_i 和 b_j 来说，假设 $b_j < a_i$ ，则由于子序列 A 和 B 已经有序，可以很容易计数逆序的数量：当 b_j 被附加到结果序列 C 中时，由于 $b_j < a_i$ 的关系，此时 b_j 小于子序列 A 中剩余的所有元素，因此对逆序对数所作的贡献就是此时子序列 A 中尚未进入结果序列 C 的元素数量 $m - i + 1$ ；相反，当元素 a_i 附加到结果序列 C 中时，不会对逆序对数作出贡献，因为在子序列 B 中且小于 a_i 的元素 b_j 至 b_k (或者 b_n ，如果 $a_i \geq b_n$) 已经附加到结果序列 C 中，此时 a_i 小于子序列 B 中剩余的所有元素 (或者子序列 B 已经为空)。可将上述算法概括为以下的伪代码描述^[36]。

Merge-and-Count(A, B)

- (1) 维护两个指针 $current_A$ 和 $current_B$ ，初始时分别指向子序列 A 和子序列 B 的第一个元素；
- (2) 维护变量 $count$ 计数逆序对数，初始时为 0；
- (3) 当子序列 A 和 B 都不为空时：令 a_i 和 b_j 为 $current_A$ 和 $current_B$ 所指向的元素，将 a_i 和 b_j 中的较小者附加到结果序列 C 中，如果 b_j 是较小的元素，则将 $count$ 增加序列 A 中仍有元素的数量，然后将较小的元素所在序列的指针向后移动一个位置；
- (4) 当子序列 A 和 B 中的某个为空时，将另外一个序列的所有元素附加到结果序列 C 中；
- (5) 返回逆序对数 $count$ 和结果序列 C。

不难看出，计数逆序数对的过程 Merge-and-Count 的时间复杂度为 $O(n \log n)$ ，将其作为一个子过程融入归并排序中，则最终可以在 $O(n \log n)$ 的时间内得到整个数列的逆序对数。

```
//-----4.4.1.cpp-----//
const int MAXN = 100010;

int n, data[MAXN], tmp[MAXN];

// 将两个有序的区间合并，同时统计逆序对数。
long long mergeAndCount(int left, int middle, int right) {
    long long count = 0;
    int i = left, j = middle + 1, k = 0;
    while (i <= middle && j <= right)
        tmp[k++] =
            // 比较 data[i] 和 data[j]，注意到区间 [left, middle] 和 [middle + 1, right]
            // 已经按升序排列，如果 data[i] 大于 data[j]，则区间 [i, middle] 的数与 data[j]
            // 构成逆序对，因此总的逆序对数增加 middle - i + 1 对。
            data[i] <= data[j] ? data[i++] : (count += (middle - i + 1), data[j++]);
    while (i <= middle) tmp[k++] = data[i++];
}
```

```

        while (j <= right) tmp[k++] = data[j++];
        for (int i = 0; i < k; i++) data[left + i] = tmp[i];
        return count;
    }

// 将两个子区间各种排序然后归并。
long long mergeSort(int left, int right) {
    long long count = 0;
    if (left < right) {
        int middle = (left + right) >> 1;
        count += mergeSort(left, middle);
        count += mergeSort(middle + 1, right);
        count += mergeAndCount(left, middle, right);
    }
    return count;
}

int main(int argc, char *argv[]) {
    while (cin >> n, n > 0) {
        for (int i = 0; i < n; i++) cin >> data[i];
        cout << mergeSort(0, n - 1) << '\n';
    }
}
//-----4.4.1.cpp-----//

```

强化练习: 10810 Ultra-QuickSort^A, 11495 Bubbles and Buckets^B, 12071* Understanding Recursion^D, 11858 Frosh Week^C, 13212 How Many Inversions^D。

4.5 计数排序

一般情况下, 基于比较的排序算法其性能不可能好于 $O(n \log n)$, 但是若能够知道待排序元素的更多信息, 就可以使用其他的排序方法来提高效率。例如, 给定 n 个元素并且确定每一个元素值的范围都在 $[0, C)$ 之间, 而 C 比 n 小得多 (或者 n 本身不大), 那么就能够利用此信息, 使用一种线性的排序算法——计数排序 (counting sort)。

计数排序创建了 C 个桶用来存储输入数列中的各个元素值出现的次数。计数排序将对输入数列进行两次遍历。在第一次遍历中, 增加桶的计数。在第二次遍历中, 通过处理桶中得到的整个待排序序列的计数值, 重写原始的数列。

计数排序时间复杂度为 $O(n + C)$, 空间复杂度为 $O(C)$, 属于稳定排序。

```

//-----4.5.cpp-----//
void countingSort(int data[], int n, int C) {
    // 注意: 使用 new 为内置数据类型分配内存, 需要使用 “()” 强制进行初始化操作。
    int *bucket = new int[C]();
    for (int i = 0; i < n; i++) bucket[data[i]]++;
    for (int i = 0, index = 0; i < C; i++)
        while (bucket[i]-- > 0)
            data[index++] = i;
    delete [] bucket;
}
//-----4.5.cpp-----//

```

强化练习: 10057 A Mid-Summer Night's Dream^B, 11462 Age Sort^A, 11850 Alaska^A。

4.6 基数排序

基数排序 (radix sort) 是指将欲排序的数据按十进制进行数位的拆分, 根据数位从低到高逐个比较达到有序的过程。主要包括两个部分: (1) 分配, 从个位开始, 根据数位将数据分配到 0~9 号桶中; (2) 收集, 将 0~9 号桶中的数据按顺序放到数组中, 重复分配和收集的过程, 最终达到数据的最高位, 此时数组中的数已经有序。

基数排序平均时间复杂度为 $O(Dn)$, D 为数据所具有的最大位数; 空间复杂度为 $O(D)$; 属于稳定排序。

```
//++++++4.6.cpp+++++++
// 获取给定数指定位置的数字。
int getDigit(int number, int index) {
    while (number > 0 && index > 0) {
        number /= 10;
        index--;
    }
    return number % 10;
}

// 基数排序。
void radixSort(int data[], int n, int digits) {
    int *bucket[10];
    // 申请存储空间。
    for (int i = 0; i < 10; i++) {
        bucket[i] = new int[n + 1];
        bucket[i][0] = 0;
    }
    // 从低位到高位逐次排序。
    for (int index = 0; index < digits; index++) {
        // 分配。
        for (int i = 0; i < n; i++) {
            int digit = getDigit(data[i], index);
            bucket[digit][++bucket[digit][0]] = data[i];
        }
        // 收集。
        for (int i = 0, j = 0; i < 10; i++) {
            int k = 1;
            while (k <= bucket[i][0])
                data[j++] = bucket[i][k++];
            bucket[i][0] = 0;
        }
    }
    // 释放内存。
    for (int i = 0; i < 10; i++) delete [] bucket[i];
}
```

由于基数排序的数位在 0~9 之间, 因此也可以利用计数排序的变种来实现基数排序。

```
// 使用计数排序的变种对数组按指定位置排序。
void countingSort(int data[], int n, int index) {
    int *bucket = new int[10], *sorted = new int[n];
    for (int i = 0; i < 10; i++) bucket[i] = 0;
    for (int i = 0; i < n; i++) bucket[getDigitAtIndex(data[i], index)]++;
    for (int i = 1; i < 10; i++) bucket[i] += bucket[i - 1];
    for (int i = n - 1; i >= 0; i--) {
```

```
        int digit = getDigit(data[i], index);
        sorted[bucket[digit] - 1] = data[i];
        bucket[digit]--;
    }
    for (int i = 0; i < n; i++) data[i] = sorted[i];
    delete [] bucket, sorted;
}

// 基数排序。
void radixSort(int data[], int n, int digits) {
    for (int index = 0; index < digits; index++) countingSort(data, n, index);
}
//+++++4.6.cpp+++++//
```

4.7 桶排序

如果待排序数组中元素为非负整数且最大值为 K ，则可以利用此性质，使用桶排序 (bucket sort)。这里的桶可以视为存储数据的容器。算法思想为：设立 B 个桶，若元素 x 满足

$$(K/B) \times i \leq x < (K/B) \times (i + 1)$$

则将元素 x 置入第 i 个桶中，当所有元素均分配到某个桶中后，对单个桶中的元素使用插入排序（或其他基本排序方法），最后按序收集各个桶中的元素形成有序的数组。

桶排序时间复杂度为 $O(n+B)$ ，因为桶内元素的数量和最大值 K 以及桶的总数 B 有关，故桶排序的空间复杂度不确定，其算法稳定性取决于对单个桶中元素进行排序时所使用算法的稳定性。

```
-----4.7.cpp-----//
void bucketSort(int data[], int n, int K) {
    int B = 100;
    vector<int> buckets[B];
    // 将数据分布到 B 个桶中。
    for (int i = 0; i < n; i++) {
        int bi = data[i] * B / K;
        buckets[bi].push_back(data[i]);
    }
    // 对每个桶内的元素排序。
    for (int i = 0; i < B; i++) sort(buckets[i].begin(), buckets[i].end());
    // 按序收集每个桶中的元素。
    int k = 0;
    for (int i = 0; i < B; i++)
        for (int j = 0; j < buckets[i].size(); j++)
            data[k++] = buckets[i][j];
}
//-----4.7.cpp-----//
```

4.8 查找

4.8.1 顺序查找

顺序查找 (linear search) 是最简单的查找方法, 当数据量不大或数据本身无序时, 使用此种方法较为简单¹。由于是逐个元素比较, 顺序查找的时间复杂度为 $O(n)$ 。

¹ 算法库函数中的 `find` 即提供顺序查找的功能，参见本章算法库函数一节的内容。

```

-----4.8.1.cpp-----//
// 自定义的顺序查找函数。
int find(string data[], int n, string target) {
    for (int i = 0; i < n; i++)
        if (data[i] == target)
            return i;
    return -1;
}

// 使用自定义的顺序查找函数和算法库提供的 find 函数。
int main(int argc, char *argv[]) {
    string months[12] = {
        "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"
    };
    string weekdays[7] = {
        "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday", "Sunday"
    };
    // 使用 find 库函数进行查找。
    int monthIndex = find(months, months + 12, "July") - months;
    cout << monthIndex << endl;
    // 使用自定义 find 函数进行查找。
    monthIndex = find(months, 12, "March");
    cout << monthIndex << endl;
    // 使用 find 库函数进行查找。
    int weekdayIndex = find(weekdays, weekdays + 7, "Someday") - weekdays;
    cout << weekdayIndex << endl;
    // 使用自定义 find 函数进行查找。
    weekdayIndex = find(weekdays, 7, "Sunday");
    cout << weekdayIndex << endl;
    return 0;
}
-----4.8.1.cpp-----//

```

输出为：

```

6
2
7
6

```

强化练习：10340 All in All^A。

4.8.2 二分查找

顺序查找是从序列的起始位置开始，逐个元素向后查找，效率不高。如果待查找的数据已经有序（升序或降序排列），则可以使用二分查找（binary search）来提高效率^I。假设一维数组 *data* 已经按升序排列，二分查找算法根据当前需要查找的区间 $[left, right]$ 定义一个中间位置 $middle = (left + right) / 2$ ，将待查找值 *x* 与数组元素 *data[middle]* 进行比较，有三种情况：

^I 与二分查找思想类似的还有黄金分割搜索、斐波那契搜索，感兴趣的读者请查阅相关资料以获取进一步的了解。

- (1) $x = data[middle]$, 则找到了该元素;
- (2) $x > data[middle]$, 由于数组是按升序排列的, 待寻找的值要么不在数组中, 要么只可能在右半区间 $[middle+1, right]$;
- (3) $x < data[middle]$, 待寻找的值要么不在数组中, 要么只可能在左半区间 $[left, middle-1]$ 。

由于每次查找都是在原区间的一半内进行, 又称为折半查找, 总的时间复杂度为 $O(\log n)$ 。

以一个具体的例子来说明二分查找的工作过程。设 $data[10] = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$, 待查找的值 $x = 7$, 初始时, 需要查找的区间为 $[left = 0, right = 9]$, 中间位置 $middle = (left + right) / 2 = (0 + 9) / 2 = 4$, 由于 $data[4] = 5 < x = 7$, 则应该在右半区间 $[left = middle + 1 = 5, right = 9]$ 中继续查找, 此时 $middle = (5 + 9) / 2 = 7$, 而 $data[7] = 8 > x = 7$, 应该继续在左半区间 $[left = 5, right = middle - 1 = 6]$ 中查找, 之后 $middle = (5 + 6) / 2 = 5$, $data[5] = 6 < x = 7$, 将查找区间更新为 $[left = middle + 1 = 6, right = 6]$, 最后 $middle = (6 + 6) / 2 = 6$, $data[6] = x = 7$ 。

虽然二分查找的思想很简单, 但是实际编写一个正确的二分查找函数并不像想象中的那么简单, 需要注意诸多细节, 否则很容易存在难以发现的 Bug。以下是根据上述二分查找的思想得到的一个“基本正确”的实现。

```
// 二分查找函数, 在数组中查找指定值 x, 如果找到则返回其序号, 否则返回-1。
//+++++4.8.2.cpp+++++
int binarySearch(int data[], int n, int x) {
    int left = 0, right = n - 1;
    while (left <= right) {
        int middle = (left + right) / 2;
        if (data[middle] == x) return middle;
        if (data[middle] < x) left = middle + 1;
        else right = middle - 1;
    }
    return -1;
}
```

之所以所上述实现是一个“基本正确”的实现, 是因为它还存在几个小问题(尽管不是严格意义上的算法实现问题):

(1) 使用 $middle = (left + right) / 2$ 的方式来获取中间值, 如果 $left + right$ 接近其声明的数据类型的表示上限, 则 $left + right$ 会溢出, 从而导致错误, 更为稳妥的方法是使用: $middle = left + (right - left) / 2$, 这样能最大程度的避免溢出。

(2) 如果数组中包含多个相同的目标值, 上述实现返回的序号并不一定是数组中第一个目标值的序号, 比如极端的情况——数组中的所有元素值均相同, 此时使用上述实现查找某个元素值时, 要么不存在, 要么返回的总是中间的固定位置。

3) 在 `while` 循环中包含了两次比较, 能否减少比较次数从而使得效率更高呢?

以下是一个改进的实现, 巧妙地解决了上述三个问题^[37]。

```
// 二分查找函数, 在数组中查找指定值 x, 如果找到则返回其序号, 否则返回-1。
int binarySearch(int data[], int n, int x) {
    int left = -1, right = n, middle;
    // 循环保持的条件是 left < right 且 data[left] < x ≤ data[right], 在循环结束时,
    // 如果可以找到目标值, 则只剩下两个数, 并且满足 data[left] < x ≤ data[right],
    // 要查找的序号为 right。
    while ((left + 1) != right) {
```

```
middle = left + (right - left) / 2;
// 需要保证 data[left] <= x <= data[right], 所以 left=middle, 如果取
// left=middle+1, 则有可能出现 data[left] <= x 的情况。
if (data[middle] < x) left = middle;
else right = middle;
}
// 检查序号是否符合要求。
if (right >= n || data[right] != x) right = -1;
return right;
}
//+++++4.8.2.cpp+++++
```

强化练习: 10170 The Hotel with Infinite Rooms^A, 12791 Lap^C, 12965 Angry Bids^D, 12909 Numeric Center^E。

4.8.3 方程求近似解

应用二分查找思想，可以求解一元高次方程和超越方程的近似解。由于一般的一元高次方程和超越方程并无固定的求解公式，如果能够确定方程所对应的函数在指定定义域内是单调的，则可以利用二分查找来得到近似解^[38]。

10341 Solve It^A (解方程)

解以下方程：

$$p * e^{-x} + q * \sin(x) + r * \cos(x) + s * \tan(x) + t * x^2 + u = 0$$

其中 $0 \leq x \leq 1$ 。

输入

输入包含多组测试数据并以文件结束符作为输入结束标记。每行包含一组测试数据，每组测试数据包括六个整数: p, q, r, s, t, u ($0 \leq p, r \leq 20$ 且 $-20 \leq q, s, t \leq 0$), 输入文件最多包含 2100 行。

输出

对于每组测试数据，如果有解，输出精确到小数点后 4 位的解，否则，输出“**No solution**”。

样例输入

样例输出

$$\begin{matrix} 0 & 0 & 0 & 0 & -2 & 1 \\ 1 & 0 & 0 & 0 & -1 & 2 \\ 1 & -1 & 1 & -1 & -1 & 1 \end{matrix}$$

0.7071
No solution
0.7554

分析

观察方程的组成部分，可以发现包含未知数 x 的每一项在定义域上都是单调递减函数，故整个方程所对应的函数也是单调递减的，可以应用二分搜索来确定方程的近似解¹。

¹ 在使用二分搜索时，由于题目约束条件的不同，最后得到的中间值 $middle$ 可能并不一定是解，特别是在一些搜索范围的数均是整数的时候。此时可以使用一个额外的变量来记录当前符合题目约束的解，避免从 $middle$ 的值去推断最后的解，因为这样容易出错。例如，假设需要使用二分搜索（借助实数库函数和枚举亦可）确定在区间 $[1, 100000]$ 内满足 $3x^3+x^2+11$ 的值大于 100000007 的最小正整数，则可以使用：

```
long long f(long long x) { return 3 * x * x * x * x + x * x + 11 > 100000007; }  
(转到下一页)
```

参考代码

```
#define v(x) (p * exp(-x) + q * sin(x) + r * cos(x) + s * tan(x) + t * x * x + u)
double p, q, r, s, t, u;
int main(int argc, char *argv[]) {
    while (cin >> p >> q >> r >> s >> t >> u) {
        double left = 0.0, right = 1.0, middle;
        // 迭代最多 40 次, 精度已经足够。
        for (int i = 1; i <= 40; i++) {
            middle = (left + right) / 2.0;
            if (v(middle) > 0.0) left = middle;
            else right = middle;
        }
        // 检查解是否符合要求。
        if (fabs(v(middle)) > 1e-8) cout << "No solution\n";
        else cout << fixed << setprecision(4) << middle << '\n';
    }
    return 0;
}
```

强化练习: 358 Don't Have A Cow Dude^D, 1753 Need for Speed^B, 10474 Where is the Marble^A, [11627*](#) Slalom^D, 11881 Internal Rate of Return^D, [11935](#) Through the Desert^C, [12032](#) The Monkey and the Oiled Bamboo^A, 12124 Assemble^C, [12190](#) Electric Bill^D。

4.8.4 最大值最小化问题

给定由 n 个正整数构成的序列, 要求将其划分为 k 个部分, $1 \leq k \leq n$, 使得这 k 个部分的元素和的最大值尽可能的小, 此即为最大值最小化问题。例如将序列 $<1, 2, 5, 7, 2, 9, 8>$ 划分为 3 个部分, 则 $<1, 2, 5, 7, | 2, 9, | 8>$ 的划分是最优的, 其最大值为 15。

使用穷尽搜索显然不可行, 需要另辟蹊径。直接求最小值不可行, 但是给定一个值 x , 验证是否能够使得划分的最大值不超过 x 却很容易, 方法是使用贪心策略, 从左至右对序列进行划分, 尽可能的向右侧扩展, 使得每次划分的数之和不超过 x , 同时记录划分次数 p , 如果 $p > k$, 则说明给定的 x 值较小, 使得划分数偏大, 应该调整 x 值使之变大, 从而使得划分数 p 变小; 如果 $p = k$, 说明能够得到满足条件的划分; 如果 $p < k$, 则给定的 x 值较大, 使得划分数偏小, 应该调整 x 值使之变小, 从而使得划分数 p 增大。上述过程和二分查找的思想是类似的, 因此可以通过二分搜索来确定满足条件的最小 x 值。

```
-----4.8.4.cpp-----
// 将包含 n 个正整数的序列划分为 k 个部分, 最小化划分和的最大值, 并将该值返回。
int partition(int data[], int n, int k) {

int main(int argc, char *argv[]) {
    int left = 1, right = 100000, middle, r;
    while (left <= right) {
        middle = (left + right) >> 1;
        if (f(middle)) r = middle, right = middle - 1;
        else left = middle + 1;
    }
    cout << r << '\n';
    return 0;
}
```

```

// 确定查找区间。
int left = 0, right = 0;
for (int i = 0; i < n; i++) right += data[i];
// 反复迭代直到找到目标值。
while (left <= right) {
    // 确定当前最大和 middle。
    int middle = (left + right) / 2;
    // 使用贪心策略，验证能否将序列划分为 k 个部分，使得每个部分的和均不超过 middle。
    int j = 0, p = 0, sum = 0, ok = 1;
    while (j < n && p <= k) {
        // 如果部分和与当前元素的和仍然小于等于 middle，将该元素划分到当前部分。
        if (sum + data[j] <= middle) {
            sum += data[j];
            j++;
        }
        // 划分数增加 1。这里有两种情况：一种是当前和 sum 大于 0 且小于 middle，当加上 data[j] 后将大于 middle，因此划分数需要增加 1；另外一种情况是 data[j] 本身已经大于 middle，而 sum 为 0，此时将不断将 p 增加 1 直到不再满足条件 p ≤ k 而最终退出循环。
        else {
            sum = 0;
            p++;
        }
    }
    // 如果部分和不为 0，说明有部分元素还未划分，则划分数 p 至少增加 1。
    if (sum > 0) p++;
    // 根据划分数判断。如果 p > k，很显然，不能将 n 个数按指定的 middle 值划分为 k 个部分，说明 middle 值太小，应该将区间向右调整。当 p ≤ k 时，说明 middle 较小或者恰好合适，可以将区间向左半部分调整。
    if (p > k)
        left = middle + 1;
    else
        right = middle - 1;
}
return left;
}
//-----4.8.4.cpp-----//

```

强化练习：[714 Copying Books^A](#)，[907 Winterim Backpacking Trip^C](#)，[11413 Fill the Containers^A](#)，[12097 Pie^C](#)，[12390 Distributing Ballot Boxes^D](#)，[13177 Orchestral Scores^D](#)。

扩展练习：[11516 WiFi^B](#)，[12255 Underwater Snipers^E](#)。

4.8.5 三分搜索

在某些情形下，根据题目所得到的函数可能在定义域内并不是一个单调递增（或递减）函数，而是一个单峰函数（例如抛物线函数），此时应用二分搜索无法确定函数的极值，而应用三分搜索则能够在 $O(\log n)$ 的时间内得到解。

427 Flatland Piano Movers^C（平面世界钢琴搬运）

平面世界钢琴搬运公司决定将全程质量管理的重心放在作业评估环节。该环节的部分工作是根据预先设定的搬运路线，确定钢琴能否与走廊宽度以及拐角的大小相符，使得钢琴在移动过程中能够通过这些走廊和拐角。钢琴外形为矩形且尺寸不一。走廊的拐角均为直角，不会出现‘T’形的拐角。所有的尺寸单位均为

浪^I。在钢琴的移动过程中，走廊的长度都足够长，除了当前的拐角，其他的拐角和走廊上的门对钢琴转弯没有影响，而且钢琴的宽度要比任意给出的走廊宽度要窄。注意，因为钢琴只能通过推或者拉它的较短的一边的方式来通过拐角（不存在正方形的钢琴），所以钢琴在经过拐角时要发生转弯。编写程序，对于给定尺寸的钢琴，判断其是否能够通过走廊上的拐角。

输入

输入包含多行，每行最多包含 80 个字符，以文件结束符作为输入结束的标记。每行输入由多个数对组成，每个数对内的两个数以逗号相分隔，每个数对之间至少相隔一个空格。第一个数对表示钢琴的尺寸，其余的数对表示钢琴在搬运过程中所经过的拐角两边通道的宽度。考虑以下输入：

```
600,200 300,500 837,500 350,350
```

钢琴的长宽分别为 600 浪和 200 浪，第一个拐角是从宽度为 300 浪的走廊转到宽度为 500 浪的走廊。下一个拐角是从宽度为 837 浪的走廊转到宽度为 500 浪的走廊。最后一个拐角是从宽度为 350 浪的走廊转到另外一个宽度为 350 浪的走廊。

输出

对于每一架钢琴，对给定的每个拐角，回答其能否经过这个拐角，如果能够经过，输出 ‘Y’，否则输出 ‘N’。不同钢琴的输出分开。

样例输入

```
600,200 300,500 837,500 350,350
137,1200 600,500 600,400
```

样例输出

```
YYN
YN
```

分析

题目要求判断给定尺寸的矩形是否能够在指定尺寸的拐角处转弯。类似于汽车的转弯，如果拐角过小，则汽车不能完成转弯。

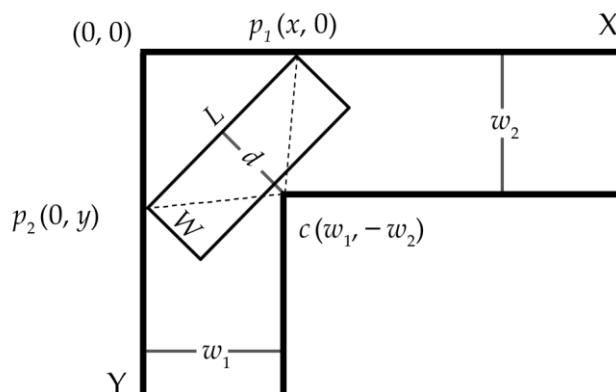


图 4-2 钢琴在拐角转弯时的几何建模。将钢琴视为一个长为 L ，宽为 W 的矩形

^I furlong, 浪，英国长度单位，1 浪=220 码≈201 米。

将钢琴在拐角转弯的过程建模为图 4-2。钢琴在拐角进行转弯的过程，可以看做一条长为 L 的线段，一个端点 p_1 固定在 X 轴上（正轴向右），另一个端点 p_2 固定在 Y 轴上（正轴向上），端点 p_1 沿着 X 轴不断向右滑动的过程。只要滑动过程中，拐角点 c 距离线段 p_1p_2 的距离 d 始终大于或等于钢琴的宽度 W ，那么钢琴就可以完成在此拐角处的转弯。根据几何关系，距离 d 是三角形 p_1p_2c 的底边 p_1p_2 上的高，而由三角形 p_1p_2c 的有向面积^I，可以得到

$$d = \frac{2 \times \Delta p_1 p_2 c}{L} = \frac{xy - w_1 y + w_2 x}{L} \geq W$$

由于点 p_1 和 p_2 是直角三角形的斜边，故有

$$x^2 + y^2 = L^2 \Rightarrow y = -\sqrt{L^2 - x^2}$$

将上式代入消去 y ，同时根据距离的关系得到下式

$$(w_1 - x)\sqrt{L^2 - x^2} + w_2 x - LW \geq 0, \quad x \in [0, L]$$

问题归结为求上式的最小值问题。如果函数的最小值大于等于 0，则钢琴可以在拐角转弯，如果小于 0，则不能完成在拐角的转弯。

由于得到的函数不是一个单调递增的函数，不能简单的使用二分查找来求极值。观察可知函数表达式是一个二次函数，其几何曲线为一条类抛物线，为凸性函数^{II}，具有极值，可以使用三分搜索法来求函数的极值^{III}。

三分搜索的思路是将变量的取值范围划分为三个不同的区间，根据函数值的大小关系，使用类似于二分查找的方法来不断缩小极值可能所处的变量范围。

假设函数 $f(x)$ 为单峰函数，自变量 x 的定义域为 $[left, right]$ ，在定义域内有最大值。

设 $leftThird = left + (right - left)/3$, $rightThird = right - (right - left)/3$ ，则 $f(leftThird)$ 和 $f(rightThird)$ 的大小关系为以下三种之一：

$f(leftThird) < f(rightThird)$: 表明最大值不可能在左区间 $[left, leftThird]$ ，应该在右区间 $[rightThird, right]$ 中寻找最大值，更新 $left = leftThird$ 。

$f(leftThird) > f(rightThird)$: 表明最大值不可能在右区间 $[rightThird, right]$ ，应该继续在左区间 $[left, leftThird]$ 中寻找最大值，更新 $right = rightThird$ 。

$f(leftThird) = f(rightThird)$: 表明最大值落在区间 $[leftThird, rightThird]$ 中，此种情况可以合并在前述的任意一种情况中。

持续缩小区间的范围，直到 $left$ 和 $right$ 之间的差值小于指定的阈值，此时变量 $left$ 所对应的函数值 $f(left)$ 即为最大值。如果要求最小值，只需在比较函数值大小时将比较操作反向或者更改区间的选择即可。

参考代码

```
const double EPSILON = 1e-6;
double L, W, w1, w2;
char comma;
```

^I 参见第 14 章第 14.6.2 小节“多边形面积”的内容。

^{II} 经与洛谷用户 (<https://www.luogu.com.cn/user/592895>) 讨论，发现对于绝大部分测试数据来说，函数是一个凸函数的说法是正确的，但是对于某些数据来说，函数所对应的图像可能并不是凸函数，请读者予以注意。

^{III} 类似的有黄金分割搜索，与三分搜索相比，黄金分割搜索可以重复利用上一步的计算结果，减小计算量。感兴趣的读者请查阅相关资料以获取进一步的了解。

```

// 计算函数值。
double f(double x) {
    return sqrt(L * L - x * x) * (w1 - x) + w2 * x - L * w;
}
int main(int argc, char *argv[]) {
    string line;
    while (getline(cin, line)) {
        istringstream iss(line);
        iss >> L >> comma >> W;
        if (L < W) swap(L, W);
        while (iss >> w1 >> comma >> w2) {
            double left = 0, right = L;
            // 三分搜索。
            do {
                double leftThird = left + (right - left) / 3;
                double rightThird = right - (right - left) / 3;
                if (f(leftThird) > f(rightThird)) left = leftThird;
                else right = rightThird;
            } while (fabs(left - right) > EPSILON);
            cout << (f(left) >= 0 ? 'Y' : 'N');
        }
        cout << '\n';
    }
    return 0;
}

```

强化练习：1476 Error Curves^D, [10385](#) Duathlon^D。

4.8.6 分散层叠

考虑以下问题：现有 k 个已按升序排序的整数列表，给定整数 x ，要求在这 k 个列表中查询大于或等于 x 的最小整数的序号。

由于给定的 k 个列表都已经是有序的，朴素的做法是在这 k 个列表中各进行一次二分查找，以得到每个列表中大于或等于 x 的最小整数的序号。假定每个列表的大小均为 n ，则在一个列表中进行查询，其时间复杂度为 $O(\log n)$ ，由于需要进行 k 次二分查找，因此总的时间复杂度为 $O(k \log n)$ 。由于在二分查找时并不需要额外的空间，因此空间复杂度为 $O(1)$ 。

我们可以对上述朴素的方法进行改进以提高效率。先对 k 个列表中的元素进行预处理以得到一组辅助信息，利用该信息可以提高后续查询的效率。相较于朴素的做法，每次都需要对每个列表查询一遍，我们可以先获得以下信息：以第 i 个列表中的第 j 个元素为目标值，得到该目标值在 k 个列表中的查询结果。也就是说，将第 i 个列表中第 j 个元素的值当成整数 x ，依次查询 k 个列表，共获得 k 个序号，这 k 个序号依次对应各个列表中大于或等于第 i 个列表中第 j 个元素的值的最小整数的序号。然后，我们将 k 个列表合并起来，将所有元素再进行一次排序，这时候，对于给定的整数 x ，我们可以在这个更大的有序列表中只进行一次二分查找，这样能够得到一个序号，该序号对应的元素是大于等于 x 的最小整数的序号 y 。结合之前进行的预处理信息，我们可以根据序号 y 所对应的元素立即得到一个列表，该列表所对应的就是在更大的列表中序号为 y 的整数在 k 个列表中的查询结果，根据该查询结果，可以很容易得到整数 x 在这个 k 个列表中的相应查询结果。

上述方法只需对合并得到的更大列表进行一次二分查找，该更大的列表的大小为 kn ，因此时间复杂度为 $O(\log(kn))$ ，相对于朴素的做法，查询效率有了很大的提升。由于需要为 k 个列表中的每个元素保存其在

各自列表中的查询结果，需要额外 $O(k^2n)$ 的空间来存储预处理结果。预处理的时间复杂度为 $O(k^2n \log n)$ 。

利用分散层叠 (fractional cascading) 技巧，我们能够结合前述两种方法的优点，既能有较小的空间复杂度，又有较小的时间复杂度。

4.9 算法库函数

4.9.1 binary_search

`binary_search` 使用二分查找算法在数组中寻找指定的元素值是否存在，要求数组中的元素已经按照严格不递减序排列。函数声明为：

```
template <class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last, const T& item);
```

当在序列中找到指定的元素时，返回 `true`，否则返回 `false`。SGI 库中的 `binary_search` 功能实现，实际上是通过 `lower_bound` 来完成的，读者可以尝试在理解 `lower_bound` 实现的基础上使用 `lower_bound` 来完成 `binary_search` 的功能。

```
-----4.9.1.cpp-----
int main(int argc, char *argv[]) {
    int numbers[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
    cout.setf(ios::boolalpha);
    cout << binary_search(numbers, numbers + 10, 5) << endl;
    cout << binary_search(numbers, numbers + 10, 100) << endl;
    return 0;
}
-----4.9.1.cpp-----
```

输出为：

```
true
false
```

强化练习：[11057 Exact Sum^A](#)，[11678 Exchanging Cards^C](#)，[12239 Bingo^C](#)。

4.9.2 find

返回给定序列的指定范围与目标值相等的第一个元素位置。函数声明为：

```
template <class InputIterator, class T>
InputIterator find(InputIterator first, InputIterator last, const T& val);
```

若查找成功，返回找到的第一个匹配元素的迭代器位置，否则返回指向给定范围结束位置的迭代器。用于获取元素在序列中的序号。

```
-----4.9.2.cpp-----
int main(int argc, char *argv[]) {
    string months[12] = {
        "January", "February", "March", "April", "May",
        "June", "July", "August", "September", "October",
        "November", "December"
    };
    string weekdays[7] = {
        "Monday", "Tuesday", "Wednesday", "Thursday",
        "Friday", "Saturday", "Sunday"
    };
}
```

```

    };
    int monthIndex = find(months, months + 12, "July") - months;
    cout << monthIndex << endl;
    int weekdayIndex = find(weekdays, weekdays + 7, "Someday") - weekdays;
    cout << weekdayIndex << endl;
    return 0;
}
//-----4.9.2.cpp-----//

```

输出为：

```

6
7

```

强化练习：[10528 Major Scales^c](#)

4.9.3 lower_bound 和 upper_bound

返回序列指定范围内第一个大于或等于（不小于）指定值的元素位置。函数声明为：

```

template <class ForwardIterator, class T>
ForwardIterator lower_bound
(ForwardIterator first, ForwardIterator last, const T& val);

```

使用该函数时，要求指定范围内的元素已经有序，查找范围是 $[first, last)$ ，区间左闭右开。默认使用小于运算符进行比较操作，也可以指定一个特定的比较函数。当未找到符合要求的元素位置时，返回 *last*。注意此 *last* 可能已经在序列范围之外。如果找到满足要求的元素位置，将其减去容器序列的起始地址即可得到该元素在序列中的序号。观察 *lower_bound* 的 SGI 库实现，可以发现其应用了二分查找算法。

```

template<typename _ForwardIterator, typename _Tp, typename _Compare>
_FORWARD_ITERATOR
__lower_bound(_ForwardIterator __first, _ForwardIterator __last,
  const _Tp & __val, _Compare __comp) {
  _typedef typename
    iterator_traits<_ForwardIterator>::difference_type _DistanceType;
  // 获得查找区间的长度。
  _DistanceType __len = std::distance(__first, __last);
  // 反复缩小查找区间，直到查找区间长度缩减为 0。
  while (__len > 0) {
    // 将查询区间减半，设置查找的中间位置。
    _DistanceType __half = __len >> 1;
    _ForwardIterator __middle = __first;
    std::advance(__middle, __half);
    // 如果中间值小于目标值，选择右半区间，否则选择左半区间，同时缩减查询区间长度。
    if (__comp(__middle, __val)) {
      __first = __middle;
      ++__first;
      __len = __len - __half - 1;
    }
    else
      __len = __half;
  }
  // 需要注意，若未查询到满足要求的值，则返回的是查找范围的结束位置。
  return __first;
}

```

在解题应用中，使用 `lower_bound` 可以方便的查找出所需要的值，不需编写容易出错的二分查找算法实现。

```
//-----4.9.3.1.cpp-----
int main(int argc, char *argv[]) {
    vector<int> numbers;
    for (int i = 1; i <= 10; i++) numbers.push_back(i);
    auto it = lower_bound(numbers.begin(), numbers.end(), 5);
    if (it != numbers.end()) {
        for (int i = 0; i <= (it - numbers.begin()); i++)
            cout << numbers[i] << " ";
        cout << endl;
    }
    it = lower_bound(numbers.begin(), numbers.end(), 20);
    if (it == numbers.end())
        cout << "I have searched to the end but found none." << endl;
    return 0;
}
//-----4.9.3.1.cpp-----//
```

输出为：

```
1 2 3 4 5
I have searched to the end but found none.
```

强化练习: 914 Jumping Champion^B, 1237 Expert Enough^A, 10125 Sumsets^A, 10706 Number Sequence^B, 12192 Grapevine^C。

`upper_bound` 返回序列指定范围内第一个大于指定值的元素位置。函数声明为：

```
template <class ForwardIterator, class T>
ForwardIterator upper_bound
(ForwardIterator first, ForwardIterator last, const T& val);
```

使用该函数时，要求指定范围内的元素已经有序，查找范围是 $[first, last)$ ，区间左闭右开。默认使用小于运算符进行比较操作，也可以指定一个特定的比较函数。当未找到符合要求的元素位置时，返回 `last`。注意此 `last` 可能已经在序列范围之外。如果找到满足要求的元素位置，将其减去 `first` 即可得到该元素在序列中的序号。SGI 库中 `upper_bound` 的实现和 `lower_bound` 类似，只不过在比较时，是比较目标值是否小于中间值（在 `lower_bound` 的实现中是比较中间值是否小于目标值，正好相反），如果目标值小于中间值，表明还可能有更小的值满足要求，因此选择的是左半区间，否则选择右半区间。

```
//-----4.9.3.2.cpp-----
int main(int argc, char *argv[]) {
    vector<int> numbers;
    for (int i = 1; i <= 10; i++) numbers.push_back(i);
    auto it = upper_bound(numbers.begin(), numbers.end(), 5);
    if (it != numbers.end()) {
        for (int i = it - numbers.begin(); i < numbers.size(); i++)
            cout << numbers[i] << " ";
        cout << endl;
    }
    it = upper_bound(numbers.begin(), numbers.end(), 20);
    if (it == numbers.end())
        cout << "I have searched to the end but found none." << endl;
}
//-----4.9.3.2.cpp-----//
```

```

    return 0;
}
//-----4.9.3.2.cpp-----/

```

输出为：

```

6 7 8 9 10
I have searched to the end but found none.

```

10049 Self-Describing Sequence^A (自描述序列)

Solomon Golomb 的自描述序列 $\langle f(1), f(2), f(3), \dots \rangle$ 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

编写程序，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 ($1 \leq n \leq 2000000000$)。输入以 $n=0$ 结束，你不应处理这一行。

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

```

分析

有两种解题方法：一种是非递推方法，即根据序列特点得到指定的 $f(n)$ 值；另外一种是递推方法，即先得到递推关系式，然后进一步根据递推关系计算指定的 $f(n)$ 值。

非递推方法：显式地生成这个序列的全部元素，显然会超出内存限制，因此需要使用特定的技巧，在内存空间有限的情况下完成这个任务。观察自描述序列不难得知，当 $n \geq 3$ 时，对于序列中的每一对元素 $(n, f(n))$ ，必将在序列中添加 $f(n) \times n$ 个元素。那么可以使用一个队列来记录将要添加的元素，同时计数拟将添加的元素总数，直到该队列将要产生的元素数量超过指定数量，在此过程中根据自描述序列的特点比对队列两端的自变量值以确定输入的函数值。该方法的关键是注意到以下事实：随着自描述序列往后延伸，有越来越多的数的函数值相同且相邻。在具体生成序列时，后续的元素并不需要实际添加到队列中而只需记录其自变量变化的范围即可。

参考代码

```

// 表示输入的结构体。
struct data { int idx, n, fn; };
bool cmp1(data d1, data d2) { return d1.n < d2.n; }
bool cmp2(data d1, data d2) { return d1.idx < d2.idx; }

```

```

// 表示输入的向量。
vector<data> Xs;
void getFn() {
    int idx = 0;
    // 输入中尚未确定函数值的元素个数。
    int unsetted = Xs.size();
    // 处理n=1和n=2时的特殊情形。
    while (Xs[idx].n <= 2) {
        Xs[idx].fn = Xs[idx].n;
        idx++;
        unsetted--;
    }
    if (!unsetted) return;
    queue<int> Q;
    // headn 为队列前端所对应的函数自变量值。
    // total 为队列中的现有元素根据自描述序列的特点最终能够产生的元素总数。
    // 由于前述已经处理了n=1和n=2时的特殊情形，故total的初值为2。
    int headn = 3, total = 2;
    // 压入初始元素，即f(3)=2。
    Q.push(2);
    // 当队列中的现有元素能够生成的元素总数小于输入中的最大值时，继续添加元素。
    while (total < Xs.back().n) {
        int fn = Q.front(); Q.pop();
        // 根据自描述序列的定义，将fn个headn压入队列。
        for (int i = 1; i <= fn; i++) Q.push(headn);
        // 比较输入和当前自变量是否匹配。
        if (Xs[idx].n == headn) {
            Xs[idx++].fn = fn;
            unsetted--;
        }
        if (!unsetted) return;
        // 计数队列总共能够产生的元素总数。
        total += fn * headn, headn++;
    }
    // 确定当前队列末尾元素所对应的自变量值。
    int endn = headn + Q.size();
    while (Q.size()) {
        int fn = Q.front(); Q.pop();
        // 比较输入和队列前端的自变量是否匹配。
        if (Xs[idx].n == headn) {
            Xs[idx++].fn = fn;
            unsetted--;
        }
        // 确定是否有输入在区间[endn, endn+fn)，根据自描述序列的定义，
        // 在此区间内的整数的函数值均为headn。
        for (int i = idx; i < Xs.size(); i++) {
            if (endn <= Xs[i].n && Xs[i].n <= (endn + fn - 1)) {
                Xs[i].fn = headn;
                unsetted--;
            }
        }
        // 所有输入的函数值均已确定，退出。
        if (!unsetted) return;
        // 更改队列前端和尾端所对应的自变量值。注意，并不需要在队列末端实际添加元素。
        // 这是非递推方法的关键，否则会造成超出内存。
    }
}

```

```

        endn += fn, headn++;
    }
}

int main(int argc, char *argv[]) {
    int idx = 0, n;
    while (cin >> n, n) Xs.push_back(data{idx++, n, 0});
    // 排序, 从小到大确定输入的函数值。
    sort(Xs.begin(), Xs.end(), cmp1);
    getFn();
    // 恢复输入顺序。
    sort(Xs.begin(), Xs.end(), cmp2);
    for (int i = 0; i < Xs.size(); i++) cout << Xs[i].fn << '\n';
    return 0;
}

```

递推方法: 将满足 $f(x-1) < f(x)$ 的自变量 x 值从小到大排成一个序列, 令该序列为 G , 从 1 开始计数, 易知 $G(1)=2$, $G(2)=4$, $G(3)=6$, $G(4)=9$, $G(5)=12$, \cdots 。为了便于问题的讨论, 不妨令 $G(0)=1$ 。由自描述序列的定义, 当 $n \geq 1$ 时, 对于自变量集合 $X=\{x \mid G(n-1) \leq x < G(n), x \in \mathbb{Z}\}$, 有 $|X|=f(n)$, 且对于任意 $x \in X$, 有 $f(x)=n$ 。进一步不难推出

$$G(n) = G(n-1) + f(n), \quad n \geq 1$$

由于已知区间 $[G(n-1), G(n))$ 内的整数 x 的数量为 $f(n)$, 且对应的自描述序列函数值 $f(x)$ 均为 n , 可以通过递推的方法得到序列 G , 再进一步根据序列 G 的特点通过库函数 `upper_bound` 得到给定输入 n 的函数值 $f(n)$ 。

参考代码

```

const int MAXN = 2000000000;
int G[700000], capacity = 0, prepared = 0;
int getFn(int n) {
    if (!prepared) {
        G[0] = 1, G[1] = 2, G[2] = 4;
        int idx = 0;
        while (G[G[idx] - 1] < MAXN) {
            for (int j = G[idx]; j < G[idx + 1]; j++)
                G[j] = G[j - 1] + idx + 1;
            idx++;
        }
        capacity = G[idx] - 1;
        prepared = 1;
    }
    return upper_bound(G, G + capacity, n) - G;
}
int main(int argc, char *argv[]) {
    int n;
    while (cin >> n) cout << getFn(n) << '\n';
    return 0;
}

```

强化练习: 406 Prime Cuts^A, 1152 4 Values Whose Sum is 0^C, 10427 Naughty Sleepy Boys^B, 10487 Closest Sums^A, 10567 Helping Fill Bates^B, 10611 The Playboy Chimp^A。

4.9.4 nth_element

该函数的作用是重排数组的元素, 使得数组序号为 n 的元素恰好为排序好的数组中的第 n 小元素。注

意，是从0开始计数序号，即第0小的元素就是数组的最小元素。其函数声明为：

```
template <class RandomAccessIterator>
void nth_element
(RandomAccessIterator first, RandomAccessIterator nth, RandomAccessIterator last);
```

默认使用小于比较运算符对元素进行比较，也可以自定义比较函数。注意表示数组范围的参数是第一个和第三个参数。由于函数采用了优化算法，其他位置的元素可能不是有序排列的，但是位于其前的元素均小于第n元素，位于其后的元素均大于第n元素。

```
//-----4.9.4.cpp-----
int main(int argc, char *argv[]) {
    string line = "bdfejgihca";
    nth_element(line.begin(), line.begin() + 5, line.end());
    cout << line << endl;
    return 0;
}
//-----4.9.4.cpp-----//
```

输出为：

```
dbacefghij
```

由输出不难看出，按照ASCII码值排列，位于序号5的元素恰为第5小的元素——字符‘f’，位于序号5之前的字符‘d’、‘b’、‘a’、‘c’、‘e’均小于字符‘f’，位于序号5之后的字符‘g’、‘h’、‘i’、‘j’均大于字符‘f’，在输出中小于字符‘f’的元素并未完全有序。

强化练习：501 Black Box⁸。

4.9.5 partial_sort

部分排序函数，排序后在指定范围之前的元素是整个序列中最小的，而且按升序排列，而在指定范围后其顺序未定。其函数声明为：

```
template <class RandomAccessIterator>
void partial_sort
(RandomAccessIterator first, RandomAccessIterator middle, RandomAccessIterator last);
```

部分排序在SGI的库实现中是使用堆排序予以实现的，以下是部分排序的使用示例。

```
//-----4.9.5.cpp-----
int main(int argc, char *argv[]) {
    string line = "987654321";
    cout << line << endl;
    partial_sort(line.begin(), line.begin() + 5, line.end());
    cout << line << endl;
    return 0;
}
//-----4.9.5.cpp-----//
```

输出为：

```
987654321
123459876
```

4.9.6 sort

sort 的作用是将数组或容器指定范围内的元素进行排序。它是一个模板函数，其声明如下：

```
template <class RandomAccessIterator>
void sort(RandomAccessIterator first, RandomAccessIterator last);

template <class RandomAccessIterator, class Compare>
void sort(RandomAccessIterator first, RandomAccessIterator last, Compare comp)
```

sort 可以使用自定义的比较函数来更改默认的排序顺序。各个版本的库实现中，sort 所采用的排序算法可能不是稳定排序算法，相同元素在排序前后位置可能会发生变化，若需要排序前后具有相同值的元素相对顺序不变，可以使用后续介绍的稳定排序函数 stable_sort。

```
//-----4.9.6.1.cpp-----
int main(int argc, char *argv[]) {
    int sample[10] = { 97, 3, 7, 13, 51, 23, 29, 17, 11, 83 };
    for (int i = 0; i < 10; i++) cout << sample[i] << " ";
    cout << endl;
    // 默认较小的数排序在前。
    sort(sample, sample + 10);
    for (int i = 0; i < 10; i++) cout << sample[i] << " ";
    cout << endl;
    // 使用模板函数 greater<int>() 指定大小关系，较大的数排序在前。
    sort(sample, sample + 10, greater<int>());
    for (int i = 0; i < 10; i++) cout << sample[i] << " ";
    cout << endl;
    return 0;
}
//-----4.9.6.1.cpp-----
```

输出为：

```
97 3 7 13 51 23 29 17 11 83
3 7 11 13 17 23 29 51 83 97
97 83 51 29 23 17 13 11 7 3
```

默认情况下，sort 使用小于比较运算符对元素进行比较操作，对于结构体等自定义的数据结构，可以通过重载小于运算符实现比较操作，也可以自行定义比较函数。

```
//-----4.9.6.2.cpp-----
struct record {
    int index, value;
    bool operator < (const record &x) const {
        if (value == x.value) return index < x.index;
        else return value < x.value;
    }
};

bool cmp(record x, record y) {
    if (x.value == y.value) return x.index > y.index;
    else return x.value > y.value;
}

int main(int argc, char *argv[]) {
    vector<record> records;
```

```

for (int i = 1; i <= 10; i++) records.push_back((record){i, i});
// 使用重载的小于运算符进行比较。
sort(records.begin(), records.end());
// 使用自定义的比较函数进行比较。
sort(records.begin(), records.end(), cmp);
return 0;
}
//-----4.9.6.2.cpp-----

```

158 Calader^D (日程表)

许多人都有日历，我们在上面速记一些日常生活中的重要事件——比如看牙医，参加瑞金特^I的 24 小时售书会或者程序竞赛等等。然而，也有一些固定的纪念日，例如伴侣的生日，结婚纪念日等，这些同样需要我们记在心上。一般来说，我们都希望在这些重要的日子临时能够得到提醒——事情越重要，提醒也越早越好。

现在请你编写程序实现重要事项到期提醒的功能。输入部分将会指定日程表需要处理的年份（在 1901 年至 1999 年之间）。请记住，在给定的年份之间，所有能被 4 整除的年份都是闰年，该年的 2 月份将是 29 天而不是 28 天。输出部分由“今天”的日期以及将要到期的日程表事件列表组成，每个事件都有相应的相对重要性提示。

输入

输入的第一行包含一个整数，表示所属年份（在 1901 至 1999 之间）。此后的若干行包含多个周年纪念日事件以及需要到期提醒服务的日期。

包含周年纪念日事件的行以字母“A”开始，后面跟随三个整数（D, M, P）表示日、月、事件的重要性，最后是事件的一个简短描述。它们以若干空格相分隔。P 是一个 1 到 7（包括 1 和 7）的整数，表示在事件到期之前的天数，在此天数之前日程表应该给出提醒服务。事件的简短描述总是会出现并且在事件优先级后第一个非空白字符开始。

包含日期的行以字母“D”开始，跟着是日数及月份。

在事件行之后的日期行数不定。所有行的长度不超过 255 个字符。输入文件以只包含字符“#”的一行结束。

输出

输出由一系列输出块组成，输入中的日期行对应一个输出块。每个输出块由指定需要提供提醒服务的日期以及后续的一系列事件描述组成。

输出需要指明事件的日期（D 和 M），以右对齐，宽度 3 输出，后跟事件的相对重要性。发生在指定日期当天的事件按照样例输出中的格式进行标记，发生在指定日期第二天的事件输出 P 个“*”号，在第三天的事件输出 P-1 个“*”号，依此类推。如果有多个事件在同一天发生，按照“*”号数量降序排列。

如果仍然不能确定输出的先后顺序，将事件按输入出现的顺序进行排列。按样例输出的格式进行输出，两个输出块之间输出一个空行。

样例输入

样例输出

^I Regent, 瑞金特, 图书销售公司, 总部位于美国新泽西州。

1993 A 23 12 5 Partner's birthday A 25 12 7 Christmas A 20 12 1 Unspecified Anniversary D 20 12 #	Today is: 20 12 20 12 *TODAY* Unspecified Anniversary 23 12 *** Partner's birthday 25 12 *** Christmas
--	---

分析

此题关键在于理解排序的规则，由于题目中的描述不是非常明确，容易得到 Wrong Answer 的结果。需要注意：(1) 对所有当天的事件按输入顺序排列；(2) 其他事件按 '*' 号的个数降序排列，如果 '*' 号数量相同，则按距离事件开始的天数升序排列；(3) 不显示 '*' 号的事件不进行输出；(4) 注意事件是周年性事件，可能有“跨年”的事件。

关键代码

```

// 记录事件的结构。
struct event {
    // days 为从 1901-01-01 开始的天数。
    int index, year, month, day, priority, days;
    string description;
};

vector<event> calendar;
int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
int todayDays;

// 将日期转换为从 1901-01-01 开始的天数以方便计算两个日期间隔的天数。
int dateToDays(int year, int month, int day) {
    int days = 0;
    for (int i = 1901; i <= year - 1; i++) days += (i % 4 == 0 ? 366 : 365);
    for (int i = 1; i <= month - 1; i++)
        days += daysInMonth[i - 1] + (i == 2 && year % 4 == 0 ? 1 : 0);
    days += day - 1;
    return days;
}

// 按题目要求对两个事件进行比较排序。
bool operator<(event x, event y) {
    if (x.days == y.days) {
        if (x.days == todayDays) return x.index < y.index;
        else return x.priority > y.priority;
    }
    else if (x.days != todayDays && y.days != todayDays) {
        if ((x.priority - x.days) != (y.priority - y.days))
            return (x.priority - x.days) > (y.priority - y.days);
        else return x.days < y.days;
    }
    else return x.days < y.days;
}

```

强化练习: 181 Hearts^D, 390 Letter Sequence Analysis^D, 454 Anagrams^B, 511 Do You Know the Way to San Jose^D, 555 Bridge Hands^A, 630 Anagrams (II)^B, 638 Finding Rectangles^C, 1610 Party Games^C, 10258 Contest Scoreboard^A, 10763 Foreign Exchange^A, 10905 Children's Game^A, 11039 Building Designing^A,

[11242](#) Tour de France^A, [11321](#) Sort Sort and Sort^A, [11369](#) Shopaholic^A, [11588](#) Image Coding^B, [11804](#) Argentina^B, [11824](#) A Minimum Land Price^B, [12028*](#) A Gift from the Setter^D, [12210](#) A Match Making Problem^B, [12269*](#) Lawn Mower^C, [12346](#) Water Gate Management^D, [12406](#) Help Dexter^C, [12541](#) Birthdates^B, [12705](#) Breaking Board^D, [13113*](#) Presidential Election^D。

扩展练习: 226 MIDI Preprocessing^E, 1219 Team Arrangement^E。

4.9.7 stable_sort

与 sort 函数不同, 使用 stable_sort 排序的数组, 具有相同值的元素在排序前后其相对位置关系不变。其函数声明为:

```
template <class RandomAccessIterator>
void stable_sort(RandomAccessIterator first, RandomAccessIterator last);
```

为了示例 stable_sort 的效果, 定义一个结构, 在结构体中定义了变量来表示结构体实例的序号。

```
//-----4.9.7.cpp-----
struct node {
    int index, value;
    bool operator<(const node& x) const {
        return value < x.value;
    }
};

int main(int argc, char *argv[]) {
    vector<node> nodes;
    for (int i = 0; i < 10; i++) {
        nodes.push_back((node){i, i % 2 == 0 ? 5 : i});
        cout << nodes[i].index << "->" << nodes[i].value << " ";
    }
    cout << endl;
    stable_sort(nodes.begin(), nodes.end());
    for (int i = 0; i < 10; i++)
        cout << nodes[i].index << "->" << nodes[i].value << " ";
    cout << endl;
    return 0;
}
//-----4.9.7.cpp-----
```

输出为:

```
0->5 1->1 2->5 3->3 4->5 5->5 6->5 7->7 8->5 9->9
1->1 3->3 0->5 2->5 4->5 5->5 6->5 8->5 7->7 9->9
```

可以看到, 具有相同值的元素在排序前后其相对位置不变, 即序号小的具有相同值的元素仍排列在前。

强化练习: [612](#) DNA Sorting^A, [632](#) Compression (II)^D, [12015](#) Google is Feeling Lucky^A。

4.9.8 unique

unique 函数的作用是“删除”相邻重复的元素。函数声明为:

```
template <class ForwardIterator>
ForwardIterator unique(ForwardIterator first, ForwardIterator last);
```

之所以给删除加上了引号, 是因为 unique 并不会真正的删除元素, 只是把去掉相邻重复元素后的数

组元素往前移，返回相邻元素不重复的数组最末一个元素的地址。原因是由于 `unique` 函数的传入参数是前向迭代器，而 `unique` 函数并不知道迭代器指向何种类型的容器，所以它不能对容器进行更改，即不能对容器中的元素进行删除操作，但是它可以改变元素的值。需要注意，`unique` 函数去掉的是相邻的重复元素，而不是数组中所有发生重复的元素，如果两个元素重复但不相邻，仅使用 `unique` 无法到达去除所有重复元素的目的，需要将其排序后再使用 `unique` 函数才能达到目的。

```
-----4.9.8.cpp-----
int number[10] = { 2, 3, 5, 5, 7, 2, 2, 3, 3, 5 };

void display(string tip) {
    cout << tip;
    for (int i = 0; i < 10; i++) cout << " " << number[i];
    cout << endl;
}

int main(int argc, char *argv[]) {
    display(" no operation:");
    unique(number, number + 10);
    display(" after unique:");
    sort(number, number + 10);
    display(" after sort:");
    int n = unique(number, number + 10) - number;
    display(" after unique:");
    cout << "no duplicated:";
    for (int i = 0; i < n; i++) cout << " " << number[i];
    cout << endl;
    return 0;
}
-----4.9.8.cpp-----
```

输出为：

```
no operation: 2 3 5 5 7 2 2 3 3 5
after unique: 2 3 5 7 2 3 5 3 3 5
    after sort: 2 2 3 3 3 3 5 5 5 7
    after unique: 2 3 5 7 3 3 5 5 5 7
no duplicated: 2 3 5 7
```

由上例可知，在未排序前，对数组使用 `unique` 函数，数组中的元素个数并未发生变化，只是把相邻的重复元素移到数组末尾，但是数组中还是存在重复的元素值，不过这些重复元素值不再是相邻的。如果要去掉所有的重复值，那么需要对数组先排序，然后使用 `unique` 函数，将 `unique` 函数返回的地址减去数组的起始地址，以得到具有不重复元素的数组范围。

强化练习：13180 The Countess' Pearls^D。

4.10 小结

在编程竞赛中，一般不需要参赛者去具体实现某种排序算法，因为常见的编程语言已经将排序算法包括到算法库中。学习排序算法的重点是理解排序算法的思想以及如何在编程中灵活运用这些排序思想，这是一个难点和重点内容。与排序密切相关的是查找，查找一般是在数据已经排序的基础上进行，在数据有序的情况下，可以应用高效的二分查找。二分查找不仅是一种算法，更是一种解决问题的思想，通过将问题空间不断的二分，从而缩小了所需解决问题的规模，最后通过小的问题的解来构造得到更大问题的解。熟练掌握二

分查找算法及其思想对解题很有帮助。

第 5 章 算术与代数

他山之石，可以攻玉。
——《诗经·小雅·鹤鸣》

由于 C++ 的标准库尚不支持高精度整数，在遇到此类题目时，首先明确是否需要使用高精度整数。如果能够使用替代方法解决问题，则应该尽量使用替代方法，这样可以免去使用 C++ 实现高精度整数所需的时间。即使确实需要使用高精度整数，如果对 Java 比较熟悉的话，应优先使用 Java 的 `BigInteger` 类进行解题，因为 `BigInteger` 类提供了高精度整数的完善实现，而作为最后的备选方案才是自己实现一个高精度整数运算类。

5.1 割鸡焉用牛刀乎

“割鸡焉用牛刀”，现在一般用作“杀鸡焉用牛刀”，出自《论语·阳货篇》：子之武城，闻弦歌之声。夫子莞尔而笑，曰：“割鸡焉用牛刀”。子游对曰：“昔者偃也闻诸夫子曰：‘君子学道则爱人，小人学道则易使也。’”子曰：“二三子！偃之言是也。前言戏之耳。”

话说孔老夫子周游列国，来到他的学生子游（名偃）治理的武城，听到了武城人民在学习歌舞拨弄琴弦的声音，心里不禁有点不以为然，但又微笑着说：“治理武城这种小地方，用不着上礼乐这种神器吧，就像用宰牛刀来杀鸡一样，你子游有点小题大做了吧？！”子游有点不服气了，反驳道：“弟子曾经听老师教导过，如果君子学习了道理就会有仁爱之心，小人学习了道理便会容易听使唤，现在我让这些老百姓学习高雅的礼乐，完全是按老师您说的去实践呐，难道我所做的是错的吗？”孔老夫子可能感觉有点自己打自己脸了，赶紧圆场，对着自己的随从说：“你们几个小子啊，给我注意喽！子游说的很对，我前面只不过是跟他开个玩笑而已，呵呵……”

在多数时候，对于 UVa OJ 上看似需要高精度整数运算的题目，也应秉承“杀鸡焉用牛刀”的态度，首先看看是否能够使用替代的方法完成解题。如果确实需要高精度整数运算，先考虑使用简化的高精度整数实现，即只实现四则运算中的一种或两种。在实现简化的四则运算时，可以将整数的数位存储在一个 `string` 中。具体表示时，`string` 中的元素存储的是数位所对应的 ASCII 字符，即使用‘0’（ASCII 码值为 48）至‘9’（ASCII 码值为 57）的字符来表示数位上的 0 至 9。

加法

加法可按照逐个数位相加的方式来实现。先将两个加数从低位到高位右对齐，从低位开始相加，设基数为 b ，两个对应数位分别为 x 和 y ，低位向高位的进位为 c ，则 $x+y+c$ 模基数 b 的值为结果数位上的值， $x+y+c$ 除以基数 b 的值为向高位的进位值。为了能够处理负数的加法，可以将加法和减法进行互相转换。

```
//++++++5.1.cpp+++++++
string add(string, string);
string subtract(string, string);

// 十进制下的四则运算。
const int BASE = 10;

// 移除计算结果的前导 0。
void zeroJustify(string &number)
```

```

{
    while (number.front() == '0' && number.length() > 1)
        number.erase(number.begin());
}

// 两个整数的加法。
string add(string number1, string number2)
{
    // 将负数的加法转换为减法。如果参加运算的两个整数总是正整数，可忽略此部分。
    if (number1.front() == '-') return subtract(number2, number1.substr(1));
    if (number2.front() == '-') return subtract(number1, number2.substr(1));
    // 将结果保存在字符串 number1 中，为了相加方便，事先调整加数，使得第一个加数的数位
    // 个数总是大于第二个加数的数位个数。由于两个正数相加，和的数位个数最多为两个加数的
    // 数位个数较大值加一，可以预先分配存储空间以方便实现。
    if (number1.length() < number2.length()) number1.swap(number2);
    number1.insert(number1.begin(), '0');
    // 相加时从低位开始加。初始时进位为 0。由于字符串中保存的是数位的 ASCII 字符，
    // 需要做相应的转换，使之成为对应的数字值。当前的数位为模基数的值，进位则为除以
    // 基数的值。
    int carry = 0, i = number1.length() - 1, j = number2.length() - 1;
    for (; i >= 0; i--, j--) {
        int sum = number1[i] - '0' + (j >= 0 ? (number2[j] - '0') : 0) + carry;
        number1[i] = sum % BASE + '0';
        carry = sum / BASE;
    }
    // 移除前导 0。
    zeroJustify(number1);
    return number1;
}

```

强化练习：[713 Adding Reversed Numbers^A](#)，[10013 Super Long Sums^A](#)，[10018 Reverse and Add^A](#)，[10035 Primary Arithmetic^A](#)，[10198 Counting^A](#)。

减法

减法的实现和加法类似，区别是被减数的某个数位不够时需要向高位进行借位。为了方便减的操作，可以根据两个数的大小和正负预先调整被减数，使得被减数的绝对值总是大于或等于减数。

```

// 辅助函数，用于判断第一个数是否不小于第二个数。
// 比较数的大小时，如果数位不等，由于都是非负整数，数位多的肯定大于数位少的；
// 如果数位相同，则从高位至低位逐个数位来进行比较。
bool greaterOrEqual(string &number1, string &number2)
{
    if (number1.length() != number2.length())
        return number1.length() > number2.length();
    // 逐个数位进行比较。
    for (int i = 0; i < number1.length(); i++)
        if (number1[i] != number2[i])
            return number1[i] > number2[i];
    return true;
}

// 两个整数的减法。
string subtract(string number1, string number2)
{

```

```

// 将负数的减法转换为加法。如果参加运算的两个整数总是正整数，可忽略此部分。
if (number1.front() == '-') {
    number1 = add(number1.substr(1), number2);
    number1.insert(number1.begin(), '-');
    return number1;
}
// 比较被减数和减数的大小，如果被减数小于减数，调整两个数使得相减的操作便于实现。
// 如果减数大于被减数则交换两个数，置计算结果为负数。
int sign = 1;
if (!greaterOrEqual(number1, number2)) {
    sign = -1;
    number1.swap(number2);
    number1.insert(number1.begin(), '0');
}
// 逐位相减，不够的向高位借位。
int borrow = 0, i = number1.length() - 1, j = number2.length() - 1;
for (; i >= 0; i--, j--) {
    int diff = number1[i] - '0' - (j >= 0 ? (number2[j] - '0') : 0) - borrow;
    borrow = 0;
    if (diff < 0) {
        diff += BASE;
        borrow = 1;
    }
    number1[i] = diff + '0';
}
// 移除前导 0。
zeroJustify(number1);
// 设置计算结果的符号位。
if (sign == -1 && number1 != "0") number1.insert(number1.begin(), '-');
return number1;
}

```

强化练习：254 Towers of Hanoi^c。

乘法

乘法采用逐行相乘然后相加的方法实现，为了表示进位，将每次相乘的结果不断左移并相加来得到最后结果。

```

// 两个整数的乘法。
string multiply(string number1, string number2)
{
    // 处理负数的乘法。如果相乘的两个整数总是正整数，可忽略此部分。
    int sign = 1;
    if (number1.front() == '-') {
        sign = sign * (-1);
        number1.erase(number1.begin());
    }
    if (number2.front() == '-') {
        sign = sign * (-1);
        number2.erase(number2.begin());
    }

    // 预分配存储空间。
    string number3(number1.length() + number2.length(), 0);
    // 从最低位开始相乘。

```

```

int length1 = number1.length() - 1, length2 = number2.length() - 1;
for (int i = length1; i >= 0; i--) {
    for (int j = length2; j >= 0; j--) {
        int k = number3.length() - 1 - (length1 - i + length2 - j);
        number3[k] += (number2[j] - '0') * (number1[i] - '0');
        number3[k - 1] += number3[k] / BASE;
        number3[k] %= BASE;
    }
}
// 将数值转换为对应的数字字符。
for (int i = 0; i < number3.length(); i++) number3[i] += '0';
zeroJustify(number3);
// 增加符号位。
if (sign == -1 && number3 != "0") number3.insert(number3.begin(), '-');
return number3;
}

```

强化练习：748 Exponentiation^A。

除法

除法采用试除法，从高位开始除，如果被除数小于除数，则将其左移一位（相当于乘以基数），加上后续一位，直到大于等于除数或者后续数位不存在，如果试除数大于除数，则将试除数减去除数，直到小于除数，计数“减的次数”即为该位的商，之后除不尽的留给低位加权后继续除。注意余数的处理。

```

// 非负整数的除法。
pair<string, string> divide(string number1, string number2)
{
    string row, quotient, remainder;
    for (int i = 0; i < number1.length(); i++) {
        // 将试除数不断左移，加上被除数的对应位。
        row.push_back(number1[i]);
        quotient.push_back('0');
        // 去除未除尽数的前导零。
        zeroJustify(row);
        // 当试除数大于除数时，将对应位的商加 1 然后减去除数，重复此步骤直到试除数
        // 小于除数。
        while (greaterOrEqual(row, number2)) {
            quotient.back() += 1;
            row = subtract(row, number2);
        }
    }
    // 获取余数。
    remainder = row;
    // 去除前导零。
    zeroJustify(quotient);
    zeroJustify(remainder);
    // 返回结果。
    return make_pair(quotient, remainder);
}

```

强化练习：10527 Persistent Numbers^C，10814 Simplifying Fractions^B。

求模

如果两个数均为大整数，求模运算需要牵涉到大整数的乘法和除法，可参考前述给出的除法实现。如果被除数是大整数，而除数是一个能够使用 int（或 long long int）数据类型表示的数，对其进行求模

运算时，可以使用下述技巧。

```
int mod(string number1, int number2)
{
    int remainder = 0;
    for (int i = 0; i < number1.length(); i++) {
        remainder = remainder * BASE + (number1[i] - '0');
        remainder %= number2;
    }
    return remainder;
}
//+++++5.1.cpp+++++
```

强化练习: 1226 Numerical Surprises^C, 10176 Ocean Deep Make it Shallow^A, 10929 You Can Say 11^A,
11344 The Huge One^B, 11879 Multiple of 17^A。

扩展练习：995 Super Divisible Numbers^D。

268 Double Trouble^D (双重麻烦)

Hudonia 的特工部门对具有特殊性质的数字非常感兴趣。当 Hudonia 遇到麻烦时，这种兴趣显得最为强烈。他们寻找的数字具有如下的性质：当你对该数进行“右移”（将此数的最后一位数字放在该数的最前面）操作后得到的数恰是原数的两倍。例如， $X=42105263157894736_{10}$ 是一个双重麻烦数，因为当对 X 进行“右移”操作后会得到： $842105263157894736_{10}=2X$ 。

上例给出的 X 是十进制下的一个双重麻烦数。任何基数大于等于 2 的数制系统，都会具有类似的双重麻烦数。在二进制中（基数 $p=2$ ）， 01_2 和 0101_2 是双重麻烦数。注意，前导零在此种情况下是必需的，以便对数字进行右移操作后能够得到相应的倍数。

特工部门似乎只对数制系统中最小的双重麻烦数感兴趣。例如，在二进制中， 01_2 是最小的双重麻烦数。在十进制中（基数 $p=10$ ），最小的双重麻烦数是 052631578947368421_{10} 。作为一名具有爱国精神的 Hudonia 人，你被要求编写程序来实现以下功能：在给定基数 p 的情况下，确定在该数制系统中的最小双重麻烦数。

输入

输入包含一系列的整数，每行一个，直到输入文件结束。给定的每个整数均小于 200。

输出

对于输入中给出的每个基数 p ，输出该数制系统下最小的双重麻烦数（包括必需的前导 0）。在输出双重麻烦数时按照样例输出的格式进行输出，顺序和输入时给定的基数顺序一致。

在输出的末尾不需增加空行。使用十进制数来表示基数为 p 的数制系统中的数位，每个数位（包括最后一位）在输出时后面跟随一个空格。

样例输入

2
10
35

样例输出

```
For base 2 the double-trouble number is
0 1
For base 10 the double-trouble number is
0 5 2 6 3 1 5 7 8 9 4 7 3 6 8 4 2 1
For base 35 the double-trouble number is
11 23
```

分析

令所求的数字为 $d_1d_2d_3\cdots d_n$, $y=d_1d_2d_3\cdots d_{n-1}$, $x=d_n$, 根据题意有

$$\begin{aligned} 2 \times d_1d_2d_3\cdots d_n &= 2 \times (d_1d_2d_3\cdots d_{n-1})(d_n) \\ &= 2 \times (y)(x) \\ &= (d_n)(d_1d_2d_3\cdots d_{n-1}) \\ &= (x)(y) \end{aligned}$$

令基数为 p , 则有

$$2yp + 2x = xp^{n-1} + y$$

移项化简得

$$y = \frac{x(p^{n-1} - 2)}{2p - 1}$$

于是有

$$d_1d_2d_3\cdots d_n = (y)(x) = \frac{x(p^{n-1} - 2)}{2p - 1} \times p + x = x \times \frac{(p^n - 1)}{2p - 1}$$

由于 x 是基数为 p 的数字的最末一位数字, 很明显 x 不能为 0, 否则数字为 0, 不符合题意要求, 那么 x 只可能为 1, 2, ..., $p-1$ 中的某个数字。由于最后得到的是一个整数, 而 x 是整数, 那么要求

$$\frac{p^n - 1}{2p - 1}$$

必须为整数, 即 $p^n - 1$ 能够被 $2p - 1$ 整除。因为满足题意要求的数字最少为两位数, 那么可以从 $n=2$ 开始枚举, 直到找到符合要求的最小 n 值。

当找到 x 和 n 的值时, 即可根据其确定相应的数字。由于 $p^n - 1$ 的各个数位均为 $p-1$ (例如, $2^8 - 1 = 11111111_2$, $10^6 - 1 = 999999_{10}$), 可以根据这个特点简化运算, 不必先求 $p^n - 1$ 的具体值然后再乘以 x , 而是直接将数位上的 $p-1$ 与 x 相乘。在进行除法时, 从高位开始除, 未除尽的留给低位加权后继续除。最后去掉多余的前导 0, 输出结果。

参考代码

```
int main(int argc, char *argv[])
{
    // 读入基数。
    int base;
    while (cin >> base) {
        // 根据等式搜索数字的位数 n 和最末尾一位数字的值 x。
        int n = 2, x, pow = base * base;
        while (true) {
            bool found = false;
            for (x = 1; x <= (base - 1); x++) {
                int r = x * (pow - 1) % (2 * base - 1);
                if (r == 0) {
                    found = true;
                    break;
                }
            }
            if (found) break;
            pow = (pow * base) % (2 * base - 1);
            n++;
        }
        // 根据等式计算数字的值。
        // 数字的最低位保存在 vector 的起始, 最高位保存在 vector 的末尾。
    }
}
```

```

vector<int> digits(n);
int carry = 0;
for (int i = 0; i < digits.size(); i++) {
    int temp = (base - 1) * x + carry;
    digits[i] = temp % base;
    carry = temp / base;
}
if (carry) digits.push_back(carry);
// 做除法。
int borrow = 0;
for (int i = digits.size() - 1; i >= 0; i--) {
    int temp = borrow * base + digits[i];
    digits[i] = temp / (2 * base - 1);
    borrow = temp % (2 * base - 1);
}
// 移除多余的前导 0 然后输出。
while (digits.size() > n) digits.erase(digits.end() - 1);
cout << "For base " << base << " the double-trouble number is" << endl;
for (int i = digits.size() - 1; i >= 0; i--)
    cout << digits[i] << " ";
cout << endl;
}
return 0;
}

```

强化练习: 290 Palindroms^C, 424 Integer Inquiry^A, 619 Numerically Speaking^B, 997 Show the Sequence^D, 10992 The Ghost of Programmers^C。

扩展练习: 10430* Dear GOD^D, 10669* Three Powers^C。

5.2 他山之石, 可以攻玉

Java 中的 `BigInteger` 类支持大整数运算的相关功能^I。在比赛环境时间紧、压力大的情况下, 如果题目需要使用大整数且支持 Java, 优先考虑使用 Java 本身自带的大整数类^{II}。`BigInteger` 类位于包 `java.math` 中, 使用前需要引入相应的命名空间:

```
import java.math.BigInteger;
```

在对 `BigInteger` 类进行初始化时, 可以使用多种方式, 其中常用的方式是将一个十进制的字符串转换为大整数, 其相应的构造函数原型为:

```

// 将 BigInteger 的十进制字符串表示形式转换为 BigInteger。该字符串表示形式包括一个可选的
// 减号, 后跟一个或多个十进制数字序列。该字符串不能包含任何其他字符 (例如, 空格)。
BigInteger(String val);

// 将指定基数的 BigInteger 的字符串表示形式转换为 BigInteger。该字符串表示形式包括一个可
// 选的减号, 后跟一个或多个指定基数的数字。该字符串不能包含任何其他字符 (例如, 空格)。
BigInteger(String val, int radix);

```

^I 参阅: <https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html>, 2020。

^{II} Python (<https://www.python.org/>, 2020) 也内置支持大整数运算, 如果题目的结论是一个可以直接计算的表达式且在线评测平台支持 Python, 使用 Python 也是一个不错的选择。

如果给定一个整数 x , 可以使用 `BigInteger.valueOf(x)` 将其转换成 `BigInteger` 实例。由于 Java 不支持运算符的重载, 大整数间的运算都是通过类的方法予以实现。需要注意的是, 参与运算的参数都必需首先转换为大整数才能进行运算。

```
// 加法。
BigInteger add(BigInteger val);

// 减法。
BigInteger subtract(BigInteger val);

// 乘法。
BigInteger multiply(BigInteger val);

// 整除。除数为 0 时抛出异常。
BigInteger divide(BigInteger val);

// 带余数的整除。
BigInteger[] divideAndRemainder(BigInteger val);

// 求模。
BigInteger remainder(BigInteger val);

// 乘方。注意乘方的次数参数是一个整数, 不是大整数。
BigInteger pow(int exponent);

// 返回两个大整数的最大公约数。
BigInteger gcd(BigInteger val);

// 将 BigInteger 与指定的 BigInteger 进行比较, 当此 BigInteger 在数值上小于、等于或大于
// val 时, 分别返回 -1, 0, 1。
int compareTo(BigInteger val);

// 将大整数转换为十进制的字符串表示形式。
String toString();

// 将大整数转换为指定基数的字符串表示形式。
String toString(int radix);
```

10213 How Many Pieces of Land?^B (有多少块土地?)

给定一块椭圆形的土地, 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, 这样可以得到 $n(n-1)/2$ 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?

输入

输入第一行包含一个整数 s ($0 < s < 3500$), 表示测试数据的组数。接下来的 s 行每行包含一个整数 n ($0 \leq n < 2^{31}$), 表示边界上可以挑选的点数。

输出

对于每组数据输出一行, 该行包含一个整数, 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
```

分析

点的个数 n 和土地被划分的块数 $g(n)$ 之间的关系可以表示为以下的通项公式：

$$g(n) = \binom{n}{4} + \binom{n}{2} + 2 = \frac{1}{24}(n^4 - 6n^3 + 23n^2 - 18n + 24)$$

直接应用 Java 提供的 BigInteger 类解题即可。

参考代码

```
-----5.2.java-----//
import java.io.*;
import java.math.BigInteger;

public class Main {
    public static void main(String[] args) throws IOException {
        BufferedReader stdin =
            new BufferedReader(new InputStreamReader(System.in));
        String line;
        line = stdin.readLine();
        int cases = Integer.parseInt(line);
        // 循环读入测试数据，计算结果。
        while (cases > 0) {
            line = stdin.readLine();
            System.out.println(getPieces(line));
            cases--;
        }
    }

    public static BigInteger getPieces(String line) {
        BigInteger n = new BigInteger(line);

        // 按公式计算结果。
        BigInteger pieces = n.pow(4);
        pieces = pieces.subtract(n.pow(3).multiply(BigInteger.valueOf(6)));
        pieces = pieces.add(n.pow(2).multiply(BigInteger.valueOf(23)));
        pieces = pieces.subtract(n.multiply(BigInteger.valueOf(18)));
        pieces = pieces.add(BigInteger.valueOf(24));
        pieces = pieces.divide(BigInteger.valueOf(24));

        return pieces;
    }
}
-----5.2.java-----//
```

强化练习：288 Arithmetic Operations With Large Integers^D, 324 Factorial Frequencies^A, 367 Halting Factor Replacement Systems^E, 485 Pascal's Triangle of Death^A, 495 Fibonacci Freeze^A, 623 500!^A, 1647 Computer Transformation^D, 10083 Division^C, 10106 Product^A, 10183 How Many Fibs^A, 10193 All You

Need Is Love^A, 10359 Tiling^B, 10433 Automorphic Numbers^C, 10494 If We Were a Child Again^A, 10519 Really Strange^B, 10523 Very Easy^A, 10551 Basic Remains^B, 10925 Krakovia^A, 11185 Ternary^A, 11448 Who Said Crisis^B, 13108 Juanma and the Drinking Fountains^D。

扩展练习: 10023* Square Root^B, 10606* Opening Doors^D, 12143* Stopping Doom's Day^E。

除了 BigInteger 类, Java 还提供了支持高精度十进制小数运算的 BigDecimal 类, 其使用方法与 BigInteger 类似, 感兴趣的读者可以查阅官方文档以获得进一步的了解^I。

强化练习: 10464 Big Big Real Numbers^C, 11821 High-Precision Number^C。

5.3 高精度整数类的实现

如果比赛环境明确不允许使用 Java 的高精度整数类, 而解题又必须使用高精度整数, 那么就需要从头开始编写一个高精度整数类来实现相关的运算。

要实现一个高精度整数类, 首先需要考虑如何表示数位。最简单直观的方式是用每个数位对应的 ASCII 字符来表示, 但是这样表示在实现乘法和除法时的效率不是很高。由于 C++ 的 vector 具有动态数组的功能, 可以考虑使用其来表示数位。在表示数位时, 为了提高效率, 每四个数字一组, 将其作为高精度整数的一位来看待。实际上是把基数从原来的 10 变成了 10000。在以下的实现中, vector 的最末元素存储的是高精度整数的最高位, 首元素存储的是最低位 (实际上顺序也可以反过来, 只不过以这样的设定实现起来更为方便一些)。

高精度整数类定义

```
//++++++5.3.cpp+++++++
// 常量, 分别表示正数、负数、相等。
const int POSITIVE = 1, NEGATIVE = -1, EQUAL = 0;

class BigInteger
{
    // 重载输出符号。
    friend ostream& operator<<(ostream&, const BigInteger&);

    // 比较操作。
    friend int compare(const BigInteger&, const BigInteger&);
    friend bool operator<(const BigInteger&, const BigInteger&);
    friend bool operator<=(const BigInteger&, const BigInteger&);
    friend bool operator==(const BigInteger&, const BigInteger&);

    // 相关运算的实现: 加法、减法、乘法、除法、模、乘方、左移。
    friend BigInteger operator+(const BigInteger&, const BigInteger&);
    friend BigInteger operator-(const BigInteger&, const BigInteger&);
    friend BigInteger operator*(const BigInteger&, const BigInteger&);
    friend BigInteger operator/(const BigInteger&, const BigInteger&);
    friend BigInteger operator%(const BigInteger&, const BigInteger&);
    friend BigInteger operator^(const BigInteger&, const unsigned int&);
    friend BigInteger operator^(const BigInteger&, const BigInteger&);
    friend BigInteger operator<<(const BigInteger&, const unsigned int&);
```

^I 参阅: <https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>, 2020。

```

public:
    BigInteger() {}

    // 将长整型及字符串转换为高精度整数的构造函数。
    BigInteger(const long long&);
    BigInteger(const string&);

    // 获取高精度整数的最后一个数位值。
    int lastDigit() const { return digits.front(); }

    ~BigInteger() {}

private:
    // 清除运算产生的前导零。
    void zeroJustify(void);

    // 采用 10000 作为基数。数位宽度为 4。
    static const int base = 10000;
    static const int width = 4;

    // 符号位和保存数位的向量。
    int sign;
    vector<int> digits;
};

```

重载输出符号

首先完成重载输出符号的功能。当为负数时，输出一个负号，正数前不输出符号。由于最高位存储在向量的末尾，故需要逆序输出。

```

// 重载输出符号以输出大整数。
ostream& operator<<(ostream& os, const BigInteger& number)
{
    os << (number.sign > 0 ? "" : "-");
    os << number.digits[number.digits.size() - 1];
    for (int i = (int)number.digits.size() - 2; i >= 0; i--)
        os << setw(number.width) << setfill('0') << number.digits[i];
    return os;
}

```

构造函数

为了方便使用，定义了两个构造函数，将长整型数和十进制数字符串转换为高精度整数。转换时先确定符号位，若参数是整数类型，则将其不断除以基数，求模得到数位；若参数是一个十进制数字字符串，则从字符串末尾开始，每四位作为一组，将其转换为整数保存到数位 `vector` 中。以下示例代码使用了 C++11 标准的 `stoi` 函数来将字符串转换为整数。

```

// 将长整型数转换为大整数。
BigInteger::BigInteger(const long long& value)
{
    if (value == 0) {
        sign = POSITIVE;
        digits.push_back(0);
    }
    else {

```

```

// 不断模基数取余得到各个数位。
sign = (value >= 0 ? POSITIVE : NEGATIVE);
long long number = abs(value);
while (number) {
    digits.push_back(number % base);
    number /= base;
}
};

// 移除前导零。
zeroJustify();
};

// 将十进制的字符串转换为大整数。
BigInteger::BigInteger(const string& value)
{
    if (value.length() == 0) {
        sign = POSITIVE;
        digits.push_back(0);
    }
    else {
        // 设置数值的正负号。
        sign = value[0] == '-' ? NEGATIVE : POSITIVE;
        // 四个数字作为一组，转换为整数存储到数位数组中。
        string block;
        for (int index = value.length() - 1; index >= 0; index--) {
            if (isdigit(value[index]))
                block.insert(block.begin(), value[index]);
            if (block.length() == width) {
                digits.push_back(stoi(block));
                block.clear();
            }
        }
        if (block.length() > 0) digits.push_back(stoi(block));
    }
    // 移除前导零。
    zeroJustify();
}

```

由于在运算中，可能在向量的末尾产生无效的 0，这些 0 位于最高位，因此需要去掉以免影响高精度整数的输出，在此定义了一个移除前导零的私有方法。

```

// 移除无效的前导零。
void BigInteger::zeroJustify(void)
{
    for (int i = digits.size() - 1; i >= 1; i--) {
        if (digits[i] > 0) break;
        digits.erase(digits.begin() + i);
    }
    if (digits.size() == 1 && digits[0] == 0) sign = POSITIVE;
}

```

比较运算

为了更为方便的完成高精度整数的四则运算，先定义比较运算，此处通过一个函数返回两个高精度整数的大小关系，根据大小关系可进一步重载相应的比较运算符。

```

// 比较两个高精度整数的大小。
// x 大于 y, 返回 1, x 小于 y, 返回 -1, x 等于 y, 返回 0。
// 为了后续除法的需要调整了实现, 使得对于未经前导零调整的整数也能够得到正确的处理。
int compare(const BigInteger& x, const BigInteger& y)
{
    // 符号不同, 正数大于负数。
    if (x.sign == POSITIVE && y.sign == NEGATIVE ||
        x.sign == NEGATIVE && y.sign == POSITIVE)
        return (x.sign == POSITIVE ? 1 : -1);
    // 确定 x 和 y 的有效数位个数, 前导零不计入有效数位。
    int xDigitNumber = x.digits.size() - 1;
    for (; xDigitNumber && x.digits[xDigitNumber] == 0; xDigitNumber--) ;
    int yDigitNumber = y.digits.size() - 1;
    for (; yDigitNumber && y.digits[yDigitNumber] == 0; yDigitNumber--) ;
    // 符号相同, 同为正数, 数位个数越多则越大, 同为负数, 数位个数越多则越小。
    if (xDigitNumber > yDigitNumber) return (x.sign == POSITIVE ? 1 : -1);
    // 符号相同, 同为正数, 数位个数越少则越小, 同为负数, 数位个数越少则越大。
    if (xDigitNumber < yDigitNumber) return (x.sign == NEGATIVE ? 1 : -1);
    // 符号相同, 数位个数相同, 逐位比较。
    for (int index = xDigitNumber; index >= 0; index--) {
        if (x.digits[index] > y.digits[index]) return (x.sign == POSITIVE ? 1 : -1);
        if (x.digits[index] < y.digits[index]) return (x.sign == NEGATIVE ? 1 : -1);
    }
    // 两数相等。
    return 0;
}

// 等于比较运算符。
bool operator==(const BigInteger& x, const BigInteger& y)
{
    return compare(x, y) == 0;
}

// 小于比较运算符。
bool operator<(const BigInteger& x, const BigInteger& y)
{
    return compare(x, y) < 0;
}

// 小于等于比较运算符。
bool operator<=(const BigInteger& x, const BigInteger& y)
{
    return compare(x, y) <= 0;
}

```

加法

接下来实现高精度整数的加法和减法。为了实现的方便, 在必要时, 可将运算进行相互转换。为此, 需要事先声明函数原型以便编译器调用。加法的实现很简单, 按照逐位相加的方式进行。

```

BigInteger operator+(const BigInteger&, const BigInteger&);
BigInteger operator-(const BigInteger&, const BigInteger&);

// 高精度整数加法。
BigInteger operator+(const BigInteger& x, const BigInteger& y)

```

```

{
    BigInteger z;
    // 如果两个加数的符号不同，转换为减法运算。
    if (x.sign == NEGATIVE && y.sign == POSITIVE) {
        z = x;
        z.sign = POSITIVE;
        return (y - z);
    }
    else if (x.sign == POSITIVE && y.sign == NEGATIVE) {
        z = y;
        z.sign = POSITIVE;
        return (x - z);
    }
    // 确保 x 的位数比 y 的位数多，便于计算。
    if (x.digits.size() < y.digits.size()) return (y + x);
    // 两个加数的符号相同时才进行加法运算。预先为结果分配存储空间。
    z.sign = x.sign + y.sign >= 0 ? POSITIVE : NEGATIVE;
    z.digits.resize(max(x.digits.size(), y.digits.size()) + 1);
    fill(z.digits.begin(), z.digits.end(), 0);
    // 逐位相加，考虑进位。
    int index = 0, carry = 0;
    for (; index < x.digits.size(); index++) {
        // 获取对应位的和。
        int sum = x.digits[index] + carry;
        sum += index < y.digits.size() ? y.digits[index] : 0;
        // 确定进位。
        carry = sum / z.base;
        // 将和保存到结果的相应位中。
        z.digits[index] = sum % z.base;
    }
    // 保存最后可能产生的进位。
    z.digits[index] = carry;
    // 移除前导零。
    z.zeroJustify();
    return z;
}

```

减法

减法的实现和加法类似，也是从低位到高位进行，采用逐位相减的方法进行。与加法考虑进位相反，减法需要考虑的是借位。

```

// 高精度整数减法。
BigInteger operator-(const BigInteger& x, const BigInteger& y)
{
    BigInteger z;
    // 当 x 和 y 至少有一个是负数，转换为加法运算。
    if (x.sign == NEGATIVE || y.sign == NEGATIVE) {
        z = y;
        z.sign = -y.sign;
        return x + z;
    }
    // 都为正数，确保 x 大于 y，便于计算。
    if (x < y) {
        z = y - x;

```

```

        z.sign = NEGATIVE;
        return z;
    }
    // 设置符号位并预先分配存储空间。
    z.sign = POSITIVE;
    z.digits.resize(max(x.digits.size(), y.digits.size()));
    fill(z.digits.begin(), z.digits.end(), 0);
    // 逐位相减，考虑借位。
    int index = 0, borrow = 0;
    for (; index < x.digits.size(); index++) {
        // 获取对应位的差。
        int difference = x.digits[index] - borrow;
        difference -= index < y.digits.size() ? y.digits[index] : 0;
        // 确定是否有借位。
        borrow = 0;
        if (difference < 0) {
            difference += z.base;
            borrow = 1;
        }
        // 保存相应位差的结果。
        z.digits[index] = difference % z.base;
    }
    // 移除前导零。
    z.zeroJustify();
    return z;
}

```

乘法

乘法采用的是朴素的方法——即一行一行相乘然后相加。该种方法是学校教育在教授四则运算时进行乘法的通用做法，此方法不仅简单而且效率在程序竞赛中也可接受。

```

// 高精度整数乘法。
BigInteger operator*(const BigInteger& x, const BigInteger& y)
{
    BigInteger z;
    // 设置符号位并预先分配存储空间。
    z.sign = x.sign * y.sign;
    z.digits.resize(x.digits.size() + y.digits.size());
    fill(z.digits.begin(), z.digits.end(), 0);
    // 一行一行相乘然后相加。
    for (int i = 0; i < y.digits.size(); i++)
        for (int j = 0; j < x.digits.size(); j++) {
            z.digits[i + j] += x.digits[j] * y.digits[i];
            z.digits[i + j + 1] += z.digits[i + j] / z.base;
            z.digits[i + j] %= z.base;
        }
    // 移除前导零。
    z.zeroJustify();
    return z;
}

```

除法

除法的实现稍复杂，总体思路是使用除数去试除，如果被除数不够则向后扩展使得被除数增大，直到大

于或等于除数，此时使用减法来获取对应数位的商。为了提高效率，以下实现中所采用的是二分试除法。

```

// 高精度整数除法，为整除运算。
BigInteger operator/(const BigInteger& x, const BigInteger& y)
{
    // z 表示整除得到的商，r 表示每次试除时的被除数。
    BigInteger z, r;
    // 设置商和被除数的符号位。
    z.sign = x.sign * y.sign;
    r.sign = POSITIVE;
    // 为商 z 和表示被除数的 r 预先分配存储空间。
    z.digits.resize(x.digits.size() - y.digits.size() + 1);
    r.digits.resize(y.digits.size() + 1);
    // 初始化值。
    fill(z.digits.begin(), z.digits.end(), 0);
    fill(r.digits.begin(), r.digits.end(), 0);
    // 从高位到低位逐位试除得到对应位的商。
    for (int i = x.digits.size() - 1; i >= 0; i--) {
        // 获取被除数，将上一次未被除尽的余数移到高位加上当前数位继续除。
        r.digits.insert(r.digits.begin(), x.digits[i]);
        // 通过二分试除法得到对应位的商。
        int low = 0, high = z.base - 1, middle = (high + low + 1) >> 1;
        while (low < high) {
            if ((y * BigInteger(middle)) <= r)
                low = middle;
            else
                high = middle - 1;
            middle = (high + low + 1) >> 1;
        }
        // 执行减法，从被除数中减去指定数量的 y。
        for (int index = 0; index < y.digits.size(); index++) {
            int difference = r.digits[index] - middle * y.digits[index];
            // 确定是否有借位产生。
            int borrow = 0;
            if (difference < 0) borrow = (z.base - 1 - difference) / z.base;
            // 高位减去借位数量。
            r.digits[index + 1] -= borrow;
            // 低位加上借位。
            difference += z.base * borrow;
            r.digits[index] = difference % z.base;
        }
        // 将对应位的商存入结果中。
        z.digits.insert(z.digits.begin(), middle);
    }
    // 移除前导零。
    z.zeroJustify();
    return z;
}

```

求模

设有正整数 x 和 y ，有

$$x \% y = x - \left\lfloor \frac{x}{y} \right\rfloor * y$$

上式中的商向下取整，由于前述实现的除法是整除，因此可以使用整除来实现商向下取整。

```
// 高精度整数求模运算。适用于同为正整数的情形。
BigInteger operator%(const BigInteger& x, const BigInteger& y)
{
    return (x - (x / y) * y);
}
```

上述求模运算需要进行除法、乘法和减法，效率不是很高，若需要提高效率，可将前述的大整数除法运算进行适当扩展，使得在得到商的同时保留余数。注意为了统一，可以设定余数总是大于等于 0。

乘方

乘方运算可以转换为乘法，采用适当技巧可减少乘法次数。

```
// 高精度整数乘方运算，乘法次数为内置整数类型。
BigInteger operator^(const BigInteger& x, const unsigned int& y)
{
    if (y == 0) return BigInteger(1);
    if (y == 1) return x;
    if (y == 2) return x * x;
    if (y & 1 > 0) return ((x ^ (y / 2)) ^ 2) * x;
    else return ((x ^ (y / 2)) ^ 2);
}

const BigInteger ZERO = BigInteger(0), ONE = BigInteger(1), TWO = BigInteger(2);

// 高精度整数乘方运算，乘方次数为高精度整数。
BigInteger operator^(const BigInteger& x, const BigInteger& y)
{
    if (y == ZERO) return BigInteger(1);
    if (y == ONE) return x;
    if (y == TWO) return x * x;
    if (y.lastDigit() & 1 > 0) return ((x ^ (y / 2)) ^ 2) * x;
    else return ((x ^ (y / 2)) ^ 2);
}
```

左移

最后给出左移运算的实现，类似的，读者可以自行尝试实现右移运算。

```
// 高精度整数左移运算，左移一位相当于将此数乘以基数。
BigInteger operator<<(const BigInteger& x, const unsigned int& shift)
{
    BigInteger z;
    // 设置符号位，复制向量中的数据。
    z.sign = x.sign;
    z.digits.resize(x.digits.size());
    copy(x.digits.begin(), x.digits.end(), z.digits.begin());
    // 移动指定位数，补零。
    for (int i = 0; i < shift; i++) z.digits.insert(z.digits.begin(), 0);
    // 移除前导零。
    z.zeroJustify();
    return z;
}
//+++++5.3.cpp+++++
```

10247 Complete Tree Labeling^C (完全树标号)

完全 k 叉树是一种特殊的 k 叉树，它的所有叶子结点位于同一层，并且所有内部结点均有 k 个分支。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，包含一个整数，表示标号方案的数量。

样例输入

```
2 2
10 1
```

样例输出

```
80
3628800
```

分析

令 $T(k, d)$ 表示给分支因子为 k ，深度为 d 的 k 叉树进行标号的方案数， $N(k, d)$ 表示分支因子为 k ，深度为 d 的完全 k 叉树所包含的结点数，根据完全 k 叉树的定义，深度为 d 的完全 k 叉树包含了 k 个深度为 $d-1$ 的完全 k 叉子树，有

$$N(k, d) = k \times N(k, d-1) + 1$$

且每个子树的标号方案数为 $T(k, d-1)$ 。则可将问题转化为将 $N(k, d)-1$ 个结点，编号为 1 至 $N(k, d)-1$ ，划分为 k 个子集，每个子集有 $N(k, d-1)$ 个结点，总共有多少种划分方法。只要确定了划分方法数，由于已知 $T(k, d-1)$ ，只需将每种划分方法得到的子集和每棵子树一一对应，每个子集的元素可以与 $T(k, d-1)$ 中的标号形成一一对应。设总的划分方法数为 x ，那么有以下关系

$$T(k, d) = x \times T(k, d-1)^k$$

而集合的划分数 x 相当于每次从 $N(k, d)-1$ 个结点中取 $N(k, d-1)$ 个结点，共取 k 次所能得到的不同取法总数。设 $A=N(k, d)-1$, $B=N(k, d-1)$ ，有

$$x = \prod_{i=0}^{k-1} \binom{A - i \times B}{B}$$

根据组合数的定义可将上式化简为

$$x = \frac{A!}{(B!)^k}$$

最终有

$$T(k, d) = \frac{(N(k, d)-1)!}{(N(k, d-1)!)^k} \times T(k, d-1)^k$$

容易知道 $T(k, 0)=1$ ，进一步化简得

$$T(k, d) = \frac{(N(k, d) - 1)!}{\prod_{i=1}^{d-1} N(k, d - i)^{k^i}}$$

强化练习: 10220 I Love Big Numbers^A, 10254 The Priest Mathematician^B, 12924 Immortal Rabbits^E。

5.4 进制及其转换

在日常生活中, 人们使用的是十进制, 即逢十进一。之所以十进制应用这么广泛, 很可能是因为人类的手指和脚趾都是十个。在最初的时候, 人们是利用简单的数指头的方法来计数的, 如果人类进化时是八个手指, 那么我们现在可能用的就是八进制了。十进制在日常生活中应用广泛, 但是使用计算机来表示却非常不方便, 因为在最初计算机的制造中, 使用晶体管的开和关两种状态来表示 1 和 0 是很方便的, 但是要用晶体管来表示 0 到 9 这十种状态却非常麻烦, 因此在计算机中使用的都是二进制, 即逢二进一。除了二进制以外, 常用的还有八进制、十六进制等。

5.4.1 R 进制数转换为十进制数

将数从一种计数制表示转换到另外一种计数制表示的过程称为进制转换。将 R 进制数从左到右写成一行, 最左边的数字具有最高的权值, 最右边的数字具有最低的权值, 则 R 进制下的数 X_R 所对应的十进制数为

$$X_R = (x_n x_{n-1} \cdots x_2 x_1)_R = x_n R^{n-1} + x_{n-1} R^{n-2} + \cdots + x_2 R^1 + x_1 R^0 = ((x_n R + x_{n-1}) R + \cdots) R + x_1$$

例如将二进制数 1101101101_2 转换为十进制数, 有

$$1101101101_2 = 1 \times 2^9 + 1 \times 2^8 + \cdots + 0 \times 2^1 + 1 \times 2^0 = 877_{10}$$

在练习中, 经常会遇到的情况是给定了 R 进制下的一个数字字符串, 要求将其转换为十进制下的整数, 使用上述方法, 可以很容易实现。

```
-----5.4.1.cpp-----
// 将 R 进制数转换为十进制数, 假设转换得到的数值均在整数数据类型表示的范围内。
int convertRToDecimal(string textOfNumber, int base)
{
    // 处理符号位。
    int sign = 1;
    if (textOfNumber.front() == '+' || textOfNumber.front() == '-')
        sign = textOfNumber.front() == '+' ? 1 : -1;
    textOfNumber.erase(textOfNumber.begin());
}

// 将字符串表示的数字从左至右进行转换。
int number = 0;
for (auto digit : textOfNumber) number = number * base + (digit - '0');
// 返回的数值需要乘以符号位。
return sign * number;
}
-----5.4.1.cpp-----
```

强化练习: 377 Cowculations^B, 575 Skew Binary^A, 636 Squares (III)^A, 10093 An Easy Problem^A, 11398 The Base-1 Number System^C, 12602 Nice Licence Plates^A。

5.4.2 十进制数转换为 R 进制数

将十进制整数转换为其他进制整数, 一般使用求余法。假设将十进制的整数 m 转换为 R 进制数, 需要

做的就是确定 R 进制中每个数位的值，根据除法的定义，设 n 为 R 整除 m 的商， r 为余数，有

$$n = \left\lfloor \frac{m}{R} \right\rfloor, \quad m = nR + r$$

那么十进制数 m 所对应 R 进制数的末位数字就是 r ，将 m 替换为 n ，继续此过程直到确定每个数位上的数值即可完成转换。

```
-----5.4.2.cpp-----
// 将十进制数转换为 R 进制数。
string convertDecimalToR(int number, int base)
{
    // 处理符号位。
    int sign = number >= 0 ? 1 : -1;
    // 取绝对值，使用求余数法进行转换。
    string textOfNumber;
    number = abs(number);
    while (number) {
        textOfNumber.insert(textOfNumber.begin(), number % base + '0');
        number /= base;
    }
    // 当为负数时添加符号位。
    if (sign < 0) textOfNumber.insert(textOfNumber.begin(), '-');
    return textOfNumber;
}
-----5.4.2.cpp-----
```

如果给定的是一个十进制小数，要求将其转换为 R 进制小数，应该如何转换呢？只需要把求余操作改成乘法操作即可。根据数制定义，假设将十进制小数 $m_1.m_2m_3m_4$ 转换为 R 进制小数，则有

$$m_1.m_2m_3m_4 = n_1R^{-1} + n_2R^{-2} + \cdots + n_kR^{-k}$$

将两边同时乘以 R ，有

$$m_1.m_2m_3m_4 \times R = n_1 + n_2R^{-1} + \cdots + n_kR^{-(k-1)}$$

由于 n_1, n_2, \dots, n_k 为整数，而

$$n_2R^{-1} + \cdots + n_kR^{-(k-1)} \leq R^{-1} + \cdots + R^{-(k-1)} < 1$$

也就是说， n_1 和 $m_1.m_2m_3m_4 \times R$ 的整数部分相等， $n_2R^{-1} + \cdots + n_kR^{-(k-1)}$ 和 $m_1.m_2m_3m_4 \times R$ 的小数部分相等。

以十进制小数转化为二进制小数为例，具体做法为：用 2 乘以十进制小数，可以得到积，将积的整数部分取出，再用 2 乘余下的小数部分，又得到一个积，再将积的整数部分取出，如此反复操作，直到积的整数部分为 0 或 1，此时的 0 或 1 即为二进制的最后一位。如果转换得到的是循环小数，则达到所要求的精度或遇到循环时即可停止¹。例如，将 0.625_{10} 转化为二进制小数：

$0.625 * 2 = 1.25$ ，整数部分为 1； $0.25 * 2 = 0.5$ ，整数部分为 0； $0.5 * 2 = 1$ ，整数部分为 1；
可得 $0.625_{10} = 0.101_2$ 。

强化练习：11005 Cheapest Base^B，11121 Base -2^A ，11701 Cantor^D。

5.4.3 任意进制数之间的相互转换

在任意进制数之间进行转换，一般是借助于十进制数作为中介进行。设起始进制为 R_1 ，终止进制为 R_2 ，

¹ 读者可参阅本章后续小节所介绍的将分数转换为小数的过程。

先将 R_1 进制下的数 X 转换为十进制下的数 Y ，然后再将数 Y 转换为 R_2 进制下的数 Z 。

902 Password Search^A (密码搜索)

在第二次世界大战期间，能够发送经过加密的信息对盟军来说非常重要。信息一般是在经过某个固定的密码加密后才被发送出去。当然，固定的密码是不安全的，需要经常更改来保持私密性。不过，需要有一种机制来将新密码发送给对方。盟军密码学小组里的一位数学家提出了一个好点子：将密码隐藏在信息中一起发送。这样，信息的接收者只需要知道密码的长度，即可从接收的信息中将密码找出来。

如果密码的长度为 N ，将接收信息中长度为 N 的所有子串列出来，其中出现频次最高的子串即为密码。找到密码后，将信息中所有与密码相同的子串移除，剩下的即为经过加密后的有效信息，使用对应的密码进行解密即可。

现在要求你编写一个程序完成以下任务：给定密码的长度和经过加密的信息，按照前述的密码编码规则来找出密码。比如，在样例输入中，给定的密码长度为 3 ($N=3$)，加密后的信息文本为“baababacb”，则密码为“aba”。因为长度为 3 的子串中，“aba”出现的频次最高（出现了两次），而其他同等长度的六个子串均只出现了一次 (baa; aab; bab; bac; acb)。

输入

输入包含多组测试数据，每组测试数据一行。每行给出了密码的长度 N ($0 < N \leq 10$) 以及经过加密的信息，信息只由小写字母组成。

输出

对每组测试数据，输出找到的密码字符串。

样例输入

```
3 baababacb
```

样例输出

```
aba
```

分析

题目很简单，只需要统计子串的频次，取频次最高的子串即为密码字符串（可以使用 `map` 来解题，为了示例数制转换，以下解题使用的是将字符串映射为整数的解题方式）。可以将字符串映射为一个整数来进行唯一标识，将所有转换后得到的整数储存到一个 `vector` 中，对其排序，相同整数（子串）会相邻排列，逐次统计相同整数的最大长度，即可找到具有最大频次的整数（子串）。为何不用整数作为数组下标来直接累加次数呢？因为转换后的整数会很大，无法在内存限制下用数组予以存储，除非使用 `map`。题目限定信息只由小写字母构成，而小写字母共有 26 个，可将其视为一个 32 进制数值系统。给定字符串

$$S = s_1 s_2 \cdots s_n$$

将字母转换为 0 至 25 的数字，然后根据字母所在位置的权值，将其转换为一个整数

$$K = (s_1 - 97) \times 32^{n-1} + (s_2 - 97) \times 32^{n-2} + \cdots + (s_n - 97) \times 32^0$$

由于 N 最大为 10，故 K 不会超过 `long long int` 数据类型的表示范围。待找到最大频次的整数后，再将其转换为字母。字符串和整数的相互转换可应用位运算进行简化。

参考代码

```
// N 为密码的长度；message 为加密后的信息。
int N;
string message;
```

```

// 将整数转换为字符串并输出。
void decode(long long password)
{
    long long mask = 0x1F;
    for (int i = 2; i <= N; i++) mask <= 5;
    for (int i = N - 1; i >= 0; i--) {
        cout << (char)('a' + ((password & mask) >> (5 * i)));
        mask >>= 5;
    }
    cout << '\n';
}

int main(int argc, char *argv[])
{
    while (cin >> N >> message) {
        // 存储转换后得到的整数。
        vector<long long> substring;
        // 生成用于获取转换后整除的掩码。
        long long mask = 0x1F;
        for (int i = 2; i <= N; i++) mask <= 5, mask |= 0x1F;
        // 将字符串中所有长度为 N 的子串转换为整数。
        long long k = 0;
        for (int i = 0; i < N; i++) k <= 5, k |= (message[i] - 'a');
        substring.push_back(k);
        for (int i = N; i < message.length(); i++) {
            k <= 5, k &= mask, k |= (message[i] - 'a');
            substring.push_back(k);
        }
        // 排序，相同整数（子串）相邻。
        sort(substring.begin(), substring.end());
        // 统计出现频次最高的整数。
        int most = 0, frequency = 0;
        long long password = -1, head = -1;
        for (auto sub : substring) {
            if (sub == head) frequency++;
            else {
                if (frequency > most) most = frequency, password = head;
                if (password == -1) password = sub;
                head = sub, frequency = 1;
            }
        }
        // 将整数转换为字符串。
        decode(password);
    }
    return 0;
}

```

强化练习: [343 What Base Is This^A](#), [355 The Bases Are Loaded^A](#), [389 Basically Speaking^A](#), [10677* Base Equality^C](#)。

扩展练习: [11952* Arithmetic^D](#)。

5.4.4 罗马计数法

古罗马人使用一套特别的计数方法，称为罗马计数法（Roman numerals），该计数法使用如下的 7 个字母来表示特定的值：

$I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M = 1000$

然后通过相应的规则来组成数字，组成数字的规则有：

(1) 字母 I, X, C, M 连续出现的次数不能超过三次。例如，I, XX, CCC 是合法的表示，而 MMMM 是不合法的表示。

(2) 字母 V, L, D 连续出现的次数不能超过一次，即 VV, LL, DD 是不合法的表示。

(3) I 可以出现在 V 和 X 之前，表示将后面的数字减一，例如，IV 表示 4, IX 表示 9。

(4) X 可以出现在 L 和 C 之前，表示将后面的数字减十，例如，XL 表示 40, XC 表示 90。

(5) C 可以出现在 D 和 M 之前，表示将后面的数字减一百，例如，CD 表示 400, CM 表示 900。

(6) I, X, C, M 连续出现时，表示相加，例如，III 表示 3, VII 表示 7。

使用常规的字符串判断方法来判断给定的罗马数字是否合法有些繁琐，借助正则表达式可以简便地实现判断。

```
//++++++5.4.4.cpp+++++++
#include <regex>

string pattern = R"(^M{0,3}(CM|CD|D?C{0,3})(XC|XL|L?X{0,3})(IX|IV|V?I{0,3})$)";
regex romanExp(pattern, regex_constants::ECMAScript);

bool validateRoman(string &roman)
{
    return regex_match(roman, romanExp);
}
```

确定给定的罗马数字是合法的表示形式之后，可以很容易将其转换为阿拉伯数字。

```
map<char, int> letters = {
    {'I', 1}, {'V', 5}, {'X', 10}, {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000},
};

int roman2Arab(string &roman)
{
    int arab = 0, idx = 0;
    while (idx < roman.length() - 1) {
        int previous = letters[roman[idx]], next = letters[roman[idx + 1]];
        if (previous < next) arab += next - previous, idx += 2;
        else arab += previous, idx += 1;
    }
    if (idx < roman.length()) arab += letters[roman[idx]];
    return arab;
}
```

反之，将阿拉伯数字转换为罗马数字也很简单。

```
vector<int> numbers = {
    3000, 2000, 1000, 900, 500, 400, 300, 200, 100,
    90, 50, 40, 30, 20, 10,
    9, 8, 7, 6, 5, 4, 3, 2, 1
};

vector<string> symbols = {
    "MMM", "MM", "M", "CM", "D", "CD", "CCC", "CC", "C",
    "XC", "L", "XL", "XXX", "XX", "X",
    "IX", "VIII", "VII", "VI", "V", "IV", "III", "II", "I"
};
```

```

};

string arab2Roman(int arab)
{
    string roman;
    while (arab > 0) {
        for (int i = 0; i < numbers.size(); i++) {
            if (arab >= numbers[i]) {
                roman += symbols[i];
                arab -= numbers[i];
                break;
            }
        }
    }
    return roman;
}
//++++++5.4.4.cpp+++++++

```

强化练习：[185 Roman Numerals^C](#), [344 Roman Digititis^A](#), [759 The Return of the Roman Empire^C](#), [10101 Bangla Numbers^A](#), [11616 Roman Numerals^B](#), [12397 Roman Numerals^D](#)。

扩展练习：[276 Egyptian Multiplication^D](#), [943 Number Format Translator^E](#)。

5.5 实数

5.5.1 分数

分数 (fraction) 可分为有理分数和无理分数。有理分数可以使用整数作为分子和分母来表示。有理分数之间可以使用通分的方法来精确地比较大小。

为了简便, 可以使用自定义结构体或者 C++ 标准库中的 `complex` 类来表示分数。

```

//-----5.5.1.cpp-----//
struct fraction {
    // numerator 表示分子, denominator 表示分母。
    long long numerator, denominator;

    // 规范化分数的表示形式, 使得分子和分母互素且分母始终为正整数。
    void normalize()
    {
        if (denominator < 0) denominator *= -1, numerator *= -1;
        if (numerator == 0 && denominator != 0) denominator = 1;
        if (numerator != 0 && denominator != 0) {
            // __gcd 是 GCC 内置的求最大公约数的函数。
            long long g = __gcd(abs(numerator), denominator);
            numerator /= g, denominator /= g;
        }
    }

    fraction operator+(const fraction& f)
    {
        fraction r;
        r.numerator = numerator * f.denominator + denominator * f.numerator;
        r.denominator = denominator * f.denominator;
        r.normalize();
        return r;
    }
};

```

```
//-----5.5.1.cpp-----//
```

在上述代码片段中, 只是给出了加法的实现, 读者可以根据分数的运算法则自行实现减法、乘法和除法。如果是使用 `complex` 类来表示分数, 一般使用其实部 `real` 表示分子, 虚部 `imag` 表示分母。在进行分数的加法或减法时, 可以先求出两个分母的最大公约数, 然后使用下述运算方式以减少溢出的可能性。

```
fraction operator+(const fraction& f)
{
    fraction r;
    long long g = __gcd(denominator, f.denominator);
    r.numerator = f.denominator / g * numerator + denominator / g * f.numerator;
    r.denominator = denominator / g * f.denominator;
    r.normalize();
    return r;
}
```

10077 The Stern-Brocot Number System^A (Stern-Brocot 代数系统)

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

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

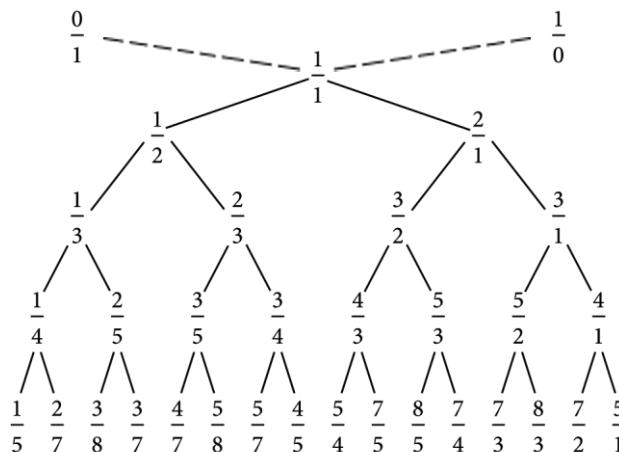
然后下一步将得到两个新分数

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

接下来是四个新分数

$$\frac{0}{1}, \frac{1}{3}, \frac{1}{2}, \frac{2}{3}, \frac{1}{1}, \frac{3}{2}, \frac{2}{1}, \frac{3}{1}, \frac{1}{0}$$

你可以将整个序列看作是一棵无限延伸的二叉树, 它的顶部看来是这样的:



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 $\frac{5}{7}$ 的一种表示方法。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。

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
RRLRRLRLLLLRLRRR
```

分析

起始时定义三个分数，左侧分数 $\frac{0}{1}$ ，中间分数 $\frac{1}{1}$ ，右侧分数 $\frac{1}{0}$ ，按照类似于中序遍历的过程，将给定的分数与中间分数比较大小后决定向左还是向右，根据 Stern-Brocot 树的规则更新相应的左侧分数和中间分数（或者中间分数和右侧分数），继续比较大小决定向左还是向右，重复此过程，直到找到指定的分数。在此过程中输出对应的向左或向右选择即为该分数的表示法。根据题目所给图形，以 $\frac{5}{7}$ 为例，可进行如下查找：

- (1) 起始时， $left = \frac{0}{1}$, $middle = \frac{1}{1}$, $right = \frac{1}{0}$ 。
- (2) $\frac{5}{7} < middle = \frac{1}{1}$, 走左侧子树，输出 L，更新 $left = \frac{0}{1}$, $middle = \frac{1}{2}$, $right = \frac{1}{1}$;
- (3) $\frac{5}{7} > middle = \frac{1}{2}$, 走右侧子树，输出 R，更新 $left = \frac{1}{2}$, $middle = \frac{2}{3}$, $right = \frac{1}{1}$;
- (4) $\frac{5}{7} > middle = \frac{2}{3}$, 走右侧子树，输出 R，更新 $left = \frac{2}{3}$, $middle = \frac{3}{4}$, $right = \frac{1}{1}$;
- (5) $\frac{5}{7} < middle = \frac{3}{4}$, 走左侧子树，输出 L，更新 $left = \frac{2}{3}$, $middle = \frac{5}{7}$, $right = \frac{1}{1}$;
- (6) $\frac{5}{7} = middle = \frac{5}{7}$, 查找过程结束。

或者使用根据矩阵运算简化得到的算法^[39]：假设分数为 $\frac{m}{n}$ ，则当 $m \neq n$ 时，如果 $m < n$ ，输出 ‘L’， $n = n - m$ ，否则输出 ‘R’， $m = m - n$ 。

仍以 $\frac{5}{7}$ 为例，有

$$\frac{m}{n} = \frac{5}{7} \xrightarrow{m < n} (L) \frac{5}{2} \xrightarrow{m > n} (R) \frac{3}{2} \xrightarrow{m > n} (R) \frac{1}{2} \xrightarrow{m < n} (L) \frac{1}{1} \xrightarrow{m = n} \text{结束}$$

相应输出为：L, R, R, L。

强化练习：[264 Count on Cantor^A](#), [493 Rational Spiral^C](#), [654 Ratio^C](#), [906 Rational Neighbor^D](#), [10408 Farey Sequences^B](#), [10976 Fractions Again^A](#), [11350 Stern-Brocot Tree^C](#), [12060 All Integer Average^C](#), [12502 Three Families^A](#), [12848 In Puzzleland \(IV\)^B](#), [12970 Alcoholic Pilots^D](#), [13071 Double Decker^D](#)。

扩展练习：[10437 Playing with Fraction^E](#), [11968* In the Airport^D](#), [12464* Professor Lazy Ph.D.^D](#), [12904 Load Balancing^D](#)。

5.5.2 连续分数

连续分数 (continued fraction) 是指具有以下形式的分数

$$x = a_0 + \cfrac{b_0}{a_1 + \cfrac{b_1}{a_2 + \cfrac{b_2}{a_3 + \dots}}}$$

其中 a_i 和 b_i 可以为有理数, 实数或者复数。如果 $b_i=1$ ($i \geq 0$), 则称为简单连续分数。如果连续分数包含的分数项有限, 则称为有限连续分数, 反之则称为无限连续分数。在表示简单连续分数时, 为了方便, 可以将其写成

$$x = [a_0; a_1, a_2, a_3, \dots]$$

任意有理数 r 均可按以下步骤将其表示成有限简单连续分数的形式: $a = \lfloor r \rfloor$, 若差值 $d = r - \lfloor r \rfloor = 0$, 停止, 否则取 $r = 1/d$, 重复上述步骤。

```
-----5.5.2.cpp-----
int main(int argc, char *argv[])
{
    int numerator, denominator;
    while (cin >> numerator >> denominator, denominator > 0) {
        cout << '[' << numerator / denominator;
        numerator %= denominator;
        bool printComma = false;
        while (numerator > 0) {
            if (printComma) cout << ',';
            else {
                cout << ';';
                printComma = true;
            }
            swap(numerator, denominator);
            cout << numerator / denominator;
            numerator %= denominator;
        }
        cout << "] \n";
    }
    return 0;
}
-----5.5.2.cpp-----
```

强化练习: [834 Continued Fractions^A](#)。

扩展练习: [10521 Continuously Growing Fractions^E](#), [11113 Continuous Fractions^D](#)。

5.5.3 分数转换为小数

如果一个分数的分子和分母均为整数且分子小于分母, 那么分数的值有可能是一个有限小数, 也有可能是一个无限循环小数 (infinite continued fraction), 如 $1/2$ 的值为 0.5 , $1/3$ 的值为 $0.3333\cdots$ 。当一个分数的值是一个无限循环小数时, 如何求它的最小循环节呢?

由于将分数转换为小数的过程是一个不断求余数的过程, 只要余数发生重复, 那么求得的商也必定开始重复, 可以利用此性质来求最小循环节。由于正整数 n 的余数只有 $0, 1, 2, \dots, n-1$ 共 n 种, 根据鸽巢原理, 以 n 为分母的分数, 对应的小数形式, 如果是循环小数, 则在第二个循环节开始前的小数位数最多不超过 n 位。

如果分子大于分母，可先进行整除取得整数部分的商，然后继续求小数部分。若分子为负数，可不考虑符号位，视为正数进行计算，最后输出时加上符号即可。

```
-----5.5.3.cpp-----
// 将有理分数转换为小数形式，若为无限循环小数则使用循环节的形式予以表示。
// 例如: 1/3=0.(3), 1789/1332=1.34(309)。
void printCycle(int numerator, int denominator)
{
    cout << numerator << '/' << denominator << '=';
    cout << (numerator / denominator) << '.';
    numerator %= denominator;
    // 特殊情况处理。
    if (numerator == 0) { cout << "0\n"; return; }
    // digits 存储小数数位, position 记录余数出现的位置, appeared 记录余数是否出现。
    vector<int> digits(denominator + 1), position(denominator + 1);
    vector<bool> appeared(denominator + 1);
    fill(appeared.begin(), appeared.end(), false);
    // 模拟除法来得到分数的小数表示。
    int index = 0;
    while (!appeared[numerator] && numerator > 0) {
        appeared[numerator] = true;
        digits[index] = 10 * numerator / denominator;
        position[numerator] = index++;
        numerator = 10 * numerator % denominator;
    }
    // 输出小数的非循环部分。
    int loopStart = 0;
    if (numerator > 0) {
        loopStart = position[numerator];
        for (int i = 0; i < position[numerator]; i++) cout << digits[i];
        cout << '(';
    }
    // 输出小数的循环部分。
    for (int i = loopStart; i < index; i++) cout << digits[i];
    if (numerator > 0) cout << ')';
    cout << '\n';
}
-----5.5.3.cpp-----
```

强化练习: 202 Repeating Decimals^A, 275 Expanding Fractions^B, 942 Cyclic Numbers^E, 13209 My Password is a Palindromic Prime Number^A。

5.5.4 小数转换为分数

给定一个有理数的小数形式（有限小数或者无限循环小数），可以按照以下方法将其转换为对应的分数形式。假设给定的小数 x 具有以下形式

$$x = 0.n_1n_2 \cdots n_k(r_1r_2 \cdots r_j) \cdots$$

其中 $n_1n_2 \cdots n_k$ 为小数的不循环部分， $r_1r_2 \cdots r_j$ 为小数的循环部分。例如: $7/22=0.3(18)\cdots$ ，其中不循环部分为 3，循环部分为 18。则 x 所对应的分数形式可以表示为

$$f = \frac{10^{k+j} \times x - 10^k \times x}{10^{k+j} - 10^k}$$

可以容易地将上述过程实现为以下代码。

```

//-----5.5.4.cpp-----//
// 将上述指定格式的小数转换为分数。
pair<long long, long long> getFraction(string fraction, int j)
{
    long long numerator, denominator;
    // 获取小数点之后的所有小数数位。
    size_t dot = fraction.find('.');
    if (dot != fractionnpos) fraction = fraction.substr(dot + 1);
    // 区分非循环小数和循环小数分别处理。
    if (j == 0) {
        numerator = stoll(fraction);
        denominator = pow(10, fraction.length());
    }
    else {
        int k = fraction.length() - j;
        string preRepeated = fraction.substr(0, k);
        if (preRepeated.length() == 0) preRepeated = "0";
        // 根据公式确定小数对应的分数表示。
        numerator = stoll(fraction) - stoll(preRepeated);
        denominator = pow(10, k + j) - pow(10, k);
    }
    // 对结果进行调整使得分数成为最简分数。
    long long g = __gcd(numerator, denominator);
    if (g > 1) numerator /= g, denominator /= g;
    return make_pair(numerator, denominator);
}
//-----5.5.4.cpp-----//

```

如果小数的循环部分较长（即分母为大整数），可以借助 Java 中的 `BigInteger` 类来完成转换。

强化练习：[332 Rational Numbers From Repeating Fractions^B](#), [10555 Dead Fraction^D](#)。

5.5.5 实数大小的比较

由于计算机内部采用浮点小数的形式来表示数学中的实数，存在一定的精度误差，使得有些数字无法被精确地表示。这样在比较实数的大小时，不能够简单地使用编程语言中的“大于”或“小于”运算来进行直接比较，而应该设定一个误差阈值（threshold），当两个数之间的差的绝对值小于此阈值时，就认为它们相等^[40]。

```

//-----5.5.5.cpp-----//
int main(int argc, char *argv[])
{
    double lower = -2.0, upper = 1.0, step = 0.05;
    double epsilon = 1e-7;

    int steps1 = 0, steps2 = 0;
    for (double i = lower; i <= upper; i += step) steps1++;
    for (double j = lower; j <= upper + epsilon; j += step) steps2++;
    cout << "steps1 = " << steps1 << " steps2 = " << steps2 << endl;
    return 0;
}
//-----5.5.5.cpp-----//

```

输出为：

```
step1=60 step2=61
```

上述代码的目的是计算在区间[−2.0, 1.0]内, 以 0.05 为步长进行递增时的总步数。如果不使用误差控制, 计数得到的结果为 60 步, 与实际的 61 步相差一步, 而采用误差控制后能够得到正确的步数。

以下给出在比较两个实数大小时常用的处理方法。

```
// 定义阈值。
const double epsilon = 1e-7;
double a, b;
// 比较 a 是否小于 b。
if (a + epsilon < b) {};
// 比较 a 和 b 是否相等。
if (fabs(a - b) <= epsilon) {};
```

强化练习: [918 ASCII Mandelbrot^D](#), [11001 Necklace^B](#), [12930 Bigger or Smaller^D](#)。

扩展练习: [697* Jack and Jill^D](#), [11816* HST^D](#)。

5.6 代数

5.6.1 多项式运算

多项式展开

与整数的乘法类似, 多项式展开 (polynomial expansion) 是将一个多项式的每一项和另外一个多项式的每一项相乘, 然后将对应次数项的系数相加得到另外一个多项式的过程。使用 `vector` 来表示多项式的系数很方便, 一般将高次项系数放在前面, 便于运算。以下是两个多项式展开的示例代码。

```
-----5.6.1.cpp-----
vector<int> multiply(vector<int> &a, vector<int> &b)
{
    vector<int> c(a.size() + b.size() - 1, 0);
    for (int i = 0; i < a.size(); i++)
        for (int j = 0; j < b.size(); j++)
            c[i + j] += a[i] * b[j];
    return c;
}
-----5.6.1.cpp-----
```

二项式定理 (binomial theorem) 和多项式定理 (multinomial theorem) 对于规则多项式幂的展开很有帮助。二项式定理可以表述为

$$(x + y)^n = \sum_{k=0}^n \binom{n}{k} x^{n-k} y^k = \sum_{k=0}^n \binom{n}{k} x^k y^{n-k}$$

设 n 是正整数, 对一切实数: x_1, x_2, \dots, x_k , 多项式定理可以表述为

$$(x_1 + x_2 + \dots + x_k)^n = \sum_{n_1+n_2+\dots+n_k=n} \binom{n}{n_1, n_2, \dots, n_k} \prod_{1 \leq i \leq k} x_i^{n_i}$$

其中

$$\binom{n}{n_1, n_2, \dots, n_k} = \frac{n!}{n_1! n_2! \dots n_k!}$$

强化练习: [927 Integer Sequences from Addition of Terms^B](#), [10105 Polynomial Coefficients^A](#), [10326 The Polynomial Equation^C](#), [11042 Complex Difficult and Complicated^C](#), [11955 Binomial Theorem^C](#)。

扩展练习: [10586 Polynomial Remains^C](#), [10719 Quotient Polynomial^B](#)。

因式分解

与多项式相乘恰好相反, 因式分解 (polynomial factorization) 的目标是将一个多项式分解为若干个多项式的乘积。因式分解一般在多项式的系数均为整数时进行。若要高效地进行分解, 首先需要了解有理根定理 (rational root theorem)。假设将多项式表示为以下形式

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0, \quad a_i \in \mathbb{Z}, \quad 0 \leq i \leq n$$

如果 a_n 和 a_0 不为零, 对于方程 $P(x)=0$ 的每个根, 可以证明: 根的最简形式一定可以表示成一个有理分数形式 $x=p/q$, 其中 $\gcd(p, q)=1$, 且 p 是 a_0 的约数, q 是 a_n 的约数。知道了方程的某个根 r , 则可以将多项式表示为

$$a_n x^n + a_{n-1} x^{n-1} + \cdots + a_1 x + a_0 = (x - r) Q(x)$$

其中 $Q(x)$ 为最高次数比 $P(x)$ 的最高次数少 1 的整数系数多项式。

强化练习: 126 The Errant Physicist^B, 498 Polly the Polynomial^A, 930 Polynomial Roots^E, 10268 498-Bis^B。

5.6.2 高斯消元法

在求解二元、三元一次方程组时, 常用的方法是加减消元法或代入消元法。例如, 给定以下的三元一次方程组

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & \textcircled{1} \\ 2x_1 + 3x_2 + x_3 = 12 & \textcircled{2} \\ 7x_1 - x_2 + 2x_3 = 0 & \textcircled{3} \end{cases} \quad (5.1)$$

使用消元法来解方程, 首先将 (5.1) 中方程①分别乘 -2 , -7 并依次加到方程②, ③上, 消去后两个方程中的未知数 x_1 , 得

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & \textcircled{1} \\ -x_2 - 9x_3 = -48 & \textcircled{2} \\ -15x_2 - 33x_3 = -210 & \textcircled{3} \end{cases} \quad (5.2)$$

然后将 (5.2) 的方程②乘以 -15 与方程③相加, 消去最后一个方程中的未知数 x_2 , 得

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & \textcircled{1} \\ -x_2 - 9x_3 = -48 & \textcircled{2} \\ 102x_3 = 510 & \textcircled{3} \end{cases} \quad (5.3)$$

从 (5.3) 的方程③易知 $x_3=5$, 将其回代入 (5.3) 的方程②可得 $x_2=3$, 然后将 x_3 、 x_2 代入 (5.3) 的方程①可得 $x_1=-1$ 。

以上即是高斯消元法 (Gaussian elimination) 在未知数较少的方程组上的应用, 而该方法也是用于求解一般的 m 个方程 n 个未知元的一次方程的通用方法。其基本思想是通过消元变形把方程组化成容易求解的同解方程组, 在消元的过程中, 不断地将未知元的系数化为零。不失一般性, 可以将方程组表示为以下的形式

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_n \end{cases} \quad (5.4)$$

为了使得消元过程书写简便, 可以将线性方程组中未知元对应的系数及方程右侧的常数项按顺序排成一张矩阵数表

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix} \quad (5.5)$$

其中 a_{ij} ($i=1, 2, \dots, m; j=1, 2, \dots, n$) 表示第 i 个方程第 j 个未知量 x_j 的系数, 则高斯消元法的消元过程可以在这张数表上进行。为了更为方便地描述高斯消元法和后续代码的实现, 先给出矩阵的定义^[41]。

数域 F 中 $m \times n$ 个数 a_{ij} ($i=1, 2, \dots, m; j=1, 2, \dots, n$) 排成 m 行 n 列, 并括以方括号的数表

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad (5.6)$$

称为数域 F 上的 $m \times n$ 矩阵, 通常记做 \mathbf{A} 或 $\mathbf{A}_{m \times n}$, 其中的 a_{ij} 称为矩阵 \mathbf{A} 的第 i 行第 j 列元素, 当 $a_{ij} \in \mathbb{R}$ (实数域) 时, \mathbf{A} 称为实矩阵; 当 $a_{ij} \in \mathbb{C}$ (复数域) 时, \mathbf{A} 称为复矩阵。当 $m=n$ 时, 称 \mathbf{A} 为 n 阶矩阵, 或者 n 阶方阵。线性方程组 (5.4) 对应的矩阵 (5.5) 称为方程组 (5.4) 的增广矩阵, 记做 (\mathbf{A}, b) , 其中由未知元系数排成的矩阵 \mathbf{A} 称为线性方程组的系数矩阵。根据矩阵的相关结论, 只有当系数矩阵的秩和增广矩阵的秩相等时, 方程组才有解, 否则无解。若系数矩阵和增广矩阵的秩均等于未知元的个数, 则方程组有唯一解。

高斯消元法解线性方程组的消元步骤可以在增广矩阵上进行。以列主元高斯消元法为例^I, 其具体步骤为: 选取第 i 个需要消去的未知数, 为了减少误差, 提高数值稳定性, 将此未知数具有最大系数的方程式与当前所处理的方程式交换, 如果当前方程式第 i 个未知数的系数不为零, 则将其系数调整为 1, 接着将其余方程式减去当前方程式的相应倍数以消去这些方程式中第 i 个未知数, 持续此步骤, 直到系数矩阵成为对角矩阵, 位于增广矩阵最右侧一列的数即为解。下面举例说明应用增广矩阵解线性方程组的方法。

求线性方程组

$$\begin{cases} x_1 + x_2 + x_3 + x_4 = 10 & ① \\ 3x_1 - 2x_2 - x_3 + 6x_4 = 20 & ② \\ 4x_1 - 3x_2 - x_3 + 2x_4 = 3 & ③ \\ x_1 - x_2 + 7x_3 + 5x_4 = 40 & ④ \end{cases} \quad (5.7)$$

线性方程组 (5.7) 的增广矩阵为

$$(\mathbf{A}, b) = \begin{bmatrix} 1 & 1 & 1 & 1 & | & 10 \\ 3 & -2 & -1 & 6 & | & 20 \\ 4 & -3 & -1 & 2 & | & 3 \\ 1 & -1 & 7 & 5 & | & 40 \end{bmatrix} \quad (5.8)$$

处理第一个方程式, 找到矩阵 (5.8) 中未知数 x_1 系数绝对值最大的行, 此处第③行的系数最大, 将其交换到①行并将系数调整为 1, 得

$$(\mathbf{A}, b) = \begin{bmatrix} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & | & \frac{3}{4} \\ 3 & -2 & -1 & 6 & | & 20 \\ 1 & 1 & 1 & 1 & | & 10 \\ 1 & -1 & 7 & 5 & | & 40 \end{bmatrix} \quad (5.9)$$

然后将矩阵 (5.9) 的②, ③, ④行分别减去①行的 3, 1, 1 倍, 消去所在行的未知数 x_1 , 得

^I 相应的还有数值稳定性更强的全主元高斯消元法。

$$(A, b) = \left[\begin{array}{cccc|c} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & \frac{3}{4} \\ 0 & \frac{1}{4} & -\frac{1}{4} & \frac{18}{4} & \frac{71}{4} \\ 0 & \frac{7}{4} & \frac{5}{4} & \frac{1}{2} & \frac{37}{4} \\ 0 & \frac{1}{4} & \frac{4}{4} & \frac{2}{2} & \frac{4}{4} \\ 0 & -\frac{1}{4} & \frac{29}{4} & \frac{9}{2} & \frac{157}{4} \end{array} \right] \quad (5.10)$$

接着处理第二个方程式，将矩阵 (5.10) 的②, ③, ④行中未知数 x_2 系数绝对值最大的调整到第②行，并将系数调整为 1，得

$$(A, b) = \left[\begin{array}{cccc|c} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & \frac{3}{4} \\ 0 & 1 & \frac{5}{7} & \frac{2}{7} & \frac{37}{7} \\ 0 & \frac{1}{4} & -\frac{1}{4} & \frac{18}{4} & \frac{71}{4} \\ 0 & \frac{1}{4} & \frac{4}{4} & \frac{4}{4} & \frac{4}{4} \\ 0 & -\frac{1}{4} & \frac{29}{4} & \frac{9}{2} & \frac{157}{4} \end{array} \right] \quad (5.11)$$

然后将矩阵 (5.11) 的①, ③, ④行分别减去②行的 $-3/4, 1/4, -1/4$ 倍，消去所在行的未知数 x_2 ，得

$$(A, b) = \left[\begin{array}{cccc|c} 1 & 0 & \frac{2}{7} & \frac{5}{7} & \frac{45}{14} \\ 0 & 1 & \frac{5}{7} & \frac{2}{7} & \frac{37}{7} \\ 0 & 0 & -\frac{3}{7} & \frac{31}{7} & \frac{115}{7} \\ 0 & 0 & \frac{52}{7} & \frac{32}{7} & \frac{284}{7} \end{array} \right] \quad (5.12)$$

继续此过程，最后矩阵可以变换为

$$(A, b) = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 4 \end{array} \right] \quad (5.13)$$

此时，可得 $x_1=1, x_2=2, x_3=3, x_4=4$ 。

在消元过程中，如果出现矩阵的某一行系数全为零，但是最右侧的常数项不为零，则此方程组无解，称此方程组为不相容方程组。如果未出现不相容方程，但是最右侧常数亦为零，则方程组可有无穷多组解。

以下给出高斯消元法的参考实现，需要注意以下几点：

(1) 由于未知元的系数在计算机中使用浮点数表示，为了减小误差，选择需要消去的未知元时，其系数的绝对值要尽可能的大。

(2) 如果给定的是整数方程，且方程数量较多，在进行消元时需要消去最大公约数。

(3) 当系数为整数且解也要求为整数时，需要使用类似于分数通分的技巧消去未知元，在具体实现时稍有差异，但总体思路是一致的。

```
//-----5.6.2.cpp-----//
// 列主元高斯消元法求解线性方程组 Ax=b。
bool gaussianElimination(vector<vector<double>> &A, vector<double> &b)
{
    // 把 b 存放在 A 的右边以便后续处理。
```

```

int n = A.size();
for (int i = 0; i < n; i++) A[i].push_back(b[i]);
// 按序处理方程。
for (int i = 0; i < n; i++) {
    // 为减少误差, 将正在处理的未知元系数的绝对值最大的方程式与第 i 个方程式交换。
    int pivot = i;
    for (int j = i; j < n; j++) {
        if (fabs(A[j][i]) > fabs(A[pivot][i])) pivot = j;
    }
    swap(A[i], A[pivot]);
    // 方程组无解或具有无穷多个解。
    if (fabs(A[i][i]) < EPSILON) return false;
    // 把正在处理的未知元的系数变为 1, 更新同方程其他未知元的系数。
    for (int j = i + 1; j <= n; j++) A[i][j] /= A[i][i];
    // 从第 j 个式子中消去第 i 个未知元。
    for (int j = 0; j < n; j++) {
        if (i != j)
            for (int k = i + 1; k <= n; k++)
                A[j][k] -= A[j][i] * A[i][k];
    }
    // 存放在矩阵 A 最右边的元素即为解。
    for (int i = 0; i < n; i++) b[i] = A[i][n];
    return true;
}
//-----5.6.2.cpp-----

```

在某些情况下, 可能会出现未知元的个数和方程个数不相等的情形, 需要根据系数是否为零适当更改主元的选择以使得消元能够继续进行。

```

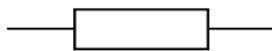
// E 为方程的个数, U 为未知元的个数。
for (int i = 0, j = 0; i < E && j < U; ) {
    // 判断待选未知元系数是否为 0, 若为 0, 需要选择同列系数不为 0 的未知元。
    // isZero() 返回系数 A[i][j] 是否为零。
    if (A[i][j].isZero()) {
        for (int m = i + 1; m < E; m++)
            if (!A[m][j].isZero()) {
                swap(A[i], A[m]);
                break;
            }
    }
    // 若该列的未知元系数均为 0, 则对当前方程式上的下一个未知元进行消元操作。
    if (A[i][j].isZero()) { j++; continue; }
    // 当前行当前列未知元的系数不为零, 消去其他行该列未知元。
    for (int n = U; n >= j; n--) A[i][n] = A[i][n] / A[i][j];
    for (int m = 0; m < E; m++) {
        if (i != m)
            for (int n = U; n >= j; n--)
                A[m][n] = A[m][n] - (A[m][j] * A[i][n]);
    }
    // 若当前列的未知元系数不为零, 则跳转到下一个方程的下一个未知元继续进行消元操作。
    i++, j++;
}

```

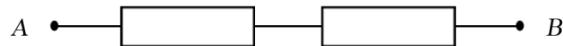
345 It's Ir-Resist-Able!^D (难以抗“阻”)

电阻是电路中的常见元件。每个电阻有两端, 当有电流通过时, 因为电阻具有“阻碍”电流流动的特性,

会导致一部分电流转化为热量。电阻“阻碍”电流流动能力的大小以一个正数值来衡量，称之为电阻的“阻值”，它的单位为欧姆（Ohms），以下是电路图中常见的电阻的图示：

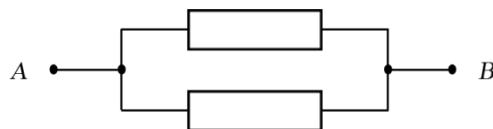


当两个电阻以串联的方式连接时：



它们的等效阻值为两个电阻的阻值之和。例如，将两个阻值分别为 100 欧姆和 200 欧姆的电阻串联，其等效阻值为 300 欧姆。如果将更多的电阻串联起来，它们的总阻值为各个电阻的阻值之和。

电阻也能以并联的方式相连接：



假如两个电阻的阻值分别为 100 欧姆和 150 欧姆，当它们以并联的方式连接时，在 A 点和 B 点之间的等效阻值为：

$$\frac{1}{\frac{1}{100} + \frac{1}{150}} = \frac{1}{\frac{3}{300} + \frac{2}{300}} = \frac{1}{\frac{5}{300}} = \frac{300}{5} = 60 \text{ 欧姆}$$

将三个电阻以并联方式相连，则三个阻值分别为 100 欧姆，150 欧姆，300 欧姆的电阻其等效阻值为：

$$\frac{1}{\frac{1}{100} + \frac{1}{150} + \frac{1}{300}} \text{ 欧姆}$$

在本问题中将给出多组关于电阻的阻值以及连接方式的数据。每个电阻的两端都可能会作为连接点，这些连接点使用唯一的正整数进行区分，称为标记，每个电阻通过两个连接点的标记以及一个实数值表示的阻值来指定。例如输入：

1 2 100

指定了在连接点 1 和 2 之间有一个阻值为 100 欧姆的电阻。两个串联的电阻可能会以下面的方式予以指定：

1 2 100
2 3 200

当知道了给定电阻的阻值及其连接方式，通过运用上述给出的求等效电阻的法则，可以确定该电阻网络中任意两个连接点之间的等效阻值。在某些特殊情形，使用上述的方法可能并不能够确定等效电阻的值，不过本题的测试数据中不包含这样的数据。

注意

(1) 在测试数据中，可能某些电阻并不对指定的两个连接点之间的等效电阻有影响，例如样例输入中的最后一组数据，连接点 1 和 2 之间的电阻并未使用。只有当电流通过电阻时，它才会影响总的等效电阻值。

(2) 测试数据不会出现电阻的两端连接在同一个连接点的情况，换句话说，电阻的两个端点的连接点标记不会相同。

输入

输入包含多组测试数组。每组测试数据的第一行包含三个整数 N , A 和 B 。 A 和 B 表示需要计算等效电

阻的两个连接点的标记。 N 表示总的电阻数量，不超过 30 个。 N, A, B 均为 0 表示测试数据结束，不需处理该行。在每组测试数据的第一行之后有 N 行数据，每行包含一个电阻的描述，以其两个端点的连接点标记和以实数值表示的阻值予以指定。

输出

对于每组测试数据，先输出测试数据组的序号（从 1 开始为数据组编号），然后输出其等效电阻，精确到小数点后两位。

样例输入

```
2 1 3
1 2 100
2 3 200
0 0 0
```

样例输出

```
Case 1: 300.00 Ohms
```

分析

题目所求的是电阻网络的等效阻值，需要应用物理学中关于电学的基尔霍夫（电路）定律（Kirchhoff's laws）进行解决^[42]，该定律是电路中电流和电压所遵循的基本规律ⁱ。根据基尔霍夫第一定律，假设进入某结点的电流为正值，离开这结点的电流为负值，则所有涉及该结点的电流值代数和等于零。以方程表达，对于电路的任意结点满足

$$\sum_{k=1}^n I_k = 0$$

如果知道了每个连接点的电势，根据欧姆定律，电流值等于电势差除以电阻值，则有

$$\sum_{i=1}^k \frac{V_i - V_j}{R_{ij}} = 0$$

可以得到关于电势的方程组，使用高斯消元法解该方程组得到各个连接点的电势值，再确定终点的流入电流（或者起点的流出电流），就可以根据起点和终点间的电势差及电流求出两个连接点之间的等效阻值。其中起点和终点的电势值可以预先设置为已知值。

题目中所给 N 最大不超过 30，则连接点的个数最大不超过 60，但连接点的编号范围未予指定，测试数据中连接点的编号可能会出现 10000 这样的编号值，如果使用二维数组存储两个连接点之间的等效阻值，很可能因为二维数组的空间不足导致运行时错误，需要将编号进行适当变换。

参考代码

```
const int MAXN = 100;
const double EPSILON = 1e-8;

// 不考虑其他连接点影响时，两个连接点间的阻值。
double resistor[MAXN][MAXN];

// 各个连接点的电势。
double voltage[MAXN];
```

ⁱ 1847 年，Kirchhoff 发表的报告中以公式形式总结了电网络理论中两条重要的定律：Kirchhoff 电流定律——电网络中每个结点上各支路电流代数和为零；Kirchhoff 电压定律——电网络中每一圈路内各支路电压代数和为零。

```

// 列主元高斯消元法求解线性方程组 Ax=b。
bool gaussianElimination(vector<vector<double>> &A, vector<double> &b)
{
    // 此处代码略, 请参考前述给出的高斯消元法的实现代码。
}

int main(int argc, char *argv[])
{
    int N, A, B, X, Y, cases = 0;
    double R;
    while (cin >> N >> A >> B, N > 0) {
        cout << "Case " << ++cases << ": ";
        // 记录电阻的序号, 按序重新编号, 以免初始给定的序号太大导致二维数组溢出。
        map<int, int> indexer;
        // 初始化电阻值。
        for (int i = 0; i < MAXN; i++)
            for (int j = 0; j < MAXN; j++)
                resistor[i][j] = 0.0;
        // 读入各个电阻的阻值。为了方便下一步计算, 先取阻值的倒数。
        for (int i = 0, label = 0; i < N; i++) {
            cin >> X >> Y >> R;
            if (indexer.find(X) == indexer.end()) indexer[X] = label++;
            if (indexer.find(Y) == indexer.end()) indexer[Y] = label++;
            resistor[indexer[X]][indexer[Y]] += 1.0 / R;
            resistor[indexer[Y]][indexer[X]] += 1.0 / R;
        }

        // C 为电阻的个数, 也是方程组的个数。
        int C = indexer.size();
        // 求出两个连接点之间直接相连的并联电阻的等效阻值。
        for (int i = 0; i < C; i++)
            for (int j = 0; j < C; j++)
                resistor[i][j] = 1.0 / resistor[i][j];

        // matrix 存储增广矩阵。
        vector<vector<double>> matrix(C, vector<double>(C, 0.0));
        vector<double> voltage(C, 0.0);
        // 结点电势, 取起点电势为 1000, 终点电势为 0。
        voltage[indexer[A]] = 1000.0;
        voltage[indexer[B]] = 0.0;
        // 由于起点和终点的电势为已知预设值, 构建方程使得方程组所对应的系数矩阵为方阵。
        matrix[indexer[A]][indexer[A]] = matrix[indexer[B]][indexer[B]] = 1.0;

        // 构建由电势未知元形成的方程组。
        for (int node = 0; node < C; node++) {
            // 起点和终点的电势已经预设, 不需再建立方程式。
            if (node == indexer[A] || node == indexer[B]) continue;
            // 对阻值不为零的结点建立方程, 阻值为零表示两个结点间无电路连接。
            for (int other = 0; other < C; other++)
                if (resistor[node][other] > EPSILON) {
                    // 单位电势下从 node 流到 other 的电流。
                    double inI = 1.0 / resistor[node][other];
                    // 基尔霍夫电流定律: 所有进入某结点的电流与所有离开该结点的
                    // 电流数值的代数和为零。
                }
        }
    }
}

```

```

        matrix[node][other] += inI;
        matrix[node][node] -= inI;
    }
}

// 使用列主元高斯消元法求解各结点的电势。
gaussianElimination(matrix, voltage);

// 计算终点流入的电流之和。
double current = 0.0;
for (int node = 0; node < C; node++)
    if (resistor[node][indexer[B]] > EPSILON)
        current += (voltage[node] - voltage[indexer[B]]) /
            resistor[node][indexer[B]];
// 根据欧姆定律计算等效阻值。
if (fabs(current) < EPSILON)
    cout << "0.00 Ohms\n";
else {
    cout << fixed << setprecision(2);
    cout << (voltage[indexer[A]] - voltage[indexer[B]]) / current;
    cout << " Ohms\n";
}
}
return 0;
}

```

强化练习: 1560 Extended Lights Out^E, 11319 Stupid Sequence^D, 12849 Mother's Jam Puzzle^E, 12850* Skating Puzzle^E。

扩展练习: 684* Integral Determinant^D, 10109* Solving Systems of Linear Equations^D, 11542 Square^D。

5.7 幂与对数

在 C++ 中, 有以下两个主要的幂运算函数:

```

#include <cmath>

double exp(double x);      // 返回自然对数 e 的 x 次幂
double pow(double base, double exponent);    // 返回底数 base 的 exponent 次幂

```

其中 `exp` 用于计算底数是自然对数的幂。自然对数 `e` 是一个特殊的常量, 可以使用泰勒级数 (Taylor series) 将其表示为

$$e = \frac{1}{0!} + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!} + \cdots = \sum_{n=0}^{\infty} \frac{1}{n!} = 2.718281828459045 \cdots$$

当底数大于 1 时, 幂增长非常快, 可以很容易超过 `unsigned long long int` 的表示范围。如果是整数的幂次, 在一定范围内, 可以使用 `double` 数据类型来比较其幂的大小。

701 The Archeologist's Dilemma^A (考古学家的烦恼)

一位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。他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, 他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。

编写程序，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

输入包含一个不超过 2147483648 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
```

分析

朴素的思路是不断计算 2 的幂，直到前面特定的数位恰为给定的数字，此种方法虽然可以得到结果，但是很容易超出题目时间和内存限制。

更好的方法是利用对数。假设丢掉的数字共有 k 位，现有的数字为 n ，要求的最小幂次为 x ，有以下不等式

$$n10^k < 2^x < (n+1)10^k \quad (5.14)$$

由于不等式左右均为正数，取 10 的对数得到下式

$$\frac{k + \log_{10} n}{\log_{10} 2} < x < \frac{k + \log_{10}(n+1)}{\log_{10} 2} \quad (5.15)$$

根据不等式 (5.15)，结合向下取整的库函数 `floor`，从 1 开始枚举 k 的值直到满足不等式，输出 x 即可。使用对数的方法虽然能获得通过，但是对于比较大的数，计算时间仍旧很长。

那么是否可能出现无解的情况呢？答案是不会。不等式 (5.14) 两边除以 10^k 可得

$$n < \frac{2^x}{10^k} < (n+1) \quad (5.16)$$

由不等式 (5.16) 可知，有 $n = \lfloor \frac{2^x}{10^k} \rfloor$ 。由于 $\log_{10} 2$ 为无理数，而 x 和 k 为有理数，令 $m = \log_{10} \left(\frac{2^x}{10^k} \right) = x \log_{10} 2 - k$ ，显然 m 为无理数，由于 x 和 k 可取任意正整数值，不论给定何值，通过调整 x 和 k 的值都能使得 m 与其无限接近，则 m 在实数域上是稠密的，进而在非负实数上是稠密的，则 $10^m = \frac{2^x}{10^k}$ 在非负实数上也是稠密的，对其向下取整，那么 $\lfloor \frac{2^x}{10^k} \rfloor$ 能够得到所有的自然数。简单来说，2 的幂其数字组成是无限的，肯定有整数 x 满足题意要求，所以不能输出“no power of 2”。

扩展练习：113 Power of Cryptography^A，10509 R U Kidding Mr. Feynman^B，11636 Hello World^A，11752 The Super Powers^C。

对数是幂的逆运算。关于对数有以下常用的运算规则 ($0 < a, 0 < b, 0 < c$)

$$a^{\log_a b} = b, \log_a b^c = c \log_a b, \log_a b = \frac{\log_c b}{\log_c a}$$

$$\log_c(a * b) = \log_c a + \log_c b, \log_c \frac{a}{b} = \log_c a - \log_c b$$

在 C++ 的实数函数库中，有以下与对数相关的函数：

```
#include <cmath>

double log(double x);      // 返回以 e 为底的对数值
double log10(double x);    // 返回以 10 为底的对数值
double log2(double x);    // 返回以 2 为底的对数值
```

在具体应用时，需要注意参数必须是一个正的实数，小于等于 0 的值取对数会发生溢出，返回的是一个无穷大的值（或者 NaN，即 Not a Number 的缩写，非数，表示非未定义或不可表示的值）。如果对若干整数进行除法运算然后再取对数，需要注意先将其转换为浮点数再进行除法运算，否则进行的将是整除运算，会导致结果发生错误。

10883 Supermean^D（超级平均数）

你知道如何计算 n 个数的平均值吗？当然，仅仅知道这些这对我来说仍然不够。我需要的是超级平均数。“什么是超级平均数？”，你一定会问。我这就告诉你：将 n 个数按升序排列，先计算这个序列相邻两个数的平均数，这样会得到 $n-1$ 个数，它们仍然是按升序排列的，重复此过程，直到你最后得到一个数，它就是超级平均数。我试着编写程序来完成这个任务，但是它太慢了，:-)，你能帮帮我吗？

输入

输入的第一行包含一个整数，表示测试数据的组数 N 。接着有 N 组数据，每组数据的第一行是一个整数 n ($0 < n \leq 50000$)，表示接下来的 n 行每行包含一个实数，大小在 -1000 和 1000 之间，按升序排列。

输出

对于每组测试数据，先输出测试数据组的编号 ‘Case #x:’，接着输出超级平均数的值，结果四舍五入到小数点后 3 位。

样例输入

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

样例输出

```
Case #1: 3.000
```

分析

初看似乎没有什么规律，可以先从 n 较小的情况进行观察。设序列为 d_i , $1 \leq i \leq n$, 当 $n=1, 2, 3, 4, 5, 6$ 时，通过手工计算可以得知超级平均数 M_n 分别为

$$\begin{aligned} M_1 &= \frac{d_1}{2^0} \\ M_2 &= \frac{d_1 + d_2}{2^1} \\ M_3 &= \frac{d_1 + 2d_2 + d_3}{2^2} \\ M_4 &= \frac{d_1 + 3d_2 + 3d_3 + d_4}{2^3} \\ M_5 &= \frac{d_1 + 4d_2 + 6d_3 + 4d_4 + d_5}{2^4} \\ M_6 &= \frac{d_1 + 5d_2 + 10d_3 + 10d_4 + 5d_5 + d_6}{2^5} \end{aligned}$$

观察可知系数构成杨辉三角——第 i 项的系数为 $C(n-1, i-1)$, 即

$$M_n = \frac{C_{n-1}^0 d_1 + C_{n-1}^1 d_2 + \cdots + C_{n-1}^{n-2} d_{n-1} + C_{n-1}^{n-1} d_n}{2^{n-1}} = \frac{\sum_{i=1}^{n-1} C_{n-1}^{i-1} d_i}{2^{n-1}}$$

但是 n 最大可为 50000, 直接计算 $C(n-1, i-1)$ 或者 2^{n-1} 会导致溢出。由于给定的数都是在 -1000 至 1000 之间, 其平均数也必定在此范围, 所以可以借助对数进行计算。在具体计算组合数时, 可利用下述递推公式

$$C_{n-1}^i = C_{n-1}^{i-1} \cdot \frac{n-i}{i}, \quad 1 \leq i \leq n$$

以简化运算。注意当序列元素值为 0 或为负数时的求和处理。

参考代码

```
int main(int argc, char *argv[])
{
    int cases, n;
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        double t = 0.0, di, mean = 0.0;
        cin >> n;
        for (int i = 0; i < n; i++) {
            cin >> di;
            if (i) t += log2((double)(n - i) / i);
            if (fabs(di) > 0) {
                double sign = (di > 0 ? 1.0 : -1.0);
                mean += sign * pow(2, t + log2(fabs(di)) - (n - 1));
            }
        }
        cout << "Case #" << c << ":" ;
        cout << fixed << setprecision(3) << mean << '\n';
    }
    return 0;
}
```

强化练习: [11241 Humidex^D](#), [11384 Help is Needed for Dexter^B](#), [11556 Best Compression Ever^C](#), [11666 Logarithms^D](#), [11986 Save from Radiation^D](#), [12416 Excessive Space Remover^C](#), [12808 Banning Balcony^D](#)。

扩展练习: [1639 Candy^E](#), [11029 Leading and Trailing^C](#)。

5.8 实数函数库

C++中包含多个处理实数的函数, 它们都包括在头文件<cmath>中。

```
#include <cmath>

double sqrt(double x); // 返回指定数值的平方根, 要求参数为非负实数
double cbrt(double x)C++11; // 返回指定数值的立方根
double hypot(double x, double y)C++11; // 根据给定的直边长度, 返回三角形的斜边长度
double ceil(double x); // 向上取整, 返回不小于 x 的最小整数
double floor(double x); // 向下取整, 返回不大于 x 的最大整数
double fmod(double numerator, double denominator); // 返回浮点数的余数
```

以上函数多用于解一元二次方程或概率相关的题目中, 因为概率论中经常会出现比较大的整数, 使用内置的整数类型无法表示, 需要使用 `double` 或 `long double` 数据类型。

846 Steps^A (数轴行走)

假设有一条直线，上面标记了一些整数位置，你需要通过单步行走从一个整数位置到达另一个整数位置，每一步的步长必须是非负整数，且相对于前一步的长度来说，只能大 1，或者相等，或者小 1。如果要求第一步和最后一步的步长必须为 1，则从整数位置 x 走到另外一个整数位置 y 最少需要多少步？

输入与输出

输入第一行包含一个整数 n ，表示测试数据的组数。每组测试数据包含两个整数 x 和 y ， $0 \leq x \leq y < 2^{31}$ 。对于每组测试数据，输出从 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
```

分析

可以证明：在本题条件限制下，如果所走步数构成的数列具有左右对称的性质，则总步数是最少的。如果采取左右对称的走法，设两点距离为整数 d ，整数 $n = \sqrt{d}$ ，则最大步数为 n 步时能达到的距离是 $1+2+3+\cdots+(n-1)+n+(n-1)+\cdots+3+2+1=n^2$ 。比较两点距离 d 与 n^2 ，若相等，表明只需走 $2(n-1)+1=2n-1$ 步；否则，若剩余距离 $d-n^2$ 在 1 到 $n+1$ 之间，只需插入一步即可；若 $d-n^2$ 大于 $n+1$ ，则需多插入两步。

强化练习：138 Street Number^A, [386 Perfect Cubes^A](#), 474 Heads/Tails Probability^A, 545 Heads^C, 617 Nonstop Travel^C, 860 Entropy Text Analyzer^C, 880 Cantor Fractions^B, 10693 Traffic Volume^B, [10784 Diagonal^A](#), [10916 Factstone Benchmark^B](#), 11335 Discrete Pursuit^D, 11342 Three-Square^B, [11565 Simple Equations^A](#), [11614 Etruscan Warriors Never Play Chess^A](#), 11715* Car^A, [11970 Lucky Numbers^B](#), [13142 Destroy the Moon to Save the Earth^E](#), [13217 Amazing Function^E](#)。

扩展练习：11692* Rain Fall^D, 11714 Blind Sorting^C, [11847 Cut the Silver Bar^C](#), 12027 Very Big Perfect Squares^D, [12842 Courier Problem^E^{\[43\]}](#)。

5.9 小结

本章主要包括四个重点内容：(1) 高精度整数四则运算的实现。不过由于编程竞赛大多数允许使用 Java 或者 Python，高精度整数运算的代码实现并不是一个障碍，重点在于从题目中抽象出递推关系，得到结果的表达式，从而进一步使用高精度整数进行运算。(2) 浮点数运算误差的处理。由于计算机表示的原因，无法精确表示所有的小数，因此在计算中不可避免的存在误差，如果在计算中误差控制不当，很可能导致最后的结果与参考输出有差异，从而使得无法通过评测。(3) 二项式定理和多项式定理。二项式定理和多项式定理在多项式的展开以及排列组合中指数的计算中非常有用。(4) 实数函数库的使用。对于某些特定题目来说，熟练掌握和使用实数函数库能够达到事半功倍的效果。

第 6 章 组合数学

是故，《易》有太极，是生两仪，两仪生四象，四象生八卦。
——《易传·系辞上传》

在编程竞赛中，有关组合数学、概率论的题目经常出现。本章介绍了在 UVa OJ 题库中出现的此类题目所涉及的常见知识点，如果读者需要更为深入地了解组合数学、概率论的特定内容，建议阅读相应的入门类书籍^[44]。由于组合数学中牵涉到的整数一般都比较大，常常需要高精度整数的支持，所以熟悉 Java 中 BigInteger 类的使用非常有必要。

6.1 计数原理

6.1.1 加法原理

加法原理 (addition principle of counting): 假设完成一项任务有 n 类不同的方法，只要采用一类方法中的任何一种即可完成这项任务，第一类方法包括 m_1 种方法，第二类方法包括 m_2 种方法，…，第 n 类方法包括 m_n 种方法，而且所有方法中任意两种方法均是不同的，则完成此项任务共有 $m_1+m_2+\dots+m_n$ 种方法。

在解题中应用加法原理，其关键是要找到一个“切入点”，以便将计数分解为若干个独立（不相容）的部分，这样才能保证既不重复也不遗漏地进行计数。

11401 Triangle Counting^A (三角形计数)

给定长度为 1, 2, …, n 的 n 条木棒，从中任意挑选 3 条木棒拼接成三角形，确定能够拼接得到的不同三角形个数。此处“不同三角形”的含义是两个三角形至少有一条边的长度不相同。

输入

输入包含多组测试数据，每组测试数据由一个正整数 n ($3 \leq n \leq 1000000$) 构成，输入最后以一个小于 3 的整数作为结束标记，此组测试数据不需处理。

输出

对于每组测试数据，输出不同三角形的数量。

样例输入

```
5
0
```

样例输出

```
3
```

分析

由于 n 的上限较大，使用朴素的穷尽法效率较低，会超出时间限制。为了能够不重复地统计三角形的数量，先将三角形按最长边进行分类计数。令 $C(x)$ 表示三角形中最长边为 x 时的不同三角形数量，同时另外的两条边的边长分别为 y 和 z ，根据三角形的边不等式有

$$y + z > x$$

易得

$$x - y < z < x$$

由于题目约束三角形的三条边长均不相同，则当 $y=1$ ，不等式无解；当 $y=2$ 时， z 只有 1 个解： $x-1$ ；当

$y=3$ 时 z 有 2 个解: $x-2, x-1, \dots$; 当 $y=x-1$, z 有 $x-2$ 个解: $2, 3, \dots, x-1$ 。观察易得, 解的数量构成了一个等差数列, 故总的解数量为

$$\frac{(x-1)(x-2)}{2}$$

但是这并不是 $C(x)$ 的值。按照题意要求, $y=z$ 的解是不符合要求的, 故需要将其剔除。当 $y=z$ 时, 有 $x/2+1 \leq y=z \leq x-1$, 不符合题意的解其数量为

$$(x-1) - \left(\frac{x}{2} + 1\right) + 1 = \frac{x-2}{2}$$

由于 $y=2, z=x-1$ 和 $z=2, y=x-1$ 是相同的三角形, 符合题意的解实际上将相同的三角形统计了两次, 故需要将结果减半, 即

$$C(x) = \frac{(x-1)(x-2)}{2} - \frac{x-2}{2} = \frac{(x-2)^2}{4}$$

最后, 由于题目所求为最大边长不超过 n 的不同三角形数量 $F(n)$, 有

$$F(n) = C(1) + C(2) + \dots + C(n) = F(n-1) + C(n)$$

强化练习: 11480 Jimmy's Balls^C, 11554 Hapless Hedonism^C。

扩展练习: [11375 Matches](#)^D。

6.1.2 乘法原理

乘法原理 (multiplication principle of counting): 假设有 k 项任务 T_1, T_2, \dots, T_k 需要相继完成, 如果完成任务 T_1 有 n_1 种不同的方法, 完成任务 T_2 有 n_2 种不同的方法, \dots , 完成任务 T_k 有 n_k 种不同的方法, 在完成任务的过程中, 每次从各个任务的方法中挑选出一种方法相继执行可以完成 k 项任务, 那么相继完成 k 项任务共有 $n_1 \times n_2 \times \dots \times n_k$ 种方法。

应用乘法原理的关键也是先将计数分成若干个独立的步骤, 然后再计数每个步骤能够采用的不同方法数, 最后总的方法数就是各个步骤方法数的乘积。

182 Bonus Bonds^D (债券彩票)

积贫症者 (Impecunia) 联合政府从不征税, 而是 (有时候是强制性的) 通过出售债券彩票来募集资金。最初, 债券编号共 8 位数字, 包括 1 位数字前缀和 7 位数字后缀, 数字前缀为 1 至 9 的某个数字, 表示出售该债券的地区编号, 简称区号。由于债券彩票模式非常成功, 编号方案也相应有所改变, 增加了 2 位数字, 使得债券编号变为 10 位数。为了与原有的编号方案兼容, 在 10 位数的编号方案中, 从左侧数第 3 位 (即从右侧数第 8 位数字) 仍然表示区号。与此同时, 政府新成立了一个“中心区”, 区号为 0。为了安全考虑, 债券的编号不能全为 0。虽然原有的债券编号都是从 0 开始, 但是由于区号非 0, 故而符合要求。中心区由于其区号为 0, 为了符合要求, 其出售的债券需要从 0000000001 开始编号。

每个月, 中奖号码会从每个区分别抽取产生。抽奖设备会生成一系列数字, 然后从这些数字中按照 10 个一组的方式组合成一个中奖号码, 但是这样存在潜在的问题——生成的中奖号码所对应的债券可能尚未售出。由于抽奖设备有些老旧, 很容易出问题, 组织者不但希望生成的中奖号码是已经出售的债券编号, 而且要求生成的中奖号码的每一位数字与已出售的债券在相应位置的数字分布一致。假如我们希望从给定区域抽取 N 个中奖号码, 则设置参数使得设备生成 10 组 N 位的数字, 中奖号码的每一位从对应组的数字中抽取。第一个中奖号码由 10 组数字的第一位数字组成, 第二个中奖号码由 10 组数字的第二位数字组成, 依此类推。对于每一组数字, 组织者会调整设备参数使得生成的数字分布和该地区已经出售的债券编号对应位置的数字

分布相匹配。审计官会生成一张数字分布表以便对参数设置情况进行核查。

编写程序，针对任意一次抽奖为审计官生成一张数字分布表。对于每一个区域，程序读入一个债券编号以便计算数字分布，该编号表示将在该区域出售的**下一张**债券的号码，由于将全部信息输出显得太过冗长，因此你的程序只需要输出指定位置的数字分布即可。

输入

输入包含多行，每行包含一个编号，表示在特定区域出售的**下一张**债券的号码。后面是一个在 1 到 10 之间的整数，表示需要计算数字分布的编号位置。注意，某些区域在输入中可能出现不止一次，而其他某些区域可能一次也不出现。输入最后以包含“**0000000000 0**”的一行作为结束标志。

输出

输出由多张数字分布表组成，每张表对应输入中的一行。每张分布表包含 10 行，每行包含一个数值，表示 0 到 9 的数字在该指定位置出现的次数。次数在输出时以宽度 11 右对齐输出。相邻两张分布表之间以一个空行隔开。

样例输入

```
4810000000 1
0000000000 0
```

样例输出

```
100000000
100000000
100000000
100000000
80000000
100000000
100000000
100000000
100000000
100000000
```

分析

如果使用朴素的穷尽搜索，由于输入范围较大，会发生超时。可以根据加法原理和乘法原理进行计数，从而提高效率。根据题意获取区号，然后将区号从编号中去除，获取已出售的债券张数以及需要输出次数的指定位置，按下述步骤进行处理：

(1) 特例处理。如果已出售的债券张数为 0，或者出售的债券张数只有 1 张且区号为 0，则指定的位置上各个数字的出现次数均为 0。

(2) 其他情形。由于可取的数字个数会因其所在位置发生变化，需要分别处理。设区号为 r ，已出售的最后一张债券编号为 $d_1d_2d_3d_4d_5d_6d_7d_8d_9$ （由于已经去除了区号，故债券编号只有 9 位数字）， t 为某个数字在该位置上出现的次数，可以分以下几种情况处理：

(2a) 如果指定位置是 3，即区号位，对于数字 d 来说，如果 $d=r$ ，出现次数 $t=d_1d_2d_3d_4d_5d_6d_7d_8d_9$ ；如果 $d \neq r$ ，出现次数 $t=0$ 。

(2b) 如果指定的位置是 1，即首位，对于数字 d 来说，如果 $d < d_1$ ，出现次数 $t=100000000$ ；如果 $d = d_1$ ，那么出现次数 $t=d_2d_3d_4d_5d_6d_7d_8d_9 + 1$ ；如果 $d > d_1$ ， $t=0$ 。

(2c) 如果指定的位置是 10，即末位，对于数字 d 来说，如果 $d \leq d_9$ ，出现次数 $t=d_1d_2d_3d_4d_5d_6d_7d_8 + 1$ ；如果 $d > d_9$ ，出现次数 $t=d_1d_2d_3d_4d_5d_6d_7d_8$ 。

(2d) 如果指定的位置是其他位置，则数字 d 出现的次数和指定位置之前、位置本身和位置之后的数

值有关。例如, 如果指定的位置为 6, 即需要确定 d_5 上各个数字的出现次数, 那么有: 若 $d < d_5$, 则 $t = d_1d_2d_3d_40000 + 10000$; 若 $d = d_5$, 则 $t = d_1d_2d_3d_40000 + d_6d_7d_8d_9 + 1$; 若 $d > d_5$, 则 $t = d_1d_2d_3d_40000$ 。

需要注意的是, 对于数字 0 来说, 如果区号也为 0 (即为中心区), 则由于该区是从 000000001 开始编号, 那么数字 0 出现的次数需要在原有计数的基础上减去 1 才是正确的计数结果。

强化练习: [11038* How Many 0's^B](#), [11204 Musical Instruments^C](#), [11538 Chess Queen^B](#), [11962 DNA II^D](#), [12463 Little Nephew^C](#)。

扩展练习: [11115 Uncle Jack^B](#), [11310 Delivery Debacle^B](#)。

6.2 排列与组合

从 n ($n \geq 1$) 个不同元素中任意取出 m ($n \geq m \geq 1$) 个元素, 按照取的先后顺序排列起来形成一列, 称为从 n 个不同元素中取出 m 个元素的一个排列 (permutation)。当 $m = n$ 时, 将能够得到的所有不同排列的全体称为全排列 (full permutation)。由于有 n 个不同的元素, 在构成全排列时, 第一个元素有 n 种选择, 第二个元素有 $(n-1)$ 种选择, …, 最后一个元素有 1 种选择, 根据乘法原理, 全排列共有 $n \times (n-1) \times \cdots \times 2 \times 1$ 种排列方法, 记为 n 的阶乘, 通常表示为

$$P(n, n) = n! = \prod_{i=1}^n i, \quad 1 \leq n$$

如果定义 $0! = 1$, 进一步推广, 则有

$$P(n, k) = \frac{n!}{(n-k)!}, \quad 0 \leq n, \quad 0 \leq k \leq n$$

由于阶乘增长很快, 可以很容易达到 32 位整数的表示上限, 即使是使用 64 位整数, 也无法存储 $25!$ 所对应的整数值。当 n 很大时, 可以根据斯特林公式 (Stirling's approximation)^I

$$n! \approx \sqrt{2\pi n} \left(\frac{n}{e}\right)^n, \quad \pi \text{ 为圆周率, } e \text{ 为自然对数的底}$$

计算 $n!$ 的近似值。结合实数函数库中的 $\log10$ 函数, 对估计 $n!$ 的位数也很有帮助。

强化练习: [1185* Big Number^C](#), [12712 Pattern Locker^D](#), [12869* Zeroes^D](#), [12934 Factorial Division^D](#)。

扩展练习: [10323* Factorial You Must be Kidding^A](#)。

组合 (combination) 是指从 n ($n \geq 1$) 个不同元素中任意取 k ($n \geq k \geq 0$) 个元素的不同取法, 一般将其记为 $C(n, k)$ 或 C_n^k 或 $\binom{n}{k}$, 有

$$C(n, k) = C_n^k = \binom{n}{k} = \binom{n}{n-k} = \frac{P(n, k)}{P(k, k)} = \frac{n!}{(n-k)! k!}, \quad 1 \leq n, \quad 0 \leq k \leq n$$

如果 n 个元素中有部分元素重复, 将 n 个元素进行全排列能够得到的不同排列方式总数为

$$\binom{n}{m_1, m_2, \dots, m_k} = \frac{n!}{m_1! m_2! \cdots m_k!}, \quad m_1 + m_2 + \cdots + m_k = n, \quad m_i \text{ 表示第 } i \text{ 个不同元素的重复次数}$$

例如将字符串 “aabdeef” 进行全排列, 能够得到的不同字符串排列种数为

$$P = \frac{7!}{2! 1! 1! 2! 1!} = 1260$$

^I 詹姆斯·斯特林 (James Stirling, 1692—1770), 苏格兰数学家, 斯特林数和斯特林公式以他的名字命名。

在某些情况下, 为了便于计算组合数, 还可以使用以下递推关系式^I

$$C(n, k) = C(n - 1, k) + C(n - 1, k - 1)$$

其中 $0 \leq n, 1 \leq k, C(n, 0) = C(n, n) = 1$ 。

```
long long Cnk[41][41];
for (int i = 0; i <= 40; i++) {
    Cnk[i][0] = Cnk[i][i] = 0;
    for (int j = 1; j < i; j++)
        Cnk[i][j] = Cnk[i - 1][j] + Cnk[i - 1][k - 1];
}
```

因为阶乘增长得很快, 在某些情况下, 虽然给定组合数的最终结果在 64 位整数的表示范围内, 但直接按照定义计算, 中间结果会超出 64 位整数的表示范围, 此时需要采用一些特殊的技巧。其中的一种技巧是将阶乘的各个数进行素因子分解, 消去分子和分母共同的素因子使得将要计算的中间数值变小, 从而达到在有限的整数表示范围计算出组合数的值的目标。另外一种技巧是不进行素因子分解, 而是求计算式中分子的各个乘数和分母的各个乘数间的最大公约数, 使用最大公约数来除分子和分母, 导致中间结果数值变小, 从而使得而最终的计算结果在 64 位整数的表示范围内。

强化练习: [10338 Mischievous Children^A](#), [11076 Add Again^C](#), [12001 UVa Panel Discussion^D](#)。

扩展练习: [10219* Find the Ways^A](#), [10375 Choose and Divide^B](#)。

6.2.1 康托展开和康托逆展开

给定一个长度为 n ($n \geq 1$) 的字符串, 其字符两两互不相同, 如果将字符串的全排列按字典序排列, 然后从 1 开始为每个排列赋予一个序号, 则可以将字符串的全排列映射到 1 至 $n!$ 的正整数。例如, 给定字符串 “abc”, 它的全排列按照字典序排列为: {abc, acb, bac, bca, cab, cba}, 如果为全排列中的每个字符串从 1 开始进行编号, 则全排列中各个字符串的序号与正整数 1 至 $3!$ 构成一一对应的关系。反之, 给定 1 至 $3!$ 的任意一个整数, 它唯一对应一个排列。从数学上看, 这形成了一个双射。

当给定字符串和序号, 如何确定该字符串的全排列中指定序号所对应的排列呢? 例如, 给定字符串 “abcdefg”, 要求确定此字符串的全排列中序号为 83 的排列。朴素的方法是列出所有排列, 然后逐个计数以确定排列。很明显, 此种方法效率较低, 当字符串较长时不可用, 需要使用其他更为高效的方法。

可以证明, 从 0 到 $n! - 1$ 之间的任何整数 m , 可以唯一地将其表示成以下形式^[45]

$$m = a_{n-1} \times (n-1)! + a_{n-2} \times (n-2)! + \cdots + a_2 \times 2! + a_1 \times 1!$$

其中, $a_i \in \mathbb{Z}_0^+, 0 \leq a_i \leq i, i = 1, 2, \dots, n-1$ 。例如

$$23 = 3 \times 3! + 2 \times 2! + 1 \times 1!, n = 4$$

此即康托展开 (Cantor expansion)^{II}。

如果给定字符串在全排列中的序号, 如何根据序号确定相应的排列呢? 例如, 给定字符序列 $\langle A, B, C,$

^I 从 n 个物品中取 k 个物品的方法, 根据是否包含物品 x , 可以分为两类: 一类是不包含物品 x , 一类是包含物品 x 。如果不包含物品 x , 相当于从除物品 x 之外的 $n-1$ 个物品中取 k 个物品, 这一类方法共有 $C(n-1, k)$ 种。如果是包含物品 x , 这类方法还需要从其他 $n-1$ 个物品中取 $k-1$ 个物品才凑成 k 个物品, 这类方法共有 $C(n-1, k-1)$ 种。由加法原理可得: $C(n, k) = C(n-1, k) + C(n-1, k-1)$ 。

^{II} 格奥尔格·费迪南德·路德维希·菲利普·康托 (Georg Ferdinand Ludwig Philipp Cantor, 1845—1918), 德国数学家, 集合论的创始人, 建立和发展了超穷数理论。

$D, E>$, 从 1 开始编号, 某个排列在此字符串的全排列中的序号为 83, 那么这个排列是什么呢? 可以根据康托展开并结合辗转相除法得到, 其操作步骤为¹:

(1) 字符串长度 $n=5$, 序号是从 0 开始计数, 因此需要确定

$$82 = a_4 \times 4! + a_3 \times 3! + a_2 \times 2! + a_1 \times 1!$$

中 a_4, a_3, a_2, a_1 的值, 其中 a_i 对应于“在尚未出现的元素中当前元素的排位 (从 0 开始计数)”;

(2) 由于 $82 \div 4! = 24$ 商数为 3, 余数为 10, 故 $a_4=3$ 。此时所有元素均未出现, 按字典序构成序列 $\langle A, B, C, D, E \rangle$, 从 0 开始计数, 其中第 3 大的元素为 D , 故排列的第一个字符为 D ;

(3) $10 \div 3! = 6$ 商数为 1, 余数为 4, 故 $a_3=1$ 。尚未出现的元素按字典序构成序列 $\langle A, B, C \rangle$, 从 0 开始计数, 第 1 大的元素为 B , 故排列的第二个字符为 B ;

(4) 依前述过程推导, 可知 $a_2=2, a_1=0$ 。由此可得第三个字符为 E , 第四个字符为 A ;

(5) 最后剩余字符 C , 因此第五个字符为 C 。

则对应序号 83 的排列为 $\langle D, B, E, A, C \rangle$ 。可以使用 STL 中的 `next_permutation` 函数对结果予以验证。

```
-----6.2.1.1.cpp-----
int main(int argc, char *argv[]) {
    string s = "ABCDE";
    int indexer = 1;
    do {
        if (indexer == 83) {
            // 输出为: DBEAC。
            cout << s << '\n';
            break;
        }
        indexer++;
    } while (next_permutation(s.begin(), s.end()));
    return 0;
}
-----6.2.1.1.cpp-----
```

注意在使用康托展开时, 要求给定序列中的元素互不相同, 如果字符序列中有相同的字符, 使用上述方法会得到错误的结果, 此时需要通过组合方法来进行计算。

以下是康托展开的参考实现。

```
-----6.2.1.2.cpp-----
long long factorial[20] = {1};

// 康托展开。
void cantorExpansion(string s, long long n) {
    n %= factorial[s.length()];
```

¹ 也可以采用相反的顺序进行计算。由于

$$m = a_{n-1}(n-1)! + a_{n-2}(n-2)! + \cdots + a_2 \cdot 2! + a_1 \cdot 1!$$

将等式两边同时除以 2, 可得

$$\frac{m}{2} = \frac{a_{n-1}(n-1)!}{2} + \frac{a_{n-2}(n-2)!}{2} + \cdots + \frac{a_3 \cdot 3!}{2} + \frac{a_2 \cdot 2!}{2} + a_1$$

则 m 除以 2 的余数就是 a_1 , 商即为 m_1 , 然后 m_1 除以 3 的余数就是 a_2 , ..., 依此类推。

```

        for (int i = s.length() - 1; i >= 0; i--) {
            long long idx = n / factorial[i];
            cout << s[idx];
            s.erase(s.begin() + idx);
            n %= factorial[i];
        }
        cout << '\n';
    }

int main(int argc, char *argv[]) {
    string s;
    long long n;
    for (int i = 1; i < 20; i++) factorial[i] = factorial[i - 1] * i;
    while (cin >> s >> n) {
        sort(s.begin(), s.end());
        cantorExpansion(s, n);
    }
    return 0;
}
//-----6.2.1.2.cpp-----//

```

根据康托逆展开 (inverse Cantor expansion)，可以将某个不包含重复字符的字符串映射为一个整数，此整数即对应于该字符串在组成此字符串的字符全排列中的序号。例如，给定字符序列 $\langle A, B, C, D, E \rangle$ ，需要确定目标序列 $\langle C, E, A, D, B \rangle$ 在字符序列 $\langle A, B, C, D, E \rangle$ 全排列中的序号，可按以下步骤确定：

(1) 由于字符序列长度为 5，需要确定表达式

$$m = a_4 \times 4! + a_3 \times 3! + a_2 \times 2! + a_1 \times 1!$$

中 a_4, a_3, a_2, a_1 的值，进而确定 m 值，其中 a_i 对应于“在尚未出现的元素中当前元素的排位（从 0 开始计数）”；

(2) 第一个字符 C ，由于此时所有元素均未出现，目标序列为 $\langle C, E, A, D, B \rangle$ ，按照字典序并从 0 开始计数， C 是该序列中第 2 大的元素，故 $a_4=2$ ；

(3) 第二个字符 E ，由于字符 C 已经出现，因此尚未出现的元素构成序列 $\langle E, A, D, B \rangle$ 中，按照字典序并从 0 开始计数， E 是第 3 大的元素，故 $a_3=3$ ；

(4) 第三个字符 A ，同理可知，在序列 $\langle A, D, B \rangle$ 中， A 是第 0 大的元素，故 $a_2=0$ ；

(5) 第四个字符 D ，在序列 $\langle D, B \rangle$ 中， D 的是第 1 大的元素，故 $a_1=1$ 。

故有

$$m = 2 \times 4! + 3 \times 3! + 0 \times 2! + 1 \times 1! = 67$$

由于是从 1 开始为全排列进行编号，而使用康托逆展开得到的 m 是从 0 开始计数的，因此序列 $\langle C, E, A, D, B \rangle$ 所对应的序号为 68。可以使用 STL 中的 `next_permutation` 函数对结果予以验证。

```

//-----6.2.1.3.cpp-----//
int main(int argc, char *argv[]) {
    string s = "ABCDE", t = "CEADB";
    int indexer = 1;
    do {
        if (s == t) {
            // 输出为：68。
            cout << indexer << '\n';
            break;
        }
        indexer++;
    }
}

```

```

    } while (next_permutation(s.begin(), s.end()));
    return 0;
}
//-----6.2.1.3.cpp-----//

```

强化练习：[11525 Permutation^C](#)，[12335 Lexicographic Order^D](#)。

153 Permalex^A（递变序）

给定一个字符串，我们能够通过重新排列字符的方式来得到一个新的字符串。如果我们在重排时加上顺序（比如说按照字典序），那么这些字符串在所有排列中的位置序号，能够使用一个唯一的数字来表示。例如，字符串‘ acab ’一共产生了以下 12 种不同的排列：

aabc	1	acab	5	bcaa	9
aacb	2	acba	6	caab	10
abac	3	baac	7	caba	11
abca	4	baca	8	cbaa	12

因此字符串‘ acab ’在此全排列中的序号为 5。

编写程序，读入一个字符串，确定该字符串在其全排列生成的字符串中的位置序号。注意字符串的全排列数量可以很大，然而我们保证给定的字符串中其位置序号不会大于 $2^{31}-1=2147483647$ 。

输入与输出

输入包含多行，每行包含一个字符串。每个字符串不超过 30 个小写字母，可能有相同的字母。输入以只包含字符‘#’的一行结束。

输出包含多行，对应输入的每一行。每行由表示给定字符串在全排列中的位置序号构成，该序号以右对齐宽度 10 进行输出。

样例输入

```
bacaa
#
```

样例输出

```
15
```

分析

采用朴素的方式生成所有排列（例如，使用库函数 `next_permutation` 来生成所有排列）并计数序号的做法会导致超时。由于给定字符串可能包含重复的字符，因此不能直接使用康托逆展开，但是可以从康托逆展开得到启发。给定一个包含 n 个不同字符的字符串 $s = "c_1c_2\dots c_n"$ ， $x_i < x_j$ ， $0 \leq i < j < n$ 。令 s_i 和 s_j 是 s 的全排列中两个不同的排列， s_i 在全排列中的序号为 i ， s_j 在全排列中的序号为 j ，且有 $i < j$ ，定义 s_i 到 s_j 的“变换距离”为 i 和 j 之差的绝对值。根据上述定义， $s_0 = "c_1c_2\dots c_n"$ ，如果给定 $s_k = "c_kc_1c_2\dots c_{k-1}c_{k+1}\dots c_n"$ ，即 s_k 是将 s 的第 k 个字符 c_k 作为首字符，其他字符仍按照原有顺序排列而得到的字符串，那么 x 的值应该如何确定呢？从 s_0 开始，按照全排列的顺序，对 s_0 进行 $(n-1)!-1$ 次变换操作，可以得到 $s' = "c_1c_n c_{n-1}\dots c_2"$ ，再进行一次变换，可以得到 $s'' = "c_2c_1c_3\dots c_n"$ ，可知 s_0 和 s'' 的变换距离为 $(n-1)!$ 。类似的，从 $s' = "c_2c_1c_3\dots c_n"$ 开始，按照全排列的顺序，对 s' 进行 $(n-1)!-1$ 次变换操作，可以得到 $s''' = "c_2c_n c_{n-1}\dots c_3c_1"$ ，再进行一次变换操作，可以得到 $s'''' = "c_3c_1c_2c_4\dots c_n"$ ，可知 s_0 和 s'''' 的变换距离为 $2 \times (n-1)!$ 。依此类推，可知 s_0 和 s_k 的变换距离为 $(k-1) \times (n-1)!$ 。由于给定的是 n 个不同字符，在进行变换时，可以将其中的一部分字符作为一个整体进行全排列，因此可以提高计数的效率。

在本题的约束条件中，给定的是可能包含重复字符的字符串，不能直接进行计算，但是仍然可以使用“将

其中的一部分字符作为一个整体进行全排列”的思路来提高计数效率，下面以样例输入为例进行具体说明。给定的字符串是“bacaa”，需要确定此字符串在全排列中的序号。该字符串的初始排列为“aaabc”，为了观察规律，不妨先考察“aaabc”是如何变换到“acbaa”的。不难看出，“aaabc”和“acbaa”的首字符相同，均为第0大的字符‘a’占据首位，变换的过程相当于将“aaabc”的最后4个字符“aabc”进行了一次全排列，接着将“acbaa”再变换一步就成为“baaac”，第1大的字符‘b’占据首位，之后的4个字符按字典序排列，因此“aaabc”和“baaac”的“变换距离”就是“aabc”全排列的个数12，即变换距离为12。接着继续观察“baaac”是如何变换到“bacaa”的。由于“baaac”和“bacaa”的首字符相同，因此可以将首字符去掉不予考虑，只考虑“aaac”和“acaa”的变换距离，与前述变换类似，也是相当于将“aac”进行了一次全排列成为“caa”，因此“aaac”和“acaa”的变换距离就是“aac”的全排列个数3再减去1，即变换距离为2。最终，“bacca”在全排列的序号为 $12+2+1=15$ 。

可将上述过程归纳为以下步骤：先将给定的字符串按字典序排列以构成初始字符串S，与目标字符串T进行比较，如果 $S < T$ ，表明仍需要进行变换；此时比较两者的首字符，如果相同，表明只需对S和T除首字符之外的其他字符继续进行上述的变换过程，否则“变换距离”将是S除首字符之外的其他所有字符的全排列数量。由于给定的字符串中可能包含重复字符，在计算全排列时需要使用包含重复字符的全排列数量计算公式。

参考代码

```
// 数字1至30的素因子分解表。
int prime[31][5] = {
    {0}, {0}, {1, 2}, {1, 3}, {2, 2, 2}, {1, 5}, {2, 2, 3}, {1, 7},
    {3, 2, 2, 2}, {2, 3, 3}, {2, 2, 5}, {1, 11}, {3, 2, 2, 3},
    {1, 13}, {2, 2, 7}, {2, 3, 5}, {4, 2, 2, 2, 2}, {1, 17},
    {3, 2, 3, 3}, {1, 19}, {3, 2, 2, 5}, {2, 3, 7}, {2, 2, 11}, {1, 23},
    {4, 2, 2, 2, 3}, {2, 5, 5}, {2, 2, 13}, {3, 3, 3, 3},
    {3, 2, 2, 7}, {1, 29}, {3, 2, 3, 5}
};

// 根据包含重复元素的全排列计数公式  $P = n! / (m_1! m_2! \cdots m_k!)$  确定给定字符串全排列的数量。
long long fullPermutation(string line) {
    vector<int> dividend, divisor;
    // 获得n!的素因子分解式。
    for (int i = 2; i <= line.length(); i++)
        for (int j = 1; j <= prime[i][0]; j++)
            dividend.push_back(prime[i][j]);
    // 计数不同字符的数量。
    int alpha[26] = {0};
    for (int i = 0; i < line.length(); i++)
        alpha[line[i] - 'a']++;
    // 获得 $m_1! m_2! \cdots m_k!$ 的素因子分解式。
    for (int i = 0; i < 26; i++)
        for (int j = 2; j <= alpha[i]; j++)
            for (int k = 1; k <= prime[j][0]; k++)
                divisor.push_back(prime[j][k]);
    // 消去公共的素因子。
    for (int i = 0; i < divisor.size(); i++)
        for (int j = 0; j < dividend.size(); j++)
            if (divisor[i] == dividend[j]) {
                dividend.erase(dividend.begin() + j);
                divisor.erase(divisor.begin() + i);
            }
    return dividend.size();
}
```

```

        break;
    }
    // 计算积。
    long long product = 1;
    for (int i = 0; i < dividend.size(); i++)
        product *= dividend[i];
    return product;
}

/*
递归计算给定字符串在全排列中的序号。以样例输入字符串“bacaa”为例来解析代码的行为。初始时
current=“bacaa”, origninal=“bacaa”。对 current 排序后, current=“aaabc”。由于 current
的长度为 5, 循环总共执行 5 次操作。每次操作先复制 current 的当前内容到另外一个字符串 next 中,
对 next 从第二个字符开始进行排序, 此时 next=“aaabc”, 由于集合 cache 为空, 故将“aaabc”插
入集合中, 比较 next 与 original, 可以发现“aaabc”<“bacaa”且 next 的首字符小于 original
的首字符, 那么很显然, “aaabc”和“bacaa”之间的“变换距离”至少是“aabc”的全排列个数 P。对
“aaabc”进行 P-1 次变换后字符串成为“acbaa”, 再进行一次变换将成为“baaac”。为了得到“acbaa”
进行一次变换后的字符串“baaac”, 参考实现通过将 current 的首字符移动到 current 的末尾并从字符串
的第二个字符开始排序以得到下一个待变换字符串, 由于集合 cache 记录了所有已经计数的字符串, 因此
不会对重复的字符串进行计数, 从而保证了结果的正确性。
*/
long long permutation(string current, string original) {
    int counter = current.length();
    long long index = 0;
    set<string> cache;
    sort(current.begin(), current.end());
    while (counter--) {
        string next(current);
        sort(next.begin() + 1, next.end());
        if (cache.find(next) == cache.end()) {
            cache.insert(next);
            if (next < original) {
                if (next.front() < original.front())
                    index += fullPermutation(current.substr(1));
                else if (current.length() > 1)
                    index += permutation(next.substr(1), original.substr(1));
            }
        }
        current += current.front();
        current.erase(0, 1);
    }
    return index;
}

int main(int argc, char* argv[]) {
    string line, original;
    while (getline(cin, line), line != "#") {
        if (line.length() == 0) continue;
        // 获得的结果是从 0 开始编号的计数, 需要调整到从 1 开始计数。
        cout << setw(10) << right << (permutation(line, line) + 1) << '\n';
    }
    return 0;
}

```

强化练习: 530 Binomial Showdown^A, 941 Permutations^B, 10460 Find the Permuted String^C。

扩展练习: 911 Multinomial Coefficients^D。

6.2.2 方程的整数解个数

给定一个黑盒，黑盒中有红、绿、蓝、黑 4 种颜色的木球若干个，若已知有 10 个木球，那么总共有多少种可能的组合？直观地看，逐个枚举并不是一种有效的方法。如果用向量 (x_1, x_2, x_3, x_4) 来定义木球的可能组合，其中 x_1 表示红色木球的数量， x_2 表示绿色木球的数量， x_3 表示蓝色木球的数量， x_4 表示黑色木球的数量，那么可能的组合数就是和为 10 的非负向量 (x_1, x_2, x_3, x_4) 的个数。如果将上述问题一般化，假设有 r 种颜色且有 n 个木球，那么可能的组合就是满足 $x_1 + x_2 + \dots + x_r = n$ 的非负整数向量 (x_1, x_2, \dots, x_r) 的个数。要计算这个数，可以先考虑符合上述条件的正整数向量 (x_1, x_2, \dots, x_r) 的个数，可以使用隔板法（Stars and Bars，又称插空法）¹ 对其进行计数。



图 6-1 隔板法（插空法）

如图 6-1 所示, 将 n 个 1 排列成一列, 则 n 个 1 之间有 $n-1$ 个“空隙”, 在这 $n-1$ 个空隙中任意选出 $r-1$ 个空隙放入一个假想的“隔板”, 则会将这 n 个 1 分隔为 r 个互不相连的部分, 划分后的结果恰好对应满足 $x_1+x_2+\cdots+x_r=n$ 的一种正整数解: 使得 x_1 等于第一个隔板之前 1 的个数, x_2 等于第一个和第二个插入的隔板之间的 1 的个数, \cdots , x_r 等于最后一个插入的隔板后面的 1 的个数。也就是说, $x_1+x_2+\cdots+x_r=n$ 的正整数解以一对一的方式对应于从 $n-1$ 个空隙中选择 $r-1$ 个空隙的结果, 由此得出, $x_1+x_2+\cdots+x_r=n$ 的不同的正整数解的个数等于从 $n-1$ 个空隙里选择 $r-1$ 个空隙的方法个数, 因此有以下结论: 共有 $\binom{n-1}{r-1}$ 个不同的正整数向量 (x_1, x_2, \cdots, x_r) 满足 $x_1+x_2+\cdots+x_r=n$, $x_i>0$, $1 \leq i \leq r$ 。

不难得出, $x_1+x_2+\cdots+x_r=n$ 的非负整数解个数与 $y_1+y_2+\cdots+y_r=n+r$ 的正整数解个数是相同的, 因为对于 $x_1+x_2+\cdots+x_r=n$ 的任意一组解(x_1, x_2, \dots, x_r), 均可通过令 $y_i=x_i+1 (1 \leq i \leq r)$ 的方式来得到 $y_1+y_2+\cdots+y_r=n+r$ 的一组解(y_1, y_2, \dots, y_r)。因此有以下结论: 共有 $\binom{n+r-1}{r-1}$ 个不同的正整数向量(x_1, x_2, \dots, x_r)满足 $x_1+x_2+\cdots+x_r=n$ 。

根据上述结论易得 4 种颜色的木球构成总数是 10 个木球的组合方案数为 $\binom{10+4-1}{4-1} = \binom{13}{3} = 286$ 种。

强化练习: 10541* Stripe^C, 13135 Homework^D。

6.3 Pólya 计数定理

由于 Pólya 计数定理与群的概念有密切的关系,为了便于理解,本节先从基础的概念,如等价关系、群、置换群等开始介绍,之后是 Burnside 引理,最后给出 Pólya 计数定理的一般表示形式和具体应用。

¹ 国外的组合数学教材中亦将其称之为“Stars and Bars”计数定理，读者可进一步查阅 Wikipedia 上关于该定理的介绍：[https://en.wikipedia.org/wiki/Stars_and_bars_\(combinatorics\)](https://en.wikipedia.org/wiki/Stars_and_bars_(combinatorics)), 2020。

6.3.1 基本概念

等价关系

设 A 和 B 为两个集合, 如果对于任意 $a \in A$, B 中都唯一存在元素 $f(a)$ 与之对应, 则称 f 为由 A 到 B 的一个映射, 记作 $f: A \rightarrow B$ 。此时 $f(a)$ 称为 a (在 f 下) 的像 (image), a 称为 $f(a)$ (在 f 下) 的原像 (preimage) 或反像 (inverse image)。如果 A 中任意两个不同元素在 f 下的像都不同, 则称 f 为单射 (injection)。如果 B 中任一元素在 A 中都有原像, 则称 f 为满射 (surjection)。既是单射又是满射的映射称为双射 (bijection), 或一一对应 (one-to-one) [46]。

给定集合 A 和 B , A 和 B 的元素构成的有序对 (ordered pair) 的全体记为

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

称上述有序对的全体 $A \times B$ 为 A 和 B 的笛卡尔积 (Cartesian product), 简称卡氏积或直积。将集合 A 上的一个二元运算 (binary operation) 定义为由 $A \times A$ 到 A 的一个映射。给定 $A \times A$ 的某个子集 R , 将子集 R 称为 A 上的关系 (relation)。当 $(a, b) \in R$ 时, 称 a 与 b 具有关系 R , 记为 aRb , 当 $(a, b) \notin R$ 时, 称 a 与 b 不具有关系 R , 记为 $a \bar{R} b$ 。对于 $A \times B$ 中的任意一个有序对 (a, b) , 它要么属于 R , 要么不属于 R , 因此一定有 aRb 或者 $a \bar{R} b$ 。

对于 $A \times A$ 上的关系 R , 可以定义如下的性质:

- (1) 反身性 (reflexivity): 如果对于所有的 $a \in A$, 均有 aRa , 则称 R 具有反身性。
- (2) 对称性 (symmetry): 如果对于所有的 $a, b \in A$, 当 aRb 时, 均有 bRa , 则称 R 具有对称性。
- (3) 传递性 (transitivity): 如果对于所有的 $a, b, c \in A$, 当 aRb 且 bRc 时, 均有 aRc , 则称 R 具有传递性。

在数学上, 将具有反身性, 对称性, 传递性的关系 R 称为等价关系 (equivalence relation), 此时 A 中互相等价的元素组成的子集称为一个等价类 (equivalence class)。

群

给定一个集合 $G = \{a, b, c, \dots\}$ 以及 G 上的运算 (operation) \circ , 如果 G 和 \circ 满足以下性质:

- (1) 封闭性 (closure): 对于任意的 $a, b \in G$, $a \circ b \in G$ 成立。
- (2) 结合律 (associativity) 成立: 对于任意的 $a, b, c \in G$, $a \circ (b \circ c) = (a \circ b) \circ c$ 成立。
- (3) 存在幺元 (identity element): G 中存在一个元素 e , 使得对于 G 的任意元素 a , 均有 $a \circ e = e \circ a = a$ 。将元素 e 称为幺元。
- (4) 存在逆元 (inverse element): 对于 G 中的任意元素 a , 均存在 $b \in G$, 使得 $a \circ b = b \circ a = e$, 将元素 b 称为元素 a 的逆元, 记为 a^{-1} , 即 $b = a^{-1}$ 。

将满足上述性质的集合 G 连同运算 \circ 称为群 (group) [47], 记为 (G, \circ) 。若 G 是一个有限集, 则称 (G, \circ) 为有限群, 其中有限群 G 的元素个数称为群的阶 (order), 记为 $|G|$; 若 G 是无限集, 则称 (G, \circ) 为无限群。

^I 有的著作考虑到代数中各种定理适用性范围的不同, 采用一种“渐进”的方式来定义群。令 G 为非空集合, 定义 G 上的二元运算 (binary operation) 为函数: $G \times G \rightarrow G$ 。易知, 定义在 G 上的二元运算满足封闭性。如果 G 上的二元运算 \circ 满足结合律, 则称之为半群 (semigroup)。将存在幺元的半群称为幺半群 (monoid)。将存在逆元的幺半群称为群 (group)。

限群。为了简便，在不引起歧义的情况下， G 中的元素 a 和 b 的运算 $a \circ b$ 可以简记为 ab 。

如果群 G 中的任意两个元素 a 和 b 满足交换律 (commutativity)，即满足 $ab=ba$ ，则称 G 为阿贝尔群 (Abelian group)¹，又称交换群。

例如， $G=\{x|x\in\mathbb{R}\}$ 在加法运算 (+) 下是一个群，因其满足群定义的四个性质：

- (1) 封闭性：任意两个实数相加后仍为实数，即对于 $a, b \in G$ ， $a+b \in G$ 成立；
- (2) 结合律成立：在实数域上的加法运算满足结合律，即对于任意的 $a, b, c \in G$ ， $(a+b)+c=a+(b+c)$ ，因此加法运算是可结合的；
- (3) 存在幺元： $0 \in G$ ，且对于任意 $a \in G$ ， $0+a=a+0=a$ 成立，即 G 中存在幺元 $e=0$ ；
- (4) 存在逆元：对于任意 $a \in G$ ， $-a \in G$ ，且 $(a)+(-a)=(-a)+(a)=0$ ，故有 $(a)^{-1}=-a$ ， $(-a)^{-1}=a$ 。

类似于子集的概念，可以定义子群的概念，令 G 在 \circ 运算下是一个群，如果 H 是 G 的非空子集而且 H 在 \circ 运算下也是一个群，则称 (H, \circ) 是 (G, \circ) 的子群，记作 $H \leq G$ 。

置换群

令 M 为一个非空有限集合，将 M 的某个一对一变换称为置换 (permutation)，一对一变换是指 M 到自身的一一对应。假设 M 的元素为 a_1, a_2, \dots, a_n ，则 M 的置换 σ 可以简记为

$$\sigma = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{pmatrix}, \quad b_i = \sigma(a_i), \quad i = 1, 2, \dots, n$$

按照置换的定义，它是 M 到自身的一一对应，所以 b_1, b_2, \dots, b_n 是 a_1, a_2, \dots, a_n 的一个排列，由此可见， M 的每一个置换都对应 a_1, a_2, \dots, a_n 的一个排列，不同的置换对应不同的排列。相应地， a_1, a_2, \dots, a_n 的任意一个排列也确定 M 的一个置换，所以 M 的置换共有 $n!$ 个，其中 n 为 M 的元数。 M 上的置换也称为 n 元置换。特别地，如果

$$\sigma = \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$$

即 M 中的每个元素都与自身相对应，则称该置换 σ 为 n 元恒等置换。

按照从左向右的运算结合顺序，令 a 为 M 的任意一个元素， σ 和 τ 为 M 的任意两个置换，定义置换 σ 和 τ 的乘法为两个置换的复合

$$\sigma\tau(a) = \tau(\sigma(a))$$

即先进行置换 σ 所指定的变换，然后进行置换 τ 所指定的变换。例如，令置换

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 1 & 3 & 4 \end{pmatrix}, \quad \tau = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix}$$

有

$$\sigma\tau(1) = \tau(\sigma(1)) = \tau(2) = 4, \quad \tau\sigma(1) = \sigma(\tau(1)) = \sigma(3) = 3$$

因此有

$$\sigma\tau = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 3 & 1 & 2 \end{pmatrix}, \quad \tau\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 2 & 1 \end{pmatrix}$$

¹ 尼尔斯·亨利克·阿贝尔 (Niels Henrik Abel, 1802—1829)，挪威数学家。2002 年 1 月 1 日挪威政府设立的 Abel 奖 (Abel Prize) 即以他的名字命名。Abel 奖的目的是奖励在数学领域做出杰出贡献的人士，每年颁发一次，奖金为 600 万挪威克朗 (约合 70 万美元)。由于诺贝尔奖中并无数学奖项，该奖项一般被人们认为是诺贝尔奖的一种补充。参考：<https://www.abelprize.no/>，2020。

易知, $\sigma\tau \neq \tau\sigma$ 。一般来说, 置换的乘法不满足交换律。

令 S_n 表示 n 元集合 M 上所有 n 元置换构成的集合, 则 S_n 在置换乘法运算下是一个群, 因其满足群的定义:

- (1) 封闭性: 对于任意两个置换 $\sigma, \tau \in S_n$, 有 $\sigma\tau \in S_n$;
- (2) 结合律成立: $(\sigma\tau)\rho = \sigma(\tau\rho)$, $\sigma, \tau, \rho \in S_n$;
- (3) 存在幺元: n 元恒等置换 σ_0 是 S_n 中的幺元, 有 $\sigma_0\tau = \tau\sigma_0 = \tau$, S_n 的幺元 e 即为 σ_0 ;
- (4) 存在逆元: 每个 n 元置换都在 S_n 中存在逆元, 即

$$\begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ b_1 & b_2 & \cdots & b_n \end{pmatrix}^{-1} = \begin{pmatrix} b_1 & b_2 & \cdots & b_n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$$

由于一般情况下置换的乘法不满足交换律, 因此当 $n \geq 3$ 时, S_n 不是交换群。

置换群是群论的基础, Cayley 定理指出: 所有的有限群都可以用置换群予以表示。

强化练习: 253 Cube Painting^A。

轮换

从置换的定义可以看出, 它是有限集 M 到自身的一对一变换, 这正好符合映射的定义, 因此置换也可以使用函数的概念来定义, 即可将有限集 M 上的一个双射函数称为 M 上的一个置换。给定集合 $M = \{1, 2, 3, 4, 5, 6\}$, 定义一个在集合 M 上的双射函数 $f(x)$, 其函数值列表为

$$f(1) = 4, f(2) = 6, f(3) = 5, f(4) = 3, f(5) = 1, f(6) = 2$$

使用常用的两行表示法, 可以将置换表示为

$$\sigma = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 6 & 5 & 3 & 1 & 2 \end{pmatrix}$$

观察给出的置换, 可以发现 1 映射到 4, 4 映射到 3, 3 映射到 5, 5 映射到 1, 形成了一个循环, 循环的长度为 4; 另外 2 映射到 6, 6 又映射到 2, 也形成了一个循环, 循环的长度为 2。根据这个特点, 可以采用以下的一行表示法来表示置换, 即将置换表示为

$$\sigma = (1 \ 4 \ 3 \ 5)(2 \ 6)$$

人们发现, 使用循环的方式来表示置换, 既能反映置换的结构又能便于计算, 因此这种表示方法逐渐广为采用。一般将长度为 m 的循环记为

$$(a_1 a_2 \cdots a_{m-1} a_m) = \begin{pmatrix} a_1 & a_2 & \cdots & a_{m-1} & a_m \\ a_2 & a_3 & \cdots & a_m & a_1 \end{pmatrix}$$

即 a_1 变换为 a_2 , a_2 变换为 a_3 , \cdots , a_m 变换为 a_1 , 将其称为轮换 (cycle), 并将 m 称为轮换的长度 (length) 或阶 (order)。特别地, 当 $m=2$ 时, 2 阶轮换 (i, j) 称为 i 和 j 的对换 (transposition)。如果两个轮换 $(a_1 a_2 \cdots a_m)$ 和 $(b_1 b_2 \cdots b_n)$ 之间没有相同的元素, 则称这两个轮换是不相交的 (disjointed)。不相交的两个轮换的乘积可以交换, 例如置换 $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 2 & 5 & 4 \end{pmatrix}$ 可以写成 $(1 \ 3 \ 2)(4 \ 5)$, 也可以写成 $(4 \ 5)(1 \ 3 \ 2)$, 两者互相等价^I。不难看出, 轮换中哪个元素排列在最前对轮换所表示的置换并无影响, 也就是说, $(a_1 a_2 a_3 \cdots a_{m-1} a_m)$ 和 $(a_2 a_3 \cdots a_{m-1} a_m a_1)$ 所表示的置换本质上是相同的。

对于置换的轮换表示, 有以下两个结论:

^I 轮换的乘积运算类似于置换的复合运算。在将置换写成轮换的乘积时, 一般会将一阶轮换予以省略, 即

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 2 & 5 & 4 \end{pmatrix} = (1 \ 3 \ 2)(4 \ 5) = \left[\begin{pmatrix} 1 & 3 & 2 \\ 3 & 2 & 1 \end{pmatrix} \begin{pmatrix} 4 & 5 \\ 4 & 5 \end{pmatrix} \right] \left[\begin{pmatrix} 4 & 5 \\ 5 & 4 \end{pmatrix} \begin{pmatrix} 1 & 2 \\ 1 & 2 \end{pmatrix} \begin{pmatrix} 3 \\ 3 \end{pmatrix} \right]$$

(1) 任何一个置换都可以表示成若干个互不相交的轮换的乘积, 如果不考虑轮换的次序可以交换, 这种表示方法是唯一的。

(2) 任意一个轮换都可以表示成若干个对换的乘积。

由于任何一个置换可以表示成互不相交的轮换的乘积, 在不考虑次序的情况下, 其表示方式唯一, 人们将某个置换表示为互不相交的轮换乘积时轮换的个数称为轮换数。例如置换 $\sigma = (1 \ 2 \ 3 \ 4 \ 5) (3 \ 1 \ 2 \ 5 \ 4)$ 表示为轮换的乘积时可以写成 $(1 \ 3 \ 2)(4 \ 5)$, 故置换 σ 的轮换数为 2。

需要注意的是, 任意轮换分解成若干个对换乘积不是唯一的, 甚至连换位个数都不相同。例如,

$$(1 \ 2 \ 3) = (1 \ 3)(3 \ 2) = (2 \ 1)(1 \ 3) = (1 \ 2)(1 \ 3)(3 \ 1)(1 \ 3)$$

但是, 轮换分解成对换的乘积时, 换位个数的奇偶性是不变的, 它不随对换个数的不同而不同, 要么分解为奇数个对换之积, 要么分解为偶数个对换之积。若一个置换可以分解成奇数个对换之积, 称为奇置换; 若可以分解为偶数个对换之积, 称为偶置换。

置换的轮换表示有一个缺点, 即在省略掉 1 阶轮换时可能难以看出置换的元数, 一般需要在使用轮换表示时指定置换的元数。例如, “8 元置换 $(1 \ 5 \ 7 \ 3 \ 6)$ ” 表示

$$(1 \ 5 \ 7 \ 3 \ 6)(2)(4)(8) = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8) \\ (5 \ 2 \ 6 \ 4 \ 7 \ 1 \ 3 \ 8)$$

而“10 元置换 $(1 \ 5 \ 7 \ 3 \ 6)$ ”表示

$$(1 \ 5 \ 7 \ 3 \ 6)(2)(4)(8)(9)(10) = (1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10) \\ (5 \ 2 \ 6 \ 4 \ 7 \ 1 \ 3 \ 8 \ 9 \ 10)$$

与置换的轮换表示有关的题目, 可能会以下列形式出现: 给定 1 到 n 的一个排列和某种变换规则, 每次对当前排列按照变换规则进行一次变换, 要求确定经过多少次变换, 排列能够恢复到初始的状态。例如, 给定 1 到 10 的一个排列

$$(1, 3, 5, 9, 6, 10, 8, 2, 7, 4)$$

以及变换规则

$$\begin{array}{cccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \downarrow & \downarrow \\ 3 & 7 & 4 & 2 & 1 & 9 & 10 & 8 & 6 & 5 \end{array}$$

按照变换规则, 经过第一次变换, 原始排列变为

$$(3, 4, 1, 6, 9, 5, 8, 7, 10, 2)$$

那么, 要经过多少次变换, 排列才能恢复到原始排列状态呢?

要解决此类问题, 可以将题目给定的变换规则看成是一个置换, 使用轮换的方式来表示该置换并确定各个轮换的长度 o_i , 显然, 对于单个轮换来说, 只要经过 o_i 次变换, 该轮换所对应的部分元素的排列就能够恢复到初始的状态, 那么求出所有轮换的长度 o_i 的最小公倍数 L , 那么经过 L 次变换后, 初始排列肯定能够恢复到初始状态。按照上述方法, 给定的变换规则对应以下轮换表示的置换

$$(1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 8 \ 9 \ 10) \\ (3 \ 7 \ 4 \ 2 \ 1 \ 9 \ 10 \ 8 \ 6 \ 5) = (1 \ 3 \ 4 \ 2 \ 7 \ 10 \ 5)(6 \ 9)(8)$$

该置换包含 3 个轮换, 各个轮换的长度依次分别为 7、2、1, 则 $L=14$, 也就是说, 只需经过 14 次变换, 初始排列就能恢复到原始状态。

强化练习: 239 Time and Motion^D, 306 Cipher^B, 10409 Die Game^A, 11959 Dice^D。

扩展练习: 11774* Doom's Day^D。

轮换指标

根据前述的结论, 可将 S_n 中任一置换 P 分解成若干互不相交的轮换的乘积, 即

$$P = (a_1 a_2 \cdots a_{k_1})(b_1 b_2 \cdots b_{k_2}) \cdots (h_1 h_2 \cdots h_{k_m}), \quad k_1 + k_2 + \cdots + k_m = n$$

令其中长度为 k 的轮换出现的次数为 C_k , $k=1, 2, \dots, n$ 。如果长度为 k 的轮换出现 C_k 次, 使用 $(k)^{C_k}$ 予以表示。那么 S_n 中的置换可按分解成

$$(1)^{C_1}(2)^{C_2}(3)^{C_3} \cdots (n)^{C_n}$$

的不同而分类, 其中 $C_i=0$ 的项 $(i)^{C_i}$ 可以省略不写 ($i=1, 2, \dots, n$)。例如, $(1)(2)(3 \ 4)(5 \ 6 \ 7)$ 属于格式 $(1)^2(2)^1(3)^1$ 。将 S_n 中具有相同上述格式的置换的全体称为与格式相应的共轭类。容易得到

$$n = \sum_{k=1}^n k \cdot C_k$$

且有结论: S_n 中属于 $(1)^{C_1}(2)^{C_2}(3)^{C_3} \cdots (n)^{C_n}$ 共轭类的元素个数为

$$\frac{n!}{C_1! C_2! \cdots C_n! 1^{C_1} 2^{C_2} \cdots n^{C_n}}$$

例如, S_4 的全体为^[48]

$$(1)(2)(3)(4)$$

$$(1 \ 2)(3)(4), (1 \ 3)(2)(4), (1 \ 4)(2)(3), (2 \ 3)(1)(4), (2 \ 4)(1)(3), (3 \ 4)(1)(2)$$

$$(1 \ 2 \ 3)(4), (1 \ 2 \ 4)(3), (1 \ 3 \ 2)(4), (1 \ 3 \ 4)(2), (1 \ 4 \ 2)(3), (1 \ 4 \ 3)(2), (2 \ 3 \ 4)(1), (2 \ 4 \ 3)(1)$$

$$(1 \ 2)(3 \ 4), (1 \ 3)(2 \ 4), (1 \ 4)(2 \ 3)$$

$$(1 \ 2 \ 3 \ 4), (1 \ 2 \ 4 \ 3), (1 \ 3 \ 2 \ 4), (1 \ 3 \ 4 \ 2), (1 \ 4 \ 2 \ 3), (1 \ 4 \ 3 \ 2)$$

在 S_4 中具有 $(1)^4$ 格式的置换有 $\frac{4!}{4! \times 1} = 1$ 个, 为

$$(1)(2)(3)(4)$$

在 S_4 中具有 $(1)^2(2)^1$ 格式的置换有 $\frac{4!}{2! \times 2} = 6$ 个, 为

$$(1 \ 2)(3)(4), (1 \ 3)(2)(4), (1 \ 4)(2)(3), (2 \ 3)(1)(4), (2 \ 4)(1)(3), (3 \ 4)(1)(2)$$

在 S_4 中具有 $(1)^1(3)^1$ 格式的置换有 $\frac{4!}{1! \times 3} = 8$ 个, 为

$$(1 \ 2 \ 3)(4), (1 \ 2 \ 4)(3), (1 \ 3 \ 2)(4), (1 \ 3 \ 4)(2), (1 \ 4 \ 2)(3), (1 \ 4 \ 3)(2), (2 \ 3 \ 4)(1), (2 \ 4 \ 3)(1)$$

在 S_4 中具有 $(2)^2$ 格式的置换有 $\frac{4!}{2! \times 4} = 3$ 个, 为

$$(1 \ 2)(3 \ 4), (1 \ 3)(2 \ 4), (1 \ 4)(2 \ 3)$$

在 S_4 中具有 $(4)^1$ 格式的置换有 $\frac{4!}{4} = 6$ 个, 为

$$(1 \ 2 \ 3 \ 4), (1 \ 2 \ 4 \ 3), (1 \ 3 \ 2 \ 4), (1 \ 3 \ 4 \ 2), (1 \ 4 \ 2 \ 3), (1 \ 4 \ 3 \ 2)$$

根据共轭类的概念, 可以定义轮换指标。设 (G, \circ) 是 n 元集合 A 上的一个置换群, x_1, x_2, \dots, x_n 是 n 个未定元, 对每个 $g \in G$, 以 $b_i(g)$ ($i=1, 2, \dots, n$) 表示 g 的轮换分解式所含有的长为 i 的轮换的个数, 令

$$P_G(x_1, x_2, \dots, x_n) = \frac{1}{|G|} \sum_{g \in G} x_1^{b_1(g)} x_2^{b_2(g)} \cdots x_n^{b_n(g)}$$

将 $P_G(x_1, x_2, \dots, x_n)$ 称为 A 上的置换群 (G, \circ) 的轮换指标。根据轮换指标的定义, 有

$$P_{S_4}(x_1, x_2, x_3, x_4) = \frac{1}{24} (x_1^4 + 6x_1^2x_2 + 8x_1^2x_3 + 3x_2^2 + 6x_4)$$

6.3.2 Burnside 引理

如图 6-2 所示, 给定具有 4 个顶点的正方形:

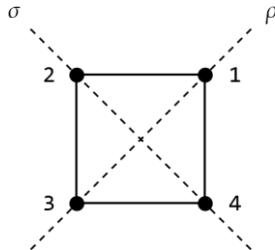


图 6-2 具有 4 个顶点的正方形

在其上定义四种变换, 第一种是绕正方形的中心点顺时针旋转 0 度, 记为 e , 第二种是沿主对角线进行的反射变换, 记为 σ , 第三种是沿副对角线进行的反射变换, 记为 ρ , 第四种是绕正方形的中心点顺时针旋转 180 度, 记为 τ 。可以使用置换来表示这四种变换, 它们分别为

$$\begin{aligned} e &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} = (1)(2)(3)(4) \\ \sigma &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 2 & 1 & 4 \end{pmatrix} = (1 \ 3)(2)(4) \\ \rho &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 4 & 3 & 2 \end{pmatrix} = (2 \ 4)(1)(3) \\ \tau &= \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix} = (1 \ 3)(2 \ 4) \end{aligned}$$

容易验证, $G=\{e, \sigma, \rho, \tau\}$ 构成置换群。

轨道

给定前述的置换群 $G=\{e, \sigma, \rho, \tau\}$, 对任意顶点进行任意次置换操作后, 可以发现这样的事实: 顶点 1 不论进行何种置换, 只能变换为 3, 而不可能变换为 2、4; 顶点 2 不论进行何种置换, 只能变换为 4, 而不可能变换为 1、3。在群 G 的作用下, 顶点 1 只能在 1、3 之间进行变换, 这种现象类似于行星按照固定的轨迹围绕太阳旋转, 因此人们将集合 $\{1, 3\}$ 称为 1 在 G 的作用下的轨道 (orbit)^I, 记为 E_1 。易知, 初始的顶点集合 $\{1, 2, 3, 4\}$ 在群 G 的作用下被划分为两个轨道, $\{1, 3\}$ 和 $\{2, 4\}$ 。

稳定子群

令 G 是 $X=\{1, 2, \dots, n\}$ 上的置换群, 显然 G 是 S_n 的一个子群, 若 k 是 1 到 n 中的某个整数, 则将 G

^I 以下是轨道更为形式化的定义。设 G 是一个群, S 是一个集合, 映射

$$f: G \times S \rightarrow S, (g, s) \mapsto f(g, s) \text{ (也简记为 } g(s) \text{)}$$

如果它满足以下两个条件: (1) 对任意 $s \in S$, 有 $e(s)=s$; (2) 对任意 $g_1, g_2 \in G, s \in S$, 有 $g_1(g_2(s))=(g_1g_2)(s)$, 则称为群 G 在 S 上的一个作用 (act)。设给定了一个群作用 $f: G \times S \rightarrow S$, 我们可以在 S 上定义一个二元关系 “~”: 对于 $s_1, s_2 \in S$,

$$s_1 \sim s_2 \Leftrightarrow \text{存在 } g \in G \text{ 使得 } g(s_1) = s_2$$

可以验证, 在这样的定义下, 群的定义中的三条性质保证了 “~” 作为等价关系的三条性质, 将 S 在这个等价关系下的等价类称为轨道。

中使 k 保持不变的置换全体称为 G 中 k 的稳定子群 (stabilizer)，记为 Z_k 。换句话说， k 的稳定子群就是那些经过变换后 k 仍为自身的置换的集合。例如，假设有置换群

$$G = \{e, (1\ 3), (2\ 4), (1\ 3)(2\ 4)\}$$

则有

$$Z_1 = \{e, (2\ 4)\}, Z_2 = \{e, (1\ 3)\}, Z_3 = \{e, (2\ 4)\}, Z_4 = \{e, (1\ 3)\}$$

相应地，对于 G 中的某个置换，如果置换前后， k 保持不变，则称 k 为不动点，亦即使用轮换表示时长度为 1 的轮换。

关于 k 的稳定子群，有以下结论：

- (1) 群 G 中关于 k 的稳定子群 Z_k 是 G 的一个子群。
- (2) 轨道-稳定子群定理 (orbit-stabilizer theorem): $|E_k| |Z_k| = |G|$, $k=1, 2, \dots, n$ 。

Burnside 引理

令 G 是 $X=\{1, 2, \dots, n\}$ 上的置换群， G 在 X 上可引出不同的轨道 (即等价类)，其个数为

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} c(g)$$

其中 $c(g)$ 表示置换 g 中不动点的数量，亦即当置换 g 使用轮换的乘积形式表示时长度为 1 的轮换的个数。换句话说，Burnside 引理提供了一种方法，可以通过计数 G 中置换的不动点个数的总和然后取平均值 (将总和除以置换群的阶 $|G|$) 来得到轨道的数目。例如，令

$$G = \{e, (1\ 3), (2\ 4), (1\ 3)(2\ 4)\}$$

由前述讨论可知， G 是阶为 4 的置换群，对于 G 中的四个置换，其不动点个数分别 4、2、2、0，则轨道数目为

$$|X/G| = \frac{1}{4}(4 + 2 + 2 + 0) = 2$$

与 G 划分为两个轨道 {1, 3} 和 {2, 4} 的结果相符。

那么如何在解题中应用 Burnside 引理呢？注意到 Burnside 引理是求轨道数目，而轨道指的是等价类。在解题中，等价类的意义可以理解为在置换变换下具有相同效果的操作数目，其中最为常见的就是给指定物品进行染色的方案数。可以将其归结为以下步骤：(1) 在不考虑旋转的情况下，列出所有不同的染色方案；(2) 确定不同的变换种数，从而确定置换的种数；(3) 在每种置换下确定经过置换后仍然相同的染色方案，这些染色方案都属于不动点；(4) 将不动点的总数除以置换的种数即为所求。

在解题中，常见的是正多边形和正多面体的变换群，其中正多面体又以正六面体 (正方体) 最为常见。对于正多边形的变换群，有旋转变换和翻转变换两种。 n ($n \geq 3$) 个顶点构成的正多边形在进行旋转变换时，沿着某个固定的方向依次转动 i 个 ($i=0, 1, \dots, n-1$) 个顶点时，所构成置换的轮换数为 n 和 i 的最大公约数 $\gcd(n, i)$ 。例如，当 $n=6$ 时，顺时针转动多边形，有

$$\text{转动 0 个顶点, 轮换数为 } \gcd(6, 0) = 6: \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 1 & 2 & 3 & 4 & 5 & 6 \end{pmatrix} = (1)(2)(3)(4)(5)(6)$$

$$\text{转动 1 个顶点, 轮换数为 } \gcd(6, 1) = 1: \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 1 \end{pmatrix} = (1\ 2\ 3\ 4\ 5\ 6)$$

$$\text{转动 2 个顶点, 轮换数为 } \gcd(6, 2) = 2: \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 3 & 4 & 5 & 6 & 1 & 2 \end{pmatrix} = (1\ 3\ 5)(2\ 4\ 6)$$

$$\text{转动 3 个顶点, 轮换数为 } \gcd(6, 3) = 3: \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 4 & 5 & 6 & 1 & 2 & 3 \end{pmatrix} = (1\ 4)(2\ 5)(3\ 6)$$

转动 4 个顶点, 轮换数为 $\gcd(6, 4) = 2$: $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 5 & 6 & 1 & 2 & 3 & 4 \end{pmatrix} = (1\ 5\ 3)(2\ 6\ 4)$

转动 5 个顶点, 轮换数为 $\gcd(6, 5) = 1$: $\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 6 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} = (1\ 6\ 5\ 4\ 3\ 2)$

如图 6-3 所示, 对于翻转变换来说, 当 n 为奇数和偶数时其相应的置换有所不同。当 n 为奇数时, 翻转轴为某个顶点和其对边中点的连线, 共有 n 种翻转方式, 所形成置换的轮换数为 $(n+1)/2$, 其中包含 1 个长度为 1 的轮换, $(n-1)/2$ 个长度为 2 的轮换。当 n 为偶数时, 有两种类型的翻转轴, 一种是以某个顶点及其相对顶点的连线为翻转轴, 所形成置换的轮换数为 $(n+2)/2$, 其中包含 2 个长度为 1 的轮换, $(n-2)/2$ 个长度为 2 的轮换; 另外一种是经过对边中点的连线为翻转轴, 所形成置换的轮换数为 $n/2$, 包含 $n/2$ 个长度为 2 的轮换。

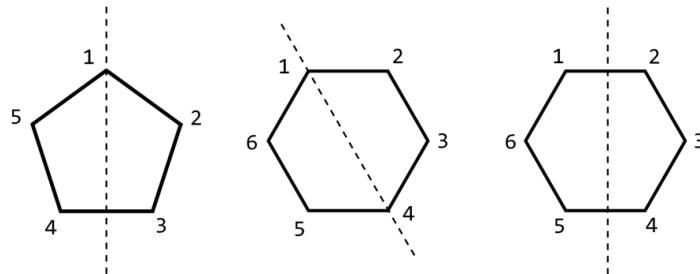


图 6-3 正五边形和正六边形的翻转轴

对于正方体而言, 其常见的转动群的阶均为 24, 如面转动群、顶点转动群、边转动群, 可以将其在不同转动方式下的共轭类列表如下:

转动方式	面共轭类	顶点共轭类	棱共轭类	个数
静止不动	$(1)^8$	$(1)^8$	$(1)^8$	1
面心一面心, ± 90 度	$(4)^2$	$(1)^2(4)^1$	$(4)^3$	6
面心一面心, 180 度	$(2)^4$	$(1)^2(2)^2$	$(2)^6$	3
棱心一棱心, 180 度	$(2)^4$	$(2)^3$	$(1)^2(2)^5$	6
空间对角线, ± 120 度	$(3)^2(1)^2$	$(3)^2$	$(3)^4$	8

下面以一个简单的示例来说明如何在解题中应用 Burnside 引理。给定黑色和白色两种颜色, 要求给一个包含四个相同方格的正方形进行染色, 假定绕正方形的中心旋转后相同的染色方案认为是同等的染色方案, 那么总共有多少种不同的染色方案呢? 如图 6-4 所示, 对正方形进行染色, 如果不区分绕正方形的中心旋转后相同的染色方案, 共有 16 种染色方案:

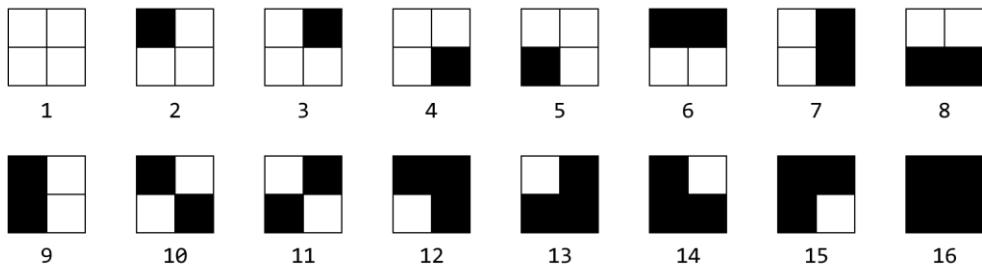


图 6-4 使用黑色和白色两种颜色对具有四个方格的正方形进行染色

如果需要确定绕正方形中心旋转后不同的染色方案数目，则等价于求 16 种染色方案在旋转变换下被划分成的轨道数量，即将绕正方形中心顺时针旋转 0 度、90 度、180 度、270 度视为在染色方案集合 $X=\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16\}$ 上的四种置换，即

(1) 顺时针旋转 0 度所对应的置换：

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \end{pmatrix}$$

(2) 顺时针旋转 90 度所对应的置换：

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 3 & 4 & 5 & 2 & 7 & 8 & 9 & 6 & 11 & 10 & 13 & 14 & 15 & 12 & 16 \end{pmatrix}$$

(3) 顺时针旋转 180 度所对应的置换：

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 4 & 5 & 2 & 3 & 8 & 9 & 6 & 7 & 10 & 11 & 14 & 15 & 12 & 13 & 16 \end{pmatrix}$$

(4) 顺时针旋转 270 度所对应的置换：

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 1 & 5 & 2 & 3 & 4 & 9 & 6 & 7 & 8 & 11 & 10 & 15 & 12 & 13 & 14 & 16 \end{pmatrix}$$

根据 Burnside 引理，轨道数目为所有置换不动点数量总和的均值，为

$$|X/G| = \frac{1}{|G|} \sum_{g \in G} c(g) = \frac{1}{4} (16 + 2 + 4 + 2) = 6$$

即旋转后不同的染色方案数目为 6 种，如图 6-5 所示，它们分别是：



图 6-5 绕正方形中心旋转后不同的染色方案

6.3.3 Pólya 计数定理

利用 Burnside 引理进行计数需要首先列出 n^m 种可能的染色方案，然后找出在每个置换下保持不变的染色方案数，对于某些计数问题，如果颜色种数 m 和对象数 n 较大，则使用 Burnside 引理进行计数将非常繁琐，此时可以利用 Pólya 计数定理 (Pólya enumeration theorem) 进行计数。

Pólya 计数定理

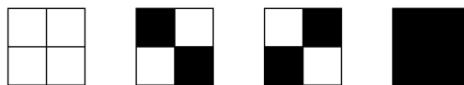
令 $G=\{a_1, a_2, \dots, a_s\}$ 是 n 个对象的置换群，用 m 种颜色给这 n 个对象着色，则不同的着色方案数为

$$|X/G| = \frac{1}{|G|} \sum_{i=1}^g m^{c(a_i)} = \frac{1}{|G|} \{m^{c(a_1)} + m^{c(a_2)} + \dots + m^{c(a_g)}\}$$

其中 $c(a_i)$ 为置换 a_i 的轮换数 (即将 a_i 分解为不相交的轮换乘积时轮换的个数), $i=1, 2, \dots, g$ 。

Pólya 计数定理实际上是 Burnside 引理的一种具体化, 它使用了轮换的性质来使得计算不动点的数量更为简便。根据 Burnside 引理, 等价类的数量为不动点总数除以置换的种类数, 需要确定在各个置换下不动点的数量。在将置换分解为不相交轮换的乘积后, 如果某个染色方案在此置换下为不动点, 显然要求在某个轮换内的元素使用相同的颜色, 而各个轮换之间使用的颜色可以相互独立, 因此根据乘法原理, 在此置换下, 不动点的数量只与颜色数和轮换数有关。换句话说, 令染色颜色数为 n , 置换的轮换数为 c , 则置换的不动点数目为 n^c 。

现在使用 Pólya 计数定理来计算在旋转变换下不同的染色方案数量。以顺时针旋转 180 度为例, 此时的不动点有 4 个, 它们分别是:



为方格按以下方式编号:

1	2
4	3

容易看出, 1 号和 3 号方格, 2 号和 4 号方格必须涂上相同的颜色才能保持顺时针旋转 180 度后图形保持不变。将正方形顺时针旋转 180 度所对应的置换表示为轮换, 为

$$\begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix} = (1 \ 3)(2 \ 4)$$

为了保证顺时针旋转 180 度后图形不变 (即成为不动点), 属于同一个轮换内的方格必须使用相同的颜色, 由于每个轮换使用的颜色与其他轮换使用的颜色之间可以相互独立, 根据乘法原理, 置换表示为轮换后, 其轮换数为 2, 则不动点的数目为 2^2 。

根据上述结论, 可以有以下更为简便的方法来计数不同染色方案数量的方法。如图 6-6 所示, 将正方形顺时针旋转 0 度、90 度、180 度、270 度后, 其编号分别为:

1	2
4	3
3	2
2	1

图 6-6 从左至右依次为顺时针旋转 0 度, 顺时针旋转 90 度, 顺时针旋转 180 度, 顺时针旋转 270 度后的编号

由于四种角度的选择对应四种置换, 将其使用轮换予以表示, 分别是:

(1) 顺时针旋转 0 度所对应的置换

$$\sigma_1 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 1 & 2 & 3 & 4 \end{pmatrix} = (1)(2)(3)(4)$$

(2) 顺时针旋转 90 度所对应的置换

$$\sigma_2 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 4 & 1 & 2 & 3 \end{pmatrix} = (1 \ 4 \ 3 \ 2)$$

(3) 顺时针旋转 180 度所对应的置换

$$\sigma_3 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 3 & 4 & 1 & 2 \end{pmatrix} = (1 \ 3)(2 \ 4)$$

(4) 顺时针旋转 270 度所对应的置换

$$\sigma_4 = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 2 & 3 & 4 & 1 \end{pmatrix} = (1 \ 2 \ 3 \ 4)$$

根据 Pólya 计数定理, 不同的染色方案数为

$$|X/G| = \frac{1}{4}(2^4 + 2^1 + 2^2 + 2^1) = \frac{1}{4}(16 + 2 + 4 + 2) = 6$$

可以看到, 与使用 Burnside 引理计算得到的结果一致。如果使用 4 种颜色为格子染色, 则不同的染色方案数为

$$|X/G| = \frac{1}{4}(4^4 + 4^1 + 4^2 + 4^1) = \frac{1}{4}(256 + 4 + 16 + 4) = 70$$

在这种情况下, 如果使用 Burnside 引理进行计算, 需要列出所有 $4^4=256$ 种染色方案, 较为繁琐, 而应用 Pólya 计数定理进行计数显然更为方便。

10601 Cube^D (正方体)

给定 12 根同等长度的木棒, 每根木棒都被染上了特定的颜色。你的任务是确定使用这些木棒作为正方体的边能够构造出的不同正方体的数量。如果构建得到的两个正方体经过旋转之后重合时对应边的颜色相同, 则认为这两个正方体相同。

输入

输入的第一行是一个整数 T ($1 \leq T \leq 60$), 表示测试数据的组数。接着是 T 组测试数据。每组测试数据由一行共 12 个整数构成。每个整数表示对应序号木棒的颜色, 颜色以数字表示, 在 1 到 6 之间。

输出

对于每组测试数据输出一行, 该行包含一个整数——使用给定颜色的木棒所能构建的符合题意限制的正方体数量。

样例输出

样例输出

3	1
1 2 2 2 2 2 2 2 2 2 2 2	5
1 1 2 2 2 2 2 2 2 2 2 2	312120
1 1 2 2 3 3 4 4 5 5 6 6	

分析

正方体转动群的阶为 24, 其中边置换群可以分成以下四种类型:

(1) 静止不动的置换, 只有 1 种, 属于 $(1)^{12}$ 形式的共轭类, 为

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{pmatrix} = (1)(2)(3)(4)(5)(6)(7)(8)(9)(10)(11)(12)$$

(2) 以正方体的两个相对的面的中心连线为轴, 旋转 ± 90 度、 180 度的置换, 因为正方体有 6 个面, 故有 3 种轴线的旋转, 共 9 种置换。其中旋转 ± 90 度的置换同属 $(4)^3$ 的共轭类, 有 6 种, 旋转 180 度的置换同属 $(2)^6$ 形式的共轭类, 有 3 种, 分列如下

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 5 & 4 & 8 & 12 & 9 & 1 & 3 & 11 & 6 & 2 & 7 & 10 \end{pmatrix} = (1 \ 5 \ 9 \ 6)(2 \ 4 \ 12 \ 10)(3 \ 8 \ 11 \ 7)$$

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 6 & 10 & 7 & 2 & 1 & 9 & 11 & 3 & 5 & 12 & 8 & 4 \end{pmatrix} = (1 \ 6 \ 9 \ 5)(2 \ 10 \ 12 \ 4)(3 \ 7 \ 11 \ 8)$$

$$\begin{aligned}
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 2 \ 3 \ 4)(5 \ 6 \ 7 \ 8)(9 \ 10 \ 11 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 2 & 3 & 4 & 1 & 6 & 7 & 8 & 5 & 10 & 11 & 12 & 9 \end{array} \right) = (1 \ 4 \ 3 \ 2)(5 \ 8 \ 7 \ 6)(9 \ 12 \ 11 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 4 \ 3 \ 2)(5 \ 8 \ 7 \ 6)(9 \ 12 \ 11 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 3 \ 11 \ 9)(2 \ 7 \ 10 \ 6)(4 \ 8 \ 12 \ 5) \\
 & \left(\begin{array}{cccccccccccc} 3 & 7 & 11 & 8 & 4 & 2 & 10 & 12 & 1 & 6 & 9 & 5 \end{array} \right) = (1 \ 3 \ 11 \ 9)(2 \ 7 \ 10 \ 6)(4 \ 8 \ 12 \ 5) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 9 \ 11 \ 3)(2 \ 6 \ 10 \ 7)(4 \ 5 \ 12 \ 8) \\
 & \left(\begin{array}{cccccccccccc} 9 & 6 & 1 & 5 & 12 & 10 & 2 & 4 & 11 & 7 & 3 & 8 \end{array} \right) = (1 \ 9 \ 11 \ 3)(2 \ 6 \ 10 \ 7)(4 \ 5 \ 12 \ 8) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 9)(2 \ 12)(3 \ 11)(4 \ 10)(5 \ 6)(7 \ 8) \\
 & \left(\begin{array}{cccccccccccc} 3 & 4 & 1 & 2 & 7 & 8 & 5 & 6 & 11 & 12 & 9 & 10 \end{array} \right) = (1 \ 3)(2 \ 4)(5 \ 7)(6 \ 8)(9 \ 11)(10 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 11)(2 \ 10)(3 \ 9)(4 \ 12)(5 \ 8)(6 \ 7)
 \end{aligned}$$

(2) 以正方体的空间对角线, 即对角顶点连线为轴, 旋转±120度构成的置换, 因为有4组对角顶点, 故有4种轴线的选择, 共8种置换, 同属 $(3)^4$ 形式的共轭类, 为

$$\begin{aligned}
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 2 \ 6)(3 \ 10 \ 5)(4 \ 7 \ 9)(8 \ 11 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 2 & 6 & 10 & 7 & 3 & 1 & 9 & 11 & 4 & 5 & 12 & 8 \end{array} \right) = (1 \ 6 \ 2)(3 \ 5 \ 10)(4 \ 9 \ 7)(8 \ 12 \ 11) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 6 \ 2)(3 \ 5 \ 10)(4 \ 9 \ 7)(8 \ 12 \ 11) \\
 & \left(\begin{array}{cccccccccccc} 6 & 1 & 5 & 9 & 10 & 2 & 4 & 12 & 7 & 3 & 8 & 11 \end{array} \right) = (1 \ 12 \ 7)(2 \ 5 \ 11)(3 \ 4 \ 8)(6 \ 9 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 12 \ 7)(2 \ 5 \ 11)(3 \ 4 \ 8)(6 \ 9 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 12 & 5 & 4 & 8 & 11 & 9 & 1 & 3 & 10 & 6 & 2 & 7 \end{array} \right) = (1 \ 7 \ 12)(2 \ 11 \ 5)(3 \ 8 \ 4)(6 \ 10 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 7 \ 12)(2 \ 11 \ 5)(3 \ 8 \ 4)(6 \ 10 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 7 & 11 & 8 & 3 & 2 & 10 & 12 & 4 & 6 & 9 & 5 & 1 \end{array} \right) = (1 \ 5 \ 4)(2 \ 9 \ 8)(3 \ 6 \ 12)(7 \ 10 \ 11) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 5 \ 4)(2 \ 9 \ 8)(3 \ 6 \ 12)(7 \ 10 \ 11) \\
 & \left(\begin{array}{cccccccccccc} 5 & 9 & 6 & 1 & 4 & 12 & 10 & 2 & 8 & 11 & 7 & 3 \end{array} \right) = (1 \ 4 \ 5)(2 \ 8 \ 9)(3 \ 12 \ 6)(7 \ 11 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 4 \ 5)(2 \ 8 \ 9)(3 \ 12 \ 6)(7 \ 11 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 4 & 8 & 12 & 5 & 1 & 3 & 11 & 9 & 2 & 7 & 10 & 6 \end{array} \right) = (1 \ 8 \ 10)(2 \ 3 \ 7)(4 \ 11 \ 6)(5 \ 12 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1 \ 8 \ 10)(2 \ 3 \ 7)(4 \ 11 \ 6)(5 \ 12 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 8 & 3 & 7 & 11 & 12 & 4 & 2 & 10 & 5 & 1 & 6 & 9 \end{array} \right) = (1 \ 10 \ 8)(2 \ 7 \ 3)(4 \ 6 \ 11)(5 \ 9 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 10 & 7 & 2 & 6 & 9 & 11 & 3 & 1 & 12 & 8 & 4 & 5 \end{array} \right) = (1 \ 10 \ 8)(2 \ 7 \ 3)(4 \ 6 \ 11)(5 \ 9 \ 12)
 \end{aligned}$$

(4) 以正方体的对棱中心连线为轴线, 旋转180度构成的置换, 因为有6组对棱, 故有6种轴线的选择, 共6种置换, 同属 $(1)^2(2)^5$ 形式的共轭类, 为

$$\begin{aligned}
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (1)(11)(2 \ 5)(3 \ 9)(4 \ 6)(7 \ 12)(8 \ 10) \\
 & \left(\begin{array}{cccccccccccc} 1 & 5 & 9 & 6 & 2 & 4 & 12 & 10 & 3 & 8 & 11 & 7 \end{array} \right) = (3)(9)(1 \ 11)(2 \ 8)(4 \ 7)(5 \ 10)(6 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (3)(9)(1 \ 11)(2 \ 8)(4 \ 7)(5 \ 10)(6 \ 12) \\
 & \left(\begin{array}{cccccccccccc} 11 & 8 & 3 & 7 & 10 & 12 & 4 & 2 & 9 & 5 & 1 & 6 \end{array} \right) = (2)(12)(1 \ 7)(3 \ 6)(4 \ 10)(5 \ 11)(8 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (2)(12)(1 \ 7)(3 \ 6)(4 \ 10)(5 \ 11)(8 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 7 & 2 & 6 & 10 & 11 & 3 & 1 & 9 & 8 & 4 & 5 & 12 \end{array} \right) = (4)(10)(1 \ 8)(2 \ 12)(3 \ 5)(6 \ 11)(7 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (4)(10)(1 \ 8)(2 \ 12)(3 \ 5)(6 \ 11)(7 \ 9) \\
 & \left(\begin{array}{cccccccccccc} 8 & 12 & 5 & 4 & 3 & 11 & 9 & 1 & 7 & 10 & 6 & 2 \end{array} \right) = (5)(7)(1 \ 12)(2 \ 11)(3 \ 10)(4 \ 9)(6 \ 8) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (5)(7)(1 \ 12)(2 \ 11)(3 \ 10)(4 \ 9)(6 \ 8) \\
 & \left(\begin{array}{cccccccccccc} 12 & 11 & 10 & 9 & 5 & 8 & 7 & 6 & 4 & 3 & 2 & 1 \end{array} \right) = (6)(8)(1 \ 10)(2 \ 9)(3 \ 12)(4 \ 11)(5 \ 7) \\
 & \left(\begin{array}{cccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \end{array} \right) = (6)(8)(1 \ 10)(2 \ 9)(3 \ 12)(4 \ 11)(5 \ 7)
 \end{aligned}$$

根据Burnside引理, 确定不同的正方体搭建方案数量, 需要确定搭建方案在各种置换下的不动点数量。也就是说, 可以先确定利用给定颜色的木棒能够得到的不同排列, 然后使用一种固定的顺序(比如从前到后的顺序)来使用这些木棒搭建正方体, 得到的就是不考虑“旋转后相同”这个条件时所有不同的正方体搭建方案数。接着确定这些搭建方案在对应的置换下是否仍然保持不变, 如果不变, 那么这种搭建方案在对应的置换下就是一个不动点。确定给定的木棒能够得到的不同排列可以使用回溯法获得(题目限定只有6种颜色,

可以直接使用算法库函数 `next_permutation` 来生成所有的不同排列)。由于有 24 种置换, 最后不同的正方体搭建方案为不动点总数除以 24 所得到的值。

除了使用上述较为“原始”的方法计算不动点的数量, 还存在更为“巧妙”的方法来得到不动点数量, 其关键就是利用置换的轮换表示。根据置换的轮换分解定理, 置换可以表示为若干个不相交轮换的乘积, 对于每个轮换来说, 不难理解存在以下的事实: 只有轮换中的元素均使用相同的颜色才可能使得染色方案为不动点。假设某个置换的轮换数为 n , 确定在此置换下的不动点数量, 等价于求解以下组合数学问题: 使用 1 到 6 共六种颜色, 每种颜色可用次数为 c_i 次 ($i=1, 2, \dots, 6$), 为 n 个盒子中的球染色, 每个盒子包含 m_j 个球 ($j=1, 2, \dots, n$), 求所能得到的不同染色方案数量。对于本题来说, 要求同一个盒子内的球必须染成相同颜色, 且存在约束

$$\sum_{i=1}^6 c_i = \sum_{j=1}^n m_j$$

因此组合计算相对简化。以面心连线为轴旋转 ± 90 度为例, 此时的置换是(4)³ 形式的共轭类, 相当于为 3 个盒子染色, 每个盒子包含 4 个球, 不难推出, c_i 必须为 4 的倍数, 否则将无法满足同一个盒子内的球染色相同的要求, 对于其他类型的置换也有相似的结论。令给定置换的轮换数为 x , 每个轮换的长度 k , 可以将问题转换为以下组合数学问题: 使用 1 到 6 共六种颜色, 每种颜色可用次数为 c_i/k 次 ($i=1, 2, \dots, 6$), 为 x 个物品染色, 求所能得到的不同染色方案数量, 其中 c_i 能够被 k 整除。此处有一个例外需要处理, 即以对棱中心为轴旋转 180 度的置换, 它属于(1)²(2)⁵ 形式的共轭类, 此时可以先将两条置换前后不变的棱先染色, 然后将剩余的颜色分成 5 组, 即可同法处理。

参考代码

```
int Cnk[16][16] = {0}, rods1[6], rods2[6];

long long work(int k)
{
    // 统计物品数和颜色数。
    int x = 0;
    for (int i = 0; i < 6; i++) {
        if (rods2[i] % k) return 0;
        rods2[i] /= k;
        x += rods2[i];
    }
    // 根据乘法原理计数染色方案数。
    long long r = 1;
    for (int i = 0; i < 6; i++) {
        r *= Cnk[x][rods2[i]];
        x -= rods2[i];
    }
    return r;
}

int main(int argc, char *argv[])
{
    // 预处理组合数, 计算从 n 个物品中挑选 k 个物品的不同取法。
    for (int i = 0; i <= 12; i++) {
        Cnk[i][0] = Cnk[i][i] = 1;
        for (int j = 1; j < i; j++)
            Cnk[i][j] = Cnk[i - 1][j] + Cnk[i - 1][j - 1];
    }
}
```

```

}

// 处理数据。
int cases = 0;
cin >> cases;
for (int cs = 1; cs <= cases; cs++) {
    memset(rods1, 0, sizeof(rods1));
    for (int i = 0, color; i < 12; i++) {
        cin >> color;
        rods1[--color]++;
    }
    long long r = 0;
    // 静止不动的置换有 1 种。
    memcpy(rods2, rods1, sizeof(rods1));
    r += work(1);
    // 以相对面心的连线为轴旋转  $\pm 90$  度, 有  $3 \times 2 = 6$  种置换。
    memcpy(rods2, rods1, sizeof(rods1));
    r += 3 * 2 * work(4);
    // 以相对面心连线为轴旋转  $180$  度, 有  $3 \times 1 = 3$  种置换。
    memcpy(rods2, rods1, sizeof(rods1));
    r += 3 * work(2);
    // 以空间对角线为轴旋转  $\pm 120$  度, 有  $4 \times 2 = 8$  种置换。
    memcpy(rods2, rods1, sizeof(rods1));
    r += 4 * 2 * work(3);
    // 以对棱中心为轴旋转  $180$  度的置换, 有  $6 \times 1 = 6$  种置换。
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < 6; j++) {
            memcpy(rods2, rods1, sizeof(rods1));
            // 预先为两条置换前后不变的棱分配颜色, 剩余的分成 5 组进行染色, 每组包含 2 条棱。
            rods2[i]--, rods2[j]--;
            if (rods2[i] < 0 || rods2[j] < 0) continue;
            r += 6 * work(2);
        }
    }
    // 应用 Burnside 引理。
    cout << r / 24 << '\n';
}
return 0;
}

```

强化练习：10294* Arif in Dhaka^D，10733* The Colored Cubes^C，11255 Necklace^D。

扩展练习：11540 Sultan's Chandelier^E。

6.4 鸽笼原理

鸽笼原理 (pigeonhole principle)，又称狄利克雷抽屉原理 (Dirichlet drawer principle)^I，该原理的简单形式可以表述为：将 n 只鸽子放进 m 个笼子中， $n > m \geq 1$ ，则至少有一个笼子会包含至少两只鸽子。可以用反证法证明鸽笼原理。假设每个笼子内最多有一只鸽子，则鸽子数量最多为 m ，但这与鸽子数量为 n 且 $n > m$ 的条件相矛盾，故至少有一只笼子有至少两只鸽子。注意原理中的提法，是“至少有一个笼子会包

^I 约翰·彼得·古斯塔夫·勒热纳·狄利克雷 (Johann Peter Gustav Lejeune Dirichlet, 1805—1859)，德国数学家，解析数论的奠基人，第一个给出“函数”概念的现代正式定义，对傅里叶级数 (Fourier series) 等若干解析数学中的课题也有重要贡献。

含至少两只鸽子”，而不是“至少有一个笼子包含两只鸽子”，例如有 10 只鸽子，放入 3 个笼子中，**不管如何放置，至少有一个笼子中的鸽子数量大于等于 3**，符合“至少有一个笼子会包含至少两只鸽子”提法，但不符合“至少有一个笼子包含两只鸽子”的提法，**因为按照 3、3、4 的方法放置，没有一个笼子包含两只鸽子**。

该原理看似简单，但其所包含的思想却非常深刻，常常有意想不到的应用。举个例子，试证明以下命题：从 1 到 $2N$ 的自然数中任意取出 $N+1$ 个不同的自然数，取出的自然数中必定至少有两个数互素^I。初看似乎难以入手，但是利用鸽笼原理可以巧妙地予以解决。不妨将 $2N$ 个数从小到大排成一列，每两个相邻的自然数视为位于同一个盒子中，那么总共可以看成 N 个盒子，其中第一个盒子放有 1 和 2，第二盒子放有 3 和 4，…，第 N 个盒子放有 $2N-1$ 和 $2N$ ，现在从这 N 个盒子中任意取出 $N+1$ 个数，那么一定有一个盒子中的两个数都被取了出来，而盒子里的数是相邻的自然数，它们一定是互素的，所以，取出来的 $N+1$ 个自然数必定有一对数互素^{II}。

鸽笼原理的简单形式实际上是其一般形式的一个特例，一般形式可以表述为：设 p_1, p_2, \dots, p_n 为正整数，将 $p_1+p_2+\dots+p_n-n+1$ 只鸽子放入 n 个笼子中，要么第 1 个笼子包含至少 p_1 只鸽子，要么第 2 个笼子包含至少 p_2 只鸽子，…，要么第 n 个笼子包含至少 p_n 只鸽子。同样的，可以使用反证法来证明其一般形式。假设结论不成立，那么所有笼子的鸽子数最多为

$$(p_1 - 1) + (p_2 - 1) + \dots + (p_n - 1) = p_1 + p_2 + \dots + p_n - n$$

而总共鸽子数为 $(p_1 + p_2 + \dots + p_n - n) + 1$ ，产生矛盾，因此可以得知至少有一个笼子包含至少 p_i ($1 \leq i \leq n$) 只鸽子。

需要注意的是，鸽笼原理只能够解决某些问题解决方案的存在性问题，并不能给出具体的解决方案。在解题中应用鸽笼原理，关键的步骤是需要确定鸽子（物体或对象）、鸽笼（期望特征的种类）以及可计算的鸽子和笼子数量^[49]。

11237 Halloween Treats^D（万圣节糖果）

每年的万圣节都存在同样的问题：邻居们只愿意把一定总量的糖果给来访的孩子们，而不管来访的小孩的数量，这样来得晚的小孩可能得不到糖果。为了避免这样的情况，孩子们决定将糖果全部集中然后再平分。根据去年万圣节的经验，孩子们知道每位邻居会给多少糖果，因为孩子们更关心公平而不是糖果的数量，因此他们想从所有的邻居中选出一部分来进行拜访，然后平分从这些邻居手中获得的糖果。孩子们要求至少都能够平分到 1 颗糖果，如果糖果的数量不能完全被平分，孩子们会不高兴。你的任务是帮助孩子们找到满意的拜访方案。

输入

输入包含多组测试数据。测试数据的第一行包含两个整数 c 和 n ($1 \leq c \leq n \leq 100000$)，表示孩子的数量和邻居的数量，下一行是以空格分隔的 n 个整数 a_1, a_2, \dots, a_n ($1 \leq a_i \leq 100000$)， a_i 表示孩子拜访序号为 i 的邻居时所能获得的糖果数量。输入以两个 0 结束。

^I 两个数互素是指它们之间的最大公约数为 1，例如 9 和 20 互素。

^{II} 将相邻的自然数看成是放在一个盒子中，是为了描述证明的方便，并不影响证明的正确性。也就是说，从 $2N$ 个连续整数中抽取 $N+1$ 个整数，必定会抽取到两个相邻的整数，因为为了保证已抽取的数不相邻，只能间隔抽取，而这样至多只能抽取 N 个。

输出

对于每组测试数据，输出一行，给出能够满足题意要求的邻居序号（序号 i 表示第 i 个邻居，他将给孩子们 a_i 颗糖果）。假如没有方案能够满足题意要求，输出“no sweets”。如果有多种方案能够满足要求，输出任意一种即可。

样例输入

```
4 5
1 2 3 7 5
0 0
```

样例输出

```
3 5
```

分析

构建前缀和数组 s ，数组元素 s_i 保存的是糖果数量序列 a_1, a_2, \dots, a_i 的和，根据题意，有 $1 \leq s_i < s_{i+1}$ ， $1 \leq i$ 。前缀和数组 s 共有 n 个元素，将前缀和 s_i 逐个取 c 的模，其结果只可能是 $0, 1, \dots, c-1$ 共 c 种，题意给出 $c \leq n$ ，根据鸽笼原理，以下两种情形至少有一种成立：(1) s 中至少有一个元素模 c 为 0 ；(2) s 中至少有两个元素模 c 的值相同。如果 s 中的某个元素 s_i 模 c 的值为 0 ，很显然， a_1, a_2, \dots, a_i 的和是 c 的倍数。如果 s 中两个元素模 c 的值相同，不妨令其为 s_i 和 s_j ，那么 a_{i+1}, \dots, a_j 的和肯定为 c 的倍数（令 $s_i = xc + r, s_j = yc + r$ ，有 $s_j - s_i = (y - x)c$ ，为 c 的倍数）。由于条件 $n \geq c$ ，每组数据都能够找到满足题意要求的方案。由于不需要输出糖果的数量，只需要输出序号，前缀和数组 s 保存模 c 的值即可。可以使用辅助数组记录模 c 的值第一次出现时的序号以便判断余数是否重复。注意余数为 0 时的处理，如果余数为 0 ，表明从第一项开始到当前项的和满足题意要求。

强化练习：10620 A Flea on a Chessboard^C，11898 Killer Problem^D，12036 Stable Grid^C。

6.4.1 拉姆齐理论

拉姆齐理论（Ramsey theory）起始于 20 世纪二三十年代，最初由英国数学家拉姆齐^I提出。该理论中的拉姆齐定理是鸽笼原理一个重要推广，也称为广义鸽笼原理。拉姆齐定理的完整表述稍显复杂，不过可以通过相对简单的等价提法来获得初步的了解，它们也是拉姆齐问题最基本、最特殊的形式。其中较为直观的特例为：假如有 6 个（或 6 个以上的）人，一定可以找到其中的 3 个人，使得这 3 个人要么彼此之间互不相识，要么彼此之间互相认识。如果使用图论的方式来表述，则可以表示成：如果将 n 个顶点中的每一对顶点之间使用一条边连接，就得到了一个 n 顶点的完全图，记为 K_n ，当 $n \geq 6$ 时，如果用红、蓝两种颜色给 K_n 的边染色，使得每条边染上一种颜色，由于 K_n 有 $C(n, 2)$ 条边，则不同的染色方案数为 $2^{C(n, 2)}$ ，在所有的染色方案中，至少存在一种染色方案，在该方案中，具有一个单色的三角形，即有一个三角形的三条边染色相同。数字 6 是使得上述结论能够成立的最小正整数，对于小于 6 的正整数，都可以找出一种方案，使得单色三角形不存在。概括起来就是说，对 K_6 的边进行使用红（或蓝）染色，则要么存在一个红色三角形或者要么存在一个蓝色三角形。

进一步地，可以定义拉姆齐数：对于任意给定的两个正整数 p 和 q ，如果存在最小的正整数 $r(p, q)$ ，使得当 $n \geq r(p, q)$ 时，对 K_n 任意的红蓝二色边染色， K_n 中存在红色的 K_p 或蓝色的 K_q ，则称 $r(p, q)$ 为拉姆齐

^I 弗兰克·普兰顿·拉姆齐（Frank Plumpton Ramsey，1903—1930），英国哲学家、数学家、经济学家，去世时年仅 26 岁，但他在短暂的一生中对许多领域做出了开创性的贡献。

数。拉姆齐理论就是研究拉姆齐数相关性质的理论，拉姆齐于1930年证明了拉姆齐数 $r(p, q)$ 的存在性。有关拉姆齐数的性质、拉姆齐定理的进一步介绍，感兴趣的读者可以参考组合数学或离散数学教材中的相关内容。

6.5 容斥原理

容斥原理 (inclusion-exclusion principle) 是计数理论中的一个基本原理。计数要求既没有重复也没有遗漏。在计数时，先将若干集合中的所有对象数量计算出来，然后减去重复计算的对象数量，这样就能够保证最后得到的计数结果既无重复也无遗漏。假如有两个有限集 A 和 B ，则容斥原理可以表示为

$$|A \cup B| = |A| + |B| - |A \cap B|$$

类似的，三个有限集 A, B, C 的容斥原理可以表示为

$$|A \cup B \cup C| = |A| + |B| + |C| - |A \cap B| - |A \cap C| - |B \cap C| + |A \cap B \cap C|$$

更一般的，设 A_i 是有限集， $i=1, 2, \dots, n$ ($n \geq 2$)，容斥原理可以表示为

$$\left| \bigcup_{i=1}^n A_i \right| = \sum_{i=1}^n |A_i| - \sum_{1 \leq i < j \leq n} |A_i \cap A_j| + \sum_{1 \leq i < j < k \leq n} |A_i \cap A_j \cap A_k| - \dots + (-1)^{n+1} |A_1 \cap \dots \cap A_n|$$

观察计算公式的各个子项，子项中进行运算的集合个数为奇数时相加，为偶数时则相减，故可根据子项中集合个数的二进制表示中 1 的数量来确定各个子项的符号。

10325 The Lottery^B (抽奖)

孟加拉国运动委员会 (The Sports Association of Bangladesh) 因为最近举行的“Jodi Laiga Jai”抽奖而遇到了大麻烦。由于参与者众多以致该委员会无法处理所有的彩票号码。经过紧急会商后，委员会决定弃用一部分彩票号码。但是他们应该如何选择需要被弃用的彩票号码呢？Mr. Nondo Dusal 对于历史问题非常感兴趣，他提出了一个方案来解决上述问题（你可能会好奇他是如何想到这个解决方案的，答案在于他最近阅读了有关约瑟夫问题的一些资料）。总共有 N 张彩票，编号为 1 到 N ，Mr. Nondo 会随机选择 M 个正整数，然后从 1 到 N 的号码中挑出能够被这 M 个正整数中某个数整除的号码，剩余的不能被 M 个正整数中任意一个整数所整除的号码将予以保留用于抽奖。正如你所知，任意整数能够被 1 所整除，因此 Mr. Nondo 不会选择整数 1 作为 M 个数中的一个数。现在给定 N ， M 和 M 个随机选择的整数，你需要确定能够用于抽奖的彩票数量。

输入

每组测试数据起始一行包含两个整数 N ($10 \leq N < 2^{31}$) 和 M ($1 \leq M \leq 15$)，接着一行包含 M 个整数，这些整数均不大于 N 。输入以文件结束符作为结束标记。

输出

对于每组测试数据输出一个整数 R ，表示在 1 到 N 这 N 个整数中有 R 个数不能被 M 个数中的任意一个数整除。

样例输入

```
10 2
2 3
20 2
2 4
```

样例输出

```
3
10
```

分析

题目描述可以概述为以下计数问题：给定正整数 N 和 M 个大于 1 且小于 N 的不同整数，统计从 1 到 N 这 N 个整数中有多少个数不能被 M 个数中的任意一个数所整除。直接计数符合要求的数存在困难，可以使用排除法进行计数。令 $U = \{x \mid 1 \leq x \leq N\}$ ， A_i 表示 U 中能被第 i 个整数整除的数构成的集合，则题意所求为 $|U| - |A_1 \cup A_2 \cup \dots \cup A_M|$ 。由于 1 到 N 中能被 x 整除的数的个数为 $\lfloor N/x \rfloor$ ，结合容斥原理可以计算 $|A_1 \cup A_2 \cup \dots \cup A_M|$ 。

参考代码

```
int main(int argc, char *argv[]) {
    long long N, M, numbers[16];
    while (cin >> N >> M) {
        for (int i = 0; i < M; i++) cin >> numbers[i];
        long long cnt = 0;
        for (int i = 1; i < (1 << M); i++) {
            long long lcm = 1;
            for (int j = 0; j < M; j++)
                if (i & (1 << j)) {
                    // __gcd 是 GCC 内置函数，用于计算两个数的最大公约数。
                    lcm = lcm / __gcd(lcm, numbers[j]) * numbers[j];
                    if (lcm > N) break;
                }
            // __builtin_popcount 是 GCC 内置函数，其作用是计数给定整数的二进制表示中
            // 位为 1 的个数，而通过此数可以确定在展开式中子项的符号。
            cnt += (N / lcm) * ((__builtin_popcount(i) % 2) ? 1 : (-1));
        }
        cout << (N - cnt) << '\n';
    }
    return 0;
}
```

扩展练习：[1047 Zones^D](#)，[10882 Koerner's Pub^D](#)，[11246 K-Multiple Free Set^C](#)，[11806* Cheerleaders^D](#)。

6.5.1 错排问题

设有一排共 n 个方格，从左至右依次标以 $1, 2, \dots, n$ ， $S = \{1, 2, \dots, n\}$ 是一个集合，对于 S 的一个全排列，总是可以对照方格，检查每个元素是否在相应的位置上。如果 S 的一个全排列中没有一个元素在相应的位置上，则称该全排列为一个错位排列（derangement）。求所有的错位排列的数量问题就是错位排列问题。

例如，排列 21453 是 12345 的错位排列，而 21543 不是 12345 的错位排列，因为 4 在原来的位置上。 n 个物体的错位排列数记为 D_n ，可以利用容斥原理求出 D_n 的通项公式

$$D_n = n! \left(1 - \frac{1}{1!} + \frac{1}{2!} - \frac{1}{3!} + \dots + (-1)^n \frac{1}{n!} \right), \quad n \geq 1$$

也可由递推关系表示为

$$D_n = (n-1)(D_{n-1} + D_{n-2}), \quad n \geq 3, \quad D_1 = 0, \quad D_2 = 1$$

或者

$$D_n = nD_{n-1} + (-1)^n, \quad n \geq 2, \quad D_1 = 0$$

根据递推关系，可以容易的得到 D_n ($n \geq 1$) 的前若干项为：0, 1, 2, 9, 44, 265, 1854, 14833, 133496, \dots 。

强化练习: 10497 Sweet Child Makes Trouble^C, 11282 Mixing Invitations^C, 12024 Hats^C。

6.6 初等数列

6.6.1 等差数列

等差数列 (arithmetic progression), 或称算术数列, 其首项为 a_1 , 从第二项开始, 每一项与前一项的差值为一个固定的常数 d , 即 $a_n - a_{n-1} = d$, 其通项公式为

$$a_n = a_1 + (n-1)d, \quad n \geq 2$$

前 n 项和为

$$S_n = \frac{n(a_1 + a_n)}{2} = \frac{dn^2 + (2a_1 - d)n}{2}$$

强化练习: 326 Extrapolation Using a Difference Table^B, 10025 The ? 1 ? 2 ? ... ? n = k Problem^A, 10079 Pizza Cutting^A, 10790 How Many Points of Intersection^A, 11254 Consecutive Integers^B, 12751 An Interesting Game^B, 12918 Lucky Thief^C, 13161 Candle Box^D。

扩展练习: 12004 Bubble Sort^C, 12908 The Book Thief^D。

6.6.2 等比数列

等比数列 (geometric progression), 或称几何数列, 其首项为 a_1 , 从第二项开始, 每一项与前一项的比值为一个固定的常数 q , 即 $a_n/a_{n-1} = q$, 其中 $n \geq 2$, $a_{n-1} \neq 0$, $q \neq 0$ 。其通项公式为

$$a_n = q^{n-1}a_1, \quad n \geq 2, \quad q \neq 0$$

前 n 项和为

$$S_n = \begin{cases} na_1 & q = 1 \\ \frac{a_1(1 - q^n)}{1 - q} & q \neq 1 \end{cases}$$

6.6.3 其他数列

其他常见的还有平方数数列和立方数数列。平方数数列是由从 1 开始的平方数构成的数列:

$$a_n = n^2, \quad n \geq 1$$

其前 n 项和为

$$S_n = \sum_{k=1}^n k^2 = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6} = \frac{n(n+1)(2n+1)}{6}$$

立方数数列是由从 1 开始的立方数构成的数列, 即

$$a_n = n^3, \quad n \geq 1$$

其前 n 项和为

$$S_n = \sum_{k=1}^n k^3 = (1 + 2 + \dots + n)^2 = \frac{n^4 + 2n^3 + n^2}{4} = \left[\frac{n(n+1)}{2} \right]^2$$

强化练习: 413 Up and Down Sequences^B, 10014 Simple Calculations^A, 10302 Summation of Polynomials^A, 12149 Feynman^A。

扩展练习: 13096 Standard Deviation^D。

6.7 计数序列

6.7.1 斐波那契数

斐波那契数 (Fibonacci number) 最初由斐波那契^I发现, 因此得名。斐波那契在研究兔子的生育过程中提出以下问题: 假设一对兔子中的雌性兔子每月生育一对雌雄各异的后代, 后代经过一个月的发育, 互相交配后每月也可生育一对雌雄各异的后代, 问一年后兔子的总对数。在第一个月内, 最初所给的一对兔子将生育一对兔子, 故在第一个月的月末, 总共有 2 对兔子; 第二个月, 唯有最初的一对兔子会生育后代, 故在第二个月末, 兔子总数为 3 对。在第三个月, 最初的一对兔子和第一个月生育的兔子均会生育一对后代, 故在第三个月末, 兔子总数为 5 对。按月份, 兔子的总对数为 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 一年后, 兔子的总数对 377 对。观察所形成的序列, 可以发现从第三项开始, 每项为前两项之和, 如果为序列增加两个初始值 $F_1=F_2=1$, 则构成了斐波那契序列。在现代数学中, 斐波那契数的首项常被定为 0, 因此其递推关系可以定义为

$$F_n = F_{n-1} + F_{n-2}, \quad n \geq 2, \quad F_0 = 0, \quad F_1 = 1$$

令人惊奇的是, 全部为整数的斐波那契数的通项公式可以用无理数表示为

$$F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right), \quad n \geq 0$$

而且数列的后一项与前一项的比值无限接近黄金分割率 (Golden ratio)

$$\lim_{n \rightarrow \infty} \frac{F_{n+1}}{F_n} = \varphi = \frac{1+\sqrt{5}}{2} = 1.6180339887 \dots$$

在有关斐波那契数的问题中, 关键是根据题目的条件推出斐波那契数的递推关系, 然后问题解决就相对简单了。以下列出了一些与斐波那契数有关的计数问题^{II[50]}。

(1) 给定一个 n 级台阶, 规定在登上台阶的过程中, 每次只能向上走一级或者两级台阶, 那么登上第 n 级台阶的不同方法数为第 $n+1$ 项斐波那契数 ($n \geq 1$)。

(2) 将两块玻璃叠在一起, 光线从上方射入, 假设光线只会被两块玻璃的交界处和下层玻璃的底面所反射, 则光线穿过玻璃或被反射离开玻璃且改变方向的次数不超过 n 次的可能情形总数为第 $n+2$ 项斐波那契数 ($n \geq 0$)。

(3) 将正整数 n 表示成 1 或者 2 相加的形式, 例如 $3=1+1+1=1+2=2+1$, 在考虑加数顺序的情况下, 不同的表示方法数为第 $n+1$ 项斐波那契数 ($n \geq 1$)。

(4) 使用 1×2 的多米诺骨牌完美覆盖 $2 \times n$ 棋盘的方案数为第 $n+1$ 项斐波那契数 ($n \geq 1$)。使用 1×1 和 1×2 的多米诺骨牌完美覆盖 $1 \times n$ 棋盘的方案数为第 $n+1$ 项斐波那契数 ($n \geq 1$)。

(5) 沿着杨辉三角 (Yang Hui's triangle, 又名帕斯卡三角, Pascal's triangle), 的对角线, 从左向上的二项式系数之和等于斐波那契数。即对于 $n \geq 0$, 第 n 项斐波那契数 F_n 满足

$$F_n = \binom{n-1}{0} + \binom{n-2}{1} + \binom{n-3}{2} + \dots + \binom{n-k}{k-1}, \quad n \geq 0, \quad k = \left\lfloor \frac{n+1}{2} \right\rfloor$$

斐波那契数还具有许多有趣的性质, 以下列举若干常用的恒等式。

(1) 令 s_n 表示前 n 项斐波那契数的和, 有

^I 列昂纳多·斐波那契 (Leonardo Fibonacci, 1170—1240), 意大利数学家, 著有《Liber Abacci》(珠算原理) 一书。

^{II} 有大量关于斐波那契数以及它们在植物学、计算机科学、地理学、物理学以及其他领域应用的文献, 甚至有一个学术刊物《斐波那契季刊》(The Fibonacci Quarterly) 专门报道关于它们的研究。

$$s_n = F_0 + F_1 + \cdots + F_n = \sum_{i=0}^n F_i = F_{n+2} - 1, \quad n \geq 0$$

$$(2) \quad F_n^2 - F_{n+1}F_{n-1} = (-1)^{n-1}, \quad n \geq 1; \quad F_n^2 - F_{n+r}F_{n-r} = (-1)^{n-r}F_r^2, \quad n \geq 0, \quad n \geq r.$$

$$(3) \quad \gcd(F_m, F_n) = F_{\gcd(m, n)}; \quad \gcd(F_n, F_{n+1}) = \gcd(F_{n+1}, F_{n+2}) = \gcd(F_n, F_{n+2}) = 1.$$

$$(4) \quad F_{kn} \equiv 0 \pmod{F_n}, \quad n \geq 1, \quad k \geq 1.$$

由于斐波那契数增长很快，在解题中一般都需要应用高精度整数，所以熟悉高精度整数的加法或者 Java 中大整数类的使用非常有必要。以下给出使用 Java 的 BigInteger 类求斐波那契数的代码框架。

```
//-----6.7.1.1.java-----
import java.io.*;
import java.util.*;
import java.math.*;

public class Main
{
    public static void main(String args[]) throws IOException
    {
        BigInteger[] fibs = new BigInteger[10002];
        fibs[0] = new BigInteger("0");
        fibs[1] = new BigInteger("1");
        for(int i = 2; i <= 10000; i++)
            fibs[i] = fibs[i - 1].add(fibs[i - 2]);
        Scanner cin = new Scanner(System.in);
        while(cin.hasNext()) {
            int n = cin.nextInt();
            System.out.println(fibs[n]);
        }
    }
}
//-----6.7.1.1.java-----
```

强化练习: 900 Brick Wall Patterns^A, 1646 Edge Case^E, 10334 Ray Through Glasses^A, 10450 World Cup Noise^A, 10579 Fibonacci Numbers^A, 11000 Bee^A, 11385 Da Vinci Code^B, 11582 Colossal Fibonacci Numbers^D, 12459 Bees' Ancestors^A, 12620 Fibonacci Sum^D。

扩展练习: 10236^I The Fibonacci Primes^D, 10862 Connect the Cable Wires^B, 11161 Help My Brother (II)^C。

矩阵快速幂

为了便于应用，斐波那契数的递推关系也可以表示成矩阵的形式

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F_{n-1} \\ F_{n-2} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \quad n \geq 2$$

而且还具有以下结论

$$\begin{bmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{bmatrix} \equiv \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^{n-1} \pmod{m}, \quad n \geq 2, \quad m > 0$$

利用以上结论，使用矩阵快速幂技巧可以方便的求出某个斐波那契数模 m 的值。

^I 10236 The Fibonacci Primes。结论：第 i 项 Fibonacci 数为 Fibonacci 素数，当且仅当 i 为素数（除 $F_4=3$ 例外）。

```

//-----6.7.1.2.cpp-----//
// 表示矩阵的结构体, one 为系数矩阵。
struct matrix {
    long long cell[2][2];
    matrix(long long a = 0, long long b = 0, long long c = 0, long long d = 0) {
        cell[0][0] = a, cell[0][1] = b, cell[1][0] = c, cell[1][1] = d;
    }
} one(1, 1, 1, 0);

// 模。
int mod;

// 矩阵相乘。
matrix multiply(const matrix &a, const matrix &b)
{
    matrix r;
    for (int i = 0; i < 2; i++)
        for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++) {
                r.cell[i][j] += a.cell[i][k] * b.cell[k][j] % mod;
                r.cell[i][j] %= mod;
            }
    return r;
}

// 迭代形式的矩阵快速幂。
matrix matrixPow(long long k)
{
    // r 为结果矩阵, 初始值为单位矩阵。
    matrix r(1, 0, 1, 0);
    // cm 为斐波那契数的系数矩阵。
    matrix cm(1, 1, 1, 0);
    while (k) {
        if (k & 1) r = multiply(r, cm);
        cm = multiply(cm, cm);
        k >>= 1;
    }
    return r;
}

// 递归形式的矩阵快速幂。
matrix matrixPow(long long k)
{
    if (k == 1) return one;
    matrix r = matrixPow(k >> 1);
    r = multiply(r, r);
    if (k & 1) r = multiply(r, one);
    return r;
}
//-----6.7.1.2.cpp-----//

```

强化练习: [10229 Modular Fibonacci^A](#), [10518 How Many Calls^B](#), [10689 Yet Another Number Sequence^B](#)。

扩展练习: [10655 Contemplation Algebra^C](#), [10870 Recurrences^C](#), [12470 Tribonacci^C](#)。

斐波那契进制

可以定义一种进制，将十进制数使用斐波那契数来表示，称之为斐波那契进制。与二进制数类似，在斐波那契进制数中，每个数位上的数值只有‘0’或‘1’，数位对应的权值是斐波那契数。设斐波那契数列为

$$F_0 = 0, F_1 = 1, F_2 = 1, F_3 = 2, \dots, F_n = F_{n-1} + F_{n-2}, n \geq 2$$

则斐波那契进制数可以定义为

$$x_f = f_i f_{i-1} \cdots f_2 f_1 = f_i * F_j + f_{i-1} * F_{j-1} + \cdots + f_2 * F_3 + f_1 * F_2, f_i \in (0, 1), i \geq 1, j \geq 2$$

其中，任意两个相邻的数位不能都为1。例如

$$128_{10} = 90_{10} + 34_{10} + 5_{10} + 1_{10} = 1010001001_f$$

实际上，斐波那契进制是齐肯多夫定理 (Zeckendorf's theorem) 的一种应用形式。齐肯多夫^I证明，对于给定的任意正整数，可以将其表示成一个或者多个不相邻的斐波那契数之和。更为准确的表述是：存在正整数 $c_i \geq 2$ ，且 $c_{i+1} > c_i + 1$ ， $i \geq 0$ ，使得对于任意正整数 N ，有

$$N = \sum_{i=0}^k F_{c_i}$$

其中 F_n 表示第 n 项斐波那契数 ($n \geq 0$)。

在实际应用中，给定一个十进制数，可以很容易将其转换为斐波那契进制数，使用贪心算法进行转换即可。

```
//-----6.7.1.3.cpp-----
const int MAXF = 64;
long long fibs[MAXF] = {1, 2};

string getFinary(long long n)
{
    // 可以预先计算斐波那契数备用而不是每次生成时重复计算。
    for (int i = 2; i < MAXF; i++) fibs[i] = fibs[i - 1] + fibs[i - 2];
    bitset<64> finary(0);
    while (n) {
        for (int i = MAXF - 1; i >= 0; i--)
            if (n >= fibs[i]) {
                finary.set(i);
                n -= fibs[i];
                break;
            }
    }
    string f = finary.to_string();
    while (f.size() && f.front() == '0') f.erase(f.begin());
    return f;
}
//-----6.7.1.3.cpp-----//
```

强化练习：[763 Fibinary Numbers^B](#)，[948 Fibonaccimal Base^B](#)，[11089 Fi-Binary Number^C](#)，[12281 Hyper Box^C](#)。

^I 爱德华·齐肯多夫 (Edouard Zeckendorf, 1901—1983)，比利时人，医生、陆军军官、数学家，因为对斐波那契数性质的研究及证明齐肯多夫定理而闻名。

扩展练习：1258* Nowhere Money^D。

6.7.2 卡特兰数

卡特兰数（Catalan number）是组合学中经常出现在各种问题中的计数数列，以数学家卡特兰^I的名字来命名，其定义为

$$C_n = \frac{1}{n+1} \binom{2n}{n} = \frac{(2n)!}{(n+1)! n!} = \frac{1}{n+1} \prod_{k=1}^n \frac{n+k}{k}, \quad n \geq 1, \quad C_0 = 1$$

其递推关系式为

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k} = \frac{4n-2}{n+1} C_{n-1}, \quad n \geq 1, \quad C_0 = 1$$

根据公式，容易求得卡特兰数的前二十项为（ $n \geq 0$ ）：1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862, 16796, 58786, 208012, 742900, 2674440, 9694845, 35357670, 129644790, 477638700, 1767263190。可以看到，Catalan 数增长得很快（事实上它是以指数形式增长），所以涉及 Catalan 数的问题一般都需要使用大整数运算。

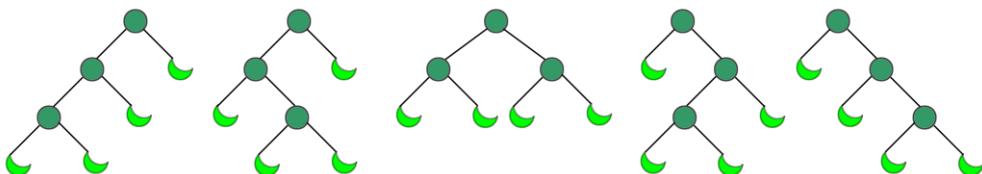
很多计数问题的结果都可以归结为卡特兰数^{[51][52]}，下面列举若干和卡特兰数相关的计数问题^{III[53]}。

- 在长度为 $n+1$ 的字符串中插入 n 对平衡括号的方法数（一对括号内包含的字符数至少为 2）。例如 $n=3$ 时，在长度为 4 的字符串中插入 3 对平衡括号，有 5 种不同的方法：((ab)(cd)), (((ab)c)d), ((a(bc))d), (a((bc)d)), (a(b(cd)))。
- 用 n 对括号可以构造出的平衡表达式的数量。最左边的括号 l 一定和某个右括号 r 配对，它们组合在一起把字符串划分成两个平衡的部分：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 已经用掉一对括号，而左右两个部分也必须是平衡的括号表达式。因此有以下递推关系式

$$C_n = \sum_{k=0}^{n-1} C_k C_{n-1-k}$$

这正是卡特兰数。

- 具有 $n+1$ 个叶结点的有根满二叉树计数。例如具有 4 个叶结点的有根满二叉树共有 5 种^{III}。

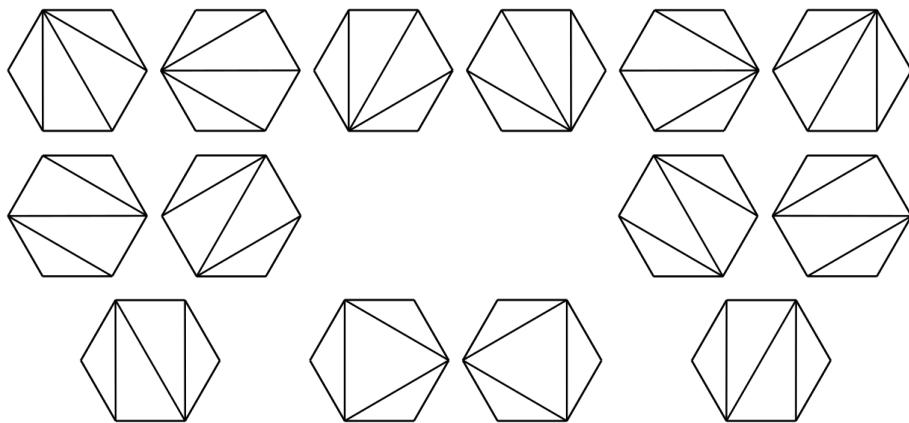


^I 欧仁·查尔斯·卡特兰 (Eugène Charles Catalan, 1814—1894)，比利时数学家。

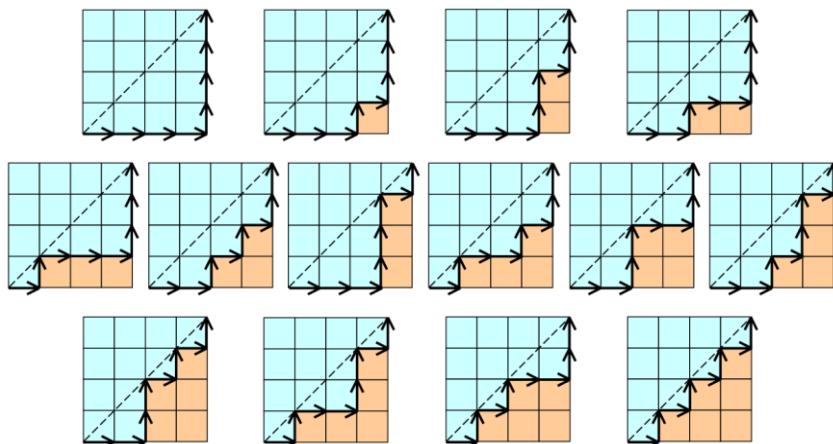
^{II} 在 Richard P. Stanley 的组合学著作《Enumerative Combinatorics》(卷 2) 第 6 章中，列举了与卡特兰数有关的问题。2013 年，Stanley 对 Catalan 数有关的问题做了补遗 (Catalan addendum)，在补遗中，增加了对卡特兰数的组合论、代数论含义阐释的示例，同时也增加了对 Motzkin 数和 Schröder 数进行阐释的示例，读者可以通过以下链接下载该补遗：<http://www-math.mit.edu/~rstan/ec/catadd.pdf>, 2020。

^{III} 图片来源：https://commons.wikimedia.org/wiki/File:Catalan_number_binary_tree_example.png。

- 具有 n 个结点的有根二叉树计数。
- 具有 $n+2$ 条边的凸多边形不同的三角剖分计数。例如正六边形的三角剖分计数为 14^{I} 。



- 在 $n \times n$ 的网格上，每次只能向右或向上走一格，在不穿越网格主对角线的情况下，从左下角 $(0, 0)$ 走到右上角 (n, n) 的不同路径计数^{II}。



除卡特兰数外，还有超卡特兰数（super Catalan number），超卡特兰数的第 n 项 ($n \geq 0$) 表示在长度为 $n+1$ 的字符串中插入平衡括号的方法数（一对括号内的字符数至少为 2，且括号的对数没有限制，但是将整个字符串括起来的括号忽略不计）。例如 $n=3$ 时，在长度为 4 的字符串中插入平衡括号，有 11 种不同的加括号方法：abcd, (ab)cd, a(bc)d, ab(cd), (abc)d, a(bcd), ((ab)c)d, (a(bc))d, (ab)(cd), a((bc)d), a(b(cd))。超卡特兰数的递推关系为

$$S_n = \frac{3(2n-1)S_{n-1} - (n-2)S_{n-2}}{n+1}, \quad n \geq 2, \quad S_0 = S_1 = 1$$

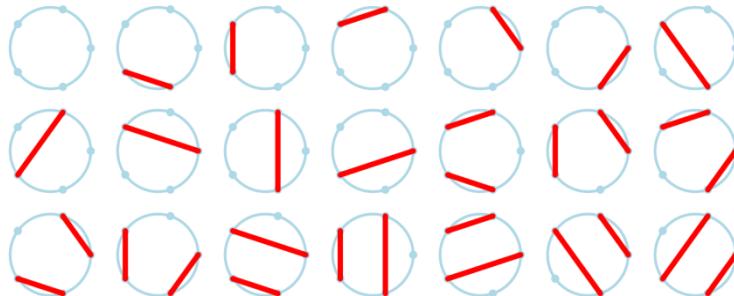
根据递推关系，容易得到此序列的前二十项 ($n \geq 0$)：1, 1, 3, 11, 45, 197, 903, 4279, 20793, 103049, 518859, 2646723, 13648869, 71039373, 372693519, 1968801519, 10463578353, 55909013009, 300159426963,

^I 图片来源：<https://commons.wikimedia.org/wiki/File:Catalan-Hexagons-example.svg>。

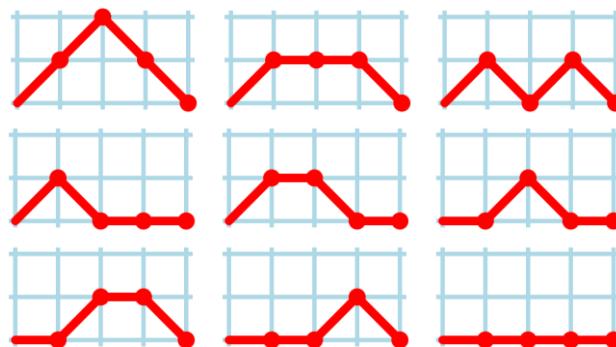
^{II} 图片来源：https://commons.wikimedia.org/wiki/File:Catalan_number_4x4_grid_example.svg。

1618362158587。

下面再介绍一个与卡特兰数相关的计数——莫提金数(Motzkin number)。莫提金数以数学家莫提金^I的名字命名, 它是在圆上的 n 个不同点之间画不相交弧的方法计数 (一条弧只需连接两个点, 不需要连接所有的点)。如下图所示^{II}, 在圆上的 5 个点之间画不相交的弧的方法共有 21 种。



莫提金数的第 n 项也是长度为 n 的莫提金路径的计数($n \geq 1$)。莫提金路径是指在矩形网格的右上上限, 从 $(0, 0)$ 走到 $(n, 0)$, 每次向右侧走一格 (可以选择向右、斜向上、斜向下), 且任意时刻不能位于横坐标轴 $y=0$ 之下的走法计数。如下图所示^{III}, 从 $(0, 0)$ 到 $(4, 0)$ 的莫提金路径计数为 9。



莫提金数具有以下递推形式

$$M_n = M_{n-1} + \sum_{k=0}^{n-2} M_k M_{n-2-k} = \frac{2n+1}{n+2} M_{n-1} + \frac{3n-3}{n+2} M_{n-2}, \quad n \geq 2$$

^I 西奥多·塞缪尔·莫提金 (Theodore Samuel Motzkin, 1908—1970), 美国籍以色列裔数学家, 出生于德国柏林, 其父亲为乌克兰籍犹太裔人。Motzkin 在孩童时期即显现出突出的数学天赋, 在其 15 岁时开始接受大学教育。1934 年, 就读于巴塞尔大学 (University of Basel), 师从 Alexander Ostrowski 并获得博士学位。1948 年, Motzkin 移居美国, 先后工作于哈佛大学 (Harvard College)、波士顿大学 (Boston College), 1950 年受聘于加州大学洛杉矶分校 (University of California at Los Angeles, UCLA), 后成为该校教授并一直工作到退休。

参阅: <http://www-history.mcs.st-andrews.ac.uk/Biographies/Motzkin.html>, 2020。

^{II} <https://commons.wikimedia.org/wiki/File:MotzkinChords5.svg>, 作者 Robertd, 版权协议 CC BY 3.0。此图片采用“知识共享 署名 3.0—未本地化版本”(CC BY 3.0) 许可协议进行许可, 若需查看该许可协议, 读者可以访问 <http://creativecommons.org/licenses/by/3.0/> 或者写信到 Creative Commons, PO Box 1866, Mountain View, CA 94042, USA。

^{III} <https://commons.wikimedia.org/wiki/File:Motzkin4.svg>, 作者 Robertd, 版权协议 CC BY 3.0。

莫提金数和卡特兰数具有以下的关系

$$M_n = \sum_{k=0}^{\lfloor n/2 \rfloor} \binom{n}{2k} C_k, \quad n > 0$$

容易求得莫提金数的前二十项为 ($n \geq 0$): 1, 1, 2, 4, 9, 21, 51, 127, 323, 835, 2188, 5798, 15511, 41835, 113634, 310572, 853467, 2356779, 6536382, 18199284。

10157 Expressions^C (括号表达式)

设 X 是所有合法括号表达式构成的集合, X 中的元素只由左括号 ‘(’ 和右括号 ‘)’ 字符构成。集合 X 的定义如下: (1) 空字符串属于 X ; (2) 如果 A 属于 X , 则 (A) 属于 X ; (3) 如果 A 和 B 属于 X , 则 AB 属于 X 。例如, 以下的括号表达式是合法的 (因此属于集合 X): $((())()$, $((())())$)。下列括号表达式是不合法的 (因此不属于集合 X): $((())(($, $(())()$)。

设 E 是一个合法的括号表达式 (因此 E 是一个属于集合 X 的字符串), 定义 E 的长度为字符串所包含的括号字符的数量, E 的深度 $D(E)$ 递归定义如下

$$D(E) = \begin{cases} 0; & \text{如果 } E \text{ 为空字符串} \\ D(A) + 1; & E = A, A \in X \\ \max(D(A), D(B)); & E = AB, A \in X, B \in X \end{cases}$$

例如, 括号表达式 “ $((())()$ ” 的长度为 8, 深度为 2。给定正整数 n 和 d , 确定长度为 n 深度为 d 的合法括号表达式的数量。

输入

输入包含多组测试数据。每组测试数据一行, 包含两个正整数 n 和 d , $2 \leq n \leq 300$, $1 \leq d \leq 150$ 。输入最多包含 20 组数据, 其中可能包含空行。

输出

对于输入中的每组数据, 输出一个整数, 表示长度为 n 深度为 d 的合法括号表达式的数量。

提示: 长度为 6 深度为 2 的合法括号表达式只有三个: $((())()$, $((())()$, $((())()$)。

样例输入

```
6 2
300 150
```

样例输出

```
3
1
```

分析

解题关键是获得递推关系, 可以参考 Catalan 数的推导过程以获得启发。

(1) 由于括号的数量必须是偶数才可能得到合法的表达式, 所以当 n 为奇数时, 合法表达式的数量为 0。

(2) 若深度为 d 超过 n 个括号所能得到的合法表达式最大深度时, 合法表达式的数量为 0。

(3) 当 $d=1$ 时, 合法的括号表达式只有一种。

(4) 设 $T(m, d)$ 表示括号对数为 m 深度不超过 d 的合法括号表达式的总数, E 是一个深度为 d , 括号对数为 m 的合法表达式, 则表达式 E 的最左边的括号 l 一定和某个右括号 r 配对, 他们合在一起把表达式划分为两个合法的括号表达式——在 l 和 r 之间的部分 X 以及 r 右边的部分 Y , 则 $E=(X)Y$, 设左边部分有 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, 则括号表达式 X 的深度最大为 d

-1 , 括号表达式 Y 的深度最大为 d 。则括号对数为 m , 深度为 d 的合法表达式数量为 $T(m, d) - T(m, d-1)$, 有

$$T(m, d) = \sum_{i=0}^{m-1} T(i, d-1)T(m-1-i, d)$$

强化练习: [991 Safe Salutations^B](#), [10223 How Many Nodes^A](#), [10303 How Many Trees^B](#), [10312 Expression Bracketing^C](#)。

扩展练习: [1478 Delta Wave^E](#), [10007 Count the Trees^A](#)。

6.7.3 欧拉数

欧拉数 (Eulerian number)^I 表示元素 $\{1, 2, \dots, n\}$ 的排列中, 恰好包含 k 次下降的排列个数, 用记号 $\langle n \rangle_k$ 来表示。 k 次下降的含义是假设 $\{1, 2, \dots, n\}$ 的某个排列为 $\{A_1, A_2, \dots, A_n\}$, 则该排列中恰好存在 k 处满足 $A_i < A_{i+1}$ ($1 \leq i \leq n-1$)。

欧拉数有以下递推关系

$$\langle n \rangle_k = (k+1) \langle n-1 \rangle_k + (n-k) \langle n-1 \rangle_{k-1}$$

其通项公式为

$$\langle n \rangle_k = \sum_{i=0}^{k+1} (-1)^i \binom{n+1}{i} (k+1-i)^n$$

6.7.4 斯特林数

斯特林数 (Stirling number) 分第一类斯特林数和第二类斯特林数。第一类斯特林数表示由 n 个元素组成的排列中, 恰好包含 k 个循环 (permutation cycle) 的排列个数, 通常用 $s(n, k)$ 或记号 $[n]_k$ 表示。第一类斯特林数可能为负数, 在使用时一般使用其绝对值, 记为 $c(n, k)$ 。第一类斯特林数具有以下递推关系

$$c(n, k) = [n]_k = |s(n, k)| = (-1)^{-k} s(n, k)$$

$$s(n, k) = s(n-1, k-1) - (n-1) * s(n-1, k), \quad 1 \leq k < n$$

边界条件为: $s(k, 0) = 0, k \geq 1, s(k, k) = 1, k \geq 0$ 。

第二类斯特林数表示把 n 个元素划分为 k 个非空集合的方案数, 通常用 $S(n, k)$ 或记号 $\{n\}_k$ 表示, 其值为非负整数。其递推关系为

$$S(n, k) = k * S(n-1, k) + S(n-1, k-1), \quad 1 \leq k < n$$

边界条件为: $S(k, 0) = 0, k \geq 1, S(k, k) = 1, k \geq 0$ 。

第二类斯特林数和贝尔数 (Bell number)^{II} 关系密切。贝尔数的第 n 项表示将包含 n 个元素的集合划分为非空集合的方法数。贝尔数满足以下关系

^I 莱昂哈特·欧拉 (Leonhard Euler, 1707—1783), 瑞士数学家, 物理学家, 天文学家, 逻辑学家及工程师。他在数学的许多分支中作出了重要且具有影响力的发现, 例如无限小积分及图论, 同时在其他的数学分支如拓扑学和解析数论中也作出了开创性的贡献。

^{II} 埃里克·坦普尔·贝尔 (Eric Temple Bell, 1883—1960), 数学家, 科幻小说作家, 出生于苏格兰, 在美国生活了其一生中的大部分时间。他在发表非小说类作品时使用原名, 而在发表小说类作品时使用 John Taine 的名字。

$$B_{n+1} = \sum_{k=0}^n \binom{n}{k} B_k$$

根据第二类斯特林数的定义, 可以容易得到

$$B_n = \sum_{k=0}^n \{n\}_k$$

6.7.5 调和级数

调和数列 (harmonic number) 是所有自然数的倒数构成的数列, 即

$$1, \frac{1}{2}, \frac{1}{3}, \dots, \frac{1}{n}, \dots$$

调和级数 (harmonic series) 则是指调和数列的和, 即

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} = \sum_{k=1}^n \frac{1}{k}, \quad n \geq 1$$

调和级数这个名称最初来源于音乐, 它与音乐中的泛音序列 (harmonic series) 英文名称相同 (泛音序列是表示泛音强度的数列)。尽管随着 n 的增大, 调和级数的每一项逐渐趋于 0, 但是整个调和级数却是发散的, 即调和级数随着 n 的增大, 趋向于无穷大。虽然调和级数是发散的, 但其发散的速度却非常缓慢, 例如, 它前 10^{43} 项之和不足 100, 由于这个违反直觉的特性, 有许多“佯谬”与调和级数有关, “橡皮筋上的蠕虫”即为一例。问题“橡皮筋上的蠕虫”可以描述如下: 假设一条 1 米长的橡皮筋从左至右水平放置, 橡皮筋上有一只“长生不老”的蠕虫, 它从橡皮筋的左端出发, 向右端爬行, 每分钟爬行 1 厘米, 而橡皮筋沿着蠕虫爬行的方向每分钟也均匀伸长 1 米, 即在第二分钟结束的时候, 橡皮筋是 2 米, 第三分钟结束时, 橡皮筋是 3 米……假设蠕虫在橡皮筋伸展时, 与橡皮筋的整体相对位置不变, 那么这只蠕虫能够爬到橡皮筋的另外一端吗? 初看蠕虫似乎不可能到达橡皮筋的右端, 但是通过数学推理可知蠕虫必定能够爬到橡皮筋的另外一端。以下是推理过程。在第一分钟正好结束的时候, 蠕虫爬过了 1 厘米, 在橡皮筋伸长之前, 蠕虫位于整条橡皮筋长度的 1% 处, 还有 99% 的距离需要爬。接着橡皮筋均匀延长 1 米, 长度变为 2 米, 由于蠕虫的相对位置不变, 它的位置也跟随橡皮筋的伸长而发生改变 (这是正确理解此问题的关键, 即蠕虫和橡皮筋左端的距离会随着橡皮筋的均匀伸长而发生改变, 但是它所处的相对位置不变, 即和橡皮筋左端距离与整个橡皮筋长度的百分比在伸展的前后保持不变), 此时蠕虫仍位于橡皮筋长度的 1% 处, 但由于橡皮筋已经伸展为 2 米, 此时蠕虫与橡皮筋左端的距离已经变为 2 厘米; 接着第二分钟正好结束时, 蠕虫又向右爬出 1 厘米, 位于距离橡皮筋左端 3 厘米处, 此时蠕虫位于橡皮筋长度的 1.5% 的地方, 还有 98.5% 的距离需要爬, 接着橡皮筋伸展为 3 米, 此时蠕虫的相对位置不变, 因此蠕虫会位于距离橡皮筋左端 4.5 厘米处, 接着第三分钟结束, 蠕虫又向右爬出 1 厘米, 到达 5.5 厘米处, 处于橡皮筋长度的 1.83% 处, 还有 98.17% 的距离需要爬过, 在第三分钟结束时, 橡皮筋伸展为 4 米……如此重复。可以将蠕虫在第 n 分钟正好结束时爬过的距离表示为相对于橡皮筋长度的比值:

$$D = \frac{1}{100} \left(1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n} \right) = \frac{H_n}{100}$$

很明显, 当 $H_n \geq 100$ 时, 蠕虫即爬到了橡皮筋的右端, 此时 n 至少为 10^{43} 。以人类的日常生活经验来说, 这是一个难以想象的大数——蠕虫爬到橡皮筋右端的时间至少是宇宙诞生到现在的时间 (约 140 亿年) 的 10^{27} 倍, 橡皮筋的长度至少要达到 10^{27} 光年!

强化练习: 651 Deck^c。

6.7.6 其他序列

OEIS, 英文全称为 On-Line Encyclopedia of Integer Sequences, 中文名称为“整数数列线上大全”，是一个专门记录整数数列的网站，几乎任何有趣的整数数列都可以在该网站上查询到，其网址为：<https://oeis.org/>。该网站包含了众多数列的研究成果，不仅包含各种结论，还附有各个结论的论文出处、作者等信息，方便读者查阅。网站为每个整数数列赋予了一个编号，通过该编号可以快速获得相应整数数列的信息，在后续行文中，会以序号来指代某个整数数列，例如“数列 A000027”意指 OEIS 序号为 A000027 的整数数列（该数列为自然数数列）。

在某些在线编程竞赛中，如果可以使用万维网资源，可以采用如下的技巧来解决有关计数的题目：根据题意，手工计算简单情形下的前几项结果，然后在 OEIS 上查询该结果数列，运气好的话，几乎都可以在该网站上找到结果数列的出处。OEIS 会以表格的形式（实际是一个文本文件，可通过浏览器直接打开）提供数列的前面若干项，而题目中所涉及的结果项数一般均在表格所提供的结果项数范围内，因此可以使用“打表”的方式来构造解题程序。

强化练习：580* Critical Mass^A, 11660* Look-and-Say Sequences^D。

扩展练习：1224* Tile Code^D, 10643* Facing Problem with Trees^D, 10843* Anne's Game^C, 11028* Sum of Product^D, 11270* Tiling Dominoes^D, 11719* Gridland Airports^D, 12022* Ordering T-Shirts^D。

6.8 概率论

概率论 (probability theory)，是研究自然科学和社会科学中随机现象相应规律的数学分支，是研究统计学必不可少的重要工具，其理论和方法已经成为各个行业工作者不可或缺的一种基本工具。在编程竞赛中，有关概率论的题目时常会出现，其特点是编程代码量一般较少，但推导出结论的思维过程对解题者的要求却很高。题目有两种常见的形式：一种是仅涉及概率论的基本概念和公式，不过要顺利得出解题所需的计算式有时却不容易，需要对概率论达到一定的熟练程度之后才能够做到得心应手；另外一种是以概率论作为题目的背景，与其他算法或解题技巧相结合，例如最为常见的是与动态规划相结合，这种题目一般难度相对较高。由于篇幅所限，本节仅列出与解题有关的概率论知识点，同时介绍了一些经典的概率问题，然后附上练习题，如果读者需要更进一步地了解概率论的相关内容，建议读者阅读相应的概率论基础教程^{I[54]}。概率论与动态规划相结合的问题，因其递推关系一般不易推导，相对难度较高，故将这部分习题放置在后续第 11 章“动态规划”再予以介绍。

6.8.1 基本概念

以下是概率论中的一些基本概念，理解和掌握这些基本概率非常重要，这是顺利解题的基础。在分析问题时，要培养将所求状态定义为事件的习惯，这样才能充分应用相关的集合论结果。

随机事件

在概率论中，称具有下述三个特点的试验为随机试验 (random trial)，简称试验^[55]：

(1) 试验可以在相同的条件下重复地进行；

^I 推荐读者阅读 Sheldon M. Ross 所著的《A First Course in Probability, Ninth Edition》，该教程对读者的前置要求不高，只需具备初等微积分知识即可。教程中采用了大量生动的例子来阐释这些理论和方法在实际生活中的运用，使得读者在获得概率论知识的同时，也能够体会到概率论的应用魅力。

- (2) 试验的所有可能结果在试验前已经明确，并且不止一个；
 (3) 试验前不能确定试验后会出现哪一个结果。

在试验中，每一个可能出现的结果称为样本点。全体样本点组成的集合成为样本空间，记做 S 。随机试验样本空间的子集称为随机事件 (random event)，简称为事件。在试验后，如果出现随机事件 A 中所包含的某个样本点，称事件 A 发生，否则称事件 A 不发生。在每次试验后，如果必定有 S 中的一个样本点出现，即 S 为必然事件，空集 \emptyset 是样本空间的一个子集，因而也是一个事件。由于空集 \emptyset 不包含任何一个样本点，因此每次试验后 \emptyset 必定不发生，称 \emptyset 为不可能事件。必然事件 S 与不可能事件 \emptyset 是两个特殊的事件。

随机事件之间的关系与运算

由于事件是一个集合，事件之间的关系与事件之间的运算满足集合论的相关结论。以下介绍随机事件之间的关系与运算。

给定一个随机试验， S 是它的样本空间，下列事件 A, B, C 与 A_i ($i=1, 2, \dots$) 均为 S 的子集。

(1) 如果 A 是 B 的子集，那么称事件 B 包含事件 A ，含义是事件 A 发生必定导致事件 B 发生。例如，事件 A 表示“灯泡寿命不超过 200 小时”，事件 B 表示“灯泡寿命不超过 300 小时”，则事件 B 包含事件 A 。

(2) 如果 A 是 B 的子集，且 B 是 A 的子集，则事件 A 与事件 B 相等。

(3) 事件 $A \cup B = \{\omega: \omega \in A \text{ 或 } \omega \in B\}$ ，称为事件 A 与事件 B 的“和事件”(或称“并事件”)，它的含义是：当且仅当事件 A 与事件 B 中至少有一个发生时，事件 $A \cup B$ 发生。用 $\bigcup_{i=1}^n A_i$ 表示 n 个事件 A_1, \dots, A_n 的和事件；用 $\bigcup_{i=1}^{\infty} A_i$ 表示可列无限个事件 A_1, A_2, \dots 的和事件。

(4) 事件 $A \cap B = \{\omega: \omega \in A \text{ 且 } \omega \in B\}$ ，称为事件 A 和事件 B 的“积事件”(或称“交事件”)，它的含义是：当且仅当事件 A 与事件 B 同时发生时，事件 $A \cap B$ 发生，积事件也可以记作 AB 。用 $\bigcap_{i=1}^n A_i$ 表示 n 个事件 A_1, \dots, A_n 的积事件；用 $\bigcap_{i=1}^{\infty} A_i$ 表示可列无限个事件 A_1, A_2, \dots 的积事件。

(5) 事件 $A - B = \{\omega: \omega \in A \text{ 且 } \omega \notin B\}$ ，称为事件 A 与事件 B 的差事件，它的含义是：当且仅当事件 A 发生且事件 B 不发生时，事件 $A - B$ 发生。

(6) 如果 $A \cap B = \emptyset$ ，称事件 A 与事件 B 互不相容(或互斥)，它的含义是：事件 A 与事件 B 在一次试验之后不会同时发生。例如，事件 A 表示“灯泡寿命不超过 200 小时”，事件 B 表示“灯泡寿命至少为 300 小时”，则 $AB = \emptyset$ ，即 A 与 B 互不相容。

(7) 事件 $S - A$ 称为事件 A 的对立事件(或称逆事件、余事件)，记作¹ $A^c = S - A$ 。它的含义是：当且仅当事件 A 不发生时，事件 A^c 发生，于是 $A \cap A^c = \emptyset$ ， $A \cup A^c = S$ 。由于事件 A 也是的对立事件，因此称事件 A 与事件 A^c 互逆(或互余)。

事件之间的运算满足下述定律：

- (1) 交换律： $A \cup B = B \cup A, A \cap B = B \cap A$ ；
- (2) 结合律： $A \cup (B \cup C) = (A \cup B) \cup C, A \cap (B \cap C) = (A \cap B) \cap C$ ；
- (3) 分配律： $A \cup (B \cap C) = (A \cup B) \cap (A \cup C), A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ ；

¹ 国内的概率论书籍一般将事件 A 的对立事件记做 \bar{A} 。本书为了表示上的便利，在不引起歧义的情况下，使用 A^c 的记法，请读者注意，表示对立事件的右上角小标不为斜体。

(4) 德·摩根律^I: $(A \cap B)^c = A^c \cup B^c$, $(A \cup B)^c = A^c \cap B^c$ 。

概率的公理化定义及性质

概率的公理化定义^{II}: 给定一个随机试验, S 是它的样本空间, 对于任意一个事件 A , 规定一个实数, 记做 $P(A)$ 。如果 $P(\cdot)$ 满足下列三条公理, 那么就称 $P(A)$ 为事件 A 的概率。

(1) **非负性**: 对于任意一个事件 A , $0 \leq P(A) \leq 1$;

(2) **规范性**: $P(S) = 1$;

(3) **可列可加性**: 当可列无限个事件 A_1, A_2, \dots 两两不相容时, 有

$$P(A_1 \cup A_2 \cup \dots) = P(A_1) + P(A_2) + \dots$$

强化练习: 12114 Bachelor Arithmetic^B。

等可能概型

在一次试验后, 随机事件 A 可能发生, 也可能不发生, 随机事件发生可能性的大小可以用区间 $[0, 1]$ 中的一个数来描述, 这个数称为概率。如果样本空间中的每个样本在一次试验后出现的可能性相等, 则称之为等可能概型。

一般称具有下列两个特征的随机试验的数学模型为古典概型:

(1) 试验的样本空间 S 是有限集, 不妨记作 $S = \{\omega_1, \dots, \omega_n\}$;

(2) 每个样本点在一次试验后以相等的可能性出现, 即

$$P(\{\omega_1\}) = \dots = P(\{\omega_n\})$$

在古典概型中, 如果事件 A 包含 n_A 个样本点, 总的样本点数量为 n , 那么规定

$$P(A) = \frac{n_A}{n}$$

使用这种方法计算得到的概率称为古典概率。

假定样本空间 S 是某个区域 (可以是一维, 也可以是二维、三维), 每个样本点等可能地出现, 规定事件 A 的概率为

$$P(A) = \frac{m(A)}{m(S)}$$

其中, $m(S)$ 在一维情形下表示长度, 在二维情形下表示面积, 在三维情形下表示体积, 用这种方法得到的概率称为几何概率。例如: 在数轴上, 给定闭区间 $[-10, 10]$, 则在此区间内任意取一个实数, 其绝对值大于 5 的概率为 $10/20=50\%$; 在给定三角形的外接圆中任意选择一个点, 则此点恰在此三角形内的概率为三角形面积 S_t 与外接圆的面积 S_c 之比; 在由 27 个单位立方体构成的 $3 \times 3 \times 3$ 较大立方体中任选一个空间点, 则

^I 奥古斯都·德·摩根 (Augustus De Morgan, 1806—1871), 英国数学家、逻辑学家。奥古斯都·德·摩根首先发现了在命题逻辑中存在着以下的关系: (1) 非 $(P \text{ 且 } Q) \equiv (\text{非 } P) \text{ 或 } (\text{非 } Q)$; (2) 非 $(P \text{ 或 } Q) \equiv (\text{非 } P) \text{ 且 } (\text{非 } Q)$ 。德·摩根定律在数理逻辑的定理推演中, 在计算机的逻辑设计中以及数学的集合运算中都起着重要的作用。

^{II} 公理 (axiom) 是指理论体系中不言自明的、无需证明的某些基本设定, 是推导出其他结论的基本依据。现代的数理科学大多采用公理化定义。例如欧式几何的五大公设, 由于第五条公设与前四条公设是独立的, 故将最后一条公设稍作更改就能得到非欧几何。又例如狭义相对论的两条基本原理——光速不变原理和等效性原理, 也是狭义相对论这个体系的公理。公理在其相应体系内是最为基本的假设, 无法通过逻辑推导予以证明, 除非存在更为基本的理论体系, 通过更为基本的理论体系进行论证可以推导出上层理论的公理, 而这一更为基本的理论体系中同样也存在相应的公理。

该空间点恰位于这 27 个单位立方体中的某一个的概率为 1/27。

强化练习: [1636 Headshot^C](#), [11628* Another Lottery^C](#)。

扩展练习: [11346* Probability^D](#), [11722* Joining with Friend^D](#), [11971* Polygon^D](#)。

生日问题

如果房间里有 n 个人, 那么没有两个人的生日是同一天的概率是多大? 当 n 多大时, 才能保证此概率小于 50%?

解 不考虑闰年的影响, 假设每年有 365 天, 则每个人的生日都有 365 种可能, 所有 n 个人一共是 365^n 种可能, 假定每种结果的可能性都一样, 则所求事件的概率为

$$p = \frac{365 \times 364 \times \cdots \times (365 - n + 1)}{365^n} = \frac{\prod_{i=1}^n (366 - i)}{365^n}$$

通过简单的计算可知, 当 $n \geq 23$ 时, $p < 0.5$, 即房间里的人数超过 23 人, 则至少有两人为同一天生日的概率为 $1 - p > 0.5$ 。从直觉上看, 该结论似乎让人觉得不可思议, 但是从另一个角度来理解, 由于每两个人生日相同的概率为 $365/365^2 = 1/365$, 而 23 个人一共可以组成 $\binom{23}{2} = 253$ 对, 显然生日相同的概率并不低。当房间内有 50 个人时, 至少有两个人生日在同一天的概率约为 97%, 如果有 100 个人, 则至少有两个人生日在同一天的概率大于 99.9999%。

扩展练习: [10217* A Dinner with Schwarzenegger^D](#)。

6.8.2 条件概率和独立事件

给定一个随机试验, S 是它的样本空间, 对于 S 中的任意两个事件 A, B , 如果 $P(B) > 0$, 称

$$P(A|B) = \frac{P(AB)}{P(B)} \quad (6.1)$$

为在已知事件 B 发生的条件下事件 A 发生的条件概率 (conditional probability)。条件概率也满足概率的公理化定义中的三条公理。根据条件概率的定义, 当 $n \geq 2$ 且 $P(A_1 \cdots A_n) > 0$ 时, 可以用条件概率的定义证明

$$P(A_1 \cdots A_n) = P(A_1)P(A_2|A_1) \cdots P(A_n|A_1 \cdots A_{n-1}) \quad (6.2)$$

公式 (6.2) 又称为任意个事件交的概率的乘法规则。经常使用的情形是: 当 $n=2$ 时, 如果 $P(A) > 0$, 则 $P(AB) = P(A)P(B|A)$; 当 $n=3$ 时, 如果 $P(ABC) > 0$, 则 $P(ABC) = P(A)P(B|A)P(C|AB)$ 。

取球问题

假设盒子中有 a 个红球和 b 个白球, $a > 0$ 且 $b > 0$ 。现在无放回的取出两个球, 若每次取球时, 盒子中每个球被取中的可能性相同, 则取出的两个球都是红球的概率是多少?

解 令 R_1 和 R_2 分别表示第一次与第二次取出红球的事件, 若第一次取出的是红球, 那么盒子中剩下 $a - 1$ 个红球和 b 个白球, 有

$$P(R_2|R_1) = \frac{a - 1}{a + b - 1}$$

由于第一次取出红球的概率 $P(R_1) = a/(a+b)$, 则根据条件概率的公式定义 (6.1) 可得

$$P(R_1R_2) = P(R_1)P(R_2|R_1) = \frac{a}{a+b} \cdot \frac{a-1}{a+b-1} = \frac{a(a-1)}{(a+b)(a+b-1)}$$

强化练习: [542 France '98^B](#), [11181 Probability|Given^C](#)。

扩展练习: [10169* Urn-Ball Probabilities^D](#)。

系统可靠度

对于任意两个事件 A, B , 如果等式 $P(AB)=P(A)P(B)$ 成立, 那么称事件 A 与事件 B 相互独立。应用独立性的概念可以解决实际中串、并联系统的可靠性问题。一个产品(或一个元件, 或一个系统)的可靠性可以用可靠度来度量, 可靠度指的是产品能够正常工作(即在规定的时间内和规定的条件下完成规定功能)的概率。如果一个系统中, 各个元件能否正常工作都是相互独立的, 那么可以根据随机事件的独立性来获得整个系统的可靠性。下面介绍常见系统可靠度的计算方法。

设一个系统由 n 个元件串联而成, 第 i 个元件的可靠度为 p_i , $i=1, \dots, n$, 试求这个串联系统的可靠度。设事件 A_i 表示“第 i 个元件正常工作”, $i=1, \dots, n$, 由于“串联系统能正常工作”等价于“ n 个元件都正常工作”, 因此整个串联系统的可靠度为

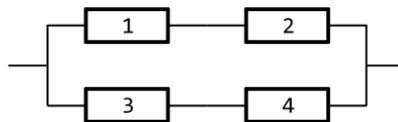
$$P(A_1 \cdots A_n) = \prod_{i=1}^n P(A_i) = \prod_{i=1}^n p_i$$

设一个系统由 n 个元件并联而成, 第 i 个元件的可靠度为 p_i , $i=1, \dots, n$, 试求这个并联系统的可靠度。设事件 A_i 表示“第 i 个元件正常工作”, $i=1, \dots, n$, 由于“并联系统能正常工作”等价于“ n 个元件中至少有一个元件正常工作”, 因此整个并联系统的可靠度为

$$P(A_1 \cup \cdots \cup A_n) = 1 - P(A_1^c \cdots A_n^c) = 1 - \prod_{i=1}^n P(A_i^c) = 1 - \prod_{i=1}^n (1 - p_i)$$

混联系统可靠度

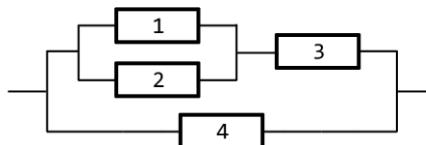
(1) 给定如下的一个由电阻构成的混联电路, 假设每个电阻均独立工作且可靠度均为 p , 则整个混联电路的可靠度为多少?



解 电阻 1 和 2、电阻 3 和 4 分别构成串联电路, 然后整体再构成并联电路。令事件 A_i 为“电阻 i 正常工作”, $i=1, 2, 3, 4$ 。事件 F 为“混联电路能够正常工作”, 有

$$P(F) = 1 - (1 - P(A_1 A_2))(1 - P(A_3 A_4)) = 1 - (1 - p^2)(1 - p^2) = 2p^2 - p^4$$

(2) 给定如下的一个由电阻构成的混联电路, 假设每个电阻均独立工作且可靠度均为 p , 则整个混联电路的可靠度为多少?



解 电阻 1 与 2 组成一个并联的子系统 S_1 , 根据并联系统的可靠度计算公式, 可知子系统 S_1 的可靠度为 $1 - (1 - p)^2 = 2p - p^2$, 把此子系统 S_1 作为一个新的电阻, 它与电阻 3 组成一个串联的子系统 S_2 , 根据串联系统的可靠度计算公式, 可知子系统 S_2 的可靠度为 $(2p - p^2)p$, 整个系统由子系统 S_2 和电阻 4 并联而成, 由并联系统的可靠度公式可知整个混联系统的可靠度为

$$1 - [1 - (2p - p^2)p](1 - p) = p + 2p^2 - 3p^3 + p^4$$

灭绝概率

假设某种细胞通过分裂进行繁殖且存活时间固定不变, 即单个细胞存活 t 小时后会发生凋亡^I。在单个细胞的存活期间, 通过分裂最多能够产生 n 个子细胞, 其中产生 i 个子细胞的概率为 p_i , $0 \leq i \leq n$ 。设初始时只有 1 个细胞, 则经过 m 个存活周期后, 该细胞及其所有子细胞均已凋亡的概率是多少?

解 如果分别计算每个细胞及其子细胞的凋亡概率则情形较为复杂, 不易处理, 如果将单个细胞和其子细胞作为一个整体来对待, 则可以简化问题的解决。由于各个细胞凋亡相互之间属于独立事件, 则可做如下的假设: 令 $Ap[m]$ 为单个细胞在 m 个存活周期后全部发生凋亡的概率 (包括在 m 个存活周期之前就已经凋亡的情形), 按照单个细胞产生 i 个子细胞的概率, $Ap[m]$ 可以递归表示为

$$Ap[m] = p_0 + p_1 * Ap[m - 1] + p_2 * (Ap[m - 1])^2 + \cdots + p_n * (Ap[m - 1])^n, \quad m \geq 1, \quad Ap[0] = 0$$

也就是说, 单个细胞在经过 m 个存活周期后, 自身及其后代子细胞全部已经发生凋亡的概率是以下情形概率的总和: 产生 i 个子细胞, 然后这 i 个子细胞及其各自的后代在 $m-1$ 个存活周期内全部凋亡的概率。由于 i 个子细胞各自在 $m-1$ 个周期内凋亡的概率为 $Ap[m-1]$ 且互为独立事件, 则 i 个子细胞全部凋亡的概率为 $(Ap[m-1])^i$, 而初始细胞产生 i 个子细胞的概率为 p_i , 那么根据乘法规则, 初始单个细胞产生 i 个子细胞然后在 $m-1$ 个存活周期内凋亡的概率为 $p_i \times (Ap[m-1])^i$ 。需要注意, p_0 指的是细胞产生 0 个后代的概率, 也就是经过一个存活周期后直接凋亡而没有子细胞产生的概率。

强化练习: 561 Jackpot^D, 10056 What is the Probability^A, 11021 Tribles^C, 12461* Airplane^B。

6.8.3 全概率公式与贝叶斯公式

设 A 和 B 为两个事件, 可以将 B 表示为

$$B = BA \cup BA^c$$

这样, B 中的结果, 要么同时属于 B 和 A , 要么只属于 B 但不属于 A , 显然, BA 和 BA^c 是互不相容的, 因此, 根据概率公理, 有

$$\begin{aligned} P(B) &= P(BA) + P(BA^c) \\ &= P(B|A)P(A) + P(B|A^c)P(A^c) \\ &= P(B|A)P(A) + P(B|A^c)[1 - P(A)] \end{aligned} \tag{6.3}$$

公式 (6.3) 说明事件 B 发生的概率等于“在 A 发生的条件下 B 发生的条件概率”与“在 A 不发生的条件下 B 发生的条件概率”的加权平均, 其中加在每个条件概率上的权重就是作为条件的事件发生的概率。这是一个非常有用的公式, 它使得我们能够通过以第二个事件发生与否作为条件来计算第一个事件的概率。也就是说, 在许多问题中, 直接计算第一个事件的概率很困难, 但是一旦知道第二个事件发生与否就容易计算第一个事件的概率。

公式 (6.3) 还可以进一步推广——如果 n 个事件 A_1, \dots, A_n 满足条件: (1) A_1, \dots, A_n 两两互不相容, (2) $A_1 \cup \dots \cup A_n = S$, 那么称这 n 个事件 A_1, \dots, A_n 构成样本空间 S 的一个划分 (或构成一个完备事件组)。设 n 个事件 A_1, \dots, A_n 构成样本空间 S 的一个划分, 记 B 是一个事件且

$$B = \bigcup_{i=1}^n BA_i$$

^I 细胞凋亡 (apoptosis) 是指生物体为维持内环境稳定, 由基因控制的细胞的自主有序死亡。

当 $P(A_i) > 0$ ($i=1, \dots, n$) 时, 有

$$P(B) = \sum_{i=1}^n P(B|A_i)P(A_i) \quad (6.4)$$

称公式 (6.4) 为全概率公式, 即 $P(B)$ 等于 $P(B|A_i)$ 的加权平均, 每项的权为事件 A_i 发生的概率。对于事件 A_1, A_2, \dots, A_n , 其中一个或者仅有一个发生, 可以通过 A_i 中一个发生的条件概率来计算 $P(B)$ 。

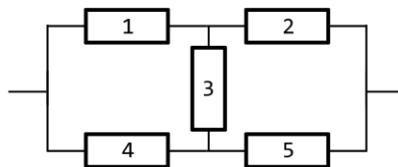
利用公式 (6.4) 可以证明, 当 $P(B) > 0$ 时, 有

$$P(A_j|B) = \frac{P(B|A_j)P(A_j)}{\sum_{i=1}^n P(B|A_i)P(A_i)} \quad (6.5)$$

公式 (6.5) 称为贝叶斯公式 (Bayes's theorem), 根据英国哲学家托马斯·贝叶斯命名。如果把事件 A_i 设想为关于某个问题的各个可能的“假设条件”, 则贝叶斯公式可以这样理解: 在试验之前对这些假设条件所作的判断 (即 $P(A_i)$), 可以如何根据试验的结果来进行修正。

桥式系统可靠度

给定如下的一个由电阻构成的桥式电路, 假设每个电阻均独立工作且可靠度均为 p , 则整个桥式电路的可靠度为多少?



解 根据电阻 3 是否能够正常工作进行分析。当电阻 3 能正常工作时, 只要 1 和 4 两个电阻中任意一个正常工作且 2 和 5 两个电阻中任意一个正常工作, 桥式系统就能正常工作; 若电阻 3 不能正常工作, 只要 1 和 2 两个电阻都正常工作或者 4 和 5 两个电阻都正常工作, 桥式系统就能正常工作。令事件 A_i 为“电阻 i 正常工作” ($i=1, 2, 3, 4, 5$), 事件 F 为“桥式系统正常工作”, 则事件 A_3 表示电阻 3 正常工作, 事件 A_3^c 表示电阻 3 不能正常工作, 显然 A_3 和 A_3^c 为互不相容事件, 进一步地, FA_3 和 FA_3^c 亦为互不相容事件, 因此, 根据条件概率定义及全概率公式有

$$\begin{aligned} P(F) &= P(F|A_3)P(A_3) + P(F|A_3^c)P(A_3^c) \\ &= P((A_1 \cup A_4)(A_2 \cup A_5))P(A_3) + P((A_1A_2) \cup (A_4A_5))(1 - P(A_3)) \\ &= (1 - (1 - p)^2)(1 - (1 - p)^2)p + (1 - (1 - p^2)(1 - p^2))(1 - p) \\ &= 2p^5 - 5p^4 + 2p^3 + 2p^2 \end{aligned}$$

蒙提霍尔问题

蒙提霍尔问题 (Monty Hall problem)^{[56][57]}, 最初来源于美国的一档电视游戏类节目——Let's Make a Deal, 因其主持人为 Monty Hall 而得名。问题可以描述如下: 在电视游戏节目中, 主持人设置了三道门, 分别为 1 号、2 号、3 号, 其中一扇门后面有一辆小轿车作为奖品, 另外两扇门后各为一只山羊。竞猜者可以随机挑选一扇门, 比如, 选择 1 号门, 之后主持人在 2 号门和 3 号门中选择一扇藏有山羊的门打开, 比如说 3 号门, 然后让你作出选择: 你是否愿意放弃选择 1 号门转而选择 2 号门?

解 更换门的选择能够使得中奖率更高吗? 从直觉来看, 有两种主要的意见: 第一种意见认为剩下的两扇门必定一扇门是山羊, 另外一扇门是小轿车, 换选后中奖概率是 $1/2$, 而不换选的中奖概率是 $1/3$; 第二种意见认为只剩下两扇尚未选择的门, 而门后面有小轿车的概率同样是 $1/3$, 所以换选和不换选的中奖概

率不变。但是通过谨慎的理论分析和计算机数据模拟都可以得出以下正确的结论：不换选的中奖概率为 $1/3$ ，换选的中奖概率为 $2/3$ 。这个结论非常反直觉，使得当时的很多读者（其中 $1/10$ 的人具有博士学位）写信给刊登这个结论的杂志，表示他（她）们拒绝相信“换选具有 $2/3$ 的中奖概率”是正确的结论。

实际上转换一下思维方式就很容易理解这个问题，从而明白正确结论就是换选门具有更高的中奖概率。假设竞猜者选择的是 1 号门，则有以下可能性：

1号门	2号门	3号门	不换选	换选
小轿车	山羊	山羊	小轿车	山羊
山羊	小轿车	山羊	山羊	小轿车
山羊	山羊	小轿车	山羊	小轿车

容易看出，不换选中奖的概率为 $1/3$ ，换选后中奖概率为 $2/3$ 。如果竞猜者初始选择 2 号门或者 3 号门具有同样的结果。

以上通过列举的方式比较直观地说明了为什么换选之后中奖概率为 $2/3$ ，下面应用概率论方法从条件概率的角度进行解释。定义以下事件：事件 A 为竞猜者的初始选择为中奖选择；事件 B 为竞猜者的初始选择不是中奖选择；事件 R_1 为竞猜者坚持原选择中奖；事件 R_2 为竞猜者改变选择并中奖。显然事件 A 和 B 互为独立事件，且 $A \cup B = S$ ，因为 $P(A) = 1/3$ ，则 $P(B) = 1 - P(A) = 2/3$ 。进一步地，若竞猜者坚持原选择，显然 $P(R_1) = P(A) = 1/3$ ；若竞猜者更改选择，定义事件 C 为“除去竞猜者选择的门和主持人打开的门而余下的门后面有小轿车”，则根据条件概率的定义有 $P(C|A) = 0$ ， $P(C|B) = 1$ ，根据全概率公式有

$$P(R_2) = P(C) = P(A)P(C|A) + P(B)P(C|B) = \frac{1}{3} \times 0 + \frac{2}{3} \times 1 = \frac{2}{3}$$

扩展蒙提霍尔问题

将蒙提霍尔问题适当扩展可以得到以下问题：假设有 n 扇门， $3 \leq n$ ，其中只有一扇门后藏有小轿车，当你选定一扇门后，主持人把另外 b 扇藏有山羊的门打开， $1 \leq b \leq n-2$ ，则换选门后的中奖概率为多少？

解 仍然沿用前述的事件定义，易知此种情形， $P(A) = 1/n$ ， $P(B) = (n-1)/n$ ， $P(R_1) = P(A) = 1/n$ 。当事件 A 发生时，由于只有 1 扇门后有奖品，则事件 C 发生的概率为 0，即 $P(C|A) = 0$ ，当事件 B 发生时，由于已经打开了 b 扇没有奖品的门，奖品只可能在剩余的 $n-b-1$ 扇门中（除去竞猜者已经选择的一扇门），竞猜者选择一扇门中奖的概率为 $1/(n-b-1)$ ，则 $P(C|B) = 1/(n-b-1)$ ，根据全概率公式有

$$P(R_2) = P(C) = P(A)P(C|A) + P(B)P(C|B) = \frac{1}{n} \times 0 + \frac{n-1}{n} \times \frac{1}{n-b-1} = \frac{n-1}{n(n-b-1)}$$

于是，换选门后的中奖概率为 $(n-1)/(n(n-b-1))$ 。

在理解上述问题的基础上，读者可以继续思考以下问题^I：

(1) 假设不是一扇门后面有奖品，而是 a 扇门后面有奖品， $3 \leq n$ ， $1 \leq a \leq n-2$ ， $1 \leq b \leq n-a-1$ ，其他

^I (1) 按照类似的思路可知，中奖概率为

$$P(C) = P(A)P(C|A) + P(B)P(C|B) = \frac{a}{n} \times \frac{a-1}{n-b-1} + \frac{n-a}{n} \times \frac{a}{n-b-1} = \frac{a(n-1)}{n(n-b-1)}$$

(2) 继续按照类似的思路，中奖概率为

$$P(C) = P(A)P(C|A) + P(B)P(C|B) = \frac{a}{n} \times \frac{a-c-1}{n-b-1} + \frac{n-a}{n} \times \frac{a-c}{n-b-1} = \frac{n(a-c)-a}{n(n-b-1)}$$

规则不变，则换选门后的中奖概率是多少？

(2) 仍然是 a 扇门后面有奖品，但主持人打开另外 b 扇门，且 b 扇打开的门中有 c 扇门后包含奖品， $3 \leq n, 1 \leq a \leq n-2, 1 \leq b \leq n-2, 0 \leq c < \min(a, b)$ ，则换选门后的中奖概率是多少？

强化练习：[10491 Cows and Cars](#)^A。

赌徒破产问题

赌徒破产问题 (gambler's ruin problem) 有多种等价的描述形式，其中一种描述形式如下：假设某个赌徒拥有 i 元的赌资，其在赌场中通过抛掷骰子进行赌博，每次抛掷骰子有概率 p 赢得一元钱，有 $1-p$ 的概率输掉一元钱，则赌徒在输光之前赌资能够达到 N 元的概率是多少？

解 令 E 表示事件“开始时赌徒有 i 元，最后拥有 N 元赌资”，显然此事件和赌徒最初的钱数有关，记 $P_i = P(E)$ 。以第一次抛掷骰子的结果为条件，令 H 表示事件“第一次抛掷的结果赌徒赢得一元钱”，则根据全概率公式有

$$P_i = P(E) = P(E|H)P(H) + P(E|H^c)P(H^c) = pP(E|H) + (1-p)P(E|H^c)$$

假定第一次抛掷骰子的结果为赌徒赢得一元钱，则第一次赌博结束后的状态是：赌徒拥有 $i+1$ 元赌资，因为随后的抛掷都同前面独立并且赌徒赢得一元钱的概率都为 p ，故从该时刻开始，赌徒的赌资能够达到 N 元的概率等同于以下情形——初始时赌徒拥有 $i+1$ 元赌资，因此 $P(E|H) = P_{i+1}$ 。类似的，可得 $P(E|H^c) = P_{i-1}$ 。令 $q = 1-p$ ，可得

$$P_i = pP_{i+1} + qP_{i-1}, \quad i = 1, 2, \dots, N-1$$

根据前述 P_i 的定义，当 i 为 0 时，赌徒已经输光赌资，有 $P_0 = 0$ ，而当 i 为 N 时，已经达到目标条件，有 $P_N = 1$ 。利用求解差分方程的解法，令 $P_i = \gamma^i$ ，有

$$\gamma^i = p\gamma^{i+1} + q\gamma^{i-1} \Rightarrow p\gamma^2 - \gamma + q = 0$$

解得

$$\gamma = \frac{1 \pm \sqrt{1 - 4pq}}{2p} = \frac{1 \pm \sqrt{1 - 4p + 4p^2}}{2p} = \frac{1 \pm (1 - 2p)}{2p} = \frac{q}{p} \text{ 或 } 1$$

当 $p \neq q$ 时，差分方程有两个异根，令 $P_i = C + D(q/p)^i$ ，其中 C 和 D 是常数，根据 $P_0 = 0$ 和 $P_N = 1$ ，可得方程组

$$\begin{aligned} C + D(q/p)^N &= 1 \\ C + D(q/p)^0 &= 1 \end{aligned}$$

最后解得

$$P_i = \frac{1 - (q/p)^i}{1 - (q/p)^N}$$

当 $p = q$ 时，令 $z = q/p$ ，取上式 z 趋近于 1 时的极限，有

$$P_i = \lim_{z \rightarrow 1} \frac{1 - z^i}{1 - z^N} = \lim_{z \rightarrow 1} \frac{iz^{i-1}}{Nz^{N-1}} = \frac{i}{N}$$

因此有

$$P_i = \begin{cases} \frac{1 - (q/p)^i}{1 - (q/p)^N} & \text{如果 } p \neq \frac{1}{2} \\ \frac{i}{N} & \text{如果 } p = \frac{1}{2} \end{cases}$$

从上述结果可知，赌徒得胜的概率与其初始时所拥有的赌资有关，假设每次获胜的概率对赌徒和赌场来说，都很公平，均为 $1/2$ ，则双方哪一方赌资越多，最后赢得所有钱的可能性越大。相对于赌场所拥有的钱

来说，赌徒所拥有的赌资一般是很小的，所以赌徒最终赢的概率很小，反过来，赌场拥有的赌资很大，因此有很大的可能性赢光赌徒的所有钱，因此在此种情况下，赌徒有很大的概率会输光，即宣告破产。

强化练习：11500 Vampires^C。

6.8.4 随机变量

随机变量（random variable）是指定义在试验样本空间上的实值函数，因为随机变量的取值由试验结果确定，可以对随机变量的可能取值指定概率。以下是若干在解题中常见的随机变量类型^I。

二项随机变量

考虑一个试验，其结果分为两类，成功或者失败，令

$$X = \begin{cases} 1, & \text{当试验结果为成功时} \\ 0, & \text{当试验结果为失败时} \end{cases}$$

$p (0 \leq p \leq 1)$ 为每次试验成功的概率，则 X 的分布列为

$$\begin{aligned} p(0) &= P\{X = 0\} = 1 - p \\ p(1) &= P\{X = 1\} = p \end{aligned}$$

如果随机变量 X 的分布列由上式给出，其中 $p \in (0, 1)$ ，则称 X 为伯努利随机变量（根据瑞士数学家詹姆斯·伯努利的名字命名）。现在假设进行 n 次独立重复试验，每次试验成功的概率为 p ，失败的概率为 $1-p$ ，如果 X 表示 n 次试验中成功的次数，那么称 X 为参数是 (n, p) 的二项随机变量（binomial random variable），因此，伯努利随机变量也是参数为 $(1, p)$ 的二项随机变量。参数为 (n, p) 的二项随机变量的分布列为

$$p(i) = \binom{n}{i} p^i (1-p)^{n-i}, \quad i = 0, 1, \dots, n$$

由二项式定理可得

$$\sum_{i=0}^{\infty} p(i) = \sum_{i=0}^n \binom{n}{i} p^i (1-p)^{n-i} = [p + (1-p)]^n = 1$$

一般地，假定 n 个试验的试验结果是相互独立的，便称这 n 个试验相互独立。如果在一个试验中只关心某个事件 A 是否发生，那么称这个试验为伯努利试验（Bernoulli trial），相应的数学模型称为伯努利概率。如果把伯努利试验独立地重复做 n 次，这 n 个试验合在一起称为 n 重伯努利试验。

设事件 B_k 表示“ n 重伯努利试验中事件 A 恰发生了 k 次”， $k=0, 1, \dots, n$ ，通常记为 $P(B_k)$ 为 $P_n(k)$ 。由于 n 个试验是相互独立的，因此，事件 A 在指定的 k 个试验中发生，且在其余 $(n-k)$ 个试验中不发生的概率为

$$P_n(k) = \binom{n}{k} p^k (1-p)^{n-k}, \quad k = 0, 1, \dots, n$$

通常称 $P_n(k)$ 为二项概率，因为它恰是 $[(1-p)+p]^n$ 的二项展开中的第 k 项， $k=0, 1, \dots, n$ 。例如在抛硬币过程中，由于正面和反面出现的概率均为 $1/2$ ，故抛掷 n 次硬币，恰有 k 次出现正面（或反面）的概率为

$$P_n(k) = \binom{n}{k} p^n, \quad k = 0, 1, \dots, n$$

几何随机变量

考虑在独立重复试验中，每次成功的概率为 $p (0 < p < 1)$ ，重复试验直到试验首次成功为止，如果令 X

^I 定义源自 Sheldon M. Ross 所著的《A First Course in Probability, Ninth Edition》。

表示需要试验的次数, 有

$$P\{X = n\} = (1 - p)^{n-1}p, \quad n = 1, 2, \dots \quad (6.4)$$

式 (6.4) 之所以成立是因为要使 X 等于 n , 充分必要条件是前 $n-1$ 次试验失败而第 n 次试验成功, 由于假定各次试验都是相互独立的, 因此等式成立。

负二项随机变量

假定独立重复试验中, 每次成功的概率为 p , $0 < p < 1$, 试验持续进行直到试验累计成功 r 次为止, 如果令 X 表示试验的总次数, 则

$$P\{X = n\} = \binom{n-1}{r-1} p^r (1-p)^{n-r}, \quad n = r, r+1, \dots \quad (6.5)$$

要使得第 n 次试验时, 恰好 r 次试验成功, 那么前 $n-1$ 次试验中必定有 $r-1$ 次成功, 且第 n 次试验必然是成功, “前 $n-1$ 次试验中有 $r-1$ 次成功”的概率为

$$\binom{n-1}{r-1} p^{r-1} (1-p)^{n-r}$$

而“第 n 次试验成功”的概率为 p , 由于这两个事件相互独立, 故式 (6.5) 成立。对于任意随机变量 X , 如果 X 的分布列由式 (6.5) 给出, 那么就称 X 是参数为 (r, p) 的负二项随机变量 (negative binomial random variable)。根据该定义, 几何随机变量是参数为 $(1, p)$ 的负二项随机变量。

强化练习: 557 Burge^C。

扩展练习: 10218* Let's Dance^D。

6.8.5 期望

概率论和统计学中, 随机变量的期望是一个重要的概念。在编程竞赛中, 经常出现的是计算离散型随机变量或连续型随机变量的期望, 而在连续型随机变量的期望中, 又以计算均匀随机变量的期望较为常见。

离散型随机变量的期望

如果随机变量只能取得有限个值, 或者是无穷个值但能按一定次序逐个列出, 其值域为一个或若干个有限或无限区间, 这样的随机变量称为离散型随机变量。对于一个离散型随机变量 X , 定义 X 的概率分布列 (probability mass function) $p(a)$ 为

$$p(a) = P\{X = a\}$$

分布列 $p(a)$ 最多在可数个 a 上取正值, 即如果 X 的可能值为 x_1, x_2, \dots , 那么

$$p(x_i) \geq 0, \quad i = 1, 2, \dots$$

$$p(x) = 0, \quad \text{所有其他 } x$$

由于 X 必定取值于 $\{x_1, x_2, \dots\}$, 故有

$$\sum_{i=1}^{\infty} p(x_i) = 1$$

如果 X 是一个离散型随机变量, 其分布列为 $p(x)$, 那么 X 的期望 (expectation) 或期望值 (expected value), 记为 $E[X]$, 定义为

$$E[X] = \sum_{x: p(x) > 0} x p(x)$$

其中 $p(x_i)$ 亦可使用随机变量 X 出现的频率 $f(x_i)$ 替代, 类似于加权平均, 即

$$E[X] = x_1 * p(x_1) + x_2 * p(x_2) + \dots + x_n * p(x_n)$$

$$= x_1 * f(x_1) + x_2 * f(x_2) + \cdots + x_n * f(x_n)$$

或者表示成

$$E[X] = \sum_{k=1}^{\infty} x_k p_k = \sum_{k=1}^{\infty} x_k f_k$$

例如, 假设 X 表示掷一枚均匀 (六面) 骰子出现的点数, 由于骰子出现 1 到 6 点的概率相同, 均为 $1/6$, 则一次投掷所获得的点数期望

$$E[X] = 1 \times \frac{1}{6} + 2 \times \frac{1}{6} + 3 \times \frac{1}{6} + 4 \times \frac{1}{6} + 5 \times \frac{1}{6} + 6 \times \frac{1}{6} = \frac{7}{2}$$

在离散型随机变量的期望中, 经常应用的一个性质是一组随机变量的和的期望等于这组随机变量各自期望的和, 即对于随机变量 X_1, X_2, \dots, X_n , 有

$$E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i]$$

例如, 求 n 次投掷骰子所得点数之和的期望。令 X 表示点数之和, 则

$$X = \sum_{i=1}^n X_i$$

其中 X_i 表示第 i 次投掷骰子时所获得的点数, 因为 X_i 从 1 到 6 取值的概率相等, 则

$$E[X_i] = \sum_{i=1}^6 i * \frac{1}{6} = \frac{21}{6} = \frac{7}{2}$$

于是

$$E[X] = E\left[\sum_{i=1}^n X_i\right] = \sum_{i=1}^n E[X_i] = \frac{7n}{2}$$

连续型随机变量的期望

设 X 是一个随机变量, 如果存在一个定义在实数轴上的非负函数 f , 使得对于任意一个实数集 B , 满足

$$P\{X \in B\} = \int_B f(x) dx$$

则称 X 为连续型随机变量 (continuous random variable), 函数 f 称为随机变量 X 的概率密度函数 (probability density function), 或称密度函数。从几何上来看, X 属于 B 的概率可以由概率密度函数 $f(x)$ 在集合 B 上的积分得到, 因为 X 必须取某个值, 根据概率公理中全集的概率为 1 的性质, f 必须满足

$$1 = P\{X \in (-\infty, \infty)\} = \int_{-\infty}^{+\infty} f(x) dx$$

所有关于 X 的概率都可以由 f 得到。例如, 令 $B = [a, b]$, 可得

$$P\{a \leq X \leq b\} = \int_a^b f(x) dx$$

若令 $a = b$, 可得

$$P\{X = a\} = \int_a^a f(x) dx = 0$$

也就是说, 连续型随机变量取任何固定值的概率都等于 0, 则对于一个连续型随机变量 X , 有

$$P\{X < a\} = P\{X \leq a\} = F(a) = \int_{-\infty}^a f(x) dx$$

前述的离散型随机变量的期望定义可以改写为

$$E[X] = \sum_x x P\{X = x\}$$

如果 X 是一个连续型随机变量, 密度函数为 $f(x)$, 对于很小的 dx 有

$$f(x)dx \approx P\{x \leq X \leq x + dx\}$$

因此可以用类似的方法定义连续型随机变量的期望为

$$E[X] = \int_{-\infty}^{+\infty} x f(x) dx$$

如果一个随机变量 X 的密度函数为

$$f(x) = \begin{cases} \frac{1}{b-a}, & a < x < b \\ 0, & \text{其他情形} \end{cases}$$

则称随机变量 X 在 (a, b) 区间上均匀分布 (uniformly distribution)。相应地, 随机变量 X 在 (a, b) 上的期望为

$$E[X] = \int_{-\infty}^{+\infty} x f(x) dx = \int_a^b \frac{x}{b-a} dx = \frac{b^2 - a^2}{2(b-a)} = \frac{b+a}{2}$$

即某个区间上均匀随机变量的期望就等于该区间中点的值。

解决连续型随机变量的期望一般需要应用积分^[58]。

条件期望

当 X 和 Y 的联合分布为离散分布时, 对于 $P\{Y=y\} > 0$ 的 y 值, 给定 $Y=y$ 的条件之下, X 的条件分布列定义为

$$p_{X|Y}(x|y) = P\{X = x | Y = y\} = \frac{p(x, y)}{p_Y(y)}$$

对于所有满足 $p_Y(y) > 0$ 的 y , X 在给定 $Y=y$ 之下的条件期望为

$$E[X|Y = y] = \sum_x x P\{X = x | Y = y\} = \sum_x x p_{X|Y}(x|y)$$

类似的, 设 X 和 Y 有连续型联合分布, 其联合密度函数为 $f(x, y)$, 对于给定的 $Y=y$, 当 $f_Y(y) > 0$ 时, X 的条件密度函数定义为

$$f_{X|Y}(x|y) = \frac{f(x, y)}{f_Y(y)}$$

给定 $Y=y$ 的条件下, 假定 $f_Y(y) > 0$, X 的条件期望为

$$E[X|Y = y] = \int_{-\infty}^{+\infty} x f_{X|Y}(x|y) dx$$

记 $E[X|Y]$ 表示随机变量 Y 的函数, 它在 $Y=y$ 处的值为 $E[X|Y=y]$, 注意到 $E[X|Y]$ 本身也是一个随机变量。

条件期望一个非常重要的性质就是随机变量 X 的期望可以由随机变量 $E[X|Y]$ 的期望来确定, 该性质又称为全期望公式 (law of total expectation), 即

$$E[X] = E[E[X|Y]]$$

如果 Y 是离散型随机变量, 其形式为

$$E[X] = \sum_y E[X|Y = y] P\{Y = y\}$$

如果 Y 是连续型随机变量, 密度函数为 $f_Y(y)$, 其形式为

$$E[X] = \int_{-\infty}^{+\infty} E[X|Y=y]f_Y(y)dy$$

根据以上结论, 可以很容易计算某随机变量 X 在给定条件之下的条件期望, 然后再对条件期望求平均, 最后得到的结果即为随机变量 X 的期望。

期望的性质

期望具有“线性”性质, 即对于两个相互独立的随机变量 X 和 Y , 有

$$E[X \pm Y] = E[X] \pm E[Y]$$

$$E[XY] = E[X]E[Y]$$

$$E\left[\frac{X}{Y}\right] = \frac{E[X]}{E[Y]}, E[Y] \neq 0$$

解决有关期望问题的关键是分析问题的条件, 恰当地定义期望的形式, 找到所求期望的“递推关系”, 有时候递推关系并不明显, 此时可以列举一些简单的情形, 推算出相应的期望, 以便在此过程中发现和总结规律。

10900 So You Want to Be a 2ⁿ-aire?^c (你想成为 2ⁿ富翁吗?)

玩家最初拥有 1 美元奖金, 共回答 n 个问题。对于每个问题, 他有两种选择:

(1) 不回答问题, 退出游戏, 奖金为玩家当前所拥有的美元数。

(2) 回答该问题, 如果答错, 退出游戏且一无所有; 回答正确, 所得奖金为玩家当前所拥有美元数的两倍, 然后接着回答下一个问题。

当回答完最后一个问题后, 玩家退出游戏。玩家希望将他能够获得的期望奖金最大化。每当回答某个问题时, 玩家有概率 p 能够正确回答这个问题, 对于每个问题来说, 概率 p 是一个在闭区间 $[t, 1]$ 均匀分布的随机变量。

输入

输入包含多组测试数据, 每组包含两个数值, 整数 n 和实数 t , $1 \leq n \leq 30$, $0 \leq t \leq 1$ 。输入最后一行以‘0 0’结束, 此组测试数据不需处理。

输出

对于每组测试数据, 假设玩家采用最优的游戏策略, 确定玩家能够获得的期望奖金。输出的结果保留小数点后 3 位小数。

样例输入

```
1 0.5
1 0.3
2 0.6
24 0.25
0 0
```

样例输出

```
1.500
1.357
2.560
230.138
```

分析

题意并未明确说明何为“最优策略”，需要解题者根据题意自行加以确定^I。假设玩家当前的奖金数量为 C 且游戏尚未结束，此时玩家有概率 p 能够正确回答当前问题（注意概率 p 的取值在区间 $[t, 1]$ 上均匀分布，故 p 取固定值的概率为 0），如果回答正确，则玩家拥有的奖金数量将为 $2C$ ，如果回答错误，奖金数量将为 0。那么，在当前情况下，玩家的奖金 P 的期望为

$$E[P] = 2C * p + 0 * (1 - p) = 2pC$$

由于玩家的目标是最大化期望奖金，从理性的角度讲，如果 $2pC$ 大于 C ，玩家应该选择答题，否则应该选择不答题。将上述情况一般化，设 X_k 表示尚有 k 个问题未回答时的奖金， P_k 表示在尚有 k 个问题未回答的情况下，正确回答下一个问题的概率，那么题目所求为 $E[X_n]$ 。令 $f(k)$ 表示尚有 k 个问题未回答时的期望奖金，即 $f(k) = E[X_k]$ ，此时玩家有概率 P_k 能够获得期望奖金 $f(k-1)$ ，使用 $E[X_k | P_k = p]$ 表示当 $P_k = p$ 时 X_k 的条件期望，若玩家采用最优游戏策略，则有

$$E[X_k | P_k = p] = \max(2^{n-k}, p \cdot f(k-1)), p \in [t, 1], 1 \leq k \leq n$$

当 $k=0$ 时，游戏结束，此时奖金数量为 2^n ，因此有

$$f(0) = E[X_0] = 2^n$$

但是由于概率 P_k 均匀分布于区间 $[t, 1]$ ，故 $E[X_k]$ 实际上是一个条件期望，根据全期望公式，有

$$E[X_k] = E[E[X_k | P_k]] = \int_{-\infty}^{+\infty} E[X_k | P_k = p] f_{P_k}(p) dp$$

其中 $f_{P_k}(p)$ 是 P_k 的密度函数，由于 P_k 在区间 $[t, 1]$ 上均匀分布，故 P_k 是一个均匀随机变量，有

$$f_{P_k}(p) = \begin{cases} \frac{1}{1-t}, & t \leq p \leq 1 \\ 0, & p < t \text{ 或者 } p > 1 \end{cases}$$

因此有

$$f(k) = E[X_k] = \frac{1}{1-t} \int_t^1 \max(2^{n-k}, p \cdot f(k-1)) dp, f(0) = E[X_0] = 2^n$$

观察积分表达式，由于包含取最大值函数，需要根据具体情况计算，如果满足条件

$$\frac{2^{n-k}}{f(k-1)} \leq t$$

则不论 p 在区间 $[t, 1]$ 上取何值，均有 $p \cdot f(k-1) \geq 2^{n-k}$ ，故只需对一次函数 $p \cdot f(k-1)$ 进行积分，于是

$$f(k) = E[X_k] = \frac{1}{1-t} \int_t^1 p \cdot f(k-1) dp = \frac{(1+t) \cdot f(k-1)}{2}$$

若

$$\frac{2^{n-k}}{f(k-1)} > t$$

则需采取分段积分的方式进行计算，令

$$m = \frac{2^{n-k}}{f(k-1)}$$

则

^I 参阅：https://www.algorithmist.com/index.php/UVa_10900, 2020。

$$f(k) = E[X_k] = \frac{1}{1-t} \left(\int_t^m 2^{n-k} dp + \int_m^1 p \cdot f(k-1) dp \right) = \frac{2^{n-k}(m-t)}{1-t} + \frac{(1-m^2) \cdot f(k-1)}{2(1-t)}$$

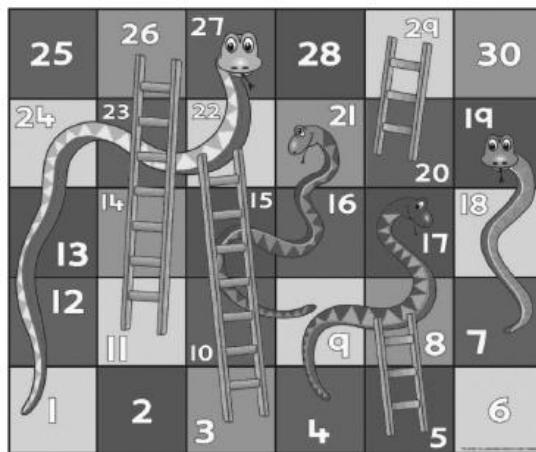
还有一类“概率—期望”问题，当前状态的期望与前置状态的期望密切相关，可以将每个状态的期望设置成一个未知数，通过构建一个多元一次线性方程组，借助高斯消元法来解方程，从而得到需要的期望。除此之外，期望还常常与动态规划发生关联，因为推导期望递推关系的过程和动态规划中推导递推关系的过程类似，如果题目中存在需要最优决策的要求，且观察题目条件满足最优子结构的性质，则问题可以归结为期望的动态规划问题。

12910 Snakes and Ladders^E (蛇梯棋)

蛇梯棋是孩子们（当然了，可爱的宠物狗也一样）非常喜欢的一种游戏。一般来说，游戏是在多个玩家之间进行的，但是 Toby 不喜欢学校里的其他小孩，他想一个人玩。游戏非常简单，棋盘高为 H 宽为 W ，Toby 从棋盘编号为 1 的方格出发，目标是编号为 $H \times W$ 的方格。

每个回合，Toby 通过抛掷一个均质的骰子来获得点数，然后在棋盘上前进点数所对应的步数。如果在回合结束时，Toby 位于某个“梯子”的底部，则他立即沿着“梯子”向上到达“梯子”的顶部，若 Toby 位于某条“蛇”的头部，则他立即沿着“蛇”向后到达“蛇”的尾部。

需要注意，均质的骰子掷出 1 到 6 点的概率是相同的。样例输入中第 3 组测试数据所对应的棋盘如下图所示，以此组测试数据来示例说明游戏如何进行。



蛇梯棋（此图对应的是样例输入的第 3 组测试数据）

当 Toby 玩游戏接近结束时，假设 Toby 位于方格 29，他掷出骰子，如果得到的是 1 点，他向前走一步到达方格 30，游戏结束，Toby 获胜；如果得到的点数是 2 点，则 Toby 向前走一步然后又退一步，到达方格 29；如果得到的点数是 3 点，则 Toby 向前走一步然后向后退 2 步，到达方格 28；如果得到的点数是 4 点，则 Toby 向前走 1 步然后向后退 3 步，到达方格 27，由于方格 27 位于“蛇”的头部，因此 Toby 沿着“蛇”到达其尾部方格 1。

现在 Toby 想知道的是在游戏结束之前需要花费多长时间，因此请求你计算赢得游戏时所经过的期望回合数（抛掷骰子的次数）。输入保证总是可以到达目标方格且最大的期望回合数不超过 100000。起始方格不会是“梯子”的底部，目标方格也不会是“蛇”的头部。

输入

输入包含多组测试数据，每组测试数据的第一行包含三个整数 W, H, S 。 W 和 H 的含义如前所述， S 表示蛇梯的数量。接着的 S 行，每行包含两个整数 u_i 和 v_i ，表示一旦位于方格 u_i ，将会立即跳转到方格 v_i 。如果 $u_i < v_i$ ，表示这是一个“梯子”，如果 $u_i > v_i$ ，表示这是一条“蛇”。输入保证对于任意 $i \neq j$ ， $u_i \neq u_j$ 且对于任意 i, j ， $u_i \neq v_j$ 。输入以文件结束符表示结束，每组测试数据之后包含一个空行。约束： $1 \leq W, H \leq 12$ ； $W \times H \geq 7$ ； $0 \leq S \leq (W \times H)/2$ ； $1 \leq u_i, v_i \leq W \times H$ 。

输出

对于每组测试数据输出一个数值，该数值表示在给定的条件下完成游戏的期望回合数。如果输出的数值和正确答案的差在 10^{-2} 以内，则结果将被认为是正确的。

样例输入

7 1 0

6 5 0

6 5 8

3 22

17 4

5 8

19 7

21 9

11 26

27 1

20 29

样例输出

6.00000000

13.04772792

19.83332560

分析

令 Toby 在方格 i 时完成游戏的期望回合数为未知数 x_i ，由于每个方格的期望回合数都可以通过关联方格的期望回合数予以表示，则根据题目约束可以得到一个多元一次方程组。以样例输入的第一组测试数据为例，考虑到抛掷骰子得到 1 到 6 点的概率均为 $1/6$ 且超过最大方格编号后的“回退”规则，可以得到以下方程组

$$\begin{cases} x_1 = 1 + \frac{x_2}{6} + \frac{x_3}{6} + \frac{x_4}{6} + \frac{x_5}{6} + \frac{x_6}{6} + \frac{x_7}{6} \\ x_2 = 1 + \frac{x_3}{6} + \frac{x_4}{6} + \frac{x_5}{6} + \frac{x_6}{6} + \frac{x_7}{6} + \frac{x_6}{6} \\ x_3 = 1 + \frac{x_4}{6} + \frac{x_5}{6} + \frac{x_6}{6} + \frac{x_7}{6} + \frac{x_6}{6} + \frac{x_5}{6} \\ x_4 = 1 + \frac{x_5}{6} + \frac{x_6}{6} + \frac{x_7}{6} + \frac{x_6}{6} + \frac{x_5}{6} + \frac{x_4}{6} \\ x_5 = 1 + \frac{x_6}{6} + \frac{x_7}{6} + \frac{x_6}{6} + \frac{x_5}{6} + \frac{x_4}{6} + \frac{x_3}{6} \\ x_6 = 1 + \frac{x_7}{6} + \frac{x_6}{6} + \frac{x_5}{6} + \frac{x_4}{6} + \frac{x_3}{6} + \frac{x_2}{6} \\ x_7 = 0 \end{cases}$$

通过使用高斯消元法求解此方程组, 解得 $x_1=6$, 则题目所求即为 x_1 的值^I。

需要注意的是, 由于“梯子”和“蛇”的存在, 某个方格的期望回合数会和超出 6 步以外的方格的期望回合数发生关联, 以样例输入的第三组数据的方格 1 为例, 有

$$x_1 = 1 + \frac{x_2}{6} + \frac{x_3}{6} + \frac{x_4}{6} + \frac{x_5}{6} + \frac{x_6}{6} + \frac{x_7}{6}, \quad x_3 = x_{22}, \quad x_5 = x_8$$

整理可得

$$6x_1 - x_2 - x_4 - x_6 - x_7 - x_8 - x_{22} = 6$$

强化练习: [11013* Get Straight^D](#), [11605* Lights Inside a 3D Grid^D](#), [11667 Income Tax Hazard \(II\)^E](#), [12230 Crossing Rivers^D](#)。

扩展练习: [10828 Back to Kernighan-Ritchie^D](#), [11291 Smeech^D](#), [12730* Skyrk's Bar^E](#)。

逃生时间问题

一个矿工在井下迷了路, 迷路的地方有三个门, 若选择走第一个门, 那么经过 3 个小时, 他能到达安全之处, 若选择第二个门, 那么经过 5 个小时, 他会回到原地。若选择走第三个门, 那么经过 7 个小时才回到原地。假定工人在任何时候都是随机地选择一个门, 问这个工人走到安全之处, 平均需要多长时间?

解 设 X 表示该矿工为到达安全之处所需的时间 (单位: 小时), 又设 Y 为他首次选择的门的号码, 则

$$E[X] = \sum_{i=1}^3 E[X|Y=i]P\{Y=i\} = \frac{1}{3}(E[X|Y=1] + E[X|Y=2] + E[X|Y=3])$$

如果矿工选择第二扇门或者第三扇门, 将分别于 5 小时和 7 小时后回到原地, 则此时问题的状态将与刚开始一样, 因此有

$$\begin{aligned} E[X|Y=1] &= 3 \\ E[X|Y=2] &= 5 + E[X] \\ E[X|Y=3] &= 7 + E[X] \end{aligned}$$

则

$$E[X] = \frac{1}{3}(3 + 5 + E[X] + 7 + E[X])$$

^I 从图论的角度, 将 Toby 位于方格 i 的期望回合数视为图的顶点, 则 Toby 的所有状态将构成一个有向图。由于“蛇”的存在和游戏规则的约束 (当前方格数加上骰子的点数超过 $H \times W$ 将回退相应的步数), 使得此有向图包含圈。令 Toby 在方格 i 时完成游戏的期望回合数为 $dp[i]$, 可以将 $dp[i]$ 表示成若干关联方格的期望回合数的和, 通过使用备忘技巧, 理论上可以求得 $dp[1]$, 进而得到解。在本题中, 由于沿着“蛇”的头部可以到达尾部, 题目约束所对应的实际上是一个有向有圈图, 常规的动态规划是在有向无圈图上进行, 有向无圈图上的动态规划满足应用动态规划的基本条件之一——无后效性, 而在有向有圈图上进行动态规划不满足“无后效性”原则, 如果不加限制, 会在动态规划过程中形成无限循环, 从而无法得到解。处理的技巧是在动态规划过程中将经过同一个状态的次数 $depth$ 也作为动态规划的一个参数, 当反复经过同一个状态使得 $depth$ 达到设定的阈值时, 就可以认为误差已经满足要求, 直接将该回溯层次状态下的期望回合数 $dp[i][depth]$ 设置为 0, 这样就相当于为无限循环设置了一个出口, 从而能够使得递归结束并得到一个满足题目精度要求的近似解。此种技巧类似于求解多元一次方程组的雅克比 (Jacobi) 迭代方法。不过本题最大的期望回合数为 100000, 通过动态规划模拟 Jacobi 迭代法很容易导致递归层次太大而造成内存溢出, 又或者递归层次太小而与正确值的差值太大从而得到错误的解, 从而导致递归的深度不易控制, 因此在此种情况下并不适用动态规划方法解题, 使用高斯消元法求解较为适宜。

解得

$$E[X] = 15$$

强化练习：10777 God! Save Me^D。

奖券收集问题

设一共有 n 种不同的奖券，假定有一人在收集奖券，每次得到一张奖券，而得到的奖券在这 n 种奖券中均匀分布，求出当这个人收集到全套 n 张奖券的时候，他收集到的奖券张数的期望值。

解 假设已经收集了 k 种不同的奖券，令 $p = k/n$ ，考虑得到一张新的奖券需要再获得额外 m 张奖券的概率，亦即额外的 m 张奖券中，前 $m-1$ 张奖券是不需要的奖券，最后 1 张是需要的奖券，不需要的奖券出现的概率为 p ，需要的奖券出现的概率为 $1-p$ ，则得到一张新的奖券需要再获得额外 m 张奖券的概率为

$$p^{m-1}(1-p)$$

根据期望的定义，得到新的奖券所需要的额外奖券张数的期望

$$M = \sum_{m=1}^{\infty} mp^{m-1}(1-p) = (1-p)(1 + 2p + 3p^2 + 4p^3 + \dots)$$

令

$$S = 1 + 2p + 3p^2 + 4p^3 + \dots$$

则

$$pS = p + 2p^2 + 3p^3 + 4p^4 + \dots$$

两式相减得

$$(1-p)S = 1 + p + p^2 + p^3 + \dots = \frac{1}{1-p}$$

则有

$$M = (1-p)S = \frac{1}{1-p} = \frac{n}{n-k}$$

因此总的奖券张数期望为

$$N = \sum_{k=0}^{n-1} \frac{n}{n-k} = n \left(1 + \frac{1}{2} + \dots + \frac{1}{n} \right)$$

强化练习：10288 Coupons^C。

6.9 博弈论

博弈论（game theory）是一门关于决策的数学理论分支，在经济学的理论模型研究中应用非常广泛。博弈论本身的内容很多，但绝大多数并不便于使用编程竞赛的方式予以体现，因此在实际的竞赛中，考察内容主要集中在无偏博弈（impartial game）的胜负状态判断这一方面。

无偏博弈是指满足下列条件的博弈（游戏）：

- (1) 两名玩家轮流进行游戏操作直到游戏结束；
- (2) 当某名玩家按照游戏规则无法进行下一步操作时，游戏结束，赢家产生（或出现平局）；
- (3) 玩家在进行游戏时，每一次游戏操作的可选择性是有限的；例如，在 Nim 游戏中，玩家需要取走至少一枚石子，但所取石子数受限于其选择的石子堆中石子的总数；
- (4) 所有操作对于双方玩家来说都是对等的，同时双方玩家均可观察到游戏当前状态的一切信息，亦即信息公开（perfect information），玩家仅仅是先手与后手的区别；

(5) 所有游戏操作是确定性的 (deterministic)，不存在随机性；

按上述条件限制，Nim 游戏是无偏博弈，中国象棋和国际象棋不是无偏博弈，因为在象棋游戏中，玩家只能移动己方棋子，其他的类似于扑克、色子类游戏也不属于无偏博弈，因为它们具有随机性。

策梅洛^I于 1913 年证明，对于无偏博弈，如果不会出现平局，则给定某种初始的游戏状态，要么先手具有必胜策略，要么后手具有必胜策略^[59]。策梅洛证明的核心思想是对于无偏博弈来说，游戏可能出现的状态是有限的，在游戏的每一步，玩家都通过选择一步操作而减少了剩余游戏状态的数量，因此游戏必定在有限个步骤内结束，因此可以由最终状态逆向推导出前置状态的胜负。

本节首先介绍经典的 Nim 游戏，然后由 Nim 游戏引出 Sprague-Grundy 定理，再介绍由 Nim 游戏变形和扩展得到的其他游戏的胜负态分析，最后介绍类 Nim 游戏的 PN 态分析策略。

6.9.1 Nim 游戏

Nim 游戏^{II} (Nim game，中文翻译为“拈”游戏) 是一种非常有趣的数学博弈游戏，其规则为：有若干堆石子 (石子也可由其他物品代替)，每堆石子的数量有限但不固定，两名玩家轮流从某个石子堆中取至多一枚石子。进行某次取石子操作时，一旦选择从某个石子堆中拾取石子，则不能再从其他石子堆拾取，最后将石子取完的玩家判定为胜。

初看哪个玩家获胜纯属运气成分，但实际上该游戏遵循一个确定的规律。我们从最简单的情形开始，看是否可以从中找出一些规律。为便于说明，约定先进行游戏操作的玩家为先手，下一轮进行游戏操作的玩家为后手。

(1) 假设只有一堆石子，那么很显然，先手必胜，因为他 (她) 可以选择一次性把所有石子取走。

(2) 如果有两堆石子，若两堆石子的数量不同，则先手可以从数量较大的石子堆中取出若干石子，使得剩下的两堆石子数量相同，之后使用模仿策略即可获胜。使用模仿策略，在很多游戏中也是一种有效的取胜策略。模仿策略本质上是游戏中存在可逆操作 (reversible moves)，通过可逆操作可以使得当前玩家“逆转”上一玩家对局势进行的改变，从而使得局势恢复到对当前玩家有利的状态。例如，在《萨姆·劳埃德的数学趣题》^[60]一书中，有一则关于航海家哥伦布^{III}的趣题：给定一张矩形的餐巾纸和足够数量且大小相同的熟鸡蛋，两个玩家轮流在餐巾纸上放鸡蛋，最后一个放鸡蛋的玩家获胜。如果先手不加考虑使用模仿策略，未在关键位置放置鸡蛋，后手只需根据中心对称原则，以矩形的中心为对称点，在先手放置鸡蛋的对称位置按同样方式放置一枚鸡蛋，这样一来，先手必输无疑。但是，如果先手在餐巾纸的中央先将一枚鸡蛋一端朝下，磕破鸡蛋壳让它立起来，先行占据中央位置，从而改变先后手的顺序，就能模仿后手放置鸡蛋的位置，使得总是关于中心对称放置鸡蛋，这样就能够使得先手必胜。同样的，在 Nim 游戏中，先手通过取出若干石子使得两堆石子数量相同，之后先手可以模仿后手的操作——后手从某个石子堆取 x 枚石子，则先手从另外一个石子堆取 x 枚石子。这样先手总是最后一个取石子的人，因此必胜。

(3) 如果两堆石子数量相同，那么根据前述分析，先手不论采取什么策略都将使得两堆石子的数量不同，此时后手可以使用前述介绍的策略成为必胜玩家。

(4) 再增加一点思考的难度，如果是三堆石子，情况是怎样呢？按照前述的分析，先手似乎应该尽量

^I 恩斯特·弗雷德里克·费迪南·策梅洛 (Ernst Friedrich Ferdinand Zermelo, 1871—1953)，德国人，逻辑学家、数学家。

^{II} Nim 游戏源于德国的 Nimm! 游戏，Nimm 在德语中含义为 “Take”，即“取”。

^{III} 克里斯托弗·哥伦布 (Christopher Columbus, 约 1451—1506)，意大利探险家、航海家、殖民者。

使得剩下的局面是两堆相同数量的石子，以便应用模仿策略。如果三堆石子中有两堆石子的数量相同，那么先手必胜，因为先手可以取掉两堆数量相同的石子之外的一堆石子，使得剩下的局面是两堆相同数量的石子，这样就可以利用模仿策略，最终能够获得胜利。

但如果三堆石子的数量均不相同，那么究竟应该如何取才能保证必胜呢？由于可做的选择增多，似乎难以得到一个有效的策略。分析到这里，似乎有些许规律可循——胜败的关键好像是和石子堆的某种“平衡”有关，但确切的规律不易得出，随着石子堆数的增加，分析难度逐渐加大。由此可见，破解 Nim 游戏的思维难度不是一般人所能轻易超越的。也正因为如此，Nim 游戏的完整解决方案在游戏出现约 100 年后才出现。1901 年，数学家布顿¹深入研究了 Nim 游戏，并从数学上给出了一个简单而优美的结论^[61]。在布顿所发表的论文中，他将某种石子数量的组合状态称之为“safe combination”——如果先手能够通过取子操作将石子数量置于此种状态下，则在后续游戏过程中，先手按最优策略进行取子，后手不可能获胜。在这里，不妨将“safe combination”意译为“安全态”。那么安全态如何确定呢？假设有三堆石子，数量分别为 9, 5, 12，将其表示为 4 位二进制数，按次序排列如下：

$$\begin{array}{rccccc} 9 & \rightarrow & 1 & 0 & 0 & 1 \\ 5 & \rightarrow & 0 & 1 & 0 & 1 \\ 12 & \rightarrow & 1 & 1 & 0 & 0 \end{array}$$

可以看到，从二进制表示的第 0 位到第 3 位，每列上 1 的个数总和都是 2 的倍数，则 9, 5, 12 的石子数量组合就是一个安全态。接着布顿证明了以下两个结论：

- (1) 假设先手通过取子操作留下了一个安全态，则后手不论进行何种操作，留下的局面都不可能是安全态。
- (2) 假设先手通过取子操作留下了一个安全态，后手经过取子操作使得局面变为不安全态，先手总是能够找到一种取子方法，将后手留下的局面恢复为安全态。

在安全态中，要求石子数量的二进制数各位上 1 的数量总是偶数，朴素的方法是将各个数字解析为二进制数再逐位统计位为 1 的个数。不过，存在更为巧妙的方法——异或运算。根据异或运算的规则，如果两个位相异，则结果为 1，否则为 0。由于安全态中各个位上 1 的个数为偶数，某位上连续异或的结果为 0，所有位连续异或的结果同样为 0。也就是说，只需将表示石子数量的所有数进行异或运算，如果结果为 0，则是一个安全态，如果不为 0，则是一个非安全态。更进一步，如果初始局面是一个不安全态，则先手总能够通过某种取子策略使之成为安全态，因此先手具有必胜策略，反之后手具有必胜策略。正因为异或运算和 Nim 游戏联系如此紧密，在博弈论中，异或运算有另外一个名称——“Nim 加”运算。

如果初始给定的局面是不安全态，抑或后手通过取子操作将 Nim 游戏变为不安全态，先手应该如何操作才能将其恢复为安全态呢？还是通过异或运算。根据前述讨论，先手的目标是通过取子操作使得异或结果变为 0，观察异或结果二进制表示最高位的 1，选取石子数二进制表示对应位也为 1 的某堆石子，从中取走一部分石子，使得此位变为 0，而且异或结果中其余为 1 的位也发生反转，那么就可以达到最终的异或结果为 0 的目的，从而成为安全态^[62]。

下面，通过示例来说明如何进行操作。假设有三堆石子，每堆石子的石子数目分别为 14, 79, 246，将其转换为二进制数，分别为：

¹ 查尔斯·莱昂纳德·布顿 (Charles Lenard Bouton, 1869—1922)，美国数学家。

14	→	0	0	0	0	1	1	1	0
79	→	0	1	0	0	1	1	1	1
246	→	1	1	1	1	0	1	1	0
⊕	—	—	—	—	—	—	—	—	—
		1	0	1	1	0	1	1	1

容易知道，三堆石子数目的异或值为 10110111_2 ，是一个不安全态。异或结果最高位的 1 位于序号为 7 的二进制位（从二进制数的右侧开始计数，第一个二进制位的序号为 0），观察石子数的二进制表示，序号为 7 的二进制位为 1 的是第三堆石子，因此需要从第三堆石子取走适量的石子，使得最终的异或结果为 0，使得当前状态转变为安全态。将三堆石子数目的异或值与表示第三堆石子数目的二进制数再进行一次异或，可以得到异或值 01000001_2 ，对应十进制数 65，这表明需要将第三堆的石子数目变成 65 才能使得最终的异或结果为 0，因此，需要从第三堆石子中取走 $246 - 65 = 181$ 颗石子。

14	→	0	0	0	0	1	1	1	0
79	→	0	1	0	0	1	1	1	1
65	→	0	1	0	0	0	0	0	1
⊕	—	—	—	—	—	—	—	—	—
		0	0	0	0	0	0	0	0

强化练习：10165 Stone Game^B。

扩展练习：11859 Division Game^D。

将 Nim 游戏的规则稍加修改，即最后取石子的玩家判定为负（misère Nim game，反 Nim 游戏），那么游戏的必胜策略将不再相同。布顿在论文中也研究了此种情形。我们来看看比较简单的情形，检查一下必胜策略发生了何种变化。

(1) 假设只有一堆石子，如果只有一枚石子，由于先手必须先取石子，则先手为负。如果石子数量为 $n > 1$ ，则先手可以取 $n - 1$ 枚石子，留下一枚石子，则先手必胜。根据安全态的定义，只有一堆石子的情况下，不论石子数量多少，均为不安全态，在正常的 Nim 游戏中，先手是必胜的，而在反 Nim 游戏中，只有一枚石子使得先手必败。

(2) 假设有两堆石子，则有几种情况：(a) 两堆石子数量均为 1，此时先手必胜；(b) 石子数量为 1 和 $n > 1$ ，先手也必胜；(c) 两堆石子数量均为 $C > 1$ ，则先手必败，因为后手可使用模仿策略取得必胜。将两堆石子标记为 A 和 B ，若先手将 A 堆石子全部取走，则后手可取走 B 堆的 $C - 1$ 枚石子，使得先手必败；若先手取走 A 堆的 $C - 1$ 枚石子，则后手将 B 堆 C 枚石子全部取走，同样使得先手必败；若先手取走 A 堆中 $x < C - 1$ 枚石子，则后手模仿先手取走 B 堆石子中的 x 枚石子，使得剩余的两堆石子数量仍然相同，但都大于 1，这样仍会使得先手必败。(d) 两堆石子数量均大于 1，但不相等，则先手必胜，因为先手可将数量较多的石子堆取走一部分，使得剩余两堆石子的数量相同，之后使用前述的第 (c) 种情形的策略即可获胜。

(3) 假设有三堆石子，在游戏进行到最后，如果先手能够获胜，则先手留下的局面应该是 $(1, 1, 1)$ 或者 $(1, 0, 0)$ ，在这两种情况下，后手将不得不拾取最后一枚石子。如果先手留下局面 $(1, 1, 0)$ ，则后手可以通过拾取一枚石子使得先手为负。为了能够留下局面 $(1, 1, 1)$ 或者 $(1, 0, 0)$ ，则先手不应该留下 $(1, 1, n > 1)$ 或者 $(1, n > 1, 0)$ 的局面，否则先手将必败，因为后手可从大于 1 的石子堆中取出若干石子，使得局面变成 $(1, 1, 1)$ 或者 $(1, 0, 0)$ ，这样先手就不得不拾取最后一枚石子，从而必败。

综合以上简单情况下的情形，可以得到下述结论：前述的安全态定义不变，则哪个玩家先达到安全态就是必胜者，除了下述两种特殊情形——石子的堆数为奇数，且所有石子堆的数量均为 1，此种情形为安全态；若石子的堆数为偶数，且所有石子堆的数量均为 1，不是安全态。也就是说，对于 $(1, 1, 1, 1, 1)$ 这样

的局面，虽然异或结果为不为 0，但是由于包含奇数堆数量为 1 的石子堆，属于安全态，按反 Nim 游戏规则，先手将此局面留给后手，则先手必胜（这很容易理解，因为两个玩家每次只能取一枚石子，后手必定是取最后一枚石子的玩家）；而对于 (1, 1, 1, 1) 这样的局面，虽然异或结果为 0，但是由于包含偶数堆数量为 1 的石子堆，属于不安全态，按反 Nim 游戏规则，若先手将此局面留给后手，则先手必败。

按照反 Nim 游戏规则，如何确定在给定初始局面时，先手和后手的胜负情况呢？根据前述的结论，需要根据石子数量的异或结果以及只包含 1 枚石子的石子堆数这两个变量来考虑。假设至少有一个石子堆所包含的石子不为 1，胜负情况的判定和正常 Nim 游戏是相同的。若所有石子堆均只包含 1 枚石子，则奇数个这样的石子堆对于先手来说是必败局面，反之，偶数个这样的石子堆对于先手来说是必胜局面。

强化练习：[1566 John^E](#)。

6.9.2 Sprague-Grundy 定理

实际上，可以将前述的 Nim 游戏一般化，得到一个处理此类问题的一般原则，即应用 Sprague-Grundy 定理来对一般化的 Nim 问题及其扩展问题进行判断。Sprague-Grundy 定理证明了这样的一个结论^{[63][64]}：对类似于 Nim 游戏的其他无偏博弈游戏形式，可以将其分解为一系列的子游戏，每个子游戏可以将其状态归结为一个 Grundy 值（或称 Nim 值，Nimber），初始游戏胜负的判定可以归结为各个子游戏异或结果的判定，如果子游戏 Grundy 值的异或结果不为 0，则对某个玩家具有必胜策略，否则对另外一个玩家有必胜策略。那么如何计算 Grundy 值呢？我们以 Nim 游戏的一个变形为例予以介绍。假设在取石子的过程中，每次只能取 a_1, a_2, \dots, a_k 枚石子（ a_i 中一定有一个为 1， $1 \leq i \leq k$ ，以便游戏能够终止），考虑具有 x 枚石子的某个石子堆，该石子堆的 Grundy 值就是——**此石子堆任意一步所能转移到的状态的 Grundy 值之外的最小非负整数**。定义 mex（minimal excludant）运算，这是施加于一个集合的运算，表示最小的不属于这个集合的非负整数，则 x 个石子的 Grundy 值为

$$sg(x) = \text{mex}\{sg(y), y \text{ 是 } x \text{ 的后继}\}$$

因为只从字面上不易理解，下面仍以 Nim 游戏为例解释如何计算 Grundy 值。在 Nim 游戏中，按照游戏规则，从包含一颗石子的石子堆中只能取 1 颗石子，则剩余的石子数量为 0，即石子堆的状态从 1 颗转移到 0 颗，则按照前述 Grundy 值的定义，包含 1 颗石子的石子堆的 Grundy 值 $sg[1]$ 为除 $sg[0]$ 外的最小非负整数，由于 0 颗石子的石子堆在 Nim 游戏中无法再转移到其他状态，故定义 $sg[0] = 0$ ，则 $sg[1] = 1$ 。类似的，包含 2 颗石子的石子堆可以转移到包含 1 颗或 0 颗的石子堆，则 $sg[2]$ 为除 $sg[1]$ 和 $sg[0]$ 之外的最小非负整数，于是 $sg[2] = 2$ 。依此类推，可得 $sg[x] = x$ ，则最后 Nim 游戏的胜负可以由各个石子堆的 Grundy 值 $sg[x]$ 的异或结果决定，而 $sg[x] = x$ ，等同于各个石子堆中包含的石子数数值进行异或。

从 Nim 游戏的 Grundy 值计算过程可以看到，Grundy 值的计算具有递归性质，可以使用动态规划的备忘技巧按下述方式进行计算 Nim 游戏中各个石子堆的 Grundy 值¹：

```
//-----6.9.2.1.cpp-----//
const int MAXN = 1000;
int sg[MAXN];

// 使用下述递归过程计算 Grundy 值之前需要将数组 sg 全部初始化为 -1。
// memset(sg, -1, sizeof(sg));
```

¹ 有关动态规划中备忘技巧的应用，请读者参阅本书第 11 章“动态规划”的内容。

```

int grundy(int x)
{
    if (~sg[x] >= 0) return sg[x];
    set<int> s;
    for (int i = 1; i < x; i++)
        s.insert(grundy(x - i));
    int g = 0;
    while (s.find(g) != s.end()) g++;
    return sg[x] = g;
}
//-----6.9.2.1.cpp-----

```

在计算得到各个石子堆石子数 x_i 所对应的 Grundy 数后, 对其按类似于 Nim 加的操作进行异或运算, 最后即可根据结果来判断先手的胜负情况。

需要注意的是, 上述使用递归的计算方法, 只有当石子数量较小时才能进行有效计算, 对于石子数较大的情形, 计算效率不高。如果只有一堆石子, 且石子数量在 10^6 级别, 可按照 PN 态分析, 使用动态规划予以解决。按照游戏规则, 当最后没有石子可取时, 先手为负, 也就是说在有 a_1, a_2, \dots, a_k 枚石子时, 先手是必胜的, 那么可以类推得到如下的结论: (1) 如果对于 $a_i, 1 \leq i \leq k$, $x - a_i$ 是必败态的话, x 就是必胜态。反过来, (2) 如果对于 $a_i, 1 \leq i \leq k$, $x - a_i$ 是必胜态的话, x 就是必败态。由于已经知道当剩余 0 枚石子时先手是必败的, 则可以使用动态规划算法自底向上计算各种石子数量时先手的胜败状态。

```

//-----6.9.2.2.cpp-----
const int MAXN = 1000010;
int main(int argc, char *argv[])
{
    int x, k, a[16];
    bool win[MAXN];
    while (cin >> x) {
        cin >> k;
        for (int i = 0; i < k; i++) cin >> a[i];
        win[0] = false;
        for (int i = 1; i <= x; i++) {
            win[i] = false;
            for (int j = 0; j < k; j++)
                win[i] |= ((a[j] <= i) && !win[i - a[j]]);
        }
        if (win[x]) cout << "The first player wins.\n";
        else cout << "The second player wins.\n";
    }
    return 0;
}
//-----6.9.2.2.cpp-----

```

强化练习: 1482* Playing With Stones^D, 10404 Bachet's Game^A。

6.9.3 Nim 游戏和 Sprague-Grundy 定理扩展

在理解 Nim 游戏胜负判定和掌握 Sprague-Grundy 定理后, 我们来看看如何处理由基本的 Nim 游戏衍生出的变形和扩展问题^{[65][66][67]}。

Tic-Tac-Toe

经典的 Tic-Tac-Toe 游戏是在 3×3 的二维棋盘上进行, 按照前述给出的无偏博弈的定义, 该游戏不属于无偏博弈, 因为游戏双方执子不同, 一方为 ‘X’, 一方为 ‘O’。如果将规则更改, 双方均执 ‘X’ (或 ‘O’),

则游戏为无偏博弈。在新规则下, 给定某种游戏的状态, 必定有一方玩家具有获胜 (或平局) 的策略, 具体可以使用后续介绍的博弈树 PN 状态分析方法予以解决。

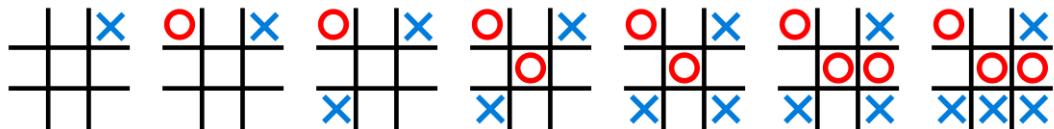


图 6-7 二维 Tic-Tac-Toe 游戏一种可能的着子过程

此处讨论将棋盘展开为一维, 并规定双方均执 ‘X’ 子的情形。设棋盘为 $1 \times N$ 的方格, 其中 $N \geq 3$, 双方轮流在棋盘上放置 ‘X’ 棋子。如果某个玩家着子后使得棋盘上出现 3 个 ‘X’ 棋子相邻的情形, 则判定该玩家获胜。给定游戏的某个中间状态, 此时棋盘上有些位置已经放置了 ‘X’ 棋子, 但未出现相邻三个棋子均为 ‘X’ 的情形, 请确定该游戏中间状态是否可能具有必胜策略, 如果有的话, 下一步在哪个方格着子才能保证有必胜策略?



图 6-8 一维 Tic-Tac-Toe 游戏的一种局面

更改规则后, 可知其符合无偏博弈的定义, 因此必定有一方具有必胜策略。可以应用 Sprague-Grundy 定理解决此问题。要应用 Sprague-Grundy 定理解决问题, 需要确定什么是初始状态, 什么是终止状态, 并且确定子游戏状态。从图 6-8 可以看到, 两个“近邻”的 ‘X’ 包围了连续的空格, 玩家在此空格中落子, 落子后相当于将原有的连续空格划分为两个新的部分, 后续操作可以在这两个新的部分中继续进行, 因此可以将连续的空白视为一个子游戏。容易得知, 先手在一个位置放置棋子后, 放置棋子的前后各两个位置将成为“禁区”, 如果后手在“禁区”内放置棋子会导致后手必败, 如图 6-9 所示。



图 6-9 禁区

因此对于一个长度为 n 的连续空白区域, 只有 $n-4$ 个空格可以放置棋子 (对于出现在棋盘起始和结尾的连续空白区域, 只有 $n-2$ 个方格可以放置棋子)。在剔除禁区后, 如果连续的空白方格数为 m , 则在这 m 个方格中的任何一个方格落子会将其划分为两个部分, 假设在第 i 个方格落子, 考虑禁区的影响, 此时前半部分为 $i-3$ 个方格, 后半部分的有效方格数为 $m-i-2$ 个方格。也就是说, 从初始的 m 个有效方格, 在第 i 个方格落子后会变成两个独立的状态, 设这两个独立状态的 Grundy 值分别为 $grundy[i-3]$ 和 $grundy[m-i-2]$, 则在第 i 个方格落子后的状态的 Grundy 值 $grundy[i]$ 为 $grundy[i-3]$ 与 $grundy[m-i-2]$ 的异或, 根据定义初始状态 m 个方格的 $grundy[m]$ 为

$$grundy[m] = \text{mex}\{grundy[\max(0, i-3)] \wedge grundy[\max(0, m-i-2)] | 1 \leq i \leq m\}$$

边界条件: $grundy[0]=0$ 。当 m 值较小时, 使用前述介绍的递归方式计算 Grundy 值, 其效率尚可接受, 但是当 m 较大时使用上述递推式进行递归计算效率较低, 可以改用直接递推的方式进行计算。

```

//-----6.9.3.cpp-----
int sg[10001], visited[512];
void grundy()
{
    sg[0] = 0, sg[1] = sg[2] = sg[3] = 1;
    for (int i = 4; i <= 10000; i++) {
        memset(visited, 0, sizeof(visited));
        for (int j = 3; j <= 5; j++)
            if (j <= i)
                visited[sg[i - j]] = 1;
        for (int j = 1; j < i - 5; j++) visited[sg[j] ^ sg[i - j - 5]] = 1;
        for (int j = min(5, i); j >= 3; j--) visited[sg[i - j]] = 1;
        for (int j = 0; ; j++)
            if (!visited[j])
                sg[i] = j;
                break;
    }
}
//-----6.9.3.cpp-----

```

此外，需要注意的是，使用前述的 Grundy 值判断胜负还需注意边界情况，如果通过放置一枚‘X’棋子立即获胜，则不考虑 Grundy 值和异或值是否为 0，只有在一步以内不能获胜的情形下使用 Grundy 值进行判断。

11534 Say Goodbye to Tic-Tac-Toe^D (和井字游戏说再见)

Alice 对井字游戏 (Tic-Tac-Toe) 已经有些厌烦了，她经常在电脑上玩这个游戏。她之所以感到厌烦是因为在这个游戏上她已经是专家级别，她总是能够和电脑打成平手。除此之外，Alice 对“成双成对”的东西感到厌倦，这意味着她不想再看到两个相邻的‘X’或者两个相邻的‘O’。为此，Bob 为 Alice 创造了一款新的电脑游戏。以下是这款两人电脑游戏的规则：

- 游戏在一个 $1 \times N$ 的网格上进行；
- 每一个回合，相应玩家可以对未被标记的方格标记‘X’或‘O’；
- 在游戏中不能出现相邻的两个方格都标记为‘X’的情形；
- 在游戏中不能出现相邻的两个方格都标记为‘O’的情形；
- 标记最后一个方格的玩家获胜。

下面以 $N=3$ 为例来说明游戏如何进行。假设第一个玩家将最左边的方格标记为‘X’，那么网格看起来如下图所示：



从这个状态开始，第二名玩家的必胜策略是将最右边的方格标记为‘O’，如下图所示：



按照规则，不能出现两个相邻的‘X’或两个相邻的‘O’，而第一个方格已经标记为‘X’，第三个方格已经标记为‘O’，那么后续第一名玩家将无法继续进行标记，因此第二名玩家将获胜。但是上述操作并不是第一名玩家的最优操作策略，假如第一名玩家在最开始的时候将中间的那个方格标记为‘X’或者‘O’，那

么第一名玩家将获胜。

Alice 总是第一个进行标记，而且经常是和 Bob 玩这个游戏，Bob 在进行若干标记操作后可能会离开，将游戏交由电脑代理，电脑总是按照最优策略对方格进行标记。

给定 Bob 离开后的游戏状态，你的任务是确定 Alice 是否可能在对阵电脑时获得胜利。

输入

输入第一行包含一个整数 T ($T < 10001$)，表示测试数据的组数。接着的 T 行每行包含一个长度为 N ($0 < N < 101$) 的字符串，表示 Bob 离开后的游戏状态，字符串中除了包含 ‘X’ 和 ‘O’ 之外，还包含字符 ‘.’，表示尚未标记的方格。

输出

对于每组测试数据，如果 Alice 能够获胜，输出 “Possible.”，否则输出 “Impossible.”。

样例输入

```
3
...
X..
0
```

样例输出

```
Possible.
Impossible.
Possible.
```

分析

给定一个 $1 \times N$ 的空白网格，选择网格中的任意一个方格进行标记之后，其效果相当于将网格分成了左右两个部分，如果标记的是 ‘X’，那么左边部分的最右侧一个空格不能标记 ‘X’，右边部分最左侧的一个空格不能标记 ‘X’，若标记的是 ‘O’，情况类似。

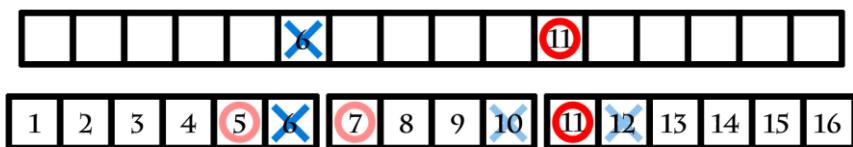


图 6-10 当 $N=16$ 时，Alice 在方格 6 标记 ‘X’，Bob 在方格 11 标记 ‘O’，然后 Bob 离开。此时游戏将被划分为三个部分：(a) 方格 1 至方格 5 为一个子游戏；(b) 方格 7 至方格 10 为一个子游戏；(c) 方格 12 至方格 16 为一个子游戏。子游戏 (a) 包含 5 个方格，相当于 $N=5$ ，但是增加了限制条件：最右侧一个方格只能标记 ‘O’，而其他方格则无限制；子游戏 (b) 包含 4 个方格，相当于 $N=4$ ，其最左侧方格只能标记 ‘O’，最右侧方格只能标记 ‘X’；子游戏 (c) 包含 5 个方格，相当于 $N=5$ ，其最左侧方格只能标记 ‘X’，其他方格的标记则无限制。根据 Sprague-Grundy 定理，整个游戏的胜负由子游戏 (a)、(b)、(c) 的 Grundy 值的异或结果决定

那么可以将标记后所产生的左侧和右侧的未标记部分视为子游戏，根据 Sprague-Grundy 定理，整个游戏的胜负由玩家进行操作后所产生的子游戏决定。根据游戏规则，不能出现两个相邻的 ‘X’ 或者两个相邻的 ‘O’，那么每一个子游戏由网格的长度以及左右两端方格能够进行的标记种类确定。网格的一端可以是已标记 ‘X’ 或 ‘O’ 或未标记，则对于长度为 n 的网格来说，可能具有 9 种不同的状态。

在具体实现中还需注意两个细节^I：(1) Bob 离开时的游戏状态可能是轮到 Alice 进行标记操作，或者是轮到 Bob 进行标记操作，如果是轮到 Alice 进行标记操作，若游戏的 Grundy 值为 0，则 Alice 能够获胜，如果是轮到 Bob 操作，游戏的 Grundy 值为 0，则 Alice 将无法获胜，因此需要计数游戏已经进行的回合数，根据回合数确定当前进行标记的玩家，进而确定 Alice 是否能够获胜。(2) 当网格的长度 $n=0$ 时，玩家将无法进行标记操作，也就是说，该游戏状态无法继续变化为其他游戏状态，因此应当定义当网格的长度 $n=0$ 时的 Grundy 为 0，而不论其两端能够进行何种标记。

参考代码

```

const int MAXN = 110;

string grid;
int T, sg[MAXN][3][3];
map<char, int> key = {{'X', 1}, {'O', 2}};

int dfs(int n, int l, int r)
{
    if (~sg[n][l][r]) return sg[n][l][r];
    if (n == 0) return sg[n][l][r] = 0;
    int visited[MAXN] = {};
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= 2; j++) {
            if (i == 1 && j == 1) continue;
            if (i == n && j == r) continue;
            int grundy = dfs(i - 1, l, j) ^ dfs(n - i, j, r);
            visited[grundy] = 1;
        }
    for (int g = 0; ; g++) if (!visited[g]) return sg[n][l][r] = g;
}

int main(int argc, char *argv[])
{
    memset(sg, -1, sizeof(sg));
    cin >> T;
    while (T--) {
        cin >> grid;
        int l = 0, r = 0, moves = 0, n = 0, grundy = 0;
        for (auto c : grid) {
            if (c == '.') n++;
            else {
                r = key[c];
                grundy ^= dfs(n, l, r);
                l = r, n = 0, moves++;
            }
        }
        grundy ^= dfs(n, l, 0);
        if (moves & 1) grundy = !grundy;
        cout << (grundy ? "Possible." : "Impossible.") << '\n';
    }
}

```

^I 本题的参考实现应用了回溯法和动态规划技巧。关于“回溯法”，请参阅本书第8章“回溯法”内容；关于“备忘技巧”，请参阅本书第11章“动态规划”中第11.2节“备忘”的内容。如果读者没有上述两方面的知识储备，建议在学习这两章后再回过头来理解此节的内容。

```

    }
    return 0;
}

```

强化练习：[10561 Treblecross^D](#)，[11840 Tic-Tac-Toe^D](#)。

扩展练习：[1378* A Funny Stone Game^{E\[68\]}](#)。

阶梯 Nim 游戏

阶梯 Nim 游戏 (staircase Nim game) 的规则如下：给定一个 n 级的阶梯，序号为 $0, 1, \dots, n-1$ ，每级阶梯上放置有若干数量的硬币，玩家 A 和玩家 B 轮流从第 i 级阶梯上取若干硬币，放置到第 $i-1$ 级阶梯上， $n \geq 2$ 且 $i \geq 1$ ，无法完成操作的玩家输掉游戏。假设玩家 A 和玩家 B 均按照最优策略进行操作，玩家 A 是否具有必胜策略呢？

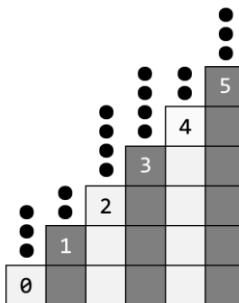


图 6-11 阶梯 Nim 游戏 ($n=6$)

之所以称为阶梯 Nim 游戏，显然和 Nim 游戏有关，那么它是如何与 Nim 游戏发生关联的呢？观察阶梯，如果从 0 开始为阶梯进行计数，可以将阶梯分为两种类型：偶数级的阶梯—— $0, 2, \dots, 2j$ 以及奇数级的阶梯—— $1, 3, \dots, 2j+1$ ，其中 $j \geq 0$ 。如果玩家 A 从偶数级阶梯取若干硬币放置到奇数级阶梯上，玩家 B 可以将这些硬币再次转移到下一个偶数级阶梯上，导致玩家 A 的操作形同无效。例如，如图 6-11 所示，玩家 A 将第 2 级阶梯上的 2 个硬币移动到第 1 级阶梯上，则玩家 B 可以选择将移动到第 1 级阶梯上的 2 个硬币“原样”移动到第 0 级阶梯上。换句话说，任意一个玩家从偶数级阶梯上移动硬币到奇数级阶梯上是无效的操作，因为后续玩家可以利用“对称性”策略将移动到奇数级阶梯上的硬币再次移动到下一级的偶数级阶梯上。因此可以得出以下结论：偶数级阶梯上的硬币数量对游戏的输赢不产生影响。相反的，从奇数级阶梯向偶数级阶梯转移硬币却不具有这种“对称性”策略。当玩家 A 从奇数级阶梯上移动若干硬币到偶数级阶梯上时，玩家 B 并不能总是能够按照相同的方法将游戏状态恢复到原样，因为游戏在最终状态时，所有硬币必定在第 0 级阶梯上，而在第 0 级阶梯时，已经是最低一级的阶梯，无法再向一级阶梯移动。

从以上的分析可知，只有奇数级阶梯上的硬币数量会对游戏的输赢产生影响，而偶数级阶梯上的硬币数量则无关紧要。从奇数级阶梯向偶数级阶梯移动硬币时，其最终效果等同于在常规 Nim 游戏中将其“取走”（不是真正的取走，这些硬币仍然在偶数级阶梯上，只不过不会对后续的游戏输赢再产生影响），那么问题转化为对奇数级阶梯上的硬币进行常规的 Nim 游戏，按 Nim 游戏的胜负判断规则，只需对奇数级阶梯上的硬币数量进行异或即可。

扩展练习：[12499* I am Dumb 3^E](#)。

Nim-K 游戏

Moore 对 Nim 游戏进行了一般化并提出了 Nim-K 游戏^[69]。原有的 Nim 游戏规定只能对 1 堆石子进行操作，Moore 放宽了限制，可以对至少 1 堆，至多 k 堆石子进行操作，从这些选定的石子堆中移除任意数目的石子。最后移除所有石子的玩家获胜。

6.9.4 PN 态分析

在公平的组合游戏中，可以把所有可能出现的状态视为图中的顶点，如果从一个状态可以通过一步操作转移到另外一个状态，则在两个顶点间具有一条有向边，将所有状态和其可达状态间连接有向边后所得到的即为状态图。如果游戏不会出现平局，即状态图是有向无圈图，则所有的状态可以分为两种——P 态 (P-position) 和 N 态 (N-position)。P 态表示该状态是前一位玩家能够获胜的局势 (Previous player winning position)，其含义是：假设双方玩家都采取最佳策略，则某个玩家从 P 态开始玩游戏必定无法获胜，亦即 P 态对当前玩家来说是必败态。而 N 态表示该状态是下一位玩家能够获胜的局势 (Next player winning position)，其含义是：假设双方玩家都采取最佳策略，某个玩家从 N 态开始玩游戏必定能够获胜，亦即 N 态对当前玩家来说是必胜态。在大部分的游戏中，按照游戏规则，终止状态均为 P 态 (必败态)。对于任意一个 P 态 (必败态)，它要么是一个终止状态，要么它所有可以转移到的状态都是 N 态 (必胜态)，而对于任意一个 N 态 (必胜态)，它至少有一个后继状态是 P 态 (必败态)。以 Nim 游戏为例，当给定的石子数目为 (0, 1, 1) 时，其异或值为 0 时，则此状态是一个 P 态，即对于当前玩家来说是一个必败态，而当给定的石子数目为 (1, 1, 1) 时，其异或值为 1，则此状态是一个 N 态，即对于当前玩家来说是一个必胜态。在某些组合游戏的分析中，不需要大费周章地进行回溯分析，只需寻找其中的 P 态和 N 态互相转换的规律即可确定必胜策略。

847 A Multiplication Game^A (乘法游戏)

Stan 和 Ollie 正在玩一种与乘法有关的游戏。给定一个整数 n , $1 < n < 4294967295$, 从 $p=1$ 开始，两名玩家轮流将 p 乘以 2 到 9 之间的一个整数，首先使得 $p \geq n$ 的玩家获胜。游戏总是先从 Stan 开始。

输入与输出

输入中每行包含一个整数 n , 对于每行输入输出一行。假定两名玩家均采取最优策略，如果 Stan 获胜，输出 “Stan wins.”，否则输出 “Ollie wins.”。

样例输入

```
162
17
34012226
```

样例输出

```
Stan wins.
Ollie wins.
Stan wins.
```

分析

正向推导：由于可以取[2, 9]之间的任意一个数相乘，初始时 $p=1$ 且 Stan 先乘，则 $n \in [2, 9]$ 时 Stan 必胜，由于第一步 Stan 至少需要乘以 2，则当 $n \in [10, 18]$ 时 Ollie 必胜，同理可推导得出当 $n \in [19, 162]$ 时 Stan 必胜，当 $n \in [163, 324]$ 时 Ollie 必胜，当 $n \in [325, 2916]$ 时 Stan 必胜……不难发现规律：令 Stan 的某个必胜区间为 $[x, y]$ ，则紧接着下一个 Stan 能够必胜的区间为 $[2y+1, 18y]$ 。

逆向推导：假设取 $n=180$ ，由于可以从 2 到 9 之间任选一个数进行乘法，那么只要 $p \geq 20$ ，则下一个玩家就可将 p 乘以 9 使得 $p \geq 180$ 。也就是说，先使得 $p \geq 20$ 的玩家将必败，则玩家应该尽量使得 $p \geq 20$ 延

迟到来，但是由于每次最小必须乘以 2，则先使得 $p \geq 10$ 的玩家将必胜，继续推导，先使得 $p \geq 2$ 的玩家将必败，则可知 Stan 必败。可以使用递归进行逆向推导，从而得到最终的胜负结果。

参考代码

```
int main(int argc, char *argv[])
{
    long long n;
    while (cin >> n) {
        while (n > 18) n = (n + 17) / 18;
        if (n <= 9) cout << "Stan wins.\n";
        else cout << "Ollie wins.\n";
    }
    return 0;
}
```

以下再介绍若干常见的游戏形式，对于此类游戏，可以应用 PN 态分析予以解决^I。

巴什博弈

给定 n 个物品，两个人轮流取，一次最少取走 1 个最多取走 m 个，最后取光的人获胜， $n > m > 0$ 。给定 n 和 m ，判断先手是否具有必胜策略。为了便于讨论，将此种游戏形式称为取物游戏。

分析 巴什博弈 (Bash game) 可以使用 Sprague-Grundy 定理予以解决，而通过 PN 态分析可以得到更为简洁的解决方案。为了便于理解，不妨先观察一种更为直观的巴什博弈形式：设 A 和 B 两名玩家轮流从 1 开始报数，每次最少报 1 个数，最多报 4 个数，谁先报到 100 则为赢家。使用逆向思维进行思考，由于每次至少报 1 个数，最多报 4 个数，如果某个玩家先报到 95，则无论对方玩家报几个数，该玩家都能够一次性报到 100，因此问题转化为谁先报到 95 谁必胜，同理可以继续推导出，谁先报到 90, 85, …, 10, 5 谁就必胜。显然，初始时无论 A 玩家报几个数，B 都能先报到 5，因此在此种情形下，B 玩家必胜。如果将规则更改，最多能够报 5 个数，A 和 B 谁必胜呢？同理可知谁先报到 94, 88, …, 10, 4 谁就必胜，于是 A 玩家可最先报到 4，因此 A 玩家必胜。取物游戏和报数游戏本质是一样的，只不过将报 x 个数替换为取 x 个物品，因此在取物游戏中，谁先取走第 $n-(m+1)$ 个物品，谁就必胜，进而谁先取走 $n-2(m+1)$, $n-3(m+1)$, …, $n-k(m+1)$ 个物品谁必胜，如果 n 能够被 $m+1$ 整除，显然 B 玩家可先到达第一个必胜状态，否则 A 玩家可先到达第一个必胜状态。因此有

$$\text{Bash}[n, m] = \begin{cases} n\%(m+1) = 0, & \text{后手必胜} \\ n\%(m+1) \neq 0, & \text{先手必胜} \end{cases}$$

威佐夫博弈

有两堆物品，各有 n 个和 m 个， $n \geq 0$, $m \geq 0$ ，A 和 B 两位玩家轮流取物品，有两种取法：一种是一次选择一堆物品，至少取一个或者全部取完；另外一种是选择两堆物品，从两堆物品取走相同数量的物品。每次取物品时，可以任选一种取法，先取完所有物品的为赢家。当给定 n 和 m 时，A 和 B 谁具有必胜策略？

分析 威佐夫博弈 (Wythoff's game) 由威佐夫首先予以完整解决而得名^{II}。由规则易知状态(0, 0)为 P

^I Elwyn R. Berlekamp, John H. Conway, Richard K. Guy 三人合著有四卷本的博弈类游戏专著《Winning Ways for Your Mathematical Plays》，此书介绍了大量游戏的数学本质并给出了详尽的游戏策略分析，非常具有参考价值。

^{II} 威廉·亚伯拉罕·威佐夫 (Willem Abraham Wythoff, 1865—1939)，荷兰数学家。

态局势，则 $(0, x)$ 和 (x, x) 为N态局势， $x>0$ 。根据PN态分析，可知具有最小 n 和 m 值的P态局势依次为 $(0, 0), (1, 2), (3, 5), (4, 7), (6, 10), (8, 13), \dots$ 。威佐夫从数学角度研究了此类博弈，并得出了P态局势具有下述性质：令 (a_i, b_i) 表示某种局势，其中 a_i 和 b_i 分别为两堆物品的数量，且 $a_i < b_i, i=0, 1, \dots, n$ ，有

- (1) $a_0=b_0=0$, a_i 是在之前的P态局势中尚未出现的最小非负整数，且 $b_i=a_i+i$;
- (2) 任何非负整数都包含且仅包含在一个P态局势中;
- (3) 任意操作都可以使得P态局势变为N态局势;
- (4) 必有一种操作可以使得N态局势变为P态局势。

通过进一步的研究，威佐夫得出了P态局势的“通项公式”

$$a_i = \left\lfloor i * \left(\frac{1 + \sqrt{5}}{2} \right) \right\rfloor, \quad b_i = a_i + i, \quad i \geq 0$$

令人惊奇的是， $(1 + \sqrt{5})/2$ 恰为黄金分割率(golden ratio)。通过通项公式，可以根据给定的物品数量 n 和 m 确定初始状态是否为P态局势，如果为P态局势，则先手必败，否则先手必胜。

```
//-----6.9.4.cpp-----//
const double GOLDEN_RATIO = (1 + sqrt(5)) / 2;

int main(int argc, char *argv[])
{
    int ai, bi;
    while (cin >> ai >> bi) {
        if (min(ai, bi) == (int)(GOLDEN_RATIO * abs(ai - bi))) cout << "A Lose!\n";
        else cout << "A Win!\n";
    }
    return 0;
}
//-----6.9.4.cpp-----//
```

斐波那契博弈

有一堆物品，共有 n 件，A和B两名玩家轮流从中取物品。先手至少取1件，最多取 $n-1$ 件，之后每次取物品时，玩家能够取走的物品数量不能超过上一名玩家取走物品数量的2倍，但仍然需要至少取走1件，取走最后一件物品的玩家获胜。请确定该游戏是否有必胜策略。

分析 斐波那契博弈(Fibonacci nim)来源于“One Pile”游戏。“One Pile”的游戏规则是给定一堆共 n 件物品，两位玩家轮流取走至少 a 件至多 q 件的物品， $1 \leq a \leq q < n$ ，取走最后一件物品的玩家获胜。当 n 是 $a+q$ 的倍数时，若先手取走 x 件物品，后手只要取走 $a+q-x$ 件物品，使得一轮被取掉的物品总是 $a+q$ 件，则后手总是能够取走最后一件物品，因此先手必败，后手必胜；反之，若 n 不为 $a+q$ 的倍数，则先手第一次取走 $n\%(a+q)$ 件物品后，使得剩余的物品数量是 $a+q$ 的倍数，按前述分析，后手面临的是P态局势，因此先手必胜，后手必败。因此“One Pile”游戏的结论是：当且仅当 n 为 $a+q$ 的倍数时为P态局势，否则为N态局势。不难看出，前述介绍的巴什博弈实际上是“One Pile”游戏当 $a=1, q=m$ 时的特例。

在斐波那契博弈中，后手能够取走的物品数量上限是动态变化的，得到正确的结论具有一定的困难。根据规则，先手第一次能够取走的物品件数至多为 $(n-1)/3$ (除法为整除)，否则后手可一次性取完剩余的物品，导致先手必败。当 n 为2时，由于先手至少取1件，最多取 $n-1$ 件，则先手只能取1件，剩下的1件由后手取走，因此后手必胜，则 $n=2$ 时为P态局势；当 $n=3$ 时，先手只能取1件物品，后手必胜；当 $n=4$ 时，

先手可以先取 1 件物品，则后手按规则只能取 1 件或者 2 件物品，不能一次性把物品全部取完，因此先手必胜。继续推导下去，可以得出当 n 为 5、8、13、21…时亦为 P 态局势，观察容易得知该序列恰为斐波那契数列，这并不是巧合，可以证明：先手必败当且仅当 n 为斐波那契数^[70]。

由于证明过程稍显繁琐，下面根据证明的核心思想予以简要说明。根据齐肯多夫定理^I，任意一个正整数可以表示为若干个（从第 2 项开始的）不相邻斐波那契数之和，进而可以表示为斐波那契进制数，例如 33_{10} 可以表示为

$$33_{10} = 21_{10} + 8_{10} + 3_{10} + 1_{10} = 1010101_f$$

而斐波那契数有具有以下性质

$$F_{i+1} < 2F_i, \quad F_{i+2} > 2F_i, \quad i \geq 2$$

当 n 不为斐波那契数时，将其转换为斐波那契进制数，此时数位中至少有两个 1，此时玩家 A 可以采取如下策略保证必胜：先取走位于最右侧的数位 1 所对应的物品件数，由于 $F_{i+2} > F_i$ 的性质，玩家 B 必定无法一次性取走位于更高位的数位 1 所对应的物品件数，而且玩家 B 根据规则取走物品后，要么剩下的物品能够使得玩家 A 一次性取完，要么剩下的物品数量表示为斐波那契进制数后仍然会包含至少两个为 1 的数位，如果是后者，玩家 A 继续执行“取走位于最右侧的数位 1 所对应的物品件数”的策略即可，从而使得玩家 A 总是能够取走最后一件物品，故先手必胜。也就是说，当 n 为不为斐波那契数时是必胜态。反之，若 n 为斐波那契数，则不论玩家 A 取走的物品数量为何，剩下的物品要么能够被玩家 B 一次性取完，要么表示为斐波那契进制数后至少有两个数位为 1，如果是后者，根据前述的分析，此时的状态对于玩家 B 来说就是一个必胜态。

强化练习：11249* Game^D，11489 Integer Game^B，12293 Box Game^B，12469 Stones^D。

扩展练习：1567* A Simple Stone Game^E。

6.10 小结

在编程竞赛中，与组合数学相关的题目主要分成三个部分：排列与组合、概率论、博弈论。与组合数学相关的题目，解题关键是从题目中抽象出递推关系，可能需要使用高精度整数运算。对于许多计数序列来说，可以先手工推算出前几项，然后通过“整数数列线上大全”（OEIS）进行查询，往往可以得到完整的序列或者与之相关的提示。在本章的众多数列中，斐波那契数列是需要重点掌握的内容，该数列本身及其矩阵表示形式和相应的斐波那契进制，在编程竞赛中多有涉及。

排列组合与母函数（Generating Function，又称生成函数）密切相关，母函数是解决排列和组合问题的有力工具。在难度较大的排列组合题目中，母函数是一个重点的考察内容，由于篇幅所限，本书并未介绍生成函数，如果读者想要进一步提升自己的解题能力，建议查阅相关资料，了解母函数的概念和相关应用。

与概率论相关的题目，主要是与期望有关。要想顺利的解决此类题目，一方面需要对概率论的基本概率理解透彻，另外一方面需要养成在解决概率问题时定义事件的习惯，这样才能充分利用概率论中的相关结论。

与博弈论相关的题目主要考察思维，其基本形式差异不大，主要难点在于从题目中抽象出博弈模型，然后利用 Sprague-Grundy 定理予以解决。

^I 参阅本章第 6.7.1 小节“斐波那契数”中“斐波那契进制”的有关内容。

第 7 章 数论

今有物不知其数，三三数之剩二；五五数之剩三；七七数之剩二。问物几何？
——《孙子算经》^I

在线评测题库经常会出现一些以数论中某些结论为基础的题目，而在既往的国际比赛中，拥有深厚数学功底的选手往往能够获得较好的成绩。数论是纯粹数学的一个分支，主要研究整数的性质，被誉为“最有趣和最优美”的数学分支。数论中的一些问题形式上非常简单，即使是普通人也能够理解，但是证明它却需要高深的数学知识，例如费马大定理^{II}、哥德巴赫猜想^{III}等。数学家对数论中难题的研究从某种意义上推动了数学的发展，催生了大量的新思想和新方法，因此，希尔伯特^{IV}将费马大定理比喻成“一只会下金蛋的母鸡”。鉴于数论的重要性，高斯^V也曾赞誉道：“数学是科学的皇后，数论是数学的皇后”。

由于数论所包含的内容很多，不可能面面俱到地全部罗列出来。在比赛中常常是某个数论结论构成了一道题目的主题，如果解题者熟悉该结论，可以在短时间内得到解题方案。反之，可能需要耗费大量时间才能获得正确的结果，所以本章只是介绍了程序竞赛中最常出现的一些内容，建议感兴趣的读者首先阅读概要介绍数论的书籍^{[71][72]}，然后再通过更深入的初等数论教材进一步学习^{[73][74]}。

7.1 素数

对于整数 $a > 1$ ，如果它只能被 1 和自身整除，则称 a 为素数 (prime number, 或称质数)^{VI}。100 以内的素数一共有 25 个，它们是：2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97。除了 2 是唯一的偶素数外，其他素数均为奇数。

对于大于 1 的整数 a 来说，如果 a 不是素数，则称 a 为合数 (composite)。合数可以表示成素数的乘积，如果不考虑乘积的顺序，其表示方式是唯一的，此即算术基本定理 (fundamental theorem of arithmetic，又称唯一分解定理，unique factorization theorem) —— 任意合数可以表示成有限个素数的乘积，而且在不考虑素数乘积顺序的情况下其表示方式唯一。

^I 《孙子算经》，作者生平和编写年份不详，约成书于公元四、五世纪，是中国古代重要的数学著作。“鸡兔同笼”问题（今有雉兔同笼，上有三十五头，下有九十四足，问雉兔各几何？）亦出自该著作。

^{II} 皮耶·德·费马 (Pierre de Fermat, 1607—1665)，法国人，律师，数学家。费马大定理 (Fermat Last Theorem)：当 $n \geq 3$ 时，不定方程 $x^n + y^n = z^n$ 无 $xyz \neq 0$ 的整数解。费马大定理已于 1993 年 6 月由英国数学家 Andrew Wiles 所解决，这一问题相当于证明：在代数曲线 $\xi^n + \eta^n = 1$ ($n \geq 3$) 上无非显然的有理点。

^{III} 克里斯蒂安·哥德巴赫 (Christian Goldbach, 1690—1764)，德国人，律师，数学家。哥德巴赫猜想 (Goldbach's conjecture)：任意大于 2 的偶数可以表示为两个素数的和，简称“1+1”。目前最好的结果由我国数学家陈景润利用筛法获得：任意大于 2 的偶数可以表示为一个素数及一个不超过两个素数的乘积的和，简称“1+2”。

^{IV} 戴维·希尔伯特 (David Hilbert, 1862—1943)，德国数学家。

^V 约翰·卡尔·弗里德里克·高斯 (Johann Carl Friedrich Gauss, 1777—1855)，德国数学家、物理学家。

^{VI} 截至 2018 年 12 月 7 日，已知的最大素数为 $2^{82589933} - 1$ ，这是一个梅森素数 (Mersenne prime number)。最新的记录可以访问：<https://primes.utm.edu/>，2020。

素数的个数有无穷多个，这个结论可以通过反证法予以证明。这个证明最先由古希腊数学家欧几里得^I给出，证明过程简短而精彩，在此照录如下：

假设素数的个数是有限的，将所有的素数依次列出为 p_1, p_2, \dots, p_n ，设有整数 $M = p_1 \times p_2 \times \dots \times p_n + 1$ ，则 M 不是一个合数，因为 M 不能被 p_1, p_2, \dots, p_n 中的任意一个素数整除，相除总是会产生余数 1，而 M 又不应是素数，因为 $M = p_1 \times p_2 \times \dots \times p_n + 1$ ，必定会大于 p_1, p_2, \dots, p_n 中的任意一个素数，如果 M 是素数，与 p_1, p_2, \dots, p_n 已经包含了所有素数的假设相矛盾，由于 M 是一个大于 1 的正整数，不可能既不是合数又不是素数，产生矛盾，故认为最初的假设是错误的，结论是素数的个数是无穷的^[75]。

160 Factors and Factorials^A (因数和阶乘)

N 的阶乘（写作 $N!$ ）定义为从 1 到 N 的所有整数的乘积，即

$$N! = N \times (N - 1)!, \quad 1! = 1$$

阶乘增长得非常快，例如 $5! = 120$, $10! = 3628800$ 。要表示如此大的数，一种方法是给出该数的素因数分解中各个素数出现的次数，按照此种方法，825 可以表示成 (0 1 2 0 1)，意即素因子分解式中不含 2，包含一个 3，两个 5，不包含 7，包含一个 11。编写程序读入一个整数 N ($2 \leq N \leq 100$)，将 N 的阶乘所包含的各个素数的次数逐一列出。

输入

输入包含多行，每行只包含一个整数 N 。输入以包含 0 的一行结束。

输出

输出包含多个输出块，每个输出块对应输入中的一行。每个输出块以 N 开始，按右对齐宽度 3 输出，后跟字符 ‘!’，一个空格，‘=’，最后列出在 $N!$ 的素因子分解式中各个素数出现的次数。

素因子出现的次数以右对齐宽度为 3 进行输出，每行包含 15 个数（每个输出块的最后一行可能不足 15 个数）。每个输出块的非首行按照样例输出给出的格式进行缩进处理。

样例输入

```
5
53
0
```

样例输出

```
5! = 3 1 1
53! = 49 23 12 8 4 4 3 2 2 1 1 1 1 1 1
1
```

分析

解题思路非常直接，将阶乘中的每个整数逐一进行素数分解，统计相应素数的次数然后输出即可。可以先列出 1 到 100 间的素数以备用。

强化练习：[369 Combinations^E](#), [993 Product of Digits^A](#), [10110* Light More Light^A](#)。

^I 欧几里得·亚历山大 (Euclid of Alexandria, 生卒年月不详)，古希腊数学家，著有《几何原本》一书。

7.1.1 素数判定

判定一个给定整数 a 是否为素数，朴素的方法是根据素数的定义，使用不断试除的方式来进行。即用 2 到 $a-1$ 之间的数去除 a ，如果 a 能被整除，则 a 不是素数，否则 a 是素数。

由于 2 是唯一的偶素数，只要 a 是不等于 2 的偶数，则 a 是合数。在试除时，在排除了 a 是偶素数后，只需使用奇数去除且并不需要一直除到 $a-1$ ，而只需要除到 $\lfloor \sqrt{a} \rfloor$ 。因为如果 a 是合数，它的任意两个因数不可能都大于 $\lfloor \sqrt{a} \rfloor$ ，否则设这两个因数为 b 和 c ，则

$$b \cdot c \geq (\lfloor \sqrt{a} \rfloor + 1)^2 > (\sqrt{a})^2 = a$$

产生矛盾。更进一步，如果已经知道了 2 到 $\lfloor \sqrt{a} \rfloor$ 之间的素数，则只需使用这些素数去除，如果 a 不能被整除，则 a 是素数。因为 a 不能被 2 到 $\lfloor \sqrt{a} \rfloor$ 之间的素数所整除，也必定不能被 2 到 $\lfloor \sqrt{a} \rfloor$ 之间的合数所整除。

```
//-----7.1.1.cpp-----
// 判断给定的整数是否为素数。
bool isPrime(int a)
{
    // 特殊情形的判断。
    if (a <= 1) return false;
    // 判断是否为唯一的偶素数。
    if (a == 2) return true;
    // 不为 2 的偶数是合数。
    if (!(a & 1)) return false;
    // 试除法确定是否为素数。
    int factor = (int)sqrt(a);
    for (int i = 3; i <= factor; i += 2)
        if (a % i == 0)
            return false;
    return true;
}
//-----7.1.1.cpp-----
```

强化练习：[543 Goldbach's Conjecture^A](#)，[10042 Smith Numbers^A](#)，[10168 Summation of Four Primes^A](#)，[10200* Prime Time^A](#)，[10650* Determinate Prime^B](#)，[10699 Count the factors^A](#)，[10852 Less Prime^A](#)，[10924 Prime Words^A](#)，[12802 Gift From the Gods^B](#)。

7.1.2 米勒-拉宾素性测试

当给定的整数 a 较小时，使用试除法可以较快地确定其是否为素数，但是当 a 很大时（例如 10^{60} ），使用试除法将不现实，此时需要使用其他的方法来判定 a 是否为素数。米勒-拉宾素性测试（Miller-Rabin primality test）^I是判断一个给定整数是否为素数的有效算法。该算法属于概率算法，因为它使用概率来表示某个数是素数的可能性，当概率达到一定数值时，即可认为给定的整数为素数。

米勒-拉宾素性测试的数学基础是费马小定理（Fermat's little theorem），该定理指出：给定任意素数 p ，

^I 加里·李·米勒（Gary Lee Miller, 1947—），卡内基梅隆大学数学教授。迈克尔·奥泽·拉宾（Michael Oser Rabin, 1931—），以色列数学家、计算机科学家，1976 年图灵奖获得者。米勒首先提出了基于广义黎曼猜想的素性测试确定性算法，由于广义黎曼猜想并没有被证明，其后由拉宾教授作出修改，提出了不依赖于该假设的素性测试随机化算法。

对于整数 a , 如果 $1 < a < p$, 有以下结论成立^I

$$a^{p-1} \equiv 1 \pmod{p} \quad (7.1)$$

注意, 费马小定理的逆命题——“给定某个正整数 p , 如果对于任意选定的某个正整数 a , $1 < a < p$ 且 a 均满足同余式 (7.1), 则 p 是素数”——不成立, 有无限多个合数符合逆命题, 这些合数称为卡迈克尔数 (Carmichael number)^{II}。

强化练习: 10006 Carmichael Numbers^A。

可以证明, 如果 p 是一个奇素数且 $e \geq 1$, 则同余方程

$$x^2 \equiv 1 \pmod{p^e} \quad (7.2)$$

仅有两个解: $x=1$ 和 $x=-1$ 。当给定一个奇素数 p , 则 $p-1$ 为偶数, 而且 $p-1$ 可以表示成如下形式

$$p-1 = 2^s \cdot d \quad (7.3)$$

其中 s 和 d 均为正整数, d 为奇数。根据 (7.1) 和 (7.2) 两个结论, 可以证明, 对于 $1 < a < p$, 有

$$a^d \equiv 1 \pmod{p} \text{ 或者 } a^{2^r \cdot d} \equiv -1 \pmod{p}, \quad 0 \leq r \leq s-1 \quad (7.4)$$

根据结论 (7.4) 的逆否命题, 只要我们找到一个基数 a , 使得

$$a^d \not\equiv 1 \pmod{p} \text{ 并且 } a^{2^r \cdot d} \not\equiv -1 \pmod{p}, \quad 0 \leq r \leq s-1 \quad (7.5)$$

那么 p 不是素数。米勒-拉宾素性测试即基于此推论。给定奇数 $n > 2$, 每当随机选取 $1 < a < n$ 之间的一个数作为基数进行素性测试, 如果均满足结论 (7.4) 的要求, 则 n 有可能是一个素数, 而且测试通过的次数越多, n 为素数的可能性就越大, 但若有一次测试不满足上述等式, 那么可以确定 n 为合数。给定数 n , 若 n 通过了一次测试, 则 n 不是素数的概率为 $\frac{1}{4}$, 若数 n 通过了 k 次测试, 则 n 是素数的概率为 $1 - \frac{1}{4^k}$ 。

以下是米勒-拉宾素性测试的参考实现。

```
//-----7.1.2.cpp-----//
// 进行素性测试时的最大迭代次数。
const int K = 20;

// 以加法和乘法结合的方式进行模运算, 以便最大限度地避免溢出。
long long multiplyMod(long long a, long long b, long long mod)
{
    long long x = 0, y = a % mod;
    while (b) {
        if (b & 1) x = (x + y) % mod;
        y = (y * 2) % mod;
        b >>= 1;
    }
    return x;
}
```

^I 费马小定理实际上是欧拉定理的一个特例。欧拉定理 (Euler's theorem): 设 m 为自然数, a 为整数, 且 a 和 m 的最大公约数为 1, 则有

$$a^{\varphi(m)} \equiv 1 \pmod{m}$$

其中 $\varphi(m)$ 表示欧拉函数。 $\varphi(m)$ 定义为小于等于 m 的正整数中与 m 的最大公约数为 1 的整数个数。若 m 为素数, 显然有 $\varphi(m) = m - 1$, 从而推出费马小定理成立。

^{II} 罗伯特·丹尼尔·卡迈克尔 (Robert Daniel Carmichael, 1879—1967), 美国数学家。卡迈克尔数必须是“无平方数”(不能被任何素数的平方所整除), 且至少是三个素数的积。因为这个原因, 卡迈克尔数在正整数中所占的比例非常小。例如, 在 1 到 10^8 之间只有 255 个卡迈克尔数。最小的卡迈克尔数是 $561 = 3 \times 11 \times 17$ 。

```

}

// 快速幂取模。
long long modulo(long long base, long long exponent, long long mod)
{
    long long x = 1, y = base;
    while (exponent) {
        if (exponent & 1) x = multiplyMod(x, y, mod);
        y = multiplyMod(y, y, mod);
        exponent >= 1;
    }
    return x;
}

// 米勒-拉宾素性测试。
bool isPrime(long long p)
{
    // 初步筛除。
    if (p < 2) return false;
    if (p == 2) return true;
    if (!(p & 1)) return false;

    // 准备。
    long long q = p - 1;
    while (!(q & 1)) q >>= 1;

    // 执行测试。
    for (int i = 0; i < K; i++) {
        long long a = rand() % (p - 1) + 1;
        long long t = q;
        long long mod = modulo(a, t, p);
        while (t != p - 1 && mod != 1 && mod != p - 1) {
            mod = multiplyMod(mod, mod, p);
            t <<= 1;
        }
        if (mod != p - 1 && !(t & 1)) return false;
    }
    return true;
}
//-----7.1.2.cpp-----

```

值得一提的是，Java 的 `BigInteger` 类提供了米勒-拉宾素性测试的实现，其方法声明为：

```

// 当大整数可能为素数时，返回 true，如果返回 false，则大整数确定为合数。如果传入的参数
// certainty 小于等于 0，则返回 true。参数 certainty 是大整数为素数的可能性的一种度量，
// 如果方法返回 true，则表明此大整数为素数的可能性超过  $(1 - 1/2^{certainty})$ 。方法的运行时间
// 与 certainty 的大小成正比。
public boolean isProbablePrime(int certainty)

```

熟悉 Java 语言的读者可以方便地应用此方法对给定的大整数进行素性测试。

强化练习：[11287 Pseudoprime Numbers^C](#)。

7.1.3 高斯素数

复数 (complex) 为二元有序实数对，通常记为 $z = a + bi$ ， a 和 b 为实数， i 为虚数单位，定义 $i = \sqrt{-1}$ 。

高斯整数 (Gaussian integer) 是指实部 (real part) a 和虚部 (imaginary part) b 均为整数的复数。与有理整数类似，在高斯整数中也有素数的概念，称之为高斯素数 (Gaussian prime)。自然数中的素数不能分解为除 1 和自身外的乘积，类似的，高斯素数也不能分解为其他高斯整数的乘积。注意，在分解时要求分解得到的高斯整数不能包含 0、1、 $+1$ 、 $+i$ 、 $-i$ 。在自然数中某个数是素数，在高斯整数中不一定是高斯素数，例如 $2 = (1+i)(1-i)$ ，故 2 不是高斯素数。

对于高斯素数的判定有以下结论：

(1) 如果 $a \neq 0$ 且 $b \neq 0$ ，则判断复数的范数^I $a^2 + b^2$ 是否为有理素数 (即自然数中的素数)，如果是则为高斯素数。

(2) 如果 $a = 0$ ，则判断 $|b|$ 是否为有理素数且 $|b| \equiv 3 \pmod{4}$ ，如果满足则为高斯素数；

(3) 如果 $b = 0$ ，则判断 $|a|$ 是否为有理素数且 $|a| \equiv 3 \pmod{4}$ ，如果满足则为高斯素数；

可以进一步扩展，定义类似于 $a + b\sqrt{-k}$ 的高斯整数，其中 k 为自然数。在判断这些高斯整数是否为高斯素数时，规则与上述介绍的结论类似，只不过在求复数的范数时需要将 $\sqrt{-k}$ 加以考虑。有理整数域中的许多性质可以类推到高斯整数域中，不过有些基本的性质对所有的 k 并不成立，例如算术基本定理 (唯一分解定理) 只有当 $k \in \{1, 2, 3, 7, 11, 19, 43, 67, 163\}$ 时才成立。例如，当 $k=5$ 时，有 $6 = 2 \times 3 = (1 + \sqrt{-5})(1 - \sqrt{-5})$ ，故此时算术基本定理不成立。

强化练习：960 Gaussian Primes^D。

扩展练习：1415* Gauss Prime^E。

7.1.4 生成素数序列

当需要确定给定范围内的每个整数是否为素数且范围不是很大时 (例如小于 10^8)，可以使用埃氏筛法 (sieve of Eratosthenes)^{II}。该方法生成素数的速度较试除法要快得多。埃氏筛法是根据“当前素数的所有倍数都是合数”的事实来筛除不是素数的候选整数，如果某数 a ($a \geq 2$) 不是之前确定的任何素数 p ($2 \leq p < a$) 的倍数，根据算术基本定理的逆否命题，如果 a 不能被分解为除本身以外的素数的乘积，则 a 不是合数，那么 a 必定是素数。

^I 若 $z = x + yi$ 是复数，定义 z 的绝对值 (absolute value, 或称模，modulus) 为

$$|z| = \sqrt{x^2 + y^2}$$

进一步地，定义 z 的范数 (norm) 为

$$N(z) = |z|^2 = x^2 + y^2$$

^{II} 埃拉托斯特尼 (Eratosthenes，前 276 年—约前 194 年)，古希腊时期的数学家、地理学家、天文学家兼诗人和乐理家。他是第一个使用科学方法对地球的周长进行测量的人，所测结果在当时的条件下具有很高的精度。

参阅：<https://en.wikipedia.org/wiki/Eratosthenes>，2020。

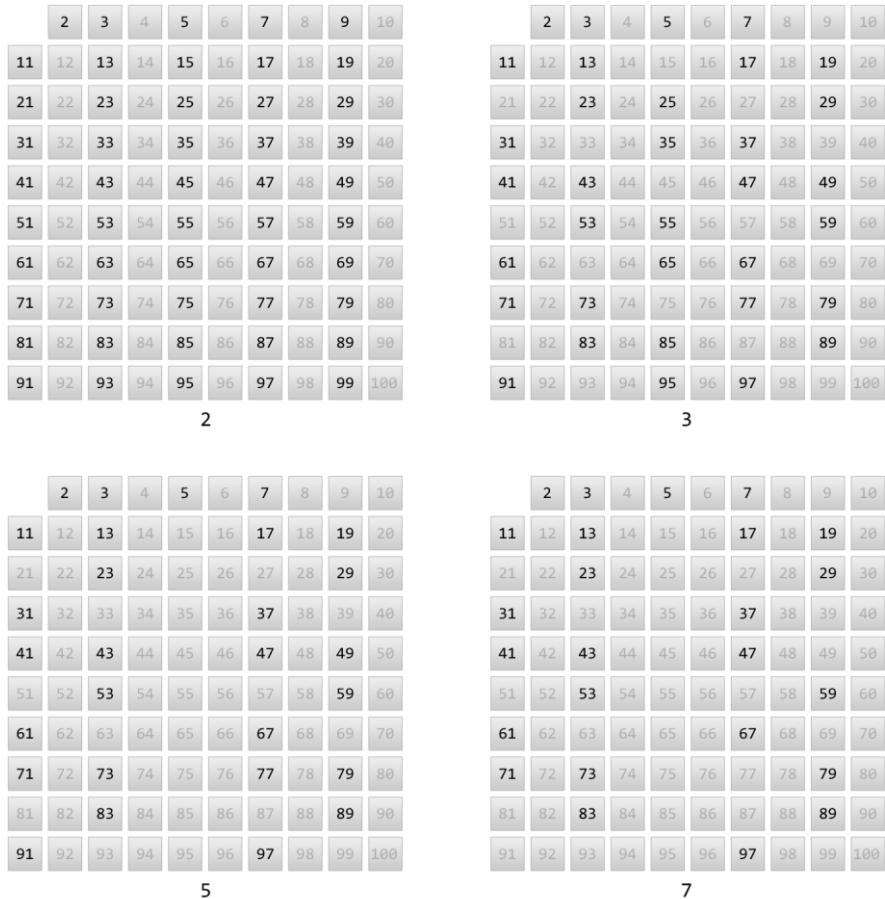


图 7-1 埃氏筛法在 2 到 100 之间的数上的应用。进行 2、3、5、7 四个数的所有倍数的筛除即可获得小于 100 的所有素数

以下是埃氏筛法的参考实现，寻找得到的素数仍然存放在原数组中以便后续使用。

```
//++++++7.1.4.cpp+++++++
const int MAXN = 1000000;

int primes[MAXN], cnt;

// 第一种方式，边筛选素数边记录。
void sieve1()
{
    // 从最小的素数开始筛选。
    for (int i = 2; i < MAXN; i++)
        if (!primes[i]) {
            // 记录筛选得到的素数。
            primes[cnt++] = i;
            // 如果 i 为素数则将其倍数标记为非素数。
            for (int j = i + i; j < MAXN; j += i)
                primes[j] = 1;
```

```

    }

// 第二种方式, 先筛选素数再记录。
void sieve2()
{
    // 筛除非素数。
    for (int i = 2; i * i < MAXN; i++)
        if (!primes[i])
            for (int j = i * i; j < MAXN; j += i)
                primes[j] = 1;
    // 记录筛选得到的素数。
    for (int i = 2; i < MAXN; i++)
        if (!primes[i])
            primes[cnt++] = i;
}

```

强化练习: 686 Goldbach's Conjecture (II)^A, 1195 Calling Extraterrestrial Intelligence Again^D, [1210 Sum of Consecutive Prime Numbers^B](#), [10015 Joseph's Cousin^B](#), [10140 Prime Distance^A](#), [10311 Goldbach and Euler^A](#), [10394 Twin Primes^A](#), [10490* Mr. Azad and His Son^A](#), [10539 Almost Prime Numbers^A](#), [10948 The Primary Problem^A](#), [11086* Composite Prime^C](#), [11408* Count DePrimes^B](#)。

扩展练习: 11510 Erdős Unit Fractions^E。

在上述埃氏筛法的实现中, 对于某些合数, 可能会出现多次重复标记其不为素数的情形, 可以予以改进, 使得对合数只进行一次标记以提高效率。

```

void sieve3()
{
    // 初始时假定所有数为素数。
    // 从最小的素数开始筛除。
    for (int i = 2; i < MAXN; i++) {
        // 代码 1: 记录筛选得到的素数。
        if (!primes[i]) primes[cnt++] = i;
        // 标记合数。注意, 不论 i 是否是素数均需要进行标记操作。
        for (int j = 0; j < cnt && i * primes[j] < MAXN; j++) {
            // 代码 2: 将素数的倍数标记为合数。
            primes[i * primes[j]] = 1;
            // 代码 3: 退出标记操作。
            if (i % primes[j] == 0) break;
        }
    }
}

```

为什么上述改进可以实现对任意合数只进行一次标记呢? 首先, 代码 1 记录当前确定的素数; 其次, 根据算术基本定理, 任意大于 2 的合数 a 可以分解为素数的乘积, 则 a 必定是某个小于 a 的素数的倍数, 在代码 2 的标记过程中, 根据确定的素数, 标记了指定范围内所有素数的所有倍数, 由于任意合数 a 必定是某个小于 a 的素数的倍数, 因此只要是指定范围内的合数, 均会予以标记。那么如何保证某个合数不会被重复标记呢? 依靠代码 3 来实现, 在标记合数 $c_1 = i_{c_1} \times prime[j]$ 后, 如果 i_{c_1} 能够被素数 $prime[j]$ 所整除, 不妨令 $b = i_{c_1}/prime[j]$, 那么拟要标记的合数 $c_2 = i_{c_1} \times prime[j+1]$, $c_3 = i_{c_1} \times prime[j+2]$ ……将会被后续标记过程中的 $c_2 = (b \times prime[j+1]) \times prime[j]$, $c_3 = (b \times prime[j+2]) \times prime[j]$ ……所标记, 实际对 c_2 进行标记的 $i_{c_2} = b \times prime[j]$

$+1] > i_1$, 对 $c_3 = b \times prime[j+2] > i_2 \dots \dots$ 若当前仍予以标记就会造成重复, 因此需要在此处退出标记操作。综上所述, 代码 1 记录了当前的所有素数, 代码 2 保证了所有合数均会被标记, 代码 3 保证了合数不会被重复标记, 这样既获取了所有素数, 又保证了标记合数过程中的不重不漏, 其时间复杂度为 $O(n)$ 。

如果给定的范围较大, 使用整数数组予以表示可能会超时, 此时可以使用位来标记某个数是否为素数, 从而避免超出内存限制^I。

```
#define GET(x) (B[x >> 5] & (1 << (x & 0x1F)))
#define SET(x) (B[x >> 5] |= (1 << (x & 0x1F)))

const int MAXN = 7000000, MAXB = 100000001;
int primes[MAXN], cnt, B[MAXB >> 5] = {3};

// 素数筛选完毕后, 若需检查给定范围内的某个整数 i 是否为素数,
// 使用 GET(i) 检查结果其是否为 0, 为 0 表示 i 为素数, 否则 i 为合数。
void sieve4()
{
    for (int i = 2; i < MAXB; i++) {
        if (!GET(i)) primes[cnt++] = i;
        for (int j = 0; j < cnt && i * primes[j] < MAXB; j++) {
            SET(i * primes[j]);
            if (i % primes[j] == 0) break;
        }
    }
}
//+++++7.1.4.cpp+++++
```

强化练习: 897 Anagrammatic Primes^B, 967 Circular^C, [10235 Simply Emirp^A](#), [10533 Digit Primes^A](#), [10871 Primed Subsequence^C](#), 12542 Prime Substring^C。

扩展练习: [1644 Prime Gap^C](#), [10856 Recover Factorial^C](#), [11105 Semi-Prime H-Numbers^C](#), [11415 Count the Factorials^D](#), [12218 An Industrial Spy^D](#)。

7.1.5 素因子分解

根据算术基本定理, 任意合数可以表示成素数的乘积形式, 即给定合数 c , 可以将其表示为

$$c = p_1^{e_1} \cdot p_2^{e_2} \cdots p_n^{e_n}, p_i \text{ 为素数, } e_i \text{ 为正整数}$$

在生成素数序列后, 假设序列中最大的素数为 p_{max} , 利用已有的素数, 可以很容易将小于等于 $p_{max} \times p_{max}$ 的数进行素因子分解。

^I 在整数数组 B 中, 每个元素都是一个整数, 占 4 个字节, 即一个数组元素有 32 个位, 可以标记 32 个整数。也就是说, B[0] 中的 32 个二进制位可以用于标记整数 0 至 31 是否为素数, B[1] 中的 32 个二进制位可以用于标记整数 32 至 63 是否为素数……依此类推。为了方便地将序号 x 映射到整数数组 B 中的某个位, 可以使用示例代码中所示的两个宏定义予以实现。由于每 32 个整数对应 B 中的一个元素, 因此整数 x 所对应的 B 中的元素为 B[x / 32], 即 B[x >> 5], 接下来需要确定 x 在 B[x >> 5] 对应着哪一个二进制位。由于是每 32 个整数“一段”, 整数 x 在第 x / 32 段, x 在段内的序号就是 x 除以 32 后所得的余数, 使用位运算来表示的话, 就是 x 对应二进制数的最后 5 个二进制位的值, 因此, x & 0x1F 表示整数 x 在段内的序号, 结合获取单个二进制位值的方法, 容易理解 (B[x >> 5] & (1 << (x & 0x1F))) 就可以获取 x 所对应的二进制位值。

```

//++++++++++++++++++++7.1.5.cpp++++++++++++++++++++
// 注意: 使用筛法生成的素数范围要足够大, 至少不小于给定最大整数的平方根。
map<int, int> factors;
for (int i = 0; i < cnt; i++) {
    // 已经不可能存在素因子时提前退出以提高效率。
    if (primes[i] * primes[i] > n) break;
    while (n % primes[i] == 0) {
        n /= primes[i];
        factors[primes[i]]++;
    }
}
// 如果最后 n 不能被除尽 (即 n 大于 1), 表明 n 是一个不在生成素数范围之内的素数。
if (n > 1) factors[n]++;

```

完成素因子分解后，可以回答以下问题——合数 c 总共有多少个不同的约数？根据乘法原理， c 的不同约数个数为

$$\tau(c) = \prod_{i=1}^k (e_i + 1) = (e_1 + 1)(e_2 + 1) \cdots (e_k + 1)$$

进一步地可以根据素因子分解的结果得到这些不同的约数。

```

vector<int> divisors = {1};
for (auto f : factors) {
    int base = 1, countOfDivisors = divisors.size();
    for (int i = 1; i <= f.second; i++) {
        base *= f.first;
        for (int j = 0; j < countOfDivisors; j++)
            divisors.push_back(divisors[j] * base);
    }
}
// 排序并去除重复的约数。
sort(divisors.begin(), divisors.end());
divisors.erase(unique(divisors.begin(), divisors.end()), divisors.end());
//+++++7.1.5.cpp+++++//
```

类似的，可以证明， c 的所有不同约数的和为

$$\sigma(c) = \prod_{i=1}^k \frac{p_i^{e_i+1} - 1}{p_i - 1} = \frac{p_1^{e_1+1} - 1}{p_1 - 1} \times \frac{p_2^{e_2+1} - 1}{p_2 - 1} \times \cdots \times \frac{p_{k-1}^{e_{k-1}+1} - 1}{p_{k-1} - 1} \times \frac{p_k^{e_k+1} - 1}{p_k - 1}$$

例如, $12=2^2\times 3$, 12 的约数有 1、2、3、4、6、12, 而

$$\sigma(12) = 1 + 2 + 3 + 4 + 6 + 12 = \frac{2^{2+1} - 1}{2 - 1} \times \frac{3^{1+1} - 1}{3 - 1} = 28$$

强化练习: [294 Divisors^A](#), [516 Prime Land^A](#), [583 Prime Factors^A](#), [884 Factorial Factors^A](#), [10484 Divisibility of Factors^C](#), [10622 Perfect P-th Powers^A](#), [10780 Again Prime No Time^B](#), [10880 Colin and Ryan^B](#), [11347 Multifactorials^D](#), [11353 A Different Kind of Sorting^C](#), [11466* Largest Prime Divisor^A](#), [11728 Alternate Task^B](#), [11960 Divisor Game^C](#), [12043 Divisors^C](#), [12090* Counting Zeroes^D](#), [12703 Little Rakin^D](#), [12805 Raiders of the Lost Sign^D](#), [13067 Prime Kebab Menu^B](#), [13131 Divisors^D](#)。

扩展练习: 107* The Cat in the Hat^A, 640 Self Numbers^A, 1246 Find Terrorists^C, 10061* How Many Zero's and How Many Digits^B, 10290 {Sum+=i++} to Reach N^D, 10742 The New Rule in Euphonia^C, 10958* How Many Solutions^D, 11099 Next Same-Factored^D, 11395* Sigma Function^D, 11476 Factorizing Largest

Integers^D, [11610*](#) Reverse Prime^D, [11876](#) N + NOD (N)^B, [12005*](#) Find Solutions^D, [12137](#) Puzzles of Triangles^E。

7.1.6 完全数

给定一个大于 1 的整数 n , 如果 n 的所有真因数 (真因数是指除 n 本身以外的 n 的其他因数) 之和等于 n , 则称 n 为完全数 (perfect number, 又称完美数或者完备数)。如果取 6 的真因数, 即除 6 本身外的 6 的因数, 有 1, 2, 3, 而 $1+2+3=6$, 因此 6 是一个完全数。完全数的数量非常少, 前 10 个完全数是^I: 6, 28, 496, 8128, 33550336, 8589869056, 137438691328, 2305843008139952128, 2658455991569831744654692615953842176, 191561942608236107294793378084303638130997321548169216。

对于较小的 n (例如, $n < 10^6$), 判断 n 是否为完全数, 可以列出它所有的真因数, 将其相加, 检查是否等于 n 。定义 $\sigma(n)$ 表示 n 的所有不同约数的和, 对于稍大的 n (例如, $n < 10^{12}$), 可以结合素因子分解, 利用前述介绍的求合数 n 的所有不同约数和的公式来确定 $\sigma(n)$ 。

```
-----7.1.6.cpp-----//
#define GET(x) (B[x >> 5] & (1 << (x & 0x1F)))
#define SET(x) (B[x >> 5] |= (1 << (x & 0x1F)))

const int MAXB = 100000010, MAXN = 7000000;

int P[MAXN], B[MAXB >> 5], cnt;

int main(int argc, char *argv[])
{
    for (int i = 2; i < MAXB; i++) {
        if (!GET(i)) P[cnt++] = i;
        for (int j = 0; j < cnt && i * P[j] < MAXB; j++) {
            SET(i * P[j]);
            if (i % P[j] == 0) break;
        }
    }

    long long n;
    while (true) {
        cout << "Enter a number which is not bigger than 10^12. Enter 0 to quit.\n";
        cin >> n;
        if (n == 0) break;
        if (n < 0 || n > pow(10, 12)) continue;
        long long r = 1, nn = n;
        for (int i = 0; i < cnt && P[i] * P[i] <= n; i++) {
            if (n % P[i]) continue;
            long long p = P[i];
            while (n % P[i] == 0) {
                n /= P[i];
                p *= P[i];
            }
            r *= (p - 1) / (P[i] - 1);
        }
        if (n > 1) r *= 1 + n;
    }
}
```

^I 参见: <https://oeis.org/A000396>。

```

        cout << "σ(" << nn << ") = " << r << '\n';
    }

    return 0;
}
//-----7.1.6.cpp-----//

```

欧几里得关于完全数有一个结论：如果 $2^p - 1$ 是素数，则 $2^{p-1}(2^p - 1)$ 是完全数。

欧拉证明：如果 n 是偶完全数，则 n 形如

$$n = 2^{p-1}(2^p - 1)$$

其中 $2^p - 1$ 是梅森素数。例如， $6 = 2^{2-1}(2^2 - 1)$, $p = 2$; $28 = 2^{3-1}(2^3 - 1)$, $p = 3$; $496 = 2^{5-1}(2^5 - 1)$, $p = 5$; $8128 = 2^{7-1}(2^7 - 1)$, $p = 7$; $33550336 = 2^{13-1}(2^{13} - 1)$, $p = 13$; $8589869056 = 2^{17-1}(2^{17} - 1)$, $p = 17$; $137438691328 = 2^{19-1}(2^{19} - 1)$, $p = 19$ ……

是否存在奇完全数呢？直到今天，虽然一直没有间断尝试，但没有人发现奇完全数，且目前人们已知道不存在小于 10^{300} 的奇完全数。

强化练习：[1180 Perfect Numbers^C](#), [13185 DPA Numbers I^B](#), [13194 DPA Numbers II^D](#)。

7.2 整除性

如果 a 和 b 为整数且 $a \neq 0$, a 整除 (divides) b 是指存在整数 c 使得 $b = ac$, 如果 a 整除 b , 称 a 是 b 的一个因子, 且称 b 是 a 的倍数, 将其记为 $a|b$, 如果 a 不能整除 b , 则将其记为 $a \nmid b$ 。

强化练习：[1648 Business Center^A](#), [10190 Divide But Not Quite Conquer^A](#), [10633 Rare Easy Problem^A](#), [11296 Counting Solutions to an Integral Equation^C](#)。

扩展练习：[10493 Cats With or Without Hats^C](#), [11298 Dissecting a Hexagon^D](#), [11526* H\(n\)^B](#)。

7.2.1 最大公约数

给定两个正整数 a 和 b , a 和 b 的最大公约数 (greatest common divisor, 或称最大公因子) 定义为能够同时整除 a 和 b 的最大正整数, 记为 $\gcd(a, b)$, 有

$$\gcd(a, b) = \max\{k, k|a \text{ 且 } k|b\}$$

根据算术基本定理, 任何正整数可以表示成素数的乘积, 假设正整数 a 与 b , 将其表示为素数的乘积

$$a = 2^{x_1} \times 3^{x_2} \times 5^{x_3} \dots, \quad b = 2^{y_1} \times 3^{y_2} \times 5^{y_3} \dots$$

那么很显然

$$\gcd(a, b) = 2^{\min(x_1, y_1)} \times 3^{\min(x_2, y_2)} \times 5^{\min(x_3, y_3)} \dots$$

实际编码中使用上述方法求最大公约数并不是很方便, 因为对整数进行素因子分解目前尚无有效的方法, 一般都是通过使用素数不断试除的方法来找到给定整数的所有素因子。更为高效的方法是应用欧几里得算法 (又称辗转相除法)。它是一个递归算法, 可以将其表示为

$$\gcd(a, b) = \begin{cases} a & b = 0 \\ \gcd(b, a \bmod b) & b \neq 0 \end{cases}$$

下面对其作简要地证明。首先考虑边界情况, 因为任何整数都能整除 0, 故有 $\gcd(a, 0) = a$, 所以边界情况是正确的。再来考虑一般情况, 先考虑 $a \geq b$ 的情况, 设 a 除以 b 的商为 q , 余数为 r , 有

$$a = b \cdot q + r$$

考虑 $\gcd(b, r)$, 它既能整除 $b \cdot q$, 也能整除 r , 那么 $\gcd(b, r)$ 必定能够整除两者的“和”—— $b \cdot q + r = a$,

所以 $\gcd(b, r)$ 既能整除 a 也能整除 b , 那么可知 $\gcd(b, r)$ 是 a 和 b 的一个公约数(但不一定是最大公约数), 由于 $\gcd(a, b)$ 是 a 和 b 的最大公约数, 有

$$\gcd(b, r) \leq \gcd(a, b)$$

再考虑 $\gcd(a, b)$, 它既能整除 $b \cdot q$, 也能整除 a , 那么 $\gcd(a, b)$ 必定能够整除两者的“差”—— $a - b \cdot q = r$, 所以 $\gcd(a, b)$ 既能整除 b 也能整除 r , 那么可知 $\gcd(a, b)$ 是 b 和 r 的一个公约数, 由于 $\gcd(b, r)$ 是 b 和 r 的最大公约数, 有

$$\gcd(a, b) \leq \gcd(b, r)$$

那么可得

$$\gcd(a, b) = \gcd(b, r) = \gcd(b, a \bmod b)$$

以上证明是基于 $a \geq b$ 的, 对于 $b < a$ 的情况, 由于 $\gcd(a, b) = \gcd(b, a)$, 同理可证。

接下来估算一下欧几里得算法的时间复杂度。不妨设 $a \geq b$, 有以下结论:

$$\text{若 } b > \frac{a}{2}, \text{ 则 } a \bmod b = a - b < \frac{a}{2}; \text{ 若 } b \leq \frac{a}{2}, \text{ 则 } a \bmod b < b < \frac{a}{2}$$

则每进行一次递归调用, 参数的大小将减小为原来的一半, 因此其时间复杂度近似为 $O(\log \max(a, b))$ ^I, 效率还是很高的。

以下是欧几里得算法的递归实现和迭代实现。

```
//++++++7.2.1.cpp+++++++
// 递归实现。
int gcd1(int a, int b)
{
    if (a < b) swap(a, b);
    return b ? gcd1(b, a % b) : a;
}

// 使用模运算的迭代实现。
int gcd2(int a, int b)
{
    if (a < b) swap(a, b);
    int t;
    while (b != 0) t = a, a = b, b = t % b;
    return a;
}
```

在某些特殊情况, 也可以使用减法替代模运算。

```
// 使用减法运算的迭代实现。
int gcd3(int a, int b)
{
    if (a == 0) return b;
    if (b == 0) return a;
    while (a != b) if (a > b) a -= b; else b -= a;
    return a;
}
```

^I 加布里尔·拉梅 (Gabriel Lamé, 1795—1870) 证明: 用欧几里得算法计算两个正整数的最大公约数时, 所需的除法次数不会超过两个整数中较小的那个十进制数的位数的 5 倍, 简称拉梅定理。由拉梅定理可以得到以下推论: 求两个正整数 a, b , $a > b$ 的最大公约数需要 $O((\log a)^3)$ 次的位运算。

```

}
//++++++++++++++++++++++++++++++++++++++7.2.1.cpp+++++++++++++++++++++

```

在 GCC 所使用的库实现中, 头文件`<algorithm>`包含了一个内置的求最大公约数的函数:

```

template <typename _EuclideanRingElement>
_EuclideanRingElement __gcd
(_EuclideanRingElement __m, _EuclideanRingElement __n)
{
    while (__n != 0) {
        _EuclideanRingElement __t = __m % __n;
        __m = __n;
        __n = __t;
    }
    return __m;
}

```

可以看到, 它也是使用欧几里得算法来计算最大公约数。在程序竞赛环境下, 为了节省时间, 可以直接使用这个内置函数来求最大公约数。

10407 Simple Division^A (简单除法)

在被除数 n 和除数 d 之间的整除运算会产生商 q 和余数 r , 其中 q 是使得 $q \times d$ 尽可能大但同时满足 $q \times d \leq n$ 以及 $r = n - q \times d$ 的整数。

按照上述的整除定义, 对于任意给定的一组整数, 都会存在某个整数 d , 使得该组整数中的任意一个整数除以 d 都会得到相同的余数。

输入

输入包含多组数据, 每组数据一行。每行包含一个非零整数序列, 以空格分隔。每行数据的最后一个数为 0, 不计入整数序列。每个整数序列至少 2 个整数, 至多不超过 1000 个整数, 序列中的整数不一定相同。输入的最后一行包含一个整数 0, 不需处理该行输入。

输出

对于每组测试数据, 输出能够找到的最大整数 d , 使得 d 整除序列中的每个数得到的余数相同。

样例输入

```
701 1059 1417 2312 0
```

样例输出

```
179
```

分析

令整数序列为 a_1, a_2, \dots, a_n , 所求的最大整数为 d , 余数为 r , 根据题意, 有

$$a_1 = d \cdot q_1 + r, \quad a_2 = d \cdot q_2 + r, \quad \dots, \quad a_n = d \cdot q_n + r$$

容易推出任意两个整数 a_i 和 a_j 之间的差都能够被 d 整除, 因此寻找所有整数差的最大公约数即可。具体实现时, 并不需要计算任意两个整数之间的差值的最大公约数, 只需将序列按升序排列, 计算相邻两个整数间差值的最大公约数即可, 因为任意两个整数之间的差值均可以通过按升序排列的相邻两个整数间差值的和获得。此外还需要注意处理序列中整数有部分相同、全部相同且为负数等特殊情形, 可以通过 `unique` 函数去除重复值, 若去除重复值后序列只剩下一个整数, 表明序列中整数全部相同, 输出其绝对值即为所求。

参考代码

```

int main(int argc, char *argv[])
{
    int n;
    vector<int> ns;
    while (cin >> n, n != 0) {
        ns.push_back(n);
        while (cin >> n, n != 0) ns.push_back(n);
        sort(ns.begin(), ns.end());
        ns.erase(unique(ns.begin(), ns.end()), ns.end());
        if (ns.size() == 1) cout << abs(ns.front()) << '\n';
        else {
            int g = ns[1] - ns[0];
            for (int i = 2; i < ns.size(); i++)
                g = gcd(g, ns[i] - ns[i - 1]);
            cout << g << '\n';
        }
        ns.clear();
    }
    return 0;
}

```

强化练习: 408 Uniform Generator^A, 412 Pi^A, [10139](#) Factovisors^A, [11827](#) Maximum GCD^A, [12068](#) Harmonic Mean^C, [12708](#) GCD The Largest^B。

扩展练习: [1642*](#) Magical GCD^D, [10368](#) Euclid's Game^B, [10951](#) Polynomial GCD^D。

如果两个正数 a 和 b 的最大公约数为 1, 则称 a 和 b 互素 (relative prime, 或称互质)。很显然, 两个不同素数的最大公约数为 1, 有时两个非素数的最大公约数也可能为 1, 例如 4 和 9。大于 1 且相邻的两个自然数总是互素的。互素的两个数具有以下性质: 设 $a < b$, 则 ka ($1 \leq k \leq b$) 除以 b 的余数会取遍 0 至 $b-1$ 且不会发生重复。例如 5 和 7 互为素数, 则 $5k$ ($1 \leq k \leq 7$) 除以 7 的余数依次为: 5、3、1、6、4、2、0, 取遍了 0 至 6 的余数值。

强化练习: [571*](#) Jugs^B。

7.2.2 扩展欧几里得算法

使用欧几里得算法可以求得最大公约数, 对其进行适当扩展可以求解形如

$$ax + by = c \quad (a, b, c \in \mathbb{Z}) \quad (7.6)$$

的不定方程的整数解。事实上, 只有 $\gcd(a, b)$ 能够整除 c 时, 方程 (7.6) 才可能有整数解, 因为将方程两边同时除以 $\gcd(a, b)$ 可得

$$\frac{a}{\gcd(a, b)}x + \frac{b}{\gcd(a, b)}y = \frac{c}{\gcd(a, b)} \quad (7.7)$$

如果存在整数解, 则方程 (7.7) 的左边仍然是整数, 要求方程 (7.7) 的右边同样是整数, 即要求 $\gcd(a, b)|c$ 。

为了求解方程 (7.6), 首先构造一个新的不定方程

$$ax + by = \gcd(a, b) \quad (7.8)$$

令 $a' = b$, $b' = a \bmod b$, 有

$$a'x' + b'y' = \gcd(a', b') = \gcd(b, a \bmod b) \quad (7.9)$$

结合欧几里得算法中的等式

$$\gcd(a, b) = \gcd(b, a \bmod b)$$

可以得到

$$ax + by = a'x' + b'y' = bx' + (a \bmod b)y' = bx' + \left(a - \left\lfloor \frac{a}{b} \right\rfloor \cdot b\right)y' \quad (7.10)$$

整理可得

$$ax + by = ay' + b\left(x' - \left\lfloor \frac{a}{b} \right\rfloor y'\right) \quad (7.11)$$

根据多项式恒等定理, 不定方程 (7.8) 的解可以表示为

$$\begin{cases} x = y' \\ y = x' - \left\lfloor \frac{a}{b} \right\rfloor y' \end{cases} \quad (7.12)$$

也就是说, 知道了不定方程 (7.9) 的解, 就可以得到不定方程 (7.8) 的解。可以对不定方程 (7.9) 继续上述过程直到 $\gcd(a', b')$ 中的 $b'=0$, 此时 $\gcd(a', 0) = |a'|$, 可以解得

$$\begin{cases} x' = 1, a' \geq 0 \text{ 或 } x' = -1, a' < 0 \\ y' = 0 \end{cases}$$

再通过此解不断往回代入即可得到中间各个方程的解, 从而最终得到方程 (7.8) 的解, 由于 $\gcd(a, b)$ 整除 c , 将方程 (7.8) 的解乘以 $c/\gcd(a, b)$ 即可得到不定方程 (7.6) 的解。

扩展欧几里得算法的应用非常广泛, 可以应用于求解不定方程、线性同余方程、模的逆元等。以下是扩展欧几里得算法的递归和迭代实现, 其中递归实现相较于迭代实现更容易理解和编写。

```
-----7.2.2.cpp-----
// 递归实现。
void extgcd(int a, int b, int &x, int &y)
{
    if (b == 0) x = 1, y = 0;
    else {
        extgcd(b, a % b, x, y);
        int t = x - a / b * y;
        x = y, y = t;
    }
}

// 迭代实现。
void extgcd(int a, int b, int &x, int &y)
{
    int x0, y0, x1, y1, r, q;

    x0 = 1, y0 = 0, x1 = 0, y1 = 1;
    x = 0, y = 1;
    r = a % b, q = (a - r) / b;

    while (r) {
        x = x0 - q * x1, y = y0 - q * y1;
        x0 = x1, y0 = y1, x1 = x, y1 = y;
        a = b, b = r;
        r = a % b, q = (a - r) / b;
    }
}
-----7.2.2.cpp-----
```

7.2.3 线性同余方程

称形式类似于

$$ax \equiv c \pmod{b} \quad (a, b, c, x \in \mathbb{Z}) \quad (7.13)$$

的方程为线性同余方程（又称一次同余方程，因为在同余方程中，未知数的幂仅为一次）。显然当 $a=0$ 时，只有 $c=0$ 时，同余方程 (7.13) 才有解，此时任意整数 x 均为其解。若 $a \neq 0$ ，则可将其转化为二元一次不定方程，进而使用扩展欧几里得算法解决。求解同余方程 (7.13) 等价于求解

$$ax + by = c \quad (a, b, c, x, y \in \mathbb{Z}) \quad (7.14)$$

令 $d = \gcd(a, b)$ ，根据前述求解不定方程得到的结论，只有当 $d \mid c$ 时，不定方程 (7.14) 才有解，且有 d 个不同的基本解。由扩展欧几里得算法求出不定方程 (7.14) 的一个基本解 x_0 之后，则同余方程 (7.13) 的所有模 b 且互不同余的基本解 x 可以表示为

$$x = x_0 + \frac{bt}{d}, \quad t = 0, 1, \dots, d-1$$

10090 Marbles^B (弹珠)

我有 n 颗弹珠（小玻璃球）并且打算买一些盒子来装下它们。共有两种类型的盒子可供购买：类型 1——每个盒子花费为 c_1 塔卡^I，可以装下 n_1 颗弹珠；类型 2——每个盒子花费为 c_2 塔卡，可以装下 n_2 颗弹珠。

我希望购买的盒子都能够装满弹珠而且使得购买费用最少。因为确定如何将弹珠分装到这些盒子中对于我来说不是件容易的事，所以我寻求你的帮助，同时我也希望你的程序能够高效地得出答案。

输入

输入包含多组测试数据。每组测试数据的第一行包含一个整数 n ($1 \leq n \leq 2000000000$)，第二行包含 c_1 和 n_1 ，第三行包含 c_2 和 n_2 。 c_1, c_2, n_1, n_2 均为正整数且其值小于 2000000000 。输入最后一行 n 为 0，表示输入结束。

输出

对于每组测试数据，输出最小花费的解决方案（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，输出“failed”。若存在解决方案，你可以假定解总是唯一的。

样例输入

```
43
1 3
2 4
40
5 9
5 12
0
```

样例输出

```
13 1
failed
```

分析

令第一种盒子的数量为 x ，第二种盒子的数量为 y ，则本题可以转化为求解不定方程

$$n_1x + n_2y = n$$

的正整数解问题，且要求得到的解能够使得表达式 $c_1x + c_2y$ 的值最小。首先需要根据情况判断不定方程是否有解，若有解，则先求同余式 $n_1x \equiv n \pmod{n_2}$ 的解。若 $\gcd(n_1, n_2) = 1$ ，该同余式在模 n_2 的意义下有唯一解，即扩展欧几里得算法输出的 x' 和 y' ，前述同余式的解为

^I Taka, 塔卡，孟加拉国货币单位。

$$x = \frac{nx'}{\gcd(n_1, n_2)} + \frac{kn_2}{\gcd(n_1, n_2)}, \quad y = \frac{ny'}{\gcd(n_1, n_2)} - \frac{kn_1}{\gcd(n_1, n_2)}, \quad k \in \mathbb{Z}$$

若有 $c_1n_2 < c_2n_1$, 则 x 和 y 在保证为正整数的情况下 x 越大, 花费约少, 若 $c_1n_2 > c_2n_1$, 则 x 应为最小的正整数则花费较少, 若相等, 则任意满足条件的 x, y 花费相同。为什么会是这样呢? 因为假设有 M 颗弹珠, 全部用第一种规格的盒子装, 花费是 $M/(n_1c_1)$, 若用第二种规格的盒子装, 则花费为 $M/(n_2c_2)$, 若有 $M/(n_1c_1) < M/(n_2c_2)$, 则用第一种规格的盒子越多, 花费越省, 化简可以得到: $c_1n_2 < c_2n_1$, 反之 $c_1n_2 > c_2n_1$, 则用第二种规格的盒子越多, 花费越省, 若相等, 则任意满足方程的非负整数 x, y 所产生的花费都是相等的。注意本题中数据类型的使用, 因为 $1 \leq n \leq 2000000000$, 所以最好使用 `long long int` 型数据以避免中间计算结果溢出而导致错误的结果。

强化练习: [10104 Euclid Problem^A](#), [10673 Play with Floor and Ceil^A](#)。

7.2.4 最小公倍数

给定两个正整数 a 和 b , a 和 b 的最小公倍数 (least common multiple) 定义为能够同时被 a 和 b 的整除的最小正整数, 记为 $\text{lcm}(a, b)$, 有

$$\text{lcm}(a, b) = \min\{k \mid k > 0, a|k \text{ 且 } b|k\}$$

将 a 和 b 表示为素数的乘积

$$a = 2^{x_1} \times 3^{x_2} \times 5^{x_3} \dots, \quad b = 2^{y_1} \times 3^{y_2} \times 5^{y_3} \dots$$

那么很显然

$$\text{lcm}(a, b) = 2^{\max(x_1, y_1)} \times 3^{\max(x_2, y_2)} \times 5^{\max(x_3, y_3)} \dots$$

也就是说, 最大公倍数的素因子次数是 a 和 b 对应素因子次数的最大值, 这个性质在求多个数的最小公倍数时很有帮助。结合最大公约数的素因子分解式, 有 $\text{gcd}(a, b) \leq \text{lcm}(a, b)$ 且 $\text{gcd}(a, b)$ 必定能够整除 $\text{lcm}(a, b)$, 同时可以得到

$$\text{lcm}(a, b) \cdot \text{gcd}(a, b) = ab, \quad \text{lcm}(a, b) = \frac{ab}{\text{gcd}(a, b)}$$

10680 LCM^B (最小公倍数)

大家应该都知道最小公倍数的概念。例如, 4 和 6 的最小公倍数是 12。对于两个以上的整数也可以定义它们的最小公倍数, 例如, 2, 3, 5 的最小公倍数为 30。类似的, 我们可以对前 N 个自然数定义最小公倍数, 例如, 前 6 个自然数的最小公倍数为 60。可以预见, 前 N 个自然数的公倍数增长将会很快, 所以我们感兴趣的并不是其精确值, 而只是其最后一位非零的数位值。你必须编写程序高效地把它找出来。

输入

输入中每行包含一个最大不超过 1000000 的非零正整数。输入最后一行为 0, 提示输入结束, 不需处理该行。你最多需要处理 1000 行输入。

输出

对于每行输入, 输出 1 到 N 的最小公倍数的最后一位非零数位值。

样例输入

3	6
0	

样例输出

分析

设整数 1 到 N 的最小公倍数为 M , 对 M 进行素因子分解, 任取素因子 p , 令其幂次为 k , 根据最小公倍数的定义, 取 1 到 N 的任意一个整数, 将其进行素因子分解后, p 的次数 k' , 定有 $k' \leq k$, 而且有 $p^{k'} \leq N$ 。也就是说, 只需求出小于 N 的素数 p 的最大次幂即可得到 M 素因子分解式中所包含的该素数的幂次, 进而根据素因子 p 的幂次就能够获得最后非零数位的值。在这里需要注意的是对素因子 2 和 5 的处理, 因为素因子 2 和 5 相乘会产生 0 数位, 如果只是简单的通过模 10 操作取非 0 末位, 所得到的值可能并不正确, 例如 $5 \times 5 \times 2$ 和 $5 \times 2 \times 5$, 5×5 模 10 为 5, 继续乘 2 模 10 取非 0 数位为 1, 而 5×2 模 10 取非 0 数位为 1, 再乘 5 模 10 为 5, 结果不一致, 正确的结果应该为 5。此时, 可以先处理 2 和 5 的幂次。因为对于同一个数 N 来说, 当 $2^x \leq N$, $5^y \leq N$ 时, 必定有 $x \geq y$, 而同等数量的 2 和 5 相乘最后的非零数位值为 1, 只需考虑 2^{x-y} 的最后数位即可。对于其他的素因子, 由于相乘不会产生 0 数位, 可以正常处理。

强化练习: [10717 Mint^B](#), [10791 Minimum Sum LCM^B](#), [10892 LCM Cardinality^B](#), [11388 GCD LCM^A](#), [12852 The Miser's Puzzle^A](#)。

扩展练习: [11889 Benefit^B](#)。

7.2.5 欧拉函数

欧拉函数 (Euler's totient function 或 Euler's phi function), 一般记做 φ , 定义为小于等于 n 的正整数中与 n 互素的数的个数。更为形式化的定义是: $\varphi(n)$ 为正整数 k 的个数, $1 \leq k \leq n$, 且 $\gcd(k, n) = 1$ 。很明显, $\varphi(1) = 1$ 。对于任意素数 p , 因为小于 p 的正整数均与 p 互素, 故有 $\varphi(p) = p - 1$ 。对于合数 m 来说, 由于合数除了 1 和本身以外至少还有一个其他因子, 故有 $\varphi(m) \leq m - 2$ 。

如果 p 为素数, 令 $m = p^k$, 由于 p 是 m 的唯一素因子, 则对小于 p^k 的正整数 n , 如果 n 不能整除 p^k , 则必定 p 也不能整除 n , 那么除了 p 的倍数外, 其他数均与 p^k 互素, 小于 p^k 的 p 的倍数有 $\{p, 2p, 3p, \dots, p^{k-1}p\}$, 总共 p^{k-1} 个, 则

$$\varphi(p^k) = p^k - p^{k-1}, \quad p \text{ 为素数}, \quad k \geq 1$$

在数学中, 对于定义域为正整数的函数 $f(x)$ 来说, 如果 $f(1) = 1$, 且有

$$f(x_1 x_2) = f(x_1) f(x_2), \quad \gcd(x_1, x_2) = 1$$

那么称函数 $f(x)$ 为积性的 (multiplicative), 例如函数 $f(x) = x^2$ 。可以证明欧拉函数也是积性函数, 也就是说对于正整数 m 和 n 来说, 如果 $\gcd(m, n) = 1$, 有

$$\varphi(mn) = \varphi(m)\varphi(n), \quad \gcd(m, n) = 1$$

对于任意正整数 m , 可将其进行素因子分解为

$$m = p_1^{m_1} \times p_2^{m_2} \times \dots \times p_k^{m_k}$$

而对于任意两个素数 p_1 和 p_2 , 有 $\gcd(p_1, p_2) = 1$, 结合欧拉函数是积性函数的性质, 可以推出

$$\varphi(m) = \prod_{p|m} \varphi(p^{m_p}) = \prod_{p|m} (p^{m_p} - p^{m_p-1}) = \prod_{p|m} p^{m_p} (1 - p^{-1}) = m \prod_{p|m} \left(1 - \frac{1}{p}\right) \quad (7.15)$$

根据等式 (7.15) 即可计算给定整数的欧拉函数值。朴素的方法是先用埃氏筛法求得小于 m 的所有素数, 然后对 m 进行素因子分解, 再根据公式计算欧拉函数值。

```
//++++++7.2.5.cpp+++++++
const int MAXN = 1000000;
int primes[MAXN], cnt = 0;
```

```

int getPhi(int m)
{
    int phi = m;
    // 寻找m的不同素因子, 然后根据公式计算。
    for (int i = 0; i < cnt; i++) {
        if (primes[i] > m) break;
        if (m % primes[i] == 0) {
            while (m % primes[i] == 0)
                m /= primes[i];
            phi -= phi / primes[i];
        }
    }
    // 可能m未被除尽, 则m是素因子之一。
    if (m > 1) phi -= phi / m;
    return phi;
}

```

强化练习: [10179 Irreducible Basic Fractions^A](#), [10299 Relatives^A](#)。

扩展练习: [11064 Number Theory^B](#)。

由于上述计算方式需要不断地寻找给定数的素因子, 而且包含了较多的模运算和除法, 其效率相对较低, 可以使用两种方式予以改进。第一种方式是在筛法求素数的同时使用公式 (7.15) 求欧拉函数值。

```

const int MAXN = 1000000;
int primes[MAXN], phi[MAXN] = {0, 1}, cnt = 0;

void sieve(int *primes, int n, int &cnt)
{
    cnt = 0, iota(phi, phi + n, 0);
    for (int i = 2; i < n; i++)
        if (phi[i] == i) {
            primes[cnt++] = i;
            for (int j = i; j < n; j += i)
                phi[j] -= phi[j] / i;
        }
}

```

强化练习: [10820 Send a Table^B](#)。

第二种方式是利用欧拉函数自身的递推关系, 结合埃氏筛法求素数的过程, 计算欧拉函数值。下面先进行递推关系的推导。对于正整数 m , 可以将其进行素因子分解, 表示成

$$m = p_1^{k_1} \cdot p_2^{k_2} \cdots p_n^{k_n}$$

考虑 $\frac{m}{p_1}$ 的素因子分解情况, 如果 $k_1=1$, 有

$$\frac{m}{p_1} = p_2^{k_2} \cdot p_3^{k_3} \cdots p_n^{k_n}$$

根据欧拉函数计算公式 (7.15) 有

$$\varphi(m) = m \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_n}\right) = \frac{m(p_1 - 1)}{p_1} \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_n}\right)$$

以及

$$\varphi\left(\frac{m}{p_1}\right) = \frac{m}{p_1} \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_n}\right)$$

那么有

$$\varphi(m) = (p_1 - 1)\varphi\left(\frac{m}{p_1}\right)$$

若 $k_1 > 1$, 则

$$\frac{m}{p_1} = p_1^{k_1-1} p_2^{k_2} \cdots p_n^{k_n}$$

有

$$\varphi\left(\frac{m}{p_1}\right) = \frac{m}{p_1} \left(1 - \frac{1}{p_1}\right) \left(1 - \frac{1}{p_2}\right) \cdots \left(1 - \frac{1}{p_n}\right)$$

则

$$\varphi(m) = p_1 \varphi\left(\frac{m}{p_1}\right)$$

也就是说，如果 p_1 能够整除 m/p_1 ，则 m 的欧拉函数值是 m/p_1 欧拉函数值的 p_1 倍，否则是其 p_1-1 倍。这个结论对 m 的任意一个素因子均成立，根据此结论可以在埃氏筛法求素数的过程中，同时求出欧拉函数值。

```

const int MAXN = 1000000;
int primes[MAXN], phi[MAXN] = {0, 1}, cnt = 0;

void sieve(int *primes, int n, int &cnt)
{
    cnt = 0, iota(primes, primes + n, 0);
    for (int i = 2; i < n; i++) {
        if (primes[i] == i) {
            // 素数的欧拉函数值为其值减 1。
            phi[i] = i - 1;
            primes[cnt++] = i;
            // 标记非素数，将 i 的倍数所对应的元素值设置为它的最小素因子。
            for (int j = i + i; j < n; j += i)
                if (primes[j] == j)
                    primes[j] = i;
        }
        else {
            // 非素数保存的是它的最小素因子，根据递推关系计算欧拉函数值。
            int k = i / primes[i];
            if (k % primes[i] == 0) phi[i] = primes[i] * phi[k];
            else phi[i] = (primes[i] - 1) * phi[k];
        }
    }
}
//+++++7.2.5.cpp+++++/

```

强化练习: 13132 Laser Mirrors^E。

扩展练习：12995* Farey Sequence^E。

欧拉函数有一个有趣的性质: n 的所有约数的欧拉函数值之和等于 n , 即

$$\sum_{d|n} \varphi(d) = n$$

例如, $\varphi(6) = \varphi(1) + \varphi(2) + \varphi(3) + \varphi(6) = 1 + 1 + 2 + 2 = 6$ 。

下面介绍一些欧拉函数相关的应用。知道了欧拉函数的值，如何求得小于 n 且与 n 互素的正整数的和呢？根据求最大公约数的欧几里得算法，可以得到

$$\gcd(a, b) = \gcd(a, a - b) = \gcd(b, a - b), \quad a > b > 0$$

也就是说, 如果 k 和 n 互素, $n-k$ 也和 n 互素, 则根据欧拉函数的定义, 小于 n 且与 n 互素的正整数之和为

$$S = \sum_{\substack{k=1 \\ \gcd(k, n)=1}}^n k = \frac{n}{2} \cdot \varphi(n), \quad n \geq 1$$

例如, 与 7 互素的数有 1、2、3、4、5、6, 其和为 $21 = 7/2 \times \varphi(7) = 7/2 \times 6 = 21$ 。计算得到与 n 互素的数之和之后, 小于 n 且与 n 非互素的正整数之和为

$$S = \sum_{\substack{k=1 \\ \gcd(k, n)>1}}^n k = \frac{(1 + (n-1))(n-1)}{2} - \frac{n}{2} \cdot \varphi(n) = \frac{n(n-1-\varphi(n))}{2}, \quad n > 1$$

11417 GCD^A (最大公约数)

给定整数 N , 你需要计算如下的 G 值, 其定义为

$$G = \sum_{i=1}^{i < N} \sum_{j=i+1}^{j \leq N} \text{GCD}(i, j)$$

此处 GCD 的含义为整数 i 和 j 的最大公约数。对于那些理解上述求和公式存在困难的人, 可以将上述公式的意义看成如下的代码形式:

```
G = 0;
for(i = 1; i < N; i++)
    for(j = i + 1; j <= N; j++)
        G += GCD(i, j);    // 函数 GCD 的作用是返回 i 和 j 的最大公约数。
```

输入

输入最多有 100 行, 每行包含一个整数 N ($1 < N < 501$), N 的含义如题意所述。输入最后以一个 0 结束, 该行不需处理。

输出

对于每行输入输出一行, 包含 N 所对应的 G 值。

样例输入

```
10
0
```

样例输出

```
67
```

分析

因为题目中 N 的范围很小, 最朴素的方法就是预先计算 1 到 N 的所有 G 值。为了尽量避免重复计算, 可以利用根据 G 值的定义得到的递推关系

$$G_n = G_{n-1} + \sum_{i=1}^{n-1} \text{GCD}(i, n)$$

参考代码

```
int main(int argc, char *argv[])
{
    int n, sum[501] = {0};
    for (int i = 1; i < 501; i++) {

```

```

    sum[i] += sum[i - 1];
    for (int j = 1; j < i; j++)
        sum[i] += __gcd(j, i);
}
while (cin >> n, n > 0) cout << sum[n] << '\n';
return 0;
}

```

当 N 较小时, 使用上述方式计算并不会超出时间限制。但是当 N 较大时 (例如 $N > 100000$), 肯定会超时, 此时需要寻求其他更为高效的方法。观察递推关系式, 可以发现关键在于如何高效地计算小于 N 的数与 N 的最大公约数。按其公约数大小, 将 $N=20$ 时的“数对”罗列出来可得

$$\begin{aligned}
 \gcd(k, n) = 1, & (1, 20)(3, 20)(7, 20)(9, 20)(11, 20)(13, 20)(17, 20)(19, 20) \\
 \gcd(k, n) = 2, & (2, 20)(6, 20)(14, 20)(18, 20) \\
 \gcd(k, n) = 4, & (4, 20)(8, 20)(12, 20)(16, 20) \\
 \gcd(k, n) = 5, & (5, 20)(15, 20) \\
 \gcd(k, n) = 10, & (10, 20)
 \end{aligned}$$

观察以上公约数的结果, 可以发现公约数为 1 的“数对”实际上是 $N=20$ 时与 N 互素的“数对”乘以 1, 公约数为 2 的“数对”实际上是 $N=10$ 时与 N 互素的“数对”乘以 2, 公约数为 4 的“数对”实际上是 $N=5$ 时与 N 互素的“数对”乘以 4……规律非常明显, 即 1 到 $N-1$ 的数与 N 组成的“数对”的最大公约数之和与 N 的因子的欧拉函数值有关。利用上述规律, 可以得到以下更为高效的代码。

参考代码

```

const int MAXN = 501;
long long phi[MAXN] = {0}, sum[MAXN] = {0};

void preCalculate()
{
    iota(phi, phi + MAXN, 0);
    for (int i = 2; i < MAXN; i++) {
        if (phi[i] == i) {
            for (int j = i; j < MAXN; j += i)
                phi[j] = phi[j] / i * (i - 1);
        }
        for (int j = i, k = 1; j < MAXN; j += i, k++)
            sum[j] += k * phi[i];
    }
}

int main(int argc, char *argv[])
{
    preCalculate();
    for (int i = 1; i < MAXN; i++) sum[i] += sum[i - 1];
    int n;
    while (cin >> n, n > 0)
        cout << sum[n] << '\n';
    return 0;
}

```

强化练习: 11424 GCD Extreme (I)^C, 11426 GCD Extreme (II)^C。

扩展练习: 10990 Another New Function^C, 11317 GCD + LCM^D, 11327 Enumerating Rational

Numbers^C。

7.2.6 莫比乌斯函数

莫比乌斯函数 (Möbius function)^I, 一般记做 μ , 是作莫比乌斯反演的时候一个很重要的系数, 下面给出其定义的来源^{II}。

若 f 是算术函数, F 是它的和函数

$$F(n) = \sum_{d|n} f(d)$$

按照定义分别展开 $F(n)$, $n=1, 2, \dots, 8$, 可得

$$\begin{aligned} F(1) &= f(1) \\ F(2) &= f(1) + f(2) \\ F(3) &= f(1) + f(3) \\ F(4) &= f(1) + f(2) + f(4) \\ F(5) &= f(1) + f(5) \\ F(6) &= f(1) + f(2) + f(3) + f(6) \\ F(7) &= f(1) + f(7) \\ F(8) &= f(1) + f(2) + f(4) + f(8) \end{aligned}$$

从上述方程可以解出 $f(n)$ 在 $n=1, 2, \dots, 8$ 处的值, 即

$$\begin{aligned} f(1) &= F(1) \\ f(2) &= F(2) - F(1) \\ f(3) &= F(3) - F(1) \\ f(4) &= F(4) - F(2) \\ f(5) &= F(5) - F(1) \\ f(6) &= F(6) - F(3) - F(2) + F(1) \\ f(7) &= F(7) - F(1) \\ f(8) &= F(8) - F(4) \end{aligned}$$

注意到 $f(n)$ 等于形式为 $\pm F(n/d)$ 的一些项之和, 其中 $d \mid n$, 从上述结果中, 可能存在一个等式, 其形式类似于

$$f(n) = \sum_{d|n} \mu(d)F(n/d)$$

其中 μ 是算术函数。如果等式成立, 通过计算可以得出: $\mu(1)=1$, $\mu(2)=-1$, $\mu(3)=-1$, $\mu(4)=0$, $\mu(5)=-1$, $\mu(6)=1$, $\mu(7)=-1$, $\mu(8)=0$ 。根据 $F(n)$ 的定义, 可以推导得出 μ 算术函数的若干性质。若 p 是素数, 则有 $F(p)=f(1)+f(p)$, 推出

$$f(p) = F(p) - f(1) = F(p) - F(1) = \mu(1)F\left(\frac{p}{1}\right) + \mu(p)F\left(\frac{p}{p}\right)$$

^I 奥古斯特·费迪南德·莫比乌斯 (August Ferdinand Möbius, 1790—1868), 德国人, 数学家, 理论天文学家。最有名的成果是发现了单侧曲面, 即莫比乌斯带 (Möbius strip) ——把一个纸带旋转半圈再把两端粘上之后即可得到。

^{II} 参阅: Kenneth H. Rosen 著, 夏鸿刚译, 《初等数论及其应用 (第 6 版)》, 北京: 机械工业出版社, 2015, 第 199—200 页。

则有 $\mu(p) = -1$ 。进一步地, 由于

$$F(p^2) = f(1) + f(p) + f(p^2)$$

有

$$f(p^2) = F(p^2) - (F(p) - F(1)) - F(1) = F(p^2) - F(p) = \mu(1)F\left(\frac{p^2}{1}\right) + \mu(p)F\left(\frac{p^2}{p}\right) + \mu(p^2)F\left(\frac{p^2}{p^2}\right)$$

则要求对于任意素数 p , 有 $\mu(p^2)=0$ 。类似的可以推理得出对任意素数 p 及整数 $k>1$, 有 $\mu(p^k)=0$ 。如果猜测 μ 是积性函数, 则 μ 的值就由 n 的素因子分解式中所有素因子的幂值决定, 这就得到莫比乌斯函数 $\mu(n)$ 的定义

$$\mu(n) = \begin{cases} 1 & \text{如果 } n = 1 \\ (-1)^r & \text{如果 } n = p_1 p_2 \cdots p_r, \text{ 其中 } p_i \text{ 为不同的素数} \\ 0 & \text{其他情形} \end{cases}$$

为了便于理解, 这里对定义作进一步的解释。给定大于 1 的正整数 n , 根据算术基本定理, 可以将其唯一分解为素数的乘积, 如果 n 的素因子分解式中有任意一个素因子的指数大于 1, 定义 $\mu(n)=0$, 例如 $8=2^3$, $44=2^2 \times 11$, 故 $\mu(8)=\mu(44)=0$ 。如果素因子分解式中所有因子的指数都为 1, 则计数素因子的个数 r , 如果 r 为偶数, 定义 $\mu(n)=1$, 否则 $\mu(n)=-1$, 例如 $22=2 \times 11$, 因子个数为 2, 是偶数, 故 $\mu(22)=1$, $30=2 \times 3 \times 5$, 因子个数为 3, 是奇数, 故 $\mu(30)=-1$ 。由于正整数 1 比较特殊, 将 $\mu(1)$ 的值定义为 1。计算单个正整数的莫比乌斯函数值, 可以直接采用定义的方式进行计算。

```
/*+++++7.2.6.cpp+++++*/
const int MAXN = 1 << 20;
int primes[MAXN] = {0}, cnt = 0;

int getMobius(int n)
{
    if (n == 1) return 1;
    int divisors = 0;
    for (int i = 0; i < cnt && n > 1; i++) {
        if (n % primes[i] == 0) {
            divisors++;
            int exponent = 0;
            while (n % primes[i] == 0) {
                exponent++;
                n /= primes[i];
            }
            if (exponent > 1) return 0;
        }
    }
    return (divisors & 1) * (-2) + 1;
}
```

莫比乌斯函数具有一个很好的性质，即

$$\sum_{d|n} \mu(d) = [n = 1]$$

即给定正整数 n ，如果 n 不为 1，则 n 的所有因数的莫比乌斯函数值之和为 0，否则为 1。例如，6 的因数有 1、2、3、6，而 $\mu(1)+\mu(2)+\mu(3)+\mu(6)=0$ 。根据此性质可以递推地求出 n 的莫比乌斯函数值。

```
int mobius[MAXN] = {0};
```

```

void getMobius()
{
    for (int i = 1; i < MAXN; i++) {
        int sigma = (i == 1 ? 1 : 0), delta = sigma - mobius[i];
        mobius[i] = delta;
        for (int j = i + 1; j < MAXN; j += i)
            mobius[j] += delta;
    }
}
//++++++++++++++++++++++++++++++++++++++7.2.6.cpp+++++++++++++++++++++++++++++++++++++

```

强化练习: [10738 Riemann vs Mertens^B](#)。

7.3 模算术

当数 a 不能被数 b 整除时, 会有余数产生, 为了表示余数部分, 人们发明了一个记号——“mod”, 称为模 (modulus)。例如, $9 \bmod 5 = 4$, $101 \bmod 3 = 2$ 。特别的, 当数 a 能被数 b 整除时, 有 $a \bmod b = 0$ 。关于模的运算称为模算术 (modular arithmetic), 在实际应用中, 模运算都是针对整数, 特别是正整数, 尽管模的概念可以应用于实数。特别地, 如果数 a 和数 b 关于 m 的模相等, 则记做 $a \equiv b \pmod{m}$ 。

强化练习: [382 Perfection^A](#), [616 Coconuts Revisited^B](#), [974 Kaprekar Numbers^B](#), [10050 Hartals^A](#), [10174 Couple-Bachelor-Spinster Numbers^C](#), [10212 The Last Non-Zero Digit^B](#), [10879 Code Refactoring^A](#), [11313 Gourmet Games^B](#), [11638* Temperature Monitoring^D](#), [11723 Numbering Roads^A](#), [11805 Bafana Bafana^A](#), [11934 Magic Formula^A](#), [12554 A Special Happy Birthday Song^A](#)。

扩展练习: [11247 Income Tax Hazard^C](#)。

7.3.1 整数拆分

使用 C++ 内置的模运算, 可以容易地将一个整数拆分为单个数字或者将其逆序。

```

int rn = 0;
while (n > 0) {
    rn = rn * 10 + n % 10;
    n /= 10;
}

```

强化练习: [256 Quirksome Squares^A](#), [1225 Digit Counting^A](#), [1583 Digit Generator^A](#), [10424 Love Calculator^A](#), [10591 Happy Number^A](#), [10922 2 the 9s^A](#), [10994 Simple Addition^B](#), [11332 Summing Digits^A](#), [11371 Number Theory for Newbies^B](#), [12290 Counting Game^C](#), [12527 Different Digits^A](#), [12895 Armstrong Number^B](#), [13140 Squares, Lists and Digital Sums^D](#)。

7.3.2 可乐兑换

给定 n 瓶可乐, 将可乐喝完后会产生 n 个空瓶, 若假定 m 个空瓶可以兑换一瓶新的可乐 (可以向售货商“借”若干空瓶, 但需要归还同等数量的空瓶), 确定能够兑换的总的可乐瓶数。注意, 新兑换的可乐在喝完后会产生新的空瓶, 这些空瓶也可以继续用来兑换可乐。按照上述假设, 则总共能够喝到的可乐瓶数为

$$T = n + \frac{n}{m-1} = \frac{nm}{m-1}$$

注意, 上式中的除法为整除。理解上述结果的关键是认识到 $m-1$ 个空瓶等价于一瓶可乐, 即使用 $m-1$ 个空瓶, 再向商家“借”一个空瓶, 凑成 m 个空瓶, 兑换得到一瓶可乐, 将可乐喝完会产生一个空瓶, 将此空瓶还给商家即可。

强化练习: [10346 Peter's Smokes^A](#), [11150 Cola^A](#), [11689 Soda Surpler^A](#), [11877 The Coco-Cola Store^A](#)。

7.3.3 模运算规则

在解题中, 需要熟悉的是模的加法、减法、乘法、乘方运算规则。

加法规则: $(x + y) \bmod n = ((x \bmod n) + (y \bmod n)) \bmod n$ 。

减法规则: $(x - y) \bmod n = ((x \bmod n) - (y \bmod n)) \bmod n$, 可以将最后结果加上 n 的正整数倍以便将结果调整为正整数。即: $(x - y) \bmod n = ((x \bmod n) - (y \bmod n) + kn) \bmod n$, $k > 0$ 。

乘法规则: $xy \bmod n = (x \bmod n)(y \bmod n) \bmod n$ 。

乘方规则: $x^y \bmod n = (x \bmod n)^y \bmod n$ 。

在某些题目中, 可能需要计算某个数的幂模另外一个数的结果, 但由于幂次较大, 不便于直接计算, 如果直接使用乘方规则, 则效率为 $O(n)$, 无法在限制时间内获得通过, 需要转而使用 $O(\log n)$ 的计算方法。

```
-----7.3.3.cpp-----
long long modPow(long long n, long long k, long long mod)
{
    if (k == 0) return 1;
    long long r = modPow(n * n % mod, k >> 1, mod);
    if (k & 1) r = r * n % mod;
    return r;
}
-----7.3.3.cpp-----
```

根据模运算规则, 可以得到一些有趣的结论^I。例如, 在学校教育阶段大家都学过检验一个整数是否能被 3 整除, 只需检验该整数各位数相加之和能否被 3 整除即可, 为什么是这样呢? 根据模运算规则, 有同余式 $10 \equiv 1 \pmod{3}$ 成立, 因此有 $10^k \equiv 1 \pmod{3}$ 成立, 则有

$$\begin{aligned} (a_k a_{k-1} \cdots a_2 a_1 a_0)_{10} &= a_k 10^k + a_{k-1} 10^{k-1} + \cdots + a_1 10 + a_0 \\ &\equiv a_k + a_{k-1} + \cdots + a_1 + a_0 \pmod{3} \end{aligned}$$

同样的, 对于 9 来说, 由于 $10 \equiv 1 \pmod{9}$ 成立, 因此有 $10^k \equiv 1 \pmod{9}$ 成立, 则有

$$\begin{aligned} (a_k a_{k-1} \cdots a_2 a_1 a_0)_{10} &= a_k 10^k + a_{k-1} 10^{k-1} + \cdots + a_1 10 + a_0 \\ &\equiv a_k + a_{k-1} + \cdots + a_1 + a_0 \pmod{9} \end{aligned}$$

亦即检验一个整数是否能被 9 整除, 只需检验该整数各位数相加之和能否被 9 整除即可。类似的, 因为 $10 \equiv -1 \pmod{11}$, 有

$$\begin{aligned} (a_k a_{k-1} \cdots a_2 a_1 a_0)_{10} &= a_k 10^k + a_{k-1} 10^{k-1} + \cdots + a_1 10 + a_0 \\ &\equiv a_k (-1)^k + a_{k-1} (-1)^{k-1} + \cdots + a_2 - a_1 + a_0 \pmod{11} \end{aligned}$$

这表明 $(a_k a_{k-1} \cdots a_2 a_1 a_0)_{10}$ 能被 11 整除的充要条件是对 n 的各位数字交替相加减, 所得到的整数 $a_0 - a_1 + a_2 - \cdots + (-1)^k a_k$ 能被 11 整除。

强化练习: [188 Perfect Hash^B](#), [1230 MODEX^B](#), [10127 Ones^A](#), [10162 Last Digit^B](#), [10489 Boxes of Chocolates^A](#), [10515 Powers Et Al.^A](#), [13216 Problem With A Ridiculously Long Name But With A Ridiculously Short Description^A](#)。

扩展练习: [1069* Always an Integer^D](#), [10710 Chinese Shuffle^D](#), [11036 Eventually Periodic Sequence^D](#),

^I 关于同余在整除性检验中的更多应用可以参阅: Kenneth H. Rosen 著, 夏鸿刚译, 《初等数论及其应用, 第 6 版》, 第 139—142 页。

11609 Teams^C, 11718 Fantasy of a Summation^C, 12318 Digital Roulette^D。

7.3.4 模的逆元

两个整数相除求模的处理较为特殊, 不存在与前述类似的模运算规则, 例如, $2 \equiv 8 \pmod{10}$, 但是 $2/2 = 1 \neq 8/2 = 4 \pmod{10}$ 。当有除法存在时, 只能在特定情况下将其转化为乘法的求模运算。

若存在正整数 a, b, m , 满足 $a \times b \equiv 1 \pmod{m}$, 则称 b 为 a 在模 m 下的逆元, 一般记作 $a^{-1} \equiv b \pmod{m}$ 。逆元存在以下性质:

$$b^{-1} \equiv a \pmod{m}, \quad x/a \equiv y \times b \pmod{m}$$

根据前述的费马小定理, 对于任意素数 p , 如果整数 a 满足 $1 < a < p$, 有

$$a^{p-1} \equiv 1 \pmod{p}$$

根据上述结论, 有 $a^{p-1} \equiv a \times a^{p-2} \equiv 1 \pmod{p}$, 则 a 的逆元为 a^{p-2} 。假设有 $ab \equiv c \pmod{p}$, 根据逆元的性质有 $ab/a \equiv b \equiv c \times a^{p-2} \pmod{p}$, 该同余式在求解某些模方程时非常有用。

另外一种求逆元的方法是利用扩展欧几里得算法进行求解。给定模数 n , 求 a 模 n 的逆元相当于求解 $ax \equiv 1 \pmod{n}$, 该同余方程可以转化为求解不定方程 $ax + ny = 1$, 利用前述的扩展欧几里得算法, 若 $\gcd(a, n) \neq 1$, 则 a 模 n 的逆元不存在, 否则将扩展欧几里得算法得到的解 x_0 调整到区间 $[0, n-1]$ 即为逆元。

128 Software CRC^A (软件式循环冗余检查)

你在家拥有很多个人计算机的公司工作。你的老板, Penny Pincher 博士^I, 有时候想将这些计算机互相连接起来, 但是又不愿意花钱购买网卡。就在这时, 你无意中提到每台计算机都附带了免费的异步串行接口, Pincher 博士当然不会放过这个省钱的机会, 她指派你编写必要的软件使得这些计算机能够通过这些串口互相通信。

你懂得一些关于网络传输的知识, 知道在信息传输中容易发生错误, 常用的解决方法是在每条发送的数据末尾附加一段错误检查信息。这段错误检查信息(在大多数情况下)能够让接收程序检测出传输中发生的错误。你立马进了图书馆, 借了你能找到的最厚的关于传输方面的参考书, 花掉了整个周末的时间(没有加班费的加班时间)来研究错误检查。

最终你得出 CRC (Cyclic Redundancy Check, 循环冗余检查) 是适用于当前问题的最好解决方案, 并且写了一张便条给 Pincher 博士, 告知她你拟将提出的错误检查解决方案的工作机制细节。

CRC 生成

将被传输的信息视为一个长的正二进制数, 信息的第一个字节作为该二进制数的最高位, 第二个字节作为次高位, 按此约定, 这个二进制数可以称之为“ m ”(对于整段信息来说)。在进行传输时, 除了传输这“ m ”个字节外, 还需要传输两个额外字节, 总共是 $m+2$ 个字节, 这两个额外字节是此段信息的两字节 CRC 码。

通过适当选择 CRC 码, 最终构成的信息“ $m2$ ”能够被一个 16 位的特定值 g 整除。这使得接收程序能够很容易确定在传送过程中是否发生了传输错误, 接收程序只需将接收到的信息除以 g , 若余数为 0, 则认为在传输过程中没有发生错误。

你注意到在多数参考书中建议的 g 值均为奇数, 但各不相同, 你选定 34943 作为“ g ”的值(生成器的值)。

^I 姓名含义为“吝啬鬼”, 为命题人设置的幽默。

输入与输出

你设计了一个算法，为可能发送的信息计算其对应的 CRC 码。为了测试该算法，你要编写了一个测试程序，该程序从输入中读入一行信息（不包括行末的换行符），为该行信息计算 CRC 码，并将 CRC 码以十六进制进行输出。每行输入包含的 ASCII 字符不超过 1024 个。输入最后以第一列为字符‘#’的一行结束。注意，在输出 CRC 码时，其数值范围为十进制的 0 到 34942。

样例输入

```
this is a test
A
#
```

样例输出

```
77 FD
0C 86
```

分析

给定的信息是一个字符序列，不妨令其为 $C = \langle c_1, c_2, \dots, c_n \rangle$ ，则信息对应的二进制数为 $c_1 c_2 \dots c_n$ ，其中 c_1 为最高位，令附加的两个 CRC 码字节为 $x_1 x_2$ ，则最后形成的二进制数为 $B = c_1 c_2 \dots c_n x_1 x_2$ ，由于二进制数 B 中的每一个数位均为 1 个字节，即 8 个位，那么可以按照 2^8 进制，即 256 进制进行处理，最终二进制数 B 转换为十进制数为

$$B = c_1 c_2 \dots c_n x_1 x_2 = c_1 \cdot 2^{8(n+1)} + c_2 \cdot 2^{8n} + \dots + c_n \cdot 2^{16} + x_1 \cdot 2^8 + x_2 \cdot 2^0$$

由于 $g = 34943$ ，且有 $B \bmod g = 0$ ，那么根据模运算的相应规则，有

$$\begin{aligned} & (c_1 \cdot 2^{8(n+1)} + c_2 \cdot 2^{8n} + \dots + c_n \cdot 2^{16} + x_1 \cdot 2^8 + x_2 \cdot 2^0) \bmod g = \\ & ((c_1 \cdot 2^{8(n+1)} + c_2 \cdot 2^{8n} + \dots + c_n \cdot 2^{16}) \bmod g) + ((x_1 \cdot 2^8 + x_2 \cdot 2^0) \bmod g) \bmod g = 0 \end{aligned}$$

只要求出 $(c_1 \cdot 2^{8(n+1)} + c_2 \cdot 2^{8n} + \dots + c_n \cdot 2^{16}) \bmod g$ ，则根据模算术规则及余数为 0 的结果，可以求出 $x_1 \cdot 2^8 + x_2 \cdot 2^0$ 。为了方便解题，可以预先计算 2 的相应乘方模 g 的值以备用。

强化练习：[374 Big Mod^E](#)，[550 Multiplying by Rotation^B](#)，[568 Just the Facts^A](#)，[641 Do the Untwist^B](#)，[700 Date Bugs^B](#)。

7.3.5 离散对数

离散对数 (discrete logarithm) 问题：给定正整数 a, b, p ，其中 p 是素数且 $\gcd(a, p) = 1$ ，确定满足同余式

$$a^x \equiv b \pmod{p}$$

的 x 值，要求 x 为非负整数。由于 p 为素数，由数论的相关结论可知， x 必定有解，且最小解小于 p^1 。

^I x 必定有解的证明，请读者查阅数论相关的教材以获得进一步的了解。此处仅简要说明在 x 有解的情况下，必定有最小解。由于 $\gcd(a, p) = 1$ ，由欧拉定理，有

$$a^{\varphi(p)} \equiv 1 \pmod{p} \Rightarrow a^{y\varphi(p)} \equiv 1 \pmod{p}, \quad y \in \mathbb{N}^*$$

而

$$x \bmod \varphi(p) = x - y\varphi(p) < \varphi(p) < p, \quad y \in \mathbb{N}^*$$

考虑到

$$a^{x \bmod \varphi(p)} \equiv \frac{a^x}{a^{y\varphi(p)}} \equiv a^x \pmod{p}, \quad y \in \mathbb{N}^*$$

故 x 必有解且有

$$x < \varphi(p) < p$$

可以证明, 当 p 为素数时, 正整数 a 的 1 次幂到 $p-1$ 次幂模 p 的值互不相同, 亦即模 p 的值取遍 1 到 $p-1$ 。例如, 取 $p=5$, $a=2$, 有

$$2^1 \equiv 2 \pmod{5}, 2^2 \equiv 4 \pmod{5}, 2^3 \equiv 3 \pmod{5}, 2^4 \equiv 1 \pmod{5}$$

根据上述结论, 可以应用小步大步 (baby-step and giant-step) 算法来求解 x 。此算法的基本思想为分块处理, 以便于提高搜索的效率。令 $s=\lceil \sqrt{p} \rceil$, 则可以将 x 表示为^I

$$x = g \times s + r, 0 \leq g < s, 0 \leq r < s$$

那么有 $a^x = a^{g \times s} \times a^r$ 。通过枚举 r 的值, 可以得到 a^r , 将其放入一个有序表中 (例如 STL 中的 map 数据结构), 使得 r 和 a^r 形成映射。然后从 0 开始, 从小到大枚举 g 的值, 检查是否有 a^r 满足

$$a^{g \times s} \times a^r \equiv b \pmod{p}$$

由于 p 为素数, $a^{g \times s}$ 存在模 p 的逆元 $(a^{g \times s})^{p-2}$, 相当于检查是否有 a^r 满足

$$a^r \equiv b \times (a^{g \times s})^{p-2} \pmod{p}$$

若能找到这样的 a^r , 则停止枚举。由于 r 和 a^r 构成映射, 由 a^r 可得到 r , 进而可由 $x=g \times s + r$ 得到最小解。在具体计算 $(a^{g \times s})^{p-2}$ 时, 可以使用快速幂技巧以提高计算效率。

```
//-----7.3.5.cpp-----
typedef long long ll;

ll modPow(ll n, ll k, ll mod)
{
    if (k == 0) return 1LL % mod;
    ll r = modPow(n, k >> 1, mod);
    r = r * r % mod;
    if (k & 1) r = r * n % mod;
    return r;
}

ll modLog(ll x, ll n, ll p)
{
    map<ll, ll> hash;
    ll s = (ll)sqrt((double)p);
    while (s * s <= p) s++;
    ll baby = 1;
    for (ll r = 0; r < s; r++) hash[baby] = r, baby = baby * x % p;
    ll giant = 1;
    for (ll b = 0; b < s; b++) {
        ll xr = n * modPow(giant, p - 2, p) % p;
        if (hash.count(xr)) return b * s + hash[xr];
        giant = giant * baby % p;
    }
}
```

^I 亦可将 x 定义为

$$x = g \times s - r, 1 \leq g \leq s, 0 \leq r < s, s = \lceil \sqrt{p} \rceil$$

这样可以避免计算模的逆元, 因为根据模运算的乘法规则, 有

$$a^x \equiv a^{g \times s - r} \equiv b \pmod{p} \Rightarrow a^{g \times s} \equiv b a^r \pmod{p}$$

只需在初始时将 $b a^r$ 存入有序表与 r 对应即可, 然后枚举 g 来查找 $a^{g \times s}$ 是否在有序表中存在。注意, 在此种情形下, 需要从 1 开始枚举 g 。此种分解方式与正文中的分解方式无本质差别, 目标均在于不重复地生成从 0 到 $p-1$ 的所有整数。

```

    }
    return -1;
}
//-----7.3.5.cpp-----//

```

强化练习：12792* Shuffled Deck^D。

扩展练习：11916* Emoogle Grid^D。

7.3.6 中国剩余定理

大约在公元100年成书的《孙子算经》^I提出了如下问题：

今有物不知其数，三三数之剩二，五五数之剩三，七七数之剩二，问物几何？

翻译成现代数学语言就是求解下列一元同余方程组

$$\begin{aligned}x &\equiv 2 \pmod{3} \\x &\equiv 3 \pmod{5} \\x &\equiv 2 \pmod{7}\end{aligned}$$

《孙子算经》中给出了答案“二十三”以及具体的解法。将该解法一般化并进行形式化地表述即为中国剩余定理 (Chinese Remainder Theorem, CRT)：设有正整数 m_1, m_2, \dots, m_k 两两互素，则同余方程组

$$\begin{aligned}x &\equiv a_1 \pmod{m_1} \\x &\equiv a_2 \pmod{m_2} \\&\dots \\x &\equiv a_k \pmod{m_k}\end{aligned}$$

有整数解，并且在模 $M = m_1 \times m_2 \times \dots \times m_k$ 下的解是唯一的，解为

$$x \equiv (a_1 M_1 M_1^{-1} + a_2 M_2 M_2^{-1} + \dots + a_k M_k M_k^{-1}) \pmod{M}$$

其中 $M_i = M/m_i = m_1 m_2 \dots m_{i-1} m_{i+1} \dots m_k$, M_i^{-1} 为 M_i 模 m_i 的逆元， $1 \leq i \leq k$ 。

中国剩余定理并不是一种算法，它是一种描述性定理，可以为解题提供方法和思路。根据该定理，结合扩展欧几里得算法，可以容易地求得同余方程组的解。

```

//-----7.3.6.cpp-----//
int CRT(int a[], int m[], int n)
{
    int M = 1, r = 0;
    for (int i = 0; i < n; i++) M *= m[i];
    for (int i = 0, Mi, x, y; i < n; i++) {
        Mi = M / m[i];
        // 根据扩展欧几里得算法求 Mi 模 m[i] 的逆元。
        extgcd(Mi, m[i], x, y);
        r = (r + a[i] * Mi * x) % M;
    }
    if (r < 0) r += M;
    return r;
}
//-----7.3.6.cpp-----//

```

强化练习：756* Biorhythms^B。

^I 《孙子算经》和《孙子兵法》的作者不是同一人。《孙子兵法》的作者是春秋时期的孙武，而《孙子算经》的作者及生平均不详，只能从书名推断得知该书与一位名叫孙子的人有关。

扩展练习: [11754* Code Feat^D](#)。

7.3.7 波拉德 ρ 启发式因子分解算法

对于较大的整数来说, 使用前述介绍的素因子分解方法, 效率不是很高。波拉德^I于 1975 年提出了一种因子分解的“奇特”方法——波拉德 ρ 启发式因子分解算法(以下简称“波拉德 ρ 算法”)^[76]。

令 n 是一个大合数, p 是它的最小素因子, 选取若干正整数 $x_0, x_1, x_2, \dots, x_s$, 使得它们有不同的模 n 最小非负剩余, 但它们模 p 的最小非负剩余不是全部不同的。通过概率公式可以证明, 在 s 与 \sqrt{p} 相比时较大, 与 \sqrt{n} 相比时又较小, 而且正整数 $x_0, x_1, x_2, \dots, x_s$ 随机选取, 这是可能发生的^[77]。

一旦找到整数 x_i 和 x_j , $0 \leq i < j \leq s$, 满足 $x_i \equiv x_j \pmod{p}$, 但 $x_i \not\equiv x_j \pmod{n}$, 则 $\gcd(x_i - x_j, n)$ 是 n 的非平凡因子, 这是因为 p 整除 $x_i - x_j$, 但 n 不整除 $x_i - x_j$, 可以使用欧几里得算法迅速求出 $\gcd(x_i - x_j, n)$ 。然而, 对于每对 (i, j) , $0 \leq i < j \leq s$, 求 $\gcd(x_i - x_j, n)$ 共需要求 $O(s^2)$ 个最大公因子。

可以使用下面的方法寻找这样的整数 x_i 和 x_j : 首先随机选取种子值 x_0 , 而 $f(x)$ 是最高次数大于 1 的整系数多项式, 然后利用递归定义

$$x_{k+1} \equiv f(x_k) \pmod{n}, \quad 0 \leq x_{k+1} < n, \quad k \in \mathbb{Z}_{\geq 0}$$

计算 x_{k+1} 。多项式 $f(x)$ 的选取应该使得有很高的概率在出现重复之前生成适当多的整数 x_i 。经验表明, 多项式 $f(x) = x^2 + 1$ 在这一检验中表现良好。

波拉德 ρ 启发式因子分解算法

波拉德 ρ 算法通过下列步骤重复选择 x_1 和 x_2 , 直到求出一个合适的“数对”—— (x_1, x_2) 。

- (1) 选择 x_1 , x_1 可以通过伪随机数生成器获得(或者手动指定), 称 x_1 为种子;
- (2) 运用一个生成函数得到 x_2 , 使得 n 不能整除 $x_1 - x_2$, 一个常用的生成函数为:

$$x_2 = f(x_1) = x_1^2 + a$$

其中 a 可以选 1 或者通过随机数生成器获得;

(3) 计算 $\gcd(x_1 - x_2, n)$, 如果它不是 1, 则是 n 的一个因子; 如果它是 1, 返回步骤(1) 并用 x_2 代替 x_1 重复算法过程。

以下是波拉德 ρ 启发式因子分解算法的一种参考实现^{II}。

```
//-----7.3.7.cpp-----//
// 利用模运算规则将乘法转换为加法以尽量避免溢出。
long long multiplyMod(long long a, long long b, long long mod)
{
    long long r = 0, m = a % mod;
    while (b) {
        if (b & 1) r = (r + m) % mod;
        m = (m * 2) % mod;
        b >= 1;
    }
}
```

^I 约翰·迈克尔·波拉德 (John Michael Pollard, 1941—), 英国数学家。

^{II} 需要注意, 为了尽可能提高算法获得有效的输出的概率, 在实现时并不是使用前后两个生成的 x 值的差与 n 值求最大公约数, 而是固定一个已经生成的 x 值, 令其为 y , 使用后续生成的 x 值与此固定值 y 的差来求最大公约数, 每当求值的次数达到 2 的幂次数时, 使用新生成的 x 值来更新固定值 y 。

```

    return r;
}

// 波拉德 rho 启发式因子分解算法。
long long pollardRho(long long n, long long c)
{
    // 以系统时间作为随机数生成器的种子。
    srand(time(NULL));
    long long i = 1, k = 2;
    long long x = rand() % n, y = x;
    while (true) {
        i++;
        x = (multiplyMod(x, x, n) + c) % n;
        // __gcd 是 GCC 的一个内置函数，用于求最大公约数。
        long long d = __gcd(abs(y - x), n);
        if (d != 1 && d != n) return d;
        if (y == x) return n;
        if (i == k) y = x, k <= 1;
    }
}
//-----7.3.7.cpp-----

```

因为算法在计算过程中使用的是伪随机数，而伪随机数是有范围的，如果列出算法中所计算的 x 值，就会发现最终要重复某个值，以每个 x 值为图的一个顶点，从上一个 x 值到下一个 x 值连接一条有向边，会得到一个类似于希腊字母 ρ (rho) 形状的图形，因此算法得名为波拉德 ρ 算法。

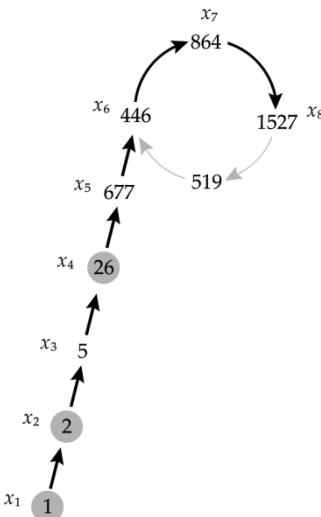


图 7-2 波拉德 ρ 启发式因子分解算法。 $x_{i+1} = (x_i^2 + 1) \bmod 1817$, $x_1 = 1$ 。 $1817 = 23 \cdot 79$ ，黑色箭头指向在因子 79 被发现之前程序执行过程所产生的数，灰色箭头指向在迭代过程中未到达的数 (446 除外)。1817 的素因子在到达 $x_8 = 1527$ 时发现，此时 $\gcd(26 - 1527, 1817) = 79$ ，第一个重复的 x 值为 446

应用波拉德 ρ 算法一般结合米勒-拉宾素性测试进行。对于给定的一个较大的整数，先使用素性测试检查其是否为素数，如果不是素数，则使用波拉德 ρ 算法找出其一个较小的因子，之后继续进行分解，这样可以较为有效地加快因子分解的过程。需要注意，波拉德 ρ 算法是一种随机算法，因此其既不能保证运行

时间也不能保证其运行成功，不过此过程在实际应用中还是非常有效的，其期望时间复杂度为 $O(n^{1/4})$ 。

强化练习：10392* Factoring Large Numbers^A。

7.4 日期和时间转换

7.4.1 日期转换

目前国际通用的日历系统来源于罗马历 (Roman calendar)，罗马历在罗马帝国的建立和衰落过程中经历了数次变革。最初的罗马历称为罗穆拉斯历，据传为罗马神话中罗马城的缔造者罗穆拉斯 (Romulus) 所创立。该历法将一年定为 304 天，分为 10 个月，分别为：Martius (31 天)，Aprilis (30 天)，Maius (31 天)，Iunius (30 天)，Quintilis (31 天)，Sextilis (30 天)，September (30 天)，October (31 天)，November (30 天)，December (30 天)，其中 Martius 被定为第一个月，December 定为第十个月，而在第十个月之后和次年第一个月之前尚有 51 天 (相当于现今的 1 月和 2 月)，并未分配到罗穆拉斯历的任何月份中。

罗马帝国的第二任国王努玛·庞皮留斯 (Numa Pompilius) 对罗穆拉斯历进行了改革。由于当时罗马人认为偶数是不吉利的数字，因此将罗穆拉斯历中每个具有 30 天的月份中抽出一天共 6 天，再加上尚未分配到月份中的天数 51 天，总共 57 天，分为两个月，分别命名为：Ianuarius (29 天)，Februarius (28 天)，同时将原先的第一个月 Martius 定为第三个月，转而将 Ianuarius 定为第一个月，形成了现在的一年十二个月的格局。这样，除了二月份以外，其他月份为 29 天或 31 天，均为奇数，唯独二月份有 28 天，是偶数，按照当时罗马人的信仰，这是一个不吉利的数字，不过二月份被当时的罗马人定为净化月 (month of purification)，并设有涤罪节 (Februia)，因此该月为 28 天虽然不吉利，但也正好与该月份的人们的活动相符。现在的二月份英文单词为 February，来源于拉丁文 “februare”，其含义即为 “净化”。

公元前 45 年，盖乌斯·尤利乌斯·凯撒 (Gaius Julius Caesar)^I再次对历法进行了改革，他采纳古希腊天文学家 (亚历山大港的) 索西琴尼 (Sosigenes of Alexandria) 的建议，将一年定为 365 天，分为十二个月，每四年为一个闰年，闰年的第二个月有 29 天，平年第二个月为 28 天，平均起来每年为 365.25 天，非常接近回归年的实际长度，该历法在历史上被称为凯撒历。在凯撒建立新的历法时，耶稣基督尚未诞生，所以那时候的年号与现今不同，当时采用的是罗马建城纪年。如今日历上公元元年的确立是在公元 525 年由迪奥尼修斯·伊希格斯 (Dionysius Exiguus) 开始，迪奥尼修斯为了能够方便计算耶稣的复活日，将耶稣诞生的那一年定为公元元年，在英文中用 A.D. 表示，A.D. 来源于中世纪的拉丁文 Anno Domini，其意义是 “主的生年”，公元元年之前，用 B.C. 表示，意为 “耶稣之前” (Before Christ)。

罗马天主教皇格列高利十三世 (Pope Gregory XIII) 在凯撒历的基础上进行了改革，对闰年设置做了一些调整，形成了格列高利历 (Gregorian calendar)。按凯撒历，每四百年中有一百个闰日，格列高利历将一百个闰日减为九十七个，能被 4 整除但能被 100 整除的年份不再有闰日，不过能被 400 整除的年份仍有闰日。格列高利十三世做出此项调整出于宗教原因，也是为了精确计算耶稣复活日。因为一个回归年为 365.24219 天，凯撒历到公元 1582 年已经累积了十天的误差，与实际的季节相比产生了差异，为了消除积累的误差，格列高利十三世下令公元 1582 年 10 月 4 日 (凯撒历) 的下一天为公元 1582 年 10 月 15 日 (格列高利历)，而星期继续保持连续不变，凯撒历的 1582 年 10 月 4 日为星期四，因此格列高利历 1582 年 10 月

^I 盖乌斯·尤利乌斯·凯撒 (拉丁文：Gaius Julius Caesar，前 100 年—前 44 年)，罗马共和国末期的军事统帅、政治家，罗马共和体制转向罗马帝国的关键人物，欧洲史称 “凯撒大帝” 及 “罗马共和国的独裁者”。

15 为星期五。经过这样的调整便完成了从凯撒历到格列高利历的更替，同时产生了这样的一个效应——公元 1582 年 10 月 5 日至 14 日这十天“凭空消失”了。其实是因为两种历法版本的更替，这十天从来也没有存在过，所以又称作“消失的十天”。修改置闰法则后，格列高利历的平均历年长度为 365.2422 天，非常接近真实的回归年长度，可以说已经相当精确。

历史上各国采用格列高利历的时间不同，因此也造成在相当长一段时间内，各国使用的历法表示的日期并不统一，例如英国和美国在 1752 年采用，中国在 1912 年采用，俄罗斯在 1918 年才采用，这导致某些历史事件的发生日期记录稍有差异。不过到今天，格列高利历已被世界广泛采用，成为国际通用历法。

强化练习：505 Moscow Time^D, 518 Time^D, 10070 Leap Year or Not Leap Year^A, 10942 Can of Beans^D, 13275 Leap Birthdays^C。

给定一个日期，求经过指定天数后的日期，朴素的方法是将日期转换为天数，加上经过的天数，再转换为日期。将日期转换为天数的方法是先设定一个基准日期，将其作为第一天，对应的经过的天数定为 1（即当天也算在经过的天数之内，这样便于计算），将其他日期相对于基准日期所经历的天数计算出来。如果将公元 1 年 1 月 1 日定为基准日期，若要将日期 $yyyy-mm-dd$ ($yyyy$ 表示年份，为正整数， mm 表示月，取值为 1 到 12， dd 表示日，取值为 1 到 31) 转换为天数，需要计算在 $yyyy$ 年 1 月 1 日之前所经过的天数，加上在 $yyyy$ 年的 mm 月 1 日之前所经过的天数，最后加上天数 dd 即为相对基准日期所对应的天数值。将相对于基准日期的天数转换为日期的过程与前述步骤正好相反。

```
//-----7.4.1.cpp-----
const int daysOfYear = 365;
const int daysOf4Years = daysOfYear * 4 + 1; // 1461
const int daysOf100Years = daysOf4Years * 25 - 1; // 36524
const int daysOf400Years = daysOf4Years * 100 - 3; // 146097

// 定义某月 1 日之前所经过的天数，区分平年和闰年。
const int daysBeforeMonth[2][13] = {
    {0, 31, 59, 90, 120, 151, 181, 212, 243, 273, 304, 334, 365},
    {0, 31, 60, 91, 121, 152, 182, 213, 244, 274, 305, 335, 366}
};

// 表示日期的结构体。
struct date {
    int yyyy, mm, dd;
    date (int yyyy = 0, int mm = 0, int dd = 0): yyyy(yyyy), mm(mm), dd(dd) {}
};

// 判断给定的年份是否为闰年。
bool isLeapYear(int yyyy)
{
    return ((yyyy % 4 == 0 && yyyy % 100 != 0) || yyyy % 400 == 0);
}

// 以公元元年 1 月 1 日为基准日期，将公元元年以后的日期转换为天数。
int toDays(int yyyy, int mm, int dd)
{
    int days = 0, year = yyyy - 1;
    days += (year / 400) * daysOf400Years, year %= 400;
    days += (year / 100) * daysOf100Years, year %= 100;
    days += (year / 4) * daysOf4Years, year %= 4;
    days += year * daysOfYear;
```

```

    days += daysBeforeMonth[isLeapYear(yyyy)][mm - 1] + dd;
    return days;
};

// 将天数转换为日期。
date toDate(int days)
{
    int yyyy = 0, mm = 0, dd = 0, cntOf100Years, cntOfYear;

    // 确定年份。注意每 100 年和每 4 年转换时的处理。
    yyyy += (days / daysOf400Years) * 400, days %= daysOf400Years;
    yyyy += (cntOf100Years = days / daysOf100Years) * 100, days %= daysOf100Years;
    yyyy += (days / daysOf4Years) * 4, days %= daysOf4Years;
    yyyy += (cntOfYear = days / daysOfYear), days %= daysOfYear;
    if (days == 0) {
        if (cntOf100Years == 4 || cntOfYear == 4) return date(yyyy, 12, 30);
        return date(yyyy, 12, 31);
    }

    // 对于剩余天数为 0 的情况需要根据最后得到的年数进行特殊处理。
    yyyy++;
    int leapYear = isLeapYear(yyyy);
    for (int i = 1; i <= 12; i++)
        if (daysBeforeMonth[leapYear][i] >= days) {
            mm = i;
            dd = days - daysBeforeMonth[leapYear][i - 1];
            break;
        }
    return date(yyyy, mm, dd);
}
//-----7.4.1.cpp-----//

```

893 Y3K Problem^B (3000 年问题)

编写程序确定指定日期 D 经过 N 天后的日期 D' , N 为小于 1000000000 的整数, D 为 3000 年 1 月 1 日前的某个日期。注意在标准的格列高利历中, 除了闰年有 366 天外, 平年为 365 天。能被 4 整除但不能被 100 整除的年份为闰年, 能够被 400 整除的年份也是闰年, 其他年份为平年, 因此 1900 年不是闰年, 1904 年, 1908 年, …, 1996 年是闰年, 2000 年也是闰年。在平年中, 每月的天数依次为: 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31, 在闰年中, 二月份有 29 天。

输入

输入包含多行, 每行包含 4 个由空格分隔的整数, 第一个整数为从指定日期经过的天数 (在 0 和 999999999 之间), 接着是按 $DD\ MM\ YYYY$ 格式给出的日期, 其中 DD 表示日 (1 到 31), MM 表示月 (1 到 12), $YYYY$ 表示年 (1998 至 2999), 上述给出的数值范围均为闭区间。输入的最后一行为四个 0, 表示输入结束。

输出

对于每行输入, 输出一行, 表示输入的日期在经过指定的天数后的日期。输出时的日期格式与输入时的日期格式相同。

样例输入

样例输出

```
1 31 12 2999
0 0 0 0
```

```
1 1 3000
```

分析

先确定一个基准日期，将给定日期转换为距离此基准日期的天数，加上指定的偏移天数后再将天数转换为日期即可。过程中牵涉到简单的整除和模运算，需要注意细节。

参考代码

```
int main(int argc, char *argv[])
{
    int days, dd, mm, yyyy;

    while (cin >> days >> dd >> mm >> yyyy) {
        if (yyyy == 0) break;
        date next = toDate(toDays(yyyy, mm, dd) + days);
        cout << next.dd << ' ' << next.mm << ' ' << next.yyyy << '\n';
    }

    return 0;
}
```

由于 C++ 中并未内建支持日期操作的辅助类库，使用 Java 提供的 `Calendar` 类显得更为简便¹。`Calender` 类是一个抽象类，且 `Calendar` 类的构造方法是 `protected` 的，所以无法使用 `Calendar` 类的构造方法来创建对象，而需使用 Java 的 API 中提供的 `getInstance` 方法来创建对象。

```
// 实例 c 的值为系统的当前时间。
Calender c = Calender.getInstance();
```

`Calendar` 类有两个常用的方法，即 `set` 方法和 `get` 方法，它们的作用是设置 `Calendar` 对象实例的时间。其方法声明为：

```
// 使用年、月、日设置日期。
public final void set(int year, int month, int date);
// 设置日期的字段值。
public void set(int field, int value);

// 获取日期的字段值。
public int get(int field);
```

在 `set` 和 `get` 方法中，参数 `field` 代表需要设置的字段类型，常见类型为：

<code>Calendar.YEAR</code>	年份
<code>Calendar.MONTH</code>	月份
<code>Calendar.DATE</code>	日期，月份的第几天
<code>Calendar.DAY_OF_MONTH</code>	日期，与 <code>Calendar.DATE</code> 字段意义相同
<code>Calender.DAY_OF_YEAR</code>	年度的第几天，从 1 开始计数
<code>Calendar.HOUR</code>	12 小时制的小时数
<code>Calendar.HOUR_OF_DAY</code>	24 小时制的小时数

¹ 参阅：<https://docs.oracle.com/javase/7/docs/api/java/util/Calendar.html>, 2020。

Calendar.MINUTE	分钟
Calendar.SECOND	秒
Calendar.DAY_OF_WEEK	星期几

需要注意的是，在设置或获取 `Calendar` 实例的日期字段值时，月份是从 0 开始计数，年份和天数从 1 开始计数，因此设置日期为 2019 年 12 月 31 日，需要按下述方式设置¹：

```
Calendar c = Calendar.getInstance();
c.set(2019, 11, 31);
```

除了 `set` 和 `get` 方法，`Calendar` 类中还有若干在解题中非常有用的方法，以下列举一二。

```
// 在 Calendar 对象中的某个字段上增加或减少一定的数值，增加时 amount 的值为正，  
// 减少时 amount 的值为负。  
public abstract void add(int field, int amount);  
  
// 判断当前日期对象是否在 when 对象所指定的日期之后，如果是返回 true，否则返回 false。  
public boolean after(Object when);  
  
// 判断当前日期对象是否位于另外一个日期对象之前。  
public boolean before(Object when);
```

12148 Electricity^D (用电量)

为了节省开支，Martin 和 Isa 每天都会将电表的读数记下来，以便了解每天的用电量。但有时候，他们会忘记查看电表，导致记录有些地方不连续。给定一份电表读数的记录，确定能够准确得出用电量的天数及总的用电量。

输入

输入包含多组测试数据。每组测试数据的第一行包含一个整数 N ，表示此组测试数据进行了多少次电表记录， $2 \leq N \leq 10^3$ 。接下来的 N 行中，每行包含四个整数 D ， M ， Y 和 C ，以单个空格分隔，表示记录中的日期 ($1 \leq D \leq 31$)，月份 ($1 \leq M \leq 12$)，年份 ($1900 \leq Y \leq 2100$)，以及当天电表的读数 ($0 \leq C \leq 10^6$)。给定的 N 行记录以日期递增的顺序给出，可能包含闰年。电表的读数严格递增（也就是说，没有两个读数是相同的）。你可以假定给定的所有日期都是合法的。请注意，某年是闰年，是指该年份可以被 4 整除但不能被 100 整除或者能被 400 整除。输入的最后包含一个数字 0，表示输入结束。

输出

对于每组测试数据，输出一行，包含以单个空格分隔两个整数：第一个整数表示能够准确得出用电量的天数，第二个整数表示这些天中用电量的总和。

样例输入

```
5
```

样例输出

```
2 5
```

¹ `Calendar.MONTH` 的返回值和历法种类有关，在格列高利和儒略历法中，一月份所对应的返回值为 0，最后一个月的返回值和历法中一年内包含多少个月份有关。`Calendar.DAY_OF_WEEK` 的返回值是 {SUNDAY=1, MONDAY=2, TUESDAY=3, WEDNESDAY=4, THURSDAY=5, FRIDAY=6, SATURDAY=7} 之一。需要注意，按西方国家的习惯，SUNDAY 是一个星期的第一天。

```
9 9 1979 440
29 10 1979 458
30 10 1979 470
1 11 1979 480
2 11 1979 483
0
```

分析

题意是指输入中两个相邻的日期如果相差 1 天，那么用电量可以准确计算。例如，样例输入中第三行的日期为 1979 年 10 月 29 日，第四行的日期为 1979 年 10 月 30 日，两者相差 1 天，那么这 1 天内的用电量可以准确计算，为 $470 - 458 = 12$ 。解题思路非常简单而直接，将输入中的日期解析为 Java 的 Calendar 对象，使用 add 方法将天数加 1，然后判断日期是否相同即可。

参考代码

```
import java.util.*;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n;
        while ((n = scan.nextInt()) > 0) {
            int days = 0, consumption = 0;
            int D1, M1, Y1, C1, D2, M2, Y2, C2;
            Calendar c = Calendar.getInstance();
            D1 = scan.nextInt(); M1 = scan.nextInt(); Y1 = scan.nextInt();
            C1 = scan.nextInt();
            for (int i = 1; i < n; i++) {
                D2 = scan.nextInt(); M2 = scan.nextInt(); Y2 = scan.nextInt();
                C2 = scan.nextInt();
                c.set(Y1, M1 - 1, D1);
                c.add(Calendar.DAY_OF_MONTH, 1);
                D1 = c.get(Calendar.DAY_OF_MONTH);
                M1 = c.get(Calendar.MONTH) + 1;
                Y1 = c.get(Calendar.YEAR);
                if (D1 == D2 && M1 == M2 && Y1 == Y2) {
                    days++;
                    consumption += C2 - C1;
                }
                D1 = D2; M1 = M2; Y1 = Y2; C1 = C2;
            }
            System.out.printf("%d %d\n", days, consumption);
        }
    }
}
```

强化练习：150 Double Time^D, 300 Maya Calendar^A, 602 What Day Is It^B, 631 Microsoft Calendar^E, 10028 Demerit Points^D, 11219 How Old Are You^A, 11356 Dates^C, 11947 Cancer or Scorpio^B, 12019 Doom's Day Algorithm^A, 13025* Back to the Past^B。

7.4.2 时间转换

在钟表发明之前，人们一般是通过太阳的位置来确定时间，例如当太阳上升到日顶时，定为中午 12 时。在钟表发明后，记录本地时间变得更精确，但是各地的时间仍然是不同的，后期随着交通工具的发展，铁路、

汽车、飞机的出现，使得人们可以在短时间内跨越很长的距离，从而带来时间上的急剧变化——可能在出发的时候还是凌晨，但经过两个小时的旅行到达目的地时已是当地时间的下午。1858 年，为了克服本地时间不同所带来的混乱以及方便时间的统一，意大利数学家 Quirico Filopanti 在其出版的书中首先提出一种基于时区（time zone）的世界性计时系统，但其仅停留在书本上，只在其死后很久才为人所知，未对现实的计时系统产生影响。1879 年，加拿大人 Sandford Fleming 再次提出了以时区为基础的世界计时系统，并不断在各种国际会议上建议采纳他的计时系统。Sandford Fleming 的不断努力和现实需要终于促使世界采用了他的计时系统。1884 年，在华盛顿召开的一次国际经度会议（又称国际子午线会议）上，将全球划分为 24 个时区（东、西各 12 个时区），规定英国（格林尼治天文台旧址）为中时区（零时区）、东 1—12 区，西 1—12 区。每个时区横跨经度 15 度，时间正好是 1 小时。最后的东、西第 12 区各跨经度 7.5 度，以东、西经 180 度为界。每个时区的中央经线上的时间就是这个时区内统一采用的时间，称为区时，相邻两个时区的时间相差 1 小时。每个时区都有一个缩略代码，例如 CCT 表示中国北京时间，EDT 表示美国东部夏令时等。中国全境虽然跨越了五个时区（东五区至东九区），但是使用的是单一时区制，即使用的均是东八区（UTC +8）时间，比协调世界时间（Coordinated Universal Time, UTC）快八个小时。

在有关时间转换的题目中，一般是先将时间转换为协调世界时间，然后根据各个时区的偏移转换为当地时间。在此过程中，可以使用 map 数据结构将时区代码与偏移相关联。

对于钟面时间转换，则一般采取将时间换算为从零点开始经过的分钟数或者秒数，加上偏移后再使用模运算转换为其他时间格式。

10339 Watching Watches^D (监视钟表)

人们常说，不走时的表比每天慢一秒的表报时更准确。不走的表起码每天能够指示两次准确的时间，但是每天慢一秒的表只有每隔 43200 天才能指示一次准确的时间。这种说法适用于老式的具有 12 个小时刻度的手表，它的指针是持续走动的（大多数电子表在停止后就无任何显示了）。给定两块同样的手表，对时到午夜零点，两块表均以各自固定的速率走时，但是每天分别慢 k 秒和 m 秒，那么下次两块表同时指示完全一致的时间是什么时刻？

输入

输入包含多行，每行包含两个不同的非负整数 k 和 m ，其值在 0 到 256 之间，表示两块表每天各自慢多少秒钟。

输出

对于每行输入输出一行，先输出 k, m ，然后输出钟表上的时间，精确到分钟。符合要求的时间范围为 01:00 至 12:59。

样例输入

```
1 2
0 7
```

样例输出

```
1 2 12:00
0 7 10:17
```

分析

对于具有 12 个小时刻度的表来说，每天慢一秒，则只有在慢 12 个小时后才能刚好和走时正确的表所指示时间完全一致，12 个小时为 43200 秒，故每天慢一秒的表每隔 43200 天才指示一次准确的时间。类似的，两块表各自慢 k 秒和 m 秒，则经过

$$d = \frac{43200.0}{\text{abs}(k - m)}$$

天后 (`abs` 表示取绝对值, `d` 为浮点数), 两块表所指示的时间就会再次完全一致。此时对于每天慢 `k` 秒的表来说, 在这 `d` 天的时间, 它一共走了

$$s = d * (24 * 60 * 60 - k)$$

秒的时间, 将其转换为钟面时间即可。使用每天慢 `m` 秒的表来计算会得到同样的结果。

参考代码

```
int main(int argc, char *argv[])
{
    int k, m;
    while (cin >> k >> m) {
        double seconds = fmod(43200.0 / abs(k - m) * (86400.0 - k), 86400.0);
        int minutes = seconds / 60 + 0.5, hours = (minutes / 60) % 12;
        cout << k << ' ' << m << ' ';
        cout << setw(2) << setfill('0') << (hours ? hours : 12) << ':';
        cout << setw(2) << setfill('0') << (minutes % 60) << '\n';
    }
    return 0;
}
```

强化练习: [579 Clock Hands^A](#), [10371 Time Zones^C](#), [10683 The Decadary Watch^B](#), [11650 Mirror Clock^A](#), [11677 Alarm Clock^A](#), [11958 Comming Home^B](#), [12136 Schedule of a Married Man^B](#), [12439 February 29^B](#), [12531 Hours and Minutes^B](#), [12822 Extraordinarily Large LED^D](#)。

7.5 小结

数论可以说是比较难驾驭的内容, 在编程竞赛中, 往往是一个数学结论撑起一道题目。数论的一个重要内容是研究素数的性质, 因此在早期的编程竞赛中经常出现与素数有关的题目, 掌握素数的判定方法, 特别是线性素数筛非常必要。求公约数的欧几里得算法, 其辗转相除求公约数的思想需要掌握, 而由欧几里得算法衍生得到的扩展欧几里得算法是求解一次同余式的一个简便方法。欧拉函数和莫比乌斯函数(包括本书未予介绍的莫比乌斯反演定理)在一些难题中经常出现。

在此基础上, 读者可以逐步扩展自己的数论知识面, 包括但不限于初等数论的原根、平方剩余、二次同余式、二次互反律; 离散数学的群、置换群、循环群; 高等数学中的快速傅里叶变换、卷积、泰勒级数; 线性代数中的矩阵的逆、行列式及运算。

第 8 章 回溯法

路漫漫其修远兮，吾将上下而求索。

——屈原¹，《离骚》

在 UVa OJ 的在线题库中，有一部分题目无法通过直接模拟或者应用既定算法的方式来解决，而是需要搜索所有可能的情形以找到符合条件的解或者最优解，此时可以尝试使用回溯法（backtracking）解题。回溯法本质上是暴力搜索（brute force search），相当于在题目所蕴含的隐式图中进行深度优先遍历（Depth First Search, DFS），由于考虑了所有可能的情形，必定能够得到解，不过大多数情况下程序得到所有解的时间会比较长。由于计算机运算速率的大幅提高，在问题空间不是很大的情况，通过回溯法往往都能够顺利将题目予以解决。更进一步地，对于某些特定的题目，可以预先生成所有解，然后再“打表”提交，而不一定总是“实时”计算问题的解。对于某些搜索空间较大的题目，还需要结合剪枝技巧来进一步缩小搜索空间，提高程序的运行效率，以便能够在规定的时间限制内获得解。

8.1 八皇后问题

八皇后问题（eight queens puzzle）是回溯法中的一个经典入门问题。根据国际象棋的规则，在没有其他棋子阻挡的情况下，皇后能够攻击位于同一行、同一列、同一对角线上的对方棋子。如果在一个 8×8 的国际象棋棋盘上放置 8 个皇后，使得它们相互之间无法攻击对方，那么总共有多少种不同的放置方案呢？为了简化问题的讨论，关于棋盘对称的放置方法也视为不同的方案。图 8-1 给出了两种可行的放置方案。

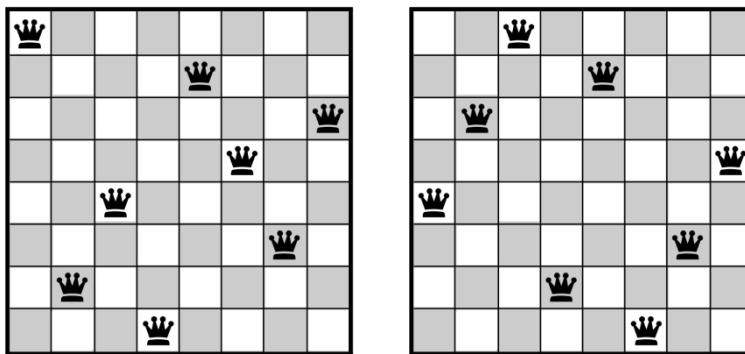


图 8-1 八皇后问题的两种可行放置方案

由于皇后攻击规则的特殊性，无法直接通过组合方法来计数可行的放置方案，而是需要通过遍历所有可能的放置位置来确定放置方案的数量。如前所述，回溯法相当于在隐式图中进行深度优先遍历，因此需要将

¹ 屈原（约前 342 年—前 278 年），名正则，字灵均，又名平，字原，楚武王熊通之子屈瑕的后代，楚国大夫。

题目中的约束条件转换成某种“状态”，这些“状态”和隐式图中的顶点一一对应¹。对于当前“状态”，按照题目给定的规则，能够从当前“状态”衍生出若干后继“状态”，从当前“状态”转移到它的某个后继“状态”就相当于沿着隐式图中的一条有向边从一个顶点到达另外一个顶点。从整体来看，回溯是一个从“初始状态”出发，不断地在“状态”及其后继“状态”之间转移并逐步深入的过程。在此过程中，需要记录已经访问的“状态”，以免重复访问。回溯通过不重复的遍历，最终会达到“终止状态”——满足可行解部分（或全部）要求的一种状态，其所要满足的条件具体由题目所给的约束来确定。此时根据可行解所要满足的条件对“终止状态”进行检查，如果“终止状态”符合可行解的要求，则表明回溯过程找到了一个合法解，将其“记录在案”，接着从“终止状态”回退到上一层的“状态”继续进行搜索。在回溯过程中，可能某条遍历“路径”并不能到达“终止状态”，此时就需要立即回退到上一层的“状态”继续进行搜索。以八皇后问题为例，“状态”就是指棋盘的大小以及已经放置的皇后数量和位置；“初始状态”就是 8×8 的空棋盘；“终止状态”就是 8×8 的棋盘上放置了8个皇后；对“终止状态”进行合法性检查就是检查已经放置的8个皇后是否满足“不能相互攻击”的约束条件。

考虑到棋盘是二维的，可以使用一个二维数组 `board` 来保存棋盘状态，`board[i][j]` 为1表示在棋盘的第 i 行第 j 列放置了一个皇后，为0则表示此位置尚未放置皇后。如果按照朴素的方法逐个位置尝试皇后的放置位置，则由于每行有8个位置，总的搜索空间将是 $8^8 = 16777216$ 种。但是进一步仔细分析的话，由于每一行和每一列只能有一个皇后，则第一行有8个位置可选，第二行有7个位置可选，第三行有6个位置可选……仅考虑避免行和列的攻击时，皇后的可选放置方案数为 $8! = 40320$ 种，只需对这些放置方案进行对角线规则的验证，如果符合则是一种合法的放置方案，搜索空间明显减少。由此可以得到下述的八皇后问题回溯法解题框架。

```
// board 记录具体的放置方案，clnUsed 记录已经使用的列，cnt 记录放置方案数。
int board[8][8] = {}, clnUsed[8] = {}, cnt = 0;

// 使用递归来实现回溯，参数 row 表示当前为第几行选择放置皇后的位置。从 0 开始计数行。
void dfs(int row) {
    // 递归的出口。
    // 如果递归深度已经到达第 8 层，表明已经放置了 8 个皇后，此时可以结束递归。
    // 检查当前放置方案是否满足“对角线”规则，如果满足则是可行解，将方案输出并计数。
    if (row == 8) {
        // checkBoard 检查棋盘状态是否满足“对角线”约束，printBoard 输出棋盘状态。
        if (checkBoard()) { printBoard(); cnt++; return; }
    }
    // 枚举当前行的所有列，检查是否存在满足“行列”约束的列。从 0 开始计数列。
    for (int cln = 0; cln < 8; cln++) {
        // 检查该列是否已经被使用。
        if (clnUsed[cln]) continue;
        // 将未使用的列标记为已使用状态，同时记录放置皇后的位置。
        clnUsed[cln] = 1, board[row][cln] = 1;
        // 递归，进入下一行继续选择可以放置的皇后的列。
        dfs(row + 1);
        // 将已使用的列恢复为未使用状态，以便在递归回退时能够再次使用此列。
    }
}
```

¹ 应用回溯法的题目所蕴含的隐式图一般为有向图。如果读者对图论不太熟悉，那么这段关于回溯法和隐式图深度优先遍历关系的叙述可能不易理解，建议在学习第9章“图遍历”之后再回过头来对这段话进行理解。

```
    clnUsed[cln] = 0, board[row][cln] = 0;  
}
```

根据上述所给出的回溯法实现框架，应该不难编写一个完整的程序来解决八皇后问题。作为练习，请读者先尝试完成代码的编写并进行调试，然后再与后续给出的参考实现进行比较。需要注意的是，在回溯过程中，每使用了一个对应候选位置，都需要予以标记，以便后续将其值还原，为下次使用做准备，否则将导致“遗漏”部分搜索空间，从而无法得到所有可行解。

如何检查棋盘上放置的皇后是否满足对角线规则呢？朴素的方法是逐个枚举皇后所在位置两条对角线上的各个方格，检查这些方格中是否放置了皇后。显然，此种方法效率较低，更为巧妙的是使用图 8-2 所示的方法来进行对角线检查。

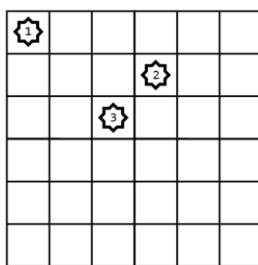


图 8-2 在逐一给每个皇后选择列位置后, 假设第 1 个皇后选择了第 1 列, 第 2 个皇后选择了第 4 列, 第 3 个皇后选择了第 3 列, 此时第 3 个皇后与第 1 个、第 2 个皇后均存在对角线冲突。使用一维数组按行序记录皇后的列选择, 可以表示为 $\{1, 4, 3\}$, 则第 3 个皇后和第 1 个皇后的行序号差的绝对值为 2, 等于两者选择的列序号 3 和 1 差的绝对值, 同时第 3 个皇后和第 2 个皇后的行序号差的绝对值为 1, 与两者选择的列序号 3 和 4 的差的绝对值亦相等。可以证明, 如果存在对角线冲突, 则后续选择的皇后的列序号与已经选择的皇后的列序号差的绝对值会与其相应行序号差的绝对值相等, 与之相反, 如果绝对值不等, 则肯定不存在对角线冲突。可以利用此结论来进行对角线规则的检查

通过进一步地深入思考，还可以应用以下优化技巧：

(1) 由于是逐行考虑皇后可以放置的列，实际上可以在选择状态记录时省去“行”这一维度，只记录列的选择，即将选择状态记录数组缩减为一维，而不是原来的二维。

(2) 由于放置方案可能在回溯过程的中途就已经不满足对角线规则,若能够及早剔除这些“次品”,无疑会提高搜索的效率。可以在确定当前行所选列后立即进行对角线规则检查,而不是总在最后时刻才进行对角线规则检查。与此同时,只对当前行所选列和已选列之间进行对角线规则检查,而不必在所有已选列之间进行对角线规则检查。因为在之前的回溯中已经对这些已选列之间进行了对角线规则检查,不必再次进行检查。

```
-----8.1.1.cpp-----//  
const int MAXN = 8;  
int board[MAXN] = {0}, clnUsed[MAXN] = {0}, cnt = 0;  
  
// 检查当前行所选列与已选列是否满足对角线规则。  
bool checkBoard(int row, int selected) {  
    for (int cln = 0; cln < row; cln++)  
        if (abs(row - cln) == abs(selected - board[cln]))  
            return false;  
    return true;  
}
```

```

    return true;
}

// 输出放置方案，放置皇后的位置以‘Q’表示，未放置皇后的位置以‘*’表示。
void printBoard() {
    for (int row = 0; row < MAXN; row++) {
        for (int cln = 0; cln < MAXN; cln++) {
            cout << (board[row] == cln ? " Q" : " *");
        }
        cout << '\n';
    }
    cout << '\n';
}

// 使用递归来实现回溯。
void dfs(int row) {
    // 当行数达到棋盘的最大行数时表明回溯发现了一个可行解。
    if (row == MAXN) { printBoard(); cnt++; return; }
    // 未达到棋盘最大行数，继续进行递归回溯。
    for (int cln = 0; cln < MAXN; cln++) {
        // 为当前行选择列后立即进行对角线规则检查。
        if (clnUsed[cln] || !checkBoard(row, cln)) continue;
        // 标记已选列，回溯进入下一层。
        clnUsed[cln] = 1, board[row] = cln;
        dfs(row + 1);
        clnUsed[cln] = 0, board[row] = -1;
    }
}

int main(int argc, char *argv[]) {
    dfs(0);
    cout << cnt << '\n';
    return 0;
}
//-----8.1.1.cpp-----//

```

以下是上述代码运行后的部分输出：

```

Q * * * * * * *
* * * * Q * * *
* * * * * * * Q
* * * * * * Q *
* * * Q * * * *
* * * * * * * Q
* Q * * * * * *
* * * Q * * * *

```



```

Q * * * * * * *
* * * * * Q * *
* * * * * * * Q
* * Q * * * * *
* * * * * * Q *
* * * Q * * * *
* Q * * * * *
* * * * Q * * *

```



```

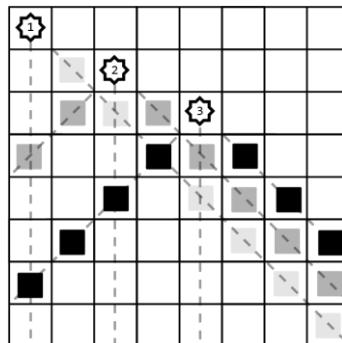
...

```

由输出可知，在 8×8 的棋盘上放置8个皇后，只有92种放置方案符合要求，只占搜索空间的 $1/438$ 左右。随着棋盘的增大，符合要求的放置方案占搜索空间的比例将进一步缩小，而搜索时间却大幅度增加，直至回溯法变得低效而不适用¹。

位运算优化

在前述八皇后问题的实现中，在为第 r 行选定皇后所能放置的列之后，需要从第0行遍历到第 $r-1$ 行进行对角线规则检查，其耗时随着 r 的增大而逐渐增加。由于每次选择列之后都需要进行一次这样的检查，累积起来是一笔不小的时间开销。能否通过某种方法快速确定不存在对角线冲突的可选列呢？答案是肯定的。可以通过位向量标记实现这个目标，从而达到优化程序提高效率的目的^[78]。使用位向量标记的关键是如何使用位来表示已选列和可选列，考虑到为当前行选择某列会对后续行的可选列数量及位置产生影响，可以使用如图8-3所示的方法来构造位向量。特别的，当 n 较小时（例如 $n \leq 20$ ），三个位向量可以使用三个int数据类型变量来代替。



¹ 当 $n \geq 1$ 时，在 $n \times n$ 的棋盘上放置 n 个互不攻击的皇后的不同方法数对应于OEIS编号为A000170的数列（只列举了数列的前20项，方括号前为放置方案数，方括号内为棋盘的大小）：1[1], 0[2], 0[3], 2[4], 10[5], 4[6], 40[7], 92[8], 352[9], 724[10], 2680[11], 14200[12], 73712[13], 365596[14], 2279184[15], 14772512[16], 95815104[17], 666090624[18], 4968057848[19], 39029188884[20], …。参阅：<https://oeis.org/A000170>, 2020。

图8-3 八皇后问题位向量的构造。当在某行选择一个可行列放置皇后之后，会对紧接其后的一行立即产生了两个“禁止列”——所选择列的左侧一列和右侧一列。而且，每向后一行，前面已选列所产生的“禁止列”各向左（右）侧移动一个位置。如果令位向量 M 表示当前已经选择的列，位向量 L 表示已选列在左侧产生的“禁止列”，位向量 R 表示已选列在右侧产生的“禁止列”，那么后续行的可选列位置为三个位向量先进行“或”运算然后进行“取反”运算后仍为0的二进制位所对应的列，即可选列 C 为 $\sim(L \mid M \mid R)$ 中为0的二进制位所代表的列。假设以8个二进制位表示列状态，最左侧的二进制位表示第1列，最右侧的二进制位表示第8列，则可以将回溯过程中前三行位向量的选择描述如下：（1）第一行， $L=M=R=00000000$ ，则 $C=\sim(L \mid M \mid R)=11111111$ ，共有8个位置可选，若选择第1列，则 $M=(M \mid 10000000)=10000000$ ；更新 $L=(L \mid 10000000) \ll 1=00000000$ ， $R=(R \mid 10000000) \gg 1=01000000$ ；（2）第二行： $L=00000000$ ， $M=10000000$ ， $R=01000000$ ，则 $C=\sim(L \mid M \mid R)=00111111$ ，有6列位置可选，若选择第3列，则 $M=(M \mid 00100000)=10100000$ ，更新 $L=(L \mid 00100000) \ll 1=01000000$ ， $R=(R \mid 00100000) \gg 1=00110000$ ；（3）第三行： $L=01000000$ ， $M=10100000$ ， $R=00110000$ ，则 $C=\sim(L \mid M \mid R)=00001111$ ，有4列位置可选，若选择第5列，则 $M=(M \mid 00001000)=10101000$ ，更新 $L=(L \mid 00001000) \ll 1=00100000$ ， $R=(R \mid 00001000) \gg 1=00011100$ 。依此类推，可继续为后续行选择可行列并更新相应的位向量。

由此可以得到以下优化的实现代码：

```
-----8.1.2.cpp-----//
// n 表示棋盘的大小。
int n;

// 递归实现回溯搜索。参数 D 为回溯的层次，L 表示左侧的禁止列，M 表示已选择列，R 表示右侧禁止列。
int dfs(int D, int L, int M, int R) {
    // 回溯层次达到 n 表明这是一种符合要求的放置方案。
    if (D == n) { cnt++, return; }
    // 查找当前行的可选列。
    for (int i = 0; i < n; i++)
        // 通过位运算技巧选择可行列。
        if (((1 << i) & (L | M | R)) == 0)
            // 记录已选列、左侧禁止列、右侧禁止列，然后继续下一层回溯。
            dfs(D + 1, (L | (1 << i)) << 1, M | (1 << i), (R | (1 << i)) >> 1);
}
-----8.1.2.cpp-----//
```

在上述代码片段中，还未彻底做到全部使用位运算。为了进一步挖掘位运算的提速潜力，可以将“查找当前行的可选列”这一环节也使用位运算来实现。与此同时，考虑到回溯到达第 n 层时，参数 M 的低 n 位必定全为1，只需检查参数 M 与低 n 位全为1的“特定标记”是否相等即可判定是否已经达到回溯的目标，这样可以省略表示递归深度的参数。由此可以得到下述进一步优化的实现代码：

```
-----8.1.3.cpp-----//
// n 为棋盘大小，cnt 记录可行放置方案数，N_ONES 为“特定标记”。
int n, cnt, N_ONES;

// N_ONES 为低 n 位全为 1 的“特定标记”。
N_ONES = (1 << n) - 1;

// 使用递归实现回溯搜索。L 表示左侧的禁止列，M 表示已选择的列，R 表示右侧的禁止列。
void dfs(int L, int M, int R) {
    int available, cln;
```

```

// 如果回溯已经达到第 n 层，则已选择列标记 M 的低 n 位必定全为 1，会与标记 N_ONES 相等，  

// 因此可以利用这个特点来判断是否已经成功放置了 n 个皇后。  

if (M != N_ONES) {  

    // available 表示当前可选列。  

    available = N_ONES & (~ (L | M | R));  

    // 当 available 不为 0 时表示还有可选列。  

    while (available) {  

        // 利用位运算技巧得到 available 最右侧为 1 的位，表示当前可行的一种列选择。  

        cln = available & (~available + 1);  

        // 将已选择的列从可选列中剔除。  

        available ^= cln;  

        // 记录已经选择的列，左侧禁止列标记向左移一位，右侧禁止列标记向右移一位，继续回溯。  

        dfs((L | cln) << 1, M | cln, (R | cln) >> 1);  

    }  

}  

else cnt++;  

}  

//-----8.1.3.cpp-----//

```

在最后一种实现中，充分利用了位运算的速度优势，因此在计算 $n \leq 15$ 时的棋盘放置方案时，速度较快。但由于 n 皇后问题本身的特殊性，随着棋盘的增大其搜索空间呈指数级增长，所以对于较大的 n （例如 $n \geq 20$ ），位运算依然显得“力不从心”。

167 The Sultan's Successor^A（苏丹继任者）

努比亚（Nubia）的苏丹^I没有儿子，因此她决定在去世的时候，将国家分成 k 个地区，每个地区由某个在指定测试中表现最佳的人来继承。由于可能出现单个人继承多个或全部地区的情况，她需要确保只有更高智商的候选人才能成为她的继任者，因此苏丹发明了一种特殊的测试方法。在一个喷泉幽鸣、暗香环绕的大厅里，放置了 k 个国际象棋棋盘，每个棋盘的方格均有一个 1 到 99 之间的数，同时给了 8 个宝石镶嵌的皇后棋子。每一个苏丹继任者候选人需要将 8 个皇后放置在棋盘上，使得它们不会互相攻击，同时 8 个皇后放置的方格内的数之和至少和苏丹已经给定的数一样大。（对于不熟悉国际象棋规则的人来说，在放置时要求棋盘的每一行和每一列以及对角线上都只能有一个皇后）。

编写程序读入棋盘以及棋盘上的数字，确定在符合给定条件下所能得到的数字和的最大值（你心里明白苏丹不仅是一个国际象棋好手，同时也是一个优秀的数学家，因此你怀疑她给的数值可能是能够获得的最大数值）。

输入

输入第一行包含 k 值，后面是 k 组 64 个数组成的矩阵，每组有 8 行，每行 8 个数，给定的数均为正数且小于 100。最多不超过 20 组棋盘。

输出

输出由 k 个数值组成，表示 k 组棋盘的数值和，以右对齐宽度 5 进行输出。

样例输入

1

样例输出

260

^I Sultan, 某些穆斯林国家统治者的称号。

1	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	52	53	54	55	56
57	58	59	60	61	62	63	64

分析

应用回溯法构建所有可能的放置方案，找到符合要求的放置方案，然后对各个放置方案求数值和，找到数值和的最大值。对于 n 皇后问题来说，使用回溯法解题，实际上是获得 1 到 n 的整数的所有排列中符合要求的特定排列。因此，对于本题中给定的棋盘规模来说，一种更为简便的方法是直接使用库函数中的 `next_permutation` 来生成 1 到 8 的所有排列而不需要“大动干戈”去具体实现回溯法。

强化练习：[259 Software Allocation^B](#), [524 Prime Ring Problem^A](#), [552 Filling the Gaps^D](#), [677 All Walks of Length n From the First Node^C](#), [750 8 Queens Chess Problem^A](#), [932 Checking the N-Queens Problem^E](#), [989 Su Doku^C](#), [10576 Y2K Accounting Bug^B](#), [10957 So Doku Checker^D](#), [11085 Back to the 8-Queens^A](#), [11195 Another n-Queen Problem^C](#)。

扩展练习：[653 Gizilch^D](#), [1309* Sudoku^{E^{\[79\]}}](#), [10513* Bangladesh Sequences^E](#)。

N 皇后可行解问题

对于 N 皇后问题，如果只需要求出一种可行的放置方案，可根据数学方法快速得到解^[80]。设棋盘大小为 $n \times n$ ，要求在其上放置 n 个皇后且互相不能攻击，可按下列方法得到可行的放置方案（从第 1 行到第 n 行逐行给出放置皇后的列序号，从棋盘最左侧的列开始计数，即最左侧列的序号为 1）。

若 $n \bmod 6 \neq 2$ 且 $n \bmod 6 \neq 3$ ，有：

(1) 若 n 为偶数，则：2, 4, 6, 8, …, n , 1, 3, 5, 7, …, $n-1$;

(2) 若 n 为奇数，有：2, 4, 6, 8, …, $n-1$, 1, 3, 5, 7, …, n ;

若 $n \bmod 6 = 2$ 或者 $n \bmod 6 = 3$ ，令 $k = n/2$ （除法为整除），有：

(1) 若 k 为偶数， n 为偶数： $k, k+2, k+4, \dots, n, 2, 4, 6, \dots, k-2, k+3, k+5, \dots, n-1, 1, 3, 5, \dots, k+1$;

(2) 若 k 为偶数， n 为奇数： $k, k+2, k+4, \dots, n-1, 2, 4, 6, \dots, k-2, k+3, k+5, \dots, n-2, 1, 3, 5, \dots, k+1, n$;

(3) 若 k 为奇数， n 为偶数： $k, k+2, k+4, \dots, n-1, 1, 3, 5, \dots, k-2, k+3, k+5, \dots, n, 2, 4, 6, \dots, k+1$;

(3) 若 k 为奇数， n 为奇数： $k, k+2, k+4, \dots, n-2, 1, 3, 5, \dots, k-2, k+3, k+5, \dots, n-1, 2, 4, 6, \dots, k+1, n$ 。

```
//-----8.1.4.cpp-----//  
// n 皇后问题可行解快速构造。
```

// 数组 `clnAtRow` 保存的是各行放置皇后的列序号（从棋盘最左侧开始计数列，行和列均从 0 开始计数）。

```
void nQueen(int *clnAtRow, int n) {  
    if (n % 6 != 2 && n % 6 != 3) {  
        int row = 0;  
        for (int y = 2; y <= n; y += 2) clnAtRow[row++] = y - 1;
```

```

        for (int y = 1; y <= n; y += 2) cInAtRow[row++] = y - 1;
    } else {
        int k = n / 2;
        int intervals[2][4][2] = {
            {{k, n}, {2, k - 2}, {k + 3, n - 1}, {1, k + 1}},
            {{k, n - 1}, {1, k - 2}, {k + 3, n}, {2, k + 1}}
        };
        int row = 0;
        for (int x = 0; x < 4; x++) {
            int start = intervals[k % 2][x][0], end = intervals[k % 2][x][1];
            for (int y = start; y <= end; y += 2)
                cInAtRow[row++] = y - 1;
        }
        if (n % 2) cInAtRow[row++] = n - 1;
    }
}
//-----8.1.4.cpp-----//

```

幻方构造

幻方 (magic square) 是指将 $1, 2, \dots, n^2$ 填入一个 $n \times n$ 的方阵，使得方阵的每一行、每一列、两条对角线上的数字和均相等，其中 n 称为幻方的阶 (order)。数学家已经证明，对于任意 $n > 2$ ，均存在对应的幻方。幻方中同一行数字相加得到的和 M 称为幻数 (magic constant)，其值的大小只和阶有关，可以通过公式

$$M = \frac{n(n^2 + 1)}{2}$$

予以计算。

朴素的方法是使用回溯来搜索可能的数字组合，然后检查其是否构成幻方，不过这样做效率不高。有趣的是，人们发现可以使用固定的策略来构造幻方。根据幻方的阶，可以将其分为以下三种情形进行构造。

(1) 奇数阶幻方的构造，其中 $n = 2k + 1$, $k \geq 1$ 。可以使用一种称为 Siamese 法 (又称 De la Loubère 法或阶梯法) 的策略来构造奇数阶幻方。具体步骤为：将 1 填入方阵第一行的中间一列，视方阵的第一行和最后一行、第一列和最后一列连续 (即将方阵的上下和左右看成是相连的)，对于后续的每一个数 y ，将其放置在前一个数 x 的右上角 (亦即行数减 1 列数加 1 的位置)。如果按照前述策略所确定的位置上已经有数存在，则将需要放置的数 y 置于 x 的正下方，继续幻方的构造。读者可以通过观察下图所示的三阶幻方构造过程来获得更为直观地理解。

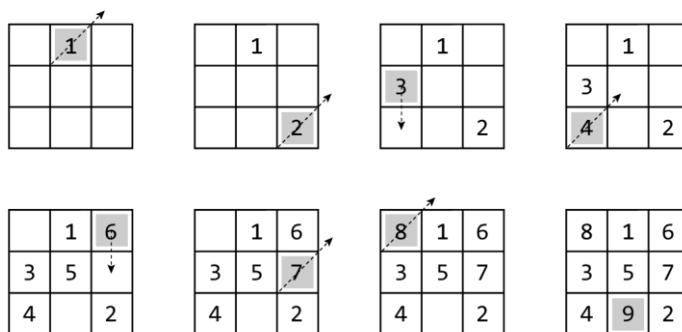


图 8-4 三阶幻方构造

```
//++++++8.1.5.cpp+++++++
const int MAXN = 32;
// n = 2 * k + 1, k >= 1.
void fillMagic(int n, int magic[][][MAXN]) {
    for (int i = 0, j = n / 2, k = 1; k <= n * n; k++) {
        magic[i][j] = k;
        if (k % n) {
            i = (i - 1 + n) % n;
            j = (j + 1) % n;
        } else i = (i + 1) % n;
    }
}
```

(2) 双偶数阶幻方的构造, 其中 $n=4k$, $k\geq 1$ 。首先定义“互补”的概念, 如果两个数字的和等于 n 阶幻方的最大数字和最小数字的和, 则称这一对数关于 n 阶幻方互补, 例如 $8+57=65=64+1$, 则称 8 和 57 关于八阶幻方互补。在构造双偶数阶幻方时, 先将 1 到 n^2 的数按照行优先顺序填入方阵中, 然后将整个方阵从左上角开始, 划分为 k^2 个 4×4 的子方阵, 如果某个数位于子方阵的主、副对角线, 则将其替换为与之互补的数, 其实际效果相当于将这些数关于整个方阵进行中心对称的位置对调, 亦即将位于方格 (i, j) 的数与位于方格 $(n-1-i, n-1-j)$ 的数交换, 其中 (i, j) 必须是位于子方阵主、副对角线上的方格。

1	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	52	53	54	55	56
57	58	59	60	61	62	63	64

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

图 8-5 八阶幻方构造

```
const int MAXN = 32;
// n = 4 * k, k >= 1.
void fillMagic(int n, int magic[][][MAXN]) {
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (((i % 4 == 0 || i % 4 == 3) && (j % 4 == 0 || j % 4 == 3)) ||
                ((i % 4 == 1 || i % 4 == 2) && (j % 4 == 1 || j % 4 == 2)))
                magic[i][j] = n * n - (i * n + j);
            else
                magic[i][j] = i * n + j + 1;
}
```

(3) 单偶数阶幻方的构造, 其中 $n=4k+2$, $k\geq 1$ 。此种情形的幻方构造稍复杂。其步骤是: (a) 将整个方阵划分为 4 个大小为 $(2k+1)\times(2k+1)$ 的子方阵, 按照左上、右下、右上、左下的顺序依次使用 Siamese 法填充这 4 个奇数阶子幻方, 按照顺序, 相邻两个子幻方的对应元素值相差 $n^2/4$; (b) 对于第 0 列至第 $n/4-1$ 列、第 $n-n/4+1$ 列至第 $n-1$ 列, 将这些列位于第 i 行的数与第 $i+n/2$ 行的数对换, 其中 $0\leq i < n/2$;

(c) 在前述对换过程中, 有两个方格($n/4, 0$)和($n/4, n/4$), 前者不应对换却发生了对换, 后者应该对换但未发生对换, 需要予以特殊处理。

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

17	24	1	8	15	67	74	51	58	65
23	5	7	14	16	73	55	57	64	66
4	6	13	20	22	54	56	63	70	72
10	12	19	21	3	60	62	69	71	53
11	18	25	2	9	61	68	75	52	59
92	99	76	83	90	42	49	26	33	40
98	80	7	14	16	73	55	57	64	41
79	81	13	20	22	54	56	63	70	47
85	87	19	21	3	60	62	69	71	28
86	93	25	2	9	61	68	75	52	34

图 8-6 十阶幻方构造——(a) 按 Siamese 法构造奇数阶子幻方

17	24	1	8	15	67	74	51	58	65
23	5	7	14	16	73	55	57	64	66
4	6	13	20	22	54	56	63	70	72
10	12	19	21	3	60	62	69	71	53
11	18	25	2	9	61	68	75	52	59
92	99	76	83	90	42	49	26	33	40
98	80	82	89	91	48	30	32	39	41
79	81	88	95	97	29	31	38	45	47
85	87	94	96	78	35	37	44	46	28
86	93	100	77	84	36	43	50	27	34

92	99	1	8	15	67	74	51	58	40
98	80	7	14	16	73	55	57	64	41
79	81	13	20	22	54	56	63	70	47
85	87	19	21	3	60	62	69	71	28
86	93	25	2	9	61	68	75	52	34
17	24	76	83	90	42	49	26	33	65
23	5	82	89	91	48	30	32	39	66
4	6	88	95	97	29	31	38	45	72
10	12	94	96	78	35	37	44	46	53
11	18	100	77	84	36	43	50	27	59

图 8-7 十阶幻方构造——(b) 对换

92	99	1	8	15	67	74	51	58	40
98	80	7	14	16	73	55	57	64	41
79	81	13	20	22	54	56	63	70	47
85	87	19	21	3	60	62	69	71	28
86	93	25	2	9	61	68	75	52	34
17	24	76	83	90	42	49	26	33	65
23	5	82	89	91	48	30	32	39	66
4	6	88	95	97	29	31	38	45	72
10	12	94	96	78	35	37	44	46	53
11	18	100	77	84	36	43	50	27	59

92	99	1	8	15	67	74	51	58	40
98	80	7	14	16	73	55	57	64	41
4	81	88	20	22	54	56	63	70	47
85	87	19	21	3	60	62	69	71	28
86	93	25	2	9	61	68	75	52	34
17	24	76	83	90	42	49	26	33	65
23	5	82	89	91	48	30	32	39	66
79	6	13	95	97	29	31	38	45	72
10	12	94	96	78	35	37	44	46	53
11	18	100	77	84	36	43	50	27	59

图 8-8 十阶幻方构造——(c) 特殊情况处理及最后构造完毕的幻方

```

const int MAXN = 32;
// 幻方构造辅助填充函数。
void helper(int n, int offseti, int offsetj, int offsetk, int magic[][][MAXN]) {
    for (int i = 0, j = n / 2, k = 1; k <= n * n; k++) {
        magic[i + offseti][j + offsetj] = k + offsetk;
        if (k % n) {
            i = (i - 1 + n) % n;
            j = (j + 1) % n;
        }
        else
            i = (i + 1) % n;
    }
}
// n = 4 * k + 2, k >= 1。
void fillMagic(int n, int magic[][][MAXN]) {
    // 使用 Siamese 方法填充四个子方阵。
    helper(n / 2, 0, 0, 0, magic);
    helper(n / 2, n / 2, n / 2, n * n / 4, magic);
    helper(n / 2, 0, n / 2, n * n / 2, magic);
    helper(n / 2, n / 2, 0, n * n / 4 * 3, magic);
    // 对换。
    for (int i = 0; i < n / 2; i++)
        for (int j = 0; j < n / 4; j++)
            swap(magic[i][j], magic[i + n / 2][j]);
    for (int i = 0; i < n / 2; i++)
        for (int j = n - n / 4 + 1; j < n; j++)
            swap(magic[i][j], magic[i + n / 2][j]);
    // 特殊情况处理。
    for (int j = 0; j <= n / 4; j += n / 4)
        swap(magic[n / 4][j], magic[n / 4 + n / 2][j]);
}
//++++++++++++++8.1.5.cpp+++++++

```

强化练习: 1266 Magic Square^D, 10087 The Tajmahal of ++Y2k^D, 10094 Place the Guards^C。

扩展练习：10741 Magic Cube^D。

8.2 搜索

如前所述, 回溯法实质上是利用深度优先遍历的思想对隐式图所表示的问题空间进行一次穷尽搜索, 这种穷尽搜索类似于“大海捞针”(look for a needle in a haystack), 但它是一种有序的过程, 最终是将所有可能作为解的候选答案进行了一遍检查, 从而将符合要求的解筛选出来。一般情况下, 可以直接应用回溯法解决的题目所涉及的问题空间都相对较小, 按照既定的回溯法步骤均可以在规定时限内完成。

8.2.1 单向搜索

对于某些回溯类题目来说, 使用回溯法进行搜索的过程是一个单向的过程, 即从初始状态往目标状态搜索, 则称之为单向搜索。单向搜索根据其使用方式又可以将其划分为以下的若干子类型。

完全搜索

在某些情形下, 回溯法需要从初始状态出发, 通过逐步构建解的方式直到达到解所具有的数量限制(例如, 长度, 高度, 宽度, 字符数), 而在构建解的过程中, 可能还需要满足一些约束(例如, 相邻字符不能相等, 数组的和为负数), 在最后还可能对候选解再加以进一步地检查, 以查看其是否符合一些其他限制条件。由于此类回溯法搜索了所有可能的解, 故将其归类为完全搜索。

10503 The Dominoes Solitaire^B (单人多米诺游戏)

给定宽度为 2, 高度为 1 的若干枚多米诺骨牌, 每枚多米诺骨牌牌面上有两个数字。将两枚多米诺骨牌固定放置在一行的两端, 在中间留下 n 块宽度为 2 的空白, 再给定 m 枚多米诺骨牌, 请问是否可从 m 枚骨牌中选出某些骨牌填满这 n 块空白, 并且使得相邻骨牌连接处的数字相同。例如, 给定初始骨牌(0, 1)和(3, 4), 中间留有 3 块宽度为 2 的空白, 4 枚候选骨牌为(2, 1), (5, 2), (2, 2), (3, 2), 则可以按如下方式进行填充:

(0, 1)(1, 2)(2, 2)(2, 3)(3, 4)

输入

输入包含多组测试数据。每组测试数据的第一行为整数 n , 表示空白的块数, 接着一行是整数 m , 表示候选骨牌的数量, $n \leq m \leq 14$ 。接着两行表示两端的固定骨牌, 每行包含两个整数, 数字出现的顺序为固定骨牌上的数字, 顺序为从左至右。接着是 m 行, 也是每行包含两个整数, 表示候选骨牌上的数字。当 n 为 0 时, 输入结束。

输出

对于输入中的每组数据, 如果存在满足要求的放置方案, 输出 YES, 否则输出 NO。

样例输入

3 4 0 1 3 4 2 1 5 6 2 2 3 2 0

样例输出

YES

分析

题目要求从 m 枚骨牌中选出 n 枚排成一行，使得起始和结尾数字与指定的数字相匹配，且相邻骨牌连接处的数字相同，很明显，必须搜索全部问题空间来筛选符合要求的解。但是直接应用生成全排列的方法来构造从 m 枚骨牌中选出 n 枚骨牌的所有排列，其数量较大，而且只能在生成排列后才能进行检查，不利于应用“相邻骨牌连接处的数字相同”这个限制条件。故按以下步骤进行回溯：

(1) 确定回溯的初始状态和终止状态。由于是要求“相邻骨牌连接处的数字相同”，那么就从此入手，将连接处的数字视为状态，初始状态即为给定的位于最左侧的固定骨牌的右侧数字，终止状态即为给定的位于右侧固定骨牌的左侧数字。例如，在样例输入中，初始状态为 1，终止状态为 3。

(2) 根据当前状态和候选骨牌进行回溯。由于初始状态为 1，那么需要确定候选骨牌中是否有一侧数字为 1 的骨牌可用，如果有这样的骨牌未使用，则将其标志为已使用状态，因为候选骨牌只能被使用一次，后续不能再使用。找到符合要求的骨牌后，回溯进入下一层，对应的状态将改变为已使用骨牌的对侧数字，已填充的空白块数加 1。需要注意的是，在本题中候选骨牌是可以翻转使用的，如骨牌(2, 3)可作为(3, 2)使用，但数字相同的骨牌翻转使用效果相同，为了不增加回溯的深度，可以预先判断并予以剔除。

(3) 确定是否达到终止状态。当回溯的深度达到预期时，需要对解进行检查，查看其是否符合题意要求。在本题中，当填满的空格块数达到 n 时，需要检查当前状态，即骨牌末尾处的数字是否和终止状态的数字相同。如果相同，表明找到了符合要求的骨牌排列方案，否则此搜索分支应该停止。为了减少不必要的搜索，在找到符合要求的方案后，可以设置完成标识以便尽早退出回溯过程。

参考代码

```

int n, m, dominoes[15][2], used[20];
int dot1, dot2, head, tail;
int finished = 0;

void dfs(int depth, int ending) {
    // 已经得到解，尽早退出。
    if (finished) return;
    // 回溯达到预定深度，检查解是否符合要求。
    if (depth == n) {
        if (ending == tail) finished = 1;
        return;
    }
    // 回溯未达到预定深度，根据限制条件改变回溯状态并进入下一层回溯。
    for (int i = 0; i < m; i++) {
        if (used[i]) continue;
        if (dominoes[i][0] == ending) {
            used[i] = 1;
            dfs(depth + 1, dominoes[i][1]);
            used[i] = 0;
        }
        if (dominoes[i][0] != dominoes[i][1] && dominoes[i][1] == ending) {
            used[i] = 1;
            dfs(depth + 1, dominoes[i][0]);
            used[i] = 0;
        }
    }
}

int main(int argc, char *argv[]) {

```

```

while (cin >> n, n > 0) {
    // 读入初始状态、终止状态、候选骨牌。
    cin >> m;
    cin >> dot1 >> dot2; head = dot2;
    cin >> dot1 >> dot2; tail = dot1;
    for (int i = 0; i < m; i++) cin >> dominoes[i][0] >> dominoes[i][1];
    // 回溯并输出解。
    memset(used, 0, sizeof(used));
    finished = 0;
    dfs(0, head);
    cout << (finished ? "YES" : "NO") << '\n';
}
return 0;
}

```

强化练习: 148 Anagram Checker^B, 193 Graph Coloring^A, 211 The Domino Effect^D, 347 Run Run Runaround Numbers^B, 399 Another Puzzling Problem^D, 416 LED Test^B, 441 Lotto^A, 604 The Boggle Game^C, 843 Crypt Kicker^B, 868 Numerical Maze^D, 996 Find the Sequence^E, 1052* Bit Compressor^E, 1262 Password^C, 10001 Garden of Eden^C, 10063 Knuth's Permutation^B, 10186 Euro Cup 2000^D, 10202 Pairsumonious Numbers^B, 10344 23 Out of 5^A, 10637 Coprimes^C, 10950 Bad Code^D, 11201 The Problem of the Crazy Linguist^D, 11236 Grocery Store^B, 11451 Water Restrictions^D, 11961 DNA^D, 13004 At Most Twice^D。

扩展练习: 165 Stamps^B, 179 Code Breaking^D, 205 Getting There^E, 225 Golygons^D, 265 Dining Diplomats^E, 283 Compress^E, 387 A Puzzling Problem^D, 592 Island of Logic^D, 10624* Super Number^C, 10923* Seven Seas^D, 11471* Arrange the Tiles^D, 11753 Creating Palindrome^D。

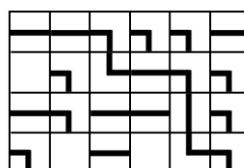
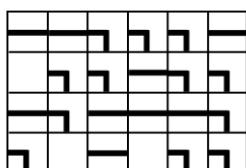
在完全搜索类型的回溯中, 一个难点是如何记录回溯的状态和根据当前状态决定下一步回溯的选择。由于很多回溯类题目的背景都是在矩阵中进行, 而矩阵是由单元格组成, 因此可以设置一个二维数组来标记当前单元格是否已经被使用。有时仅仅标记单元格是否被使用尚不足够, 因为回溯状态可能具有方向性, 在某个单元格的向左方向已经被使用, 还可以选择向右的方向, 因此可能要同时记录单元格的位置和被使用的方向, 需要三维甚至四维数组来记录回溯的当前状态。

10582 ASCII Labyrinth^D (ASCII 迷宫)

给定一个 $m \times n$ 的迷宫, 迷宫由印有图案的方形木块组成, 图案包括三种:



你可以对木块进行旋转, 但不能将木块移动到其他方格。给定迷宫的初始状态, 确定有多少种不同的木块排列方式, 使得木块上的图案能够形成连续的线段, 连接左上角和右下角的单元格。下图给出了初始的木块排列和满足要求的一种木块排列方式。



输入

输入第一行为一个整数 c ，表示测试数据的组数。每组测试数据第一行包含两个整数 m 和 n ，表示迷宫的行数和列数。接着是以 ASCII 表示的迷宫，由 ‘+’、‘-’、‘*’、‘|’、空格构成。迷宫大小满足 $m \times n \leq 64$ 的限制。

输出

对于输入中的每组测试数据，输出一个整数，表示满足要求的不同路径总数。

样例输入

```
1
4 6
+---+---+---+---+---+---+
|---|---|---|---|---|---|
|***|***|**|**|**|***|
|---|---|*|*|*|---|
+---+---+---+---+---+---+
|---|---|---|---|---|---|
|---|**|**|***|**|**|
|---|*|*|---|*|*|
+---+---+---+---+---+---+
|---|---|---|---|---|---|
|***|**|***|***|***|**|
|---|*|---|---|---|*|
+---+---+---+---+---+---+
|---|---|---|---|---|---|
|**|---|***|---|**|**|
|*|---|---|---|*|*|
+---+---+---+---+---+---+
```

样例输出

```
Number of solutions: 2
```

分析

从题目描述中可以容易地得出判断，进入空白图案的木块后无法再向其他木块前进，因此是“死胡同”。很明显，也不能走到迷宫边界之外。进入包含横向图案的木块后只能沿着原有的方向继续前进，不能改变方向。进入带折线图案的木块，只能左转和右转。由于题目要求的是找出所有可能的不同路径，因此在迷宫的行进过程中需要记录已经走过的木块，在具体实现时，可以使用一个二维数组予以记录。对于进入木块后行进方向的改变，可以根据当前行进方向和木块所具有的图案进行判定。

在下述实现中，考虑了输入可能给出“旋转版本”图案的情况，并考虑了输入中可能包含空格的情况（尽管木块上的图案是可以旋转的，但 UVa OJ 上的评判数据似乎较弱，并未包含“旋转版本”的测试数据，所有评判数据均以初始给定的样式出现，题目中未完全明确的说明这一点而只是给出了暗示。uDebug 上的测试数据给出了只包含一个单元格的迷宫，此种情况下，不论单元格内是否包含图案，不同路径数均为 2。不过评判数据中似乎并未包含这样的测试数据）。因为从初始的左上角木块只能向右和向下走，因此不妨就假设左上角的木块图案包含的就是直线，不影响结果的正确性，可以便于编码的实现。

参考代码

```
string line;
int m, n, maze[70][70], used[70][70], paths, cases;
int offset[4][2] = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

// d 为方向: 0 表示向右, 1 表示向下, 2 表示向左, 3 表示向上。
void dfs(int i, int j, int d) {
```

```

if (i == m && j == n) { paths++; return; }
// 根据当前木块的图案和行进方向确定下一个到达的木块和后续行进方向。
used[i][j] = 1;
for (int next = 0; next <= 3; next++) {
    if ((maze[i][j] == 1 || maze[i][j] == 3) && next > 0) continue;
    if (maze[i][j] == 2 && (next == 0 || next == 2)) continue;
    int nextd = (d + next) % 4;
    int ii = i + offset[nextd][0], jj = j + offset[nextd][1];
    if (ii >= 1 && ii <= m && jj >= 1 && jj <= n)
        if (!used[ii][jj] && maze[ii][jj])
            dfs(ii, jj, nextd);
}
used[i][j] = 0;
}

int main(int argc, char *argv[]) {
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        // 读入迷宫。
        cin >> m >> n; cin.ignore(1024, '\n');
        memset(maze, 0, sizeof(maze));
        for (int i = 1; i <= m; i++) {
            getline(cin, line); getline(cin, line); getline(cin, line);
            for (int j = 1, k = 0; j <= n; j++) {
                while (line[k] != '|') k++;
                for (k++; line[k] != '|'; k++)
                    if (line[k] == '*')
                        maze[i][j]++;
            }
            getline(cin, line);
        }
        getline(cin, line);
        // 回溯并输出结果。
        paths = 0, maze[1][1] = 3;
        dfs(1, 1, 0); dfs(1, 1, 1);
        cout << "Number of solutions: " << paths << '\n';
    }
    return 0;
}

```

强化练习: 258 Mirror Maze^D, 296 Safebreaker^C, 710 The Game^D, [11283 Playing Boggle^C](#)。

扩展练习: 10501* Simplified Shisen-Sho^E。

在完全搜索中, 有一类题目并不需要显式地使用回溯来寻找解, 而是可以通过多重循环的方式来构造出所有可能的解, 在此过程中可能还需要进行额外的操作, 例如考虑解中元素的顺序, 或者需要使用剪枝技巧以缩短运行时间等。

强化练习: [608 Counterfeit Dollar^A](#), [735 Dart-a-Mania^B](#), [10365 Blocks^B](#), [10570 Meeting with Aliens^C](#), [10660 Citizen Attention Offices^B](#), [10662 The Wedding^C](#), [10997* Medals^D](#), [11052 Economic Phone Calls^D](#), [11490 Just Another Problem^D](#), [12169 Disgruntled Judge^C](#), [12205 Happy Telephones^C](#), [12337 Bob's Beautiful Balls^D](#), [12665 Joking with Fermat's Last Theorem^D](#), [12801* Grandpa Pepe's Pizza^D](#), [12844 Outwitting the Weighing Machine^D](#), [12938 Just Another Easy Problem^D](#), [13018 Dice Cup^D](#), [13059 Tennis Championship^B](#)。

扩展练习: 1051 Bipartite Numbers^E, 10483 The Sum Equals the Product^D, 13103 Tobby and Seven^E。

构造全排列

在回溯相关的题目中，如果问题空间较小且需要搜索全部问题空间获得最优解，可以应用库函数中的 `next_permutation` 来进行全排列的生成，将全排列映射到问题空间，从而可以通过检查某个排列是否为可行解。需要注意的是，使用该函数前需要将元素进行排序，使得元素为递增序，后续应用此函数才能生成所有的全排列，否则会漏掉一部分排列。

强化练习：102 Ecological Bin Packing^A, 418 Molecules^C, 628 Passwords^A, 725 Division^A, 921 A Word Puzzle in the Sunny Mountains^E, 11412 Dig the Holes^D。

扩展练习：618 Doing Windows^D, 1079^{*} A Careful Approach^D。

构造所有子集

在解题时，有时需要枚举给定大小为 n 的集合的所有子集，使用下述方法可以高效地列出大小从 1 到 n 的所有子集^[81]。

```
-----8.2.1.cpp-----//
// 输出子集所包含的元素。
void print(int flag[], int idx) {
    for (int i = 0; i < idx; i++)
        cout << setw(2) << right << flag[i];
    cout << '\n';
}

// 利用递归生成子集。
void generate(int flag[], int idx, int last, int n) {
    if (idx < last) {
        if (idx == 0) {
            for (int i = 0; i < n; i++) {
                flag[idx] = i;
                generate(flag, idx + 1, last, n);
            }
        } else {
            for (int i = flag[idx - 1] + 1; i < n; i++) {
                flag[idx] = i;
                generate(flag, idx + 1, last, n);
            }
        }
    } else print(flag, idx);
}

// 逐个枚举大小从 1 到 n 的子集。
void enumeratingSubset(int n) {
    for (int i = 1; i <= n; i++) {
        int *flag = new int[n];
        generate(flag, 0, i, n);
        delete [] flag;
    }
}
-----8.2.1.cpp-----//
```

强化练习：471 Magic Numbers^B, 574 Sum It Up^A, 598 Bundling Newspapers^C, 947 Master Mind Helper^D。

构造特定子集

有些题目要求的并不是所有排列而是满足一定条件的全排列的子集，在这种情况下，可以通过回溯来生成所有子集，然后再对生成的子集应用题目所给定的约束条件进行检查。很多时候，题目给定的限制是选择或不选择某个事物、经过或不经过某个位置等，可以应用二进制“位”来表示选择，用“位运算”来对限制进行模拟。

强化练习：[124 Following Orders^A](#), [648 Stamps^D](#), [872 Ordering^B](#), [10475 Help the Leaders^D](#), [10776 Determine The Combination^B](#), [11659 Informants^D](#), [12694 Meeting Room Arrangement^C](#)。

扩展练习：[549 Evaluating an Equations Board^E](#), [10858 Unique Factorization^C](#)。

8.2.2 双向搜索

单向搜索是从起始状态开始向着终止状态（或者相反，从终止状态开始向着初始状态）不断进行搜索，搜索“路径”是一条“单行道”。但对于某些搜索空间较大的题目，仅使用单向搜索可能容易超时，此时同时从起始状态和终止状态开始相向进行搜索可以减小搜索空间，更快地得到解，这样的搜索方法称之为双向搜索，又称为“中间相遇搜索”（meet in middle search）。

11212 Editing a Book^D（编辑书稿）

给定 n 个等长的段落，编号为 1 至 n 。现在要求将其按照 $1, 2, \dots, n$ 的顺序排列。利用剪贴板，你可以使用快捷键复制（Ctrl-X）和粘贴（Ctrl-V）轻松地实现。在此过程中，要求在粘贴前不能剪切两次，但是可以一次性剪切多个相邻段落，然后粘贴到其他地方。当然，段落在粘贴时的和被剪切时的顺序一致。

例如，为了将段落{2, 4, 1, 5, 3, 6}调整为有序，你可以剪切段落 1 并将其粘贴在段落 2 前，然后剪切段落 3 并将其粘贴在段落 4 前。又例如，对于段落{3, 4, 5, 1, 2}，一次复制和粘贴就能够将此段落序列恢复为有序，总共有两种方式可以实现：一种是剪切{3, 4, 5}并将其粘贴在{1, 2}之后，或者剪切{1, 2}并将其粘贴在段落{3, 4, 5}之前。

输入

输入包含至多 20 组测试数据，每组测试数据包括两行输入数据，第一行包含一个整数 n ，表示段落的数量，第二行包含序列 $1, 2, 3, \dots, n$ 的一个排列，表示段落的当前状态。最后一组测试数据后面紧跟一个 0，此行数据不需处理。

输出

对于每组测试数据，输出测试数据组数以及使得段落恢复有序状态所需的最少剪切/粘贴操作次数。

样例输入

```
6
2 4 1 5 3 6
5
3 4 5 1 2
0
```

样例输出

```
Case 1: 2
Case 2: 1
```

分析

不难得出，对于任意一个长为 n 的段落序列，至多只需 $n-1$ 次剪切/粘贴操作即可将其恢复为有序状态：按照 1 到 $n-1$ 的顺序逐次将第 i 个段落剪切并粘贴到第 $i+1$ 个段落前即可。也就是说，对于 $n=9$ 的测试数据，最多只需 8 次操作即可完成操作。但是从初始状态出发，每一步可能需要尝试的剪切/粘贴组合达 10^2

数量级, 如果使用单向搜索, 则需要 10^{16} 级别的计算量, 在时间限制内显然不可接受。如果从初始状态 $1, 2, 3, \dots, n$ 和给定状态分别开始搜索, 则只需各搜索 4 步即可, 则计算量为之前的一半, 为 10^8 数量级, 通过优化剪切/粘贴的模拟操作, 可以在时间限制内获得解决方案。由于是求最少操作次数, 使用 BFS 解题可以更快地得到解。本题的难点是如何快速地获取某个段落序列在一次操作之内所能获得的其他段落序列, 如果此环节处理效率较低, 容易导致超时。

扩展练习: 707 Robbery^D。

8.3 剪枝

由于回溯是通过遍历所有的可能方案然后从中寻找可行解, 当搜索空间很大时, 对一些明显不可能得到可行解的搜索分支继续进行搜索, 会浪费时间, 降低搜索效率。如果能够越早发现这些不可能构成可行解的搜索分支并将之“剪除”, 则可以节省搜索时间, 从而提高效率。

剪枝 (prune) 是对搜索分支进行剪除的形象类比, 它类似于园丁对园木的“剪枝”——剪除已经死掉或不符合要求的枝叶——使得树的营养尽可能提供给需要的地方。在搜索中, 剪枝将明显不可能得到解的搜索分支从搜索中去除, 使得有限的 CPU 时间用在可能得到解的搜索分支上。但是剪枝并不是越多越好, 因为剪枝本身也是需要代价的 (某些剪枝需要消耗较多的 CPU 时间), 如果花费较多的 CPU 时间进行某个剪枝, 而剪枝得到的效益却不能超过所消耗的 CPU 时间, 这样的剪枝将失去意义。

剪枝的具体做法需要根据具体问题而定, 通常是根据题目所设的限制条件来进行, 需要一些逻辑推理能力和想象力。一般来说, 可以将剪枝分为两类, 一类是可行性剪枝, 另外一类是最优化剪枝。可行性剪枝是指在搜索过程中遇到不可能到达目标状态的分支予以剪除, 例如在最长简单路径搜索过程中, 如果顶点 A 和顶点 B 之间无通路, 那么再去搜索 A 到 B 的路径显然是浪费时间。最优化剪枝是指当前选择的代价与最优选择代价相比已经不具有优势, 那么很显然, 可以不必对此搜索分支继续进行搜索。

古人有云: “熟读唐诗三百首, 不会吟诗也会吟”^I, 提高剪枝水平还是需要通过多做练习, 研习他人好的剪枝思路来不断的增强自身思维能力。以下是若干搜索类型题目的题解, 通过题解中的分析, 期望读者能够对剪枝的一般思路和方法建立一种较为直观的印象。

307 Sticks^C (木棒)

Geroge 将同样长度的木棒随机切断, 直到所有木棒的长度均不大于 50 单位长度。现在他想把木棒还原成原来的状态, 但是他忘记了原来有多少根木棒, 也忘记了原来每根木棒的长度是多少。请帮助 Geroge 设计一个程序, 确定木棒最小的可能长度。所有给出的木棒长度均为大于 0 的整数。

输入

输入包含多组测试数据。每组测试数据由两行组成, 第一行为一个整数, 表示在切割后木棒的数量; 第二行包含各木棒的长度数据, 以空格分隔。输入的最后一行包含一个整数 0。

输出

对于输入中的每组数据, 输出原来木棒最小的可能长度。

样例输入

样例输出

^I 出自清代乾隆年间蘅塘退士孙洙所编选《唐诗三百首》的序言, 此为原句, 现多作“熟读唐诗三百首, 不会作诗也会吟”或“熟读唐诗三百首, 不会写诗也会吟”。

9
5 2 1 5 2 1 5 2 1
0

6

分析

基本思路是先预设一个长度，然后通过尝试选取木棒进行拼接验证的方式来确定该长度是否可行。在拼接时需要考虑所有可能的拼接组合可能。由于搜索空间很大，单纯使用回溯无法在规定时间限制内得到答案，必须进行剪枝。剪枝可以通过以下几个方面来进行：

(1) 由于木棒的原始数量和长度均为整数，给定一组数据后，该组数据中木棒的初始长度不能是任意的。设初始时木棒长度为 L ，切割后所有木棒的长度和为 S ，则 L 的可能值有一个范围，最小为切割后现有木棒长度的最大值，最大为 S ，且 L 必须能够整除 S 。

(2) 将木棒按切割后的长度从大到小的顺序进行取用，有利于减少回溯时的递归层数。

(3) 若第 i 根木棒和第 $(i-1)$ 根木棒长度相同且第 $(i-1)$ 根木棒未能被成功使用，则第 i 根木棒肯定也不会被使用。如果先前的拼接是正确的，则第 $(i-1)$ 根木棒应该会被使用，没有使用是因为不能构成满足要求的木棒而剩余，那么同样长度的第 i 根木棒在后续过程中肯定也不会被使用，表明不必在此搜索分支上继续搜索。

(4) 在每次选取木棒时，当前已经拼接的长度加上选取的第 i 根木棒后，应该不大于预设的长度 L ，否则无法拼接出满足要求的木棒；第 i 根木棒开始的后续所有木棒长度之和加上已经拼接的长度必须不小于预设长度 L ，否则即使用完后续的所有木棒也得不到长度为 L 的木棒。

以下的代码中，在进行拼接验证时，拼接的子目标是逐次完成一段目标长度的木棒，当完成一段后，继续拼接下一段，直到拼接完成的木棒数达到预设数量或中途无法完成拼接而退回到上一层回溯。

参考代码

```

// stick[i] 存储切割后第 i 根木棒的长度;
// remain[i] 存储从第 i 根木棒开始后续所有木棒的长度和;
// used[i] 表示第 i 根木棒是否已经被使用;
// goal 表示每次预设的原始木棒长度;
// total 表示原始木棒的数量;
// n 为切割后木棒的数量。
int stick[105], remain[105], used[105], goal, total, n;

// 使用回溯法对预设的木棒原始长度进行拼接验证。
// idx 表示可用木棒的搜索起始序号;
// done 表示当前段木棒拼接是否完成;
// sum 表示当前段木棒已经拼接的长度;
// cnt 表示已经得到的原始长度的木棒数量。
bool dfs(int idx, int done, int sum, int cnt) {
    // 检查当前段木棒是否拼接完毕。
    if (sum == goal) {
        // 检查是否已经完成所有拼接。
        if (cnt + 1 == total) return true;
        if (dfs(0, 1, 0, cnt + 1)) return true;
        return false;
    }
    // 当前段木棒拼接完成，选取尚未被使用的木棒继续下一段目标长度木棒的拼接。
    if (done == 1) {

```

```

for (int i = 0; i < n; i++) {
    if (used[i]) continue;
    used[i] = 1;
    if (dfs(i + 1, 0, stick[i], cnt)) return true;
    used[i] = 0;
    break;
}
} else {
    // 当前段木棒拼接尚未完成, 继续选取可用的木棒完成。idx 限制了可用木棒的起始序号。
    for (int i = idx; i < n; i++) {
        // 剪枝: 已拼接的长度加上当前木棒长度不能大于目标值; 加上后续所有木棒长度不能
        // 小于目标值 (否则使用剩余的所有木棒也无法完成拼接)。
        if (!used[i] && sum + stick[i] <= goal && sum + remain[i] >= goal) {
            // 剪枝: 当前木棒和上一木棒长度相同且上一木棒未使用, 那么使用当前木棒肯定
            // 也不可能完成拼接。
            if (i && stick[i] == stick[i - 1] && !used[i - 1]) continue;
            // 尝试使用当前木棒。
            used[i] = 1;
            if (dfs(i + 1, 0, sum + stick[i], cnt)) return true;
            used[i] = 0;
            // 剪枝: 某次拼接已经达到了预设长度, 但是未能成功返回, 说明预设长度不正确,
            // 不必再继续尝试进行拼接。
            if (sum + stick[i] == goal) return false;
        }
    }
    return false;
}

int main(int argc, char *argv[]) {
    while (cin >> n, n) {
        // 读入数据, 统计所有木棒的长度和。
        int length = 0;
        for (int i = 0; i < n; i++) {
            cin >> stick[i];
            length += stick[i];
        }
        // 将切割后的木棒长度按照从大到小的顺序进行排列。
        sort(stick, stick + n, greater<int>());
        // 计算从第 i 根木棒开始的后续木棒长度和, 为可行性剪枝做准备。
        remain[n] = 0;
        for (int i = n - 1; i >= 0; i--) remain[i] = remain[i + 1] + stick[i];
        // 从小到大遍历可能的木棒初始长度, 回溯进行拼接验证。
        for (goal = stick[0]; goal <= length; goal++) {
            // 确定木棒的初始长度和木棒的原始数量。
            if (length % goal) continue;
            total = length / goal;
            // 进行拼接验证, 成功则输出。
            memset(used, 0, sizeof(used));
            if (dfs(0, 1, 0, 0)) {
                cout << goal << '\n';
                break;
            }
        }
    }
    return 0;
}

```

1

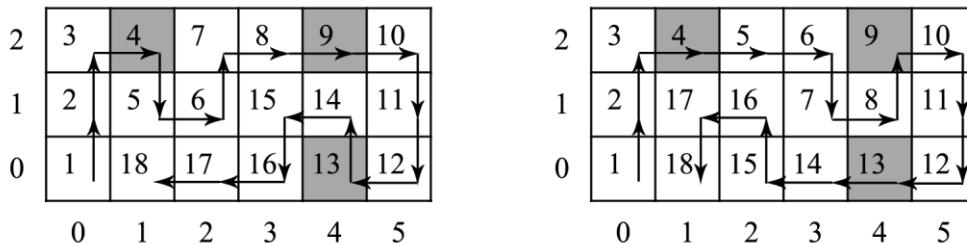
强化练习: 228 Resource Allocation^E, 269 Counting Patterns^E, 301 Transportation^B, 437 The Tower of Babylon^A, 519 Puzzle (II)^D, 624 CD^A, 12840* The Archery Puzzle^D。

扩展练习：219 Department of Redundancy Department^E。

1098 Robot on Ice^D (冰上机器人)

受哈尔滨冰雕的启发，来自北极机器人与自动机大学（Arctic University of Robotics and Automata）的参赛队员决定程序竞赛结束后在校园内举办自己的冰雪节。他们打算在冬天结冰的时候，从学校附近的一个湖里获取冰块。为了便于监测湖中冰层的厚度，他们先将湖面网格化，然后安置一个轻巧的机器人逐个方格测量冰层的厚度。在网格中有三个特殊方格被指定为“检查点”，对应着机器人在检查过程中经过整个行程的四分之一、二分之一、四分之三的位置，机器人在这三个特殊“检查点”会发送相应的进度报告。为了避免对冰面造成不必要的磨损和划痕而影响后续的使用，机器人需要从网格左下角坐标为 $(0, 0)$ 的方格出发，经过所有方格仅且一次，然后返回位于坐标为 $(0, 1)$ 的方格。如果有多种路线符合要求，则机器人每天会使用一条不同的路线。机器人只能沿北、南、东、西四个方向每次移动一个方格。

给定网格的大小和三个检查点的位置，编写程序确定有多少种不同的检查路线。例如，湖面被划分为 3×6 的网格，三个检查点按访问的顺序分别为 $(2, 1)$, $(2, 4)$ 和 $(0, 4)$ ，机器人必须从 $(0, 0)$ 方格开始，路经 18 个方格，最后终止于 $(0, 1)$ 方格。机器人必须在第 $4 (= \lfloor 18/4 \rfloor)$ 步的时候经过 $(2, 1)$ 方格，在第 $9 (= \lfloor 18/2 \rfloor)$ 步的时候经过 $(2, 4)$ 方格，第 $13 (= \lfloor 3 \times 18/4 \rfloor)$ 步的时候经过 $(0, 4)$ 方格，只有两种路线符合要求，如下图所示。



需要注意: (1) 当网格的大小不是 4 的倍数时, 在计算步数时使用整除; (2) 某些情况下可能不存在符合要求的路线, 例如给定一个 4×3 的网格, 三个检查点分别为 $(2, 0)$, $(3, 2)$ 和 $(0, 2)$, 那么将不存在从 $(0, 0)$ 方格出发, 结束于 $(0, 1)$ 方格且满足要求的路线。

输入

输入包含多组测试数据。每组测试数据的第一行包含两个整数 m 和 n , $2 \leq m, n \leq 8$, 表示网格的行数和列数, 接着的一行包含六个整数 $r_1, c_1, r_2, c_2, r_3, c_3$, 其中 $0 \leq r_i < m$, $0 \leq c_i < n$, $i=1, 2, 3$ 。输入的最后一行包含两个 ‘0’。

输出

从 1 开始输出测试数据的组数，输出以下不同路线的数量：机器人从行 0 列 0 出发，在行 0 列 1 结束，在第 $[i \times m \times n/4]$ 步时经过行 r_i 和列 c_i , $i=1, 2, 3$, 能够路经所有方格仅且一次的路线。输出格式参照样例输出。

样例输出

```
3 6
2 1 2 4 0 4
4 3
2 0 3 2 0 2
0 0
```

样例输出

```
Case 1: 2
Case 2: 0
```

分析

本题要求使用完全搜索来得到所有可能的路线，从图论角度看，等价于求隐式图中所有可能的哈密顿回路^I，如果不加以剪枝，难以在限定时间内获得通过。根据题目约束，令机器人当前所在行为 r ，列为 c ，已经行进的步数为 $moves$ ， $used[i][j]=0$ 表示该方格尚未访问， $used[i][j]=1$ 表示该方格已经访问，可以进行以下的剪枝来选择下一步移动的候选方格：

- (1) 进入该方格后，如果 $r < 0$ 或者 $r \geq m$ 或者 $c < 0$ 或者 $c \geq n$ ，表明机器人已经位于网格之外，则该方格不能作为候选方格；
- (2) 机器人进入的方格状态 $used[i][j]=1$ ，即方格已经访问，则该方格不能作为候选方格；
- (3) 如果进入的方格坐标为 $(0, 1)$ ，但行进步数不等于 $m \times n$ ，则表明提前到达了达终止方格 $(0, 1)$ ，那么该方格不能作为候选方格；
- (4) 当前行走步数 $moves$ 达到某个检查点所对应的步数时，但所处方格不是对应的检查点，那么此方格不能作为候选方格；
- (5) 如果当前方格是特定的检查点，但对应的步数 $moves$ 与要求的步数不相符合，那么此方格不能作为候选方格；
- (6) 如果从当前方格到达下一个检查点所需要的最少步数（最少步数可通过两个方格坐标分量的差值的绝对值之和得到）与当前已经行走的步数 $moves$ 之和大于下一个检查点所要求的步数，那么此方格不能作为候选方格；
- (7) 如果访问某个方格后，导致剩余尚未访问的方格不连通，则这些方格不能作为候选方格。可以通过从终止方格 $(0, 1)$ 进行一次 DFS 来确定尚未访问方格的连通性。

使用上述剪枝技巧已经能够在限定时间内获得 Accepted。为了提高搜索效率，还可以对候选方格再进行进一步的优化和剪枝^{II}。首先进行统计，对于当前所在方格的候选方格，统计候选方格的“可选邻近方格”——到达此候选方格后可以选择前进的方格——的数目。

根据“可选邻近方格”数量为 1 的候选方格的数目来确定下一步的行走方向：

- (1) 如果“可选邻近方格”的数量为 1 的候选方格有多个，表明一旦进入这些候选方格，将无法继续访问其他方格，因此不能进入这些候选方格中的任意一个；
- (2) 如果“可选邻近方格”的数量等于 1 的候选方格只有一个，那么应该选择此候选方格，因为需要访问所有方格仅且一次；
- (3) 如果“可选邻近方格”的数量为 1 的候选方格不存在，则表明进入候选方格中的任意一个之后，均有至少二种方向可供选择行走，因此轮流选择上、下、左、右四个可行的候选方格之一进行回溯。

^I 读者可以参见 10.2.3 小节“哈密顿回”中的内容。

^{II} 参阅：<https://uva.onlinejudge.org/board/viewtopic.php?f=59&t=71655>, 2020。

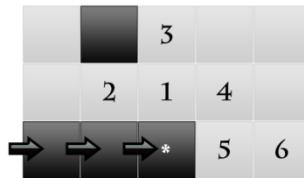


图 8-9 深色方格为已访问方格, 浅色方格为未访问方格, 当前行走到标记了“*”号的方格, 接下来可以选择进入 1 号方格或者 5 号方格。与 1 号方格相邻的为 2 号、3 号、4 号方格, 其中 2 号和 3 号方格的“可选相邻方格”数量均为 1, 4 号方格的“可选相邻方格”数量为 3, 按照剪枝技巧, 不应选择 1 号方格作为下一步行走的候选方格。对于 5 号方格来说, 与其相邻的有 4 号和 6 号方格, 其中 4 号方格的“可选相邻方格”数量为 3, 6 号方格的“可选相邻方格”数量为 1, 满足剪枝技巧的条件, 因此应该优先选择 5 号方格作为下一步行走的方格。

更进一步地, 由于前述的搜索方法采用的是单向搜索的方式, 还可以运用双向搜索以减少递归深度从而提高搜索效率, 不过这样编码的难度较高, 不易实现。

参考代码

```

int m, n, cnt, used[8][8], T;
int offset[4][2] = {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
int checkIn[4][2], checkMove[4], visited[8][8];

int dfs(int r, int c) {
    if (r < 0 || r >= m || c < 0 || c >= n) return 0;
    if (visited[r][c] || used[r][c]) return 0;
    visited[r][c] = 1;
    return 1 + dfs(r + 1, c) + dfs(r - 1, c) + dfs(r, c + 1) + dfs(r, c - 1);
}

inline bool go(int r, int c, int moves, int spots) {
    // 范围检查。
    if (r < 0 || r >= m || c < 0 || c >= n) return false;
    // 进入的方格未被使用。
    if (used[r][c]) return false;
    // 检测当坐标到达检查点时, 步数是否符合要求。
    if (r == checkIn[spots][0] && c == checkIn[spots][1])
        if (moves != checkMove[spots])
            return false;
    // 检测步数达到检查点的步数时, 坐标是否符合要求。
    if (moves == checkMove[spots])
        if (r != checkIn[spots][0] || c != checkIn[spots][1])
            return false;
    // 检测到达下一个检查点所需最少步数是否符合要求。
    int needed = abs(r - checkIn[spots][0]) + abs(c - checkIn[spots][1]);
    if (moves + needed > checkMove[spots])
        return false;
    // DFS 确定未访问的方格的连通性。
    used[r][c] = 1;
    memset(visited, 0, sizeof(visited));
    int unused = dfs(0, 1);
    used[r][c] = 0;
    if (unused != (T - moves))

```

```

        return false;
    // 检查是否到达终止方格。
    if (r == 0 && c == 1)
        if (moves != T)
            return false;
    // 符合要求的候选方格。
    return true;
}

void backtrack(int r, int c, int moves, int spots) {
    if (moves == T) { cnt++; return; }
    if (r == checkIn[spots][0] && c == checkIn[spots][1]) spots++;
    int rr, cc;
    for (int k = 0; k < 4; k++) {
        rr = r + offset[k][0], cc = c + offset[k][1];
        if (go(rr, cc, moves + 1, spots)) {
            used[rr][cc] = 1;
            backtrack(rr, cc, moves + 1, spots);
            used[rr][cc] = 0;
        }
    }
}

int main(int argc, char *argv[]) {
    int cases = 0;
    checkIn[3][0] = 0, checkIn[3][1] = 1;
    while (cin >> m >> n) {
        if (m == 0) break;
        for (int i = 0; i < 3; i++) {
            cin >> checkIn[i][0] >> checkIn[i][1];
            checkMove[i] = (i + 1) * m * n / 4;
        }
        T = m * n, checkMove[3] = m * n;
        memset(used, cnt = 0, sizeof(used));
        used[0][0] = 1;
        backtrack(0, 0, 1, 0);
        cout << "Case " << ++cases << ":" << cnt << '\n';
    }
    return 0;
}

```

10364 Square^B (正方形)

给定一组长度不等的木棍，你能够将其首尾相连使之构成一个正方形吗？

输入

输入第一行包含整数 N ，表示测试数据的组数。每组测试数据由一行构成，起始为一个整数 M ($4 \leq M \leq 20$)，表示木棍的数量，接着是 M 个整数，表示每根木棍的长度，长度为 1 至 10000 之间的某个整数。

输出

对于每组测试数据，输出一行。如果可以将所有木棍拼接成一个正方形，输出 **yes**，否则输出 **no**。

样例输入

```
3
4 1 1 1 1
```

样例输出

```
yes
no
```

5 10 20 30 40 50	yes
8 1 7 2 6 4 4 3 5	

分析

由于本题的测试数据规模较大，即使采取较为高效的剪枝技巧，还是难以在时间限制内获得通过。观察题目的约束，题目只是要求判断给定的木棍是否可以构成正方形，那么在回溯过程中可以将已经回溯的状态予以保存，当某次回溯再次到达此状态时可以直接返回不可行，因为此回溯过程只要一次成功即可，既然在此前的回溯过程中到达了此状态，但是未能成功返回，则说明此状态的木棍拼接并不可行，因此提前结束回溯可以显著缩短搜索时间。题目给定最多只有 20 根木棒，则可以使用一个整数数组来表示木棒的使用状态，结合位运算技巧，可以快速进行状态的标记和查询，以便在回溯过程中获取下一根可用的木棒，此技巧和动态规划中使用的备忘技巧类似。

参考代码

```

int n, stick[22], size, ONES, marked[(1 << 20) + 10] = {};

bool dfs(int used, int depth, int side) {
    if (marked[used]) return false;
    if (side == size) return dfs(used, depth + 1, 0);
    if (depth == 4) return true;
    marked[used] = 1;
    int available = ONES & (~used), next, bit;
    while (available) {
        next = available & (~available + 1);
        available ^= next;
        bit = __builtin_ctz(next);
        if (side + stick[bit] > size) continue;
        if (bit && stick[bit] == stick[bit - 1] && !(used & (next >> 1))) continue;
        if (dfs(used | next, depth, side + stick[bit])) return true;
    }
    return false;
}

int main(int argc, char *argv[]) {
    int cases;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> n;
        int side = 0;
        for (int i = 0; i < n; i++) {
            cin >> stick[i];
            side += stick[i];
        }
        if (side % 4 != 0) {
            cout << "no\n";
            continue;
        }
        size = side / 4;
        sort(stick, stick + n, greater<int>());
        if (stick[0] > size) {
            cout << "no\n";
            continue;
        }
        ONES = (1 << n) - 1;
    }
}

```

```

        memset(marked, 0, (1 << n) * sizeof(int));
        cout << (dfs(0, 0, 0) ? "yes" : "no") << '\n';
    }
    return 0;
}

```

强化练习: 10164 Number Game^D, 10419* Sum-Up the Primes^D, 10447* Sum-Up the Primes (II)^E。

11196 Birthday Cake^F (生日蛋糕)

今天是我母亲的生日, 我想为她制作一个生日蛋糕。蛋糕共有 m 层, 每层都是一个圆柱体, 整个蛋糕的体积恰好为 $n\pi$ 。从蛋糕的底部开始, 各层蛋糕的序号依次为 $1, 2, \dots, m$, 构成第 i 层蛋糕的圆柱体底面半径为 r_i , 高度为 h_i , 两者均为正整数, 为了使蛋糕看起来更为美观, 要求各层蛋糕满足以下条件: 对于任意 $i < m$, $r_i > r_{i+1}$, $h_i > h_{i+1}$ 。

整个蛋糕的表面都将被抹上冰淇淋, 为了使蛋糕的制作成本稍微低一些, 我们决定使用尽可能少的冰淇淋来实现这个目标。换句话说, 蛋糕的表面积应该尽可能的小 (蛋糕的最底面不计入表面积, 因为该面不需要涂抹冰淇淋)。

输入

输入包含至多 10 组测试数据。每组测试数据包含两个整数 n, m ($n < 100001, m < 11$), 分别表示蛋糕的体积和总层数。最后一组测试数据后跟着一个 0, 该行不需处理。

输出

对于每组测试数据, 输出测试数据的组数以及一个整数 S , 表示蛋糕的最小表面积为 $S\pi$ 。如果无法制作出符合要求的蛋糕, 输出 0。

样例输入

```
100 2
1000 3
0
```

样例输出

```
Case 1: 68
Case 2: 316
```

分析

如果单纯使用回溯不加剪枝, 将难以在限制时间内获得通过。令 r 为圆柱体的底面半径, h 为圆柱体的高, 则其体积 $V = \pi r^2 h$, 侧面积 $S_{\text{侧}} = 2\pi r h$, 底面积 $S_{\text{底}} = \pi r^2$, 表面积 $S_{\text{表}} = S_{\text{侧}} + 2S_{\text{底}}$, 且体积和侧面积存在关系 $S_{\text{侧}} = 2V/r$ 。

由于要求构成蛋糕的圆柱体从下层往上层半径和高均递减, 则在回溯时考虑从大到小枚举圆柱体的半径和高。在回溯时定义以下参数: $depth$: 表示回溯的层次, 即当前是为第几层蛋糕确定半径和高, 从 0 开始计数, 最底层的蛋糕对应第 0 层; $volume$: 当前蛋糕的总体积; $surface$: 当前蛋糕需要涂抹冰淇淋的总表面积, 由题意不难推知, 该表面积等价于构成蛋糕的各个圆柱体的侧面积加上最底层圆柱体的底面积; r : 当前层蛋糕的底面半径; h : 当前层蛋糕的高度; $best$: 满足体积为 n 层数为 m 的蛋糕的最小表面积。根据题目约束, 可以在回溯过程中应用以下优化技巧以提高效率:

(1) 当 $volume$ 大于 n 或 $surface$ 大于等于 $best$ 时予以剪枝;

(2) 当回溯层次大于 0 时, 即确定第 2 层 (或以上) 蛋糕的底面半径和高时, 根据体积和侧面积存在的关系 $S_{\text{侧}} = 2V/r$, 结合当前蛋糕的体积和目标体积的差, 可以得到由剩余体积而导致至少还需要增加的表面积 $remain = 2 \times (n - volume)/r$, 如果总的蛋糕表面积 $surface + remain \geq best$ 时予以剪枝;

(3) 根据剩余的体积 $n - volume$ 以及当前回溯的层次 m 可以得到当前层蛋糕的最大和最小可能底面半径;

(4) 根据当前蛋糕的体积 $volume$, 枚举的当前层蛋糕的底面半径 r_i , 后续每层蛋糕的最小可能高度 $m - depth$, 可以得到至少还需增加的体积 $V_{增} = r_i^2 \times (m - depth)$, 若 $volume + V_{增} > n$, 可以略过在底面半径为 r_i 时对当前层蛋糕高度 h_i 的继续搜索;

(5) 根据剩余的体积 $n - volume$ 以及当前回溯层数 m 、枚举的当前层蛋糕的底面半径 r_i 可以确定当前层蛋糕的最大和最小可能高度;

(6) 根据当前蛋糕的体积 $volume$, 枚举的当前层蛋糕的高度 h_i , 后续每层蛋糕的最小可能底面半径 $m - depth$, 可以得到至少还需增加的体积 $V_{增} = (m - depth)^2 \times h_i$, 若 $volume + V_{增} > n$, 可以不需进入下一层回溯;

(7) 考虑当前层蛋糕对体积的贡献 $V_{增} = r_i^2 \times h_i$, 由剩余的体积可以估算至少还需增加的表面积 $remain = 2 \times (n - volume - V_{增}) / r_i$, 如果总的蛋糕表面积 $surface + remain \geq best$ 时予以剪枝;

(8) 根据给定的初始蛋糕体积和层数, 可以预估得到最小表面积 $best$ 和初始的最大底面半径 r 以及最大高度 h 。

值得一提的是, 本题中对底面半径和高度的搜索方向显著影响了搜索效率, 从大到小进行搜索比从小到大进行搜索明显要快得多。可能的原因是在枚举较大的底面半径时, 缩小了高度的范围, 从而更快地得到一个较优的最小表面积, 进而能够剪除更多的搜索分支, 最终带来效率的较大提升。

参考代码

```
int n, m, best;

void dfs(int depth, int volume, int surface, int r, int h) {
    // 优化技巧 (1)。
    if (volume > n || surface >= best) return;
    // 回溯层次达到要求, 检查体积是否满足条件, 若满足更新最小表面积。
    if (depth == m) {
        if (volume == n) best = min(best, surface);
        return;
    }
    // 优化技巧 (2)。
    if (depth) {
        int remain = (n - volume) * 2 / r;
        if (surface + remain >= best) return;
    }
    // 优化技巧 (3)。
    int maxr = sqrt(1.0 * (n - volume) / (m - depth));
    for (int ri = min(maxr, r); ri >= (m - depth); ri--) {
        // 优化技巧 (4)。
        if (volume + ri * ri * (m - depth) > n) continue;
        // 优化技巧 (5)。
        int maxh = (n - volume) / (ri * ri);
        for (int hi = min(maxh, h); hi >= (m - depth); hi--) {
            // 优化技巧 (6)。
            if (volume + (m - depth) * (m - depth) * hi > n) continue;
            int volumeDiff = ri * ri * hi, areaDiff = 2 * ri * hi;
            if (!depth) areaDiff += ri * ri;
            // 优化技巧 (7)。
        }
    }
}
```

```

        int remain = (n - volume - volumeDiff) * 2 / ri;
        if (surface + remain >= best) break;
        dfs(depth + 1, volume + volumeDiff, surface + areaDiff, ri - 1, hi - 1);
    }
}

int main(int argc, char *argv[]) {
    int cases = 0;
    while (cin >> n) {
        if (n == 0) break;
        cin >> m;
        // 优化技巧 (8)。
        best = 4 * n;
        dfs(0, 0, 0, sqrt(n), n);
        if (best == (4 * n)) best = 0;
        cout << "Case " << ++cases << ":" << best << '\n';
    }
    return 0;
}

```

强化练习: 222 Budget Travel^B, 525 Milk Bottle Data^E, 560 Magic^D, 1217 Route Planning^E, 10890* Maze^D, 11127 Triple-Free Binary Strings^D。

扩展练习: 229 Scanner^D, 835* Square of Primes^D, 11199 Equations in Disguise^E。

8.3.1 正方形剖分

完美正方形剖分 (perfect square dissection) 是指给定一个边长为 n 的正方形, 确定能否将其剖分为若干大小各异的小正方形, 使得这些小正方形能够拼接得到原始的正方形。如果放宽问题的限制, 剖分得到的小正方形不要求大小各异, 则可以得到以下问题。

10270 Bigger Square Please...^D (拼接正方形)

Tomy 有许多正方形纸片, 边长从 1 到 $N-1$ 不等。实际上, 每种边长的正方形他都有无数张。他曾经以拥有这些正方形而自豪, 但是某一天, 他突然想要得到一个更大的正方形——边长为 N 的正方形。虽然他手头上并没有这样的正方形, 但是他可以通过将边长更小的正方形纸片拼接起来的以得到他想要的正方形。例如, 一个边长为 7 的正方形, 可以由以下 9 个更小的正方形组成。



注意, 在拼接过程中, 不能留下空白区域, 也不能将纸片放置在正方形的范围之外, 而且纸片也不能互相重叠。正如你所猜想的, Tomy 希望使用的纸片张数尽可能地少, 你能帮助他完成这个任务吗?

输入

输入第一行包含一个单独的整数 T , 表示测试数据的组数。每组测试数据为一个单独的整数 N ($2 \leq N \leq 50$)。

输出

对于每组测试数据, 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

```
1
3
```

样例输出

```
4
1 1 2
1 3 2
3 1 2
3 3 2
```

分析

令需要拼接的正方形边长为 N , 根据题意易知, 拼接方案是若干平方数之和。那么问题转化为如何将 N 表示为若干平方数之和且平方数的个数最小, 这可以通过回溯法解决。但是, 得到了一个将 N 拆分为平方数之和的方案, 并不表示就能将这些大小的纸片拼接成边长为 N 的正方形。例如, 对于 $N=5$, 可以拆分为 9 与 16 的和, 虽然 9 和 16 都是平方数, 但是实际上无法将一张边长为 3 和一张边长为 4 的纸片拼接为边长为 5 的正方形。所以, 在生成一个平方数和方案后, 需要进行验证放置, 若能放置, 则表明此方案可行, 予以记录, 将所有可行的方案记录后, 挑选其中纸片数最小的方案即为所求。这样的话, 将 N 拆分为平方数之和是一层回溯, 将拆分方案尝试放置又是一层回溯, 需要通过两层回溯来解决本问题。

考虑到需要两层回溯来搜索可能的解决方案, 如果不进行充分剪枝, 在限定的时间内无法获得通过。在将 N 拆分为平方数之和的这一环节, 如果能够得到一个拆分方案 (尽管不是最优的, 但是它的总个数较小且能实际放置), 将此方案中平方数的个数作为阈值可“剪除”很多无效的搜索分支。可以按照以下方法构造一个符合上述要求的非最优方案: 左上角放置一张边长为 $(N-2)$ 的纸片, 然后在右侧和下方尽可能多地放置边长为 2 的纸片, 剩余的空间放置边长为 1 的纸片, 这样总的纸片数 $N_{threshold} \leq 1 + \lfloor N/2 \rfloor + \lfloor (N-2)/2 \rfloor + 4$ (其中 “ $\lfloor x \rfloor$ ” 表示对 x 向下取整), $N_{threshold}$ 与 N 接近, 可以作为一个较好的剪枝阈值。通过回溯发现了总个数更少的“平方数之和”方案后, 则开始尝试实际放置: 建立一个网格 (可以使用二维数组表示), 当填充边长为 A 的纸片时, 在网格中查找是否有起始坐标为 (x, y) 且边长为 A 的空白区域, 若无此类空白区域, 表明该方案无法实际放置; 若能找到, 则枚举所有这样的起始位置, 逐一进行放置。在尝试某位置后, 需要将该区域标记为已填充, 若后继填充不成功返回时则撤销标记。对于已经生成但不能实际拼接的拆分方案需要予以记录, 在后继生成的方案中, 若有方案与记录的方案相同, 则直接忽略, 不必再次浪费时间进行搜索。

当 $N \bmod 2 = 0$ 或者 $N \bmod 6 = 3$ 时, 有特殊的解法。当 $N \bmod 2 = 0$ 时 (即 N 为偶数), 拼接方法为: 用 4 张边长为 $N/2$ 的纸片拼接得到边长为 N 的正方形。当 $N \bmod 6 = 3$ 时, 放置方法与 $N=3$ 的方案类似, 只不过将相应的纸片边长增加同样的数量。与此同时, 平方数 25 的拼接方案和 5 的拼接方案类似, 49 的拼接方案和 7 的拼接方案类似, 则在 2~50 之间的数, 只需要求出素数的拼接方法即可。万维网上已经有 2~50 之间的素数的拼接方案和最少需要纸片数, 如果只是为了解题, 利用这些信息能够显著减少计算时间,

也可以生成拼接方案后再提交^I。

8.3.2 关灯问题

关灯问题 (lights out puzzle)^{II}: 给定 $R \times C$ 的网格, 其中 $R \geq 1, C \geq 1$, 网格上每个方格内包含一盏电灯, 当按下某个方格内电灯的开关时, 会将此方格及其上下左右四个方格内的电灯亮灭状态反转, 亦即原先点亮的灯会熄灭, 原先熄灭的灯会点亮。假定按下某个方格内的电灯开关一次为一个步骤, 给定初始的电灯亮灭状态 (有的方格内电灯是亮的, 有的方格内电灯是灭的), 请确定是否可通过若干步骤将所有电灯熄灭, 如果可以则确定所需要的最少步骤数。如图 8-9 所示, 这是一个 8×8 的网格中电灯的初始亮灭状态。

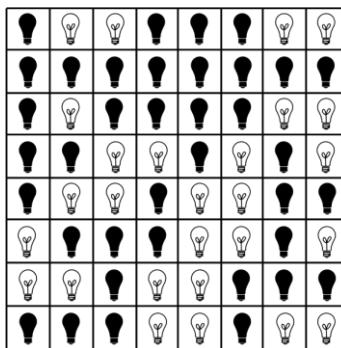


图 8-9 初始电灯状态。点亮的电灯为白色, 熄灭的电灯为黑色

分析

将网格视为一个矩阵, 亮的灯对应矩阵中为 1 的元素, 熄的灯对应矩阵中为 0 的元素, 则图 8-9 所示的初始状态可以使用一个 01 矩阵予以表示。

$$L = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}$$

按下网格内位于 (i, j) 的电灯开关等价于将初始状态矩阵 L 与一个“反转矩阵” A_{ij} 相加。其中“反转矩阵” A_{ij} 定义为一个 3×3 的矩阵, 此矩阵中只有按下的位置以及上下左右四个位置为 1, 其他位置为 0, 如果上下左右的四个位置位于边界之外则不予考虑。例如, A_{11} 对应于在网格(1, 1)按下电灯开关时所对应的“反转矩阵”, 同理可以推出 A_{12} 和 A_{22} 的含义。

^I 关于此问题, 读者可参阅: <http://mathworld.wolfram.com/MrsPerkinsQuilt.html>, 2020; 拼接方案及相应的求解代码可参阅: <http://mathpuzzle.com/perkinsbestquilts.txt>, 2020。

^{II} 参阅: <http://mathworld.wolfram.com/LightsOutPuzzle.html>, 2020。

$$\mathbf{A}_{11} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{A}_{12} = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \quad \mathbf{A}_{22} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

由于矩阵的加法满足交换律，即各个矩阵相加的顺序和最终结果无关，则可以将按下电灯的操作表示为一个线性代数方程。由于电灯的状态在反转两次后会恢复成原始的状态，因此上述线性方程可以视为模 2 状态下的线性方程组，有

$$\mathbf{L} + \sum_{ij} x_{ij} \mathbf{A}_{ij} = 0 \Rightarrow \sum_{ij} x_{ij} \mathbf{A}_{ij} = \mathbf{L}$$

使用高斯消元法在模 2 的情况下求解此线性方程组即可得到需要按下的电灯开关，计数为 1 的变量数量即为最少的操作步骤数。该方法的时间复杂度为 $O(R^3C^3)$ 。

可以证明，对于 $R=C$ 的情形，总是存在步骤数最少的唯一解。由于按下某个电灯开关时，反转的是一个“十字形”范围内电灯的状态，也就是说，在第 i 行第 j 列按下电灯，至多能使第 $(i-1)$ 行第 j 列的电灯状态发生反转，而对第 $(i-1)$ 行以前的电灯状态不可能产生影响。同时考虑到当 $R=C$ 时必定存在解，而且按下开关的先后顺序对最后结果不产生影响，因此可以先枚举第一行电灯的按下状态，由此逐步确定后续行电灯是否按下。也就是说，如果在预先确定了第 1 行电灯的按下状态后，当回溯到第 2 行的第 j 列，如果此时第 1 行的第 j 列电灯仍然是亮的，那么就只能通过按下第 2 行第 j 列的电灯开关来使得第 1 行的第 j 列的电灯变成熄灭状态，对于后续行的每一列，都可以根据上一行对应列电灯的状态来确定此列的电灯是否需要按下。换句话说，第 1 行电灯的按下状态决定了后续行所有电灯是否按下的状态。那么可以使用回溯法枚举第 1 行电灯按下与否的所有情形，之后根据第 1 行的电灯亮灭状态推导后续行电灯是否按下即可。

需要注意，如果按下电灯开关后对周围方格的影响不是一个“十字形”，而是其他不规则的模式，则关灯问题不一定存在可行解，此时需要通过回溯搜索所有可能的解来确定。当 n 较大时，还需要使用剪枝技巧才能在限定时间获得通过。一个有效的剪枝技巧是当回溯到第 3 行或第 3 行以后，若第 1 行仍有未熄灭的电灯，则此回溯分支可以剪除，因为后续已经无法通过按下某个电灯按钮来使得第 1 行的电灯状态发生反转。

强化练习：[10309 Turn the Lights Off^c](#)，[10318 Security Panel^c](#)，[11464 Even Parity^c](#)。

8.4 舞蹈链 X 算法

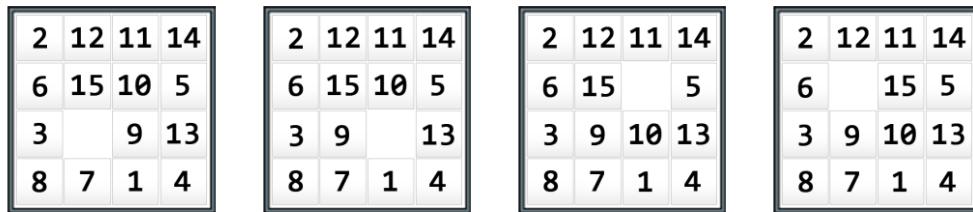
8.5 15 数码问题

[10181 15-Puzzle Problem^b](#) (15 数码游戏)

15 数码问题是一个很流行的游戏，即使你没有听说过这个名字，你也一定见过。它由 15 个滑块构成，每个滑块标记有 1 到 15 的数字，所有滑块被放置在一个边长为 4 的正方形外框中，留有一个空位。游戏的目标是移动滑块使得它们最终排列成如下形式：



唯一符合规则的移动方式是将空位旁的若干滑块中的一个移动到空位，考虑下面的一组移动：



(从左至右依次为) 起始布局, 空位右移 (R), 空位上移 (U), 空位左移 (L)。注意是空位移动而不是滑块移动

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 数码问题的初始布局，找出一种移动方法使其转化为目标布局。测试数据中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

输入的第一行包含一个整数 N ，表示测试数据的组数。接下来共 $4N$ 行输入描述了 N 个问题，每 4 行描述一个。其中 0 表示空位。

输出

对于每组输入数据输出一行。如果给定的初始状态无解，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

```
LLLLDRDRDR
This puzzle is not solvable.
```

分析

定义单个滑块的 n 值为：按行优先顺序，在当前滑块之后出现并小于当前滑块数字的滑块数量。给定某种布局，可以使用下述方法来判断是否有解¹：令 e 表示空滑块所在的行序号（从 1 开始计数），统计各滑块的 n 值总和

$$N = \sum_{i=1}^{15} n_i = \sum_{i=2}^{15} n_i$$

如果 $N+e$ 为偶数，则当前布局有解，否则无解。例如，给定以下初始布局：

¹ 参阅：<http://mathworld.wolfram.com/15Puzzle.html>, 2020。

13	10	11	6
5	7	4	8
1	12	14	9
3	15	2	0

按照行优先顺序, 在 13 号滑块之后出现且滑块数字小于 13 的滑块数量为 12 (滑块数字依次为 10、11、6、5、7、4、8、1、12、9、3、2), 在 10 号滑块之后出现且滑块数字小于 10 的滑块数量为 9 (滑块数字依次为 6、5、7、4、8、1、9、3、2) ……类似的, 可以得到其他滑块的 n 值分别为 (方括号内为滑块的数值, 方括号之前为滑块的 n 值): 9[11], 5[6], 4[5], 4[7], 3[4], 3[8], 0[1], 3[12], 3[14], 2[9], 1[3], 1[15], 0[2]。所有滑块的 n 值之和 $N=59$, 空滑块在第 4 行, 即 $e=4$, 则 $N+e=63$, 为奇数, 故以上布局不可解。

```
// 判断给定的布局是否可解。
bool solvable(vector<int> tiles) {
    int sum = 0;
    for (int i = 0; i < tiles.size(); i++) {
        if (tiles[i] == 0) sum += (i / 4 + 1);
        else {
            for (int j = i + 1; j < tiles.size(); j++)
                if (tiles[j] && tiles[j] < tiles[i])
                    sum++;
        }
    }
    return (sum % 2 == 0);
}
```

由于空滑块至少可向两个方向移动, 每增加一步, 产生的布局数量以指数形式增长, 不同的布局数量可能有 $16!=20922789888000$ 种, 远远超过穷尽搜索在限制时间内所能解决的数据规模。对于此类搜索类型题目, 可以使用以下几种方法尝试解决。

深度优先搜索

深度优先搜索 (Depth First Search, DFS)¹ 不断地向前寻找可行状态, 试图一次找到通向目标状态的路径, 它不会重复访问一个状态。由于某些问题所对应的搜索树包含大量的状态, 一般来说, DFS 只有在最大搜索深度固定的情况下才具有实用性。DFS 维护一个栈, 保存未访问过的状态, 在每次迭代时, DFS 从栈中弹出一个未访问的状态, 然后从这个状态开始扩展, 根据规定的走法计算其后继状态。如果达到了目标状态, 那么搜索终止, 否则, 任何在闭合集中的后继状态都会被忽略, 其他的未访问状态被压入栈中, 继续搜索。使用 DFS 解题的过程可以使用伪代码描述如下^[82]:

```
// initial 为初始状态, goal 为目标状态。
dfs (initial, goal) {
    // 如果初始状态即为目标状态, 不需继续搜索。
    if (initial == goal) return "Solution"
    // 将初始状态的深度置为 0。
    initial.depth = 0
    // 设置开放集, 开放集表示尚未访问的状态。此处使用栈来存储开放集。
    open = new Stack
```

¹ 此处的深度优先搜索和后续的广度优先搜索是图遍历的两种基本方式, 对图论及图遍历尚不熟悉的读者可以在阅读第 9 章“图遍历”的内容后再回过来理解本节内容。

```

// 设置闭合集，闭合集表示已经访问的状态。
closed = new Set
// 当开放集不为空且尚未找到符合要求的方案时继续搜索。
while (open is not empty) {
    // 从开放集中取出尚未访问的某个状态 n。
    n = pop(open)
    // 将状态 n 送入闭合集中。
    insert(closed, n)
    // 根据游戏规则，检查状态 n 的所有可行后继操作 m。
    foreach valid move m at n {
        // 在状态 n 上执行操作 m 以获得某个后继状态 next。
        next = state when playing m at n
        // 若闭合集中不包含状态 next，则表明该状态尚未访问，其深度增加 1。
        // 检查状态 next 是否已经为目标状态 goal，若为目标状态则返回解决方案。
        // 若不为目标状态且深度小于预设深度限制则将其置入开放集等待继续搜索。
        if (closed doesn't contain next) {
            next.depth = n.depth + 1
            if (next = goal) return "Solution"
            if (next.depth < maxDepth) insert(open, next)
        }
    }
}
// 若在限制深度内未发现解决方案则返回无解。
return "No Solution"
}

```

需要注意，在上述伪代码中是通过栈来实现 DFS，与使用递归实现的 DFS 稍有差异。使用递归实现的 DFS，在每个深度层次一般只保存一个状态，而此处使用栈实现的 DFS，在每个深度层次会生成当前状态“紧随其后下一步”的多个状态并压入栈中，相对于使用递归实现的 DFS 栈状态，栈中同层次的状态可能不止一个。

DFS 属于盲目搜索，在此过程中，大部分时间都用于在闭合集中搜索某个状态是否存在，如何快速地确定某个状态是否已经访问是提升算法效率的关键。如果能够为布局生成一个唯一键值，通过键值来判定两个布局的不同，则相对方便得多。也就是说，如果两个布局有着相同的键值则表示两个布局是等价的，若两个布局拥有不同的键值，则其必定不同，这样就能够通过检索键值是否存在来确定布局是否已经访问。观察滑块所处的正方形外框，共有 16 个位置，最大滑块数字为 15，不妨将布局视为 16 进制的数制系统，每个滑块作为一个数位，按行优先顺序将布局表示成一个 `unsigned long long int` 类型的整数。以前述的示例布局为例，可以将其唯一表示成：

$$\begin{aligned}
 S &= 13 \times 16^{15} + 10 \times 16^{14} + 11 \times 16^{13} + 6 \times 16^{12} + 5 \times 16^{11} + 7 \times 16^{10} + 4 \times 16^9 + 8 \times 16^8 + \\
 &\quad 1 \times 16^7 + 12 \times 16^6 + 14 \times 16^5 + 9 \times 16^4 + 3 \times 16^3 + 15 \times 16^2 + 2 \times 16^1 + 0 \times 16^0 \\
 &= \text{DAB657481CE93F20}_{16} \\
 &= 15759879913263939360_{10}
 \end{aligned}$$

通过此种方式，可以将布局唯一映射到一个整数，便于使用 `set` 数据结构来查询局面是否已经在闭合集中。更进一步地，如果将布局对应的整数表示为二进制数形式，则每 4 个二进制位恰好表示一个滑块，后继移动空滑块的操作可以转换为位操作，这样可以减少状态表示所占用的空间，同时使用位操作也可以提高代码效率。

因为事先无法确定某个布局所对应解的长度，如果初始时的深度限制较小，当实际解的长度大于深度限

制时，会出现 DFS 无法获得解的情形，其原因是 DFS 在达到解的深度之前就已经停止了搜索。对于本题来说，可能出现如下“诡异”的情形：某个布局距离目标布局只差几步，但由于达到了最大搜索深度限制而被放到了闭合集中，那么后续不可能再次对此布局进行扩展，即使之后的 DFS 过程在较小的深度等级访问到这个布局，它也不会继续搜索，因为这个布局已经在闭合集中。反过来，如果限制深度过大，则 DFS 会在某些无解的搜索分支上耗费大量时间，导致搜索时间大幅增加。因此，搜索深度设置是否合理直接影响着能否获得解和获得解所需要的时间。一般来说，对于可以使用 DFS 解题的题目，其搜索深度都有一定限制（或者题目本身所蕴含的搜索树深度不大，不需在题目描述中予以明确限定）。由于本题已经限定有解布局均可在 45 步之内解决，而且解的长度不应超过 50 步，则可设最大搜索深度为 50。

广度优先搜索

广度优先搜索（Breadth First Search, BFS）尝试在不重复访问状态的情况下，寻找一条从初始状态到达目标状态的最短路径。BFS 能够保证：如果存在一条从初始状态到目标状态的路径，那么找到的肯定是最短路径。BFS 和 DFS 唯一不同的就是 BFS 使用队列来保存开放集，而 DFS 使用栈。每次迭代时，BFS 从队列首部取出一个尚未访问的状态，然后从这个状态开始，确定能够到达的后继状态，如果已经达到目标状态，则结束搜索，任何已经在闭合集中的后继状态会被忽略，否则，新生产的后继状态将会放入开放集队列尾部，继续供后续搜索使用。使用 BFS 解题的过程可以使用伪代码描述如下（由于使用 BFS 解题和使用 DFS 解题差异非常小，请读者参照使用 DFS 解题的伪代码注释进行理解）：

```

bfs(initial, goal) {
    if (initial = goal) return "Solution"
    initial.depth = 0
    // BFS 使用队列来存储开放集。
    open = new Queue
    closed = new Set
    insert(open, initial)
    while (open is not empty) {
        // 从队列首部取出尚未访问的状态。
        n = head(open)
        insert(closed, n)
        foreach valid move m at n {
            next = state when playing m at n
            if (closed doesn't contain next) {
                if (next = goal) return "Solution"
                insert(open, next)
            }
        }
    }
    // BFS 搜索结束仍未找到解则返回无解。
    return "No Solution"
}

```

BFS 的优点是找到的解其长度必定最短，但缺点是空间占用较大，因为在迭代的每一步，BFS 会将下一层的所有状态压入队列，如果搜索空间或解的深度较大，将会有大量的状态停留在队列中，很容易超出内存的限制。

A*搜索

如果给定的布局存在解，BFS 能够找到一个最优解，但是可能需要访问大量的顶点，它并没有尝试对候

选走法进行排序，相反，DFS 是尽可能地向前探测路径，不过，DFS 的搜索深度必须得到限制，否则它很可能会在没有任何结果的路径上花费大量的时间。

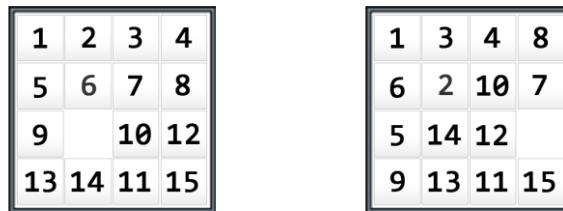


图 8-10 两个布局，左侧布局比右侧布局的“熵”要低。左侧布局到达目标状态最少只需要 3 步——“RDR”，而右侧布局到达目标状态最少需要 15 步——“LURULLLDLDDRURDR”

给定如图 8-10 所示的两个布局，直观来看，左侧布局要比右侧布局更接近目标状态，因为其“无序度”更小。直觉上，无序度较小的布局更有可能经过较少步骤的扩展后达到目标状态，相对的，无序度较大的布局在经过同等步骤的扩展后，到达目标状态的可能性要低。在搜索过程中，对于无序度低的布局可以优先考虑进行扩展，因为其更有可能在较短的步骤内到达目标状态。不过计算机并不具有类似于人类的直觉，需要程序员明确的告知特定布局的无序度，或者指定一组规则，使得计算机能够按照规则计算布局的无序度。在物理学中，表示系统无序程度时有一个物理量，称之为熵 (entropy)，熵值的增加可以使用系统增加的热量以及温度进行衡量。类似的，衡量给定布局的无序度，使用何种指标较为合适呢？观察左侧布局，之所以会得出其无序度较小的印象，是因为较多滑块处于其“正确”位置，而右侧布局则有相对较多的滑块未处于目标状态的“正确”位置，因此可以根据滑块与其“正确”位置的“距离”来度量布局的无序度。具体来说，就是使用滑块移动到其“正确”位置所需要使用的最少步骤数来量化某个滑块的无序度，所有滑块无序度的和即为整个布局的无序度。

定义以下三个函数：

$f(n)$ ：从初始状态开始，经过状态 n ，到达目标状态的最短走法序列；

$g(n)$ ：从初始状态到状态 n 的最短走法序列；

$h(n)$ ：从状态 n 到目标状态的最短走法序列；

对于给定的搜索问题，在得到实际解之前，状态 n 的 $f(n)$ 、 $g(n)$ 、 $h(n)$ 值是未知的，那么是否有一种方法来估算这些函数值，从而能够为搜索提供剪枝的决策信息呢？也就是说，能否使用一种方法得到这三个函数的尽可能准确的近似估计值，在搜索的时候，依据该估计值将那些大于估计值而不可能获得解的搜索分支予以剪除，从而提高搜索效率。这就是以下介绍的 A* 搜索。

A* 搜索在搜索时能够利用启发式信息，智能地调整搜索策略^[83]。A* 搜索是一种迭代的有序搜索，它维护一个布局的开放集合。在每次迭代时，A* 搜索使用一个评价函数 $f^*(n)$ 评价开放集合中的所有布局，选择具有最小“评分”的布局。定义：

$$f^*(n) = g^*(n) + h^*(n)$$

其中：

$f^*(n)$ ：估算从初始状态开始，经过状态 n ，到达目标状态的最短走法序列；

$g^*(n)$ ：估算从初始状态到状态 n 的最短走法序列；

$h^*(n)$: 估算从状态 n 到目标状态的最短走法序列;

星号表示使用了启发式信息¹，因此 $f^*(n)$, $g^*(n)$, $h^*(n)$ 是对实际开销 $f(n)$, $g(n)$, $h(n)$ 的估算，这些实际开销只能在得到解之后才能够知道。 $f^*(n)$ 越低，表示状态 n 越接近目标状态。 $f^*(n)$ 最关键的部分是启发式地计算 $h^*(n)$ ，因为 $g^*(n)$ 能够在搜索的过程中，通过记录状态 n 的深度计算出来。如果 $h^*(n)$ 不能够准确地区分开有继续搜索价值的状态和没有价值的状态，那么 A* 搜索不会表现得比 DFS 更好。如果能够准确地估算 $h^*(n)$ ，那么使用 $f^*(n)$ 就能够得到一个开销最小的解。可以将 A* 搜索使用伪代码描述如下：

```

a_star (initial, goal) {
    if (initial = goal) return "Solution"
    initial.depth = 0
    // A*搜索使用优先队列存储开放集。
    open = new PriorityQueue
    closed = new Set
    insert(open, initial)
    while (open is not empty) {
        n = minimum(open)
        insert(closed, n)
        if (n = goal) return "Solution"
        foreach valid move m at n do {
            next = state when playing m at n
            next.depth = n.depth + 1
            if (closed contains next) {
                prior = state in closed matching next
                if (next.score < prior.score) {
                    remove(closed, prior)
                    insert(open, next)
                }
                else insert(open, next)
            }
        }
    }
    return "No Solution"
}

```

由于 $h^*(n)$ 在算法中非常关键，而且它是高度特化的，根据问题的不同而有所差异，所以需要找到一个合适的 $h^*(n)$ 函数是比较困难的。在这里使用的是每个方块到其目标位置的曼哈顿距离之和，曼哈顿距离是该状态要达到目标状态至少需要移动的步骤数。 $g^*(n)$ 为到达此状态的深度，在这里采用了如下评估函数：

$$f^*(n) = g^*(n) + \frac{4}{3} \times h^*(n)$$

其中 $h^*(n)$ 为当前状态与目标状态的曼哈顿距离，亦可以考虑计算曼哈顿配对距离。对于本题来说，效率比单纯曼哈顿距离要高，但比曼哈顿距离乘以适当系数的方法低^[84]。

```

const int SQUARES = 4;
// 预先计算的曼哈顿距离。
int manhattan[SQUARES * SQUARES][SQUARES * SQUARES];
// 预先计算曼哈顿距离。
void getManhattan() {

```

¹ 自从 1968 年开发出此算法后，这个记法被广泛接受。

```

for (int i = 0; i < SQUARES * SQUARES; i++)
    for (int j = 0; j < SQUARES * SQUARES; j++) {
        int tmp = 0;
        tmp += (abs(i / SQUARES - j / SQUARES) + abs(i % SQUARES - j % SQUARES));
        manhattan[i][j] = tmp;
    }
}
// 计算某个布局的评分。
int getScore(vector<int> &state, int depth) {
    int gn = depth, hn = 0;
    for (int i = 0; i < state.size(); i++)
        if (state[i] > 0)
            hn += manhattan[state[i] - 1][i];
    return (gn + 4 * hn / 3);
}

```

迭代加深 A*搜索

迭代加深 A*搜索 (Iterative Deepening A* Search, IDA*) 依赖于一系列逐渐扩展的有限制的深度优先搜索。对于每次后继迭代，搜索深度限制都会在前次的基础上增加。IDA*比单独的深度优先搜索或广度优先搜索要高效得多，因为每次计算出的开销值都是基于实际的走法序列而不是启发式函数的估计。使用 IDA* 进行解题的过程可以使用伪代码描述如下：

```

ida_star (initial, goal) {
    if (initial = goal) return "Solution"
    // nowDepthLimit 为当前搜索的最大深度限制。
    nowDepthLimit = 0
    // nextDepthLimit 为已搜索状态的最小估计深度，是下一次迭代的参考深度限制。
    // 初始时，其值设置为给定布局的评分。
    nextDepthLimit = initial.score
    // 在未找到解之前继续迭代。
    while (true) {
        // 每完成一次迭代，更新 DFS 的最大深度限制。
        if (nowDepthLimit < nextDepthLimit)
            nowDepthLimit = nextDepthLimit
        else
            nowDepthLimit = nowDepthLimit + 1
        nextDepthLimit = INF
        // 根据上一次迭代得到的深度限制开始下一次 DFS。
        open = new Stack
        closed = new Set
        insert(open, initial)
        // 当开放集不为空时继续搜索。
        while (open is not empty) {
            n = pop(open)
            insert(closed, initial)
            foreach valid move m at n {
                next = state when playing m at n
                if (next = goal) return "Solution"
                if (closed doesn't contain next) {
                    // 将评分小于限制深度的布局压入栈中，对于评分大于限制深度的布局，
                    // 取最小的评分值作为下一次 DFS 的预设深度。
                    if (next.score < nowDepthLimit) insert(open, next)
                }
            }
        }
    }
}

```

```

        else nextDepthLimit = min(nextDepthLimit, next.score)
    }
}
}
return "No Solution"
}

```

IDA*搜索实际上可以看成使用启发式信息对 DFS 进行剪枝的过程。其中的剪枝阈值基于当前的实际走法步数和相对“智能”的对到达目标状态所需步数的估计。IDA*搜索的缺点是每次迭代均需要从新开始搜索，前一次迭代搜索的结果未能加以有效利用。在具体的实现中，避免重复生成之前的状态可以免去闭合集的使用，使得程序的效率更高。对于 15 数码问题来说，在移动空滑块时，如果当前将空滑块向右移动一个方格，那么下一步就应该避免将空滑块向左移动一个方格，因为这样会使得布局恢复到上一步的状态，对于将布局恢复到上一步状态的空滑块移动操作，需要将其剔除，这样就能够保证在 DFS 过程中遇到的布局是不重复的，从而免去了判断布局是否在闭合集中这一步骤，自然效率会提升。

参考代码

```

struct config {
    int hn, dir, r, c;
    bool operator<(const config &cfg) const { return hn < cfg.hn; }
};

string directions = "URLD";
int offset[4][2] = {{-1, 0}, {0, 1}, {0, -1}, {1, 0}}; // 位置偏移量
int puzzle[5][5]; // 记录布局
int Hn[16][5][5] = {};// 编号为 i 的滑块从正确位置移动到位置 [r, c] 时的熵的变化
int dHn[16][5][5][5][5] = {};// 编号为 i 的滑块从位置 [r, c] 移动到 [rr, cc] 时熵的变化
int done; // 搜索是否结束的标志
int depthLimit; // 最大深度限制
int path[64]; // 记录移动步骤

void dfs(int hn, int gn, int dir, int missingr, int missingc) {
    if (hn + gn > depthLimit) return;
    // 当布局的熵降低为零时，表明布局已经为目标状态。
    if (hn == 0) {
        done = 1;
        for (int i = 0; i < gn; i++) cout << directions[path[i]];
        cout << '\n';
        return;
    }

    // 确定从当前布局能够得到的后续布局。
    int cnt = 0;
    config next[4];
    for (int k = 0; k < 4; k++) {
        if (k == dir) continue;
        int rr = missingr + offset[k][0], cc = missingc + offset[k][1];
        if (rr < 1 || rr > 4 || cc < 1 || cc > 4) continue;
        // 更新布局的熵。
        next[cnt].hn = hn +
            dHn[puzzle[rr][cc]][rr][cc][missingr][missingc];
        next[cnt].dir = k, next[cnt].r = rr, next[cnt].c = cc;
        cnt++;
    }
}

```

```

        cnt++;
    }

    // 对具有较低熵的布局优先搜索。
    sort(next, next + cnt);
    for (int k = 0; k < cnt; k++) {
        swap(puzzle[missingr][missingc], puzzle[next[k].r][next[k].c]);
        // 记录移动的类型。
        path[gn] = next[k].dir;
        dfs(next[k].hn, gn + 1, 3 - next[k].dir, next[k].r, next[k].c);
        if (done) return;
        swap(puzzle[missingr][missingc], puzzle[next[k].r][next[k].c]);
    }
}

void IDAStar(int missingr, int missingc) {
    // 通过累加每个非空滑块的熵来确定给定布局的熵。
    int hn = 0;
    for (int r = 1; r <= 4; r++)
        for (int c = 1; c <= 4; c++)
            if (puzzle[r][c])
                hn += Hn[puzzle[r][c]][r][c];

    // 每当搜索不成功时，搜索的限制深度递增 1。
    depthLimit = 0;
    while (true) {
        done = 0;
        dfs(hn, 0, -1, missingr, missingc);
        if (done) break;
        depthLimit++;
    }
}

// 判断给定的布局是否可解。
bool solvable(vector<int> tiles) {
    int sum = 0;
    for (int i = 0; i < tiles.size(); i++) {
        if (tiles[i] == 0) sum += (i / 4 + 1);
        else {
            for (int j = i + 1; j < tiles.size(); j++)
                if (tiles[j] && tiles[j] < tiles[i])
                    sum++;
        }
    }
    return (sum % 2 == 0);
}

int main(int argc, char *argv[]) {
    // 预先计算正确位置为 [r1, c1] 的滑块移动到位置 [r2, c2] 时熵的改变。
    for (int r1 = 1; r1 <= 4; r1++)
        for (int c1 = 1; c1 <= 4; c1++)
            for (int r2 = 1; r2 <= 4; r2++)
                for (int c2 = 1; c2 <= 4; c2++)
                    Hn[(r1 - 1) * 4 + c1][r2][c2] = abs(r1 - r2) + abs(c1 - c2);

    // 预先计算编号为 i 的滑块从位置 [r1, c1] 移动到位置 [r2, c2] 时熵的变化。
}

```

```

for (int i = 1; i <= 15; i++)
    for (int r1 = 1; r1 <= 4; r1++)
        for (int c1 = 1; c1 <= 4; c1++)
            for (int r2 = 1; r2 <= 4; r2++)
                for (int c2 = 1; c2 <= 4; c2++)
                    dHn[i][r1][c1][r2][c2] = Hn[i][r2][c2] - Hn[i][r1][c1];

int cases = 0;
cin >> cases;
for (int cs = 1; cs <= cases; cs++) {
    vector<int> tiles;

    // missingr 和 missingc 表示空滑块所在的行和列。
    int missingr, missingc;
    for (int r = 1; r <= 4; r++)
        for (int c = 1; c <= 4; c++) {
            cin >> puzzle[r][c];
            tiles.push_back(puzzle[r][c]);
            if (puzzle[r][c] == 0) missingr = r, missingc = c;
        }

    // 若布局可解，则进行 IDA* 搜索。
    if (solvable(tiles)) IDAStar(missingr, missingc);
    else cout << "This puzzle is not solvable.\n";
}

return 0;
}

```

强化练习：529 Addition Chains^C, 652^{*} Eight^D, 10073^{*} Constrained Exchange Sort^D, 11163^{*} Jaguar King^D。

8.5 小结

回溯法（探索与回溯法）是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。当探索到某一步时，如果发现原先选择并不优或达不到目标，就退回一步重新进行选择。这种无法走通就回退，之后再前进的技术就称为回溯法，而满足回溯条件的某个状态就称为“回溯点”。在回溯法中，每次扩大当前部分解时，都面临一个可选的状态集合，新的部分解就通过在该集合中选择构造而成。这样的状态集合，其结构是一棵多叉树，每个树结点代表一个可能的部分解，它的儿子是在它的基础上生成的其他部分解。树根为初始状态，这样的状态集合称为状态空间树。

如果某个问题需要在所有可能的方案中确定某些方案是否符合要求，而且总的方案数不是太大（例如，约 10^6 种方案），那么使用回溯法是一种既简单又实用的方法。回溯法和穷举法类似，但有一定差异。穷举法要将一个解的各个部分全部生成后，才检查是否满足条件，若不满足，则直接放弃该完整解，然后再尝试另一个可能的完整解，它并没有沿着一个可能的完整解的各个部分逐步回退再次生成解的过程。而对于回溯法，一个解的各个部分是逐步生成的，当发现当前生成的某部分不满足约束条件时，就放弃该步所做的工作，退到上一步进行新的尝试，而不是放弃整个解重来。在回溯法实现时，最关键的就是既不重复也不遗漏地遍历所有可能的方案，然后再对方案进行合法性检查。回溯法最常见的实现形式是递归，递归实际上就是在问题所对应的隐式图中进行深度优先搜索，因此需要标记已经访问的状态以便回溯。如果标记状态不当，很容易造成遗漏某些可能的解或者重复检查方案。

回溯法一般是从初始状态向终止状态进行搜索（或者相反），是单向的过程，也可以从初始状态和终止状态同时进行搜索，在中间状态相遇，此为双向搜索，双向搜索能够减少回溯中每增加一步所造成的状态的大量增长，有利于减少空间和时间消耗，但不是所有问题都适合使用双向搜索技巧，双向搜索只适用于记录状态不需太多空间的搜索类型题目。

对于大多数搜索类型的问题来说，它的问题空间都比较大，因此需要根据问题的特点在回溯的过程中进行剪枝，将一些明显不可能得到解的搜索分支予以剪除，以提高效率。剪枝是一把双刃剑，剪枝过少对效率提升不明显，剪枝过多，用于确定是否剪除的计算量过大，反而会影响最终的效率，因此剪枝需要选择计算量较小而又能高效确定该分支是否可能走向可行解。需要具体题目具体分析，要靠解题者多接触、多练习、多思考，以期触类旁通、举一反三、熟能生巧。

对于更为复杂的问题，可以应用 A*搜索，即启发式搜索，启发式搜索是根据特定问题的特点构造的代价函数，可以更为高效的判定当前搜索路径是否更有可能走向可行解。代价函数与问题密切相关。

对于类似于数独的精确覆盖问题，还可以应用称之为舞蹈链 X 算法的搜索技巧，舞蹈链 X 算法提高了标记和回溯的效率，因此可以较好的提高搜索效率。

第9章 图遍历

其实地上本没有路，走的人多了，也便成了路。
——鲁迅¹，《故乡》

图 (graph) 是数学同时也是计算机科学的一个重要领域，是一种重要的数学模型和数据结构，通常用来描述某些事物之间的某种特定关系，它可以为运输系统、电路、人际交往、电线网络的组织结构提供抽象描述。很多不同的结构都可以用图来进行统一的形式建模，之后再运用特定的图算法对其进行解决。图论中的术语和记号非常多，每种图论专著或者论文都有可能不尽相同，为了一致性，本书第 9 章及第 10 章中有关图论的术语和记号均取自徐俊明编著的《图论及其应用》第三版的相关内容^[85]。

9.1 基本概念

图 $G=(V, E)$ 包含一个顶点集 (vertex set) V 和一个边集 (edge set) E ，其中每条边是 V 中两个点的有序对或无序对。边与它的两个端点称为关联的 (incident)，与同一条边关联的两个端点或者与同一个顶点关联的两条边称为相邻的 (adjacent)。两端点相同的边称为环 (loop)。有公共起点并有公共终点的两条边称为平行边 (parallel edges) 或者重边 (multi edges)。端点相同但方向相反的两条有向边称为对称边 (symmetric edges)。

9.1.1 图的属性

按照图具有的各种特点，可将图进行如下分类。

- 无向图和有向图。如果边 (x, y) 在边集中，同时 (y, x) 也在边集中，则称图 $G=(V, E)$ 是无向图 (undirected graph)，否则称该图是有向图 (directed graph)。常见的例子是城市中的街道，如果任意两个相邻路口之间都可以双向行驶，则可以把相邻路口之间的道路视为一条无向边，城市的交通图对应无向图。如果某些相邻路口之间的道路是单行道，则对应着一条有向边，相应交通图是有向图。
- 连通图和非连通图。如果无向图 G 中任意一对顶点之间都是连通的，则称此图是连通图 (connected graph)，否则称之为非连通图 (disconnected graph)。对于非连通图 G ，其极大连通子图称为连通分支 (connected component，或称连通分量)，连通分支数通常记为 $w(G)$ 。
- 加权图和无权图。在加权图 (weighted graph) 中， G 中的每个顶点或每条边都具有一个权值，该权值可能表示距离、时间等。在无权图中，顶点或边不具有权值。
- 有圈图和无圈图。如果图中存在一个顶点序列，从该序列的任意一个顶点出发沿着顶点间的边走能够回到起始顶点则称 G 为有圈图。在无向图中的圈称为无向圈，在有向图中的圈称为有向圈。如果一个有向图不存在圈，则称之为有向无圈图 (directed acyclic graph, DAG)。
- 简单图和非简单图。不包含环或平行边的图称为简单图，否则称为非简单图。图中顶点和边的数量分别记为 v 和 e ，定义 $v=|V|$ ， $e=|E|$ ，顶点的数量也称为阶 (order)，阶为 1 的简单图称为

¹ 鲁迅 (1881—1936)，原名周樟寿，后改名周树人，字豫山，后改豫才，“鲁迅”是他 1918 年发表《狂人日记》时所用的笔名。

平凡图 (trivial graph), 边数为 0 的图称为空图 (empty graph)。 v 和 e 都是有限的图称为有限图 (finite graph)。

强化练习: 11550 Demanding Dilemma^C。

- 嵌入图。嵌入图 (embedded graph) 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。
- 平面图和非平面图。若图 G 可以嵌入平面 (或球面), 则称 G 是平面图 (planar graph), 不能嵌入平面 (或球面) 的图称为非平面图 (non-planar graph)。
- 显式图和隐式图。如果某个应用中, 已经给到了图的具体结构, 亦即顶点和边, 只需对图进行搜索, 则在这种情形下的图称为显式图。而在其他的一些实际应用中, 很多图不是先被完整的构造出来然后再遍历, 而是在使用的时候逐步扩展, 即给出初始顶点、目标顶点、顶点间生成边的约束条件, 这样的问题所对应的图称为隐式图。
- 有标号图和无标号图。在有标号图 (labeled graph) 中, 每个顶点都被赋予一个唯一的名称, 将其与其他顶点区分开。在无标号图 (unlabeled graph) 中, 所有顶点均视为相同。在实际应用中, 大多数图的顶点都赋予了标号。在图的同构判定 (isomorphism testing) 中, 需要忽略标号对两个图的拓扑结构进行判断, 确定两个图是否完全相等。一般需要通过回溯法解决, 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。
- 完全图和非完全图。完全图 (complete graph) 是一个简单图, 图中每对顶点间有且只有一条边相连。具有 n 个顶点的完全图称为 n 阶完全图, 图中的边数为 $n(n-1)$ 。非完全图则是不满足完全图定义的图。

9.1.2 欧拉公式

欧拉于 1753 年证明, 对于连通平面图, 其顶点数 v , 边数 e , 面数 f 存在以下关系

$$v - e + f = 2$$

对于凸多边形, 也存在上述关系。如果平面图不连通, 设其连通子图数为 k , 则有

$$v - e + f = 1 + k$$

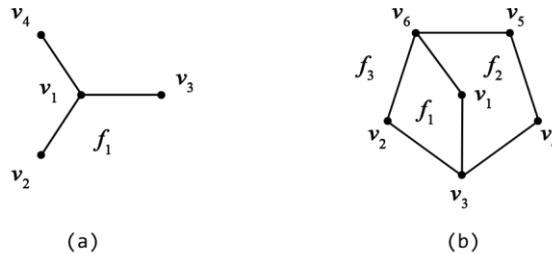


图 9-1 连通平面图 (a) 有 4 个顶点, 3 条边, 有 1 个面。连通平面图 (b) 有 6 个顶点, 7 条边, 有 3 个面。若图由 (a) 和 (b) 构成, 则总共有 10 个顶点, 10 条边, 连通子图数为 2, 面数为 3

扩展练习: 10178 Count the Faces^C。

9.1.3 路与连通

图中连接顶点 x 和 y 的链 (chain 或 walk) W 称为 xy 链, 是指顶点 x_i 和边 a_j 交错出现的序列, 即

$$W = (x =) x_{i_0} a_{i_1} x_{i_1} a_{i_2} \cdots a_{i_k} x_{i_k} (= y)$$

其中与边 a_{ij} 相邻的两个顶点 x_{i-1} 和 x_{ij} 正好是 a_{ij} 的两个端点, x 和 y 称为 W 的端点 (end-vertices), 其余的点称为内部点 (internal vertices)。 W 中边的数量 k 称为 W 的长度 (length), 简称为长。

边互不相同的链称为迹 (trail), 内部点互不相同的迹称为路 (path)。两端点相同的链 (迹、路) 称为闭链 (closed chain) (闭迹、闭路)。闭迹称为回 (circuit), 闭路称为圈 (cycle)。

指定有向图 D 中 xy 链 (迹、路) W 的方向从 x 到 y 。若 W 中所有边的方向与此方向一致, 则称 W 为 D 中从 x 到 y 的有向链 (迹、路), 记为 (x, y) 链 (迹、路)。

图中长度最大的路称为最长路 (longest path)。包含图中每个顶点的路称为 Hamilton 路。长为 $n-1$ 的路通常记为 P_n 。

9.2 图的表示

图有几种常用的表示方法, 每种表示方法都有其适用的场合, 可以根据具体解题需要灵活选择。

9.2.1 邻接矩阵

对于具有 n 个顶点的图, 可以使用一个 $n \times n$ 的矩阵来表示图中的边。令矩阵为 M , 若顶点 i 和顶点 j 之间有一条边, 则置 $M[i][j]=1$, 否则置 $M[i][j]=0$ 。此种表示方法的优点是可以快速查询某条边是否存在, 只要检查矩阵中的元素 $M[i][j]$ 是否为 1 即可, 对于新增和删除边也很便利, 只需将相应矩阵元素置为 1 或置为 0。但对于和某个顶点相连的顶点有哪些的查询, 需要对矩阵进行遍历, 不是很方便。其次当顶点数量较多但边数较少时, 邻接矩阵表示法效率不高, 矩阵中会有大量的元素为 0。

在解题应用中, 对图进行拓扑排序时, 若顶点数量不多 (例如顶点数量在 1000 个左右), 使用该种表示方法较为适宜, 因为在拓扑排序中, 关注的是某两个顶点间是否有边。

9.2.2 边列表和前向星

对于某些特定的图算法来说, 只需要维护所有边的数组即可, 不需要考虑边的具体顺序, 例如求最小生成树的 Kruskal 算法、求最大流的 Edmonds-Karp 算法等。在这些应用中, 只需从输入中读取边, 将其存放到边数组中, 对边数组进行处理后按顺序取用即可。

前向星是一种通过存储边信息来表示图的一种数据结构。通过读入每条边的信息, 将边存放在数组中, 把数组中的边按照起点顺序排列, 就完成了前向星的构造。为了方便查询某个顶点的相邻边, 可以使用另外一个数组 $head$ 来存储起点为 u 的第一条边在边数组中的位置。

```
//-----9.2.2.cpp-----//
const int MAXV = 1010, MAXE = 1000010;

// 定义边的结构。u 表示边的起始顶点, v 表示边的终止顶点, weight 表示边权。
struct edge {
    int u, v, weight;

    // 重载小于比较符。当边的起点相同时, 按终点的大小比较, 若终点相同则按照边权大小比较。
    bool operator<(const edge &e) const
    {
        if (u == e.u) {
            if (v == e.v) return weight < e.weight;
            else return v < e.v;
        } else return u < e.u;
    }
} g[MAXE];
```

```

// n 表示图中顶点的数量, m 表示图中边的数量, 均从 0 开始计数。
int n, m;

int head[MAXV];

int main(int argc, char *argv[]) {
    // 读入边数据。
    cin >> n >> m;
    for (int i = 0; i < m; i++)
        cin >> g[i].u >> g[i].v >> g[i].weight;

    // 对边进行排序。
    sort(g, g + m);

    // 确定每个顶点起始边在数组中的位置, 注意第一条边的处理。
    memset(head, -1, sizeof(head));
    head[g[0].u] = 0;
    for (int i = 0; i < m; i++)
        if (g[i - 1].u != g[i].u)
            head[g[i].u] = i;

    // 遍历边数组, 逐条输出每个顶点的邻接边。
    for (int i = 0; i < n; i++)
        for (int j = head[i]; g[j].u == i; j++) {
            cout << g[j].u << ' ' << g[j].v << ' ' << g[j].weight << '\n';
        }

    return 0;
}
//-----9.2.2.cpp-----//

```

可以看出, 前向星构造的时间复杂度主要取决于排序函数, 一般来说, 时间复杂度为 $O(E \log E)$, E 为边的数量, 空间上需要两个数组, 故空间复杂度为 $O(V+E)$ 。前向星的优点在于当顶点数量较多时仍然可以很好的处理, 适用于稀疏图, 同时可以存储重复边, 缺点是不能直接判断图中任意两个顶点间是否有边。

9.2.3 邻接表

可以为每个顶点设立一个数组, 将它的相邻顶点保存在其中, 称为邻接表 (adjacent list) 表示。动态数组可以使用 STL 中的 `vector` 来实现。在实际应用时, 一般从 0 开始对顶点进行编号 (也可以从 1 开始, 符合日常习惯), 为每个顶点创建一个 `vector`, 保存其相邻顶点, 所有存储相邻顶点的 `vector` 组合为一个更大的 `vector`, 类似于二维数组。在绝大多数图相关的题目中, 这种表示方法非常实用。

```

//-----9.2.3.cpp-----//
const int MAXV = 1010;

// n 为顶点数量, m 为边的数量。
int n, m;

// 声明存储顶点邻接表的 vector, 顶点序号从 0 开始计数。
vector<vector<int>> g(MAXV + 1);
// 如果图数据中边可能重复给出但不需要重复的边, 可以使用 set 来存储。
// vector<set<int>> g(MAXV + 1);
// 如果边的数量较多且不对边进行增删操作, 可以使用 list 来存储。

```

```

// list<list<int>> g(MAXV + 1);

int main(int argc, char *argv[]) {
    // 如果是无向图，在建立邻接表时需要添加正反两个方向的边。
    // 如果是有向图，则只需添加指定方向的边。
    cin >> n >> m;
    for (int i = 0, u, v; i < m; i++) {
        cin >> u >> v;
        g[u].push_back(v);
        g[v].push_back(u);
    }

    // 输出邻接表。
    for (int i = 0; i < n; i++) {
        if (g[i].size() > 0) {
            cout << i;
            for (int j = 0; j < g[i].size(); j++)
                cout << ' ' << g[i][j];
            cout << '\n';
        }
    }

    return 0;
}
//-----9.2.3.cpp-----

```

强化练习：10973 Triangle Counting^D。

9.2.4 链式前向星

链式前向星由前向星拓展而来，相当于将邻接表存储在数组中，而且省去了前向星的排序步骤。该种表示方法既能够快速遍历顶点的邻接边又能够节省存储空间，可以认为是遍历效率和存储效率最高的图表示方法。

```

//-----9.2.4.cpp-----
const int MAXV = 1010, MAXE = 1000010;

// 定义边结构。next 表示边数组中下一条边的位置。
struct edge { int u, v, weight, next; } g[MAXE];

int n, m;
int head[MAXV], idx;

int main(int argc, char *argv[]) {
    idx = 0;
    memset(head, -1, sizeof(head));

    // 读入边数据。这里假设读入的是无向图的边数据。注意边添加的方式。
    cin >> n >> m;
    for (int i = 0, u, v, weight; i < m; i++) {
        cin >> u >> v >> weight;
        g[idx] = edge{u, v, weight, head[u]};
        head[u] = idx++;
        g[idx] = edge{v, u, weight, head[v]};
        head[v] = idx++;
    }
}

```

```

// 输出所有顶点的邻接边。
for (int i = 0; i < n; i++)
    for (int j = head[i]; ~j; j = g[j].next)
        cout << g[j].u << ' ' << g[j].v << ' ' << g[j].weight << '\n';

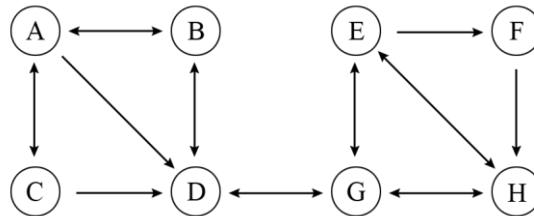
return 0;
}
//-----9.2.4.cpp-----

```

168 Theseus and the Minotaur^B (提修斯与米诺陶)

提修斯与米诺陶的神话传说是一个和充满羊肠小道的地下迷宫有关的故事。迷宫由一些相互连通的洞穴构成，有些洞穴只能从一个走向另外一个而不能反过来走。为了将米诺陶抓住，提修斯带了很多蜡烛进入迷宫，因为他发现米诺陶很怕光。开始的时候，提修斯在迷宫里漫无目的地寻找，等到他听见米诺陶靠近时，就立即点亮一支蜡烛开始追捕米诺陶。米诺陶转身选择与当前所在洞穴相连的一条道路逃跑。提修斯紧跟不放，慢慢的，提修斯到达了自点亮蜡烛以来经过的第 k 个洞穴。这时，他有足够的时间将点亮的蜡烛放在洞穴的中心，并将另外一支蜡烛点燃，继续追捕。在此追捕过程中，每当提修斯经过 k 个洞穴时，都会在第 k 个洞穴中心留下一支点亮的蜡烛，从而能够限制米诺陶的活动范围。当米诺陶进入某个洞穴后，它会按照特定的顺序查看该洞穴的离开通道，并选择第一个不会将它导向有点亮蜡烛的通道逃走。（记住，提修斯拿着点亮的蜡烛在追捕，因此米诺陶不会从进入该洞穴的通道原路逃走）。最终米诺陶被困住，使得提修斯有机会将它打败。

考虑以下的一个迷宫作为例子，在这里，米诺陶按照字典序对离开洞穴的通道进行检查：



假设提修斯在洞穴 C 时听到米诺陶从洞穴 A 靠近，在此示例中，设 k 为 3，提修斯点亮蜡烛开始追捕，依次通过洞穴 A, B, D （留下一支蜡烛）， G, E, F （留下另外一支蜡烛）， H, E, G （留下一支蜡烛）， H, E （米诺陶被困）。

编写程序模拟提修斯追捕米诺陶的过程。迷宫通过以下方式进行描述：每个洞穴以一个大写字母表示，与该洞穴相连通的洞穴按照米诺陶在逃走时选择的顺序相继给出，跟着给出第一次相遇时米诺陶和提修斯所在洞穴的编号，最后给出的是 k 值。

输入

输入包含多行。每行包含一种迷宫情形，每种情形的格式如下所示（样例输入给出的是之前讨论的情形）。每行输入不超过 255 个字符。输入以只包含 '#' 字符的一行作为结束。

输出

为每一种迷宫情形输出一行。输出时按照蜡烛被放置的顺序，给出洞穴的编号，以及米诺陶最终被困住时洞穴的编号，格式要求与样例输出的格式一致。

样例输入

```
A:BCD;B:AD;D:BG;F:H;G:DEH;E:FGH;H:EG;C:AD. A C 3
#
```

样例输出

```
D F G /E
```

分析

构建图的邻接表表示，按题目给定的规则进行模拟。每次米诺陶离开的洞穴为提修斯到达的洞穴，每隔指定数量的洞穴放置蜡烛，并标记此洞穴不能再次进入。注意对每个洞穴可达的其他洞穴按照字典序进行排序以方便决定米诺陶的逃走路线。

强化练习：173 Network Wars^D，243 Theseus and the Minotaur (II)^E，[10507 Waking up Brain^B](#)。

9.3 图遍历

图遍历（graph traversal）是指从图中的某个顶点出发，以某种方式沿着边访遍图中所有顶点的过程，在遍历过程中每个顶点仅被访问一次。图遍历操作在与图相关的题目中几乎是必需的。常用的遍历方法有两种：广度优先遍历（breadth-first search，BFS）和深度优先遍历（depth-first search，DFS）。由于遍历的本质是有序地访问所有顶点，上述两种方法的区别只不过是在遍历时访问顶点的顺序选择有所不同。

9.3.1 广度优先遍历

在广度优先遍历过程中，如果与当前顶点相连接的其他顶点尚未处理完毕，不会处理下一个未访问顶点。广度优先遍历总是尽可能“广泛”地遍历与当前顶点连接的边，遍历首先会发现与起始顶点 s 距离为 k 条边的所有顶点，然后才会发现与 s 距离为 $k+1$ 条边的顶点，此即“广度优先”的含义。

广度优先遍历使用队列来存储已经发现的顶点，其遍历过程可以使用伪代码表示如下。

```
// 从顶点 s 开始进行广度优先遍历。
bfs (顶点 s) {
    设置起始顶点 s 的为状态已访问，其他顶点状态为未访问；
    设置起始顶点的前驱顶点为空；
    将顶点 s 放入队列；
    while (队列不为空) {
        取出队列首的顶点 u;
        for 顶点 u 的每个邻接顶点 v
            if (顶点 v 未被访问) {
                将顶点 v 的状态设置为已访问；
                将顶点 v 的前驱顶点设置为 u;
                将顶点 v 放入队列中;
            }
    }
}
```

下面通过遍历一个无向连通图来示例广度优先遍历的使用。如图 9-2 所示，遍历从编号为 1 的顶点开始，目标是通过遍历确定其他顶点与编号为 1 的顶点间的最短距离。

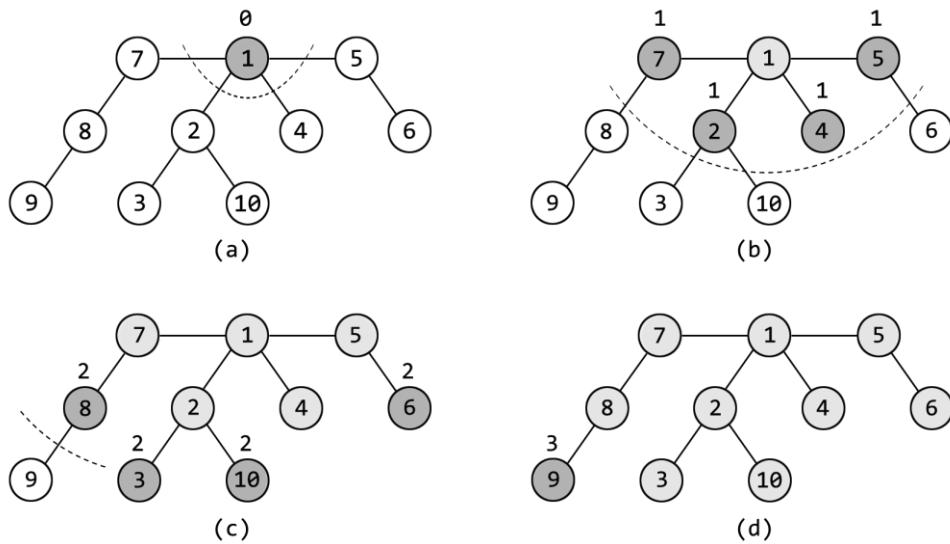


图 9-2 深灰色顶点表示已发现的顶点（上方的数值表示和顶点 1 的距离），浅灰色顶点表示已经访问过的顶点，白色顶点表示尚未发现的顶点。从编号为 1 的顶点开始遍历，在遍历过程中，已发现的顶点是已访问顶点和未访问顶点之间的边界（虚线所示为已访问顶点和未访问顶点的分界线）。（a），（b），（c），（d）表示遍历过程中四个不同的阶段

以下为图数据，其格式为：第一行为顶点的数量 n ，之后 n 行为编号从 1 到 n 的各个顶点所连接的其他顶点的数量和编号。

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

根据广度优先遍历的伪代码，可以容易地将其实现为以下代码。

```
-----9.3.1.cpp-----//
const int MAXV = 1010;

// 图中顶点的数量。
int n;

// parent 记录顶点的前驱，visited 记录顶点是否已经发现，dist 记录最短路径的边数。
int parent[MAXV], visited[MAXV], dist[MAXV];

// 以邻接表方式来表示图。
vector<int> g[MAXV];
```

```

// 广度优先遍历。
void bfs(int u) {
    // 初始化。
    memset(parent, -1, sizeof(parent));
    memset(visited, 0, sizeof(visited));
    memset(dist, -1, sizeof(dist));

    // 将起始顶点放入队列中。
    queue<int> q;
    q.push(u);
    visited[u] = 1, dist[u] = 0;

    // 当队列不为空时继续处理。
    while (!q.empty()) {
        // 取出尚未访问的顶点。
        u = q.front(); q.pop();
        // 遍历与当前顶点相连接的其他顶点。
        for (auto v : g[u])
            if (!visited[v]) {
                q.push(v);
                visited[v] = 1, parent[v] = u, dist[v] = dist[u] + 1;
            }
    }
}

int main(int argc, char *argv[]) {
    while (cin >> n) {
        // 读入图数据。
        for (int u = 1, edges; u <= n; u++) {
            g[u].clear();
            cin >> edges;
            for (int e = 1, v; e <= edges; e++) {
                cin >> v;
                g[u].push_back(v);
                g[v].push_back(u);
            }
        }

        // 使用广度优先遍历从第 1 个顶点开始遍历图。
        bfs(1);

        // 输出各顶点的父顶点以及与起始顶点间的最短距离。
        cout << "Vertex Parent Distance\n";
        for (int i = 1; i <= n; i++) {
            cout << setw(6) << right << i;
            cout << setw(10) << right << parent[i];
            cout << setw(12) << right << dist[i];
            cout << '\n';
        }
    }

    return 0;
}
//-----9.3.1.cpp-----//

```

将上述代码应用于图 9-2 所对应的数据，其输出为：

Vertex	Parent	Distance
1	-1	0
2	1	1
3	2	2
4	1	1
5	1	1
6	5	2
7	1	1
8	7	2
9	8	3
10	2	2

可以看到,由于广度优先遍历访问顶点的方式,离起始顶点近的顶点先发现,离起始顶点远的顶点后发现,可以利用这个性质来找到无权图中的最短路径。那么究竟该如何找到最短路径呢?可以借助 *parent* 数组来实现。在遍历过程中,起始顶点的前驱可以设置为一个“空”值(例如-1),表示该顶点是遍历的起始顶点,对于每一个在遍历过程中发现的顶点 v ,将其前驱设置为当前正在处理的顶点 u 。当遍历完成后,根据每一个顶点的前驱,逐次往回查找,即可得到从起始顶点到其他任意可达顶点的最短路径。在广度优先遍历过程中,每次只扩展一步,这样可以保证得到的最短距离中边数是最少的。如果边赋予的权值不等,那么得到的最短路径中,边权之和可能不是最小的。

在实际解题时,题目所给的图可能是隐式图,需要先根据题目的约束构建显式图然后再使用广度优先遍历。下面以两道难度递增的题目来示例广度优先遍历的应用。

924 Spreading the News^A (道听途说)

在大型机构中,每个员工都认识很多同事。然而,朋友关系只在员工中的少数人中间存在,小道消息也只在朋友圈中间传播。

假设以下情况:一旦某个员工知道了一条消息,在次日他会把这条消息告诉他的所有朋友。如果把第一个知道消息的员工称之为消息源,那么,在第一天的时候,消息源把消息告诉了他的朋友;第二天,消息源的朋友把消息告诉自己的朋友;第三天,消息源的朋友的朋友把消息告诉自己的朋友,如此继续。定义每天第一次(在此之前从未听说过该消息)得知该消息的人数为日知晓量,本题的目标是确定:

- (1) 最大日知晓量,即在一天中,第一次得知此消息的员工最大数量;
- (2) 首次最大日知晓量天数,在这一天,日知晓量首次达到最大日知晓量。

编写程序,在给定员工间的朋友关系以及消息源的情况下,确定消息扩散过程中的最大日知晓量以及首次最大日知晓量天数。

输入

输入的第一行包含一个整数 E ($1 \leq E \leq 2500$),表示员工的数量。员工的编号为 0 至 $E-1$ 。接下来的 E 行,每行给出一组员工的朋友关系(从编号为 0 的员工到编号为 $E-1$ 的员工)。每组朋友关系的第一个数值 N ($0 \leq N < 15$),表示该员工的朋友数量,接着 N 个不同的整数表示该员工所有朋友的编号,编号以一个空格分隔。

接下来的一行包含一个整数 T ($1 \leq T < 60$),表示消息源的个数。后续的 T 行,每行包含一个员工的编号,即消息源的编号。

输出

对于每组测试数据输出一行。假如除了消息源之外的员工均未得知该消息,输出整数‘0’;否则输出

两个整数 M 和 D , M 为最大日知晓量, D 为首次最大日知晓量天数, 中间间隔一个空格。

样例输入

```
6
2 1 2
2 3 4
3 0 4 5
1 4
0
2 0 2
3
0
4
5
```

样例输出

```
3 2
0
2 1
```

分析

由题意可知, 员工之间的朋友关系构成了一张图。员工将消息告诉朋友的整个过程就是一个广度优先遍历的过程。为每个员工设立一个 **标记**, 表示该员工是第几天才第一次知晓该消息, 可以认为消息源是于第 0 天知晓该消息, 然后利用广度优先遍历, 模拟消息传播的过程, 计数每天第一次知晓该消息的员工数量。由于某个员工要么至少有一个朋友, 要么没有朋友, 消息在 n 个员工组成的朋友圈中传播完毕, 至多需要 n 天。需要注意的是本题环境下, 员工之间的朋友关系不是相互的, 对应的图是有向图而不是无向图, 也就是说, 如果员工 A 的朋友中有 B , 表示 A 会于次日将消息告知 B , 但并不表示 B 会于得知消息的次日将消息再告诉 A 。

强化练习: 10044 Erdős Numbers^B, 10067 Playing with Wheels^A, 10187 From Dusk till Dawn^C。

扩展练习: 368 Indexing Web Pages^E, 407 Gears on a Board^D, 487 Boggle Blitz^C, 704* Color Hash^C, 10039 Railroads^C。

在有关 BFS 的题目中, 有一部分题目所蕴含的是隐式有向图。出题者一般会将题目的底层模型进行适当伪装, 但是具有解题经验的解题者往往能够立即判断出需要使用 BFS 进行解题。此类题目大致可以分为两种主要的类型。

第一种类型, 题目只要求确定从初始状态达到目标状态的最少步骤数, 并不要求给出到达目标状态的具体路径, 可按照下述步骤解题:

- (1) 判定初始状态的构成, 将其表示成一个结构体或者类, 使用相应的域来表示状态的各个属性;
- (2) 构建状态队列, 可以使用 STL 中的 `queue` 数据结构来实现; 将初始状态送入队列;
- (3) 从队列中取出位于队首的状态, 判断该状态是否已经达到目标状态。若不为目标状态, 则根据题目所给定的规则将此状态转换为后继状态并送入队列。这一步骤往往是难点所在, 解题者可能需要应对以下挑战: (a) 出题者所设置的状态转换规则较为复杂, 具体实现时需要谨慎细致; (b) 需要应用特定的技巧来高效地表示状态, 例如使用位来表示属性, 使用位运算来进行状态的转换; (c) 搜索空间较大, 需要使用类似于回溯法中的剪枝技巧来缩小搜索空间, 即设立一个已访问状态集合 S , 每次在将状态放入队列之前先检查该状态是否已经在 S 中存在, 如果已经存在, 则后续 BFS 中不需再次访问, 这样可以减小搜索空间, 提高效率。

第二种类型, 题目不仅要求确定到达目标状态的最少步骤数, 而且还需要确定具体的路径, 此时需要将题目所蕴含的隐式有向图显式进行表示, 构建完整的邻接表, 使用标准的 BFS 过程进行遍历, 记录前驱顶

点和距离，之后根据前驱顶点信息构建最短路径。

321 The New Villa^C (新别墅)

布莱克先生 (Mr. Black) 最近在市郊购买了一套别墅。这套别墅其他都好，只有一点让他不满意——尽管大多数房间都有顶灯的开关，但是这些开关所控制的却是其他房间的顶灯。房地产中介将开关的这个“特性”做为一个卖点向顾客推销，但是布莱克先生认定是由于电工的粗心大意，把不同房间的开关线路接混了而导致了这种情况的出现。

某天晚上，布莱克先生回家有点晚，当他站在别墅的大厅时，发现除了大厅以外，其他房间的灯都是关着的。由于布莱克先生怕黑，所以他不敢走进没有开灯的房间，也不敢将当前所在房间的灯关掉。

在思考了一阵子之后，布莱克先生意识到他可以利用开关的“特性”来控制灯的关闭顺序，以便他从大厅到达他的卧室时，保证除了卧室的灯是开着的之外，其他房间的灯都是关闭的。

现在需要你编写一个程序，给定别墅的描述，在最初只有大厅的灯是开着的情况下，确定如何从大厅到达卧室。你不能进入尚未开灯的房间，在进入卧室之后，需要能够使得除了卧室的灯是开着的，其他房间的灯都是关闭的。假如有多种方法到达卧室，你必须找到其中步骤数最少的方法，在本问题的环境下，“从一个房间进入另外一个房间”，“打开某个房间的灯”，“关上某个房间的灯”，均视为一个步骤。

输入

输入包含多座别墅的描述。每座别墅描述的起始一行包含三个整数 r, d, s 。 r 表示别墅的房间数，最多有 10 个房间。 d 表示别墅中门的扇数， s 表示别墅中开关的总数。房间从 1 到 r 编号，编号为 1 的房间为大厅，编号为 r 的房间为卧室。之后接着 d 行，每行包含两个整数 i 和 j ，表示房间 i 和房间 j 之间有一扇门相连。之后的 s 行，每行包含两个整数 k 和 l ，表示在房间 k 的开关控制着房间 l 的顶灯。在每两座别墅的描述之间有一个空行。输入以 $r=d=s=0$ 结束，该行不需处理。

输出

对于每座别墅，先输出测试用例的序号 (“Villa #1”, “Villa #2”，依次编号)。假如存在可行的步骤序列，使得布莱克先生能够从大厅到达卧室，且满足最后只有卧室的灯是亮着的要求，输出其中具有最少步骤数的一种方案。输出格式参照样例输出。假如不存在这样的序列，输出以下信息：“The problem cannot be solved.” 在每组测试用例的输出后面附加一个空行。

样例输入

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

样例输出

```
Villa #1
The problem can be solved in 6 steps:
- Switch on light in room 2.
- Switch on light in room 3.
- Move to room 2.
- Switch off light in room 1.
- Move to room 3.
- Switch off light in room 2.
```

分析

初看似乎可以使用回溯法解决，因为题目本质上是确定一个进入房间和离开房间的序列 $r_1r_2\cdots r_n$ ，使得对于任意房间 r_i ($1 < i \leq n$)，可以在进入 r_i 之前打开该房间的灯，同时对于任意房间 r_j ($1 \leq j < n$)，可以在离开 r_j 后关闭该房间的灯。但是题目所求为具有最少操作步骤的解决方案，由于可以多次进入同一个房间，

使用回溯法所对应的搜索空间很大，不太容易编码实现且容易超时。

可以从图论角度解决本问题。由于一个房间的灯要么是亮着的，要么是灭的，只有两种状态，且至多有 10 个房间，则所有房间灯的亮灭状态至多有 1024 种。将布莱克先生在不同房间时所有灯的亮灭状态之间的关系构建成一个有向图，通过对该图进行广度优先遍历，即可得到具有最少操作步骤的解决方案。关键是如何将房间位置和灯的状态表示成图中的顶点并找出顶点之间的边。

可以使用一个二进制数来表示灯的亮灭状态并将其转换为一个整数。例如，当 $r=3$ 时，使用 3 个二进制位来表示 3 个房间的灯，如果灯亮，该位为 1，否则为 0，则所有可能的灯的亮灭状态依次为： $000_2=0_{10}$ ， $001_2=1_{10}$ ， $010_2=2_{10}$ ， $011_2=3_{10}$ ， $100_2=4_{10}$ ， $101_2=5_{10}$ ， $110_2=6_{10}$ ， $111_2=7_{10}$ ，总的状态数为 $2^r=8$ 种，最后表示时使用对应的十进制数来标识灯亮灭状态。对于所处房间，由于灯的状态数最多为 1024 种，将房间编号数左移 10 位，然后与表示灯的状态进行“位或”操作即可得到唯一表示房间和灯状态的一个整数。进行转换时可以使用 STL 中的 `bitset` 数据结构来简化操作。

在将状态唯一映射到一个整数后，可以通过房间之间门的连接和开关的关系进一步得到各顶点之间的边。例如，令 s_i 表示在房间 r_i 时的状态， s_j 表示在房间 r_j 时的状态，如果房间 r_i 和 r_j 有门连通，则可以从状态 s_i 到达 s_j ，表明顶点 s_i 和 s_j 之间有一条有向边。如果在同一房间，可以将开关打开或关闭，得到相应状态所对应的顶点之间的有向边。

强化练习：280 Vertex^A，298 Race Tracks^D，314 Robot^C，439 Knight Moves^A，532 Dungeon Master^A，627 The Net^B，816 Abbott's Revenge^C，899 Colour Circles^D，928 Eternal Truths^D，10085 The Most Distant State^C，10097* The Color Game^C，10120* Gift?!^C，11198 Dancing Digits^D，11513 9 Puzzle^D，11573 Ocean Currents^C，11974 Switch The Lights^D，12135 Switch Bulbs^D，12826 Incomplete Chessboard^D。

扩展练习：176 City Navigation^D，224 Kissin' Cousins^E，859 Chinese Checkers^D，949 Getaway^D，985 Round and Round Maze^D，1251 Repeated Substitution with Sed^E，1253 Infected Land^E，1601 The Morning after Halloween^D，10021* Cube in the Labirint^D，10682 Forró Party^D，10993 Ignoring Digits^D，11160 Going Together^D，11329 Curious Fleas^E，12569 Planning Mobile Robot on Tree^D，12797* Letters^D。

9.3.2 深度优先遍历

深度优先遍历，它所遵循的策略是尽可能深地搜索一个图，在搜索中，对于新发现的顶点，如果它还有以此为起点而未探测到的边，就沿此边继续探测下去。当顶点 v 的所有边都已被探寻过后，搜索将回溯到发现顶点 v 作为终止顶点的那些边的起始顶点 u 。这一过程一直进行到已发现从源顶点可达的所有顶点为止。如果还存在未被发现的顶点，则选择其中一个作为源顶点并重复以上过程。整个过程反复进行，直到所有的顶点都已被发现为止。整个遍历过程形成了一个由数棵深度优先树组成的深度优先森林。

为了充分利用深度优先遍历，可以在遍历过程中为每个顶点设立额外的数据结构来记录顶点的发现时间 $dfn[u]$ 和完成时间 $ft[u]$ ，以及顶点的颜色 $color[u]$ 。 dfn 为深度优先数 (depth first number)，在遍历时，如果深度优先数越小，表明在遍历的过程中更早地发现该顶点； ft 为顶点完成对其子树进行遍历的时间 (finish time)，完成时间越早表明其越靠近深度优先树的叶结点； $color$ 表示 DFS 过程中顶点的访问状态，可以使用它来对边进行分类，开始时，每个顶点均为白色，搜索中被发现时设置为灰色，结束时设置为黑色，这样可以保证每个顶点在搜索结束时，只存在于一棵深度优先树中。

深度优先遍历利用了递归，相当于使用栈来替代广度优先遍历中的队列，显得更加简洁。当然也可以使用栈来消除递归的使用。

```
//-----9.3.2.cpp-----//
```

```

const int MAXV = 1010;
const int WHITE = 0, GRAY = 1, BLACK = 2;

// 使用邻接表来表示图。
vector<vector<int>> g(MAXV);

// parent 记录各顶点的前驱;
// dfn 记录顶点的发现时间;
// ft 记录顶点的完成时间;
// color 标记访问状态。
int parent[MAXV], dfn[MAXV], ft[MAXV], color[MAXV];

// 时间戳及顶点数量。
int dfstime, n;

// 深度优先遍历。
void dfs(int u) {
    // 记录顶点的发现时间并标记顶点为灰色, 表示该顶点已发现。
    dfn[u] = ++dfstime, color[u] = GRAY;
    // 处理与当前顶点相连接的其他顶点。
    for (auto v : g[u])
        if (!color[v]) {
            parent[v] = u;
            dfs(v);
        }
    // 记录顶点的完成时间并标记顶点为黑色, 表示该顶点遍历已完成。
    ft[u] = ++dfstime, color[u] = BLACK;
}

void search() {
    memset(color, WHITE, sizeof(color));
    memset(parent, -1, sizeof(parent));

    dfstime = 0;
    for (int u = 0; u < n; u++)
        if (!color[u])
            dfs(u);
}
//-----9.3.2.cpp-----//

```

通过在 DFS 过程中为顶点着色, 可以对输入图 $G=(V, E)$ 的边进行分类。根据图 G 上进行 DFS 所产生的深度优先森林 G_π , 可以将图的边分为四种类型:

- (1) 树边 (tree edge), 是深度优先森林中的边, 如果顶点 v 是在探寻边(u, v)时被首次发现, 那么(u, v)就是一条树边。
- (2) 反向边 (back edge), 在深度优先树中, 连接顶点 u 到它的某一祖先顶点 v 的那些边, 即在深度优先遍历过程中再次访问到了颜色为灰色的顶点。有向图中出现的自环边可以认为是反向边。
- (3) 正向边 (forward edge), 在深度优先树中, 连接顶点 u 到它的某个后裔 v 的非树边(u, v)。
- (4) 交叉边 (cross edge), 不同于上述三种的其他类型的边, 在同一棵 (或者不同) 深度优先树中的两个顶点之间的一条边, 其中一个顶点不是另一个顶点的祖先, 即在深度优先遍历过程中再次访问到了颜色为黑色的顶点。

可以证明, 在对一个无向图 G 进行深度优先搜索的过程中, G 的每一条边要么是树边, 要么是反向边。

而在有向图中, 如果出现反向边, 则表明图中存在圈, 使用这个性质可以判定给定的图是否存在圈。

11504 Dominos^A (多米诺骨牌)

多米诺骨牌非常好玩。孩子们喜欢将骨牌排成长列, 当把第一张骨牌推倒时, 它会压倒第二张骨牌, 第二张骨牌又会压倒第三张, 最后整列的骨牌都会倒下。然而有时候, 有些骨牌没能倒下, 这时需要我们手动将其推倒以便它能够继续推倒其他骨牌。

给定一些骨牌的排列, 你的任务是确定要使得所有的骨牌都倒下, 需要手动推倒的最少骨牌数。

输入

输入的第一行为一个整数, 表示输入中测试数据的组数。每组测试数据的第一行包含两个整数, 整数均不大于 100000, 第一个整数 n 表示骨牌的数量, 第二个整数 m 表示此组测试数据包含 m 行骨牌之间关系的数据。骨牌从 1 到 n 进行编号。

每组数据中接下来的 m 行, 每行包含两个整数 x 和 y , 表示如果骨牌 x 倒下将导致骨牌 y 倒下。

输出

对于每组测试数据, 输出一行, 表示如果要使所有骨牌倒下, 需要手动推倒骨牌的最少数量。

样例输入

```
1
3 2
1 2
2 3
```

样例输出

```
1
```

分析

初看似乎很简单, 直接使用深度优先遍历或者并查集求连通分支数即可以解决。但是由于题目中给出的并不是无向图而是有向图, 简单的使用深度优先遍历或者并查集算法并不总能够得到正确答案。读者可以自行尝试使用并查集和深度优先遍历进行解题, 会发现总有一些无法正确处理的情况。之所以会出现这样的问题是因为给定的有向图可能会存在有向圈, 仅靠一次单纯的深度优先遍历无法确定需要手动推倒的“关键骨牌”。另外一种解题方法是先寻找强连通分支, 对强连通分支进行“缩点”操作, 将题目给定的图转化为有向无圈图, 之后统计新图中入度为 0 的顶点数即为需要手动推倒的骨牌数¹。

参考代码

```
const int MAXV = 100010;

int visited[MAXV], cases, n, m;
vector<int> g[MAXV];
stack<int> s;

// 第一次深度优先遍历, 记录骨牌倒下的顺序, 先倒下的在栈顶, 后倒下的在栈底。
void dfs(int u) {
    visited[u] = 1;
    for (auto v : g[u])
        if (!visited[v])
            dfs(v);
}

// 第二次深度优先遍历, 将所有倒下的骨牌归为一个连通分支
void dfs2(int u) {
    visited[u] = 1;
    for (auto v : g[u])
        if (!visited[v])
            dfs2(v);
}

int main() {
    cin >> cases;
    for (int i = 0; i < cases; ++i) {
        cin >> n >> m;
        for (int j = 0; j < m; ++j) {
            int x, y;
            cin >> x >> y;
            g[x].push_back(y);
        }
        int ans = 0;
        for (int j = 1; j <= n; ++j)
            if (!visited[j])
                ans++;
        cout << ans << endl;
    }
}
```

¹ 关于求强连通分支的算法, 请读者参见第 9.4.8 小节“强连通分支”中的内容。

```

        s.push(u);
    }

// 第二次深度优先遍历, 从栈顶开始。
void rdfs(int u) {
    visited[u] = 1;
    for (auto v : g[u])
        if (!visited[v])
            rdfs(v);
}

int main(int argc, char *argv[]) {
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        // 初始化。
        while (!s.empty()) s.pop();
        for (int u = 1; u <= n; u++) g[u].clear(), visited[u] = 0;
        // 以邻接表方式读入图数据。
        cin >> n >> m;
        for (int e = 1, u, v; e <= m; e++) {
            cin >> u >> v;
            g[u].push_back(v);
        }
        // 按正常方式进行深度优先遍历, 并将顶点入栈。
        for (int u = 1; u <= n; u++)
            if (!visited[u])
                dfs(u);
        // 从栈顶开始退栈, 如果栈顶骨牌尚未推倒, 表明它是需要手动推倒的骨牌。
        int cc = 0;
        for (int u = 1; u <= n; u++) visited[u] = 0;
        while (!s.empty()) {
            int u = s.top();
            if (!visited[u]) rdfs(u), cc++;
            s.pop();
        }
        cout << cc << '\n';
    }
    return 0;
}

```

强化练习: [291 The House Of Santa Claus^A](#), [614 Mapping the Route^C](#), [988 Many Paths One Destination^C](#), [10461 Difference^C](#), [10562 Undraw the Trees^B](#), [10926 How Many Dependencies^B](#), [11518 Dominos 2^A](#), [11749 Poor Trade Advisor^C](#), [11906 Knight in a War Grid^B](#), [12376 As Long as I Learn I Live^C](#), [12648 Boss^D](#)。

扩展练习: [284 Logic^E](#), [464 Sentence/Phrase Generator^D](#), [840* Deadlock Detection^D](#), [1263 Mines^D](#), [10802* Lex Smallest Drive^D](#), [11131 Close Relatives^D](#), [12442 Forwarding Emails^B](#), [12582 Wedding of Sultan^C](#), [13015* Promotions^D](#)。

9.4 图遍历的应用

9.4.1 图的连通性

在解题中, 根据题意在建立“显式图”后, 通过 DFS (或 BFS) 可以很容易地判断从某个起始顶点出发

能否到达所有其他顶点，从而确定图的连通性（connectivity）。一般来说，解题的难点在于如何将给定的题目约束条件转换为“显式图”或者从中“提炼”出图的连通性模型。

718 Skyscraper Floors^D (摩天大楼楼层)

某座摩天大楼有一套特别的电梯系统，每部电梯只能从 Y 楼层出发，向上每经过 X 层时停留，最低停留楼层为 Y 楼层。现在需要通过电梯将某件家具从楼层 A 搬运到楼层 B ，给定电梯系统的描述，确定是否可以实现此目标。

输入

输入第一行为一个正整数 N ，表示包含 N 组测试数据。每组测试数据的第一行包含四个整数 F 、 E 、 A 、 B 。 F ($1 \leq F \leq 50000000$) 表示摩天大楼的楼层总数（即摩天大楼的楼层编号从 0 到 $F-1$ ）， E ($0 \leq E < 100$) 表示电梯的数量， A 和 B ($0 \leq A, B < F$) 是两个楼层的序号，表示需要将家具从楼层 A 搬运到楼层 B 。接着是 E 行数据，每行描述一部电梯，包含两个整数 X 和 Y ($0 < X, 0 \leq Y$)， Y 表示电梯从楼层 Y 出发， X 表示电梯每经过 X 层时停一次。例如，对于 $X=3$ ， $Y=7$ ，电梯将在以下楼层停留：第 7 层、第 10 层、第 13 层、第 16 层……

输出

对于每组测试数据输出一行。如果不借助楼梯，能够从楼层 A 将家具搬运到楼层 B ，输出 `It is possible to move the furniture.`，否则输出 `The furniture cannot be moved.`。

样例输入

```
2
22 4 0 6
3 2
4 7
13 6
10 0
1000 2 500 777
2 0
2 1
```

样例输出

```
It is possible to move the furniture.
The furniture cannot be moved.
```

分析

将电梯视为图的顶点，如果电梯 e_1 所停留的楼层与电梯 e_2 所停留的楼层有重叠，且最低重叠的楼层编号 f 在 0 和 $F-1$ 之间，则 e_1 和 e_2 之间有一条无向边（因为电梯能够在可达楼层间上下移动，因此题目模型对应的是无向图而不是有向图）。将起始楼层 A 和 B 也视为图中的一个顶点，令其编号为 E 和 $E+1$ ，若电梯 e 经停楼层 A ，则电梯 e 和楼层 A 之间具有一条无向边，同理可判断电梯 e 和楼层 B 之间是否具有无向边。建立无向图后，从顶点 E 出发进行 DFS，检查顶点 $E+1$ 是否能够访问即可判定结果。解题的难点在于判定两部电梯是否在楼层 0 到 $F-1$ 之间产生重叠。设电梯 e_1 的参数 $Ye_1=a$ ， $Xe_1=b$ ，电梯 e_2 的参数 $Ye_2=c$ ， $Xe_2=d$ ，则前述判定问题可以转换为确定二元一次不定方程

$$a + bx = c + dy, \quad x \geq 0, \quad y \geq 0$$

是否有解，亦即判定一次同余方程

$$bx \equiv c - a \pmod{d}$$

是否有解的问题。根据一次同余方程的性质，只有当 $\gcd(b, d)$ 能够整除 $c-a$ 时，同余方程才可能有解。若有解，则可以通过扩展欧几里得算法求出一个基本解 x_0 （由于限定 $x \geq 0$ ，若 $x_0 < 0$ ，需要将其调整为非负整数）。

数), 通过 x_0 可以确定一个最低楼层 $L=a+bx_0$ (若 $L < c$, 则需将其调整为不小于 c), 若 L 满足 $0 \leq L < F$ 的条件, 则电梯 e_1 和 e_2 能够互相可达。需要注意边界情形, 例如某部电梯 e 的 $Y \geq F$, 则电梯 e 无法到达楼层 A 和 B , 且电梯 e 和其他电梯间不存在无向边 (尽管与其他电梯可能在大于等于 F 的楼层重叠, 但不符合题意)。

强化练习: 10850 The Gossipy Gossipers Gossip Gossips^D, 11474 Dying Tree^D, 11841* Y-Game^D, 11966 Galactic Bonding^C, 12460 Careful Teacher^D。

扩展练习: 295* Fatman^D, 676* Horse Step Maze^D, 10876* Factory Robot^D, 11967 Hic-Hac-Hoe^E。

9.4.2 最短路径

在图的广度优先遍历过程中, 每次发现一个新的顶点时, 会将当前顶点设置为这个新顶点的前驱 (父) 顶点。所有顶点的前驱信息保存在 *parent* 数组中, 根据该数组, 可以得到从起始顶点到达某个顶点的最短路径 (shortest path), 即边数最少的路径。由于 *parent* 数组保存的是各顶点的前驱, 在构建路径的时候要求是从起点开始到终点的一条路径, 可以通过以下两种方法来构建。

第一种方法: 利用递归将路径反向。在 *parent* 数组中, 存储的是各个顶点的前驱信息, 对于起始结点来说, 其前驱一般设置为一个特殊值 (如-1), 这样在查找顶点的前驱时, 只要其前驱为特殊值则表明该顶点为遍历的起始结点。在递归过程中, 比较当前顶点的序号是否与起始顶点的序号相同, 如果相同则停止递归, 输出顶点的编号, 由于递归所具有的栈性质, 顶点输出的顺序正好构成从遍历起点开始到终点的路径。

```
// 使用递归输出路径。s 表示起始顶点, u 为当前顶点。
void findPath(int s, int u) {
    if (u != s) {
        findPath(s, parent[u]);
        cout << ' ' << u;
    }
    else cout << s;
}
```

第二种方法: 使用栈或模拟栈的功能来重建最短路径。最方便的方法是使用 *vector*。使用 *vector* 来构建路径时, 每次将顶点编号插入到 *vector* 的前端(或者每次将顶点编号附加到最后, 在结束时将 *vector* 反向) 即可得到对应的最短路径。

```
// 使用回溯来输出路径。
void findPath(int s, int u) {
    // 声明一个 vector, 存储路径上顶点的编号。
    vector<int> path;

    // u 为终止顶点的序号, s 为起始顶点的序号。每次将顶点编号插入到路径的最前端。
    // 然后将当前顶点的编号设置为其父顶点的编号, 重复此过程, 直到找到起始顶点。
    while (u != s) {
        path.insert(path.begin(), u);
        u = parent[u];
    }
    // 在退出 while 循环时, u 和 s 相同, 但起始顶点 s 的编号尚未加入。
    path.insert(path.begin(), s);

    // 输出路径。
    for (auto v : path) cout << ' ' << v;
    cout << '\n';
}
```

}

有关求解最短路径的题目，大致可以分为四种类型：(1) 单个源点，单个终点；(2) 单个源点，多个终点；(3) 多个源点，单个终点；(4) 多个源点，多个终点。对于这四种类型，处理思路是类似的。下面以一道例题来介绍具有多个源点和多个终点的图，其最短路径的求解方法。

11101 Mall Mania^C (商场狂)

给定一个 R 行 C 列的城市网格，从东到西由经路划分，从北到南由纬路划分，经路和纬路从 0 开始编号，最大不超过 2000。在城市网格中有两个商场，商场由连续的网格单元构成，给定两个商场的边界描述，要求计算两个商场间的最短距离。最短距离是指在某个商场的边界上，通过经路和纬路到达另外一个商场的需要经过的最少网格单元数量。两个商场所包含的网格单元不会重合，未包含在商场中的网格单元是连续的。此处的“连续”是指两个网格单元具有公共边。商场与商场之间不会发生相交，并且不会将任何空的网格包围起来形成“孤岛”，也就是说，只要某个网格不属于商场，则其必定与其他不属于商场的网格是连接在一起的。

输入

输入包含多组测试数据。每组测试数据均包含了两个商场的描述。每个商场的描述包含一个整数 $p \geq 4$ ，表示商场的周长，接着的一行或多行共包含了 p 个整数对 (a, s) ，按顺时针顺序给出了商场边界上经路与纬路交叉点的坐标。最后一组测试数据后是一行只包含 0 的数据，表示输入结束。

输出

对于每组测试数据，输出一个整数 d ——沿着经路和纬路从某个商场的边界到达另外一个商场的边界需要行走的最短距离。

样例输入

```
4
0 0 0 1 1 1 0
6
4 3 4 2 3 2
2 2 2 3
3 3
0
```

样例输出

```
2
```

分析

朴素的方法是列出两个商场的边界坐标，计算两点之间的曼哈顿距离。但是题目所给的测试数据规模较大，使用此种时间复杂度为 $O(V^2)$ 的方法会超时。如果将某个商场的边界上的方格均设置为起点，另外一个商场边界上的方格设置为终点，则可以使用 BFS 来求解最短距离，这样可以降低时间复杂度，能够在规定时限内获得通过。

强化练习：[429 Word Transformation^A](#), [633 A Chess Knight^D](#), [762 We Ship Cheap^A](#), [1148 The Mysterious X Network^C](#), [10009 All Roads Lead Where^A](#), [10113 Exchange Rates^C](#), [10150 Doublets^B](#), [10422 Knights In FEN^B](#), [10610 Gopher And Hawks^C](#), [10653 Bombs NO They Are Mines^A](#), [10959 The Party Part I^A](#), [10977 Enchanted Forest^C](#), [12160 Unlock the Lock^B](#)。

扩展练习：[589 Pushing Boxes^D](#), [656 Optimal Programs^D](#), [10068 The Treasure Hunt^D](#), [11049 Basic Wall Maze^C](#), [11624 Fire^B](#), [11730 Number Transformation^C](#), [11792 Krochanska is Here^D](#), [12101 Prime Path^D](#),

13295 Carroll's Scrabble^E。

9.4.3 最长简单路径

简单路径 (simple path) 是指两个顶点间的一条路径, 该路径上的所有顶点不重复。在连通图中, 两个顶点间的简单路径可能不止一条。最长路径 (longest path) 在多数情况下没有太大的意义——如果无向图中包含圈, 多次经过该圈, 则路径的长度可以无限大——所以一般讨论两个顶点间的最长简单路径 (longest simple path)。顾名思义, 最长简单路径就是具有最大长度的简单路径。对于一般图来说, 最长简单路径尚无有效算法, 所有已知算法在最坏的情况下均需要指数级别的时间。如果给定的图是无圈图, 则可以使用广度优先遍历进行求解, 只需在更新距离时取更大的距离即可^I。对于有圈图, 最长简单路径可以通过深度优先遍历 (回溯) 求出, 在求解的过程中需要记录路径上包含的顶点以避免重复。下述代码示例了给定起点 s 和终点 t 时, 如何使用深度优先遍历来寻找两个顶点之间的最长简单路径。

```
const int MAXV = 1010;

// 邻接边链表。
list<int> g[MAXV];

// t 为最长简单路径的终点, maxDist 为最长简单路径的长度, visited 标记顶点是否已经被访问。
int t, maxDist = 0, visited[MAXV];

// 深度优先搜索查找最长简单路径, s 为当前顶点。
void dfs(int s, int d) {
    if (s == t) { maxDist = max(maxDist, d); return; }
    visited[s] = 1;
    for (auto v : g[s])
        if (!visited[v])
            dfs(v, d + 1);
    visited[s] = 0;
}
```

强化练习: 539 The Settlers of Catan^A, 685 Least Path Cost^D, 10000 Longest Path^A, 10285 Longest Run on a Snowboard^A, 13038 Directed Forest^D。

扩展练习: 10029* Edit Step Ladders^B, 10051* Tower of Cubes^B, 10259 Hippity Hopscotch^C。

9.4.4 图的着色

在有关图的着色 (coloring) 问题中, 一般是给定一个无向图, 然后要求判断该图是否可二着色 (two-coloring)。利用前述介绍的图遍历可以容易地解决——使用深度 (或广度) 优先遍历着色并检查是否存在染色冲突即可。若在遍历过程中发现已处理顶点的颜色与当前顶点的颜色相同, 则表明无法按要求着色。需要注意, 给定的无向图有可能是由多个子图构成, 在判断时, 需要对每个子图使用染色过程进行判断其是否可二着色, 如果所有子图是可二着色的, 整个图才能称为可二着色的。关于图的顶点染色, 还存在以下结论:

^I 或者先对有向无圈图进行拓扑排序, 然后根据顶点的拓扑顺序使用动态规划, 更新与起点的最大距离。抑或将有向边的边权置为-1, 使用本书第10章介绍的Bellman-Ford算法求最短路径, 则最后得到的最短路径对应着原图的最长简单路径。

(1) 假设图 G 是简单图, 其中最大顶点度为 d_{max} , 则图 G 是可 $(d_{max}+1)$ 着色的。

(2) 假设图 G 是简单连通非完全图, 其最大顶点度为 d_{max} (≥ 3), 则图 G 是可 d_{max} 着色的。

强化练习: [10004 Bicoloring^A](#), [10505* Montesco vs Capuleto^B](#), [11080 Place the Guards^B](#), [11396 Claw Decomposition^B](#)。

扩展练习: [1613 K-Graph Oddity^E](#)。

9.4.5 最近公共祖先

给定一棵有根树 T 以及树中的两个结点 u 和 v , 它们的最近公共祖先 (或称最小公共祖先, Lowest Common Ancestor 或 Least Common Ancestor, LCA) 定义为结点 u 和 v 的公共祖先结点中具有最大深度的结点 w 。从定义不难推知, 最近公共祖先唯一。如图 9-3 所示, 虽然结点 a 和结点 c 都是结点 d 和 e 的公共祖先, 但 c 才是 d 和 e 的最近公共祖先而 a 不是, 因为 c 的深度要比 a 的深度大。

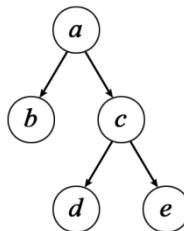


图 9-3 一棵包含 5 个结点的二叉树

基本算法

通过一次深度 (或广度) 优先图遍历, 能够确定遍历过程中各个结点的父结点, 将这些信息记录在相应数据结构中, 就可以通过此数据结构“逆向”查找, 从而得到一条从指定结点到根结点的路径, 而两个不同结点的最近公共祖先就是沿着各自的路径向根结点前进时第一次产生交汇时所处的结点。

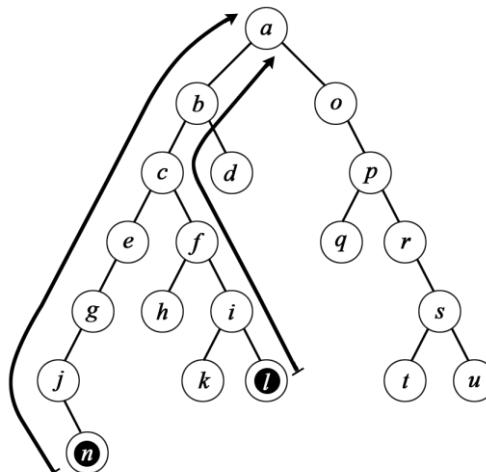


图 9-4 利用父结点信息确定 n 和 l 的最近公共祖先。 n 到根结点 a 的路径为 $\langle n, j, g, e, c, b, a \rangle$, l 到根结点 a 的路径为 $\langle l, i, f, c, b, a \rangle$, 两条路径最初于 c 发生交汇, 故 c 是 n 和 l 的最近公共祖先

根据以上思路，在求结点对(u, v)的最近公共祖先时，可以先确定结点 u 的各个祖先结点，将其放在一个集合中，不妨令其为 S_u ，然后再沿着从 v 到根结点的路径，逐次查找结点 v 以及 v 的祖先结点是否在 u 的祖先结点集合 S_u 中出现，其中最早出现的结点即为两者的最近公共祖先。

```

//-----9.4.5.1.cpp-----
const int MAXV = 10010;

// 以邻接表形式表示树。
vector<int> g[MAXV];
// parent 表示各个顶点的父顶点，根结点的父顶点为-1;
// visited 记录结点是否已被访问。
int parent[MAXV], visited[MAXV];

// 通过深度优先遍历确定各顶点的父结点。
void dfs(int u) {
    visited[u] = 1;
    for (auto v : g[u])
        if (!visited[v]) {
            parent[v] = u;
            dfs(v);
        }
}

// 通过确定两条路径的交点来确定最近公共祖先。
int getLCA(int u, int v) {
    unordered_set<int> Su;
    while (u != -1) {
        Su.insert(u);
        u = parent[u];
    }
    while (v != -1) {
        if (Su.find(v) != Su.end()) return v;
        v = parent[v];
    }
}

int main(int argc, char *argv[]) {
    // n 为顶点数量，q 为查询的数量。
    int n, q;
    while (cin >> n) {
        // 读入树。
        for (int i = 0; i < n; i++) g[i].clear();
        // n 个结点的树共有 n-1 条无向边。
        for (int i = 1, u, v; i < n; i++) {
            cin >> u >> v;
            g[u].push_back(v);
            g[v].push_back(u);
        }
        // 从根结点开始进行 DFS，确定各个顶点的父顶点。
        memset(parent, -1, sizeof(parent));
        memset(visited, 0, sizeof(visited));
        dfs(0);
        // 读入结点对 (u, v)，然后查询其最近公共祖先。
        cin >> q;
    }
}

```

```

for (int i = 0, u, v; i < n; i++) {
    cin >> u >> v;
    cout << "LCA of " << u << " and " << v << " is ";
    cout << getLCA(u, v) << '\n';
}
}
//-----9.4.5.1.cpp-----//

```

由于上述算法需要反复查询结点是否在集合中，只适用于树规模不是很大的最近公共祖先查询。在具体实现时，一般需要借助类似于标准库中的 `map` 或 `set` 数据结构来提高查询效率。如果结点数量为 $|V|$ ，查询数量为 Q ，上述算法的时间复杂度为 $O(QV\log V)$ 。

在前述算法中，需要反复查询才能确定两个结点的最近公共祖先，效率较低，可以对其进行适当改进，使得效率更高。改进的方法很简单，同样也是利用深度优先遍历得到的父结点数组，只不过在深度优先遍历的同时记录结点的深度。给定结点对 (u, v) ，如果两者的深度不同，先将深度较大的结点沿着从结点到根的路径移动，直到两个结点的深度相同，然后检查两个结点是否相同，如果相同说明最近公共祖先已经找到。若不相同，接下来两个结点每次都向上移动一个结点，一直到两个结点为同一个结点为止，此时的结点即为 u 和 v 的最近公共祖先。

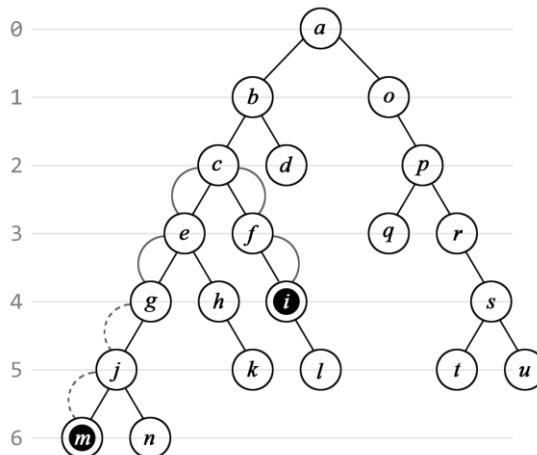


图 9-5 利用父结点和深度信息确定 m 和 i 的最近公共祖先。初始时 m 先向上移动两个结点到达 g ，此时 g 和 i 的深度相同，然后同时向上移动，到达 c ，此时为同一个结点，故 c 为 m 和 i 的最近公共祖先

第二种算法的原理非常简单，与第一种算法的差别在于利用了结点的深度，从而避免了反复查询，其时间复杂度为 $O(QV)$ 。算法效率有了一定提升，但仍然不够理想，之所以在此予以介绍是因为此方法是后续介绍的倍增算法的基础。

```

//-----9.4.5.2.cpp-----//
const int MAXV = 10010;

vector<int> g[MAXV];
int parent[MAXV], depth[MAXV], visited[MAXV];

// 通过深度优先遍历确定父结点和深度。
void dfs(int u) {
    visited[u] = 1;
    for (int v : g[u])
        if (!visited[v])
            parent[v] = u, depth[v] = depth[u] + 1, dfs(v);
}

```

```

for (auto v : g[u])
    if (!visited[v]) {
        parent[v] = u;
        depth[v] = depth[u] + 1;
        dfs(v);
    }
}

// 根据深度来确定最近公共祖先。
int getLCA(int u, int v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    while (diff--) u = parent[u];
    if (u != v)
        while (u != v) u = parent[u], v = parent[v];
    return u;
}
//-----9.4.5.2.cpp-----//

```

离线算法

给定有根树 T ，如果已经知道了所有需要查询最近公共祖先的结点对，那么可以应用更为高效的离线算法。下面介绍离线算法中巧妙而简洁的 Tarjan 算法¹，该算法由 Robert Tarjan 根据深度优先遍历的特点拓展而来。

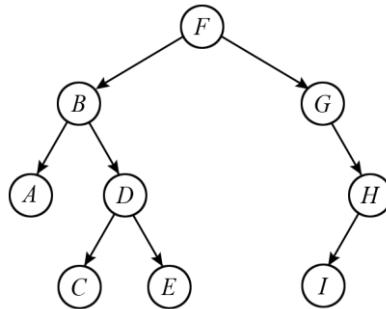


图 9-6 一棵包含 9 个结点的二叉树

为了更好地理解算法，让我们首先来了解一下最近公共祖先的一些特点。如图 9-6 所示，对于该二叉树中任意给定的结点对 (u, v) ，它们的相互关系及最近公共祖先必定是下列情况中的一种：

(1) v 是 u 的子结点（或子孙结点），很显然， u 和 v 的最近公共祖先就是 u 。例如结点对 (F, B) ，两者的最近公共祖先为结点 F 。

(2) u 是 v 的子结点（或子孙结点），很显然， u 和 v 的最近公共祖先就是 v 。例如结点对 (G, F) ，两者的最近公共祖先为结点 F 。

(3) u 是结点 w 的子结点（或子孙结点）， v 是结点 w 的子结点（或子孙结点），则 u 和 v 的最近公共

¹ Robert Endre Tarjan (1948—)，美国计算机科学家、数学家。本章包含多个以 Tarjan 命名的算法，包括离线最近公共祖先 Tarjan 算法，强连通分支 Tarjan 算法，以及求割顶和桥的 Tarjan 算法，它们均为 Tarjan 所发现。

祖先为 w (或 w 的某个子孙结点)。例如结点对 (B, G) 、 (B, I) 、 (C, G) ，它们的最近公共祖先均为结点 F 。

根据以上分析可以得出一个简单而重要的结论：给定树中的结点 u ， u 和 u 的子树结点的最近公共祖先为 u ， u 的不同子树结点之间（属于不同子树的两个结点）的最近公共祖先也为 u 。回顾深度优先遍历，其遍历过程有如下特点：在遍历完结点 u 的所有子树结点后才会对 u 本身及其父结点进行遍历。Tarjan 算法的巧妙之处在于充分利用了上述结论和深度优先遍历的特点。以下是 Tarjan 离线最近公共祖先算法的伪代码表示，在伪代码中，使用了并查集，其中的 $ancestor[i]$ 表示结点 i 的祖先（在此处，结点 i 的祖先可以为结点本身）。

```

int dfs(int u) {
    // 将 u 所在集合的代表的祖先设置为 u 本身。
    ancestor[find_set(u)] = u
    // 遍历 u 的每个子结点 v。
    for each child v of u in T
        do dfs(v)
        // 将 u 和 v 合并到一个集合中。
        union_set(u, v)
        // 将合并后集合的代表的祖先设置为 u。
        ancestor[find_set(u)] = u
    // 当 u 的所有子结点着色后才标记 u 为已着色。
    colored[u] = true
    // 对与 u 有关联的结点对 (u, v) 进行最近公共祖先查询。
    for each node v such that [u, v] in P
        // 只有当 v 也是已着色状态时查询才能得到正确的结果。
        do if colored[v] = true
            // v 所在集合的代表的祖先即为 u 和 v 的最近公共祖先。
            then lca(u, v) = ancestor[find_set(v)]
}
}

```

Tarjan 算法按照深度优先遍历过程，从一个指定结点 u 开始，逐个访问 u 的子结点 v ，递归求解结点对的最近公共祖先。在伪代码中，使用了并查集和 $ancestor$ 数组，它们是起着怎样的作用呢？由前述讨论可知，对于结点 u 和 u 的任意一个子树结点 v 来说， u 和 v 的最近公共祖先是 u 。进一步地，对于 u 的任意一个子树结点 v ，以下两者相同：(1) v 与树中除了以 u 为根的子树以外的其他的任意结点 x 的最近公共祖先；(2) u 与树中除了以 u 为根的子树以外的其他的任意结点 x 的最近公共祖先。也就是说，可以将 u 和以 u 为根的子树看成一个结点的集合，并且将 u 作为这个集合的一个代表，这个结点集合与集合之外的树中的其他结点的最近公共祖先，等同于 u 与结点集合之外的其他结点的最近公共祖先。由于树具有“自相似”的内部结构，实际上可以将树看成很多个类似集合的组合。使用集合的代表进行最近公共祖先的求解，对结果的正确性并无影响而且更为方便。考虑到并查集的特性，在此种情况下应用并查集来表示结点的集合非常合适。

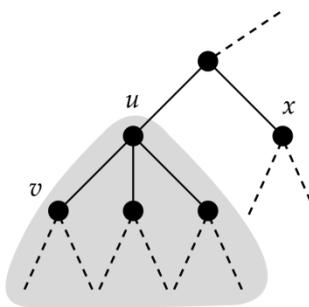
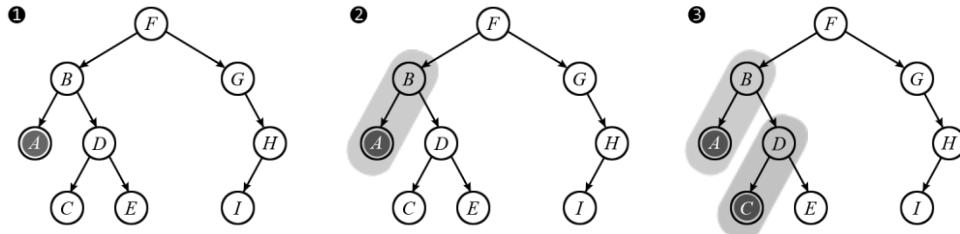
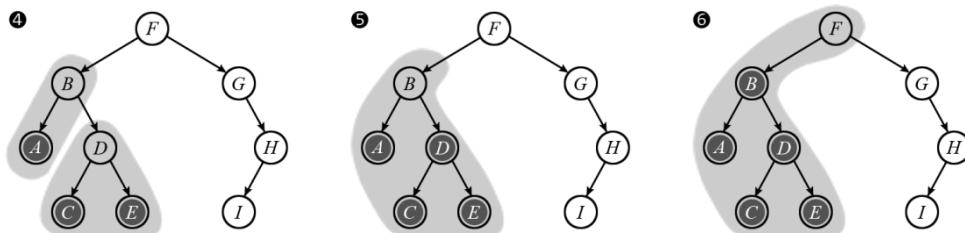


图9-7 对于 u 以及 u 的子树结点构成的灰色区域来说, 该区域内的任意一个结点 v 与灰色区域外任意一个结点 x 的最近公共祖先等价于 u 与 x 的最近公共祖先, 也就是说 u 可以作为灰色区域结点集合的“代表”

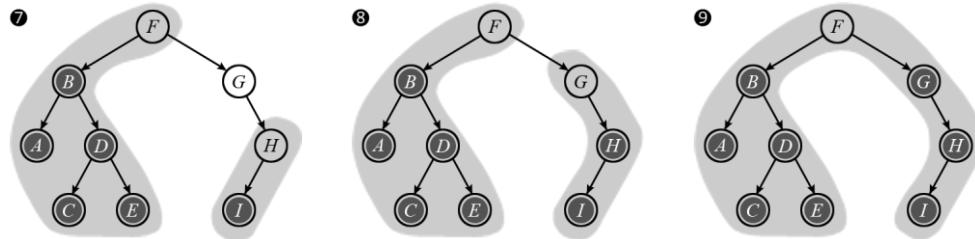
假设使用 Tarjan 算法求解如图 9-6 所示二叉树中结点对的最近公共祖先, 我们跟随算法查看每一步的执行情况来尝试理解算法的正确性。



Tarjan 算法的第 1 步到第 3 步。第 1 步: 沿着二叉树到达叶结点 A, 此时叶结点 A 自身构成一个集合, 由于结点 A 已无子结点, 不必继续递归向下对子结点进行操作, 直接将其标记为已着色。此时除了结点 A 已经被着色, 其他结点均尚未被着色, 所以还不能进行查询。第 2 步: 返回上一层结点 B, 执行将结点 B 和结点 A 合并的操作, 这样结点 A 和结点 B 在同一个集合中, 这个集合的祖先被设置为 B。第 3 步: 访问叶结点 C, 并标记结点 C 为已访问, 此时可查询结点对 (C, A) 的最近公共祖先, 可知结点对 (C, A) 的最近公共祖先为 A 所在集合的祖先, 即 B。与第 2 步类似, 将结点 C 和结点 D 合并成一个集合, 并设置这个集合的祖先为 D



Tarjan 算法的第 4 到第 6 步。第 4 步: 访问 D 的另外一个子结点 E, 由于 E 并无子结点, 在遍历 E 后会将结点 E 标记为已着色, 此时可查询结点对 (E, A), (E, C) 的最近公共祖先, 接着将结点 C, D, E 合并为一个集合, 集合的祖先设置为 D。第 5 步: 由于 D 的两个子结点均已着色, 将结点 D 标记为已着色, 此时可查询结点对 (D, A), (D, C), (D, E) 的最近公共祖先, 接着将结点 B 和 B 的子树结点合并, 集合的祖先设置为 B。第 6 步: 将结点 B 设置为已着色, 此时可查询结点 B 和任意已经着色的结点的最近公共祖先, 接着将结点 F 与 F 的子树结点合并, 并将集合的祖先设置为 F



Tarjan 算法的第 7 到第 9 步。第 7 步：访问结点 I ，并将 I 标记为已着色，此时可以查询 I 与其他已着色的结点之间的最近公共祖先，接着将 I 与 H 合并成一个集合并将集合的祖先设置为 H 。第 8 步：由于结点 H 的子结点 I 已访问，将 H 标记为已着色，此时可以查询 H 与其他已着色的结点之间的最近公共祖先，接着将 G 与 G 的子树合并成一个集合，并将集合的祖先设置为 G 。第 9 步：将结点 G 标记为已着色，此时可以查询结点 G 与其他已着色结点之间的最近公共祖先。最后将 F 的所有子树结点合并成一个集合，并将集合的祖先设置为 F ，由于 F 的所有子树结点均已访问，将 F 标记为已着色，此时可以查询结点 F 与其他已着色结点间的最近公共祖先。

图 9-8 Tarjan 算法在图 9-6 所示二叉树上的具体执行过程

下面我们接着来看，随着深度优先遍历的进行，子树的代表不断发生变化，算法是如何保证始终能够正确地计算结点对的最近公共祖先。当某个结点 u 的所有子树结点均访问完毕时，查询与另外一个已着色结点 v 之间的最近公共祖先，有两种情形。第一种情形： v 是 u 的子树结点，那么很显然， u 和 v 的最近公共祖先就是 u ，由于此时已经将 u 和 u 的子孙结点合并成一个集合，并且将该集合的祖先设置为 u ，因此查询 v 所在集合的祖先可以得到正确的结果。此种情形如图 9-9 所示。

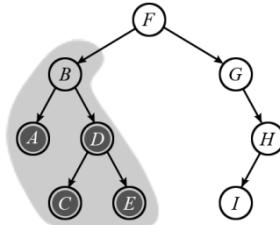


图 9-9 v 是 u 的子树结点时的情形。假设 u 为结点 D ， v 为结点 E ，由于结点 D 已经着色，根据算法的流程，查询 D 与 E 的最近公共祖先，可知其为 E 所在集合的祖先，即 D

第二种情形： v 不是 u 的子树结点，那么根据算法的流程可知 v 必定是某个结点 w 的已访问子树中的着色结点，而且 v 已经与 w 合并成一个集合，该集合的祖先被设置为 w 。根据深度优先遍历的特点，此时已着色的结点 u 所在的子树必定是 w 的子树，也就是说 u 必定是 w 的一个子孙结点，由于 w 的不同子孙结点的最近公共祖先就是 w ，因此得到 u 和 v 的公共祖先就是 v 所在集合的祖先 w 。此种情形如图 9-10 所示。

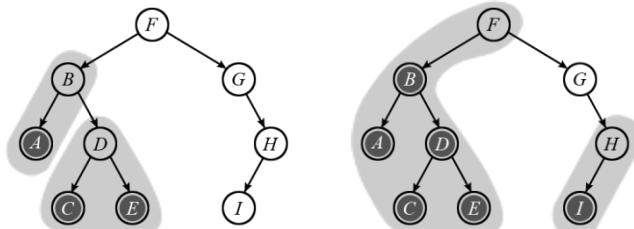


图 9-10 v 不是 u 的子树结点时的情形。对于左侧所示的图形，假设 u 为结点 E , v 为结点 C , 由于 E 已经着色，根据算法的流程，查询 E 与 C 的最近公共祖先，可知其为 C 所在集合的祖先，即 D 。对于右侧所示的图形，假设 u 为结点 I , v 为结点 C , 由于 I 已经着色，根据算法的流程，查询 I 与 C 的最近公共祖先，可知其为 C 所在集合的祖先，即 F

在算法的具体实现时需要注意以下几点：

(1) 给定的查询中，可能有的结点对中的结点先后顺序并不是按照算法中结点的先后访问顺序给出，例如给定结点对 (A, D) ，在算法实际执行过程中，在 A 着色时， D 尚未被着色，此时无法进行最近公共祖先的查询，因而也无法得到结果，应当在 D 着色后再进行查询，因为此时 A 已经被着色。所以在读取需要查询的结点对时，不仅需要读取 (A, D) 结点对，还应读取其顺序对调后的结点对 (D, A) ，因为事先并不知道两个结点的先后访问顺序，如果结点对 (A, D) 无法被查询，则应该查询结点对 (D, A) ，这样能够保证得到该结点对的最近公共祖先。

(2) 如果使用邻接表的方式存储边，当结点数量较多时，可能无法使用限定的内存表示，此时可以选用更为高效的链式前向星来存储边。

(3) 在给定有根树的过程中，结点关系可能重复给出，导致邻接链表中可能包含重复的边，因此在深度优先遍历过程中需要标记结点的访问状态，以避免重复访问而进入无限循环。

Tarjan 算法相较基本算法在效率上有很大提升，其时间复杂度为 $O(V+Q)$ ， $|V|$ 为树中结点的数量， Q 为查询的数量，可以认为是线性时间复杂度的算法。

```
//-----9.4.5.3.cpp-----//
const int MAXV = 50010, MAXE = 100010;

struct edge {
    int id, u, v, next;
};

edge data[MAXE], query[MAXE];
int idx, headD[MAXV], headQ[MAXV];
int numberofVertices, numberofQueries;
int parent[MAXV], ranks[MAXV], ancestor[MAXV], visited[MAXV];
int colored[MAXV], lca[MAXV];

// 并查集的实现从略，请读者参阅本书第 2 章“数据结构”中的相关内容。

// Tarjan 离线算法求最近公共祖先。
void dfs(int u) {
    ancestor[findSet(u)] = u;
    visited[u] = 1;
    for (int i = headD[u]; ~i; i = data[i].next)
        if (!visited[data[i].v])
            dfs(data[i].v);
            unionSet(u, data[i].v);
            ancestor[findSet(u)] = u;
    }
    colored[u] = 1;
    for (int i = headQ[u]; ~i; i = query[i].next)
        if (colored[query[i].v])
            lca[query[i].id] = ancestor[findSet(query[i].v)];
}


```

```

int main(int argc, char *argv[]) {
    int u, v;
    while (cin >> numberOfVertices) {
        idx = 0;
        memset(headD, -1, sizeof(headD));
        // 读入边，以链式前向星方式存储。对于树来说，边的数量为顶点的数量减 1。
        for (int i = 0; i < numberOfVertices - 1; i++) {
            cin >> u >> v;
            data[idx] = (edge){idx, u, v, headD[u]};
            headD[u] = idx++;
            data[idx] = (edge){idx, v, u, headD[v]};
            headD[v] = idx++;
        }
        // 将查询也视为边，以链式前向星方式存储。
        idx = 0;
        memset(headQ, -1, sizeof(headQ));
        cin >> numberOfQueries;
        for (int i = 0; i < numberOfQueries; i++) {
            cin >> u >> v;
            query[idx] = (edge){i, u, v, headQ[u]};
            headQ[u] = idx++;
            query[idx] = (edge){i, v, u, headQ[v]};
            headQ[v] = idx++;
        }
    }
    // 数据结构初始化。
    memset(visited, 0, sizeof(visited));
    memset(colored, 0, sizeof(colored));
    makeSet();
    // 因为顶点编号从 1 开始，故默认从第 1 个顶点开始进行遍历。
    dfs(1);
    // 输出顶点对的最近公共祖先。
    for (int i = 0; i < numberOfQueries; i++)
        cout << lca[i] << '\n';
    return 0;
}
//-----9.4.5.3.cpp-----/

```

在线算法

在线算法有多种，此处介绍使用倍增算法（doubly algorithm）来求解最近公共祖先问题。回顾基本算法中的第二种算法，该算法先将结点调整为相同深度后，然后同时向上移动来寻找最近公共祖先，由于每次只向上移动一个结点，其效率较低。如果能够在向上移动时跳过某些明显不可能是最近公共祖先的结点，则求解速度可以显著提高。倍增算法的基本思想即缘此而来。算法在每次向上移动结点时，总是以 2 的幂次数量进行移动，可以形象地将每次移动看成一次“跳跃”。假设总的结点数为 $|V|$ ，则取最大跳跃幂次为 $i = \lceil \log |V| \rceil$ （运算“ $\lceil f \rceil$ ”表示对实数 f 进行向上取整），即从叶结点向上跳跃 2^i 个祖先结点一定可以到达根结点。将 u 和 v 调整为相同深度，逐次检查在跳过 $2^i, 2^{i-1}, \dots, 2^1, 2^0$ 个祖先结点后， u 和 v 是否相同，如果 u 和 v 不同，说明跳过的那些祖先结点不可能是 u 和 v 的最近公共祖先，可以“安全”地忽略这些结点；如果 u 和 v 相同说明最近公共祖先的深度不会比当前所在结点的深度更小，此时应当减少跳过的结点数量，

再进行检查，直到最后 u 和 v 的父结点相同，那么 u 和 v 的最近公共祖先就是两者的父结点。

那么如何确定某个结点跳跃 2^i 个结点后到达的是哪个结点呢？可以通过深度优先遍历过程中确定的各结点父结点以及深度信息，预先构建一个数据结构来明确。该数据结构中包含了如下的信息：结点 u 的第 2^i 个祖先是哪个结点。根据 2 的幂次特性： $2^i = 2^{i-1} + 2^{i-1}$ ，结点 u 的第 2^i 个祖先结点等同于结点 u 的第 2^{i-1} 个祖先结点的第 2^{i-1} 个祖先结点。这也是为什么选择以 2 的幂次进行跳跃的原因，如果选择 3 的幂次，不存在上述特性。

```
//+++++9.4.5.4.cpp+++++/
// 预处理，根据父结点和深度信息构建结点指定幂次的祖先结点。
void getReady() {
    // u 的第  $2^d$  个父结点即为 u 的第  $2^{d-1}$  个父结点的第  $2^{d-1}$  个父结点，因为  $2^d = 2^{d-1} + 2^{d-1}$ 。
    for (int d = 0; (1 << d) <= numberofVertices; d++)
        for (int u = 0; u < numberofVertices; u++)
            if (ancestor[u][d - 1] != -1)
                ancestor[u][d] = ancestor[ancestor[u][d - 1]][d - 1];
}
```

在调整两个结点的深度时，同样也可以使用倍增方式——将深度差转换为一个二进制数 B ，如果二进制数 B 中位于第 d 个二进制位的数位为 1，则将深度较大的结点往上跳跃 2^d 个结点。

```
// 使用倍增方式调整两个结点的深度使之相同。
void adjustDepth(int &u, int &v) {
    if (depth[u] < depth[v]) swap(u, v);
    int diff = depth[u] - depth[v];
    for (int d = 0; (1 << d) <= diff; d++)
        if ((1 << d) & diff)
            u = ancestor[u][d];
}
```

例如，假设两个结点的深度差为 25，由于 25 对应的二进制数为 11001_2 ，在进行深度调整时，将深度较大的结点依次向上移动 2^0 ， 2^3 ， 2^4 个结点，这样两个结点的深度即可相同。

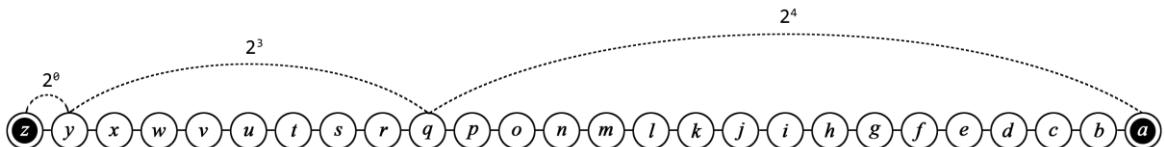


图 9-11 使用倍增方式调整两个深度差为 25 的结点使之相同的过程，结点 z 深度为 25，结点 a 深度为 0

```
const int MAXV = 100010, MAXD = 20, MAXE = 200010;

struct edge {
    int id, u, v, weight, next;
};

edge data[MAXE];
int idx, headD[MAXV];
int numberofVertices, numberofQueries;
int ancestor[MAXV][MAXD], depth[MAXV], visited[MAXV];

// 深度优先遍历确定父结点和深度。
```

```

void dfs(int u) {
    visited[u] = 1;
    for (int i = headD[u]; i != -1; i = data[i].next)
        if (!visited[data[i].v]) {
            ancestor[data[i].v][0] = u;
            depth[data[i].v] = depth[u] + 1;
            dfs(data[i].v);
        }
}

int getLCA(int u, int v) {
    adjustDepth(u, v);
    // 比较两个结点是否相同, 若不相同则每次跳跃  $2^d$  个结点。到算法最后, 只需向上跳跃 1 个结点。
    // 后两者相同, 则 u 和 v 的最近公共祖先即为 u 的父结点。
    if (u != v) {
        int maxd = log2(numberOfVertices);
        for (int d = maxd; d >= 0; d--)
            if (ancestor[u][d] != -1 && ancestor[u][d] != ancestor[v][d])
                u = ancestor[u][d], v = ancestor[v][d];
        u = ancestor[u][0];
    }
    return u;
}

int main(int argc, char *argv[]) {
    int u, v, weight;
    while (cin >> numberOfVertices, numberOfVertices) {
        idx = 0;
        memset(headD, -1, sizeof(headD));
        for (int from = 1; from <= numberOfVertices - 1; from++) {
            cin >> v >> weight;
            data[idx] = (edge){idx, from, v, weight, headD[from]};
            headD[from] = idx++;
            data[idx] = (edge){idx, v, from, weight, headD[v]};
            headD[v] = idx++;
        }

        memset(ancestor, -1, sizeof(ancestor));
        memset(visited, 0, sizeof(visited));
        memset(depth, 0, sizeof(depth));
        memset(dist, 0, sizeof(dist));

        dfs(0);
        getReady();

        cin >> numberOfQueries;
        for (int i = 0; i < numberOfQueries; i++) {
            cin >> u >> v;
            cout << getLCA(u, v) << '\n';
        }
    }
}

return 0;
}
//++++++++++++++9.4.5.4.cpp+++++++

```

倍增算法预处理过程的时间复杂度为 $O(|V|\log|V|)$, 查询过程的时间复杂度为 $O(Q\log|V|)$, $|V|$ 为树

中结点的数量, Q 为查询的数量。

强化练习: [10938 Flea Circus^C](#), [12238 Ants Colony^D](#)。

9.4.6 割顶

在无向连通图中, 如果移除某个顶点以及与其相连的边后, 导致图不再连通, 则称这个顶点为割顶 [或称割点, *cut vertex*, 又称关节点, *articulation point*]。给定一个图, 其中可能存在多个割顶。移除割顶会使图分割为两个或更多的不连通子图。在现实世界中, 国际互联网结构中的枢纽结点即是割顶, 一旦枢纽结点发生故障(例如断电或遭受袭击), 会导致依赖于枢纽结点进行消息中转的其他网络结点无法再相互联系。识别图中的割顶对于设计高可靠性的网络具有参考作用, 因为割顶的重要性, 可以采取备份措施保证整个网络的连通性。

在图中寻找割顶, 朴素的方法是逐个移除顶点以及与其关联的边, 然后检查图是否连通, 若图不再连通, 则该顶点为割顶。检查图是否连通可以使用图遍历算法对图进行一次遍历, 若所有顶点均被访问, 则表明图仍是连通的, 则移除的顶点不是割顶, 否则是割顶。具体实现时, 并不需要真正移除每个顶点及其所关联的边, 只需要在遍历过程中搜索到该顶点时跳过即可。朴素算法的时间复杂度为 $O(|V|(|V| + |E|))$ 。

```
-----9.4.6.1.cpp-----
// 使用邻接表方式表示图。每个顶点的出边使用一个集合来表示。
vector<set<int>> g;

// 标记顶点是否已访问的向量。
vector<bool> visited;

// 深度优先遍历。v 为移除的顶点, 在遍历过程中需要跳过。
void dfs(int u, int v) {
    for (auto w : g[u])
        if (w != v && !visited[w]) {
            visited[w] = true;
            dfs(w, v);
        }
}

// 获得无向连通图中割顶的个数。
int getCutVertices() {
    // 设置图的顶点数量。顶点序号从 1 开始。
    int n = g.size();
    // 当图的顶点数不大于 1 时, 割顶数为 0。
    if (n <= 1) return 0;
    // 设置访问标记向量的大小。
    visited.resize(g.size() + 1);
    // 逐个顶点判断是否为割顶。
    int count = 0;
    for (int v = 1; v <= n; v++) {
        fill(visited.begin(), visited.end(), false);
        // 顶点数为 1 的特殊情况已经处理, 此处从任意一个剩余的顶点开始遍历。
        int u = (v - 1 > 0 ? v - 1 : v + 1);
        // 设置移除的顶点和起始顶点为已访问状态, 后续检查连通时不考虑这两个顶点。
        visited[v] = visited[u] = true;
        // 深度优先遍历。
        dfs(u, v);
        // 检查图是否连通, 如果不再连通, 则该顶点是割顶。
    }
}
```

```

        for (int j = 1; j <= n; j++) {
            if (!visited[j]) {
                count++;
                break;
            }
        }
        // 返回割顶的数量。
        return count;
    }
//-----9.4.6.1.cpp-----

```

强化练习：[11902 Dominator^B](#)。

Tarjan 算法可以在 $O(|V| + |E|)$ 的时间内找到给定连通图中的所有割顶。回忆深度优先遍历，整个过程会生成数棵深度优先树组成的深度优先森林，其中的边要么为树边，要么为反向边。对于无向连通图来说，深度优先遍历只会生成一棵深度优先树。顶点 u 是割顶当且仅当满足下述两个条件中的任意一个：

- (1) 顶点 u 是生成树的根结点，且 u 至少有两个子结点。
- (2) 顶点 u 不是生成树的根结点，且顶点 u 拥有至少一个子结点 v ，以 v 为根的子树中不存在能够连接到顶点 u 的祖先结点的反向边。

第一个条件容易理解，如果根结点 u 只有一个子结点，移除根结点 u 后，图仍然是连通的，若根结点有两个子结点，移除根结点 u 后会导致其子结点不再连通。第二个条件不太容易理解，在此做进一步解释。如果顶点 u 不包含子结点，则顶点 u 为生成树的叶结点，移除顶点 u 不会导致图不连通，故顶点 u 不是割顶；若顶点 u 拥有子结点 v ，且以 v 为根的生成子树中有反向边连接到顶点 u 的祖先，那么删除顶点 u ， u 的子结点可以通过此反向边与 u 的祖先连通，因此不会改变图的连通性，故顶点 u 也不会是割顶。如果顶点 u 同时满足拥有子结点且子结点中不存在反向边连接到 u 的祖先的条件，则 u 必定是割顶。通过图 9-12 所示的无向连通图的深度优先搜索树可以更直观地理解这一点。

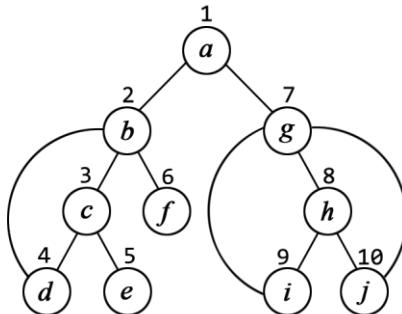


图 9-12 无向连通图的深度优先搜索树。深度优先遍历顺序为： $a, b, c, d, e, f, g, h, i, j$ ，图中数字表示各个顶点的深度优先数。从图中可知，结点 a 为深度优先搜索树的根，且 a 有两个子结点，故 a 是割顶。

结点 c 有两个子结点，其中一个子结点 d 拥有能够连接到 c 的祖先结点 b 的反向边，另外一个子结点 e 不存在反向边能够连接到 c 的任意祖先结点，将 c 移除，将使得原图不再连通，故 c 是割顶。结点 h 有两个子结点，两个子结点均有反向边能够连接到 h 的祖先结点 g ，将 h 删除，不影响图的连通性，故 h 不是割顶。

第一个条件很容易用代码实现，只需在深度遍历过程中记录顶点的子结点数量，并判断是否为根结点即可。第二个条件的判断需要更巧妙的方法——在深度优先遍历的过程中，通过 dfn 数组记录顶点深度优先数，利用该深度优先数来判断第二个条件是否满足。对图 G 中的每个顶点 u 定义一个 low 值， $low[u]$ 表示从 u 或

u 的子孙结点出发通过反向边可以达到的最小深度优先数, 即最早发现时间。 $low[u]$ 取以下三项中的最小值: 顶点 u 本身的深度优先数, u 的子孙结点中所具有的最小深度优先数, 顶点 u 通过反向边可以达到的最小深度优先数, 即

$$low[u] = \min \begin{cases} dfn[u] \\ \min\{low[v] \mid v \text{是} u \text{的子孙结点, } (u, v) \text{是树边}\} \\ \min\{dfn[v] \mid v \text{与} u \text{邻接, 且} (u, v) \text{为反向边}\} \end{cases}$$

则第二个条件对应的判断条件是: u 有一个子结点 v , 满足 $low[v] \geq dfn[u]$ ^I。这里需要注意的是, 当顶点 v 是顶点 u 的邻接顶点且 v 已经访问时, 表明无向边 (u, v) 是一条反向边, 此时顶点 u 的 low 值应该取 $\min\{low[u], dfn[v]\}$ 而不是 $\min\{low[u], low[v]\}$ 。为什么呢? 因为 $low[u]$ 表示的是顶点 u 通过邻接顶点或子孙结点的反向边所能达到的最小深度优先数, 这里的反向边只能“反向”一次 (对应的是 $dfn[v]$), 不能从反向边到达的顶点继续“反向” (对应的是 $low[v]$), 否则 low 值将失去对算法应有的意义, 从而导致错误的结果。

以下是使用深度优先遍历查找割顶的 Tarjan 算法实现, 在此实现中, 使用保存深度优先数的数组来保存各顶点的最小深度优先数。

```
-----9.4.6.2.cpp-----
// 顶点数量。
const int MAXV = 10010, MAXE = 100010;

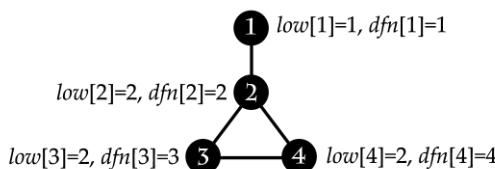
// 使用链式前向星表示图。
struct edge { int v, nxt; } g[MAXE];

// cnt 记录边的数目, head 记录顶点的第一条边在链式前向星中的序号。
int cnt, head[MAXV];

// dfn 保存各顶点的深度优先数, 如果顶点的 dfn 值为 0, 表明该顶点尚未被访问。
// ic 记录各顶点是否为割顶, 为 1 表示为割顶, 为 0 表示不是割顶。
// dfstime 为时间戳。
int dfn[MAXV], ic[MAXV], dfstime = 0;

// Tarjan 算法。调用时从任意一个顶点开始, parent 设置为 -1, 表示起点为根结点。
int dfs(int u, int parent) {
    int lowu = dfn[u] = ++dfstime, lowv, children = 0;
    // 对未访问的非根结点进行深度优先遍历并比较 low 值以判断是否为割顶。
```

^I 需要注意, 此处不能将条件 $low[v] \geq dfn[u]$ 修改为 $low[v] > dfn[u]$ 。如下图所示:



按照结点 1、2、3、4 的顺序进行 DFS, 在完成 DFS 后, 结点 3 和结点 4 的 low 值均等于结点 2 的 dfn 值, 如果按照条件 $low[v] \geq dfn[u]$ 进行判定, 结点 2 是割顶, 若按条件 $low[v] > dfn[u]$ 进行判定, 则结点 2 不是割顶, 但实际上结点 2 是割顶。

```

for (int i = head[u]; ~i; i = g[i].nxt) {
    int v = g[i].v;
    if (!dfn[v]) {
        ++children, lowu = min(lowu, lowv = dfs(v, u));
        if (lowv >= dfn[u]) ic[u] = 1;
    }
    // 此处是按照 Tarjan 算法论文中的伪代码进行实现, 实际上可以略去条件判断, 即
    // else lowu = min(lowu, dfn[v]);
    else if (dfn[v] < dfn[u] && v != parent) lowu = min(lowu, dfn[v]);
}
// 如果为树的根结点且子孙结点数量少于两个, 不是割顶。
if (parent < 0 && children == 1) ic[u] = 0;
// 返回顶点 u 的最小深度优先数。
return lowu;
}
//-----9.4.6.2.cpp-----

```

强化练习: [315 Network^A](#), [10199 Tourist Guide^A](#), [10765 Doves and Bombs^C](#)。

9.4.7 割边

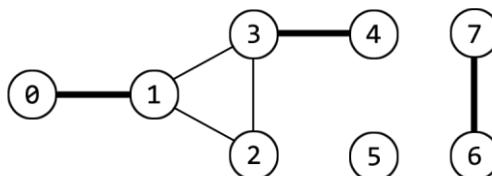
在无向连通图 G 中, 如果移除某条边后, 图不再连通, 称这样的边为割边 (cut edge) 或桥 (bridge)。朴素的方法是使用图遍历逐一判断一条边是否为桥: 先从连通图中移除该条边 (并不需要实际删除该边, 只需在遍历时跳过该条边即可), 然后使用深度优先 (或者广度优先) 遍历对图进行一次遍历操作, 检查从起点到其他各点之间是否仍相互连通, 如果是, 则该边不是桥, 否则为桥。此种方法简单直接, 但是需要对每条边都进行一次图遍历, 效率不高, 用于数据规模较小的题目尚可, 对于数据规模较大的题目一般会超时。

强化练习: [610 Street Directions^B](#)。

一个具有 n 个顶点的无向图, 至多包含 $n-1$ 条桥, 如果在任意两个顶点间再增加一条边, 则会形成圈。具有 n 个顶点且恰好包含 $n-1$ 条桥的无向图实际上就是一棵树 (tree), 如果一个无向图中的所有边均为桥, 则该无向图称为林 (forest)。利用前述的 Tarjan 算法, 对其做少量修改, 即可只进行一次深度优先遍历操作, 就能够找出无向连通图中所有的桥。在寻找割顶的 Tarjan 算法中, 如果无向图中的一条边 (u, v) 是桥, 当且仅当 (u, v) 为生成树中的边, 且满足 $dfn[u] < low[v]$ 。 (u, v) 为生成树中的边, 表明 v 是以 u 为根的深度优先树中的子孙结点, 而 $dfn[u] < low[v]$ 表明结点 v 无法通过反向边到达比 u 更早发现的顶点, 意味着只要把边 (u, v) 断开, 顶点 v 和顶点 u 的祖先顶点无法连通, 故 (u, v) 是桥。如果两个顶点间有平行边存在, 则任意一条平行边均不是桥。

[796 Critical Links^A](#) (关键连接)

在计算机网络中, 如果有至少两台主机 A 和 B 之间的所有互连路径均通过连接 L , 则连接 L 为关键连接。移除关键连接会产生两个不连通的计算机子网络, 而在子网络中的任意两台主机是互相连通的。例如下图所示的计算机网络, 有 3 条加粗的连接均为关键连接, 它们是: 0-1, 3-4 和 6-7。



假定: (1) 所有连接为双向连接; (2) 主机不存在直接连到自身的连接; (3) 两台主机如果直接相连或

者同时和一台主机互连，则这两台主机是互连的；(4) 计算机网络中可以存在不与其他子网络相连通的其他子网络。编写程序找出给定计算机网络中的所有关键连接。

输入

输入文件包含多组测试数据。每组测试数据按以下格式指定一个计算机网络：

```
no_of_servers
server0 (no_of_direct_connections) connected_server ... connected_server
...
serverno_of_servers-1 (no_of_direct_connections) connected_server ... connected_server
```

每组测试数据的第一行包含一个正整数 *no_of_servers* (可能为 0)，表示在该计算网络中主机的数量。接着的 *no_of_servers* 行数据，每行表示一台主机和其他主机的互连情况，其顺序是随机排列的。对于主机 *server_k*, $0 \leq k \leq no_of_servers - 1$ ，该行指明了与主机 *server_k* 直接连接的主机数量和相应的主机编号。你可以假定输入数据是正确无误的。样例输入数据的第一组数据对应图示中的计算机网络，第二组数据指定了一个空的计算机网络。

输出

对于每组测试数据，输出关键连接的数量和每条关键连接的信息。输出格式参见样例输出。在输出关键连接时，起始主机编号小的排列在前，如果多个关键连接具有相同的起始主机编号时，按终止主机编号升序排列。在每组测试数据后输出一个空行。

样例输入

```
8
0 (1) 1
1 (3) 2 0 3
2 (2) 1 3
3 (3) 1 2 4
4 (1) 3
7 (1) 6
6 (1) 7
5 (0)

0
```

样例输出

```
3 critical links
0 - 1
3 - 4
6 - 7

0 critical links
```

分析

题目所求的“关键连接”对应图论中桥的概念。使用 Tarjan 算法求图的所有桥即可。需要注意的是，由于给定的图并不一定是连通图，可能是由多个连通子图构成，故在寻找桥时，需要搜索所有的连通子图。

参考代码

```
const int MAXV = 2010;

struct edge {
    int start, end;
    bool operator<(const edge &e) const {
        if (start != e.start) return start < e.start;
        else return end < e.end;
    }
};
```

```

// 以邻接表方式表示图。
vector<int> g[MAXV];
vector<edge> bridge;
int dfn[MAXV], low[MAXV], visited[MAXV];

// 对 Tarjan 算法适当修改求图中的割边。
void dfs(int u, int parent, int depth) {
    visited[u] = 1; dfn[u] = low[u] = depth;
    for (auto v : g[u]) {
        if (v != parent && visited[v] == 1) low[u] = min(low[u], dfn[v]);
        if (!visited[v]) {
            dfs(v, u, depth + 1);
            low[u] = min(low[u], low[v]);
            if (dfn[u] < low[v]) bridge.push_back((edge){u, v});
        }
    }
    visited[u] = 2;
}

int main(int argc, char *argv[]) {
    int servers;
    while (cin >> servers) {
        // 构建图。
        for (int i = 0; i < servers; i++) g[i].clear();
        string s;
        for (int i = 1, u, v, c; i <= servers; i++) {
            cin >> u >> s;
            c = stoi(s.substr(1, s.length() - 2));
            for (int j = 1; j <= c; j++) {
                cin >> v;
                g[u].push_back(v), g[v].push_back(u);
            }
        }
        // Tarjan 算法求图中割边。
        bridge.clear(); memset(dfn, 0, sizeof(dfn));
        memset(low, 0, sizeof(low)); memset(visited, 0, sizeof(visited));
        for (int u = 0; u < servers; u++)
            if (!visited[u])
                dfs(u, -1, 1);
        // 输出。
        for (int i = 0; i < bridge.size(); i++)
            if (bridge[i].start > bridge[i].end)
                swap(bridge[i].start, bridge[i].end);
        cout << bridge.size() << " critical links\n";
        sort(bridge.begin(), bridge.end());
        for (int i = 0; i < bridge.size(); i++)
            cout << bridge[i].start << " - " << bridge[i].end << '\n';
        cout << '\n';
    }
    return 0;
}

```

强化练习: 1310 One-Way Traffic^E, [12783 Weak Links^D](#)。

扩展练习: [12363* Hedge Maze^D](#)。

9.4.8 强连通分支

给定有向图 D ，如果 D 中的任意两个顶点 u 和 v ，既存在从 u 到 v 的通路，也存在从 v 到 u 的通路，则称图是强连通的（strongly connected）。而强连通分支（strongly connected component，或称强连通分量）则是指有向图的一个子图，在这个子图中，任意两个顶点之间存在有向通路，即该子图是强连通的。寻找强连通分支是处理有向图的一个常见操作。在找到强连通分支后，由于其内部顶点是双向连通的，可以使用一个虚拟的新顶点来替代原来的强连通分支中的所有顶点——称之为“缩点”（condensation），这样可以将原有包含圈的有向图转化为有向无圈图，从而对下一步图的处理带来便利。求有向图强连通分支有两种常用的算法，一种是 Kosaraju 算法，另一种是 Tarjan 算法，下面分别予以介绍。

Kosaraju 算法

Kosaraju 算法通过对有向图 D 进行一次 DFS，然后对 D 的逆图 D^T 再进行一次 DFS，从而找出原图的强连通分支。逆图 D^T 是指将原有向图 D 中的边反向后得到的图，设 $D=(V, E)$ ，则 $D^T=(V, E^T)$ ， $E^T=\{(u, v) | (v, u) \in E\}$ 。Kosaraju 算法基于以下结论：有向图 D 与其逆图 D^T 具有完全相同的强连通分支。换句话说，如果有向图 D 的一个子图 D' 是强连通子图，在将图 D' 中的各条边反向之后， D' 仍然是强连通子图，但反过来不成立——如果子图 D' 是单向连通的，将各边反向后可能某些顶点之间将不再连通，即 D' 不再是强连通子图。究其原因，属于某强连通分支的顶点，边反向操作不会对其构成影响，如果不是强连通分支内的顶点，边反向会导致其不属于逆图中的某个强连通分支，因此可以将边反向的操作看成是对非强连通块的过滤处理。

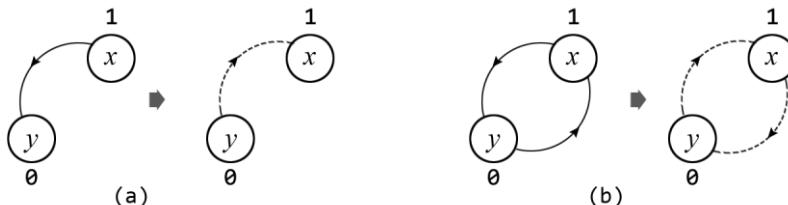


图 9-13 Kosaraju 算法中将边反向所起的“过滤”作用。(a) 顶点 y 的 DFS 完成时间为 0，顶点 x 的 DFS 完成时间为 1，将边反向后，从 DFS 完成时间晚的顶点 x 出发访问， y 无法访问，表明 x 和 y 属于不同的强连通分支。(b) 原图中从顶点 y 到顶点 x 的有向边经过反向后成为从顶点 x 到顶点 y 的有向边，从而保证能够从 DFS 完成时间晚的顶点 x 访问顶点 y ，表明顶点 x 和 y 同属一个强连通分支

Kosaraju 算法的执行步骤如下：

- 1) 构建原图 D ，对原图 D 进行 DFS，记录每个顶点在 DFS 过程中的完成时间 $ft[u]$ （注意不是顶点的发现时间 $dfn[u]$ ）；
- 2) 构建原图 D 的逆图 D^T ；
- 3) 选择从第一次遍历得到的 $ft[u]$ 最大的顶点出发，对逆图 D^T 进行 DFS，删除能够访问的顶点，这些被删除的顶点构成了一个强连通分支；
- 4) 如果还有顶点尚未删除，继续执行第 3) 步，否则算法结束。

在下述 Kosaraju 算法的实现中，第一次遍历使用普通的 DFS，记录顶点在深度优先遍历过程中的完成时间。为了简便，使用 `vector`（或者 `stack`）来记录顶点的完成访问的顺序，在 `vector` 的末尾是最后完成访问的顶点。第二次 DFS 针对逆图进行，形式上与第一次 DFS 类似，使用了 `vector` 来记录强连通分支的顶点以便输出。Kosaraju 算法的时间复杂度为 $O(|V| + |E|)$ 。

```
//-----9.4.8.1.cpp-----//
```

```

const int MAXV = 110;

// visited 记录顶点是否已被访问, n 表示顶点的数量, cscc 记录强连通分支数。
int visited[MAXV], n, cscc;
// g1 表示原图, g2 表示逆图。
vector<list<int>> g1(MAXV), g2(MAXV);
// ft 记录顶点在 DFS 中的完成时间顺序, scc 记录单个强连通分支所包含的顶点。
vector<int> ft, scc;

// 对原图进行 DFS, 按完成遍历的时间将顶点送入 vector 中。
// 越靠近 vector 末端的顶点, 其发现时间越早, 与之相对应的完成时间越晚。
void dfs(int u) {
    visited[u] = 1;
    for (auto v : g1[u])
        if (!visited[v])
            dfs(v);
    ft.push_back(u);
}

// 将原图的边反向, 构建逆图。
void reverseEdge() {
    for (int u = 1; u <= n; u++)
        for (auto v : g1[u])
            g2[v].push_back(u);
}

// 对逆图进行 DFS, 找出强连通分支。
void rdfs(int u) {
    visited[u] = 1;
    for (auto v : g2[u])
        if (!visited[v])
            rdfs(v);
    scc.push_back(u);
}

void kosaraju() {
    // 对原图进行 DFS。
    ft.clear();
    memset(visited, 0, sizeof visited);
    for (int u = 1; u <= n; u++)
        if (!visited[u])
            dfs(u);
    // 将边反向, 构建逆图。
    reverseEdge();
    // 从第一次 DFS 过程中完成时间最大的顶点开始, 对逆图进行 DFS, 找出强连通分支并输出。
    cscc = 0;
    memset(visited, 0, sizeof visited);
    while (ft.size()) {
        int u = ft.back();
        if (!visited[u]) {
            cscc++;
            scc.clear();
            rdfs(u);
            cout << "cscc = " << cscc << ':';
            for (auto v : scc) cout << ' ' << v;
            cout << '\n';
        }
    }
}

```

```

        ft.pop_back();
    }
//-----9.4.8.1.cpp-----//

```

强化练习：[247 Calling Circles^A](#)。

Tarjan 算法

Tarjan 算法以此算法的发现者 Robert Endre Tarjan 的名字命名^[86]，相较于 Kosaraju 算法，Tarjan 算法显得更为简洁而巧妙，其时间复杂度同样为 $O(|V| + |E|)$ 。该算法充分利用了 DFS 过程中记录的深度优先数。在 DFS 过程中，随着遍历的深入，各个顶点形成了一棵深度优先树。而在树的生成过程中，Tarjan 算法使用了一个栈来保存各个顶点，由于顶点的入栈顺序保持了一个称为“栈不变量”的性质，算法根据“栈不变量”将顶点划分到各个强连通分支中。以下给出的是 Tarjan 算法中通过 DFS 过程确定强连通分支根结点的伪代码表示：

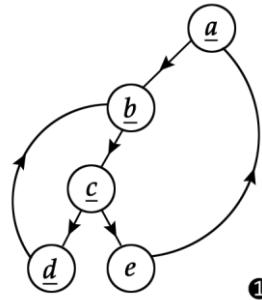
```

dfs(u) {
    // 设置结点 u 的深度优先数和 low 值。
    low[u] = dfn[u] = ++dfntime
    // 将结点 u 压入栈中。
    stack.push(u)
    // 遍历每一条边。
    for each (u, v) in E
        // 如果 v 尚未访问则继续遍历，然后再更新 u 的 low 值。
        if (v is not visited)
            dfs(v)
            low[u] = min(low[u], low[v])
        // 如果 v 已经访问但仍在栈中，表明 v 尚未成为强连通分支中的顶点，更新 u 的 low 值。
        else if (v in stack)
            low[u] = min(low[u], dfn[v])
    // 如果结点 u 是强连通分支的根则开始退栈。
    if (low[u] == dfn[u])
        // 持续退栈，退出的结点 v 为该强连通分支的一个顶点。
        repeat
            v = stack.top()
            stack.pop()
            print v
        until (u == v)
}

```

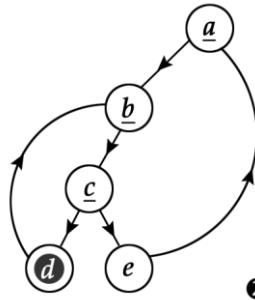
算法中结点的 *low* 值应用非常巧妙，它记录的是结点通过反向边所能达到的最早发现时间 *dfn*。回顾 DFS 过程，数组 *dfn* 记录的是结点的发现时间，称之为深度优先数，深度优先数是单调递增的，当第一次访问某个结点 *u* 时，其 *dfn[u]* 和 *low[u]* 的值是相等的。在 DFS 过程中，结点 *u* 的 *low* 值取其子树中所有结点 *low* 值的最小值，如果以结点 *u* 为根的子树中存在能够到达 *u* 的祖先结点的反向边，那么结点 *u* 的 *low[u]* 会小于 *dfn[u]*。对于结点 *u*，当其所有的子树结点均已访问完毕时，如果发现 *low[u]* 等于 *dfn[u]*，可以证明，从结点 *u* 在栈中的位置直到栈顶的任意结点 *v*，均有 *low[v]* 小于等于 *dfn[u]*，而且在这些结点中，一定有且只有一个结点 *x*，其 *low[x]* 等于 *dfn[x]*，这个结点就是某个强连通分支的根，也就是结点 *u*——该性质即为前述所提到的“栈不变量”性质。为什么会这样呢？因为从 Tarjan 算法的伪代码中可以看到，如果某个结点 *u* 的 *low[u]* 等于 *dfn[u]*，表明结点 *u* 的 *low* 值无法通过其本身或者子孙结点得到改变，那么就表示该结点

和其子孙结点不存在能够到达结点 u 的祖先结点的反向边，从而使得结点 u 要么是一个只包含一个结点的强连通分支，要么是与其若干个子孙结点共同组成一个强连通分支。读者可以从图 9-14 的算法执行过程很清楚地看到这一点。



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	2	2
<u>c</u>	3	3
<u>d</u>	4	4

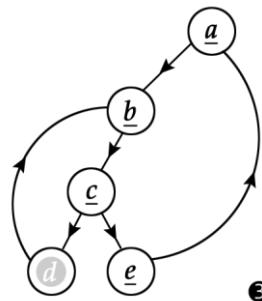
①



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	2	2
<u>c</u>	3	3
<u>d</u>	2	4

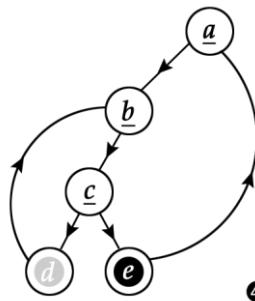
②

①：按照深度优先遍历的顺序，访问起始结点 a ，此时 $low[a]=dfn[a]=1$ ；访问结点 b ，此时 $low[b]=dfn[b]=2$ ；访问结点 c ，此时 $low[c]=dfn[c]=3$ ；访问结点 d ，此时 $low[d]=dfn[d]=4$ 。②：结点 d 有一条有向边连接到结点 b ，在深度优先遍历中，该边为反向边，因为结点 b 已经在栈中，故最终 $low[d]=\min(low[d], dfn[b])=2$



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	2	2
<u>c</u>	3	3
<u>d</u>	2	4
<u>e</u>	5	5

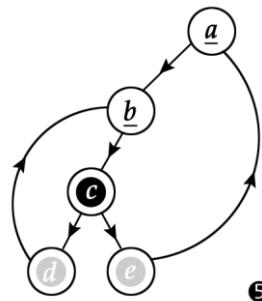
③



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	2	2
<u>c</u>	3	3
<u>d</u>	2	4
<u>e</u>	1	5

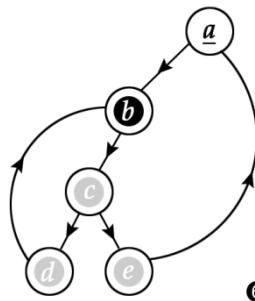
④

③：访问结点 e ，此时有 $low[e]=dfn[e]=5$ 。④：结点 e 有一条有向边连接到结点 a ，同样为反向边，由于结点 a 在栈中，故 $low[e]=\min(low[e], dfn[a])=1$



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	2	2
<u>c</u>	1	3
<u>d</u>	2	4
<u>e</u>	1	5

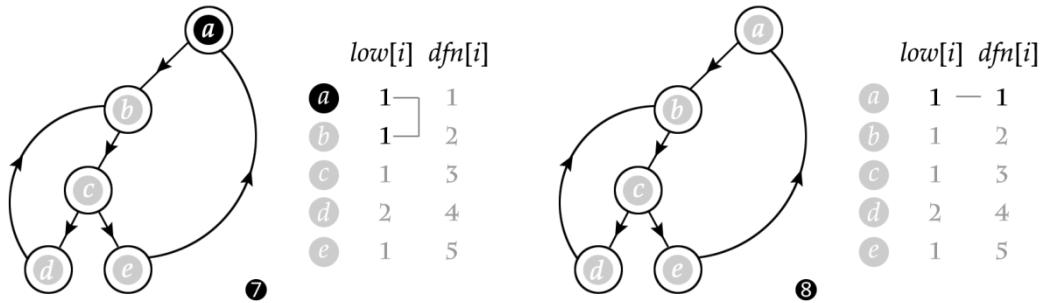
⑤



	low[i]	dfn[i]
<u>a</u>	1	1
<u>b</u>	1	2
<u>c</u>	1	3
<u>d</u>	2	4
<u>e</u>	1	5

⑥

⑤：结点 c 的子结点 d 和 e 均已访问完毕，结点 c 的 low 值取结点 d 和 e 的 low 值的较小值，即 $low[c]=\min(low[d], low[e])=1$ 。⑥：结点 b 的子结点 c 已经访问完毕，DFS 过程退回结点 b ，此时有 $low[b]=\min(low[b], low[c])=1$



⑦：DFS 过程退回结点 *a*，此时有 $low[a]=\min(low[a], low[b])=1$ 。⑧：最终，只有结点 *a* 满足 $dfn[a]$ 等于 $low[a]$ 的条件。开始退栈，一直退栈到结点 *a* 为止，从栈顶到结点 *a* 的所有结点均属于同一强连通分支，即结点 *a*、*b*、*c*、*d*、*e* 均属于同一强连通分支

图 9-14 Tarjan 算法执行过程

以下为 Tarjan 算法的具体实现。在使用栈存储访问的结点时，由于栈数据结构一般不支持查找，需要辅助数据结构来记录结点是否已经进入某个强连通分支。如果某个结点已经进入强连通分支，则该结点肯定不会在栈中。栈可以使用 STL 中的 *stack* 数据结构，或者使用一维数组来模拟。

```

//-----9.4.8.2.cpp-----
const int MAXV = 10010;

int n;
int dfstime = 0, dfn[MAXV], low[MAXV], scc[MAXV], cscc = 0;
vector<vector<int>> g;
stack<int> s;

void dfs(int u) {
    // 记录深度优先数，初始化 low 值。
    low[u] = dfn[u] = ++dfstime;
    // 将当前结点入栈。
    s.push(u);
    // 遍历所有子树，更新 low 值。
    for (auto v : g[u]) {
        if (!dfn[v]) dfs(v), low[u] = min(low[u], low[v]);
        else if (!scc[v]) low[u] = min(low[u], dfn[v]);
    }
    // 退栈，直到强连通分支的根。
    if (low[u] == dfn[u]) {
        ++cscc;
        while (true) {
            int v = s.top(); s.pop();
            scc[v] = cscc;
            if (u == v) break;
        }
    }
}

void tarjan() {
    dfstime = 0, cscc = 0;
    while (!s.empty()) s.pop();
}

```

```
    memset(dfn, 0, sizeof(dfn));
    memset(scc, 0, sizeof(scc));
    for (int i = 0; i < n; i++)
        if (!dfn[i])
            dfs(i);
}
//-----9.4.8.2.cpp-----//
```

理解强连通分支 Tarjan 算法的关键在于理解以下核心语句：

```

for (auto v : g[u]) {
    if (!dfn[v]) dfs(v), low[u] = min(low[u], low[v]);
    else if (!scc[v]) low[u] = min(low[u], dfn[v]);
}

```

`if` 语句的第一个分支：先对 u 所有尚未访问的子结点 v 进行遍历，然后再更新 u 的 low 值。这个很容易理解，因为有可能 u 的子结点（或者 u 的子孙结点）存在反向边，这些反向边能够到达 u 的祖先结点，因此 u 的 low 值能够发生改变，从而使得 u 不会是强连通分支的根，所以应该先遍历 u 的子结点后再确定 u 的 low 值。此种情形如图 9-15 所示。

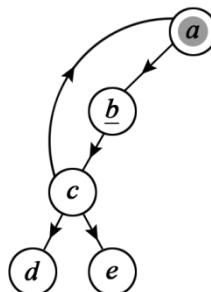


图 9-15 Tarjan 算法核心语句中第一个判断分支的理解。假设结点 a 已经访问，当前访问结点 b ，由于结点 b 的子结点 c 具有一条反向边能够到达 b 的祖先结点 a ，在遍历 c 后， b 能够具有更小的 low 值

if 语句的第二个分支：当 u 的某个子结点 v 已经访问，那么就检查 v 是否已经属于某个强连通分支，如果已经属于某个强连通分支，说明 u 和 v 之间是一条交叉边，则不予处理。因为 u 和 v 之间的边并不是有向圈的一部分，所以不应对 u 的 low 值产生影响。此种情形如图 9-16 所示。

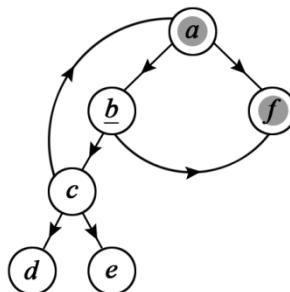


图 9-16 Tarjan 算法核心语句中第二个判断分支的理解。假设结点 a 和结点 f 已经访问，当前访问结点 b ，由于结点 b 的一条边指向结点 f ，而结点 f 已经自身构成一个强连通分支，不应对结点 b 的 low 值产生影响

若 v 尚未进入某个强连通分支，则表明 u 和 v 之间的边是有向圈的一部分。此种情形如图 9-17 所示。

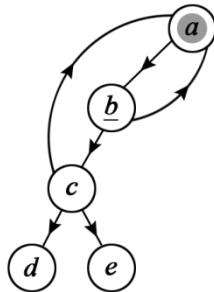


图 9-17 Tarjan 算法核心语句中第二个判断分支的理解。假设结点 a 已经访问，当前访问结点 b ，由于结点 b 的一条边指向结点 a ，则该有向边将 a 和 b 组成一个圈，此时结点 a 尚未进入某个强连通分支，因此结点 b 的 low 值应取结点 a 的 dfn 值

在 `if` 语句的第二个分支中，当结点 v 尚未进入某个强连通分支时，为什么 u 的 low 值是取 $low[u]$ 和 $dfn[v]$ 的较小值而不取 $low[u]$ 和 $low[v]$ 的较小值呢？这是因为 v 可能是 u 的祖先结点，而此时 v 的 low 值还未最终确定（因为 v 的子孙结点中可能有反向边到达具有更小 dfn 值的祖先结点），因此并不能取 $low[v]$ ，而 $dfn[v]$ 是确定的，不会发生改变，此时 $low[u]$ 的含义就是 u 通过反向边能够到达的具有最小 dfn 值的某个结点的相应的 dfn 值，符合 low 值的定义，若取 $low[u]$ 和 $low[v]$ 的较小值，则不符合 low 值的定义。值得一提的是，在求强连通分支时，如果实现时 `if` 语句的第二个分支取 $low[u]$ 和 $low[v]$ 的较小值，仍然会得到正确的结果，尽管这不符合算法的原理。但是在使用 Tarjan 算法求割顶时，这样做就很可能会导致错误的结果，因为求强连通分支本质是求边双连通分支，而求割顶实质是求点双连通分支，两者是不同的问题。

12167 Proving Equivalences^D (等价证明)

考虑如下线性代数练习题：设 A 为 $n \times n$ 的矩阵，证明下列四个命题相互等价。

- (a) 矩阵 A 不可逆。
- (b) 对于每个 $n \times 1$ 的矩阵 b ，方程 $Ax=b$ 只有一个解。
- (c) 方程 $Ax=b$ 对于任意一个 $n \times 1$ 的矩阵 b 来说都是一致的。
- (d) 方程 $Ax=0$ 只有一个平凡解 $x=0$ 。

常规的证明方法是证明一系列的隐含关系。例如，可以先证明由 (a) 可以推出 (b)，然后证明 (b) 可以推出 (c)，再证明 (c) 可以推出 (d)，最后证明 (d) 可以推出 (a)，由以上四个推断可以得出四个命题是等价的。

另外一种证明方法是先证明 (a) 和 (b) 等价，即证明 (a) 可以推出 (b)，(b) 可以推出 (a)，再证明 (b) 和 (c) 等价，最后证明 (c) 和 (d) 等价，不过这样的证明方法需要证明六个隐含关系，比前一种只需要证明四个隐含关系的证明方法，其证明量显然要多。

现在我手头上有一些类似的作业需要完成，有的作业中已经证明了一部分隐含关系。现在我想知道的是至少还需要证明多少个隐含关系才能完成所有命题等价性的证明，你能帮助我确定这个数量吗？

输入

输入的第一行是一个正整数，表示测试数据的组数，最多有 100 组测试数据。每组测试数据的第一行包含两个整数 n ($1 \leq n \leq 20000$) 和 m ($0 \leq m \leq 50000$)，分别表示命题的数量和已经证明的隐含关系的数量，

后面接着 m 行每行包含两个整数 s_1 和 s_2 ($1 \leq s_1, s_2 \leq n$ 且 $s_1 \neq s_2$), 表示 s_1 隐含 s_2 的关系已经证明。

输出

对于每组测试数据输出一行, 该行包含一个整数, 表示为了证明所有命题是等价的, 至少还需要证明多少个隐含关系。

样例输入

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

样例输出

```
4
2
```

分析

将命题视为顶点, 可以将题目所给条件构造一个有向图, 之后对该有向图进行“缩点”操作, 即先找到有向图中的所有强连通分支, 将每个强连通分支使用一个顶点予以替代。为什么可以这样操作呢? 因为某个顶点 u 和给定强连通分支 s 中的某个顶点 v 双连通, 则顶点 u 必定和该强连通分支 s 中的其他顶点双连通, 所以可以将整个强连通分支 s 使用一个顶点作为代表。进行缩点操作后, 得到的有向图中的任意两个顶点要么是单向连通, 要么不连通, 即原图成为有向无圈图, 问题转化为在这个新图中至少需要添加多少条有向边才能使之成为强连通图。可以对构造得到的有向无圈图进行一次 DFS, 这样会得到若干棵深度优先树, 如图 9-18 所示。

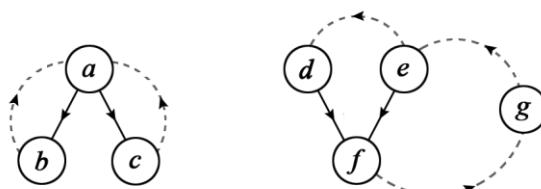


图 9-18 “缩点”操作后进行 DFS 得到的深度优先树。左侧的深度优先树中, 有 1 个结点的入度为 0, 两个结点的出度为 0, 直观地, 从出度为 0 的叶结点 b 连接一条有向边到入度为 0 的结点 a 即可构成一个有向圈, 形成一个强连通分支, 从而“消除”一个叶结点。同此法, 可添加有向边将结点 c 并入强连通分支。右侧的深度优先树中, 有一个独立的结点 g , 可以通过从叶结点 f 连接有向边到 g , 然后从 g 连接有向边到 e 形成一个强连通分支, 之后按照类似左侧深度优先树的操作, 将树变成强连通图

在深度优先树中, 直观地, 只需从出度为 0 的结点 (叶结点) 引一条有向边到入度为 0 的结点 (根结点), 即可将这些结点联系在一起, 构成一个强连通分支。经过观察, 可以得出结论: 需要添加的边数为缩点后新图中出度为 0 的顶点数 V_1 和入度为 0 的顶点数 V_2 两者的较大值—— $\max(V_1, V_2)$ 。当原图已经是强连通图时, 不需再进行缩点和后续操作, 可以进行特殊判断以节省时间。在进行缩点操作时, 新图的顶点编号可以使用强连通分支的标号数来表示。

参考代码

```
const int MAXV = 20010;

int dfn[MAXV], low[MAXV], scc[MAXV], dfstime, cscc;
int cases, n, m;
vector<list<int>> g(MAXV);
```

```

stack<int> s;

void dfs(int u) {
    // 代码省略, 请参考前述给出的 Tarjan 算法实现。
}

// Tarjan 算法求强联通分支。
void tarjan() {
    // 代码省略, 请参考前述给出的 Tarjan 算法实现。注意, 本题中顶点序号从 1 开始计数。
}

int main(int argc, char *argv[]) {
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        for (int u = 1; u <= n; u++) g[u].clear();
        // 以邻接表方式读入图数据。
        cin >> n >> m;
        for (int e = 1, u, v; e <= m; e++) {
            cin >> u >> v;
            g[u].push_back(v);
        }
        // Tarjan 算法求强连通分支。
        tarjan();
        // 如果已经是强连通的则不再继续计算。
        if (cscc == 1) cout << "0\n";
        else {
            // 将同一强连通分支中的顶点视为一个顶点, 计数其出度及入度。
            int id[MAXV] = {0}, od[MAXV] = {0};
            for (int u = 1; u <= n; u++)
                for (auto v : g[u]) {
                    if (scc[u] == scc[v]) continue;
                    od[scc[u]] = id[scc[v]] = 1;
                }
            // 计数缩点操作后新图中出度或入度为 0 的顶点数的较大值即为所求。
            int tid = 0, tod = 0;
            for (int u = 1; u <= cscc; u++) {
                if (!id[u]) tid++;
                if (!od[u]) tod++;
            }
            cout << max(tid, tod) << '\n';
        }
    }
    return 0;
}

```

除了前述介绍的 Kosaraju 算法和 Tarjan 算法, 求强连通分支还有一种算法称之为 Gabow 算法^[87], 此算法是 Tarjan 算法思想的另外一种实现, 相较于 Tarjan 算法, 虽然时间复杂度相同, 但 Gabow 算法更为巧妙, 不需要频繁更新 low 值, 因而具有更小的常数项。

强化练习: [10731 Test^C](#), [11686 Pick up Sticks^B](#), [11709 Trust Groups^B](#), [11770 Lighting Away^B](#), [11838 Come and Go^B](#)。

扩展练习: [1229 Sub-Dictionary^D](#), [13057 Prove Them All^E](#)。

9.4.9 半连通分支

有向图 D 是强连通的, 要求图 D 中任意顶点对 u 和 v 之间既存在 u 到 v 的有向通路, 也存在 v 到 u 的有向通路。如果有向图 D 中任意顶点对 u 和 v 之间, 存在 u 到 v 的有向通路或者 v 到 u 的有向通路, 则称图 D 是半连通的 (semi-connected)。若 D' 是 D 的导出子图, 且 D' 是半连通的, 则称 D' 为 D 的半连通子图。若 D' 是 D 的所有半连通子图中包含顶点数最多的, 则称 D' 是 D 的最大半连通子图。

通过前述介绍的强连通分支算法, 可以对有向图中的所有圈进行缩点操作, 使之成为有向无圈图, 则寻找最大半连通子图相当于在此有向无圈图中寻找一个“最长链”, 从缩点得到的新图中入度为 0 的顶点开始进行 DFS, 在 DFS 过程中统计链上所包含顶点的数量, 包括经过缩点后的强连通分支中包含的顶点数, 其中最长的一条路径所包含的顶点数即为最大半连通子图的顶点数。

强化练习: [11324 The Largest Clique](#)。

9.4.10 2-SAT

布尔适定性问题 (boolean satisfiability problem) 或适定性问题 (satisfiability problem) 是指给定一组布尔值变量, 确定由这些变量构成的布尔表达式是否能够全为真。一般为了方便, 将适定性问题取其英文名称前三个字母, 简称为 SAT。SAT 的一般形式是非确定性多项式 (Non-deterministic Polynomial, NP) 完全的, 只有特殊情况下才存在有效算法, 其中 2-SAT 即为一个特例。2-SAT 是指布尔表达式由若干个子句 (clause) 的合取 (conjunction, 布尔与) 构成, 而每个子句由两个布尔变量的析取 (disjunction, 布尔或) 或者单个布尔值构成, 即表达式的形式类似于

$$(b_1) \wedge (b_1 \vee b_2) \wedge (b_1 \vee \neg b_3) \wedge (\neg b_2 \vee \neg b_3) \wedge \cdots \wedge (b_i \vee b_j), \quad 1 \leq i, j \leq n$$

在表达式中出现的各种形式的布尔变量 (或布尔变量的否定) 称之为字面量 (literal), 例如 b_1 和 $\neg b_3$ 。若所有表达式中, 子句中的字面量最多为 k 个, 则称为 k -SAT 问题, 当 $k > 2$ 时, k -SAT 问题为 NP 完全的。2-SAT 存在多种有效算法, 下面介绍通过求图的强连通分支来巧妙地解决 2-SAT 问题的方法^[88]。此方法可以概述为以下步骤:

- (1) 将给定的 n 个布尔变量使用 $2n$ 个顶点予以表示, 变量 b_i 和 $\neg b_i$ 各对应一个顶点;
- (2) 根据布尔表达式在顶点间建立有向边, 从而得到变量间的拓扑关系图;
- (3) 对该有向图进行“缩点”操作, 使之转化为有向无圈图;
- (4) 如果某个强连通分支中包含一个变量所对应的两个顶点, 则原问题无解;
- (5) 对得到的有向无圈图进行拓扑排序;
- (6) 按照拓扑排序的逆序为变量赋值, 位于前面的变量尽量取真, 即可得到原问题的一组解。

接下来介绍如何将给定的布尔表达式建模为有向图。对于布尔变量 b_i , 将变量 b_i 取真时的状态使用图中的一个顶点予以表示, 为了简便起见, 仍然使用 b_i 来指示图中的这个顶点。类似的, 使用 $\neg b_i$ 表示当 b_i 取假时所对应的图中的顶点, 将图中同一个变量取真值和假值所对应的两个顶点互称为补 (complements)。假设一个子句中的析取式为 $b_i \vee b_j$, 则在对应的有向图中建立两条边, 一条边为 $\neg b_i \rightarrow b_j$, 含义为若 $\neg b_i$ 为真, 则 b_j 必定为真; 另一条边为 $\neg b_j \rightarrow b_i$, 含义为若 $\neg b_j$ 为真, 则 b_i 必定为真。为什么需要这样建立有向边呢? 因为这样可以将变量间隐含的拓扑关系表示出来。对于析取式 $b_i \vee b_j$, 变量间的拓扑关系意味着不随图的变化而始终存在的性质, 如果将上述建立的有向图全部边反向, 同时将相应的变量变换为它的“补”, 可以发现新图和原图是同构的, 此即对偶性 (duality property)。对偶性是由建立有向边的方式决定的。

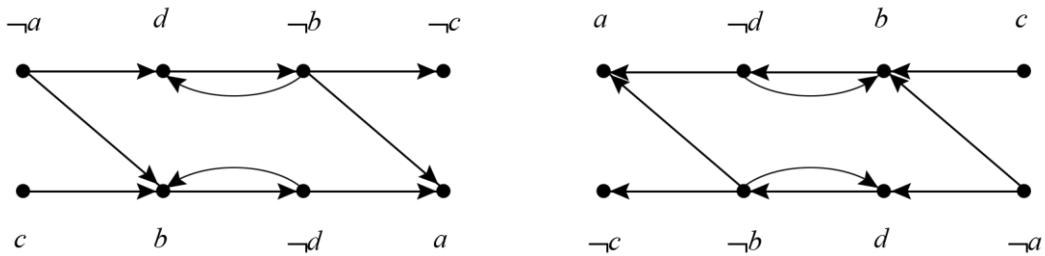


图 9-19 对偶性。左侧为根据布尔表达式 $(a \vee b) \wedge (b \vee \neg c) \wedge (\neg b \vee \neg d) \wedge (b \vee d) \wedge (d \vee a)$ 所构建的有向图。将所有顶点更换为对应顶点的补，同时将边反向所得到的右侧有向图与左侧有向图同构

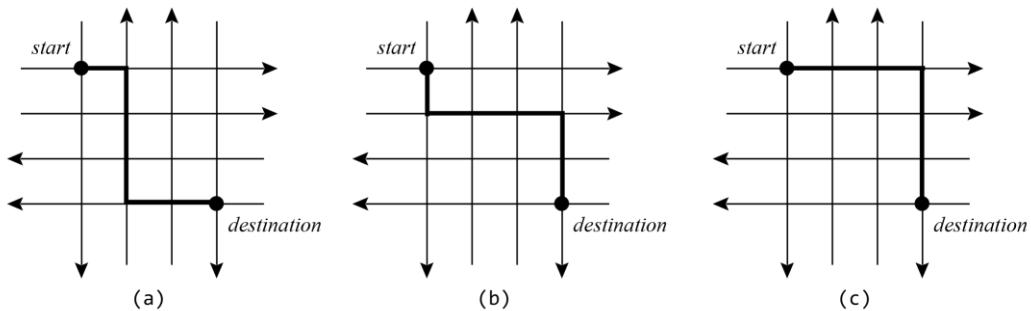
如果变量赋值使得 $b_i \vee b_j$ 为真，那么有三种情况，要么 b_i 为真，要么 b_j 为真，要么两者都为真。把拓扑关系图中的边理解为“必须”，则初始赋值结果使得 $\neg b_i$ 为真（亦即 b_i 为假），“必须”要求 b_i 为真。类似的，如果 $\neg b_i$ 为真（亦即 b_i 为假），“必须”要求 b_i 为真。根据这样的形式建立的有向图能够反映出原布尔变量间的拓扑关系。对于其他形式的析取式，可以类似地建立边。例如析取式 $b_i \vee \neg b_j$ ，则建立有向边 $\neg b_i \rightarrow \neg b_j$ 和 $b_j \rightarrow b_i$ 。对于由单个字面量构成的子句，例如 (b_i) ，则由于 $(b_i) \equiv (b_i \vee b_i)$ ，则从 $\neg b_i$ 到 b_i 建立一条有向边^I。需要注意，有些题目中给定的初始条件可能并不是前述所提到的“规则”的布尔合取式，需要使用布尔运算将其转化为符合要求的合取式，从而能够将其转化为变量间的拓扑关系图，例如 $(b_1 \wedge b_2) \vee (b_3 \wedge b_4)$ 不是建模所需要的合取式，但可以将其转化为 $(b_1 \vee b_3) \wedge (b_1 \vee b_4) \wedge (b_2 \vee b_3) \wedge (b_2 \vee b_4)$ ，从而成为期望的合取式。

对于有向图的“缩点”操作，既可以选择 Kosaraju 算法，也可以选择 Tarjan 算法，不同之处在于后续拓扑排序的处理。Tarjan 算法得到的强连通分支顺序恰为拓扑排序的逆序，而 Kosaraju 算法得到的强连通分支顺序为拓扑排序而非逆序。缩点操作完成后，如果某个变量取真、假值所对应的两个顶点位于同一强连通分支，则原问题无解，因为不可能保证同一个变量的本身和逆反同时为真。

10319 Manhattan^C（曼哈顿）

某个城市的街道由方格网构成，类似于曼哈顿的街道布局。为了缓解交通拥堵的状况，市长决定将所有街道变为单行道以提高出行效率。但是有些线路需要保持至少一条以上的“简单路径”可供通行，简单路径是指从起点到达终点至多经过一个直角拐弯的路径。给定一份列表，列表上列出了在所有街道变为单行道之后需要保持仍有至少一条以上简单路径的“起点—终点”对，要求你判断是否可以为所有街道指定单行道的方向后，这些“简单路径”能够存在。

^I 对于 $b_i \equiv b_i \vee b_i$ ，严格按照前述建立有向边的规则，应该建立两条 $\neg b_i$ 到 b_i 的有向边，但由于重边对强连通分支无影响，只需使用一条即可。同样的，对于 $\neg b_i \equiv \neg b_i \vee \neg b_i$ ，只需建立一条从 b_i 到 $\neg b_i$ 的有向边。若布尔表达式要求 b_i 和 $\neg b_i$ 同时为真，则建立的有向图中，顶点 b_i 和 $\neg b_i$ 之间有互相到达的有向边，则两者一定属于同一强连通分支，因而无解。



简单路径。(a) 是一条不合法的路径, 因为从起点到终点逆着街道方向行走; (b) 是一条合法的路径, 但不是一条简单路径, 因为有两个转弯; (c) 是一条简单路径

输入

输入包含多组测试数据。输入的第一行包含一个整数 n , 表示测试数据的组数。每组数据的第一行包含三个整数: S , $0 < S \leq 30$, 表示街道网格中经路的数量; A , $0 < A \leq 30$, 表示街道网格中纬路的数量; m , $0 < m \leq 200$, 表示需要存在至少一条简单路径的路线。接下来的 m 行, 每行定义了一条路线。每条路线的定义包含四个整数, s_1, a_1, s_2, a_2 , 表示路线的起点在经路 s_1 和纬路 a_1 的交叉口, 终点在经路 s_2 和纬路 a_2 的交叉口, $0 < s_1, s_2 \leq S$, $0 < a_1, a_2 \leq A$ 。

输出

对于每组测试数据, 如果存在一种为街道指定行驶方向的方案, 使得所有街道成为单行道后, 所给定的路线仍存在至少一条简单路径, 则输出 ‘Yes’, 否则输出 ‘No’。

样例输入

```
2
6 6 2
1 1 6 6
6 6 1 1
7 7 4
1 1 1 6
6 1 6 6
6 6 1 1
4 3 5 1
```

样例输出

```
Yes
No
```

分析

要使得街道单向通行, 必须为街道指定一个通行方向, 街道分为经路和纬路, 这两种类型的道路只能分配一个方向, 将街道的通行方向看做一个布尔变量, 经路向东通行视为真, 向西通行视为假, 纬路向南通行视为真, 向北通行视为假, 则问题转化为能否为经路和纬路指定一个值, 使得简单路径能够存在。

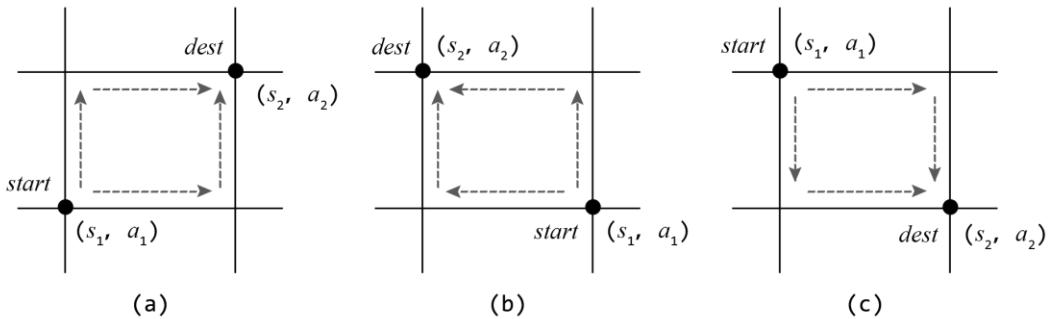


图 9-20 起点和终点处于不同位置时所得到的布尔表达式。假设向右和向上取真，向左和向下取假，位于下方的经路具有较小的序号，位于左侧的纬路具有较小的序号。(a) 起点在左下方，终点在右上方。此时 $(s_1 \wedge a_2) \vee (a_1 \wedge s_2)$ 为真；(b) 起点在右下方，终点在左上方，此时 $(\neg s_1 \wedge a_2) \vee (a_1 \wedge \neg s_2)$ 为真；(c) 起点在左上方，终点在右下方，此时 $(s_1 \wedge \neg a_2) \vee (\neg a_1 \wedge s_2)$ 为真

简单路径如何使用上述布尔变量进行表示呢？起点所在交叉点为 (s_1, a_1) ，终点所在交叉点为 (s_2, a_2) ，将交叉点的坐标视为布尔变量，如果起点和终点存在简单路径，对于图 9-20 的情形 (a)，必须使得以下布尔表达式

$$(s_1 \wedge a_2) \vee (s_2 \wedge a_1)$$

的值为真。但是上述布尔表达式并不是 2-SAT 问题所期望的表达式，为了能够应用 2-SAT 问题的解题方法，需要对其进行变换，根据布尔表达式的分解律

$$b_1 \vee (b_2 \wedge b_3) \equiv (b_1 \vee b_2) \wedge (b_1 \vee b_3)$$

可得

$$(s_1 \wedge a_2) \vee (s_2 \wedge a_1) \equiv ((s_1 \wedge a_2) \vee s_2) \wedge ((s_1 \wedge a_2) \vee a_1) \equiv (s_1 \vee s_2) \wedge (s_1 \vee a_1) \wedge (a_1 \vee a_2) \wedge (s_2 \vee a_2)$$

这样即可将题目约束转换为有向图，进而使用前述介绍的方法进行求解。

在下述实现中，类似于使用数组存储二叉树结点的方法，将同一个变量所对应的两个顶点相邻存储，取真值的顶点序数为偶数，取假值的顶点序数为奇数，这样可以通过异或操作由一个顶点的序号直接获得另外一个顶点的序号。对于本题来说，可能给出的起点和终点位置存在特殊情形，例如 $s_1=s_2$, $a_1=a_2$ ，需要进行特殊处理。

参考代码

```

const int MAXV = 256;

int dfn[MAXV], low[MAXV], scc[MAXV], dfstime, cscc;
vector<list<int>> g(MAXV);
stack<int> s;

void dfs(int u) {
    // 代码省略，请参考前述给出的 Tarjan 算法实现。
}

// Tarjan 算法求强联通分支。
void tarjan(int T) {
    // 代码省略，请参考前述给出的 Tarjan 算法实现。需要注意，本题中的顶点数量为 T。
}

```

```

// 根据布尔表达式添加有向边。
void addEdge(int u, int v) {
    g[u ^ 1].push_back(v);
    g[v ^ 1].push_back(u);
}

int main(int argc, char *argv[]) {
    int cases, S, A, T, m, s1, a1, s2, a2;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> S >> A >> m;
        for (int u = 0; u < MAXV; u++) g[u].clear();
        for (int i = 1; i <= m; i++) {
            cin >> s1 >> a1 >> s2 >> a2;
            // 为布尔变量设置对应的有向图顶点。
            s1--, a1--, s2--, a2--;
            s1 *= 2, s2 *= 2, a1 *= 2, a2 *= 2;
            a1 += 2 * S, a2 += 2 * S;
            // 根据布尔表达式建立有向图。
            if (s1 == s2 && a1 == a2) continue;
            if (a1 > a2) s1 ^= 1, s2 ^= 1;
            if (s1 > s2) a1 ^= 1, a2 ^= 1;
            if (s1 == s2) { g[s1 ^ 1].push_back(s2); continue; }
            if (a1 == a2) { g[a1 ^ 1].push_back(a2); continue; }
            addEdge(s1, a1), addEdge(s1, s2), addEdge(a1, a2), addEdge(s2, a2);
        }
        // 求强连通分支。
        T = 2 * (S + A);
        tarjan(T);
        // 判断同一个布尔变量取真值和取假值所对应的图顶点是否位于同一强连通分支。
        bool flag = true;
        for (int j = 0; j < T && flag; j += 2)
            if (scc[j] == scc[j ^ 1])
                flag = false;
        cout << (flag ? "Yes" : "No") << '\n';
    }
    return 0;
}

```

在确定 2-SAT 是有解后，如何为变量设定值以得到一组解呢？只需将缩点得到的新图进行拓扑排序，按拓扑排序的逆序为变量赋值，位于前面的变量尽量取真，相对应的变量取假，可以通过染色方法较为直观地证明，这样可以得到原问题的一组解^[89]。

由于使用 Tarjan 算法得到的强连通分支恰为拓扑排序的逆序，因此可以在求出强连通分支后，按照框架中所给出的方法为变量指定值，而无需进行更多处理。如果使用 Kosaraju 算法求强连通分支，则需将所有边反向后进行拓扑排序，此时得到的顺序即为拓扑排序的逆序。为变量指定值时，位于拓扑排序逆序前端的变量取真值，接着按照以下规则进行取值：(1) 位于同一强连通分支的变量取值相同；(2) 位于不同强连通分支但与当前强连通分支有边连接的变量取值相同；(3) 后续的强连通分支中的变量则按照继承关系取适当的值即可。具体实现读者可以参考以下代码。

```

//-----9.4.10.cpp-----//
// value 记录各个变量的取值。
vector<int> value;

```

```

// components 记录位于同一个强连通分支内的顶点。
vector<vector<int>> components;

// 获取某个变量的取值。需要注意的是，同一个变量，对应两个顶点，两个顶点的取值总是相反的。
int getValue(int idx) {
    int x = (idx & 1) ? (idx ^ 1) : idx;
    if (value[x] == -1) return -1;
    return (idx & 1) ? !value[x] : value[x];
}

// 按拓扑排序的逆序为变量赋值。
void setValue() {
    // 由于 Tarjan 算法获得的强连通分支按序号从小到大恰为拓扑排序的逆序，因此可以根据各个
    // 顶点所归属的强连通分支序号，将顶点划分到不同的强连通分支中。
    // 因为强连通分支从 1 开始计数，故在重设 component 的大小时将其设置为比 cscc 大 1。
    components.assign(cscc + 1, vector<int>());
    for (int i = 0; i < 2 * n; i++) components[scc[i]].push_back(i);
    // 初始时假定所有变量均未被赋值。
    value.assign(n, -1);
    for (int i = 1; i <= cscc; i++) {
        // 赋值原则：尽量为同一强连通分支内的变量赋予真值。
        int boolean = 1;
        // 检查是否存在冲突。存在冲突是指在之前已经为此变量赋值，且其值为假，或者从此变量
        // 出发的边所到达的变量其取值为假，根据对偶性，有向边连接的顶点其取值应该相同。
        for (auto u : components[i]) {
            if (getValue(u) == 0) boolean = 0;
            for (auto v : g[u])
                if (getValue(v) == 0) {
                    boolean = 0;
                    break;
                }
            // 存在冲突则表明只能为当前强连通分支内的顶点取假值。
            if (boolean == 0) break;
        }
        // 根据取值为该强连通分支中的顶点赋值。注意同一变量所对应的两个顶点其取值相反。
        for (auto u : components[i])
            if (u & 1) value[u ^ 1] = !boolean;
            else value[u] = boolean;
    }
}
//-----9.4.10.cpp-----

```

如果题目描述中提示某个状态或者某个值只能够取两种值的一种，那么往往提示此问题和 2-SAT 有关。对于该类题目，由于算法是相对不变的，难点在于如何从给定的题目描述中推断出其底层模型为 2-SAT 问题，进而将限制条件转化为有向边。出题者一般会将限制条件予以隐藏，解题者需要仔细分析题意得出限制关系，进而将其转化为布尔表达式。有时还需对初步得到的布尔表达式实施进一步的转换，从而得到期望的合取范式。以下列出常见的几种约束形式及对应的建图方式。

(1) 对于 b_1 和 b_2 ，两者必须一真一假，可推导出布尔表达式 $(b_1 \vee b_2) \wedge (\neg b_1 \vee \neg b_2)$ ，则建图时，从 $\neg b_1$ 到 b_2 、从 $\neg b_2$ 到 b_1 、 b_1 到 $\neg b_2$ 、 b_2 到 $\neg b_1$ 建立有向边。也就是说，选择了 $\neg b_1$ ，即 b_1 为假，就必须选择 b_2 ，即 b_2 必须为真，其他可类似推知其含义。

(2) 对于 b_1 和 b_2 , 两者不能同时为真, 可推导出布尔表达式 $(\neg b_1 \vee \neg b_2)$, 则建图时, 从 b_1 到 $\neg b_2$ 、从 b_2 到 $\neg b_1$ 建立有向边。若两者不能同时为假, 可推导出布尔表达式 $(b_1 \vee b_2)$, 则建图时, 从 $\neg b_1$ 到 b_2 、从 $\neg b_2$ 到 b_1 建立有向边。

(3) 对于 b_1 和 b_2 , 两者同时为真, 或同时为假, 可推导出布尔表达式 $(\neg b_1 \vee b_2) \wedge (b_1 \vee \neg b_2)$, 则建图时, 从 b_1 到 b_2 、从 $\neg b_2$ 到 $\neg b_1$ 、 $\neg b_1$ 到 $\neg b_2$ 、 b_2 到 b_1 建立有向边。

(4) 对于 b_1 和 b_2 , b_1 为真时 b_2 不能为假, 可推导出布尔表达式 $(\neg b_1 \vee b_2)$, 则建图时, 从 b_1 到 b_2 、从 $\neg b_2$ 到 $\neg b_1$ 建立有向边。若 b_1 为假时 b_2 不能为真, 可推导出布尔表达式 $(b_1 \vee \neg b_2)$, 则建图时, 从 $\neg b_1$ 到 $\neg b_2$ 、 b_2 到 b_1 建立有向边。

(5) 对于 b_1 , b_1 必须取真, 可推导出布尔表达式 $(b_1 \vee b_1)$, 则建图时, 从 $\neg b_1$ 到 b_1 建立有向边。若 b_1 必须取假, 可推导出布尔表达式 $(\neg b_1 \vee \neg b_1)$, 则建图时, 从 b_1 到 $\neg b_1$ 建立有向边。

强化练习: 1146 Now or Later^D, 1391 Astronauts^D, 11294 Wedding^D。

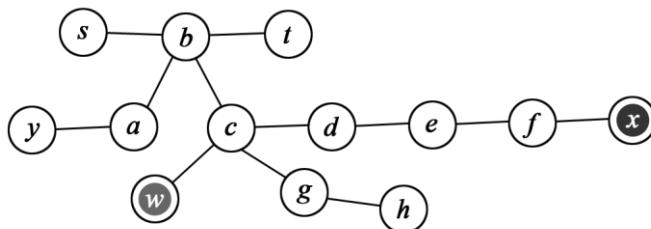
扩展练习: 359* Sex Assignments and Breeding Experiments^E, 11930 Rectangles^E。

9.4.11 图的直径

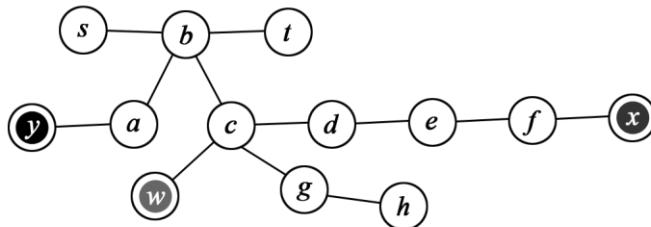
设 G 是无向图, 顶点 $x, y \in V(G)$, G 中所有 (x, y) 路的最短长度称为从 x 到 y 的距离 (distance), 记为 $d_G(x, y)$, 长度等于距离的路称为最短路 (shortest path), 若 G 中不存在 (x, y) 路, 则约定 $d_G(x, y) = \infty$ 。 G 的直径 (diameter) 定义为

$$d(G) = \max\{d_G(x, y) : \forall x, y \in V(G)\}$$

对于有向图, 可按照上述方式类似地定义其距离与直径。简单来说, 图的直径就是图中的最长简单路径。对于一般图来说, 需要使用后续介绍的 Floyd-Warshall 算法确定所有点对间的最短距离, 然后从中选出具有最大距离的最短路径, 时间复杂度为 $O(V^3)$, 当图中顶点数量较多时无法有效计算。



(a) 从 w 开始, 通过第一次 DFS (或 BFS) 能够到达的最远顶点为 x

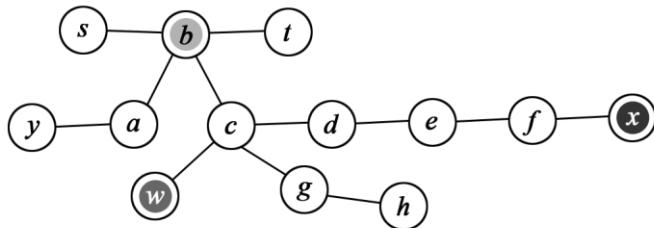


(b) 从 x 开始, 通过 DFS (或 BFS) 能够到达的最远顶点为 y , 则树的直径就是 x 和 y 之间的最短路径

图9-21 从任意一个顶点开始, 通过两次DFS(或BFS)确定树的直径

如果给定的图是树, 则存在 $O(V)$ 的解决方法。如图 9-21 所示, 从树的任意一个顶点 w 出发, 使用 DFS(或 BFS) 确定能够到达的最远顶点 x , 需要注意, 此时的最远顶点对 (w, x) 可能还不是树的最远顶点对, 需要再从最远顶点 x 开始再进行一次 DFS(或 BFS), 确定能够到达的最远顶点 y , 此时的顶点对 (x, y) 就是树的最远顶点对, 顶点对 (x, y) 之间的最短路就是树的直径。上述方法对于加权树也同样有效。

为什么上述方法可行呢? 下面给出典型情形的论证, 其他特殊情形可以使用类似的方法予以论证。容易知道, 从起始顶点 w 到 x 的最短路径与图的直径的关系只有两种, 相交或者不相交。我们首先说明 “ w 到 x 的最短路径与图的直径不相交” 这种情况不可能发生。

图9-22 假设顶点 w 与 x 的最短路径与树的直径不相交

这可以通过反证法予以证明。如果 w 到 x 的路径与直径没有交点, 设直径的两个端点为 s 和 t , 如图 9-22 所示, 令 w 与 t 的最短路径与直径相交于顶点 b , 根据假设, 顶点 x 相较于顶点 t 距离顶点 w 不会更近, 故有

$$d(w, c) + d(c, x) = d(w, x) \geq d(w, t) = d(w, c) + d(c, b) + d(b, t)$$

即

$$d(c, x) \geq d(c, b) + d(b, t) > d(b, t)$$

不等式两边同时加上 $d(s, b)$, 有

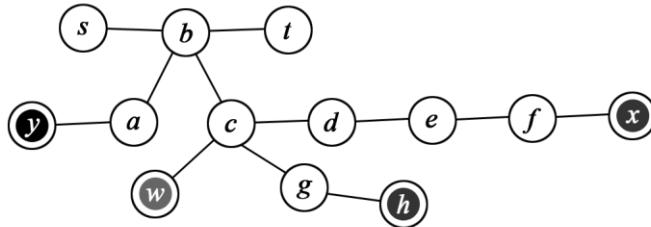
$$d(s, b) + d(c, x) > d(s, b) + d(b, t) = d(s, t)$$

而

$$d(s, x) = d(s, b) + d(b, c) + d(c, x) > d(s, b) + d(b, t) > d(s, t)$$

亦即 s 和 x 之间最短路的长度比直径还要大, 与 s 和 t 是直径的两个端点相矛盾, 故假设不成立, 即 w 不在直径上, 则 w 到 x 的路径与直径必有交点。

下面证明, 若 w 与 x 的最短路径与直径相交, 则 x 必定是直径的一个端点。若不然, 如图 9-23 所示, 假设直径的端点不为顶点 x 而是顶点 h , 根据前述论证, 顶点 w 到顶点 x 的最短路径必定与直径相交, 令交点为顶点 c , 由于顶点 x 是从顶点 w 出发距离顶点 w 最远的顶点, 则从直径的另外一个端点 y 出发, 经过交点 c , 再到达 x 的最短路长度要比假设的直径——顶点 y 和顶点 h 之间的最短路长度——不会更短, 这与假设的顶点 x 不是直径的端点而顶点 h 是端点产生矛盾。

图 9-23 假设顶点 w 与 x 的最短路径与树的直径相交

根据前述推导, x 必定是直径的一个端点, 同理可证第二次 DFS (或 BFS) 确定的顶点 y 是直径的另外一个端点, 那么显然顶点对 (x, y) 之间的距离就是直径。

与图的直径密切相关的是图的中心 (central vertex), 图的中心是指与图中其他顶点最短距离的最大值最小的顶点。可以证明, 图的中心必定位于直径所在路径上。对于树来说, 如果将边权定为 1, 则树的中心在求出其直径后就能立即确定, 树的中心即为直径所在路径上的中间顶点, 如果直径大小为偶数, 则有两个中心, 如果为奇数, 则只有一个中心。

强化练习: [10308 Roads in the North^C](#), [10459 The Tree Root^C](#)。

扩展练习: [11695* Flight Planning^D](#), [12379* Central Post Office^D](#)。

9.4.12 树的重心

给定一棵具有 n 个结点的无根树, 将其中的任意一个结点 u 作为根, 可以构成一棵以 u 为根的有根树。将该有根树以 u 为分界点分为若干棵子树, 设 v 为 u 的子结点, 令 v 所在子树的结点数为 $s_u(v)$, 进一步令

$$S_u = \max\{s_u(v)\}, v \text{ 是 } u \text{ 的子结点}$$

即 S_u 表示以 u 为根的最大子树所具有的结点数, 再令

$$S_{\min} = \min\{S_i\}, 1 \leq i \leq n$$

如果某个结点 x 满足 $S_x = S_{\min}$, 则将结点 x 称为该无根树的重心。换句话说, 计算以无根树每个结点为根结点时的最大子树大小, 将具有最小值的结点称为无根树的重心。

树的重心具有以下性质:

(1) 某个结点 x 是树的重心等价于以该结点为根的最大子树大小不大于整棵树大小的一半, 即若结点 x 满足

$$S_x \leq \frac{n}{2}$$

则结点 x 是重心。

(2) 一棵树至少有一个重心, 至多有两个重心。如果树有两个重心, 则两个重心相邻, 即两个重心之间有直接边相连, 而且此时树一定包含偶数个结点。

(3) 令 $d_u[i]$ 表示树中结点 i 与结点 u 之间最短路径的长度 (即结点 i 和结点 u 之间简单路径的边数), 定义

$$D_u = \sum_{i=1}^n d_u[i]$$

则对于树的重心 x 来说, D_x 最小, 即有

$$D_x = D_{\min} = \min\{D_i\}, 1 \leq i \leq n$$

若树有两个重心 x 和 y ，则 $D_x = D_y$ 。本性质的逆命题也成立，即对于某个结点 x 来说，如果 D_x 最小，则结点 x 是树的重心。

(4) 在一棵树上添加或删除一个叶子结点，其重心最多平移一条边的距离。

(5) 将两棵树通过连接一条边组合成一棵新树，则新树的重心在原来两棵树重心的简单路径上。

根据定义，可以通过一遍 DFS 求出树的重心，其时间复杂度为 $O(n)$ 。

```
-----9.4.12.cpp-----
const int MAXN = 10010, INF = 0x7fffffff;

// n 为树的结点数, bestU 记录单个重心。
int n;
int d[MAXN], s[MAXN];
int bestU, bestD = INF;

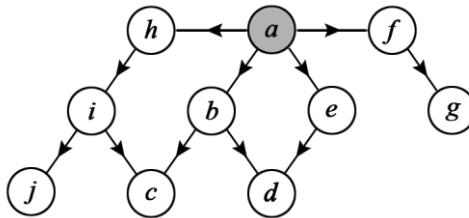
// edges 为边列表, center 记录所有重心。
vector<int> edges[MAXN], center;

void dfs(int u, int father) {
    s[u] = 1, d[u] = 0;
    for (auto v : edges[u]) {
        if (v == father) continue;
        dfs(v, u);
        s[u] += s[v];
        d[u] = max(d[u], s[v]);
    }
    d[u] = max(d[u], n - s[u]);
    // 根据定义确定树的一个重心。
    if (d[u] < bestD) bestD = d[u], bestU = u;
    // 根据性质(1)确定树的所有重心。
    if (d[u] <= n / 2) center.push_back(u);
}
-----9.4.12.cpp-----
```

9.5 拓扑排序

在有向无圈图中，将所有顶点排成一列，使得所有有向边在序列中都是从左指向右的操作称为拓扑排序 (topological sort)。拓扑排序经常用来为给定的活动 (activity) 确定完成先后次序。

在无向图中，查找圈可以仅使用单纯的 DFS，若在 DFS 过程中遇到已经访问的顶点，则表明图中存在无向圈。但是对于有向图来说，使用单纯 DFS 来查找圈的方法，并不总是能够得到正确的结果。如图 9-24 所示，使用 DFS，会重复访问顶点 c (或顶点 d)，会得到存在有向圈的错误结论。关键在于有向图中的边存在方向性，顶点 u 和顶点 v 之间存在有向边并不代表顶点 v 和顶点 u 之间存在有向边。正确的方法是在 DFS 过程为顶点标记颜色，初始时所有顶点为白色，当顶点被发现时着色为灰色，当顶点完成访问时着色为黑色，如果在遍历过程中遇到了着色为灰色的顶点，表明存在有向圈。

图 9-24 从顶点 a 开始简单 DFS，在第二次遇到到顶点 c （或顶点 d ）时会得出存在有向圈的错误结论

拓扑排序有两种常见的的实现方法，一种是根据 DFS 过程中访问顶点的完成时间来实现，另外一种方法是根据入度的性质来实现。

第一种方法根据 DFS 过程中访问顶点的完成时间来确定顶点在拓扑排序中的位置。如果有向图不存在圈，在 DFS 过程中，不会发生遇到标记为灰色顶点的情况（但如果发生这种情况，根据深度优先遍历的顶点着色性质，表明图中就存在有向圈），每个顶点在遍历过程中的完成时间都不同，越靠后完成的顶点在拓扑排序排序中应该越靠前。以下给出使用 DFS 确定拓扑排序的代码框架，其时间复杂度为 $O(V+E)$ 。在记录拓扑排序时，可以使用 STL 的 `vector` 或 `stack` 数据结构。

```
-----9.5.cpp-----
const int MAXV = 110;

int visited[MAXV];
vector<list<int>> g(MAXV);
stack<int> ts;

// 将 DFS 过程中发现的顶点按遍历完成时间入栈，从栈顶到栈底的顺序即为一种可行的拓扑排序。
void dfs(int u) {
    visited[u] = 1;
    for (auto v : g[u])
        if (!visited[v])
            dfs(v);
    ts.push(u);
}

void topologicalSort() {
    // 反复调用 DFS 过程，直到所有顶点为已访问状态。
    while (!ts.empty()) ts.pop();
    memset(visited, 0, sizeof visited);
    for (int u = 1; u <= n; u++)
        if (!visited[u])
            dfs(u);
}
-----9.5.cpp-----
```

第二种方法是根据入度的性质来实现拓扑排序，称为 Kahn 算法^[90]。对于每个顶点，计数其入度，然后找到一个入度为 0 的顶点，如果不存在这样的顶点，则必定存在圈。因为入度为 0 的顶点没有其他顶点的有向边进入，表明可以将此顶点放在拓扑排序的首位，然后将该顶点的有向边所连接的顶点的入度分别减去 1，再寻找下一个入度为 0 的顶点，放到拓扑排序的第二位，依此循环，直到将所有顶点放入拓扑排序序列中，如果尚存在不能放入排序的顶点，则表明图中存在圈。在寻找入度为 0 的顶点过程中，如果同时有多个入度为 0 的顶点，则有多种不同的拓扑排序，顶点选择的先后顺序不同会产生不同的拓扑排序。下面结合一

一道例题给出 Kahn 算法的代码实现。

200 Rare Order^A (珍藏排序)

一位珍藏本书籍收藏家最近发现了一本书，它用一种陌生的语言写成，所使用的字母表与英文字母表相同。该书包含一份简短的索引，但在索引中的项目并未按照正常英文字母表的顺序进行排列。这位收藏家尝试通过索引来确定这种陌生语言的字母表顺序，但是他很快由于这件事枯燥乏味而以沮丧告终。

现在要求你来完成收藏家的任务。确切地说，你的程序需要读入一组字符串，这组字符串是根据某种特定的字母表顺序进行排列的，你需要根据给定的字符串，确定其所依据的字母表顺序。

输入

输入包含多个以大写字母构成的字符串列表，每行一个字符串，列表中的字符串已排序。每个字符串不超过 20 个字母。每个列表以只包含字符 ‘#’ 的一行作为结束标记。列表中出现的字母并不需要全部使用，但是每个列表都会暗含所用字母确定的相对顺序。

输出

为每个字符串列表生成一行输出，输出由大写字母构成，按照生成此字符串列表的字母表顺序进行输出。

样例输入

```
XwY
ZX
ZXY
ZXW
YwWwX
#
```

样例输出

```
XZYW
```

分析

对于每个字符串列表，依次比较相邻的两个字符串。在比较两个字符串时，从字符串的第一位字母开始比较，如果相同则比较后一位字母，如果不同，则得到一对字母的相对顺序。把每个字母看成一个顶点，则一对字母的相对顺序相当于在两个顶点间建立了一条有向边，将所有依次比较得到的字母相对顺序进行转换，最后得到的是一个有向图，求字母表顺序相当于对图进行拓扑排序。在构建有向图的同时为顶点计数入度，使用队列存储入度为 0 的顶点，然后将队列中顶点所邻接的其他顶点的入度减去 1，再次寻找入度为 0 的顶点放入队列中，直到队列为空。

参考代码

```
const int MAXV = 26;

vector<int> g[MAXV];
int degreeOfIn[MAXV], visited[MAXV];

int main(int argc, char *argv[]) {
    string word; vector<string> words;
    // 读入索引中的单词。
    while (getline(cin, word)) {
        if (word != "#") {
            words.push_back(word);
            continue;
        }
        // 构建有向图。
```

```

for (int u = 0; u < MAXV; u++) g[u].clear();
memset(degreeOfIn, 0, sizeof(degreeOfIn));
memset(visited, 0, sizeof(visited));
for (int i = 0; i < words.size() - 1; i++) {
    int t = min(words[i].length(), word[i + 1].length());
    for (int j = 0; j < t; j++) {
        int u = words[i][j] - 'A', v = words[i + 1][j] - 'A';
        visited[u] = visited[v] = 1;
        if (u != v) {
            g[u].push_back(v);
            degreeOfIn[v]++;
            break;
        }
    }
}
// 将入度为 0 的顶点置入队列，删除出边，反复寻找入度为 0 的顶点。
queue<int> q;
for (int u = 0; u < MAXV; u++)
    if (visited[u] && degreeOfIn[u] == 0)
        q.push(u);
while (!q.empty()) {
    int u = q.front(); q.pop();
    for (auto v : g[u])
        if (--degreeOfIn[v] == 0)
            q.push(v);
    cout << (char)('A' + u);
}
cout << '\n';
words.clear();
}
return 0;
}

```

强化练习：[452 Project Scheduling^C](#)，[10305 Ordering Tasks^A](#)，[11060 Beverages^A](#)。

扩展练习：[192 Synchronous Design^E](#)，[10672 Marbles on a Tree^C](#)。

9.6 小结

在对图进行处理之前，首先需要以一定的方式来表示图，在表示图的几种方式中，链式前向星的效率最高，但不是说其他的几种表示方式毫无优点，它们各自在特定的应用场景能够发挥优势。例如，在 Kruskal 算法中使用边列表方式来表示图对实现算法来说非常简便。

图遍历是对图进行进一步处理前的标准操作。图遍历有两种常见的形式，一种是广度优先遍历（Breath First Search, BFS），另外一种是深度优先遍历（Depth First Search, DFS）。BFS 就是在遍历当前顶点时，先尽可能“广”地遍历与该顶点连接的顶点，类似于剥洋葱，一层一层的剥除，而 DFS 则是从一个顶点出发，尽可能“深”地沿着图中的边走，直到遇到已经访问的顶点才停止，之后进行回溯，沿着另外尚未访问的边前进，类似于在洋葱上钻孔，从表层直达核心，然后回退，从另外一个地方再开始钻孔。

之前介绍的回溯法，实际就相当于在问题所对应的隐式图中进行 DFS。对树进行的前序遍历、中序遍历、后序遍历都是 DFS，而层序遍历则属于 BFS。BFS 一般使用队列来实现。DFS 一般使用栈来实现，由于递归属于一种隐式栈，故在编程竞赛中，DFS 以递归的形式出现较为常见。

BFS 和 DFS 有各自的应用场景。一般来说，BFS 较常用于解决最短路径问题，即从给定顶点出发，到

达指定的终止顶点，最少需要经过多少条边。DFS 一般用于解决连通性问题，即给定的两个顶点之间是否具有路径连通。

BFS 和 DFS 是两种非常重要的搜索算法，从思想上来说，是采用两种不同方式对问题空间进行穷举以得到合适的解。BFS 和 DFS 还是众多算法的基础步骤，因此掌握这两种方法是非常必要的。在掌握两种遍历方法的基础上，重点需要掌握两者的拓展运用，例如强连通分支、割点、割边、2-SAT、图的直径、树的重心、拓扑排序等。

第 10 章 图算法

人有从学者，遇不肯教，而云：“必当先读百遍！”言“读书百遍，**其义自见**”。
——陈寿，《三国志·魏志·**董遇传**》

图论 (graph theory) 主要研究图的结构和性质，是数学的一个重要分支，它为图的讨论提供了相应的数学语言。在图论的发展过程中，形成了许多有效的算法，这些算法的构思非常巧妙，充分理解并灵活运用它们，会为你带来解题的乐趣和成就感。与此同时，你也会折服于发现这些算法的计算机科学先辈们的创造性思维^[91]。

10.1 基本概念

10.1.1 顶点度

设 G 是无向图，顶点 v 的顶点度 (vertex degree) 定义为 G 中与 v 关联边的数量 (环要计算两次)，记为 $d_G(v)$ 。顶点度为 d 的顶点称为 d 度点 (a vertex of degree d)，零度点称为孤立点 (isolated vertex)。

设 D 是有向图，顶点 v 的顶点出度 (vertex out-degree) 定义为 D 中以 v 为起点的有向边的数量。顶点 v 的顶点入度 (vertex in-degree) 定义为 D 中以 v 为终点的有向边的数量，顶点 v 的顶点度为顶点入度和顶点出度的和。若顶点 v 的入度和出度相等，称 v 为平衡点 (balanced vertex)，如果有向图 D 中的每个顶点均为平衡点，称该有向图为平衡有向图 (balanced digraph)。

强化练习：10928 My Dear Neighbours^B。

一般把度数为偶数的顶点称为偶点 (even vertex)，把度数为奇数的顶点称为奇点 (odd vertex)。欧拉^I于 1736 年证明：对有限图 G ，所有顶点度之和为边数的两倍。因为每条边为顶点所贡献的度数均为 2，令有限图 G 的边数为 e ，顶点为 v_1, v_2, \dots, v_n ，各自的顶点度为 d_1, d_2, \dots, d_n ，有

$$d_1 + d_2 + \dots + d_n = 2e$$

由于有限图的边数是一个整数，而所有顶点度之和也是一个整数，由上述关系可知 $d_1 + d_2 + \dots + d_n$ 必定是一个偶数，则 d_1, d_2, \dots, d_n 中必定包含偶数个奇数，亦即任意有限图必定包含偶数个奇点。在图论中，人们将该结论称之为握手引理 (handshaking lemma)。例如，参加宴会的所有人如果互相握手，则与其他人握手次数为奇数的人数必定为偶数。

扩展练习：11393* Tri-Isomorphism^D，12428* Enemy at the Gates^D。

对于有限图 G ，若将图 G 所有顶点的度数排成一列，会构成一个有限非负整数序列，记为 s ，称 s 为图 G 的度序列 (degree sequence)。如果一个由非负整数构成的有限序列 s 满足

$$\sum_{i=1}^n s_i = 2k, \quad 0 \leq s_i, \quad k \in \mathbb{Z}_0^+$$

则称该序列是可图化的 (graphic)，简称可图。若 s 是某个简单图的度序列，则称序列 s 是可简单图化的 (simple graphic)。当给定图 G 后，确定其度序列非常简单，但给定一个由非负整数构成的有限序列 s ，判定其是否为某个简单图的度序列却相对困难。可以应用 Erdős–Gallai 定理予以判定。

^I 莱昂哈德·欧拉 (Leonhard Euler, 1707—1783)，瑞士数学家。

Erdős-Gallai 定理

给定非负整数序列 $d_1 \geq d_2 \geq \dots \geq d_n \geq 0$, 该序列可表示为某个简单图的度序列的充分必要条件是: $d_1 + d_2 + \dots + d_n$ 为偶数 (根据握手引理) 且满足

$$\sum_{i=1}^k d_i \leq k(k-1) + \sum_{i=k+1}^n \min(d_i, k), \quad 1 \leq k \leq n$$

除了应用 Erdős-Gallai 定理来判断图是否可简单图化, 还可使用 Havel-Hakimi 算法进行判定并同步构建满足要求的简单图^[92]。

Havel-Hakimi 算法

由非负整数构成的递减序列 s :

$$d_1, d_2, \dots, d_n, d_i \geq d_{i+1}, \quad n > i \geq 1, \quad n \geq 2, \quad d_1 \geq 1$$

是可简单图化的^I, 当且仅当由序列 s 得到的下列序列 s_1 :

$$d_2 - 1, d_3 - 1, \dots, d_k - 1, d_{k+1} - 1, d_{k+2}, \dots, d_n, k = d_1$$

是可简单图化的。序列 s_1 中有 $n-1$ 个非负整数, 序列 s 中 d_1 后的前 d_1 个度数 (即 $d_2 \sim d_{k+1}, k = d_1$) 减 1 后构成 s_1 中的前 d_1 个数。

例如, 给定序列 “7, 3, 2, 5, 5, 4, 1, 6”, 可按以下步骤判定该序列是否可图:

- (1) 将序列按照递减序排列为: 7, 6, 5, 5, 4, 3, 2, 1。如果序列的个数小于 d_1 , 则该序列不可图。
- (2) 首项为 7, 删除首项后, 将剩余序列前 7 项减 1 后得: 5, 4, 4, 3, 2, 1, 0;
- (3) 此时首项为 5, 删除首项后, 将剩余序列前 5 项减 1 后得: 3, 3, 2, 1, 0, 0;
- (4) 首项为 3, 删除首项后, 将剩余序列前 3 项减 1 后得: 2, 1, 0, 0, 0;
- (5) 首项为 2, 删除首项后, 将剩余序列前 2 项减 1 后得: 0, -1, 0, 0;

此时序列中出现负数, 由于顶点的度不可能为负数, 故上述给定的序列是不可图的。需要注意, 在判定过程中可能需要对序列再次进行排序以保证序列具有递减的性质。

在判定给定的序列可图后, 如何根据顶点度来实际构造一个满足要求的简单图呢? 可以在前述判定过程中同步进行。将顶点度序列按递减序排列后, 记为 d_1, d_2, \dots, d_n , 令度数最大的顶点为 v_1 , 对应的顶点度为 d_1 , 在 v_1 与 d_1 之后的前 d_1 个顶点之间构建边, 则相当于完成了顶点 v_1 的构造, 此时可以删除首项 d_1 , 并把后面的 d_1 个度数减 1, 再把剩余的序列按递减序排列, 继续此过程构建边, 直到构建出完整的图。

强化练习: [10720 Graph Construction^B](#), [11387 The 3-Regular Graph^C](#), [11414 Dream^D](#), [12786 Friendship Networks^D](#)。

10.2 图的回路

10.2.1 欧拉回

对于无向连通图, 欧拉迹 (Eulerian trail) 是指包含图中每个顶点和每条边的链, 直观地说就是访问图中所有边一次并且仅一次的一条通路^{II}。特别地, 如果欧拉迹的起点和终点相同, 则称为欧拉回 (Eulerian

^I 此处“可简单图化的”的含义是序列 s 是某个简单图的度序列。简单图是指不包含平行边或自环的连通图。

^{II} 关于图论概念“迹”、“链”、“回”的定义, 请读者参阅本书第 9 章“图遍历”的内容。

circuit)。对于有向图, 如果其基图是连通的, 则称经过有向图的每条边一次并且仅一次的有向链为有向欧拉迹, 如果有向欧拉迹的起点和终点相同, 则称之为有向欧拉回。将包含欧拉回的图称为欧拉图 (Eulerian graph), 将只包含欧拉迹的图称为半欧拉图 (semi-Eulerian graph)。

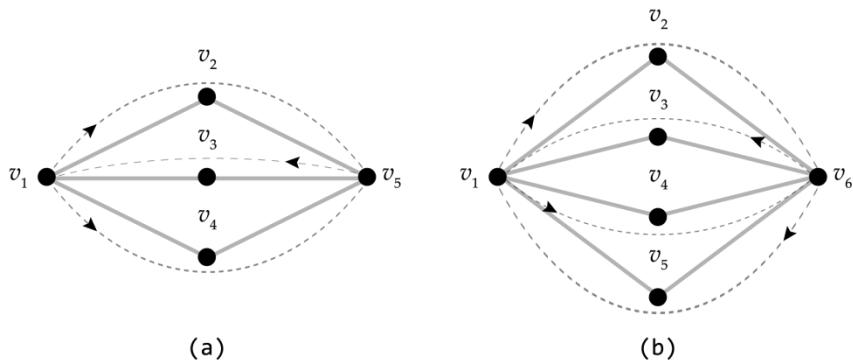


图 10-1 (a) 欧拉迹: $v_1-v_2-v_5-v_3-v_1-v_4-v_5$, 该图为半欧拉图。(b) 欧拉回: $v_1-v_2-v_6-v_3-v_1-v_4-v_6-v_5-v_1$, 该图为欧拉图

可以证明, 对于无向连通图 G , 如果 G 中只有两个奇点或者不存在奇点, 则 G 存在欧拉迹; 如果 G 所有顶点均为偶点, 则存在欧拉回¹。进一步可以推导出:

¹ 1735 年, Euler 向圣彼得堡 (Saint Petersburg) 科学院提交了一篇题为“有关位置几何的一个问题的解 (The solution of a problem relating to the geometry of position)” 的论文, 对哥尼斯堡 (Königsberg) 七桥问题进行了讨论并给出了最终结论: “如果通奇数座桥的地方不止两个, 则满足要求的路线是找不到的。然而, 如果只有两个地方通奇数座桥, 则可以从这两个地方之一出发, 找出所要求的路线。最后, 如果没有一个地方是通奇数座桥的, 则无论从哪里出发, 所要求的路线总能实现。”然而, Euler 只是说明了欧拉迹存在条件的必要性, 并未证明其充分性 [1][2][3]。第一个完整的证明由德国数学家 Hierholzer 给出, 但 Hierholzer 在将工作成果正式予以发表前于 1871 年不幸去世 [4]。在去世前不久, Hierholzer 曾向他的数学家同行展示了该证明, 后来由 Wiener 予以整理, 在 1873 年以“论不重复且不间断地走遍一个线系的可能性”为题公开发表 [5][6]。在论文中, Hierholzer 证明了以下结论: 给定一个连通图, 该图存在欧拉迹的充分必要条件是图中的奇度点不超过两个 (根据握手引理, 图中的奇度点数量必定为偶数, 则奇度点要么为 0 个, 要么为 2 个。若为 0 个, 该连通图具有欧拉回, 否则, 该连通图只有欧拉迹)。除此之外, Hierholzer 还提出了一种巧妙的方法来构建欧拉回, 人们将之称为 Hierholzer 算法。

参考:

- [1] https://en.wikipedia.org/wiki/Leonhard_Euler [OL], 2020.
- [2] <http://eulerarchive.maa.org/pages/E053.html> [OL], 2020.
- [3] Euler L. Solutio problematis ad geometriam situs pertinentis [J]. Commentarii Academiae Scientiarum Petropolitanae, 1741, 8: 128-140. 英译文见 [7]: 3-8. 中译文见 [8]: 617-624.
- [4] https://en.wikipedia.org/wiki/Carl_Hierholzer [OL], 2020.
- [5] https://gdz.sub.uni-goettingen.de/id/PPN235181684_0006 [OL], 2020.
- [6] Hierholzer C, Wiener C. Ueber die Möglichkeit, einen Linienzug ohne Wiederholung und ohne Unterbrechung zu umfahren [J]. Mathematische Annalen, 1873, 6: 30-32. 英译文见 [7]: 8-11.
- [7] Biggs N L, Lloyd E K, Wilson R J. Graph Theory 1736-1936 [M]. Oxford: Clarendon Press, 1986.

(转到下一页)

- (1) 如果图 G 是只有两个奇点的连通图，则 G 的欧拉迹必定以此两个顶点为端点；
- (2) 当图 G 不存在奇点时， G 必有欧拉回；
- (3) G 中存在欧拉回的充分必要条件是 G 为无奇点的连通图。

对于有向图 D ，如果 D 的基图连通，而且所有顶点的出度与入度相等，或者除了两个顶点外，其余顶点的出度与入度都相等，而这两个顶点中的一个顶点的出度与入度之差为 1，另一个顶点的出度与入度之差为 -1，则有向图 D 存在欧拉迹。类似的，可以推导出：

- (1) 当 D 除了出、入度之差为 1 和 -1 的两个顶点外，其他顶点的出度与入度均相等，那么 D 的有向图欧拉迹必定以出、入度之差为 1 的顶点作为起点，以出、入度之差为 -1 的顶点作为终点；
- (2) 当 D 的所有顶点其各自的出、入度相等时， D 中存在有向欧拉回；
- (3) 有向图 D 为有向欧拉图的充分必要条件是 D 的基图为连通图，且所有顶点各自的出、入度相等。

利用上述结论判断给定图是否存在欧拉迹（回）相对简单，其关键在于如何按照题目约束构造图中的顶点和边，这一环节在解题过程中相对困难，往往是题目设置者的考察所在。

10129 Play on Words^A (单词游戏)

在进行考古发掘时，有些墓室的暗门上会附带有趣的字谜，考古学家们必须先解开这些字谜才能打开暗门，因为没有其他的方法能够打开暗门，因此这些字谜显得尤为重要。

每道暗门上都有许多带有磁性的盘片，每个磁盘上有一个单词。现在需要将这些磁盘重新排列，使得磁盘上单词的第一个字母是上一个单词的最后一个字母。例如，单词“motorola”可以接在单词“acm”之后。你的任务是编写程序，读入给定的单词，确定是否可以将磁盘进行重新排列，使得磁盘上的单词满足上述要求，从而能够打开暗门。

输入

输入包含 T 组测试数据。数值 T 在输入的第一行给出。每组测试数据的第一行为一个整数 N ，表示磁盘的数量， $1 \leq N \leq 100000$ ，之后是 N 行数据，每行包含一个单词。每个单词至少包含 2 个，至多包含 1000 个小写字母（也就是说只有 ‘a’ 到 ‘z’ 的字母会出现在单词中）。相同的单词可能会出现多次。

输出

你的程序需要确定是否存在一种方法来重新排列所有磁盘，使得磁盘上的单词的第一个字母是上一个单词的最后一个字母。磁盘上的所有单词都必须使用且只使用一次。重复出现多次的同一个单词的使用次数必须达到其出现的次数。假如存在满足要求的排列方案，打印语句 `Ordering is possible.`，否则打印语句 `The door cannot be opened..`。

样例输入

```
1
2
acm
ibm
```

样例输出

```
The door cannot be opened.
```

分析

如果将单词本身视为图的顶点进行建模，问题转化为在图中寻找一条链，该链经过所有顶点一次且仅一

次，也就是后续将要介绍的哈密顿路问题。由于哈密顿路问题不易解决，需要换一个角度考虑问题。在满足要求的排列方案中，某个单词的首字母是前一个单词的尾字母，其尾字母又是下一个单词的首字母，那么可以将某个单词视为在 26 个字母的某两个字母间（这两个字母可能相同）的一条边，例如样例输入中的单词“acm”，相当于在字母‘a’和字母‘m’之间构成了一条边，则问题可以转化为能否从图中找出一条欧拉迹。从这个角度构建图，图中的顶点最多只有 26 个，处理起来更为简便和高效。

在解题中还需要注意以下几个方面：(1) 由于磁盘排列方案中，要求前一个单词的尾字母是下一个单词的首字母，因此构建得到的图是一个有向图，需要使用有向图欧拉迹的判断规则。(2) 所有 26 个字母可能并不会在图中都出现，只需要对出现的字母顶点做相应的出入度检查。(3) 在确定有向图是否具有欧拉迹之前，需要判定其基图是否连通，这个操作可以通过对基图进行一次深度优先遍历来完成，不过更为常见的做法是使用并查集来完成基图连通性的判断：在构建图的过程中，只要两个顶点间具有有向边，则将其合并到同一个集合中，在图建立完毕时，检查图中所有的顶点是否都在同一个集合中，为否则表明基图不连通，不可能存在有向欧拉迹。

参考代码

```
// 为了节省篇幅，省略了并查集相关的实现代码。
// 读者可以参阅本书第 2 章“数据结构”第 2.12 小节“并查集”中的相关内容。
int main(int argc, char *argv[]) {
    // 变量声明。
    int cases, n;
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        cin >> n;
        // letterUsed 记录单词的首尾字母；inDegree 记录入度；outDegree 记录出度。
        int letterUsed[26] = {0}, inDegree[26] = {0}, outDegree[26] = {0};
        string word;
        // 初始化并查集。
        makeSet();
        // 构建有向图，记录各顶点的出入度。
        for (int i = 1; i <= n; i++) {
            cin >> word;
            int u = word.front() - 'a', v = word.back() - 'a';
            // 假如两个顶点间存在有向边但不在同一集合中则予以合并。
            if (findSet(u) != findSet(v)) unionSet(u, v);
            letterUsed[u] = letterUsed[v] = 1;
            outDegree[u]++, inDegree[v]++;
        }
        // 判定是否存在有向欧拉迹。
        bool eulerianTrail = true;
        int moreOne = 0, lessOne = 0;
        for (int first = -1, i = 0; i < 26; i++) {
            if (!letterUsed[i]) continue;
            // 检查已经使用的字母是否位于同一集合，为否表明基图不连通，不存在有向欧拉迹。
            if (first == -1) first = i;
            if (findSet(first) != findSet(i)) { eulerianTrail = 0; break; }
            // 检查顶点的出、入度是否符合要求。
            int diff = outDegree[i] - inDegree[i];
            if (abs(diff) >= 2) { eulerianTrail = 0; break; }
            if (diff == 1 && ++moreOne > 1) { eulerianTrail = 0; break; }
            if (diff == -1 && ++lessOne > 1) { eulerianTrail = 0; break; }
        }
    }
}
```

```

    }
    // 检查出、入度相差为 1 的顶点数量是否符合要求。
    if (moreOne != lessOne) eulerianTrail = false;
    // 输出结果。
    if (eulerianTrail) cout << "Ordering is possible.\n";
    else cout << "The door cannot be opened.\n";
}
return 0;
}

```

强化练习: 10203 Snow Clearing^C, 10596 Morning Walk^C, 11586 Train Tracks^A。

扩展练习: 12118 Inspector's Dilemma^D。

如果给定的图中存在欧拉迹, 如何将其找出呢? 朴素的方法是应用深度优先遍历来寻找欧拉迹。其基本思想为: 选择一个正确的起始顶点, 使用深度优先遍历算法遍历所有的边, 在遍历过程中确保每条边只遍历一次, 中途遇到已经访问的边则回退, 在遍历的同时将访问的边按照顺序予以记录, 最后记录的边顺序就构成了一条欧拉迹。使用此方法, 其递归深度与顶点所关联的边数量及总的边的数量有关, 一般只适用于图中的边数较少的情况, 当边数较大时, 容易导致递归栈溢出。

强化练习: 302 John's Trip^B。

Fleury 算法

相较于朴素的方法, 还有一种更为巧妙的方法可以找出图中存在的欧拉迹 (回), 这就是下面要介绍的 Fleury 算法。Fleury 算法可以概括为以下三个步骤:

(1) 如果给定图只存在欧拉迹, 则选取符合要求的奇度顶点 v_0 作为起点; 如果是欧拉图, 则任取图中某个顶点 v_0 作为起点。此时的欧拉迹 (回) 为 $P_0=v_0$;

(2) 假设已经确定了 $P_i=v_0e_1v_1\cdots v_{i-1}e_iv_i$, 取边 $e_{i+1}\in E(G)\setminus\{e_1, e_2, \dots, e_i\}$, 使得边 e_{i+1} 连接了顶点 v_i 和 v_{i+1} , 并且除非无其他的选择, e_{i+1} 不是图 $G_i=G-\{e_1, e_2, \dots, e_i\}$ 的桥;

(3) 当第 2 步不能再执行时, 算法停止, 构造得到的路径 P 是图 G 的欧拉迹 (回)。

算法描述非常简洁, 但是如何将其实现为代码呢? 如果是“漫不经心”地选取边, 即使图中存在欧拉迹 (回), 也可能无法正确地将其找出。如图 10-2 所示, 从顶点 v_0 开始寻找欧拉迹, 如果起始就选择边 e_1 (连接了顶点 v_0 和 v_1), 沿着边前进将最终回到 v_1 , 但由于 e_1 已经访问, 后续将无法继续前进, 将导致 e_5 、 e_6 、 e_7 无法访问, 从而无法得到欧拉迹。

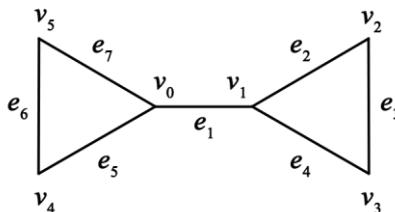


图 10-2 不恰当的选择边会导致无法得到欧拉迹

在 Fleury 算法中, 需要判断选择的边是否为桥, 对于无向图来说, 可以通过一次深度优先遍历后, 检查图的连通性来进行判定, 而对于有向图, 需要判断其基图是否仍旧连通, 使用深度优先遍历不够便利, 可以应用并查集对顶点的连通性进行判断。

以下给出一个找出给定图中欧拉迹 (回) 的 Fleury 算法框架实现。由于欧拉迹 (回) 需要选择所有边,

而每选择一条边的平均时间复杂度是 $O(E)$ ，因此 Fleury 算法总的时间复杂度为 $O(E^2)$ 。

在下述 Fleury 算法框架实现中，图采用边链表进行表示，某个顶点所关联的顶点序号存储在 `list` 中。在实现中，首先判断图的连通性，之后区分无向图和有向图分别统计顶点度；接着按照规则判断是否可能包含欧拉迹（回），如果存在则找出欧拉迹（回）的起始顶点；最后，若存在欧拉迹（回），则从选择的起始顶点出发，应用 Fleury 算法找出。在判断候选边是否为桥时，先将边临时删除，检查删除前后当前顶点所能连通顶点数量是否改变（使用并查集来实现，为了便于调用并查集，在下述代码中将并查集实现为一个类），如果发生改变，说明此边为桥，如果未改变，则可以选择该边作为可行边。确定可行边后，即可将临时删除的边进行恢复，同时记录选择的边，再将其从图中永久删除。删除边的做法是将关联的终止顶点设置为负值，之所以采用此种删除边的方法，是由于算法采用了递归的形式，有可能上一层递归调用仍然在使用当前顶点的边链表，从边链表中移除边会引起运行时错误。

```
-----10.2.1.1.cpp-----
class Graph {
private:
    // 是否为无向图，无向图和有向图在处理上有所不同。
    bool isUndirectedGraph;
    // 图的顶点数，欧拉迹的起始顶点，顶点是否使用。
    int vertices, startOfEulerianTrail, *appeared;
    // 使用边链表表示图。
    list<int> *edges;

public:
    Graph(int v, bool ug) {
        vertices = v;
        edges = new list<int>[v];
        appeared = new int[v];
        memset(appeared, 0, sizeof(int) * v);
        isUndirectedGraph = ug;
    }
    ~Graph() { delete [] edges, appeared; }

    bool findEulerianTrail();           // 寻找图中存在的欧拉迹（回）
    void addEdge(int u, int v);         // 添加边
    void removeEdge(int u, int v);      // 删除边

private:
    bool isEulerian();                 // 判断图中是否包含欧拉迹（回）
    void printTrail(int u, int v);     // 输出欧拉迹（回）的一条边
    void fleury(int u);
    int getConnectedVertices(int u);    // 使用并查集判断图中顶点的连通性
    bool isValidNextEdge(int u, int v); // 判断候选边是否为桥
    void deleteEdge(int u, int v);      // 从图中临时删除一条边
    void restoreEdge(int u, int v);     // 将临时删除的边恢复
};

// 根据顶点度判断图是否包含欧拉迹（回）。
bool Graph::isEulerian() {
    bool eulerian = true;

    // 判断图的连通性。
    int connected = count(appeared, appeared + vertices, 1);
```

```

for (int u = 0; u < vertices; u++)
    if (appeared[u]) {
        eulerian = (connected == getConnectedVertices(u));
        break;
    }
if (!eulerian) return eulerian;

// 无向图和有向图分别处理。
if (isUndirectedGraph) {
    startOfEulerianTrail = 0;
    // 统计奇度顶点的数量。
    int odd = 0;
    for (int u = 0; u < vertices; u++) {
        if (!appeared[u]) continue;
        if (edges[u].size() & 1)
            odd++, startOfEulerianTrail = u;
    }
}

// 根据规则判断是否存在欧拉迹 (回)。
eulerian = (odd == 0 || odd == 2);
} else {
    // 计数有向图中顶点的出度、入度。
    int *id = new int[vertices], *od = new int[vertices];
    memset(id, 0, sizeof(int) * vertices);
    memset(od, 0, sizeof(int) * vertices);
    for (int u = 0; u < vertices; u++)
        for (auto v : edges[u])
            od[u]++, id[v]++;
}

// 根据规则判断是否存在欧拉迹 (回)。
int moreOne = 0, lessOne = 0, evenStart = -1, oddStart = -1;
for (int u = 0; u < vertices; u++) {
    if (!appeared[u]) continue;
    int diff = od[u] - id[u];
    if (abs(diff) >= 2) { eulerian = false; break; }
    if (diff == 1 && ++moreOne > 1) { eulerian = false; break; }
    if (diff == -1 && ++lessOne > 1) { eulerian = false; break; }
    if (moreOne && oddStart < 0) oddStart = u;
    if (diff == 0 && evenStart < 0) evenStart = u;
}
delete [] id, od;
if (moreOne != lessOne) { eulerian = false; }
startOfEulerianTrail = oddStart >= 0 ? oddStart : evenStart;
}

return eulerian;
}

// 寻找图中的欧拉迹 (回)。
bool Graph::findEulerianTrail() {
    bool eulerian = isEulerian();
    if (eulerian) fleury(startOfEulerianTrail);
    return eulerian;
}

// Fleury 算法递归实现。
void Graph::fleury(int u) {

```

```

    for (auto v : edges[u])
        if (v >= 0 && isValidNextEdge(u, v)) {
            printTrail(u, v);
            removeEdge(u, v);
            fleury(v);
            break;
        }
    }

    // 输出欧拉迹 (回) 中的一条边, 可根据具体应用进行修改。
    void Graph::printTrail(int u, int v) {
        cout << u << '-' << v << '\n';
    }

    // 判断给定的边是否为可行的候选边。
    bool Graph::isValidNextEdge(int u, int v) {
        // 如果当前顶点只有一条边, 则根据 Fleury 算法只能选择该边。
        int connected = 0;
        for (auto v : edges[u])
            if (v >= 0)
                connected++;
        if (connected == 1) return true;
        // 判断删除边后顶点连通性是否改变, 未改变表明为非桥边。
        int connected1 = getConnectedVertices(u);
        deleteEdge(u, v);
        int connected2 = getConnectedVertices(u);
        restoreEdge(u, v);
        return connected1 == connected2;
    }

    // 使用并查集判断余图的连通性, 进而判断选择的边是否为桥。
    int Graph::getConnectedVertices(int source) {
        DisjointSet ds(vertices);
        ds.makeSet();
        for (int u = 0; u < vertices; u++)
            for (auto v : edges[u])
                if (v >= 0 && ds.findSet(u) != ds.findSet(v))
                    ds.unionSet(u, v);
        int connected = 0;
        for (int u = 0; u < vertices; u++)
            if (ds.findSet(source) == ds.findSet(u))
                connected++;
        return connected;
    }

    // 为图中添加一条边。
    void Graph::addEdge(int u, int v) {
        appeared[u] = appeared[v] = 1;
        edges[u].push_back(v);
        if (isUndirectedGraph) edges[v].push_back(u);
    }

    // 将边从图中永久删除。
    void Graph::removeEdge(int u, int v) {
        *find(edges[u].begin(), edges[u].end(), v) = -1;
        if (isUndirectedGraph) *find(edges[v].begin(), edges[v].end(), u) = -1;
    }

```

```

}

// 临时删除一条边。
void Graph::deleteEdge(int u, int v) {
    *find(edges[u].begin(), edges[u].end(), v) = -2;
    if (isUndirectedGraph) *find(edges[v].begin(), edges[v].end(), u) = -2;
}

// 将临时删除的边恢复。
void Graph::restoreEdge(int u, int v) {
    *find(edges[u].begin(), edges[u].end(), -2) = v;
    if (isUndirectedGraph) *find(edges[v].begin(), edges[v].end(), -2) = u;
}
//-----10.2.1.1.cpp-----

```

10441 Catenyms^D (首尾相连的单词串)

一个 **catenym** 是由两个单词组成的字符串，中间用点号隔开，而且第一个单词的最后一个字母和第二个单词的第一个字母相同。例如，以下字符串均为 **catenym**：

```

dog.gopher
gopher.rat
rat.tiger
aloha.aloha
arachnid.dog

```

一个复合的 **catenym** 是由三个或更多的单词组成的序列，相邻的单词同样使用点号隔开，而且都构成 **catenym**。例如：

```

aloha.aloha.arachnid.dog.gopher.rat.tiger

```

给定一个由小写字母构成的单词字典，尝试在字典中找出一个复合 **catenym**，使得该 **catenym** 包含字典中的每个单词一次且仅一次。

输入

输入的第一行为一个整数 t ，表示测试数据的组数。每组测试数据的第一行为一个正整数 n ， $3 \leq n \leq 1000$ ，表示字典中单词的数量。接下来共有 n 个不同的单词，每个单词由 1 至 20 个小写字母组成，每个单词独占一行。

输出

对于输入中的每组测试数据，输出字典序最小的复合 **catenym**，包含字典中每个单词一次且仅一次。如果无解，则输出“***”。

样例输入

```

1
6
aloha
arachnid
dog
gopher
rat
tiger

```

样例输出

```

aloha.arachnid.dog.gopher.rat.tiger

```

分析

虽然题目描述中说明字典中“包含 n 个不同的单词”，但是在线测试的数据中却包含重复的单词，也就是说，如果某个单词重复 k 次，则寻找的复合 catenym 必须包含此单词 k 次。解题思路是先判断给定的字典是否存在复合 catenym，如果存在则应用算法将字典序最小的复合 catenym 找出。判断是否存在复合 catenym，可将字母作为顶点，单词看成字母顶点之间的边，使用有向图的欧拉迹判定规则对建立的图进行判断。下述代码利用了前述的 Fleury 算法框架实现，不同的地方是修改了输出边的函数 printTrail，因为题目要求是将字典序最小的复合 catenym 输出，那么要求在选择时，每个单词都是尽可能字典序最小的，因此在处理输入时，先将所有单词排序，接着按单词所连接的字母顶点将“单词边”添加到图中，这样能够保证边链表中添加的边是按照字典序进行排列的，最后算法输出的复合 catenym 也是字典序最小的。

参考代码

```

// dictionary 存储字母顶点间的单词，trail 存储欧拉迹。
vector<string> dictionary[26][26], trail;

// 请读者参考前述给出的 Fleury 算法框架实现。
class Graph {
private:
    bool isUndirectedGraph;
    int vertices, startOfEulerianTrail, *appeared;
    list<int> *edges;

public:
    Graph(int v, bool ug) {
        vertices = v;
        edges = new list<int>[v];
        appeared = new int[v];
        memset(appeared, 0, sizeof(int) * v);
        isUndirectedGraph = ug;
    }
    ~Graph() { delete [] edges, appeared; }

    bool findEulerianTrail();
    void addEdge(int u, int v);
    void removeEdge(int u, int v);

private:
    bool isEulerian();
    void printTrail(int u, int v);
    void fleury(int u);
    int getConnectedVertices(int u);
    bool isValidNextEdge(int u, int v);
    void deleteEdge(int u, int v);
    void restoreEdge(int u, int v);
};

// 对输出边的函数进行了修改以适应题目的需要。
void Graph::printTrail(int u, int v) {
    trail.push_back(dictionary[u][v].front());
    dictionary[u][v].erase(dictionary[u][v].begin());
}

```

```

int main(int argc, char *argv[]) {
    int cases, n;
    string word;
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        cin >> n;
        // 读入单词列表，排序以保证字典序最小。
        vector<string> words;
        for (int i = 0; i < n; i++) {
            cin >> word;
            words.push_back(word);
        }
        sort(words.begin(), words.end());
        // 初始化保存从某个字母顶点到其他字母顶点的“单词边”。
        for (int i = 0; i < 26; i++)
            for (int j = 0; j < 26; j++)
                dictionary[i][j].clear();
        // 构建图。
        Graph g(26, false);
        for (int i = 0; i < n; i++) {
            word = words[i];
            int u = word.front() - 'a', v = word.back() - 'a';
            g.addEdge(u, v);
            dictionary[u][v].push_back(word);
        }
        // 使用 Fleury 算法寻找欧拉迹。
        trail.clear();
        if (!g.findEulerianTrail()) cout << "****";
        else {
            for (int i = 0; i < trail.size(); i++) {
                if (i > 0) cout << '.';
                cout << trail[i];
            }
        }
        cout << '\n';
    }
    return 0;
}

```

Hierholzer 算法

使用 Fleury 算法可以找出图中的一条欧拉迹（回），但其效率不是很高，对于边数量较多的图可能会导致超时，下面介绍更为高效的 Hierholzer 算法¹。

Hierholzer 算法的基本思想：从欧拉图中的任意一个顶点 u 出发，沿着离开 u 的一条边前进，到达另一个顶点，标记已经访问的边不再使用，重复此过程，直到回到顶点 u 本身。在此过程中，不会因为其他原因导致到达了某个顶点 w 而无法回到顶点 u ，因为欧拉图中任意顶点均为偶度点保证了最后必将回到顶点 u ，从而使得得到的路是一条闭迹。令上述闭迹为 t_1 ，由于可能尚未访问图中所有边，此时需要检查当前得到的闭迹 t_1 ，如果闭迹中的某个顶点 v 存在关联边，但其尚未纳入当前闭迹 t_1 ，则从顶点 v 出发，按照前述过程找到另外一条闭迹 t_2 ，然后将 t_2 合并到 t_1 即可构成一个更大的回路。反复进行上述过程，直到图中所

¹ Carl Hierholzer (1840—1871)，德国数学家。

有的边都包含至回路中，此时的回路即为所求的图的欧拉回。

需要说明，Hierholzer 算法也适用于寻找欧拉迹，不同的是需要选择合适的起点，而且在寻找欧拉迹的过程中，可能每次并不是回到当前顶点，但这并不影响最后结果的正确性。在 Hierholzer 算法的实现中，一个难点是如何确定当前的闭迹中哪些顶点仍有边尚未访问，这可以使用栈数据结构来巧妙地解决。在欧拉图中，从任意一个顶点 u 出发，“随意”沿着尚未访问的边前进，已经访问的边予以标记以便不重复进行访问，在访问的过程中，将遇到的顶点压入栈中，那么最后一定会回到顶点 u ，此时顶点 u 所有的边均已访问，可以将其输出，然后从栈顶开始退栈，检查栈顶的顶点，如果该顶点的所有边均已访问，则可以直接输出，否则继续从该顶点开始访问尚未访问的边，重复此过程，直到栈为空，此时输出的顶点序列即构成一个欧拉回路。

以下给出 Hierholzer 算法的一种实现，该实现采用邻接矩阵的方式来表示图，万维网上很多人错误地认为这是 Fleury 算法的实现，甚至某些正规的教材也未加仔细辨别导致以讹传讹^I。

```
//-----10.2.1.2.cpp-----
const int MAXV = 1010;           // 图中最大顶点数量

int stk[MAXV], top;           // 使用数组模拟栈，用于保存遍历过程所经过的顶点
int connected[MAXV][MAXV];    // 使用邻接矩阵表示图
int n;                         // 顶点的数量
int m;                         // 边的数量

void dfs(int x) {
    // 使用深度优先遍历遍历边，将遇到的顶点入栈，选择当前顶点的一条出边，删除此边。
    stk[top++] = x;
    for (int i = 0; i < n; i++) {
        if (connected[x][i]) {
            connected[x][i] = connected[i][x] = 0;
            dfs(i);
            break;
        }
    }
}

void hierholzer(int u) {
    // 将起始顶点压入栈中。
    top = 0;
    stk[top++] = u;
    // 栈顶为当前顶点，若栈不为空，则检查从当前顶点出发是否能够继续沿着尚未访问的边前进。
    while (top > 0) {
        // 检查当前顶点是否存在尚未访问的边。
        int going = 1;
        for (int i = 0; i < n; i++)
            if (connected[stk[top - 1]][i]) {
                going = 0;
                break;
            }
    }
}
```

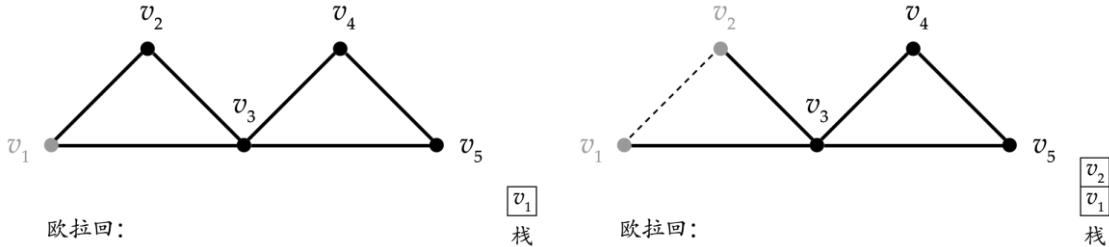
^I 例如王桂平等主编的《图论算法理论、实现及应用》，在此书第 234 页的 Fleury 算法实现实际上是 Hierholzer 算法的实现。

```

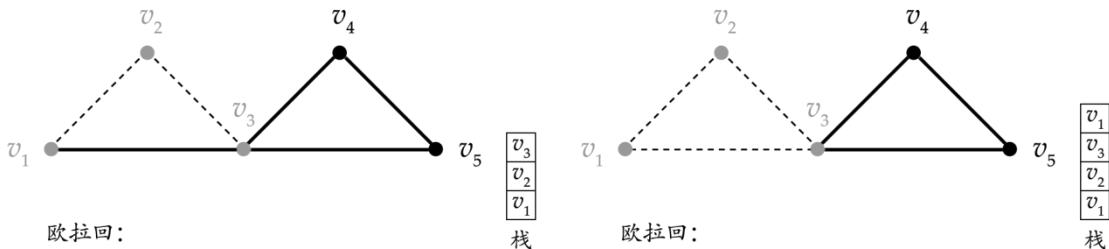
// 当前顶点所有边均已访问, 可以输出。
if (!going) cout << stk[--top] << ' ';
// 当前顶点仍有边尚未访问, 继续使用深度优先遍历沿着未访问的边前进。
else dfs(stk[--top]);
}
cout << '\n';
}
//-----10.2.1.2.cpp-----//

```

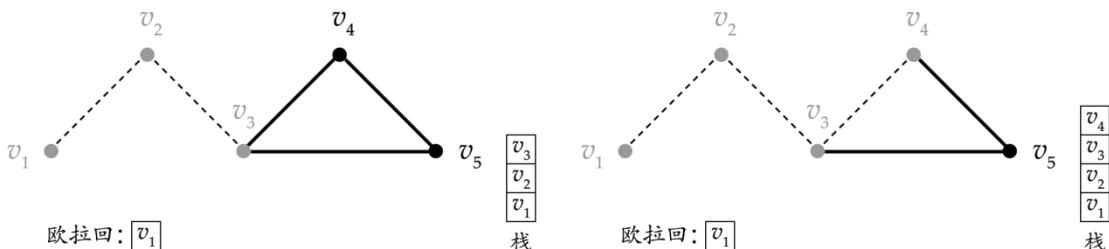
如图 10-3 所示, 这是上述参考实现在给定欧拉图上的具体执行过程, 通过观察该执行过程, 读者可以更为直观地理解 Hierholzer 算法。



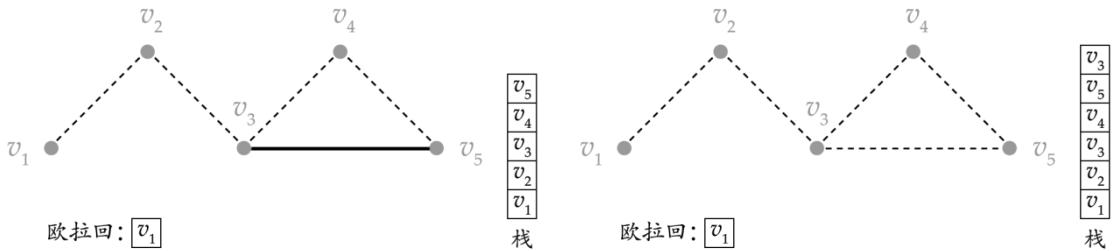
算法执行的第 1 步和第 2 步。第 1 步: 从 v_1 开始构建欧拉回, 将 v_1 压入栈中。第 2 步: 由于 v_1 尚有边未访问, 从尚未访问的边中任意选择一条进行访问, 此处选择 v_1 到 v_2 的边, 到达 v_2 后将 v_2 压入栈中, 并将边 v_1-v_2 标记为已访问 (实线表示尚未访问的边, 虚线表示已访问的边)



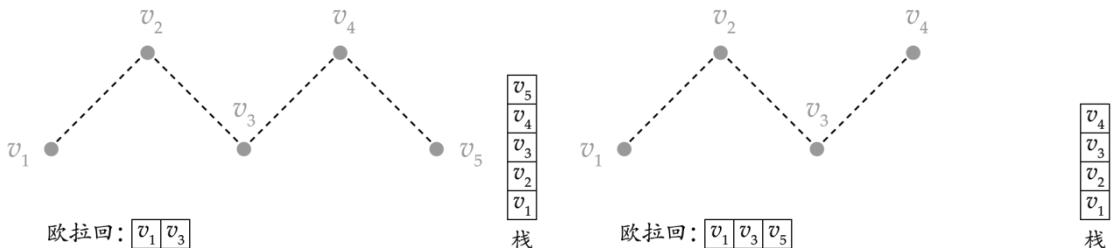
算法执行的第 3 步和第 4 步。第 3 步: 从 v_2 到达 v_3 , 将 v_3 压入栈中, 将边 v_2-v_3 标记为已访问。第 4 步: v_3 有多条边可供选择, 此处为了演示栈的作用, 选择先访问边 v_3-v_1 , 将该边标记为已访问, 然后将 v_1 入栈



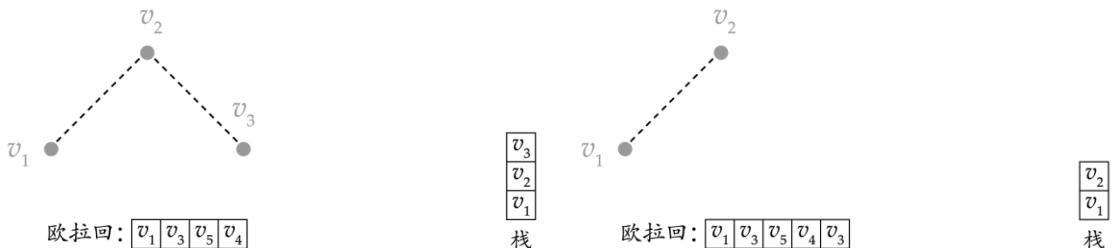
算法执行的第 5 步和第 6 步。第 5 步: 到达 v_1 后, 由于 v_1 的所有边均已访问, 此时可以退栈, 将 v_1 作为欧拉回所经过的顶点予以输出, 此时栈顶为 v_3 。第 6 步: v_3 尚有边未访问, 继续沿边 v_3-v_4 前进, 将 v_4 压入栈中



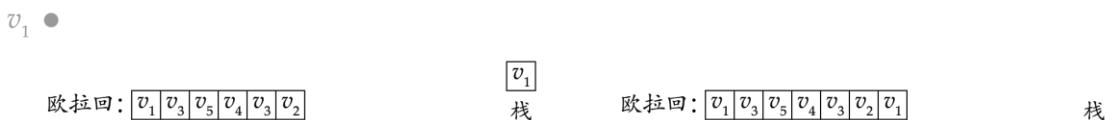
算法执行的第 7 步和第 8 步。第 7 步: 沿边 v_4-v_5 前进, 将 v_5 压入栈中。第 8 步: 沿边 v_5-v_3 前进, 将 v_3 压入栈中



算法执行的第 9 步和第 10 步。第 9 步: 由于 v_3 的所有边已访问, 将 v_3 输出, 回退到 v_5 。第 10 步: v_5 的所有边均已访问, 输出 v_5 , 回退到 v_4



算法执行的第 11 步和第 12 步。第 11 步: 由于 v_4 的所有边已访问, 将 v_4 输出, 回退到 v_3 。第 12 步: v_3 的所有边均已访问, 输出 v_3 , 回退到 v_2



算法执行的第 13 步和第 14 步。第 13 步: 由于 v_2 的所有边已访问, 将 v_2 输出, 回退到 v_1 。第 14 步: v_1 的所有边均已访问, 输出 v_1 , 执行结束

图 10-3 Hierholzer 算法的工作过程

为了提高适用性, 可以将图的邻接矩阵表示更改为邻接表形式, 同时使用 STL 中的 `stack` 容器类来实

现栈的功能。以下实现基于前述的 Fleury 算法框架，可以看到，代码相对简短，省去了 Fleury 算法中的许多步骤。由于 Hierholzer 算法只需对边进行一次遍历，故其时间复杂度为 $O(E)$ 。需要注意，在输出欧拉迹（回）时，由于采用栈的形式保存边，如果在输出欧拉回时有特殊要求，例如，要求按照经过顶点的顺序为字典序且最小，则在实际输出时需要将输出顺序反向，即从最后一条边开始输出欧拉迹（回）。读者可根据需要结合具体应用进行相应的更改。

```
//-----10.2.1.3.cpp-----//
const int MAXV = 1010;

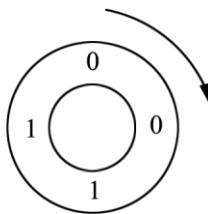
vector<int> edges[MAXV];

void hierholzer(int u) {
    stack<int> path;
    vector<int> circuit;
    int current = u;
    path.push(u);
    // 不断沿可行边前进寻找欧拉回。
    while (!path.empty()) {
        if (edges[current].size()) {
            path.push(current);
            int next = edges[current].front();
            edges[current].erase(edges[current].begin());
            current = next;
        }
        else {
            circuit.push_back(current);
            current = path.top();
            path.pop();
        }
    }
}
//-----10.2.1.3.cpp-----//
```

10040 Ouroboros Snake^D (咬尾蛇)

咬尾蛇是古埃及神话中一种虚构的蛇，它经常把尾巴放在嘴里不停地吞噬自己。环数（Ouroboros number）类似于咬尾蛇，它是一个 2^n 位的二进制数，能够生成 0 至 2^n-1 之间的所有数，方法如下：将给定环数的 2^n 个数位首尾相连形成一个环，使得环数的末位在首位之前，然后以每个数的起始位置的下一位置作为下一个数的起始位置，就可以从中取出 2^n 组 n 位二进制数，对应着 0 至 2^n-1 之间的所有数。

例如，当 $n=2$ 时，只有四个环数，它们是 0011, 0110, 1100 以及 1001。以 0011 为例，下图示例了找出所有位串的过程。



k	$00110011\dots$	$o(n=2, k)$
0	00	0
1	01	1
2	11	3
3	10	2

编写程序，计算函数 $o(n, k)$ 的值， $0 < n$, $0 \leq k < 2^n$ 。 $o(n, k)$ 表示大小为 n 的最小环数中的第 k 个数。

输入

输入包含多组测试数据。输入的第一行为测试数据的组数，接下来每行一组测试数据，每组测试数据包含两个整数 n 和 k ， $0 < n < 22$ ， $0 \leq k < 2^n$ 。

输出

对于每组测试数据，输出计算得到的函数值。

样例输入

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

样例输出

```
0
1
3
2
```

分析

在求解本题之前，有必要介绍 de Bruijn 序列（de Bruijn sequence）。在大小为 k 的字符集 A 上，可以定义阶为 n 的 de Bruijn 序列，该序列是一个环形序列，即可将其末位和首位相连构成一个环，使得在字符集 A 上长度为 n 的所有字符串以子串的形式在环中出现且仅出现一次。一般将此序列记为 $B(k, n)$ ，它的长度为 k^n 。在大多数应用中，字符集 A 取 $\{0, 1\}$ ，按此约束， $B(2, 3)$ 只有两个序列：00010111 和 11101000。以 00010111 为例，从序列的首位开始，每 3 位作为一组，共出现了：000, 001, 010, 101, 011, 111, 110, 100，恰好包含了长度为 3 的由 0 或 1 组成的所有字符串，也恰好包含了 0 至 2^3-1 之间所有数的 3 位二进制表示。de Bruijn 序列和 de Bruijn 图密切相关，构造一个阶为 n 的 de Bruijn 序列实际上可以通过在其对应的 $n-1$ 阶 de Bruijn 图中寻找一条欧拉回来完成。为了构造 $B(2, 4)$ ，可以先绘制其对应的 de Bruijn 图，其顶点是字符集 A 上长度为 $n-1$ 的子串，顶点间根据子串的相互关系构成有向边，如图 10-4 所示。

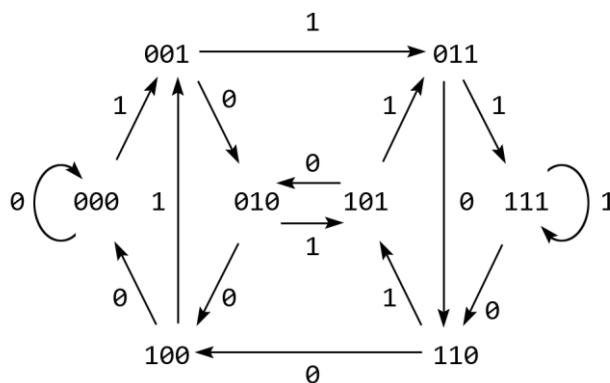


图 10-4 在字符集 $A=\{0, 1\}$ 上为了构造 $B(2, 4)$ 所建立的 de Bruijn 图。如果从某一点出发，遍历所有的边仅且一次然后再回到起点，则所有四位数字恰好只出现一次（对应一个欧拉回）；如果从某一点出发，遍历所有的顶点仅且一次，则所有三位数字恰好只出现一次（对应一条哈密顿路）

观察图可知，建立的图是一个有向图，而且所有顶点的出入度均相等，因此是有向欧拉图。可以应用之前介绍的 Hierholzer 算法构建欧拉回。容易得到，图中的一条欧拉回为：

000 $\xrightarrow{0}$ 000 $\xrightarrow{1}$ 001 $\xrightarrow{1}$ 011 $\xrightarrow{1}$ 111 $\xrightarrow{1}$ 111 $\xrightarrow{0}$ 110 $\xrightarrow{1}$ 101 $\xrightarrow{1}$ 011 $\xrightarrow{0}$ 110 $\xrightarrow{0}$ 100 $\xrightarrow{1}$ 001 $\xrightarrow{0}$ 010 $\xrightarrow{1}$ 101 $\xrightarrow{0}$ 010 $\xrightarrow{0}$ 100 $\xrightarrow{0}$ 000 (有向箭头上方的数字为顶点所经过的边)。将欧拉回所经过边按序连接起来便可得到 $B(2, 4)$ ——0111101100101000。

综上所述, 求解本题的关键是将问题建模成有向欧拉图, 然后找出一条欧拉回, 该欧拉回满足在路径的每一步所选择的边都是字典序最小的, 这样得到的就是最小的环数。

参考代码

```

int used[1 << 20][2], sequence[22][1 << 21];

void hierholzer(int n) {
    int mask = (1 << (n - 2)) - 1, u = 0;
    stack<int> path;
    vector<int> circuit;
    memset(used, 0, sizeof(int) * (1 << (n - 1)) * 2);
    path.push(u);
    while (!path.empty()) {
        int v = 0;
        for (v = 0; v <= 1; v++)
            if (!used[u][v])
                break;
        if (v <= 1) {
            path.push(u);
            used[u][v] = 1;
            u = ((u & mask) << 1) + v;
        } else {
            circuit.push_back(u);
            u = path.top();
            path.pop();
        }
    }
    int bits = circuit.back();
    mask = (1 << (n - 1)) - 1;
    for (int i = circuit.size() - 2, j = 0; i >= 0; i--, j++) {
        sequence[n][j] = ((bits & mask) << 1) + (circuit[i] & 1);
        bits = sequence[n][j];
    }
}

void trick() {
    sequence[1][0] = 0, sequence[1][1] = 1;
    for (int i = 2; i <= 21; i++) hierholzer(i);
}

int main(int argc, char *argv[]) {
    trick();
    int cases, n, k;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> n >> k;
        cout << sequence[n][k] << '\n';
    }
    return 0;
}

```

强化练习: [10054](#) The Necklace^A, [10506](#) The Ouroboros Problem^D。

扩展练习：13252* Rotating Drum^E。

10.2.2 中国投递员问题

中国投递员问题 (Chinese Postman Problem, CPP), 由我国管梅谷教授于 1960 年首先提出并加以研究^{[93][94]}。它所描述的问题可以表述如下: 假设一个投递员每次投送邮件都要走遍他所负责投递区域的每条街道, 完成投递任务后回到邮局, 他应该选择一条什么样的投递路线, 使得他所走的总路程最短?

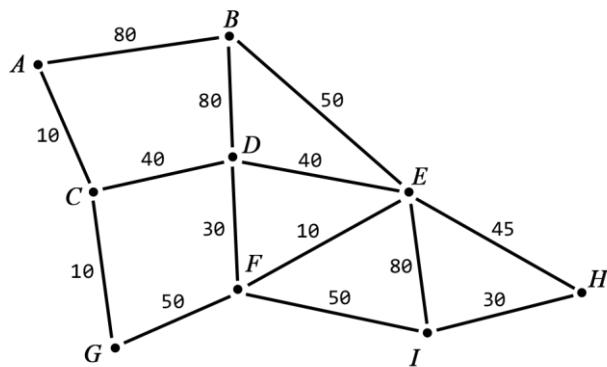


图 10-5 假设投递员起始位置为路口 A，则投递员从路口 A 出发，走过所有街道返回出发路口 A 的最短路程为 755，其中一条可行的路线是：A-B-D-E-B-A-C-D-F-E-H-I-E-F-I-F-G-C-A

可以将此问题建模为图论问题加以解决。将街道的交叉口看做图的顶点，街道视为边（无向或有向），街道的长度视为边的权，则问题转化为确定一条经过图中每条边至少一次的有向闭链，且闭链具有最小的权值。为了便于讨论，将有向闭链称为邮路（post tour），具有最小权的邮路称为最优邮路（optimal post tour）。因为欧拉图中的欧拉回具有经过所有边一次且仅一次的性质，因此解决此问题的一个思路是设法使用最小的花费将原图改造成欧拉图。根据图的性质，可将其分为以下几种情况。

对于无向图来说，如果所有顶点的度数均为偶数，则为欧拉图，只需找出图中的一条欧拉回即可。如果图中只有两个顶点的度数为奇数，其他顶点的度数均为偶数，则属于半欧拉图，那么可以先确定图中的欧拉迹，然后使用最短路径算法确定这两个奇度点之间的最短路径，先沿着欧拉迹从一个奇度点到达另外一个奇度点，然后再沿着最短路径返回起始顶点，容易理解这样的路线是最优的。当奇度点的个数大于两个时，为了将其改造成欧拉图，且添加的边的权值总和最小。根据握手引理，如果图中的具有奇度点，则奇度点的个数一定是偶数，为了将这些奇度点改造成偶度点，一个直观的方法是在这些奇度点间建边，使之配对，所建边的权是两个奇度点之间最短路径的权，约束条件是所有添加的边的权和最小。因此有以下直接的算法：

- (1) 列出图中的所有奇度点;
 - (2) 列出所有可能的奇度点配对;
 - (3) 对于每个配对, 确定此配对中两个奇度点间的最短路径;
 - (4) 从所有配对中选择某些配对进行组合, 使得所有奇度点成为偶度点, 且配对的权和最小;
 - (5) 将配对所对应的最短路径以边的形式添加到原图中;
 - (6) 寻找新图的欧拉回;

一般来说, 有关 CPP 的题目中, 给定的顶点数量一般都较小, 因此可以使用回溯法来确定具有最小权的配对组合¹, 确定奇度点间的最短路径则可以使用 Moore-Dijkstra 算法或者 Floyd-Warshall 算法。

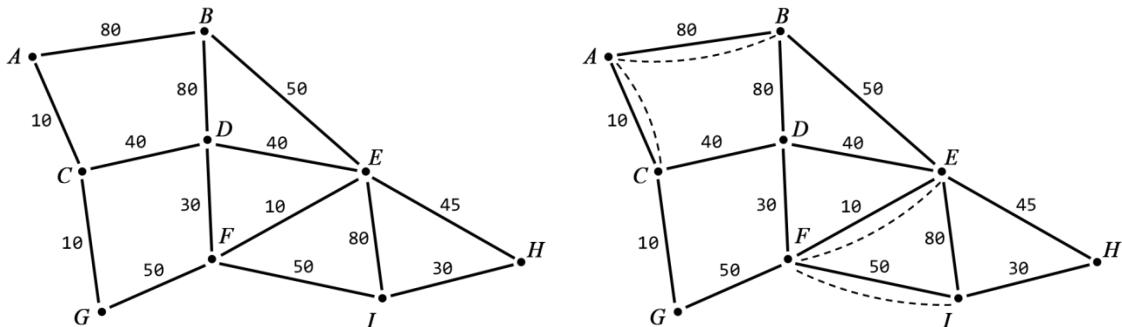


图 10-6 通过添加边的方式将无向非欧拉图“改造”为欧拉图。左侧无向图中有 4 个奇度点, 分别为 B, C, E, I 。根据 Floyd-Warshall 算法可以得到 4 个奇度点之间的最短距离矩阵为

	B	C	E	I
B	0	90	50	110
C	90	0	70	110
E	50	70	0	60
I	110	110	60	0

通过观察不难得知, 选择 B 和 C 配对, E 和 I 配对, 可以使用最小的“代价”将左侧的非欧拉图“改造”为右侧的欧拉图, 所需添加的虚线边的最小权值为 $90+60=150$ 。原有边权之和为 605, “改造”后边权之和为 755。改造完成后所有顶点均为偶度点, 一条可行的欧拉路径为: $A-B-D-E-B-A-C-D-F-E-H-I-E-F-I-F-G-C-A$

对于有向图来说, 与无向图类似, 都是尝试使用最小的代价来构造欧拉图, 对此种类型, 存在有效算法——Edmonds-Johnson 算法^[95], 感兴趣的读者可以进一步查阅相关文献。

对于混合图 (即某些街道是单向通行, 另外的街道是双向通行) 来说, CPP 属于 NP 问题, 当数据规模较小时可以使用回溯法予以解决。

强化练习: 117 The Postal Worker Rings Once^A。

扩展练习: [10296](#) Jogging Trails^C。

10.2.3 哈密顿回

哈密顿路 (Hamiltonian path) 是经过给定图 G 中的每个顶点一次并且仅一次的路。哈密顿回 (Hamiltonian circuit) 是一条哈密顿路, 且起点和终点相同, 将包含哈密顿回的图称为哈密顿图 (Hamiltonian graph), 类似于半欧拉图, 将只包含哈密顿路的图称为半哈密顿图 (semi-Hamiltonian graph)。

不同于欧拉回问题的判定, 哈密顿回问题目前尚未找到有效的判定方法。在某些特殊情况下, 可以应用一些充分条件或者必要条件来判定给定的图是否为哈密顿图, 例如前述的 de Bruijn 序列问题, 其对应的 de Bruijn 图必定存在哈密顿路。下面给出一个判断给定简单图是否有哈密顿回的充分条件。设 G 是包含 n ($n \geq 3$) 个顶点的简单图, 如果对于图中的任意两个非邻接的顶点 v 和 w 都满足

¹ 或者使用第 11 章“动态规划”中第 11.4 节的“集合型动态规划”并结合备忘技巧来确定具有最小权的配对组合。

$$\deg(v) + \deg(w) \geq n$$

则图 G 是哈密顿图^[96]。根据此充分条件可以进一步作出推论： G 是包含 n ($n \geq 3$) 个顶点的简单图，如果对任意顶点 v 均有

$$\deg(v) \geq \left\lceil \frac{n}{2} \right\rceil$$

则 G 为哈密顿图^[97]。 $\deg(u)$ 表示 u 的顶点度。对于满足上述条件的简单图，可以通过下述简单算法构建一条哈密顿回^[98]。

- 1) 忽略相邻顶点的邻接关系，将所有顶点排列成一个环。
- 2) 对于环中的两个相邻顶点 v_i 和 v_{i+1} ，如果它们在图中不邻接，进行以下两个步骤：
 - 2a) 在环中搜索序号 j ，使得 $v_i, v_{i+1}, v_j, v_{j+1}$ 是四个不同的顶点且图中存在边 (v_i, v_j) 和 (v_{j+1}, v_{i+1}) ；
 - 2b) 将环中 v_{i+1}, v_j 之间（包括 v_{i+1} 和 v_j ）的顶点反向。

对于一般图来说，找出图中的哈密顿路属于 NP 难问题，目前尚未发现有效的算法，一般是通过回溯法（深度优先搜索）予以解决。对于之前的例题 10040 Ouroboros Snake，下面给出使用深度优先搜索进行求解的方法，其目标是遍历 n 阶的 de Bruijn 图中的每个顶点恰好一次。相对于寻找欧拉回的求解方法，此解法递归深度较大，对于某些内存较小的主机，容易导致栈溢出，需要将递归实现转换为非递归实现来避免栈溢出问题。

参考代码

```
//-----10.2.3.cpp-----//
int n, k;
int used[1 << 21], sequence[22][1 << 21], top;
int bits, mask;

void dfs(int u) {
    u = (u & mask) << 1;
    for (int v = 0; v <= 1; v++) {
        if (used[u + v]) continue;
        used[u + v] = 1;
        dfs(u + v);
        sequence[n][top++] = u + v;
    }
}

void trick() {
    for (int i = 1; i <= 21; i++) {
        n = i;
        top = 0, bits = 1 << n, mask = (1 << (n - 1)) - 1;
        memset(used, 0, sizeof(int) * bits);
        dfs(0);
    }
}

int main(int argc, char *argv[]) {
    int cases;
    trick();
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        cin >> n >> k;
```

```

    cout << sequence[n][(1 << n) - 1 - k] << '\n';
}
return 0;
}
//-----10.2.3.cpp-----//

```

强化练习: 775 Hamiltonian Cycle^D。

扩展练习: [12841 In Puzzleland \(III\)](#)^D。

10.2.4 旅行商问题

旅行商问题 (Travelling Salesman Problem, TSP) 是指以下的问题: 一个旅行商计划在他所在区域内的所有城市进行商品推销, 他应该选择怎样的一条旅行路线, 以便他能够从某个城市出发, 至少去每个城市一次, 最后回到出发的城市, 使得路线的总路程最短。

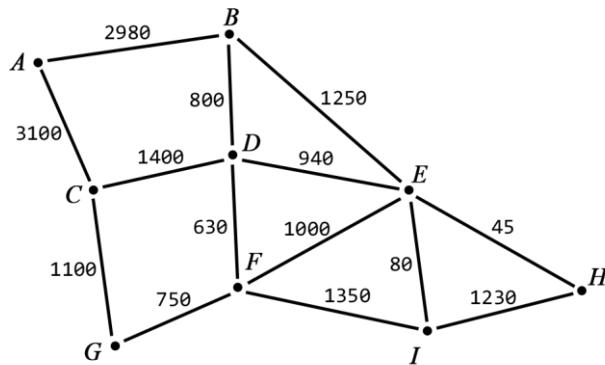


图 10-7 假设旅行商起始位置为城市 A, 则从城市 A 出发, 经过所有其他城市然后返回出发城市 A 的总路程最短为 10920, 其中一条可行的路线是: A-B-D-E-H-E-I-E-F-G-C-A

旅行商问题和中国投递员问题相似, 已经证明它属于一类 **NP 完全** 问题, 目前尚未发现有效算法。如果需要确定精确解, 则对于顶点数量较小的图可以使用回溯法 (当 $|V| \leq 10$ 时) 或者使用动态规划算法 (当 $11 \leq |V| \leq 16$ 时) 予以解决^I, 但是对于顶点数量较多的图, 尚无有效算法。如果不需要精确解, 则可以通过使用近似算法进行求解。

强化练习: [216 Getting in Line](#)^A, [10496 Collecting Beepers](#)^A。

10.3 最小生成树

对于无向连通图 $G=(V, E)$, 它的生成树 (spanning tree)^{II} 是边集合 E' 的一个子集 E' , E' 中不存在圈且连接了 V 中所有顶点。边集 E' 和相应的顶点集 V 构成了一棵树, 称为图 G 的生成树。在所有的生成树中, 具有“边权之和最小”性质的生成树称为最小生成树 (minimum spanning tree, MST)。求图的最小生成树一般是求其边权值之和, 而不是求边的数量, 因为最小生成树的边数均为 $|V| - 1$ 。

^I 关于使用动态规划算法解决 TSP 的介绍, 请读者参阅本书第 11 章“动态规划”第 11.6.3 小节“图论型动态规划”中“旅行商问题”的有关内容。

^{II} 有的图论著作也将其称为支撑树。

强化练习：[11597 Spanning Subtrees^A](#)。

在现实生活中，最小生成树有许多实际的应用。例如：城市规划中求连接各小区的光缆最小长度、自来水管网的最小铺设费用、电子集成电路连接所有引脚的最少连接方案等。求最小生成树的常用算法有 Prim 算法和 Kruskal 算法。

10.3.1 Prim 算法

Prim 算法的核心思想是贪心策略——从一个给定顶点开始逐步扩展以得到最小生成树，在每一次增加顶点时，总是选择从树中指向树外的边中权值最小的一条^{I[99]}。在以下的 Prim 算法实现中，为每个树外顶点保存了从树出发到该顶点边权最小的边（代码中的 `distToTree` 数组），由于每次只在树中增加一个顶点，所以只需要更新从该点出发的所有相邻顶点的 `distToTree` 值。

```
//-----10.3.1.1.cpp-----//
// MAXV 表示图中顶点的最大可能数量；INF 表示“无限大”距离，其值的设置和具体应用有关。
const int MAXV = 1010, INF = 0x7f7f7f7f;

// 表示边的结构体，v 表示结束顶点，weight 表示边权。
struct edge { int v, weight; };

// 使用邻接表方式表示图。
vector<edge> edges[MAXV];

// parent 表示各顶点的父顶点；
// distToTree 表示尚未加入树中的顶点与树中顶点的最小边权，可以将其理解为与树的距离；
// intree 表示顶点是否已经进入生成树中。
int parent[MAXV], distToTree[MAXV], intree[MAXV];

// n 为当前图中顶点的数量。
int n;

// Prim 算法求最小生成树并返回最小生成树的边权之和。
int prim(int u) {
    // 初始化。
    int minSumOfWeight = 0;
    for (int i = 0; i < n; i++) parent[i] = -1, intree[i] = 0, distToTree[i] = INF;
    // 从指定的顶点 u 开始逐步扩增最小生成树。
    distToTree[u] = 0;
    while (!intree[u]) {
        // 将树外的顶点添加到生成树中。
        intree[u] = 1, minSumOfWeight += distToTree[u];
        // 更新与当前顶点 u 连接的顶点 v 到生成树的距离。
        for (auto e : edges[u])
            if (!intree[e.v] && distToTree[e.v] > e.weight)
                distToTree[e.v] = e.weight, parent[e.v] = u;
        // 找到尚未加入树中且与树距离最小的顶点(边权最小)，将其标记为进入最小生成树的候选顶点。
    }
}
```

^I 此算法最初由捷克数学家 Vojtěch Jarník 于 1930 年提出，后由计算机科学家 Robert C. Prim 于 1957 年、Edsger W. Dijkstra 于 1959 年分别重新发现并再次发表。因此，该算法有时候也被称为 Jarník 算法、Prim-Jarník 算法、Prim-Dijkstra 算法，或者 DJP 算法。

```

int minDistToTree = INF;
for (int i = 0; i < n; i++)
    if (!intree[i] && minDistToTree > distToTree[i])
        minDistToTree = distToTree[i], u = i;
}
return minSumOfWeight;
}
//-----10.3.1.1.cpp-----//

```

强化练习：1208 Oreon^C，1235 Anti Brute Force Lock^C。

在上述 Prim 算法的实现中，使用邻接表的方式来表示图，每次选取从树中指向树外的具有最小边权值的边时，使用的是朴素的线性扫描方法，算法总的时间复杂度为 $O(V^2)$ 。如果顶点的数量较多，此方法的效率不高，若要提高效率，可考虑使用标准模板库中的优先队列（内部以堆排序实现，效率较高）来获取当前具有最小边权值的边。

```

//-----10.3.1.2.cpp-----//
const int MAXV = 1010, INF = 0x7f7f7f7f;

struct edge {
    int v, weight;
    edge (int v = 0, int weight = 0): v(v), weight(weight) {}
    bool operator<(const edge &e) const { return weight > e.weight; }
};

vector<edge> edges[MAXV];
int n, parent[MAXV], distToTree[MAXV], intree[MAXV];

int prim(int u) {
    // 初始化。
    int minSumOfWeight = 0;
    for (int i = 0; i < n; i++) parent[i] = -1, intree[i] = 0, distToTree[i] = INF;
    // 将起始顶点本身作为一条边加入优先队列。
    priority_queue<edge> q;
    q.push(edge(u, 0));
    // 从优先队列中获取边权值最小的边进行处理。
    while (!q.empty()) {
        edge e1 = q.top(); q.pop();
        if (!intree[e1.v]) continue;
        intree[e1.v] = 1, minSumOfWeight += e1.weight;
        for (auto e2 : edges[e1.v]) {
            if (!intree[e2.v] && distToTree[e2.v] > e2.weight) {
                distToTree[e2.v] = e2.weight;
                parent[e2.v] = e1.v;
                q.push(edge(e2.v, e2.weight));
            }
        }
    }
    return minSumOfWeight;
}
//-----10.3.1.2.cpp-----//

```

如果使用二叉最小堆来实现优先队列并选取与当前生成树具有最小距离的候选顶点，则时间复杂度可优化为 $O(E \log V)$ 。更进一步地，如果使用斐波那契堆来实现优先队列，则时间复杂度可优化为 $O(E + V \log V)$ 。

强化练习：1151 Buy or Build^D，10034 Freckles^A。

10.3.2 Kruskal 算法

Kruskal 算法的核心思想也是贪心策略——在算法的每一步，添加到森林中的边，其权值都是尽可能的小^{1[100]}。Kruskal 算法采用并查集来实现，可将其概述如下：将图中的每个顶点表示成只包含单个元素的集合，根据边权值的大小按升序排列，对于排好序的边逐一处理，如果边连接的两个顶点不在同一个集合中，则将包含这两个顶点的集合予以合并，直到最后所有顶点均包含在同一个集合中，集合中的顶点之间具有最小权值的边即构成最小生成树。

```

//-----10.3.2.cpp-----//
// MAXV 表示可能的最大顶点数量， MAXE 表示可能的最大边数量，需要根据具体题目进行设置。
const int MAXV = 110, MAXE = 12100;

// 以边列表的形式表示图。
struct edge {
    int u, v, weight;
    edge (int u = 0, int v = 0, int weight = 0): u(u), v(v), weight(weight) {}
    bool operator < (const edge &e) const { return weight < e.weight; }
} edges[MAXE];

// n 表示图中顶点的数量，m 表示图中边的数量。顶点和边的编号均从 0 开始计数。
int n, m;

// parent 记录并查集中各元素的代表，ranks 记录元素的秩。
int parent[MAXV], ranks[MAXV];

// 初始化并查集。
void makeSet() {
    for (int i = 0; i < n; i++) parent[i] = i, ranks[i] = 0;
}

// 查找元素的代表。
int findSet(int x) {
    return (parent[x] == x ? x : parent[x] = findSet(parent[x]));
}

// 合并元素。
bool unionSet(int x, int y) {
    x = findSet(x), y = findSet(y);
    if (x != y) {
        if (ranks[x] > ranks[y]) parent[y] = x;
        else {
            parent[x] = y;
            if (ranks[x] == ranks[y]) ranks[y]++;
        }
        return true;
    }
    return false;
}

// Kruskal 算法求最小生成树以及最小边权和。

```

¹ Joseph Bernard Kruskal Jr. (1928—2010)，美国数学家、计算机科学家。

```

int kruskal() {
    // sumOfWeight 记录最小边权和, cntOfMerged 记录并查集合并的次数。
    int sumOfWeight = 0, cntOfMerged = 0;
    // 初始化并查集。
    makeSet();
    // 将所有边按权值升序进行排列。
    sort(edges, edges + m);
    // 逐条边进行处理, 当两个顶点分属不同集合 (连通分支) 时, 予以合并, 累加最小边权和。
    for (int i = 0; i < m; i++) {
        if (unionSet(edges[i].u, edges[i].v)) {
            sumOfWeight += edges[i].weight;
            // 计数合并的次数, 如果合并次数等于 n-1, 表明所有顶点已在生成树中。
            if (++cntOfMerged == n - 1) break;
        }
    }
    // 通过合并次数判断是否存在最小生成树。
    if (cntOfMerged != n - 1) sumOfWeight = -1;
    return sumOfWeight;
}
//-----10.3.2.cpp-----//

```

在 Kruskal 算法中, 每进行一次合并, 就有一个顶点进入最小生成树, 在经过 $|V|-1$ 次合并后, 最小生成树就已经得到, 因此可以通过检查合并的次数来判断是否可以得到最小生成树, 这个技巧在解题时非常有用。从 Kruskal 算法的参考实现中不难看出, Kruskal 算法的时间复杂度主要取决于边排序算法, 因此可以得到 Kruskal 算法的时间复杂度为 $O(E \log E)$ 。

强化练习: [908 Re-Connecting Computer Sites^A](#), [1174 IP-TV^C](#), [1395 Slim Span^C](#), [10147 Highways^B](#), [10369 Arctic Network^A](#), [10397 Connect the Campus^A](#), [11228 Transportation System^B](#), [11631 Dark Roads^A](#), [11710 Expensive Subway^B](#), [11733 Airports^B](#), [11747 Heavy Cycle Edges^B](#), [11857 Driving Range^B](#)。

扩展练习: [1013 Island Hopping^D](#), [1040* The Traveling Judges Problem^D](#), [1216 The Bug Sensor Problem^D](#), [1265 Tour Belt^D](#), [10307 Killing Aliens in Borg Maze^B](#), [10807* Prim Prim^D](#), [11267* The Hire-a-Coder Business Model^D](#)。

10.3.3 最小生成树的扩展问题

许多相关的问题可以使用最小生成树的特性来解决。

- 最大生成树 (maximum spanning tree)。最大生成树与最小生成树正好相反, 它所求的是边权和最大的生成树。可以采用一个小技巧——将所有边权取其相反数构成一个新图, 新图的最小生成树即为原图的最大生成树——予以解决。或者, 在使用 Kruskal 算法求最小生成树时将边按照权值从大到小的顺序排列, 这样得到的生成树即为最大生成树。
- 乘积最小的生成树。如果所有边权均为正值, 要求一棵边权乘积尽量小的生成树, 则根据对数运算规则 $\log_{10}(a \cdot b) = \log_{10}(a) + \log_{10}(b)$, 只需使用边权的对数来代替, 求新图的最小生成树就是原图边权乘积最小的生成树。
- 瓶颈生成树 (bottleneck spanning tree)。对于无向图 G 来说, 瓶颈生成树的最大边权在 G 的所有生成树中是最小的, 最小生成树正好满足此性质。根据 Kruskal 算法的正确性可以很容易证明这一点。

强化练习: [1234 RACING^C](#), [10457 Magic Car^D](#), [10842 Traffic Flow^B](#)。

扩展练习: [10805* Cockroach Escape Networks^D](#)^{[101][102]}。

10.3.4 度限制最小生成树

度限制最小生成树 (minimum bounded degree spanning tree) 问题是指给定无向加权图 $G=(V, E)$ 以及正整数 $k \geq 2$, 要求确定一棵最小生成树, 该生成树中所有顶点的度最大不超过 k 。如果 $k=2$, 则问题转化为哈密顿路问题。度限制最小生成树问题被认为属于 NP 难问题, 目前尚未发现有效算法^[103]。

如果对问题的条件进一步加以约束, 度限制仅限于某个特定的顶点 v_0 , 即 T 是 G 的生成树且在此生成树 T 中 v_0 的顶点度 $d_T(v_0)=k$, 则称 T 为 G 的单顶点 k 度限制最小生成树 (k-degree bounded minimum spanning tree) ^[104]。

可采用下述算法确定单顶点 k 度限制最小生成树:

(1) 对除顶点 v_0 以外的点集 $\{v_1, v_2, \dots, v_n\}$ 求一次最小生成森林, 令最小生成森林的连通分支数为 m , 若 $m > k$, 则无解, 因为至少需要 m 条边才能将所有连通分支与顶点 v_0 连接; 若 $m \leq k$, 但顶点 v_0 与各连通分支之间可供连接的边数小于 m , 同样无解;

(2) 接着通过交换可行边来增加顶点 v_0 的度, 检查是否能够减少最小生成树的大小。具体方法是每次尝试加入一条和 v_0 关联但未使用的边, 由于加入这样的边后会形成圈, 必须删去所形成圈上的最长边来得到最小生成树, 因此需要确定圈上每个顶点到 v_0 的最长边, 选择增量最小的边进行交换 (增量可能为负值)。

算法的主要难点在于如何确定圈上每个顶点到 v_0 的最长边, 这可以通过树形动态规划预处理得到。具体方法如下: 设 $longest[v]$ 为顶点 v 与 v_0 的路径上的最长边, 对于 v_0 本身以及与 v_0 有关联边的顶点 v_x , 令 $longest[v_0] = longest[v_x] = \infty$, 对于其他顶点有: $longest[v_y] = \max(longest[father[v_y]], e(father[v_y], v_y))$, 其中 $father(v_y)$ 为 v_y 所在子树的根, 以 v_0 为根进行一次 DFS 即可得到 $longest[v]$ 。

```

//-----10.3.4.cpp-----
const int MAXV = 110, MAXE = 10010, INF = 0x7f7f7f7f;

struct edge {
    int u, v, w;
    edge (int u = 0, int v = 0, int w = 0): u(u), v(v), w(w) {}
    bool operator<(const edge &e) const { return w < e.w; }
} edges[MAXE];

vector<edge> g[MAXV], ue;

int n, m;

// 并查集。
int parent[MAXV];
void makeSet() { for (int i = 0; i < n; i++) parent[i] = i; }
int findSet(int x) {
    return parent[x] == x ? x : parent[x] = findSet(parent[x]);
}
bool unionSet(int x, int y) {
    x = findSet(x), y = findSet(y);
    if (x == y) return false;
    parent[x] = y;
    return true;
}

// 将边列表中序号为 i 的边添加到无向图中。
void addEdge(int i) {
    edge e = edges[i];

```

```

g[e.u].push_back(edge(i, e.v, e.w));
g[e.v].push_back(edge(i, e.u, e.w));
}

// chosen[i] 记录边列表中序号为 i 的边是否进入最小生成树;
// connected[i] 记录代表为 i 的强连通分支是否已经有边予以连接;
// longest[i] 记录到顶点 i 的路径上的最长边;
// link[i] 记录到顶点 i 的路径上的最长边在边列表中的序号;
// idx[i] 记录顶点 i 关联的边在边列表中的序号;
int chosen[MAXE], connected[MAXV], longest[MAXV], link[MAXV], idx[MAXV];

// 通过深度优先遍历确定根结点到其他顶点的路径上的最长边。
void dfs(int father, int u) {
    for (auto e : g[u]) {
        if (!chosen[e.u]) continue;
        if (e.v == father) continue;
        if (~father) {
            longest[e.v] = e.w;
            link[e.v] = e.u;
            if (longest[u] > longest[e.v])
            {
                longest[e.v] = longest[u];
                link[e.v] = link[u];
            }
        } else longest[e.v] = -INF;
        dfs(u, e.v);
    }
}

// 确定以顶点 u 为根的单顶点 k 度限制最小生成树。
int kdbmst(int u, int k) {
    memset(chosen, 0, sizeof(chosen));
    memset(idx, -1, sizeof(idx));

    makeSet();
    sort(edges, edges + m);

    ue.clear();
    for (int i = 0; i < m; i++) {
        edge e = edges[i];
        if (e.u == u) { ue.push_back(edge(i, e.v, e.w)); continue; }
        if (e.v == u) { ue.push_back(edge(i, e.u, e.w)); continue; }
        if (unionSet(e.u, e.v)) chosen[i] = 1;
    }
    // 确定最小生成树森林的分支数。
    int components = 0;
    connected[u] = 0;
    for (int i = 0; i < n; i++)
        if (i != u && findSet(i) == i) {
            components++;
            connected[i] = 0;
        }
    if (components > k) return INF;
    // 将指定顶点 u 与各个连通分支连接。
    sort(ue.begin(), ue.end());
    for (auto e : ue) {

```

```

int eid = e.u, scc = findSet(e.v);
idx[e.v] = eid;
if (!connected[scc]) connected[scc] = chosen[eid] = 1;
}
// 检查所有连通分支是否均有边连接。
for (int i = 0; i < n; i++)
    if (i != u && findSet(i) == i)
        if (!connected[i])
            return INF;
// 生成以 u 为根的图。
for (int i = 0; i < n; i++) g[i].clear();
for (int i = 0; i < m; i++)
    if (chosen[i])
        addEdge(i);
// 尝试替换边，增加顶点 u 的度。
while (components < k) {
    dfs(-1, 0);
    int delta = INF, selected = INF;
    for (auto e : ue) {
        if (chosen[e.u]) continue;
        if (e.w - longest[e.v] < delta) {
            delta = e.w - longest[e.v];
            selected = e.v;
        }
    }
    // 若不存在可替换的边则退出，否则更新最小生成树。
    if (delta == INF) return INF;
    chosen[link[selected]] = 0;
    chosen[idx[selected]] = 1;
    addEdge(idx[selected]);
    components++;
}
// 统计最小生成树的边权和。
int sum = 0;
for (int i = 0; i < m; i++)
    if (chosen[i])
        sum += edges[i].w;
return sum;
}
//-----10.3.4.cpp-----

```

强化练习：1537* Picnic Planning^E。

10.3.5 次最优最小生成树

设 $G=(V, E)$ 是一个无向连通图，在其上定义了权值函数 $w: E \rightarrow \mathbb{R}$ ，并假设 $|E| \geq |V|$ ，如果所有边的权值都是不同的，那么 G 的最小生成树是唯一的，但次最优最小生成树（second-best minimum spanning tree）却未必唯一。次最优最小生成树的定义如下：令 $\textcolor{red}{T}_A$ 为 G 的所有生成树的集合，并令 T 为 G 的一棵最小生成树，那么次最优最小生成树是这样的一棵最小生成树 T_s ，它满足

$$w(T_s) = \min_{T'' \in \textcolor{red}{T}_A - \{T\}} \{w(T'')\}, \quad w(T) \text{ 表示生成树 } T \text{ 的边权和}$$

求解次最优最小生成树有两种常用的方法。第一种方法思路较为直接：先求出原图的最小生成树 T ，然后逐次移除 T 中的某条边 e ，由尚未进入最小生成树 T 的边以及 T 中除 e 以外的其他边构成新图 G' ，**确定** G' 的最小生成树 T' ，取能得到的新生成树 T' 的最小值。需要注意，移除初始最小生成树 T 的一条边后，有可

能通过剩余的边无法得到最小生成树。这种情况可以通过预先进行一次图遍历，由图的连通性是否满足来予以排除。或者，在求最小生成树过程中计数已经加入生成树的顶点数，与总的顶点数进行比较，如果尚有顶点未进入最小生成树，则表明由这些剩余的边构成的图是非连通图，无法构成一棵最小生成树。由于最小生成树共有 $|V|-1$ 条边，上述方法相当于反复求 $|V|-1$ 次最小生成树，总的时间复杂度为 $O(VE\log E)$ 。

```
//-----10.3.5.1.cpp-----//
const int MAXV = 110, MAXE = 12100, INF = 0x7f7f7f7f;

// 使用边列表方式表示图。
struct edge {
    int u, v, weight, enabled;
    bool operator<(const edge &e) const { return weight < e.weight; }
} edges[MAXE];

// n 为顶点的数量，m 为边的数量。
int n, m;

// 并查集。
int parent[MAXV], ranks[MAXV];

// 并查集：初始化。
void makeSet() {
    for (int i = 0; i < n; i++) parent[i] = i, ranks[i] = 0;
}

// 并查集：查找。
int findSet(int x) {
    return (parent[x] == x ? x : parent[x] = findSet(parent[x]));
}

// 并查集：合并。
bool unionSet(int x, int y) {
    x = findSet(x), y = findSet(y);
    if (x != y) {
        if (ranks[x] > ranks[y]) parent[y] = x;
        else {
            parent[x] = y;
            if (ranks[x] == ranks[y]) ranks[y]++;
        }
        return true;
    }
    return false;
}

// 利用 Kruskal 算法求次最优最小生成树。
void kruskal() {
    // fbSumOfWeight 用于记录最小生成树的边权和;
    // intree 用于记录原图进入最小生成树的边的序号;
    // cntOfIntree 用于记录原图进入最小生成树的边的数量;
    // cntOfMerged 用于记录 Kruskal 算法中并查集的合并次数。
    int fbSumOfWeight = 0, intree[MAXV], cntOfIntree = 0, cntOfMerged = 0;
    makeSet();
    sort(edges, edges + m);
    for (int i = 0; i < m; i++)
```

```

if (unionSet(edges[i].u, edges[i].v)) {
    // 记录进入最小生成树的边序号。
    intree[cntOfIntree++] = i;
    fbSumOfWeight += edges[i].weight;
    if (++cntOfMerged == n - 1) break;
}
// 检查合并的次数以确定是否存在最小生成树，如果存在则输出最小边权和。
if (cntOfMerged != (n - 1)) {
    cout << "No MST exists!\n";
    return;
} else cout << "MST = " << fbSumOfWeight << '\n';
// 逐次移除原图最小生成树的一条边，然后对新图求最小生成树。
int sbSumOfWeight = INF, sum;
for (int i = 0; i < cntOfIntree; i++) {
    // 删除边。
    edges[intree[i]].enabled = 0;
    // 在新图上执行 Kruskal 算法。
    makeSet();
    sum = 0, cntOfMerged = 0;
    for (int j = 0; j < m; j++) {
        if (edges[j].enabled && unionSet(edges[j].u, edges[j].v)) {
            sum += edges[j].weight;
            if (++cntOfMerged == n - 1) break;
        }
    }
    // 恢复边。
    edges[intree[i]].enabled = 1;
    // 更新最小生成树的权值和。
    if (cntOfMerged == n - 1) sbSumOfWeight = min(sbSumOfWeight, sum);
}
// 检查是否存在次最小生成树，如果存在则输出其边权和。
if (sbSumOfWeight == INF) cout << "No second best MST exists!\n";
else cout << "Second best MST = " << sbSumOfWeight << '\n';
}
//-----10.3.5.1.cpp-----//

```

可以证明，如果 G 存在次最优最小生成树，令 T 是 G 的一棵最小生成树，则必定存在边 $(u, v) \in T$ 以及边 $(x, y) \notin T$ ，使得 $(T - (u, v)) \cup (x, y)$ 是 G 的一棵次最优最小生成树。那么如何高效地找到这两条边呢？考虑到最小生成树 T 的边数为 $|V| - 1$ ，如果加入任意一条未进入最小生成树的边，则 T 中将出现圈，将此圈中的任意一条边移除，仍然为一棵生成树，但不一定是最小生成树。由于替换边可能会导致 T 的权值之和发生变化，易知确定次最优最小生成树就是找出具有最小差值的替换。由此产生了第二种方法：根据 Prim 算法可以得到最小生成树 T ，对于得到的最小生成树 T 中的两个顶点 u, v ，令 $\maxEdge[u, v]$ 表示 T 中 u 和 v 之间最短路径上具有最大权值的边，若将一条不属于 T 的边 (u, v) 加入 T 中，能够引起 T 的权值之和变化最小的选择是替换 $\maxEdge[u, v]$ 所指向的边。 $\maxEdge[u, v]$ 可以通过 BFS（或者 DFS）以 $O(V^2)$ 的时间复杂度确定。具体方法是根据 Prim 算法得到的最小生成树 T ，从 T 中的每个顶点 u 出发，利用 BFS（或者 DFS）确定与 T 中其他顶点 v 最短路径上具有最大权值的边。由于 T 是生成树，故 T 中的边均为树边，当访问树边 (x, y) 时，可以通过以下递推关系确定 $\maxEdge[u, y]$ ：

$$\maxEdge[u, y] = \begin{cases} \maxEdge[u, x] & w(\maxEdge[u, x]) \geq w((x, y)) \\ (x, y) & w(\maxEdge[u, x]) < w((x, y)) \end{cases}$$

由于 T 包含 $|V| - 1$ 条边，对于单个顶点 u ，通过 BFS（或者 DFS）确定 u 到 T 中其他顶点 y 的 $\maxEdge[u, y]$ ，

$y]$ 的时间复杂度为 $O(V)$ ，故总的时间复杂度为 $O(V^2)$ 。

综上所述，可以通过以下步骤来确定次最优最小生成树：

- (1) 由 Prim 算法确定最小生成树 T ；
- (2) 从 T 中每个顶点 u 出发进行 BFS（或者 DFS），确定到 T 中其他顶点 v 的 $maxEdge[u, v]$ ；
- (3) 对于每条未进入最小生成树 T 的边 (x, y) ，计算边权差^I： $diff[x, y] = |w((x, y)) - w(maxEdge[x, y])|$ ，从边权差中选取最小的 $diff[x', y']$ ，然后在最小生成树 T 中删除 $maxEdge[x', y']$ 所对应的边，加入边 (x', y') 即可得到次最优最小生成树。

如果使用最小堆来实现 Prim 算法，步骤(1)的时间复杂度为 $O(E \log V)$ ，步骤(2)的时间复杂度为 $O(V^2)$ ，步骤(3)中需要考虑的边数 $|E| = O(V^2)$ ，故步骤(3)的时间复杂度为 $O(V^2)$ ，则上述求次最优最小生成树算法总的时间复杂度为 $O(V^2)$ 。

```
//-----10.3.5.2.cpp-----//
const int MAXV = 110, MAXE = 12100, INF = 0x7f7f7f7f;

// 边。
struct edge { int u, v, weight, next; } edges[MAXE];

// 使用链式前向星表示图，n 为顶点数，m 为边数。
int head[MAXV], n, m;

// distToTree[u] 表示顶点 u 与最小生成树的最短距离；
// intree[u] 表示顶点 u 是否已经进入最小生成树；
// parent[u] 表示最小生成树中顶点 u 的父顶点；
// used[i] 表示边列表中第 i 条边是否已经进入最小生成树。
int distToTree[MAXV], intree[MAXV], parent[MAXV], used[MAXE];

// idx[u][v] 记录顶点 u 和 v 之间具有最小权值的边 (u, v) 在边列表 edges 中的对应序号。
int idx[MAXV][MAXV];

// maxEdge[u][v] 记录最小生成树中顶点 u 和 v 之间最短路径上边的最大权值。
int maxEdge[MAXV][MAXV];

// 利用 Prim 算法确定次最优最小生成树。
// 需要注意，在此参考实现中，使用的是 Prim 算法中的 parent 数组同步更新 maxEdge[u][v]，而不是先确定生成最小树，然后再使用 BFS（或 DFS）确定 maxEdge[u][v]。
void prim(int u) {
    // 最小边权和。
    int minWeightSum = 0;
    // 初始化。
    for (int i = 0; i < n; i++) {
        distToTree[i] = INF, intree[i] = 0, parent[i] = -1;
        for (int j = 0; j < n; j++)
            maxEdge[i][j] = 0;
    }
    // 将指定的顶点置为已在最小生成树中。
}
```

^I 根据最小生成树的 Prim 算法，必有 $w((x, y)) \geq w(maxEdge[x, y])$ ，故可以去掉绝对值符号，即： $diff[x, y] = w((x, y)) - w(maxEdge[x, y])$ 。

```

distToTree[u] = 0;
while (!intree[u]) {
    for (int i = 0; i < n; i++) {
        if (intree[i]) {
            int weight = edges[ idx[ parent[u] ][ u ] ].weight;
            // 更新树中顶点与拟加入顶点间通路上的最大边权。
            if (i != parent[u])
                maxEdge[i][u] = max(maxEdge[i][parent[u]], weight);
            else
                maxEdge[i][u] = weight;
            // 根据对称性更新最大边权。
            maxEdge[u][i] = maxEdge[i][u];
        }
    }
    // 将顶点加入最小生成树并累加边权和。
    intree[u] = 1;
    minWeightSum += distToTree[u];
    // 更新尚未进入树中的顶点与树的距离。
    for (int i = head[u]; ~i; i = edges[i].next) {
        edge e = edges[i];
        if (!intree[e.v] && distToTree[e.v] > e.weight)
            distToTree[e.v] = e.weight, parent[e.v] = u;
    }
    // 选取尚未进入树中且与树具有最小距离的顶点。
    int minDistToTree = INF;
    for (int i = 0; i < n; i++)
        if (!intree[i] && minDistToTree > distToTree[i])
            minDistToTree = distToTree[i], u = i;
    }
    // 标记已经使用的边。
    memset(used, 0, sizeof(used));
    for (int i = 0; i < n; i++) {
        if (~parent[i])
            used[ idx[i][ parent[i] ] ] = used[ idx[ parent[i] ][ i ] ] = 1;
    }
    // 逐一枚举尚未使用的边，寻找替换后的最小边权差。
    int minWeightDiff = INF;
    for (int i = 0; i < m; i++)
        if (!used[i]) {
            int diff = edges[i].weight - maxEdge[edges[i].u][edges[i].v];
            if (minWeightDiff > diff) minWeightDiff = diff;
        }
    cout << minWeightSum << ' ' << (minWeightSum + minWeightDiff) << '\n';
}
//-----10.3.5.2.cpp-----//

```

注意在应用第二种方法时对平行边的处理：如果存在平行边，在选取候选边时，总是要选取具有最小边权的那条边，标记已经使用的边时，也是标记具有最小边权的那条平行边。

可以对第二种方法中求两个顶点 u 与 v 之间最短路径上最大边权的过程进行优化。由于 Prim 算法过程得到了最小生成树中各个顶点的父顶点，可以根据最小公共祖先算法确定任意两个顶点 u 与 v 的公共祖先 c ，然后根据 $parent$ 数组得到 u 与 c 之间的最大边权以及 v 与 c 之间的最大边权，取最大值即为两个顶点间通路上的最大边权。

强化练习：10462 Is There a Second Way Left^B, 10600 ACM Contest and Blackout^A, 11354 Bond^D。

扩展练习：1494* Qin Shi Huang's National Road System^D。

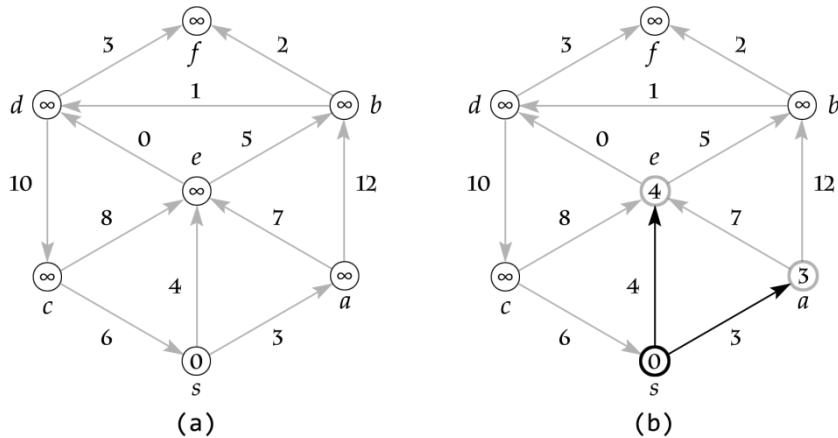
10.4 最短路径问题

最短路径 (shortest path) 问题是图论的经典问题。从图中某一顶点 (称为源点) 出发, 到达另一顶点 (称为终点) 的所有路径中 (路径可能不存在或者存在不止一条), 各边权值之和最小的路径, 称为最短路径。最短路径问题分为两类: 一类是求单个顶点和其他所有顶点的最短路径, 称为单源最短路径问题; 另一类是求所有顶点相互之间的最短路径, 称为多源最短路径问题。对于以上两类最短路径问题, 都有相应的有效算法予以解决。

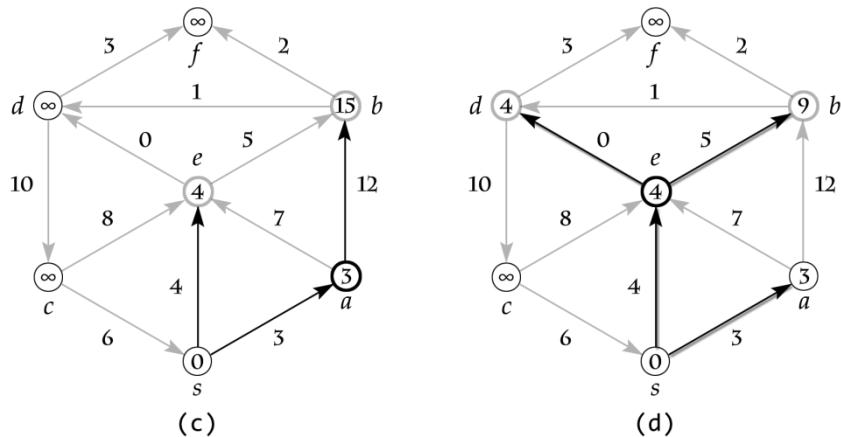
10.4.1 Moore-Dijkstra 算法

Moore-Dijkstra 算法适用于在有点权或边权的图中寻找两个顶点之间的最短路径。该算法由 Moore (1957), Dijkstra (1959), Whiting 与 Hillier (1960) 各自独立发现^[105]。Moore-Dijkstra 算法应用贪心策略进行设计, 在每次选择下一个顶点更新最短路径时, 总是选择和起点具有最小距离且尚未被访问过的顶点。

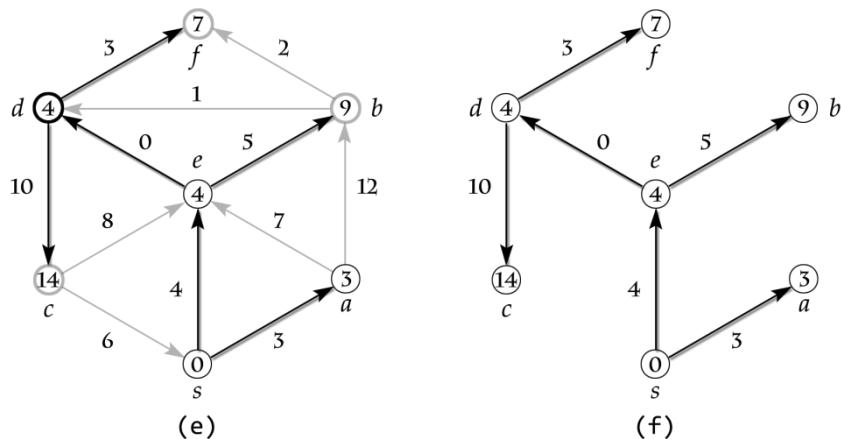
算法的工作过程可以概述如下: 先将图中的顶点划分为两个集合 S 和 T , S 表示已经求得最短路径的顶点集合, T 表示尚未求得最短路径的顶点集合。初始时 S 中只包含源点 s , 源点 s 和自身的最短路径为 0。接下来每次从集合 T 中选取一个和源点 s 具有最短路径的顶点 v_i , 通过 v_i 更新源点 s 和 T 中其他顶点间的最短路径, 更新完毕后将 v_i 加入集合 S 中, 如此循环直到集合 T 为空, 此时源点 s 和所有其他顶点间的最短路径已经求得。



(a) 有向图从源点 s 开始, 要求确定到其他顶点的最短路径。(b) 初始时, 选择源点 s 进入集合 S , 此时源点 s 距离起点的最短距离为 0, 以源点 s 为当前顶点, 更新其他尚未进入集合 S 的顶点与源点 s 的最短距离。此处更新了顶点 a 和顶点 e 与源点 s 的最短路径。



(c) 由于顶点 a 是尚未扫描的顶点中与源点 s 距离最短的顶点, 从尚未进入集合 S 的顶点中选择一个与源点 s 具有最短路径的顶点, 此处为顶点 a , 标记其进入集合 S , 以顶点 a 为当前顶点, 更新其他尚未进入集合 S 的顶点与源点 s 的最短距离。(d) 由于顶点 e 是尚未扫描的顶点中与源点 s 距离最短的顶点, 继续从集合 T 中选择顶点 e 进入集合 S , 以顶点 e 为当前顶点, 更新其他尚未进入集合 S 的顶点与源点 s 的最短距离



(e) 由于顶点 d 是尚未扫描的顶点中与源点 s 距离最短的顶点, 从集合 T 中选择顶点 d 进入集合 S , 以顶点 d 为当前顶点, 更新其他尚未进入集合 S 的顶点与源点 s 的最短距离。(f) 按照前述步骤直到所有顶点均进入集合 S 。最终得到的以源点 s 为根的最短路径树

图 10-8 Moore-Dijkstra 算法的工作示例

从算法的工作过程描述可以看出, 此算法和之前介绍的求最小生成树的 Prim 算法非常相似 (实际上, 实现代码仅有细微差别)。算法的基本原理很容易理解: 源点 s 和尚未进入集合 S 的顶点 v_i 间的最短路径要么是由源点 s 和 v_i 之间的直接边构成, 要么就是源点通过 S 中的某个顶点到达 v_i 所形成的路径。

在具体实现时, 使用 $dist[u]$ 表示源点 v_0 和顶点 u 之间最短路径的长度, $visited[u]$ 表示顶点 u 是否已经进入集合 S 。初始时 $dist[s]=0$, 其他顶点 x 和源点的最短路径长度 $dist[x]$ 设置为一个“无穷大值”, 表示起

始顶点和其他顶点间的距离尚未确定^I。 $visited[s]=1$ ，表示源点 s 已经进入集合 S ，其他顶点 $visited[u]=0$ ，表示尚未进入集合 S 。接下来从与 s 相邻接的顶点中选出具有最小边权值的一条边所连接的顶点 v_i ，将 v_i 标记为已经访问，即 $visited[v_i]=1$ ，表示已经加入集合 S 中，然后通过 v_i 来更新源点 s 到其他顶点的最短路径，重复上述过程直到所有顶点已经访问。

```
-----10.4.1.1.cpp-----
// 最大值的设置和具体应用有关，注意防止在运算过程中超出数据类型的表示范围。
const int MAXV = 1010, INF = 0x3f3f3f3f;

// 表示边的结构，v 为边的结束顶点编号，weight 为边权。
struct edge {
    int v, weight;
    edge (int v = 0, int weight = 0): v(v), weight(weight) {}
};

// 使用邻接表方式表示图。
vector<edge> edges;

// 图中顶点数量。
int n;

// dist 记录其他顶点与源点间的最短距离；parent 记录前驱；visited 记录顶点是否已经访问。
int dist[MAXV], parent[MAXV], visited[MAXV];

// 求其他顶点与顶点 u 的最短路径。
void mooreDijkstra(int u) {
    // 初始化。
    for (int i = 0; i < n; i++)
        dist[i] = INF, parent[i] = -1, visited[i] = 0;
    dist[u] = 0;
    // 反复应用贪心策略选取距离源点最近的顶点，然后从该顶点更新与其他顶点的距离。
    while (!visited[u]) {
        visited[u] = 1;
        for (auto e : edges[u])
            if (!visited[e.v] && dist[e.v] > dist[u] + e.weight)
                dist[e.v] = dist[u] + e.weight, parent[e.v] = u;
        // 选取和起始顶点具有最短距离且尚未被访问的顶点。
        int least = INF;
        for (int i = 0; i < n; i++)
            if (!visited[i] && least > dist[i])
                least = dist[i], u = i;
    }
}
-----10.4.1.1.cpp-----
```

在实现中可以根据需要设置一个数据结构，用以记录构成最短路径的顶点的各自前驱顶点，以便根据这些信息重建最短路径本身而不仅仅是获得最短路径的长度，例如上述代码中的 *parent* 数组。Moore-Dijkstra 算法除了源点 s 之外，需要将 $|V|-1$ 个顶点加入到集合 S 中，在每加入一个顶点时，要在集合 T 的 $|V|-1$

^I 该值和具体的应用有关，需要将其设置为一个“足够大”的数，使得源点 s 到其他所有顶点的最短路径长度不会超过此数。或者将其设置为 -1 ，以便在更新最短路径长度时使用额外的判断来予以区分。

个顶点中搜索和源点具有最短路径长度的顶点，进而更新源点与其他顶点的最短路径长度，因此时间复杂度为 $O(V^2+E)$ 。

强化练习：157 Route Finding^C，186 Trip Routing^B，238 Jill's Bike^E，[341 Non-Stop Travel^B](#)，[1112 Mice and Maze^B](#)，[10389 Subway^C](#)，[10436 Cheapest Way^D](#)。

扩展练习：[10350 Liftless EME^C](#)，[10816 Travel in Desert^C](#)。

在前述的 Moore-Dijkstra 算法实现中，每次通过扫描所有顶点获取与起点具有最短路径的顶点，然后开始下一次距离的更新过程。若顶点数量较多，则总的时间开销较大。如果使用优先队列来获取与起点具有最小距离的候选访问顶点，则有以下更为简洁和高效的实现，其时间复杂度为 $O((V+E)\log V)$ 。

```
//-----10.4.1.2.cpp-----//
// MAXV 表示顶点的最大数量；INF 表示顶点间不可达的距离值，具体数值和应用有关。
const int MAXV = 1010, INF = 0x3f3f3f3f;

// 表示边的结构，v 为终止顶点的编号，weight 为边权。
struct edge {
    int v, weight;
    edge (int v = 0, int weight = 0): v(v), weight(weight) {}
    // 重载小于比较符，使得优先队列为最小优先队列。离起点距离小的顶点排列在前。
    bool operator < (const edge &e) const { return weight > e.weight; }
};

// 以顶点的邻接边列表来表示图。
list<edge> edges[MAXV];

// 图中顶点数量。
int n;

// dist 表示起点和各顶点的最短路径，parent 表示最短路径上各顶点的前驱。
int dist[MAXV], visited[MAXN], parent[MAXV];

// Moore-Dijkstra 算法求单源最短路径。
void mooreDijkstra(int u) {
    // 初始化。
    for (int i = 0; i < n; i++) dist[i] = INF, visited[i] = 0, parent[i] = -1;
    dist[u] = 0;
    // 将起点作为第一个待访问的顶点。
    priority_queue<edge> q;
    q.push(edge(u, dist[u]));
    while (!q.empty()) {
        // 贪心策略：利用优先队列的性质，获取与起点具有最短路径的顶点。
        edge e1 = q.top(); q.pop();
        // 优化：检查是否有必要进行更新，因为当前最短距离可能已经比优先队列中的距离更优，这样可以避免将顶点不必要的重新放入优先队列。在某些情形下是必需的，例如习题：12878 Flowery Trails。
        if (dist[e1.v] < e1.weight) continue;
        // 对于具有最短路径顶点的所有出边，检查是否存在更短的距离，如果存在则更新距离和路径。
        for (auto e2 : edges[e1.v]) {
            if (dist[e2.v] > dist[e1.v] + e2.weight) {
                dist[e2.v] = dist[e1.v] + e2.weight;
                parent[e2.v] = e1.v;
                q.push(edge(e2.v, dist[e2.v]));
            }
        }
    }
}
```

```

        }
    }
}

//-----10.4.1.2.cpp-----//

```

强化练习：[721 Invitation Cards^C](#)，[13127 Bank Robbery^D](#)。

扩展练习：[658* It's Not a Bug It's a Feature^C](#)，[10246 Asterix and Obelix^C](#)，[11635 Hotel Booking^C](#)。

某些题目可能需要获得具有最短长度的所有路径，这可以通过在更新最短距离时记录当前顶点的多个前驱顶点予以解决。

```

// 前驱数组，用于记录每个顶点的多个前驱。
vector<int> parent[MAXV];
// 此处省略了算法执行所需要的相关初始化代码。
while (!q.empty()) {
    edge e1 = q.top(); q.pop();
    if (dist[e1.v] < e1.weight) continue;
    for (auto e2 : edges[e1.v]) {
        if (dist[e2.v] > dist[e1.v] + e2.weight) {
            dist[e2.v] = dist[e1.v] + e2.weight;
            parent[e2.v].clear();
            parent[e2.v].push_back(e1.v);
            q.push(edge(e2.v, dist[e2.v]));
        } else {
            if (dist[e2.v] == dist[e1.v] + e2.weight)
                parent[e2.v].push_back(e1.v);
        }
    }
}

```

929 Number Maze^A (数字迷宫)

考虑如下图所示的一个数字迷宫。迷宫以二维数组表示，其中只包含 0 至 9 的数字。迷宫可以按照上下左右四个方向进行遍历。将每个方格中的数字认为是某种代价的衡量，你面临的挑战是从表示入口的方格开始，找到一条具有最小“代价和”的路径到达表示出口的方格。简而言之，你需要从迷宫的左上角方格出发，寻找一条路径到达右下角的方格，该路径上所有方格的“代价和”最小。迷宫的大小为 $N \times M$ ($1 \leq N, M \leq 999$)。下图所示的数字迷宫中，具有最小“代价和”的路径 (0—7—1—2—3—4—2—5) 的代价为 24。

0	3	1	2	9
7	3	4	9	9
1	7	5	5	3
2	3	4	2	5

输入

输入包含多组迷宫。输入的第一行包含一个正整数，表示迷宫的数量。每个迷宫按以下格式给出：包含行数 N 的一行，包含列数 M 的一行，以及具体的 N 行 M 列的迷宫数字，迷宫数字以空格分隔。

输出

对于每个迷宫，输出指定的最小值。

样例输入

样例输出

```

1
4
5
0 3 1 2 9
7 3 4 9 9
1 7 5 5 3
2 3 4 2 5

```

分析

将方格视为图的顶点，方格内的数字视为相邻方格间的距离，则题目可转换为求无向连通图的单源最短路径问题。将问题所对应的隐式图构建出来，使用 Moore-Dijkstra 算法求解即可。注意下述实现中，Moore-Dijkstra 算法求出的是起始方格与其他方格的最小代价，起始方格本身的代价尚未加入路径的代价中，故在输出时需要予以“补偿”，否则结果将不正确（或者在初始化时，将起点与自身的代价设置为起始方格内的数值而不是 0 值，这样得到的最短路径就是代价，不需再“补偿”初始方格的代价）。由于本题顶点数量较多，故编码时采用 Moore-Dijkstra 算法的优先队列实现。

参考代码

```

const int MAXV = 1010, MAXE = 1000010, INF = 0x3f3f3f3f;

struct edge {
    int v;
    long long weight;
    edge (int v = 0, long long weight = 0): v(v), weight(weight) {}
    bool operator < (const edge &e) const { return weight > e.weight; }
} edges[MAXE][4];

long long dist[MAXE];
int cases, N, M, maze[MAXV][MAXV];
int offset[4][2] = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

void mooreDijkstra(int u) {
    fill(dist, dist + N * M, INF);
    dist[u] = maze[0][0];
    priority_queue<edge> q;
    q.push(edge(0, dist[u]));
    while (!q.empty()) {
        edge e1 = q.top(); q.pop();
        for (int i = 0; i < 4; i++) {
            edge e2 = edges[e1.v][i];
            if (e2.v == 0) continue;
            if (dist[e2.v] > dist[e1.v] + e2.weight) {
                dist[e2.v] = dist[e1.v] + e2.weight;
                q.push(edge(e2.v, dist[e2.v]));
            }
        }
    }
}

int main(int argc, char *argv[]) {
    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        // 读入迷宫。
        cin >> N >> M;

```

```

for (int i = 0; i < N; i++)
    for (int j = 0; j < M; j++)
        cin >> maze[i][j];
// 根据迷宫建立无向图。
for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
        int c = i * M + j;
        for (int k = 0; k < 4; k++) {
            edges[c][k].v = 0;
            int ii = i + offset[k][0], jj = j + offset[k][1];
            if (ii >= 0 && ii < N && jj >= 0 && jj < M)
                edges[c][k] = edge(ii * M + jj, maze[ii][jj]);
        }
    }
}
// 求最短距离并输出。
mooreDijkstra(0);
cout << dist[N * M - 1] << '\n';
}
return 0;
}

```

11463 Commandos^A (敢死队)

一组敢死队员需要摧毁敌人的指挥部。敌人的指挥部由多栋建筑构成，建筑间有道路相连。敢死队员在建筑的底部安放炸弹从而将之摧毁。他们在某栋建筑集结，然后利用建筑间的道路向各栋建筑渗透破坏。敢死队员可以连续摧毁建筑，但是在最终完成任务时他们必须在某栋建筑再次集结。在本问题中，给出不同的敌人指挥部的描述，编写程序确定完成任务的最短时间。每名敢死队员从一栋建筑移动到另外一栋建筑都只需相同的单位时间。你可以忽略安置炸弹的时间，每名敢死队员能够携带数量不限的炸弹，为了完成这项任务，有数量不限的敢死队员可供派遣。

输入

输入的第一行包含一个整数 $T < 50$ ，表示测试数据的组数。每组测试数据起始为一个整数 $N \leq 100$ ，表示敌人指挥部包含的建筑栋数，接着一行包含一个正整数 R ，表示连接这些建筑的道路数量，后面的 R 行每行包含两个不同的整数 $0 \leq u, v < N$ ，表示在建筑 u 和建筑 v 之间有一条道路。建筑从 0 到 $N-1$ 进行编号。每组测试数据的最后一行包含两个整数 $0 \leq s, d < N$ ，表示敢死队员初始集结的建筑编号和完成任务后集结的建筑编号。你可以假定任意两栋建筑间至多只有一条道路相连。输入保证从任何一栋建筑出发沿着给定的道路能够到达任意其他的建筑。

输出

对于每组测试数据输出一行，包含测试数据的组数以及完成任务的最短时间。

样例输入

```

1
4
3
0 1
2 1
1 3
0 3

```

样例输出

```
Case 1: 4
```

分析

题目要求确定完成任务的最短时间，由于可以派遣数量不限的敢死队员，而且队员携带的炸弹数量也不限，则完成此项任务的最短时间取决于所有建筑均被摧毁的最晚时间。由于给定的是连通图，从起点和终点能够到达任意其他的建筑，则最短时间取决于距离起点和终点距离之和最大的那栋建筑的摧毁时间。因此可以分别从起始集结点 s 和最终集结点 d 出发，计算其他建筑和起始集结点 s 的最短距离 $dist_1$ 以及与最终集结点 d 的最短距离 $dist_2$ ，然后取距离和的最大值即为任务的最短完成时间，即

$$T_{min} = \max_{0 \leq k \leq N} \{dist_1[k] + dist_2[k]\}$$

使用前述介绍的 Moore-Dijkstra 算法分别从起点 s 和终点 d 计算与其他顶点的最短路径即可。

强化练习：318 Domino Effect^B, 388 Galactic Import^C, [10171 Meeting Prof. Miguel^A](#), [10986 Sending Email^A](#), [11374 Airport Express^D](#), [11377 Airport Setup^C](#), [12047 Highest Paid Toll^C](#), [12878 Flowery Trails^D](#)。

扩展练习：[10537 The Toll Revisited^C](#), [10874 Segments^D](#)。

11367 Full Tank?^C (油箱加满?)

这个夏天，你完成了以自驾游的形式穿越整个欧洲的壮举。事后，在整理加油账单的时候，你注意到旅行时路过的每个城市的油价是有差异的，如果更明智地选择加油的地点也许能够节省不少油钱。为了帮助其他旅行者（同时也为自己下一次旅行）省钱，你决定编写程序找到在两个城市之间旅行最便宜的加油方式。假定汽车每行驶一单位的距离消耗一单位的汽油，而且出发时油箱是空的。

输入

输入第一行包含两个整数 $1 \leq n \leq 1000$ 以及 $0 \leq m \leq 10000$ ，分别表示城市和道路的数量。接着一行包含 n 个整数 p_i ， $1 \leq p_i \leq 100$ ， p_i 表示在第 i 个城市加一个单位汽油的价格。后续 m 行，每行包含三个整数， $0 \leq u, v < n$ 以及 $1 \leq d \leq 100$ ，表示在城市 u 和城市 v 之间有一条长度为 d 的道路。然后一行包含一个整数 $1 \leq q \leq 100$ ，表示查询的数量，接着 q 行，每行包含三个整数 $1 \leq c \leq 100$, s 和 e ，其中 c 表示油箱的容量， s 表示出发城市， e 表示目标城市。

输出

对于每个查询，如果从城市 s 到城市 e 不可达，输出“impossible”，否则在给定的油箱容量限制下，确定最便宜的加油方式所需的花费。

样例输入

```
5 5
10 10 20 12 13
0 1 9
0 2 8
1 2 1
1 3 11
2 3 7
2
10 0 3
20 1 4
```

样例输出

```
170
impossible
```

分析

此题考察的是解题者的图论建模能力。实际上，问题隐含给出的是一个加权有向图，但是仅仅依靠此图尚无法解决问题，因为最便宜的加油方式不仅与在何处加油有关，而且与当时的油箱剩余容量有关。除此之

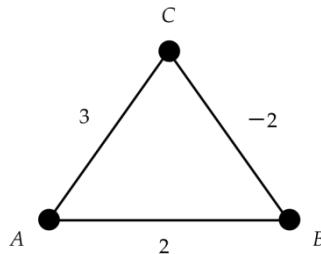
外, 还需确定从某个城市到另外一个城市是否有足够的油能够到达 (因为无法在中途进行加油)。为了表示不同城市和油量的状态, 需要两个域, 即 $(location, fuel)$, $location$ 表示某个状态所处的城市编号, $fuel$ 表示此状态下汽车油箱内所剩余的汽油数量, 则原图中的顶点将从 1000 个暴涨到约 $1000 \times 100 = 100000$ 个。在原图上构建的这个新图也称之为状态一空间 (State-Space) 图。在此状态一空间图中, 起始顶点为 $(s, 0)$, 表示在起始城市油箱为空; 终止顶点为 (e, any) , 表示在终止城市 e , 0 到 c 的任意状态油量都是可接受的。然后根据题意建边, 如果从城市 x 到 y , 汽车有足够的油能够到达, 则在状态 $(x, fuel_x)$ 和 $(y, fuel_y - length(x, y))$ 之间建立一条权值为 0 的边; 在每个城市, 从状态 $(x, fuel_x)$ 到 $(x, fuel_x + 1)$ 建立一条权值为 p_i 的边, 表示油箱在第 i 个城市从 $fuel_x$ 到达 $fuel_x + 1$ 的状态需要支付 p_i 的费用 (显然油箱加油量不能超过 c , 即有 $fuel_x \leq c$)。在新图上执行 Moore-Dijkstra 算法即可求得从起始状态到目标状态的最小费用, 最后取状态 (e, any) 的最小值即为结果。为了简便, 可以再新增一个终止状态即 $(sink, 0)$, 从 (e, any) 到 $(sink, 0)$ 引一条权值为 0 的边, 从起始状态 $(s, 0)$ 求到达终止状态 $(sink, 0)$ 的最小费用。最后检查 $(sink, 0)$ 的最小费用即可, 而不必遍历 (e, any) 的每种油箱状态以获取最小费用。

强化练习: 1025* A Spy in the Metro^C, 1027 Toll^D, 1202 Finding Nemo^D, 10354 Avoiding Your Boss^C, 10525* New to Bangladesh^D, 10967 The Great Escape^D, 11338 Minefield^D, 11833 Route Change^D, 12144 Almost Shortest Path^D, 12950 Even Obsession^D。

扩展练习: 1057* Routing^E, 1233* USHER^D, 13172 The Music Teacher^E。

10.4.2 Bellman-Ford 算法

Moore-Dijkstra 算法适用于在不包含负边权的图上解决单源最短路径问题, 如果图中存在负权值的边, 很有可能得到错误的结果。Bellman-Ford 算法^I, 能够在更一般的情况下, 即图中存在负权边时, 解决单源最短路径问题, 只要图中不包含权值总和为负值的圈^[106]。Bellman-Ford 算法使用松弛 (relaxing) 技术^{II}, 对于每个顶点 $v \in V$, 逐步减小从源点 s 到顶点 v 的最短路径权值的估计值 $dist[v]$, 直至其达到实际最短路径的权值 $\delta(s, v)$ 。



^I 1954 年, Shimbrel 在 Dijkstra 算法的基础上, 使用先进先出 (FIFO) 数据结构替换堆, 最先描述了 Bellman-Ford 算法的框架, Moore 于 1957 年给出了更为具体的算法描述, 之后由 Woodbury 和 Dantzig 于 1957 年、Bellman 于 1958 年分别独立重新发现。由于 Bellman 在算法描述中使用了 Ford 提出的松弛边的方法, 因此算法一般被称为 Bellman-Ford 算法, 而一些早期的文献则将之称为 Bellman-Shimbrel 算法或者 Shimbrel 算法。

^{II} 松弛和动态规划有密切关系, 在本书的第 11 章“动态规划”中, 从动态规划的角度对应用松弛技术的图算法进行了进一步的介绍。

图 10-9 当图中包含负权边时, Moore-Dijkstra 算法无法正确处理最短路径问题的示例。假设源点为 A , 按照 Moore-Dijkstra 算法的流程, 会立即确定与顶点 B 的最短距离为 2, 与顶点 C 的最短距离为 3, 然后标记顶点 A 为已处理顶点。接着选取与源点 A 具有最短距离的顶点 B 继续更新距离, 此时会将顶点 C 与源点 A 的最短距离更新为 0 并标记顶点 B 为已处理顶点。接着选取与源点 A 具有最短距离的顶点 C 继续更新距离, 由于顶点 A, B 均标记为已处理顶点, 更新距离的过程停止。最终得到顶点 B 与源点 A 的最短距离为 2, 顶点 C 与源点 A 的最短距离为 0。但实际上, 顶点 B 到源点 A 的最短距离为 1, 对应的最短路径为: $A-C-B$

假设有向图中有 n 个顶点且图中不存在负权值圈。如果顶点 v_1 和顶点 v_2 之间存在最短路径, 那么该路径至多有 $(n-1)$ 条边, 如果路径上的边数超过 $(n-1)$ 条, 必然会重复经过某个顶点, 从而形成圈, 当该圈的权值和为非负值时, 可以将圈从最短路径中去除, 从而使得 v_1 到 v_2 的最短路径长度缩短。Bellman-Ford 算法的过程就是通过不断的迭代, 构造一个最短路径数组序列: $dist^1[u], dist^2[u], \dots, dist^{n-1}[u]$, 其中 $dist^1[u]$ 表示从源点 s 到终点 u 只经过 1 条边的最短路径长度, 且有 $dist^1[u] = edge[s][u]$, $dist^2[u]$ 表示从源点 s 出发最多经过不构成负权值圈的 2 条边到达终点 u 的最短路径长度, \dots , $dist^{n-1}[u]$ 表示从源点 s 出发最多经过不构成负权值圈的 $n-1$ 条边到达终点 u 的最短路径长度。假设已经求出了 $dist^{k-1}[u]$, $u=0, 1, \dots, n-1$, 即从源点 s 最多经过不构成负权值圈的 $k-1$ 条边到达终点 u 的最短路径的长度, 如果采用邻接矩阵方式来表示图中的边, 那么可以使用以下的递推公式计算 $dist^k[u]$:

$$dist^1[u] = edge[s][u], s \text{ 为源点}$$

$$dist^k[u] = \min \{ dist^{k-1}[u], \min \{ dist^{k-1}[j] + edge[j][u] \} \}$$

$$j = 0, 1, \dots, n-1, j \neq u; k = 2, 3, \dots, n-1$$

在下述的 Bellman-Ford 算法实现中, 当且仅当图中不包含从源点可达的负权圈时函数返回 `true`, 否则返回 `false`。如果存在负权圈, 当继续进行迭代时, 最短距离还会减小, 因此可以利用该性质判断有向图是否包含负权值圈。

```
-----10.4.2.1.cpp-----/
// 注意 INF 的取值, 需要避免超出数据类型的表示范围, 从而防止溢出导致的结果错误。
const int MAXV = 110, MAXE = 12100, INF = 0x3f3f3f3f;

// 使用边列表来表示图。u 为边的起始顶点, v 为终止顶点, weight 为边的权值。
struct edge { int u, v, weight; } edges[MAXE];

// dist 记录最短距离, parent 记录各顶点的前驱, n 为图中顶点数量, m 为图中边的数量。
int dist[MAXV], parent[MAXV], n, m;

// s 为起始顶点的序号。
bool bellmanFord(int s) {
    // 初始化距离为无限大, 各顶点的前驱顶点为未定义。
    for (int i = 0; i < n; i++) dist[i] = INF, parent[i] = -1;
    // 起始顶点距离自身的距离为 0。
    dist[s] = 0;
    // 松弛。若顶点个数为 n, 则松弛的次数为 n-1。
    for (int k = 1, updated = 1; k < n && updated; k++) {
        // 设置标记, 当最短距离不再发生改变时及早退出。
        updated = 0;
        // 逐条边对最短距离进行更新。
        for (int i = 0; i < m; i++) {
            edge e = edges[i];
            if (dist[e.u] + e.weight < dist[e.v]) {
                dist[e.v] = dist[e.u] + e.weight;
                parent[e.v] = e.u;
                updated = 1;
            }
        }
    }
    return !updated;
}
```

```

        for (int i = 0; i < m; i++)
            if (dist[edges[i].v] > dist[edges[i].u] + edges[i].weight) {
                dist[edges[i].v] = dist[edges[i].u] + edges[i].weight;
                parent[edges[i].v] = edges[i].u;
                updated = 1;
            }
    }
    // 检查是否存在权和为负值的圈。
    for (int i = 0; i < m; i++)
        if (dist[edges[i].v] > dist[edges[i].u] + edges[i].weight)
            return true;
    return false;
}
//-----10.4.2.1.cpp-----

```

强化练习：[558 Wormholes^A](#)，[10449 Traffic^C](#)。

扩展练习：[10557* XYZZY^B](#)。

Bellman-Ford 算法的朴素实现要迭代 $|V|$ 次，每次迭代均需要扫描所有出边共 $|E|$ 次，故时间复杂度为 $O(VE)$ ，运行效率不高。究其原因，是由于在算法执行过程中存在许多不必要的判断，导致效率下降。最短路径加速算法（Shortest Path Faster Algorithm，SPFA）是 Bellman-Ford 算法的一种队列实现，减少了不必要的冗余判断，从而提高了效率^[107]。以下是 SPFA 的参考实现，使用链式前向星来表示每个顶点的出边。

```

//-----10.4.2.2.cpp-----
const int MAXV = 110, MAXE = 12100, INF = 0x3f3f3f3f;

// 每个顶点的出边表示成链式前向星。
struct edge { int u, v, weight, next; } edges[MAXE];

// dist 记录最短距离，parent 记录最短距离路径上各顶点的前驱。
int dist[MAXV], parent[MAXV];

// n 为图中顶点的数量，m 为图中边的数量。
int n, m, head[MAXV];

// visited 记录顶点是否在队列中，cnt 记录顶点进入队列的次数。
int visited[MAXV], cnt[MAXV];

// s 为起始顶点的序号。
bool spfa(int s) {
    // 初始化。
    for (int i = 0; i < n; i++)
        dist[i] = INF, parent[i] = -1, visited[i] = 0, cnt[i] = 0;
    // 压入源点，开始松弛过程。
    dist[s] = 0, visited[s] = 1;
    queue<int> q; q.push(s);
    while (!q.empty()) {
        int u = q.front(); q.pop();
        // 某个顶点进入队列次数超过图中顶点数量，表明图中存在负权值圈。
        if (cnt[u] > n) return true;
        // 标记顶点为未访问状态。
        visited[u] = 0;
        // 遍历顶点的出边，更新最短距离。
        for (int i = head[u]; ~i; i = edges[i].next) {

```

```

        int v = edges[i].v, w = edges[i].weight;
        if (dist[v] > dist[u] + w) {
            dist[v] = dist[u] + w;
            parent[v] = u;
            if (!visited[v]) {
                q.push(v);
                visited[v] = 1;
                cnt[v]++;
            }
        }
    }
    return false;
}
//-----10.4.2.2.cpp-----//

```

在 SPFA 中, 如果每个顶点都进入队列一次, 则在遍历边时每条边只会扫描一次, 时间复杂度为 $O(E)$, 假设每个顶点平均进入队列的次数为 k 次, 则算法的时间复杂度为 $O(kE)$, 实验表明 k 远小于 $|V|$ (一般来说 k 为 2 左右, 特殊情况下 k 接近 $|V|$, 具体值和图的结构有关, 算法效率是一个不确定值)。若某个顶点进入队列的次数超过 $|V|$ 次, 则表明此图中存在负权值圈, 且此顶点位于该负权值圈上, 可以使用适当的方式来记录每个顶点进入队列的次数, 例如上述示例代码中的 cnt 数组。

11721 Instant View of Big Bang^D (见证宇宙大爆炸)

虫洞能够将两个处于不同时空的恒星系统连接起来。虫洞具有以下性质: (1) 单向通行; (2) 通过虫洞的时间可以忽略不计; (3) 每个虫洞具有一个入口和一个出口, 分别位于不同的恒星系统中; (4) 每个恒星系统可能包含不止一个虫洞入口或出口; (5) 从某个恒星系统出发, 直接到达另外一个恒星系统的虫洞最多只有一个; (6) 某个虫洞的出入口不会同时在一个恒星系统中。所有虫洞在其出入口之间存在固定的时间差。比如, 某个虫洞能够使得穿越它的人到达 15 年之后, 而另外一个虫洞可能使得穿越者回到 42 年以前。一位才华横溢的物理学家想利用虫洞来研究宇宙大爆炸 (Big Bang)。因为曲速引擎尚未发明, 所以无法直接在任意两个恒星系统之间进行时间跃迁旅行。当然, 可以通过虫洞间接实现这个目标。该物理学家可以从某个合适的恒星系统出发, 之后进入一个虫洞环, 通过在虫洞环中不断地穿越, 从而到达过去的任意时刻, 这样她就能够亲眼见证宇宙大爆炸的那一刻。编写程序帮助她找出可以从哪些恒星系统开始她的旅程。

输入

输入的第一行包含整数 T , 表示测试数据的组数。每组测试数据以一个空行开始, 接着一行包含两个整数 n 和 m , n 表示恒星系统的数量 ($1 \leq n \leq 1000$), m 表示虫洞的数量 ($0 \leq m \leq 2000$)。恒星系统的编号从 0 到 $n-1$ 。接着每个虫洞一行数据, 包含三个整数 x , y 和 t , 表示该虫洞允许旅行者从恒星系统 x 出发到达恒星系统 y , 最终的时刻为 t ($-1000 \leq t \leq 1000$), 如果 t 为负数则表示到达 t 年之前, 否则表示到达 t 年之后。

输出

对于每组测试数据, 先输出测试数据的序号, 接着按照升序输出符合要求的出发恒星系统编号。如果不存在这样的出发恒星系统, 输出 “impossible”。

样例输入

2

样例输出

Case 1: 0 1 2
Case 2: impossible

```

3 3
0 1 1000
1 2 15
2 1 -42

```

```

4 4
0 1 10
1 2 20
2 3 30
3 0 -60

```

分析

本题可以归结为以下问题：给定加权有向图，确定图中是否存在权和为负值的圈，如果存在负权值圈，找出位于负权值圈上的顶点以及能够通过有向边到达负权值圈的顶点。使用前述介绍的 Bellman-Ford 算法可以容易地确定有向图是否包含负权值圈，但是如何确定哪些顶点在负权值圈上呢？设有向图的顶点个数为 n ，使用 Bellman-Ford 算法检测负权值圈的方法是以某个顶点作为起始顶点，进行 $n-1$ 次迭代，更新从起始顶点到其他顶点的最短距离，在进行 $(n-1)$ 次迭代后，再对得到的最短距离进行一次检查，如果对于某个顶点还能够获得更小的最短距离，则表明图中存在负权值圈。但是在检查过程中，满足检测条件的顶点却不一定位于负权值圈上。

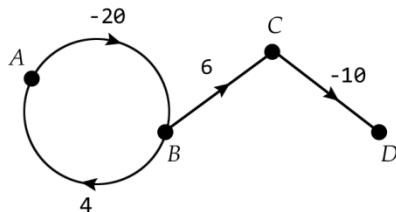


图 10-10 在进行第 n 次迭代时，最短距离继续变小的顶点不一定位于负权值圈上。以包含四个恒星系统的有向图为例，假设顶点 A 为起始顶点，经过 3 次迭代后， $dist[A] = -48$ ， $dist[B] = -52$ ， $dist[C] = -46$ ， $dist[D] = -56$ 。为进行负权值圈检查而进行第 4 次迭代，此时 $dist[B] > dist[A] + (-20)$ ， $dist[B]$ 更新为 -68 ，接着 $dist[A]$ 、 $dist[C]$ 、 $dist[D]$ 会因为 $dist[B]$ 的更新而相继更新，但顶点 C 和 D 并不在负权值圈上，且从顶点 C 或 D 出发也无法到达负权值圈 $A-B-A$ 上。

解决此问题的技巧是在最初构建有向图时将全部有向边反向。在边反向之后，有向圈仍然保持不变，且此时满足检测条件的顶点必定位于负权值圈上，或者从负权值圈上的顶点通过一系列有向边可达。否则按照 Bellman-Ford 算法的性质，在前 $(n-1)$ 次迭代后，这些顶点与起始顶点的最短距离就会固定下来，不再发生改变，除非该顶点位于负权值圈上，或者位于负权值圈上的顶点能够到达的路径上，才有可能在后续第 n 次迭代更新距离的过程中发生最短距离的改变。

在编码实现时，由于 Bellman-Ford 算法需要指定一个起始顶点，而从原有的任意一个恒星系统中选择一个作为起始顶点都可能造成部分顶点的最短距离无法更新。为了解决这一问题，可以将原有恒星系统的编号递增 1，虚拟一个编号为 0 的顶点，从此虚拟顶点向其他所有顶点各引一条权值为 0 的有向边，在 Bellman-Ford 算法中，以编号为 0 的顶点作为起始顶点，求所有其他顶点相对于此虚拟顶点的最短距离。使用 SPFA 进行编码实现相对于前述方法更为简便，此时判断某个顶点是否在负权值圈上（或从负权值圈可达）的依据是此顶点是否已经进入队列超过 n 次。同样的，在建立图结构时，需要使用反向边的方式建立

图。

在获得在负权值圈上（或从负权值圈可达）的顶点后，可能还不是题意所要求的全部解，因为有可能会有部分顶点不能被前述的检测过程“筛选”出来，此时可以通过以已经“筛选”的符合要求的顶点作为起点，在反向图上进行一次 DFS，则所有能够到达的顶点均是符合要求的解。

强化练习：10278 Fire Station^B，11280 Flying to Frederiction^D。

扩展练习：11090* Going in Cycle^{C[108]}，11097 Poor My Problem^D。

10.4.3 Floyd-Warshall 算法

如果需要确定图中每对顶点间的距离，一种方法是对每个顶点都运行一次 Moore-Dijkstra 算法来得到结果。而使用 Floyd-Warshall 算法，可以更为方便地计算每对顶点间的最短路径^[109]。该算法使用了动态规划的思想，同样采用了松弛技术。虽然算法的基本框架很简单，但是透彻理解算法的精髓却需要进行深入的思考^I。

```
const int MAXV = 110, INF = 0x3f3f3f3f;

int n, dist[MAXV][MAXV];

void floydWarshall() {
    memset(dist, INF, sizeof(dist));
    for (int i = 0; i < n; i++) dist[i][i] = 0;
    // 读入图数据，确定初始边权矩阵。
    for (int k = 0; k < n; k++)
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j]);
}
```

算法包含了一个三重循环，最外层的循环表示当前松弛的顶点，也就是说在仅通过前 k 个顶点的情况下，从顶点 i 到顶点 j 的最短路径，由于最短路径具有最优子结构的性质，亦即最短路径的子路径同样是最短路径，则在求出仅通过前 $k-1$ 个顶点的最短路径后就可以求出仅通过前 k 个顶点的最短路径。注意“无限大”—— INF ——的取值要恰当，使得在进行操作 $\text{dist}[i][k] + \text{dist}[k][j]$ 时不发生溢出，从而保证结果的正确性。一般来说，取 $\text{INF} = 0x3f3f3f3f$ 在绝大多数情况下都是合适的。

在确定最短路径的过程中，**可能还需要确定最短路径经过了哪些顶点**，这可以通过在更新最短距离时记录从顶点 i 到顶点 j 的最短路径是由哪个顶点作为中间顶点达到的，即：

```
const int MAXV = 110;
int parent[MAXV][MAXV];

for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        parent[i][j] = i;

for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (dist[i][j] > dist[i][k] + dist[k][j])
                dist[i][j] = dist[i][k] + dist[k][j];
                parent[i][j] = k;
```

^I 对于 Floyd-Warshall 算法使用松弛技术的进一步分析，请读者参阅本书第 11 章“动态规划”中第 11.3 节“松弛”的内容。

```

for (int j = 0; j < n; j++) {
    int d = dist[i][k] + dist[k][j];
    if (dist[i][j] > d) {
        dist[i][j] = d;
        parent[i][j] = parent[k][j];
    }
}

```

这样可以依据 *parent* 数组所记录的信息，通过递归重建得到最短路径的顶点序列。

```

void printPath(int i, int j)
{
    if (i != j) {
        printPath(i, parent[i][j]);
        cout << ' ';
    }
    cout << j;
}

```

如果图中存在圈，但是圈中的边均具有正的边权值，Floyd-Warshall 算法仍然是适用的。若圈中包含了负的边权值，使得整个圈的权值之和为负值，那么算法会失效，因为可以无限次经过这个圈，导致最短路径的长度仍可不断减小。Floyd-Warshall 算法的时间复杂度为 $O(V^3)$ ，因此只适用于顶点数量较少的图。

104 Arbitrage^A (套利)

套利是指在货币交易中，利用各种货币兑换率之间的微小差异来获利的行为。例如，如果 1 美元能够兑换 0.7 英镑，1 英镑能够兑换 9.5 法郎，1 法郎能够兑换 0.16 美元，那么套利者就能够从 1 美元开始，通过不断地兑换得到 $1 \times 0.7 \times 9.5 \times 0.16 = 1.064$ 美元，最终获取 6.4 美分的利润。编写程序以确定给定的货币兑换过程是否能够获得上述的额外利润。为了实现一次成功的套利，在货币兑换过程中，起始货币和最终货币必须相同，但任何货币种类都可以作为起始货币。

输入

输入包含一张或多张兑换表，要求程序确定每张兑换表中是否存在套利序列。每张兑换表以描述表大小的数字 n 开始， n 最大不超过 20，最小为 2。每张兑换表以行优先顺序给出，但是省略了对角线上的表元素（这些元素的值为 1.0），所以表的第一行表示国家 1 和其他 $(n-1)$ 个国家之间的货币兑换比率 c ，即国家 i ($2 \leq i \leq n$) 的指定数量 c 的货币能够使用国家 1 的一个单位的货币进行兑换。在输入文件中，每张兑换表由 $n+1$ 行构成，第一行包含 n ，随后的 n 行包含各个国家之间的货币兑换比率。

输出

对于输入中给出的每张兑换比率表，确定是否存在一个货币兑换序列，使得获利能够大于 1 分钱(0.01)。如果存在该兑换序列则予以输出。如果存在多个符合要求的兑换序列，输出具有最小长度的兑换序列，即具有最少兑换次数的兑换序列。由于美国国税局 (United State Internal Revenue Service, IRS) 关注长的交易序列，因此所有的获利序列必须由不超过 n 次的货币兑换构成，其中 n 表示兑换表的维度大小。兑换序列 “1 2 1” 表示两次兑换。如果不存在上述交易序列则输出 “no arbitrage sequence exists”。

样例输入

```

3
1.2 .89
.88 5.1

```

样例输出

```

1 2 1

```

1.1 0.15

分析

题目要求找出有向图中一个具有最少边数且边权的积大于或等于 1.02 的圈。可以结合动态规划和 Floyd-Warshall 算法(实际上 Floyd-Warshall 算法本身就已经应用了动态规划的思想)来解决本题。令 $profit(i, j, s)$ 表示“由货币种类 i 经过 s 步套汇到货币种类 j 时能够得到的最大获利值”，则有以下递推关系

$$profit(i, j, s) = \max_{0 \leq k < n, k \neq i, k \neq j} \{profit(i, k, s-1) \times profit(k, j, 1)\}, \quad 2 \leq s \leq n$$

更新当前值的条件是: $profit(i, k, s-1) \times profit(k, j, 1) > profit(i, j, s)$, 递推初始值: $profit(i, i, 1) = 1.0$ 。

强化练习: 125 Numbering Paths^B, 336 A Node Too Far^A, 383 Shipping Routes^A, 423 MPI Maelstrom^A, 567 Risk^A, 593 MBone^D, 1056 Degrees of Separation^C, 1247 Interstar Transport^D, 10724 Road Construction^D, 10793 The Orc Attack^C, 10803 Thunder Mountain^B, 11015 05-2 Rendezvous^B, 12319 Edgetown's Traffic Jams^D, 13249 A Contest to Meet^D。

扩展练习: 208* Firetruck^B, 436 Arbitrage (II)^B, 523 Minimum Transport Cost^C, 1198 The Geodetic Set Problem^D, 10331* The Flyover Construction^D, 10987* Antifloyd^D, 11047 The Scrooge Co Problem^D, 11693* Speedy Escape^D。

10.4.4 传递闭包

设 R 是集合 A 上的二元关系, 如果对于任意的 $a, b, c \in A$, 若 $(a, b) \in R, (b, c) \in R$, 必有 $(a, c) \in R$, 则称 R 是 A 上的传递的二元关系, 简称为 R 是 A 上的传递关系 (transitive relation), 而 R 的传递闭包 (transitive closure) 是包含 R 并且具有传递关系性质的最小二元关系, 记做 $t(R)$ 。例如, 设 A 是中国所有省会城市的集合, 定义 aRb 为 A 中的两个省会城市 a 和 b 之间有直达航班, 假设 R 是对称的, 即 $(a, b) \in R$, 必有 $(b, a) \in R$, 那么 R 的传递闭包即为最小二元关系 $t(R) = \{(a, b), a \in A, b \in A\}$, 从城市 a 可飞往城市 b (直达或经过中转) }。

给定有向图 $G = (V, E)$, 定义 iRj 当且仅当顶点 i 和顶点 j 之间有一条有向边 $e(i, j) \in E$, 那么有向图 G 的传递闭包 $t(R) = \{(i, j), i \in V, j \in V, \text{ 从顶点 } i \text{ 可到达顶点 } j \text{ (经过若干个中间顶点)}\}$ 。需要注意的是, 对于有向图来说, 传递关系一般是不对称的, 顶点 i 可以通过顶点 k 到达顶点 j 并不代表顶点 j 可以通过顶点 k 到达顶点 i , 而无向图中, 顶点 i 和顶点 j 之间的连通性是对称的。

求解传递闭包的算法称为 Warshall 算法, 由 Stephen Warshall 于 1960 年给出。算法采用动态规划思想, 利用了松弛的技巧。因为与求解所有顶点对之间最短距离的 Floyd 算法思想本质相同, 因此合称为 Floyd-Warshall 算法。算法的具体过程是通过一系列 n 阶矩阵 A^k 来构造最终阶段 n 阶传递闭包矩阵 A^n 。其中:

A^0 : 表示该矩阵不允许它的路径中包含任何中间顶点, 即从该矩阵的任意顶点出发的路径不含有中间顶点, 此即图 G 的邻接矩阵;

A^1 : 表示允许路径中包含第 1 个顶点作为中间顶点;

A^2 : 表示允许路径中包含前 2 个顶点作为中间顶点;

A^k : 表示允许路径中包含前 k 个顶点作为中间顶点;

...

A^n : 允许路径中包含全部 n 个顶点作为中间顶点。

每个后继矩阵 A^k 对其前驱矩阵 A^{k-1} 来说，在路径上只允许增加一个顶点。在初始时，邻接矩阵 A 对应算法中的 A^0 ，若 A 中的元素 $A[i, j]$ 为 1，表示顶点 i 与顶点 j 连通，为 0 则表示不连通。接下来依次增加可以作为中间顶点的数量，检查是否可以通过这些中间顶点到达目标顶点，对于 A^k 来说，如果 A^{k-1} 中 $A^{k-1}[i, j]=1$ ，那么有 $A^k[i, j]=1$ ，如果有 $A^{k-1}[i, k]=1$ 且 $A^{k-1}[k, j]=1$ ，也有 $A^k[i, j]=1$ 。由于 A^k 依赖于 A^{k-1} ，存储的 A^{k-1} 在 A^k 计算完毕后不再需要，因此可以利用滚动数组的技巧，将计算在原有数组上进行，这样可以将空间复杂度从 $O(V^3)$ 降低为 $O(V^2)$ 。

```
// 由邻接矩阵计算所有点对间的可达性。
const int MAXV = 110;

int n;
bool A[MAXV][MAXV];

// Floyd-Warshall 算法。
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            A[i][j] = A[i][j] || (A[i][k] && A[k][j]);
```

不难看出，上述实现相当于将求解任意点对间最短距离中的“`min`”和“`+`”运算换成了“布尔或”(`||`)及“布尔与”(`&&`)运算。由算法可知，只要知道了初始的传递关系，也就是 A^0 ，就可以使用 Floyd-Warshall 算法计算传递闭包，因此在实际解题应用中，关键在于构造初始的邻接矩阵。Floyd-Warshall 算法的时间复杂度为 $O(V^3)$ 。

521 Gossiping^D (闲聊)

某镇拥有自己的公交系统。公交车按照环形方式开行，且每条线路至少有两个公交站，有时候多条线路会在同一个公交站交汇。当多名司机在同一个公交站会车时，他们会互相透露自己所知道的消息，这样在离开时，所有在此公交站闲聊的司机均知道了同样的消息。每名司机在每天的同一个时刻启动公交车，而且知道一些其他司机均不知道的消息。每辆公交车均按照固定的公交线路开行，有时候，可能会出现多名司机在同一条线路上开行的情况，只不过他们的初始站点不一样。所有公交车的运行都是高度同步的。对于任意公交线路，从一个公交站到达下一个公交站所需的时间都是一样的。公交系统总共拥有 n ($0 < n < 20$) 条线路， d ($0 < d < 320$) 名司机， s ($0 < s < 50$) 个公交站，司机按照 1 到 d 的顺序进行编号，公交站按照 1 到 s 的顺序进行编号。司机闲聊俱乐部乐于知道是否存在这样一个时刻——在该时刻，所有司机都从其他司机那里知道了所有的消息。

输入

输入包含多组测试数据。每组测试数据的第一行包含以单个空格分隔的三个整数 n, d, s ，其含义如前所述。接下来的 $2n$ 行包含 n 条线路的描述，每条线路使用两行数据来描述。第一行是以单个空格相间隔的编号列表，表示在此线路上，按照公交车经过的顺序给出的公交站编号，公交车在经过线路的最后一个站后

立即返回起始站点继续开行^I。第二行描述在此条公交线路上开行司机的始发情况，描述包括若干组数值对 s_i 和 d_i ，表示编号为 d_i 的司机从编号为 s_i 的公交站启动公交车。最后一组测试数据只包含一行数据“0 0 0”，该组数据不需处理。

输出

对于输入的每组数据，除了最后一组之外，如果存在某个时刻，所有司机都从其他司机那里知道了所有的消息，输出“Yes”，否则输出“No”。

样例输入

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

样例输出

```
Yes
```

分析

本题是典型的确定隐式图中顶点间连通性的问题，可以使用 Floyd-Warshall 算法予以解决，解题关键是构造初始的邻接矩阵。由题意可知，每个司机都具有相应的开行线路和始发站点（始发站点可能并不是线路的第一个站点），司机从始发站点出发到达线路的最后一个站点，然后返回线路的第一个站点，之后再到达始发站点时就完成了一个周期的运行。如果两名司机的公交线路站点之间没有交叉或重叠，那么他们是无法在同一站点停靠的，进而无法通过闲聊交换信息。如果两名司机的线路之间有交叉或重叠，不妨设有 a 和 b 两名司机，他们各自所在线路一个运行周期所需时间分别为 x 单位时间和 y 单位时间，只需检查在 $x \times y$ 单位时间内 a 和 b 是否能够在同一站点停靠即可，如果不能在同一站点停靠，在此后的短时间内，两名司机肯定也不会在同一站点停靠，因为在经过了 $x \times y$ 单位时间后两名司机于同一时刻各自回到了他们的始发站点，开始了下一个运行周期。由于题目中站点数量并不是很多，可以对公交车的运行情况进行模拟，列出两名司机经过的站点编号，逐次检查对应时间点的站点编号是否相同。

强化练习：[334 Identifying Concurrent Events^D](#)，[869 Airline Comparison^C](#)，[1757 Secret Chamber at Mount Rushmore^A](#)。

扩展练习：[925 No More Prerequisites Please^D](#)，[1243* Polynomial-Time Reductions^D](#)。

10.4.5 最小化的最大距离

给定图 G ，两个顶点 v_1 和 v_n 之间存在 m 条不同的简单路径。如果将路径按照经过顶点的顺序逐次列出的方式来表示，为 v_1, v_2, \dots, v_n ，其中 v_1 为起点， v_n 为终点，定义 $w_i(v_j, v_{j+1})$ 为第 i 条路径上两个相邻顶点间的边权值，则最小化的最大距离（minmax-distance）定义为

^I 此处原文为“After passing the last stop listed on the line the bus goes again to the first stop listed on the line.”，从字面上理解容易产生歧义。假如某条公交线路的停靠站点为“1 2 3 4 5 6”，司机每天从站点 3 启动公交车，那么公交车依次经过的站点可能有两种理解，一种是经过站点 6 之后按原路返回，这样所经过的站点依次是“3 4 5 6 5 4 3 2 1 2 3 4 5 6 …”，一种是经过站点 6 之后直接返回站点 1 开始，这样经过的站点依次是“3 4 5 6 1 2 3 4 5 6 1 2 3 4 5 6 …”，出题者所要表达的是后一种情况。不过在真实生活中，前一种情况应该更常见。

$$\minmax(v_1, v_n) = \min\{\max\{w_i(v_j, v_{j+1}), 1 \leq j \leq n-1\}, 1 \leq i \leq m\}$$

即所有路径中两个顶点间的边权值最大值中的最小值。

最大化的最小距离 (maxmin-distance) 和最小化的最大距离类似, 即所有路径中两个顶点间的边权值最小值中的最大值。这两种距离均可以使用 Floyd-Warshall 算法予以计算。

```
const int MAXV = 110;

int n, dist[MAXV][MAXV];

// 使用 Floyd-Warshall 算法计算最小化的最大距离。
for (int k = 0; k < n; k++)
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            dist[i][j] = min(dist[i][j], max(dist[i][k], dist[k][j]));
```

强化练习: 534 Frogger^A, 544 Heavy Cargo^A, 10048 Audiophobia^A, 10099 The Tourist Guide^A。

10.4.6 差分约束系统

给定以下的不等式组

$$\begin{cases} x_1 - x_2 \leq 3 \\ x_1 - x_3 \leq -1 \\ x_1 - x_4 \leq -2 \\ x_2 - x_3 \leq -5 \\ x_2 - x_4 \leq -3 \\ x_3 - x_4 \leq 3 \end{cases}$$

每个不等式都是两个未知数的差小于等于某个常数的形式 (如果不等式是大于等于某个常数的形式, 可以将不等式两边同时乘以 -1 , 将其转换成小于等于某个常数的形式), 称这样的不等式组为差分约束系统 (system of difference constraints)。

在前述介绍的单源最短路径问题中, 最终求得的最短路径长度均满足以下三角不等式 (triangle inequality)

$$dist(v) \leq dist(u) + weight[u][v]$$

在上述不等式中, $dist(u)$ 和 $dist(v)$ 是从源点 s 到顶点 u 和 v 的最短路径长度, $weight[u][v]$ 是有向边 (u, v) 的权值。将不等式移项可得

$$dist(v) - dist(u) \leq weight[u][v]$$

该形式和差分约束系统中的不等式类似, 因此可以把一个差分约束系统转化为一个有向图, 进而使用求单源最短路径的方法进行求解。

有向图可按照以下方法构造^[110]:

- (1) 不等式组中的每个未知数 x_i 对应图中的一个顶点 v_i ;
- (2) 将所有不等式转化为图中的一条有向边, 对于不等式

$$x_i - x_j \leq c$$

将其转化为三角不等式

$$x_i \leq x_j + c$$

那么就可以转化为有向边 (v_j, v_i) , 边权值为 c ;

(3) 对构造得到的有向图求单源最短路径, 由最短路径的性质可知求得的最短路径长度必定满足所有三角不等式。

既然是求单源最短路径，必须要选定一个顶点作为源点。如果构造得到的是一个连通图，那么任意选择一个顶点作为源点均是可行的。然而，在绝大多数情况下，构造得到的有向图是非连通图，此时需要自行构造一个源点。对于上述的不等式组，可以为其再增加一个未知数 x_0 ，然后对原来的每个未知数 x_i 相对于 x_0 增加一个不等式，增加的不等式和原有的不等式形式相同，即都是两个未知数的差小于等于某个常数的形式。考虑到不等式要么无解，要么就有无数组解（因为如果有一组解 $\{x_1, x_2, \dots, x_n\}$ ，则将解加上一个常数 k 后， $\{x_1+k, x_2+k, \dots, x_n+k\}$ 肯定也是一组解，调整 k 的大小，肯定可以使得所有的 x_i 均小于等于 0），又因为 x_0 对应源点 s ，而源点的最短路径长度为 0，也就是说不等式组中存在条件

$$x_0 = 0$$

进而不等式组中可以增加下述条件

$$\begin{cases} x_1 - x_0 \leq 0 \\ x_2 - x_0 \leq 0 \\ x_3 - x_0 \leq 0 \\ x_4 - x_0 \leq 0 \end{cases}$$

这样转化得到的有向图就保证从源点 s 到其他顶点间至少存在一条路径。

对于上述构造得到的有向图来说，可以证明，不等式有解的充分必要条件是有向图中不存在负权值圈。根据前述的 Bellman-Ford 算法，可以容易地判断有向图中是否存在负权值圈。如果图中不存在负权值圈，则最后求得的从源点到各顶点的最短路径长度即是不等式的一组解。

515 King^C (国王)

从前，有一个王国的皇后怀孕了，她祈祷到：如果我生的是儿子，我希望他是一个健康的国王。九个月后，她的孩子出生了，的确，她生了一个漂亮的儿子。

但不幸的是，正如皇室家庭经常发生的那样，皇后的儿子智力发育迟钝。经过多年的学习后，也只能做整数的加法，以及比较加法的结果比给定的一个整数是大还是小。另外，用来求和的数必须排列成一个序列，因为他只能对序列中连续的整数进行求和。

老国王对他的儿子非常不满意。但他还是决定准备好一切，以便在他去世后，他的儿子能够统治王国。考虑到他儿子的能力，他规定国王需要决断的所有问题都必须表示成有限的整数序列，并且国王需要决断的只是判断这个序列的和与给定的一个约束的大小关系。作出这样的规定后，至少还有一些希望，可以让他的儿子作出一些决策。

老国王去世后，新国王开始统治王国。但很快，许多人开始不满意他的决策，决定废黜他。人们通过试图证明新国王的某些决策是错误的，从而能够名正言顺地废黜新国王。

因此，试图篡位的人们给新国王出了一些题目，让国王作出决策。问题是序列 $S = \{a_1, a_2, \dots, a_n\}$ 中取出一个子序列 $S_i = \{a_{s_i}, a_{s_i+1}, \dots, a_{s_i+n_i}\}$ ，国王有一分钟的时间可以思考，然后必须作出判断。他对子序列 S_i 中的整数求和，即 $a_{s_i} + a_{s_i+1} + \dots + a_{s_i+n_i}$ ，然后对每个子序列的和设定一个约束 k_i ，即 $a_{s_i} + a_{s_i+1} + \dots + a_{s_i+n_i} < k_i$ 或 $a_{s_i} + a_{s_i+1} + \dots + a_{s_i+n_i} > k_i$ 。

过了一会，他意识到他的判断是错误的。他不能取消他所设定的约束，但是他可以通过伪造篡位者给他的整数序列来挽救自己。他命令他的幕僚找出能够满足他所设定的这些约束条件的一个序列 S 。请编写程序来帮助国王的幕僚，判断这样的序列是否存在。

输入

输入文件中包含多组测试数据。除最后一组外，每组测试数据对应一组问题和国王关于该组数据的决策。每组数据的第一行包含两个整数 n 和 m ，其中 $0 < n \leq 100$ 表示序列 S 的长度， $0 < m \leq 100$ 为子序列 S_i 的个数。

数。接下来有 m 行国王的决策，每个决策的格式为： s_i, n_i, o_i, k_i ，其中 o_i 代表关系运算符“ $>$ ”（用“`gt`”表示）或者“ $<$ ”（用“`lt`”表示）， s_i, n_i 和 k_i 的含义如题目描述中所述。最后一组数据只有一行，包含一个 0，表示输入结束，该组数据不需处理。

输出

对输入中的每组数据，输出一行字符串。当满足约束的序列 S 不存在时，输出“`successful conspiracy`”，否则输出“`lamentable kingdom`”。

样例输入

```
4 2
1 2 gt 0
2 2 lt 2
0
```

样例输出

```
lamentable kingdom
```

分析

对于一个给定的序列 $S = \{a_1, a_2, \dots, a_n\}$ ，为其添加一个元素 $a_0 = 0$ ，使其成为序列 $S = \{a_0, a_1, a_2, \dots, a_n\}$ ，假设 A_j 表示从第 0 个元素到第 j 个元素的和，即

$$A_j = \sum_{i=0}^j a_i, \quad 0 \leq j \leq n$$

那么序列 S 的任意一个子序列 $S_i = \{a_{s_i}, a_{s_i+1}, \dots, a_{s_i+n_i}\}$ 的和 S_{s_i} 可以表示成

$$S_{s_i} = A_{s_i+n_i} - A_{s_i-1}, \quad s_i \geq 1, \quad n_i \geq 0$$

对于题目给定的限制条件

$$A_{s_i+n_i} - A_{s_i-1} > k_i \text{ 或 } A_{s_i+n_i} - A_{s_i-1} < k_i$$

由于都是整数，可以将大于、小于不等式转化为大于等于、小于等于不等式，得

$$A_{s_i+n_i} - A_{s_i-1} \geq k_i + 1 \text{ 或 } A_{s_i+n_i} - A_{s_i-1} \leq k_i - 1$$

再将大于等于不等式转化为小于等于不等式，得

$$A_{s_i-1} - A_{s_i+n_i} \leq -k_i - 1 \text{ 或 } A_{s_i+n_i} - A_{s_i-1} \leq k_i - 1$$

也就是说，可以将题目所给定的约束关系转化为一个小于等于不等式组，进而构成一个差分约束系统。对于 $A_{s_i-1} - A_{s_i+n_i} \leq -k_i - 1$ ，可以将其转化为有向边 $(A_{s_i+n_i}, A_{s_i-1})$ ，权值为 $-k_i - 1$ 。对于 $A_{s_i+n_i} - A_{s_i-1} \leq k_i - 1$ ，可以将其转化为有向边 $(A_{s_i-1}, A_{s_i+n_i})$ ，权值为 $k_i - 1$ 。最后使用 Bellman-Ford 算法来判断该差分约束系统所对应的有向图中是否存在负权值圈，如果不存在负权值圈则不等式组有解，相应的满足约束条件的序列存在，反之则满足约束条件的序列不存在。

在下述参考实现中，使用边列表的方式来存储有向边。由于为序列增加了一个值为 0 的首元素 a_0 以使得不等式组能够构建，在构造源点时，需要调整序列和 A_j 与顶点序号间的对应关系，即源点 s 对应顶点 v_0 ， A_0 对应顶点 v_1 ， A_1 对应顶点 v_2 ， \dots ， A_n 对应顶点 v_{n+1} ，因此有向图中总共有 $(n+2)$ 个顶点。

参考代码

```
// 使用边列表方式来表示有向图。
struct edge { int u, v, weight; } edges[1024];
int nedges, dist[1024], n, m, si, ni, ki;
string oi;
```

```

// 为有向图添加边。
void addEdge(int u, int v, int weight) { edges[nedges++] = edge{u, v, weight}; }

int main(int argc, char *argv[]) {
    while (cin >> n, n > 0) {
        cin >> m;
        // 将约束关系转化为有向边。
        nedges = 0;
        for (int i = 1; i <= m; i++) {
            cin >> si >> ni >> oi >> ki;
            if (oi.front() == 'g') addEdge(si + ni + 1, si, -ki - 1);
            else addEdge(si, si + ni + 1, ki - 1);
        }
        // 调整顶点数量, 添加源点与其它顶点的有向边。
        n += 2;
        for (int i = 1; i < n; i++) addEdge(0, i, 0);
        // 初始化源点到其他顶点的最短路径距离。
        dist[0] = 0;
        for (int i = 1; i < n; i++) dist[i] = (1 << 30);
        // 使用 Bellman-Ford 算法检查是否存在负权值圈。
        int iterations = 0, updated = 0;
        do {
            updated = 0;
            for (int i = 0; i < nedges; i++) {
                int weight = dist[edges[i].u] + edges[i].weight;
                if (dist[edges[i].v] > weight)
                    updated++, dist[edges[i].v] = weight;
            }
        } while (updated && iterations++ < n);
        cout << (updated ? "successful conspiracy\n" : "lamentable kingdom\n");
    }
    return 0;
}

```

强化练习: 522 Schedule Problem^E。

扩展练习: 1723* Intervals^E, 11478* Halum^D。

10.4.7 第 K 短路径问题

第 K 短路径 (the K Shortest Path, KSP) 问题是对最短路径问题的扩展。众所周知, 使用 Dijkstra 算法能够确定最短路径, 但是在某些情形下, 我们除了想知道最短路径外, 还想知道次短路径、第三短路径等等。一个现实中的例子就是在使用手机地图应用程序的时候, 需要从某个出发点到达一个目标点, 在查询驾车路线时, 地图应用程序一般会给出三条候选路径, 一条路径具有最少的行车时间, 一条具有最短的行车距离, 另外一条是备用路线, 这三条路径可能具有不同的路径长度, 地图应用程序确定这些路径就是第 K 短路径问题的实际应用。

为了便于讨论, 首先对 KSP 问题进行形式化的定义。令 P_i 表示从起点 s 到终点 t 的第 i 短路径, KSP 问题是确定路径集合 $P_k = \{p_1, p_2, p_3, \dots, p_k\}$, 使得 P_k 满足以下三个条件:

- (1) K 条路径是按次序产生的, 即对于所有的 i ($i=1, 2, \dots, K-1$), p_i 是在 p_{i+1} 之前确定的;
- (2) K 条路径是按长度由小到大排列的, 即对于所有的 i ($i=1, 2, \dots, K-1$), 均有 $c(p_i) < c(p_{i+1})$;
- (3) K 条路径是最短的, 即对于所有的 $p \in (P_{st} - P_k)$, 均有 $c(p_k) < c(p)$ 。

某些情况下, 可能并不需要上述严格的第 K 短路径, 即可能并不满足第 (2) 个或者第 (3) 个条件,

在算法中很容易实现，只需将路径长度的限制从原先的“不能相同”放松到“能够相同”即可。求解第 K 短路径有多种算法，以下介绍最为常用的 A* 算法。

A* 算法是一种启发式算法，在第 8 章“回溯法”的第 8.4 节“15 数码问题”中，已经对 A* 算法做了介绍。在 A* 算法中，对于每个状态 x ，启发函数 $f(x)$ 具有以下的形式

$$f(x) = g(x) + h(x)$$

其中 $g(x)$ 表示从初始状态到达状态 x 时的代价， $h(x)$ 表示从状态 x 到达目标状态时的估算代价。状态 x 包含两个域： u 和 fx ， $x.u$ 表示状态 x 所在的顶点， $x.fx$ 表示状态 x 已经走过的路径长度，按照前述的约定可知 $g(x) = x.fx$ 。

简便起见，先介绍如何解决有向图中的第 K 短路径问题，第 K 短路径中可以包含圈，但是有向图中不能出现负权圈（负权圈的出现可能导致最短路径失去意义，因为有可能通过无限次经过负权圈使得最短路径无限短）。典型的启发式搜索算法步骤如下：

(1) 以终止顶点 t 为起点，使用 Moore-Dijkstra 算法确定 t 到其他顶点 u 的最短距离 $d[u]$ ，这样在启发式搜索过程中，可以使用 $d[u]$ 的值作为 $h(x)$ 的参考值，此时 $h(x)$ 的精确值 $h^*(x) = d[x.u]$ 。

(2) 状态 x 的初始值设置为： $x.u = s$ ， $x.fx = 0$ 。根据启发式函数，有

$$f(x) = g(x) + h(x) = x.fx + d[x.u] = d[x.u]$$

将状态 x 及 $f(x)$ 压入优先队列，将优先队列以 $f(x)$ 为键值进行排序，具有较小 $f(x)$ 值的状态排在队列的前端；

(3) 从优先队列中取出位于队首的状态 x ，根据图中从 $x.u$ 出发的边扩展状态 y ，得到 $f(y)$ ，将状态 y 及 $f(y)$ 压入优先队列；

(4) 如果队首状态 x 所在顶点 $x.u$ 第一次到达目标顶点 t ，则表明找到了一条从 s 到 t 的最短路径，它的长度就是 $f(x)$ 。容易知道，当状态 x 第二次到达目标顶点 t 且当前的路径长度 $x.fx$ 大于最短路径的长度时，表明找到了次最短路径，…，当第 K 次到达目标顶点 t 且 $x.fx$ 大于第 $K-1$ 短路径的长度时，就找到了从 s 到 t 的第 K 短路径。

```
//-----10.4.7.cpp-----//
const int MAXV = 1010, MAXE = 100010, INF = 0x3f3f3f3f;

int n, m;
int cnt, head[MAXV];
int dist[MAXV], visited[MAXV];

struct edge { int v, w, next; } g[MAXE];
struct state {
    int u, fx;
    bool operator < (const state &s) const { return fx > s.fx; }
};

void clearEdge() {
    cnt = 0;
    memset(head, -1, sizeof head);
}

void addEdge(int u, int v, int w) {
    g[cnt] = edge{v, w, head[u]};
    head[u] = cnt++;
}

int ksp(vector<tuple<int, int, int>> &data, int s, int t, int k) {
```

```

// 根据边数据建立原有向图的逆图。
clearEdge();
for (auto d : data) addEdge(get<1>(d), get<0>(d), get<2>(d));

// 使用 Moore-Dijkstra 算法确定目标顶点 t 到其他顶点的最短距离。
for (int i = 0; i < n; i++) dist[i] = INF, visited[i] = 0;
int u = t;
dist[u] = 0;
while (!visited[u]) {
    visited[u] = 1;
    for (int i = head[u]; ~i; i = g[i].next)
        if (!visited[g[i].v] && dist[g[i].v] > dist[u] + g[i].w)
            dist[g[i].v] = dist[u] + g[i].w;
    int least = INF;
    for (int i = 0; i < n; i++)
        if (!visited[i] && dist[i] < least)
            least = dist[i], u = i;
}

// 判断起始顶点是否可达。
if (dist[s] == INF) return -1;
// 根据有向边数据建立正向图。
clearEdge();
for (auto d : data) addEdge(get<0>(d), get<1>(d), get<2>(d));

// 压入初始状态。
priority_queue<state> q;
q.push(state{s, dist[s]});

// 根据启发式规则确定第 K 短路径。
int lastPathLength = -INF;
while (!q.empty()) {
    state ste = q.top(); q.pop();
    int walked = ste.fx - dist[ste.u];
    // 此处是获取从起始顶点 s 开始，到达终止顶点 t 的第 k 短路径，各条路径的长度不同。
    // 若需要获取从 s 出发到达 t 的所有路径中，按长度从小到大依次排列后的第 k 条路径，// 则忽略长度的比较即可，只需记录到达 t 的次数。
    if (ste.u == t && walked > lastPathLength) {
        lastPathLength = walked;
        if (!(--k)) return walked;
    }
    for (int i = head[ste.u]; ~i; i = g[i].next)
        q.push(state{g[i].v, fx + g[i].w + dist[g[i].v]});
}
return -1;
}
//-----10.4.7.cpp-----//

```

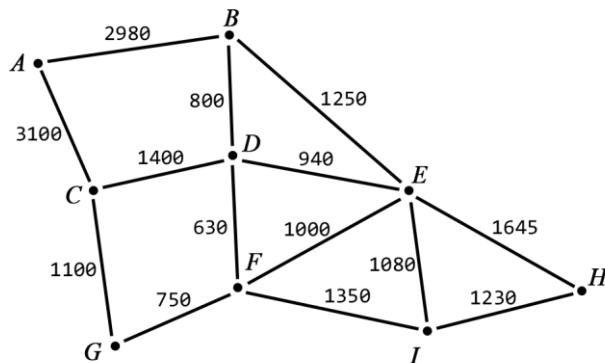
对于无向图来说，如果允许第 K 短路径出现圈，则可将一条无向边拆分为两条有向边，使用前述介绍的启发式搜索算法予以解决。如果第 K 短路径不允许出现圈，应该如何处理呢？此时需要在状态 x 中增加域，用于记录该状态经过的顶点，在扩展状态时，对于后续顶点，如果某个顶点是经过的顶点，则不能将该顶点作为后续顶点进行扩展。当路径上的顶点数目较小时，例如不大于 64 个，则可以使用一个 $long\ long$ int 型整数，以一个二进制位来表示某个顶点是否在路径上，通过位运算来判定某个顶点是否是经过的顶

点。不过此种方法存在局限，不能处理路径上具有较多顶点的情形^I。

强化练习：[10342 Always Late^C](#), [10740 Not the Best^D](#)。

10.5 网络流问题

网络流问题来源于确定铁路运输系统的运输量问题，最初由 Harris 予以形式化^[111]，可以将其近似地描述如下：城市 A 和 Z 之间拥有铁路运输网络，该铁路网络同时连接着 A 和 Z 之间的其他城市 B, C, D, …，两个城市间的货运列车具有特定的单位时间载运量，给定所有货运列车的单位时间载运量，要求货物不能在中间城市产生积压，确定该铁路网络在单位时间内从某个指定城市到其他城市的最大运输量。如图 10-11 所示的铁路运输网络，当给定各城市间铁路运输线的载运量后，如何确定城市 A 和城市 E 之间的最大运输量呢？



^I 如果需要更为高效地确定不带圈的第 K 短路径，可以考虑使用 Yen 在 1971 年提出的以其名字命名的 Yen 算法。Yen 算法的核心思想是以最短路径 P_1 或者已经求得的第 i 条最短偏离路径 P_i 为基础，在 P_i 上除了终止结点外的其他结点中确定偏离结点，构造候选路径，从候选路径集中找到最短的一条为最短偏离路径 P_{i+1} 。该算法在求 P_{i+1} 时，要将 P_i 上除了终止结点外的所有结点都视为偏离结点，并计算每个偏离结点到终止结点的最短路径，再与之前的 P_i 上起始结点到偏离结点的路径拼接，构成候选路径，进而求得最短偏离路径^{[1][2]}。令图中的顶点数为 n ，边数为 m ，需要确定路径数为 K ，算法的时间复杂度为 $O(Kn(m+n\log n))$ 。Yen 算法的优点是易于理解，可以准确地找到图中任意两结点间的 K 条最短路径，缺点是时间复杂度较高，时间代价较大，主要原因是在求 P_{i+1} 时，要将 P_i 上除了终止结点外的所有结点都视为偏离结点，从而在选择偏离结点发展候选路径时占用了大量的时间。为提高运行速度，后人在此算法的基础上不断改进和发展。比较有效的算法之一是 Martins 在 1999 年提出的 MPS 算法，其特点是简化了偏离路径的长度计算，在生成候选边时不像 Yen 算法那样计算每条候选路径的长度，而是要求更新每条弧上的减少长度，只选择长度减少最小的弧作为最短偏离路径，该算法在一般情况下可以提高运行速度，但是在最差的情况下与 Yen 算法的时间复杂度相同^[3]。

参考：

- [1] 付媛, 朱礼军, 韩红旗. K 最短路径算法与应用分析[J]. 情报工程, 2015, 1(1): 112-119.
- [2] Yen J Y. Finding the K Shortest Loopless Paths in a Network[J]. Management Science, 1971, 17(11): 712-716.
- [3] Martins E Q V, Pascoal M M B. A new implementation of Yen's ranking loopless paths algorithm[J]. Quarterly Journal of the Belgian, French and Italian Operations Research Societies, 2003, 1(2): 121-133.

图 10-11 具有载运量上限的铁路运输网络

上述问题可以转化为线性规划问题并使用单纯形方法予以解决, 不过大多数情况下可以将其转化为网絡流问题, 从而得到更为简便且有效的解决方案。网絡流问题是图论中一类常见的问题, 许多现实中的系统都包含流量, 如公路系统中的车辆流, 控制系统中的信息流, 供水系统中的水流, 金融系统中的现金流等。从问题求解的需求出发可以将网絡流问题分为: 网絡最大流, 流量有上下界的最大流和最小流, 最小费用最大流, 流量有上下界网絡的最小费用最大流等。网絡流算法也是求解其他一些图论问题的基础, 如求解图的顶点连通度和边连通度、匹配问题等。

由于最大流最小截定理是网絡流理论的基础, 而该定理包含容量网络、可行流、最大流、增广路、残留网络、残留容量、最小截等概念, 掌握这些概念是理解和应用网絡流算法的前提, 故本节按照概念、定理、算法、例题的顺序逐一予以介绍。

10.5.1 基本概念

容量网络和网絡流

设 $G=(V, E)$ 是一个有向图, 其中每条边 $e(u, v) \in E$ 均有一个权值 (一般均为非负值), 记为 $c(u, v) \geq 0$, 如果 $e(u, v) \notin E$, 则假定 $c(u, v)=0$, 称该边的权值为容量 (capacity)。在此有向图中有两个特别的顶点——源点 (source) 和汇点 (sink), 分别记为 s 和 t , 其余的每个顶点均处于从源点到汇点的某条路径上, 即对于每个顶点 $x \in V - \{s, t\}$, 存在一条路径 $s \rightarrow x \rightarrow t$, 称以上定义的加权有向网络 G 为容量网络 (capacity network)。如果任意边上的容量均为非负整数, 则称 G 为整数容量网络 (integer capacity network), 简称为整容量网络。

按照习惯, 在网絡流问题中, 一般将有向图中的边称为弧 (arc), 顶点 u 和 v 之间的有向边 $e(u, v)$ 记为弧 $a < u, v >$ 。通过容量网络 G 中每条弧 $a < u, v >$ 上的实际流量, 称为弧的流量 (flow rate), 记为 $f(u, v)$ 。所有弧上流量的集合 $f = \{f(u, v)\}$, 称为该容量网络 G 的一个网絡流 (network flow), 简称流 (flow)。

在容量网络 $G=(V, E)$ 中, 满足以下三个条件的网絡流, 称为可行流 (feasible flow):

(1) 容量限制 (capacity constraints): 对所有 $u, v \in V$, 满足 $f(u, v) \leq c(u, v)$, 即从一个顶点到另一个顶点的流量不能超过此条弧所能承载的容量。

(2) 反对称性 (skew symmetry): 对所有 $u, v \in V$, 满足 $f(u, v) = -f(v, u)$, 从顶点 u 到顶点 v 的流是其反向流求负所得, 即从顶点 u 到顶点 v 的流量和从顶点 v 到顶点 u 的流量绝对值相等, 方向相反。

(3) 流守恒性 (flow conservation): 对所有 $u \in V - \{s, t\}$, 满足

$$\sum_{v \in V} f(u, v) = 0$$

即非源点或非汇点的顶点 u 的总流量为 0, 可以理解为除了源点和汇点外, 进入顶点 u 和离开顶点 u 的流量是相等的, 不会产生流量积压在顶点 u 处的情形。

对于任何一个容量网络, 可行流总是存在—— $f = \{0\}$, 即每条弧上的流量均为 0, 该可行流称为零流 (zero flow)。如果一个网絡流只满足上述三个条件中的容量限制条件, 不满足其他两个条件, 则称此流为伪流 (pseudoflow)。在容量网络 $G=(V, E)$ 中, 具有最大流量的可行流, 称为网絡最大流 (network max-flow), 简称最大流 (maximum flow)。

链与增广路径

在容量网络 $G=(V, E)$ 中, 设有一可行流 $f = \{f(u, v)\}$, 根据每条弧上流量的多少以及流量和容量的关

系, 可将弧分为以下四种类型: (1) 饱和弧, 满足关系 $f(u, v) = c(u, v)$; (2) 非饱和弧, 满足 $f(u, v) < c(u, v)$; (3) 零流弧, 满足 $f(u, v) = 0$; (4) 非零流弧, 满足 $f(u, v) > 0$ 。

在容量网络 G 中, 称顶点序列 $(u, u_1, u_2, \dots, u_n, v)$ 为一条链 (chain), 顶点序列要求相邻两个顶点之间有一条弧, 如 $\langle u, u_1 \rangle$ 或 $\langle u_1, u_2 \rangle$ 为容量网络中的一条弧。

令 P 是 G 中从源点 s 到汇点 t 的一条链, 约定从源点 s 指向汇点 t 的方向为该链的正方向。根据弧的方向和链的方向, 可将弧分为两类: (1) 前向弧, 方向与链的方向相同的弧, 记为 P^+ ; (2) 后向弧, 方向与链的方向相反的弧, 记为 P^- 。注意前向弧和后向弧是相对的, 是相对于指定链的正方向而言。

令 f 是给定容量网络 G 中的一个可行流, P 是从源点 s 到汇点 t 的一条链, 若 P 满足下列条件:

(1) 在 P 的所有前向弧 $\langle u, v \rangle$ 上, $0 \leq f(u, v) < c(u, v)$, 即 P^+ 中的每一条弧都是非饱和弧。

(2) 在 P 的所有后向弧 $\langle u, v \rangle$ 上, $0 < f(u, v) \leq c(u, v)$, 即 P^- 中的每一条弧都是非零流弧。

那么称 P 为关于可行流 f 的一条增广路径 (augmenting path), 简称增广路 (或称增广链、可改进路)。沿着增广路改进可行流, 使可行流具有更大的流量的过程称为增广 (augmenting)。

增广路定理 (augmenting path theorem): 令容量网络 $G = (V, E)$ 的一个可行流为 f , f 为最大流的充要条件是在容量网络中不存在增广路。

10.5.2 Ford-Fulkerson 方法

网络最大流的求解算法主要有两大类: 增广路算法 (augmenting path algorithm) 和预流推进算法 (preflow-push algorithm)。

增广路算法: 由增广路定理, 要得到最大流, 需要找到一个初始的可行流, 在此可行流基础上不断进行增广, 直到无法再增广为止, 此时的可行流即为最大流。算法的关键在于寻找增广路, 如果能够快速构造接近最大流的可行流, 后续的增广可以更快地完成, 从而能够提高算法的效率。增广路算法中最为常见的是基于 Ford-Fulkerson 方法的衍生或改进算法。

预流推进算法: 预流推进算法是从一个预流出发对活跃顶点沿着允许弧进行流量增广, 每次增广称为一次推进 (push)。在推进过程中, 流一定满足容量限制条件, 但一般不满足流量守恒条件, 因此是一个伪流。如果一个伪流中, 除了源点和汇点, 其他每个顶点流出的流量之和总是小于等于流入该顶点的流量之和, 称这样的伪流为预流 (preflow), 故这类算法称为预流推进算法。预流推进算法包括一般预流推进算法和最高标号预流推进算法。在竞赛中, 一般不会出现只能使用预流推进算法才能解决的题目, 而且受篇幅所限, 在此对预流推进算法不做进一步介绍, 有兴趣的读者请参考相关资料^[112]。

Ford-Fulkerson 方法的理论基础是最大流最小截定理和增广路定理^[113], 之所以称为“方法”而不是“算法”, 缘于其提出的是一个求最大流的方法框架, 并未明确指出使用何种方式来寻找增广路。以下是 Ford-Fulkerson 方法的伪代码表示:

```
// G 为容量网络, s 为源点, t 为汇点
fordFulkerson(G, s, t) {
    初始化可行流 f 为零流;
    while 存在增广路 p
        沿增广路 p 增广可行流 f;
        返回最大流 f;
}
```

Ford-Fulkerson 方法是一种迭代方法。它从零流开始, 通过迭代, 每次寻找一条“增广路”来增加流

值。这里的“增广路”可以视为从源点 s 到汇点 t 的一条路径，沿着增广路可以压入更多的流，以增加当前可行流的流值，反复进行这一过程，直到所有增广路均被找到，此时根据增广路定理，当前的可行流已经是最大流。

Ford-Fulkerson 方法的最初实现被称为标号法 (labeling method)^[114]，其基本思想是从容量网络的任意一个可行流开始，构造出一个流量不断增加的流序列，并且终止于最大流。

标号法

(1) 从给定容量网络 G 的一个可行流 f 开始 (初始时一般取零流，后续则取经过标号过程后得到的改进流)，为源点 s 标记 $(0, +\infty)$ ，并令 $L = \{s\}$ 。

(2) 若 L 为空集，则停止， f 是最大流。若 L 不为空集，取 L 最前面的元素 u ，从 L 中删去元素 u ，对与 u 邻接且未标号的顶点 v 进行标号操作，并将顶点 v 加入 L 的后端。标号采用如下规则：如果 $a = \langle u, v \rangle \in E(G)$ 且 $f(a) < c(a)$ ，则予 v 以标号 $(+u, \min(\alpha_u, c(a) - f(a)))$ ，若 $a = \langle v, u \rangle \in E(G)$ 且 $f(a) > 0$ ，则予 v 以标号 $(-u, \min(\alpha_u, f(a)))$ ， α_u 表示经过顶点 u 可改进的流量值。

(3) 若汇点 t 被标号，转入第 (4) 步，否则转入第 (2) 步。

(4) 已被标号的顶点构成 G 中一条 f 增广路 P ，

$$s = u_0 a_1 u_1 a_2 u_2 \cdots u_{n-1} a_n (u_n =) t$$

其中对每个 $i=1, 2, \dots, n$ ，当 $a_i = \langle u_{i-1}, u_i \rangle$ 时， u_i 的标号为 (u_{i-1}, α_{u_i}) ，使用 $f(a_i) + \alpha_i$ 替代 $f(a_i)$ ；当 $a_i = \langle u_i, u_{i-1} \rangle$ 时， u_i 的标号为 $(-u_{i-1}, \alpha_{u_i})$ ，使用 $f(a_i) - \alpha_i$ 替代 $f(a_i)$ 。清除标号，转入第 (1) 步。

标号法为容量网络中的每个顶点赋予一个标号 (label)，标号有两个分量 (p, α) ，第一个分量 p 表示当前顶点的标号是从哪一个顶点获得，第二个分量 α 表示可改进的流量值。标号法的总体思想是从可行流开始 (一般选择零流)，逐步求出容量网络 G 的顶点标号序列。首先对源点 s 进行标号操作，其标号为 $(0, +\infty)$ ，第一个分量为 0，表示该顶点为源点 s ，第二个分量为 $+\infty$ ，表示源点 s 流出的流量可以是任意正数值，只要与源点 s 邻接的其他顶点能够接受从源点流出的流量。在源点 s 具有标号后，从源点 s 出发使用 BFS 进行遍历，并对遍历过程中遇到的每个顶点进行标号。

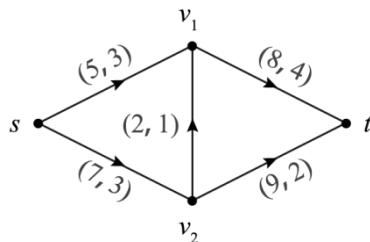


图 10-12 标号法。源点 s 有两个邻接顶点 v_1 和 v_2 ，先对 v_1 进行标号，由于弧 $\langle s, v_1 \rangle$ 的容量为 5，流量为 3，则顶点 v_1 最多只能接受来自源点 s 大小为 2 的流量，故 v_1 的标号为 $(0, 2)$ ，同理 v_2 的标号为 $(0, 4)$ 。接着对 v_1 的邻接顶点进行标号，此时 v_2 已经具有标号，不能再通过 v_1 对其进行标号。对 v_1 的邻接顶点 t 进行标号，弧 $\langle v_1, t \rangle$ 的容量为 8，流量为 4，则 t 的标号为 $(1, 2)$ ，标号过程到达汇点，可以终止此轮标号过程。

在标号法中，Ford 和 Fulkerson 创造性地提出了反向弧的概念。反向弧对于正确求解最大流是一个不可缺少的环节，反向弧的存在实质上是为增广路的构建提供了“反悔”的机会，使得后续增广路的选取能够调整，让非最优的可行流转化为更优的可行流。

由于在标号过程中需要检查与当前顶点正向和反向连接的其他顶点，在以下标号法的参考实现中，使用邻接矩阵来存储有向弧。

```

//-----10.5.2.cpp-----//
const int MAXV = 1000, INF = 0x3f3f3f3f;
const int UNLABELED = -1, UNCHECKED = 0, CHECKED = 1;

struct arc { int capacity, flow; } arcs[MAXV][MAXV];
struct flag { int status, parent, alpha; } flags[MAXV];
int source, sink;

int fordFulkerson() {
    // 反复执行标号过程直到不存在改进路。
    while (true) {
        // 首先将源点标记为已标号但尚未检查的顶点。
        memset(flags, -1, sizeof(flags));
        flags[source] = flag{UNCHECKED, -1, INF};
        queue<int> unchecked; unchecked.push(source);
        // 当汇点尚未被标记且队列非空时继续。
        while (flags[sink].status == UNLABELED && !unchecked.empty()) {
            // 检查与顶点 u 正向或反向连接的其他顶点 v。如果顶点 v 尚未被标号则予以标号。
            int u = unchecked.front(); unchecked.pop();
            for (int v = source; v <= sink; v++) {
                if (flags[v].status == UNLABELED) {
                    // 当前弧为正向弧且流量未达到容量。
                    if (arcs[u][v].capacity < INF &&
                        arcs[u][v].flow < arcs[u][v].capacity) {
                        flags[v].status = UNCHECKED, flags[v].parent = u;
                        flags[v].alpha = min(flags[u].alpha,
                                             arcs[u][v].capacity - arcs[u][v].flow);
                        unchecked.push(v);
                    }
                    // 当前弧为反向弧且有流量。
                    else if (arcs[v][u].capacity < INF && arcs[v][u].flow > 0) {
                        flags[v].status = UNCHECKED, flags[v].parent = -u;
                        flags[v].alpha = min(flags[u].alpha, arcs[v][u].flow);
                        unchecked.push(v);
                    }
                }
            }
            // 顶点 u 已经标号且已经检查完毕。
            flags[u].status = CHECKED;
        }
        // 当标号过程未能到达汇点或者汇点的调整量为 0，表明已经不存在改进路。
        if (flags[sink].status == UNLABELED || flags[sink].alpha == 0) break;
        // 汇点有标号，根据汇点的改进量沿着改进路对容量网络进行调整。
        int v = sink, u = abs(flags[v].parent), delta = flags[v].alpha;
        while (true) {
            if (arcs[u][v].flow < INF) arcs[u][v].flow += delta;
            else arcs[v][u].flow -= delta;
            // 调整到源点，退出。
            if (u == source) break;
            v = u, u = abs(flags[u].parent);
        }
    }
}

```

```

// 统计从源点流出的总流量。
int maxFlow = 0;
for (int u = source; u <= sink; u++)
    if (arcs[source][u].flow < INF)
        maxFlow += arcs[source][u].flow;
return maxFlow;
}
//-----10.5.2.cpp-----//

```

可以证明，如果在标号过程中采用 BFS，则每次修正流 f 都是通过最短 f 增广路获得，则标号过程至多进行 $|V||E|/2$ 次即可获得最大流，因此标号法总的时间复杂度为 $O(VE^2)$ 。为了简便，上述实现使用邻接矩阵来表示容量网络的有向弧，因此时间复杂度会达到 $O(V^2E^2)$ ，对于顶点数量较多的图无法高效处理，容易导致超时。

强化练习：[820 Internet Bandwidth^A](#)。

10.5.3 Edmonds-Karp 算法

Edmonds-Karp 算法是对标号法的改进，应用了残留容量和残留网络的概念^[115]。

给定容量网络 $G=(V, E)$ 及可行流 f ，弧 $\langle u, v \rangle$ 上的残留容量（residual capacity）记为 $c_f(u, v) = c(u, v) - f(u, v)$ 。每条弧的残留容量表示该弧上可以增加的流量。从顶点 u 到顶点 v 的流量减少，等效于从顶点 v 到顶点 u 的流量增加。除了正向的残留容量 $c_f(u, v)$ ，还有反向的残留容量 $c_f(v, u) = -f(u, v)$ 。

设有容量网络 $G=(V, E)$ 及其上的网络流 f ， G 关于 f 的残留网络（residual network）记为 $G_f=(V_f, E_f)$ ，其中 G_f 的顶点集 V_f 和 G 的顶点集 V 相同，即 $V_f=V$ ，对于 G 中的任何一条弧 $\langle u, v \rangle$ ，如果 $f(u, v) < c(u, v)$ ，那么在 G_f 中有一条弧 $\langle u, v \rangle \in E_f$ ，其容量为 $c_f(u, v) = c(u, v) - f(u, v)$ ，如果 $f(u, v) > 0$ ，则在 G_f 中有一条弧 $\langle v, u \rangle \in E_f$ ，其容量为 $c_f(v, u) = f(u, v)$ 。

残留网络 G_f 和原容量网络 G 有以下关系：令 f_1 是容量网络 $G=(V, E)$ 的可行流， f_2 是残留网络 G_f 的可行流，则 $f_1 + f_2$ 仍是容量网络 G 的一个可行流，且可行流的值 $|f_1 + f_2| = |f_1| + |f_2|$ 。

Edmonds-Karp 算法具体步骤为：选定一个可行流作为初始流（一般选择零流），将容量网络表示成相应的残留网络，接着从源点 s 开始出发，沿着残留容量不为零的弧对残留网络进行广度优先遍历，如果遍历过程能够到达汇点 t ，说明源点 s 和汇点 t 之间存在一条路径，该路径上弧的残留容量均大于零，也就意味着找到了一条增广路，按照此路径上的最小残留容量对原可行流进行增广，更新残留网络，再次进行广度优先遍历寻找增广路，直到某次遍历无法到达汇点 t ，说明此时的残留网络不存在从源点 s 到汇点 t 的增广路，根据增广路定理，当前的可行流已经为最大流。由于在寻找增广路时使用的是广度优先搜索，得到的增广路在残留网络中具有最少的路径边数，因此属于最短增广路径（Shortest Augmenting Path, SAP）算法。可以使用如下代码来表示 Edmonds-Karp 算法的整个过程。

```

// 使用广度优先遍历寻找增广路，确定增广路的流量，更新残留网络，直到无法再找到增广路。
int edmondsKarp(arc *arcs, int source, int sink) {
    int flow = 0;
    // 通过广度优先遍历确定是否存在从源点到汇点的路径。
    while (bfs(arcs, source, sink)) {
        // 确定增广路的容量，该容量由增广路上最小的残留容量确定。
        int volume = pathVolume(arcs, source, sink);
        // 更新当前可行流的流量。
        flow += volume;
        // 沿着增广路更新残留网络。
    }
}

```

```

        augmentPath(arcs, source, sink, volume);
    }
    // 当前已不存在从源点到汇点的路径, 返回最大流的流量。
    return flow;
}

```

为了更为方便地实现 Edmonds-Karp 算法, 可将表示有向弧的结构体进行适当修改, 增加相应的域来表示有向弧的容量和残留容量。由于在寻找增广路的过程中, 关注的是增广路径和其上的瓶颈残留容量, 对有向弧的取用顺序无特殊要求, 故可使用链式前向星数据结构, 将全部有向弧存储在数组中, 把从某个顶点出发的有向弧构造成一个链表。

```

struct arc {
    int u, v;           // 有向弧的起始和终止顶点
    int capacity;       // 弧的容量
    int residual;       // 弧的残留容量
    int next;           // 从顶点 u 出发的下一条有向弧在整个有向弧数组中的序号
};

```

在建立顶点有向弧链表的过程中, 与常规的建立链表的顺序稍有不同, 先发现的有向弧, 位于链表的后端, 后发现的有向弧, 位于链表的前端, 使用数组记录每个顶点最后发现的有向弧的序号。从有向弧在有向弧数组中的序号来看, 链表中位于前端的有向弧其序号较大, 之所以使用这种方式是因为在使用链表的过程中并不需要对链表进行双向遍历。在更新残留网络时, 由于不仅要更新前向弧, 还要更新后向弧, 为了便于访问前向弧和其对应的后向弧 (或者相反), 此处使用了一个技巧: 将正向弧和反向弧相邻存放, 使得正向弧的序号始终为偶数, 对应的反向弧序号为相邻的奇数。这样在更新容量网络的过程中, 若需获取正向弧所对应的反向弧序号 (或者相反), 使用有向弧的序号与常数 1 进行“与或” (^) 操作即可。

```

// 将有向弧添加到数组中, 建立链表。
void addArc(int u, int v, int capacity)
{
    // 建立正向弧。
    arcs[idx] = (arc){u, v, capacity, capacity, head[u]};
    head[u] = idx++;
    // 建立反向弧。
    arcs[idx] = (arc){v, u, capacity, 0, head[v]};
    head[v] = idx++;
}

```

需要注意的是, 如果给定的图是有向图, 在将其转化为容量网络时, 建立顶点 u 和 v 之间的有向弧 $<u, v>$ 时, 只需调用一次 $\text{addArc}(u, v, \text{capacity})$ 。若给定的是无向图, 则需要将顶点 u 和 v 之间的无向边转换为有向弧 $<u, v>$ 和 $<v, u>$, 不仅需要调用 $\text{addArc}(u, v, \text{capacity})$, 还需调用 $\text{addArc}(v, u, \text{capacity})$, 这样才能够建立完整的容量网络。

以下是 Edmonds-Karp 算法的参考实现。初始时先构建容量网络所对应的残留网络, 选择零流作为初始流, 从源点出发, 沿着残留容量为正的有向弧对残留网络进行广度优先遍历, 如果发现从源点到汇点的路径, 则存在增广路, 确定增广路的流量, 更新容量网络和当前可行流的流量, 直到不存在增广路, 返回最大流的流量。

```

//-----10.5.3.cpp-----//
const int INF = 0x7f7f7f7f;

```

```

struct arc { int u, v, capacity, residual, next; };

class EdmondsKarp {
private:
    arc *arcs;
    int vertices, idx, source, sink, *head, *parent, *visited;

    // 使用广度优先遍历寻找从源点到汇点的增广路。
    bool bfs() {
        memset(parent, -1, vertices * sizeof(int));
        memset(visited, 0, vertices * sizeof(int));
        visited[source] = 1;
        queue<int> q; q.push(source);
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (u == sink) break;
            // 遍历以当前顶点为起点的有向弧链表，沿着残留容量为正的弧进行遍历。
            for (int i = head[u]; ~i; i = arcs[i].next) {
                if (!visited[arcs[i].v] && arcs[i].residual > 0) {
                    q.push(arcs[i].v);
                    visited[arcs[i].v] = 1;
                    // 注意路径记录的是有向弧的序号而不是顶点的序号。
                    parent[arcs[i].v] = i;
                }
            }
        }
        // 如果遍历未能到达汇点，表明不存在增广路，当前可行流已经为最大流。
        return visited[sink];
    }

    // 将流量网络还原到初始状态。
    void restoreFlowNetwork() {
        for (int i = 0; i < idx; i++) {
            if (i & 1) arcs[i].residual = 0;
            else arcs[i].residual = arcs[i].capacity;
        }
    }
}

public:
    // v 表示顶点的数量，e 表示边的数量，s 表示源点的序号，t 表示汇点的序号。
    EdmondsKarp(int v, int e, int s, int t) {
        vertices = v;
        head = new int[v], parent = new int[v], visited = new int[v];
        arcs = new arc[e];
        idx = 0, source = s, sink = t;
        memset(head, 0xff, vertices * sizeof(int));
    }

    ~EdmondsKarp() { delete [] head, parent, visited, arcs; }

    int maxFlow() {
        // 若需多次运行最大流算法，则在每次运行后，流量网络中的流量可能已经发生改变，
        // 故需将流量网络还原到初始状态。
        restoreFlowNetwork();
        // 使用 BFS 搜索增广路。
        int flow = 0;
        while (bfs()) {

```

```

// 确定增广路的流量并更新可行流及残留网络。
int delta = INF;
for (int i = parent[sink]; ~i; i = parent[arcs[i].u])
    delta = min(delta, arcs[i].residual);
flow += delta;
for (int i = parent[sink]; ~i; i = parent[arcs[i].u]) {
    arcs[i].residual -= delta;
    arcs[i ^ 1].residual += delta;
}
}
return flow;
}
//-----10.5.3.cpp-----/

```

Edmonds-Karp 算法每次使用 BFS 寻找增广路的时间复杂度为 $O(E)$ ，可以证明，对于整数容量网络，至多需要 $O(VE)$ 次迭代即可获得最大流，故 Edmonds-Karp 的时间复杂度为 $O(VE^2)$ 。在解题应用中，主要难点在于如何将问题建模成网络流问题，很多情况下，出题者会将题目“改头换面”，使得解题者不太容易一眼看出问题的底层模型为网络流。

563 Crimewave^c (犯罪浪潮)

Nieuw Knollendam 镇的交通图由经路和纬路构成的方格网组成。作为重要的经贸中心，Nieuw Knollendam 镇建有多家银行——几乎每个交叉路口都有一家银行（但两家银行不会位于同一个交叉路口）。不幸的是，这也导致了很多劫案的发生。在这里，银行抢劫司空见惯，而且经常是一天发生好几起。这不仅对银行来说是一个大问题，对劫犯来说也是一样。在抢劫银行后，劫犯总会尝试驾车尽快逃离，在大多数时候，警察都会在后面展开高速汽车追捕。有些时候，两个逃跑的劫犯会经过同一个交叉路口，这会带来危险：撞车或者在同一地点警察太多而增加被捕的风险。

为了防止这种不愉快的事情发生，劫犯们一致同意提前计划逃跑路线。每个星期六的晚上，他们碰头并商议下周的计划，内容包括哪一天谁抢劫哪个银行，每天的逃跑路线方案等等。两条逃跑路线不会使用同一个交叉路口，有些时候因为上述条件的限制，他们未能成功制定计划，但是他们相信这样的计划应该存在。

给定一个大小为 $s \times d$ 的网格以及被抢劫的银行所处的交叉口位置，确定是否存在可能的逃离路线方案，使得任意两条逃离路线不会在同一个交叉路口相会。

输入

输入的第一行包含一个数字 p ，表示需要解决的问题数量。接下来每个问题的第一行包含三个数字，第一个数字 s ($1 \leq s \leq 50$) 表示经路的数量，第二数字 a ($1 \leq a \leq 50$) 表示纬路的数量，第三个数字 b ($b \geq 1$) 表示被抢劫的银行数。接下来的 b 行，每行给出了一个银行的位置信息，它由两个数字 x （表示银行所处经路的编号）和 y （表示银行所处纬路的编号）， $1 \leq x \leq s$, $1 \leq y \leq a$ 。

输出

输出包含 p 行，如果存在一个方案，使得任意两条逃离路线不会发生在同一个交叉路口相会的情况，则输出“possible”，否则输出“not possible”。

样例输入

```

1
6 6 10
4 1

```

样例输出

```
possible
```

```

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

```

分析

本问题实质上是边不相交独立路径 (independent and edge-disjoint paths) 问题, 可以将交叉路口视为有向图中的顶点, 将问题建模成网络最大流予以解决。由于在逃脱过程中要求两条路线不能在同一个交叉路口相会, 如果按照常规的方式在两个顶点间建立有向弧, 将难以确保此要求的实现, 此处可以使用“拆点技巧” (vertex splitting technique), 即将一个顶点拆分为两个顶点, 分别称之为“前点”和“后点”, 在“前点”和“后点”间建立容量为 1 的前向弧和反向弧, 这样在求最大流时, 由于容量限制, “前点”和“后点”间的流量最大可能为 1, 该流量要么沿着前向弧通过, 要么沿着反向弧通过, 而不可能同时在前向弧和反向弧上均有非零流量, 由此得到的各条增广路均代表着一条逃离路线, 这些增广路不会产生交会。建立源点 s , 向每家银行所处交叉路口的“前点”建立正向弧, 容量和残留容量均为 1, 反向弧容量为 1, 残留容量为 0; 建立汇点 t , 处于城镇边界上的交叉路口的“后点”向汇点 t 建立正向弧, 容量和残留容量均为 1, 反向弧容量为 1, 残留容量为 0; 其他每个顶点的“后点”与上下左右顶点的“前点”建立正向弧和反向弧, 那么可以将问题归结为如上构造的容量网络的最大流是否能够满流的问题。只要求出该容量网络的最大流, 检查是否和被抢劫的银行数量相等即可确定是否存在满足条件的逃跑路线方案。

在进行拆点操作时, “前点”的编号为 $(x-1) \times \text{avenues} + y$, 其中 x 为交叉路口的经路编号, y 为纬路编号, 后点的编号在“前点”编号的基础上增加 $\text{streets} \times \text{avenues}$, 这样可以不重复地标记拆分得到的顶点。源点的编号为 0, 汇点的编号可以任意取, 只要不与已有的顶点编号重复即可。例如, 汇点可取编号为 $2 \times \text{streets} \times \text{avenues} + 1$ 。

参考代码

```

const int MAXV = 5100, MAXA = 31000, INF = 0x7f7f7f7f;

struct arc {
    int u, v, capacity, residual, next;
};

class EdmondsKarp {
    // 代码省略, 请参考前述给出的 Edmonds-Karp 算法代码实现。
};

int problem, streets, avenues, banks;

void createGraph(EdmondsKarp &ek) {
    // 在源点和银行之间建立有向弧。
    for (int b = 1, x, y; b <= banks; b++) {
        cin >> x >> y;
        ek.addArc(0, (x - 1) * avenues + y, 1);
    }
}

```

```

// 在交叉路口之间和交叉路口与汇点间建立有向弧。
int offset[4][2] = {{1, 0}, {0, -1}, {-1, 0}, {0, 1}};
int base = streets * avenues;
for (int s = 1; s <= streets; s++) {
    for (int a = 1; a <= avenues; a++) {
        int index = (s - 1) * avenues + a;
        // 将交叉路口拆分为前点和后点并建立有向弧。
        ek.addArc(index, base + index, 1);
        // 如果交叉路口不位于城镇的边界上，则每个交叉路口的后点向上下左右四个
        // 交叉路口的前点建立有向弧，否则在交叉路口的后点和汇点间建立有向弧。
        if (s > 1 && s < streets && a > 1 && a < avenues) {
            for (int f = 0; f < 4; f++) {
                int ss = s + offset[f][0], aa = a + offset[f][1];
                if (ss >= 1 && ss <= streets && aa >= 1 && aa <= avenues)
                    ek.addArc(base + index, (ss - 1) * avenues + aa, 1);
            }
        }
        else
            ek.addArc(base + index, 2 * streets * avenues + 1, 1);
    }
}

int main(int argc, char *argv[]) {
    cin >> problem;
    for (int p = 1; p <= problem; p++) {
        cin >> streets >> avenues >> banks;
        EdmondsKarp ek(MAXV, MAXA, 0, 2 * streets * avenues + 1);
        createGraph(ek);
        cout << (ek.maxFlow() == banks ? "possible" : "not possible") << '\n';
    }
    return 0;
}

```

强化练习：[10092](#) The Problem With the Problem Setter^B，[10249*](#) The Grand Dinner^B，[10511](#) Councilling^D，[10779](#) Collector's Problem^C，[11358](#) Faster Processing Feasibility^E。

扩展练习：[1242](#) Necklace^D，[10735](#) Euler Circuit^D，[10983*](#) Buy One Get the Rest Free^D。

10.5.4 Dinic 算法

在 Edmonds-Karp 算法中，通过一次广度（或深度）优先遍历，只能确定一条增广路，之后沿着该增广路进行增广。Dinic 算法（或称 Dinitz 算法^I）^{[116][117]}在残留网络上先使用广度优先遍历生成层次网络（layered network），之后使用一次深度优先遍历寻找阻塞流（blocking flow）以完成多次增广，由于 Dinic

^I Dinic 算法提出者为 Yefim Dinitz（原为前苏联计算机科学家，现居以色列），其发表的论文为了满足杂志稿件篇幅的要求（杂志为前苏联关于算法的权威俄文期刊，要求文章不超过 4 页），对文章进行了“压缩”，使得算法的描述较为晦涩难懂。Shimon Even 和 Alon Itai（两位西方计算机科学家）经过努力，在理解 Dinitz 论文的基础上，结合 Alexander Karzanov 的阻塞流思想（Karzanov 亦为前苏联计算机科学家，其论文和 Dinitz 的论文发表在同一杂志上，也仅为 4 页），使用 BFS 和 DFS “优美而精巧”地实现了 Dinitz 算法，并开始做关于 Dinitz 算法的报告，但在讲授时使用了“Dinic's algorithm”这个错误拼写的名称，不过后续由于使用非常广泛，Dinic 算法这个名称便“约定俗成”地沿用至今。

算法能够在一次 BFS 基础生成的层次网络上使用 DFS 进行多次增广，在实践中效率一般比 Edmonds-Karp 算法要高。

Dinic 算法可以概括为以下步骤：

- (1) 初始化容量网络和网络流；
- (2) 在残留网络上使用 BFS 构建层次网络，若汇点不在层次网络中，算法结束；
- (3) 在层次网络中寻找阻塞流，使用 DFS 进行多次增广；
- (4) 若未能找到阻塞流，算法结束，否则转到步骤 (2)。

残留网络的概念已经在 Edmonds-Karp 算法中介绍，此处介绍层次网络和阻塞流的概念。层次网络实际上是按照流网络中各个顶点与源点的距离，将相同距离的顶点进行分层的预处理过程。进行此操作后，在每次使用 DFS 进行增广时，不再需要遍历每一个顶点以判断是否可能是增广路上的下一个顶点，而是根据层次网络，对当前顶点的邻接顶点进行筛选。在具体实现时，并不需要显式构建层次网络，只需获得各个顶点与源点的距离信息即可。

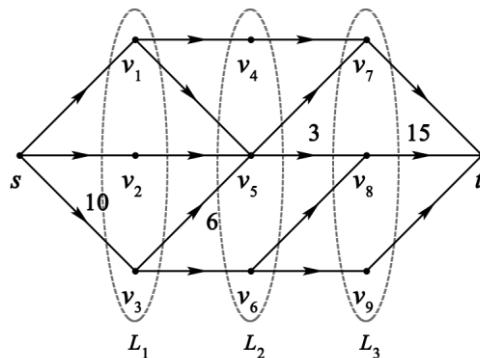


图 10-13 层次网络。在从源点 s 到汇点 t 的有向路径中，顶点 v_1, v_2, v_3 与源点具有相同的距离，故其位于层次网络的同一层 L_1 。同理 v_4, v_5, v_6 位于层次网络中的 L_2 ， v_7, v_8, v_9 位于层次网络中的 L_3 。在寻找增广路径时，当前顶点为 v_1 时，只需考虑层次比 v_1 大 1 的邻接顶点 v_4, v_5 。

阻塞流的概念也很容易理解。在增广过程中，位于增广路径上的顶点间的可用流量大小可能并不相等，增广路径所能利用的最大流量取决于该路的“瓶颈”流量，即可用流量最小的一条有向弧，这条有向弧所表示的流就成为阻塞流。注意，在考虑阻塞流时，只考虑正向弧，不考虑反向弧。

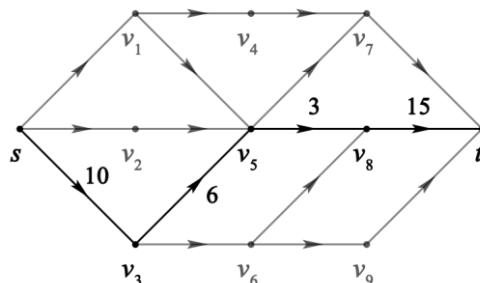


图 10-14 阻塞流。在从源点 s 到汇点 t 的增广路径中，只考虑正向有向弧，则增广路 $s - v_3 - v_5 - v_8 - t$ 的最大流量由有向弧 $\langle v_5, v_8 \rangle$ 的残余容量所“阻塞”，故名“阻塞流”。

以下给出 Dinic 算法的参考实现。实现以类的形式给出，在使用时，需要将图的顶点数量、有向弧数量、源点的序号、汇点的序号作为参数进行类的初始化。如果需要反复使用该类，可增加一个初始化方法，以便在每次构建流量网络时将表示图的链式前向星数据结构重置。

```

//-----10.5.4.cpp-----
const int INF = 0x7fffffff;

// 有向弧。
struct arc {
    int u, v, capacity, residual, next;
};

class Dinic {
private:
    // 有向弧数组。
    arc *arcs;
    // vertices 表示顶点数量，source 为源点序号，sink 为汇点序号，idx 为弧序号计数器。
    // head 记录顶点关联弧在弧数组中的起始序号，dist 记录层次网络中顶点与源点的距离。
    int vertices, source, sink, idx, *head, *dist;

    // 使用 BFS 对流网络进行分层。
    bool bfs() {
        memset(dist, -1, sizeof(int) * vertices);
        queue<int> q; q.push(source);
        dist[source] = 1;
        while (!q.empty()) {
            int u = q.front(); q.pop();
            // 到达汇点，直接退出分层过程。
            if (u == sink) break;
            for (int i = head[u]; ~i; i = arcs[i].next)
                // 沿着残余流量为正且未被标记的弧前进。
                if (arcs[i].residual > 0 && dist[arcs[i].v] < 0) {
                    dist[arcs[i].v] = dist[u] + 1;
                    q.push(arcs[i].v);
                }
        }
        return dist[sink] > 0;
    }

    // 使用 DFS 寻找阻塞流。
    int dfs(int u, int flow) {
        // 到达汇点，返回流的大小。
        if (u == sink) return flow;
        for (int i = head[u]; ~i; i = arcs[i].next) {
            int v = arcs[i].v, r = arcs[i].residual;
            // 在分层网络中沿着残余流量不为零的弧前进。
            if (dist[v] == (dist[u] + 1) && r > 0) {
                // 递归寻找阻塞流。
                int volume = dfs(v, min(r, flow));
                // 更新位于阻塞流上的正向弧和反向弧的容量。
                if (volume > 0) {
                    arcs[i].residual -= volume;
                    arcs[i ^ 1].residual += volume;
                }
            }
        }
        return volume;
    }
}

```

```

        }
    }
    return 0;
}

public:
    // v 为顶点的数量, e 为有向弧的数量, s 为源点的序号, t 为汇点的序号。
    // 为了避免运行时错误, 建议在设置 v 和 e 时都比实际需要稍大一些。
    Dinic(int v, int e, int s, int t) {
        arcs = new arc[e];
        vertices = v;
        source = s, sink = t;
        idx = 0, head = new int[v], dist = new int[v];
        memset(head, 0xff, sizeof(int) * v);
    }

    ~Dinic() { delete [] arcs, head, dist; }

    // 向流网络新增正向弧和反向弧。
    void addArc(int u, int v, int capacity) {
        arcs[idx] = (arc){u, v, capacity, capacity, head[u]};
        head[u] = idx++;
        arcs[idx] = (arc){v, u, capacity, 0, head[v]};
        head[v] = idx++;
    }

    // Dinic 算法求最大流。
    int maxFlow() {
        int flow = 0;
        while (bfs()) {
            while (int delta = dfs(source, INF))
                flow += delta;
        }
        return flow;
    }
};

//-----10.5.4.cpp-----//

```

构建层次网络, 可以使用广度优先遍历予以完成, 其时间复杂度为 $O(E)$, 使用深度优先遍历寻找阻塞流的时间复杂度为 $O(V^2)$, 因而 Dinic 算法的时间复杂度为 $O(V^2E)$ 。与 Edmonds-Karp 算法相比较, 在实际应用中, 由于流网络中的有向弧一般都比顶点的数量要多, 因而 Dinic 算法的运行效率更高。如果采用动态树 (dynamic tree) 数据结构来实现 Dinic 算法, 则寻找阻塞流的时间可以降低为 $O(E \log V)$, 总的时间复杂度可降低为 $O(VE \log V)$ 。

当前弧优化

为了进一步提高 Dinic 算法的效率, 可以使用当前弧优化 (current arc optimization)¹。在使用 DFS 进行增广的过程中, 某条弧被遍历后, 在此轮增广过程中肯定不会再被使用, 因此在遍历顶点的关联弧时不

¹ 当前弧优化是 Ahuja 和 Orlin 发明的 ISAP 算法中提出的一种优化技巧, 经过研究发现, Dinic 算法实际上可以看做是 ISAP 算法的一种特例, 因此该优化技巧可以“无缝衔接式”地应用到 Dinic 算法中。

必每次从头开始，而是可以使用一个数组，记录每个顶点当前所使用弧的序号，在后续增广时从此弧开始寻找阻塞流，这样可以在一定程度上提高效率。需要注意以下两点：一是当前弧只作用于当前轮次的 DFS 增广，每构建一次层次网络，均需要重新初始化当前弧；二是记录的当前弧是上次使用完毕时的弧序号，下次遍历时仍需要从此弧序号开始，而不是从此弧序号的下一条弧开始。

```

class Dinic {
private:
    // 在变量声明中加入记录当前弧的数组。
    int vertices, source, sink, idx, *head, *dist, *current;

public:
    Dinic(int v, int e, int s, int t) {
        // ...
        // 初始化时为记录当前弧的数组分配空间。
        idx = 0, head = new int[v], dist = new int[v], current = new int[v];
        // ...
    }

    ~Dinic() { delete [] arcs, head, dist, current; }

    int dfs(int u, int flow) {
        // ...
        // 记录当前弧。此处使用了取引用运算符，以便在更新 i 的同时更新 current[u]。
        for (int &i = current[u]; ~i; i = arcs[i].next) {
            // ...
        }
        // ...
    }

    int maxFlow() {
        int flow = 0;
        while (bfs()) {
            // 在每次构建层次网络之后、寻找阻塞流之前，都需要对当前弧进行初始化。
            for (int i = 0, i < vertices, i++) current[i] = head[i];
            while (int delta = dfs(source, INF))
                flow += delta;
        }
        return flow;
    }
}

```

强化练习：[11380 Down Went The Titanic^C](#), [12125 March of the Penguins^D](#), [12873 The Programmers^D](#)。

扩展练习：[11167 Monkeys in the Emei Mountain^D](#)。

10.5.5 ISAP 算法

在前述介绍的 Edmonds-Karp 算法和 Dinic 算法中，如果使用 BFS 来确定增广路径，则寻找得到的增广路径在可行增广路径中具有最少的边数，因此被称为最短增广路径（Shortest Augmenting Path, SAP）算法。改进最短增广路径（Improved Shortest Augmenting Path, ISAP）算法由 Ahuja 和 Orlin 提出，它是一种基于距离(distance-directed)的最大流算法^[118]，相较于 Dinic 算法，虽然其时间复杂度同为 $O(V^2E)$ ，但使用相应的优化策略后，在实践中一般具有更短的运行时间，而且可以使用非递归予以实现，适合处理数据规模较大的最大流问题。下面简要介绍此算法并给出参考实现。

在残留网络中, 相对于有向弧 $\langle u, v \rangle$ 的残余容量 r_{u-v} , 可以定义距离函数 $d: V \rightarrow \mathbb{Z}^+$, 即定义一个将顶点映射到非负整数的函数 d 。进一步地, 将残留网络中某个顶点 u 的距离函数值 $d(u)$ 称作 u 的距离标号。如果距离函数 d 满足以下两个条件, 则称该距离函数 d 是有效的 (valid):

- (1) 对于汇点 t , 有 $d(t)=0$, 即汇点 t 的距离标号为 0;
- (2) 对于残留网络中每条可用流量 r_{u-v} 大于零的弧 $\langle u, v \rangle$, 满足 $d(u) \leq d(v)+1$ 。

距离标号实际上给出了残留网络中顶点 u 与汇点 t 最短距离的下限。如果对于残留网络的任意一个顶点 u 来说, 其距离标号 $d(u)$ 恰等于从顶点 u 到汇点 t 的最短距离, 则称距离函数 d 是精确的 (exact)。可以证明, 对于源点 s 来说, 一旦 $d(s) \geq |V|$, 则表明在残留网络中不存在从源点 s 到达汇点 t 的增广路径。

对于残留网络中的一条有向弧 $\langle u, v \rangle$ 来说, 如果它满足条件 $d(u)=d(v)+1$, 则称该有向弧为允许弧 (admissible arc), 其他不满足条件 $d(u)=d(v)+1$ 的有向弧称为非允许弧 (inadmissible arc)。如果一条从源点 s 到汇点 t 的路径均由允许弧构成, 那么这条路径称为允许路径 (admissible path)。实际上, 允许路径是从源点 s 到汇点 t 的一条最短路径, 由于允许路径上的每条弧都满足残余容量 r_{u-v} 大于零的条件, 因此允许路径构成了一条最短增广路径。

以下是 ISAP 算法的伪代码表示^I。算法维护了一条不完整的允许路径 (partial admissible path), 这条路径从源点 s 出发, 到达某个顶点 u , 中间由允许弧构成路径的主体。算法基于此条路径的最后一个顶点——称为当前顶点 (current node) ——执行前进 (advance) 和后退 (retreat) 操作。如果从当前顶点出发存在允许弧 $\langle u, v \rangle$, 算法执行前进操作, 将此允许弧添加到路径的末端, 否则, 算法执行后退操作, 将路径末端的弧删除, 同时对当前顶点的距离标号执行更新操作。如果路径到达汇点 t , 表明寻找找到一条可以用于增广的路径, 算法执行增广操作, 更新最大流的流量以及残留网络本身。若在算法过程中发现 $d(s) \geq |V|$, 表明在残留网络中已不存在从源点 s 到汇点 t 的可行增广路径, 算法结束。

```

algorithm IMPROVED-SHORTEST-AUGMENTING-PATH
1  $x \leftarrow 0$ 
2 Perform a (reverse) breadth-first search of the residual network
   starting with the sink node to compute exact distance labels  $d(u)$ 
3  $u \leftarrow s$ 
4 while  $d(s) < n$ 
5   do if  $G_x$  contains an admissible arc  $\langle u, v \rangle \in E_x(u)$ 
6     then ADVANCE( $u$ )      // 发现允许弧, 执行“前进”操作
7     if  $u = t$             // 到达汇点, 发现一条允许路径
8       then AUGMENT // 执行“增广”操作
9        $u \leftarrow s$       // 开始寻找下一条允许路径
10      else RETREAT( $u$ )    // 从  $u$  出发未发现允许弧, 执行“回退”操作
11 return  $x$ 

procedure ADVANCE( $u$ )
1 let  $\langle u, v \rangle$  be an admissible arc in  $E_x(u)$ 
2  $\pi(v) \leftarrow u$     // 维护前驱列表
3  $u \leftarrow v$       // 更新顶点  $v$  为当前顶点

```

^I 在 Ahuja 和 Orlin 的论文中, 伪代码由 4 个步骤 (procedure) 构成: 主循环 (main cycle)、前进 (advance)、后退 (retreat)、增广 (augment)。

```

procedure RETREAT( $u$ )
1  $d(u) \leftarrow 1 + \min\{d(v) : \langle u, v \rangle \in E_x(u)\}$  // 重标记距离标号
2 if  $u \neq s$ 
3 then  $u \leftarrow \pi(u)$  // 回溯

procedure AUGMENT
1 using the predecessor indices  $\pi$  identity an augmenting path  $P$ 
2  $\delta \leftarrow \min\{r_{u-v} : \langle u, v \rangle \in P\}$ 
3 augment  $\delta$  units of flow along  $P$ 
4 update  $G_x$  (or,  $E_x$ )

```

Ahuja 和 Orlin 在论文中建议应用一种被人们称为当前弧优化 (current arc optimization) 技巧来提高算法的运行效率。在算法中, 维护一个有向弧列表 $\text{Arcs}(u)$, $\text{Arcs}(u)$ 包含从顶点 u 出发的全部有向弧。可以使用任意的顺序来安排有向弧在 $\text{Arcs}(u)$ 中的位置, 一旦确定后即保持有向弧的顺序固定不变。为每个顶点 u 定义一条当前弧 (current arc), 当前弧表示在 $\text{Arcs}(u)$ 中将被进行“是否满足允许弧约束”测试的那条有向弧。在初始的时候, 顶点 u 的当前弧是 $\text{Arcs}(u)$ 中的第 1 条弧, 使用允许弧的约束条件对该有向弧进行测试, 检查其是否满足条件, 如果不满足条件, 将 $\text{Arcs}(u)$ 中的下一条有向弧作为当前弧。算法重复该过程, 直到出现以下两种情形之一: 要么发现 1 条允许弧, 要么到达列表的末端。在后一种情形中, 算法宣布 $\text{Arcs}(u)$ 不包含允许弧, 接着算法重新将 $\text{Arcs}(u)$ 的第 1 条弧作为当前弧, 执行后退操作, 并重标记顶点 u 的距离标号。由于避免了每次执行后退操作时都从 $\text{Arcs}(u)$ 的第 1 条弧开始寻找允许弧, 因此能够在一定程度上提升算法的运行效率。Ahuja 和 Orlin 在论文中论证了 Dinic 算法和 ISAP 算法的关联, 证明 Dinic 算法实际上是 ISAP 算法在某种条件下的特例, 因此当前弧优化可以应用于较早提出的 Dinic 算法中。

Ahuja 和 Orlin 还建议对算法应用一种被人们称为间隙优化 (gap optimization) 的技巧来提高运行效率^[119], 实践证明, 该优化在绝大多数情况下都非常有效。观察算法的执行过程, 如果不提前退出主循环, 算法在发现最大流后仍然会进行很多无用的距离重标记操作, 如果能够提前结束算法, 自然能够带来效率的提升。基于这个原因, 可以考虑引入一个大小为 $|V| + 1$ 的一维数组 gap , 其下标从 0 到 $|V|$, 数组元素 $gap[i]$ 表示距离标号为 i 的顶点数量。算法在初始时通过 BFS 来确定顶点的距离标号, 从而为数组 gap 赋予初值。初始化完成后, 数组 gap 中的元素值具有如下特征: 数组中值为正整数的元素必定是连续的, 也就是说, 一定存在某个序号 j , 对于 $gap[0], gap[1], \dots, gap[j]$ 来说, 其值均为正整数, 而对于大于序号 j 的数组元素, 其值为 0。当算法执行回退操作, 需要为有向弧 $\langle u, v \rangle$ 的起始顶点 u 更新其距离标号, 如果将顶点 u 的距离标号从 x 增加到 y , 则将数组元素 $gap[x]$ 的值减少 1, 同时将数组元素 $gap[y]$ 的值增加 1, 再检查 $gap[x]$ 是否为 0, 如果 $gap[x]$ 为 0, 即距离标号为 x 的顶点数量为 0, 表明距离标号在 gap 数组中出现了“断层”, 亦即“间隙”。可以证明, 在此种情况下, 残留网络中已不存在增广路, 当前的可行流已经为最大流, 算法可以安全地终止。间隙优化在某种程度上来说属于一种启发式优化技巧, 但实验表明, 相较于 Edmonds-Karp 算法和 Dinic 算法, 间隙优化对 ISAP 算法的效率提升确实非常明显。可能原因是 Edmonds-Karp 算法和 Dinic 算法的时间复杂度上界都是比较紧的, 而 ISAP 算法的时间复杂度上界则比较松。

以下是 ISAP 算法的非递归参考实现, 使用链式前向星表示图, 加入了当前弧优化和间隙优化来提高运行效率。

```

//-----10.5.5.1.cpp-----//
const int MAXV = 1010, MAXE = 1 << 20, INF = 0xffffffff;

```

```

struct arc {
    int u, v, capacity, residual, next;
} arcs[MAXE];

struct ISAP {
    int idx, vertices, source, sink;
    int head[MAXV], d[MAXV], father[MAXV], current[MAXV], gap[MAXV];

    // 注意: 在使用 ISAP 算法之前, 需要使用该方法进行初始化。
    // V 表示残留网络中顶点的数量, S 为源点的序号, T 为汇点的序号。
    void initialize(int V, int S, int T) {
        idx = 0;
        vertices = V, source = S, sink = T;
        memset(head, -1, sizeof(head));
    }

    // 向残留网络添加有向弧的方法。
    void addArc(int u, int v, int capacity) {
        arcs[idx] = (arc){u, v, capacity, capacity, head[u]};
        head[u] = idx++;
        arcs[idx] = (arc){v, u, capacity, 0, head[v]};
        head[v] = idx++;
    }

    // 以汇点为起点, 使用 BFS 确定各个顶点的精确距离标号。
    bool bfs() {
        int u, v;
        for (int i = 0; i < vertices; i++) d[i] = vertices;
        queue<int> q;
        q.push(sink);
        d[sink] = 0;
        while (!q.empty()) {
            u = q.front(); q.pop();
            for (int i = head[u]; ~i; i = arcs[i].next) {
                v = arcs[i].v;
                if (d[v] == vertices && arcs[i ^ 1].residual) {
                    d[v] = d[u] + 1;
                    q.push(v);
                }
            }
        }
        // 初始化间隙数组。
        memset(gap, 0, sizeof(gap));
        for (int i = 0; i < vertices; i++) gap[d[i]]++;
        return d[source] < vertices;
    }

    int maxFlow() {
        int u, v, advanced, volume = INF, flow = 0;
        // 若无法建立有效的距离标号则退出。
        if (!bfs()) return flow;
        // 初始化当前弧。
        memcpy(current, head, sizeof(head));
        // 以源点为起点构建增广路径。
        u = father[source] = source;
        while (d[source] < vertices) {

```

```

advanced = 0;
// 从当前顶点出发寻找允许弧，使用变量引用技巧同步更新当前弧。
for (int &i = current[u]; ~i; i = arcs[i].next) {
    v = arcs[i].v;
    if (d[u] == (d[v] + 1) && arcs[i].residual) {
        // 执行前进操作。
        father[v] = u, u = v;
        volume = min(volume, arcs[i].residual);
        if (v == sink) {
            // 增广，更新最大流和残留网络。
            flow += volume;
            for (u = father[v]; u != source; u = father[u]) {
                arcs[current[u]].residual -= volume;
                arcs[current[u] ^ 1].residual += volume;
            }
            volume = INF;
        }
        advanced = 1;
        break;
    }
}
if (advanced) continue;
// 执行回退操作。
int dist = vertices - 1;
for (int i = head[u]; ~i; i = arcs[i].next)
    if (arcs[i].residual && d[arcs[i].v] < dist) {
        dist = d[arcs[i].v];
        current[u] = i;
    }
// 间隙优化。
if (--gap[d[u]] == 0) break;
d[u] = dist + 1, ++gap[d[u]];
u = father[u];
}
return flow;
}
//-----10.5.5.1.cpp-----//

```

以下是 ISAP 算法的递归形式实现，形式上与 Dinic 算法的参考实现类似，相较于之前给出的非递归实现来说，代码显得更为紧凑，不过在可理解性上可能要稍差一些。

```

//-----10.5.5.2.cpp-----//
const int MAXV = 1010, MAXE = 1 << 20, INF = 0x7fffffff;

struct arc {
    int u, v, capacity, residual, next;
} arcs[MAXE];

struct ISAP {
    int idx, vertices, source, sink;
    int head[MAXV], d[MAXV], father[MAXV], current[MAXV], gap[MAXV], q[MAXV];

    void initialize(int V, int S, int T) {
        idx = 0;
        vertices = V, source = S, sink = T;

```

```

        memset(head, -1, sizeof(head));
    }

void addArc(int u, int v, int capacity) {
    arcs[idx] = (arc){u, v, capacity, capacity, head[u]};
    head[u] = idx++;
    arcs[idx] = (arc){v, u, capacity, 0, head[v]};
    head[v] = idx++;
}

bool bfs() {
    int u, v, front, rear;
    // 初始化相关变量。
    memset(d, 0, sizeof(d));
    memset(gap, 0, sizeof(gap));
    memcpy(current, head, sizeof(head));
    // 与前述非递归实现不同，此处汇点的距离标号定为 1。
    gap[d[sink]] = 1]++;
    // 将汇点作为 BFS 的起点放入队列中。
    q[front = rear = 0] = sink;
    // 执行 BFS。
    while (front <= rear) {
        u = q[front++];
        for (int i = head[u]; ~i; i = arcs[i].next) {
            v = arcs[i].v;
            if (!d[v] && arcs[i ^ 1].residual) {
                gap[d[v]] = d[u] + 1]++;
                q[++rear] = v;
            }
        }
    }
    // 返回源点是否可达。
    return d[source] <= vertices;
}

int dfs(int u, int volume) {
    if (u == sink) return volume;
    int flow = 0;
    // 使用变量引用技巧同步更新当前弧。
    for (int &i = current[u]; ~i; i = arcs[i].next) {
        int v = arcs[i].v;
        if (d[u] == d[v] + 1 && arcs[i].residual) {
            // 沿着允许弧前进，利用递归的“隐式栈”更新最大流和残留网络。
            int tmp = dfs(v, min(volume, arcs[i].residual));
            flow += tmp, volume -= tmp;
            arcs[i].residual -= tmp, arcs[i ^ 1].residual += tmp;
            if (!volume) return flow;
        }
    }
    // 间隙优化。
    if (!(--gap[d[u]])) d[source] = vertices + 1;
    gap[+d[u]]++, current[u] = head[u];
    return flow;
}

int maxFlow() {

```

```

if (!bfs()) return 0;
int flow = dfs(source, INF);
// 由于汇点的距离标号定为 1, 故此处不等式需要加上等号。
while (d[source] <= vertices) flow += dfs(source, INF);
return flow;
}
};

//-----10.5.5.2.cpp-----/

```

强化练习: 10330 Power Transmission^A。

10.5.6 最小截问题

在容量网络 $G=(V, E)$ 中, 设 E' 是 E 的子集, 如果在 G 的基图中删去 E' 后不再连通, 则称 E' 是 G 的截 (cut)。截将 G 的顶点集 V 划分为两个子集 S 和 $T=V-S$, 将截记为 (S, T) 。如果截所划分成两个子集满足源点 V_s 在顶点集 S 中, 汇点 V_t 在顶点集 T 中, 则称该截为 $S-T$ 截。根据顶点归属集合的不同, 将截 (S, T) 中的弧 $\langle u, v \rangle$, $u \in S, v \in T$, 称为截的前向弧, 而将弧 $\langle u, v \rangle$, $u \in T, v \in S$ 称为截的反向弧。在以上定义的基础上, 可以定义以下若干概念。

截的容量 (capacity of cut): 设 (S, T) 为容量网络 $G=(V, E)$ 的截, 其容量为所有前向弧的容量之和, 记为 $c(S, T)$, 可以表示为

$$c(S, T) = \sum c(u, v), u \in S, v \in T, \langle u, v \rangle \in E$$

最小截 (minimum cut): 容量网络 $G=(V, E)$ 的最小截是指容量最小的截。

截的净流量 (flow of cut): 设 f 是容量网络 $G=(V, E)$ 的一个可行流, (S, T) 是 G 的截, 定义截的净流量 $f(S, T)$ 为

$$f(S, T) = \sum f(u, v), u \in S, v \in T, \langle u, v \rangle \in E \text{ 或 } \langle v, u \rangle \in E$$

反向弧的流量在统计截的净流量时为负值, 统计截的容量时, 不计入反向弧的容量。

在一个容量网络 $G=(V, E)$ 中, 设有任意一个流为 f , 关于 f 的任意一个截为 (S, T) , 则 f 的流量等于截 (S, T) 的净流量, 且小于或者等于截 (S, T) 的容量。

最大流最小截定理 (max-flow min-cut theorem): 给定容量网络 $G=(V, E)$, 其最大流的流量等于最小截的容量。

设容量网络 $G=(V, E)$ 的一个可行流为 f , 根据上述定理, 可以得出以下四个命题是等价的:

- (1) f 是容量网络 G 的最大流;
- (2) $|f|$ 等于容量网络最小截的容量;
- (3) 容量网络中不存在增广路;
- (4) 残留网络 G_f 中不存在从源点到汇点的路径。

根据最大流最小截定理, 如果需要求最小截的容量, 则可以将其转化为网络最大流问题进行求解。如果还需要进一步知道最小截由哪些边构成, 或者需要求出最小截将顶点划分为哪两个子集, 则可按照下述方法求解: 根据网络最大流的求解思路, 当在残留网络中从源点 V_s 出发无法遍历汇点时, 所求得的网络流就是最大流, 此时, 从源点 V_s 能遍历到的顶点就构成最小截 (S, T) 中的顶点集合 S , 其余顶点构成顶点集合 T , 因此, 可以先求得网络最大流, 然后在残留网络 G' 中, 从源点 V_s 出发, 使用 DFS, 将遍历到的顶点加入集合 S , 其余未遍历到的顶点加入集合 T , 那么连接 S 和 T 的所有弧构成了容量网络的一个最小截 (S, T) 。在最小截 (S, T) 中的所有前向弧在网络最大流中一定是饱和弧, 但饱和弧不一定就是最小截中的弧。

最小截问题中，图像分割（image segmentation）是其中的一个典型的问题，也是其他最小截问题的一个基础模型，有必要予以介绍。给定一张由像素点构成的图片，需要将图片中的物品与背景予以适当区分，这就牵涉到单个像素到底属于前景（foreground）还是属于背景（background），亦即将像素点划分为若干相邻的区域，此即图像分割。对于每个像素点 i ，赋予一个可能性值 a_i 表示它属于前景，值 b_i 表示它属于背景，当相邻的两个像素点 i 和 j 不同属于背景或者前景时，赋予一个值 p_{ij} 表示该像素对不属于同一部分时的“惩罚”，令图像中所有相邻像素点所构成的无向边 (i, j) 所构成的集和为 E ，那么图像分割即需要将所有像素点划分为两个集合 A 和 B ，并最大化以下值

$$q(A, B) = \sum_{i \in A} a_i + \sum_{j \in B} b_j - \sum_{(i, j) \in E, |A \cap (i, j)|=1} p_{ij}$$

强化练习：[1212 Duopoly^E](#)，[1515 Pool Construction^D](#)，[10480 Sabotage^B](#)，[11506 Angry Programmer^C](#)。

扩展练习：[10982* Troublemakers^D](#)。

10.5.7 最小费用最大流问题

在一般最大流问题中，考虑的仅仅是向边的流量，如果有向边不仅拥有流量还赋予了另外的边权属性——该属性表示该有向边的一种“费用”属性，例如价值、距离、高度等等，在求解最大流时，不仅需要流量最大，而且要求所使用的费用最小，此即最小费用最大流（min cost max flow, MCMF）问题。MCMF 问题和 MF 问题的求解，其基本框架是一致的，都是不断的寻找增广路直到不能继续进行增广为止，不同之处在于 MCMF 问题在寻找增广路时，每次都是寻找费用最小的增广路。可以证明，只要每次寻找的都是费用最小的增广路，则最后得到的就是最小费用最大流。寻找最小费用增广路相当于在加权图中寻找单源最短路径。MCMF 问题的建模和一般网络流问题建模步骤类似，其中的差别是“费用”的处理，对于正向弧来说，“费用”是正值，但是在对应的反向弧中，“费用”需要设置为相应的负值，因为经过反向弧时，总费用应该是抵消而不是增加，也就是说建模得到的容量网络存在负权边，在包含负权边的图中寻找最短路径需要使用 Bellman-Ford 算法，而不能使用 Moore-Dijkstra 算法。

10806 Dijkstra Dijkstra^B（狄克斯特拉，狄克斯特拉）

两名犯人准备越狱，他们计划沿着街道到达火车站然后乘车离开。为了尽量避人耳目，两名犯人到达火车站的路径不能经过同一条街道两次。经过每条街道都需要一定时间，如果存在满足要求的逃离方案，输出最少的用时。所用时间指两名犯人从监狱到达火车站所花费的时间之和。两名犯人不能同时逃离，只有当第一名犯人到达火车站之后，第二名犯人才能从监狱出发。

输入

输入包含多组测试数据，每组测试数据以整数 n ($2 \leq n \leq 100$) 开始，表示十字路口的数量，监狱位于十字路口 1，火车站位于十字路口 n 。接着是一个整数 m ，表示街道的数量，之后的 m 行给出每条街道的信息，每行由三个整数构成，分别表示该条街道所连接的十字路口编号和穿过此条街道所需花费的时间（单位为秒），所有街道的穿越时间不会超过 1000 秒或者小于 1 秒，每条街道连接不同的两个十字路口，而且不同十字路口之间最多只有一条街道连接。输入最后一行为 0。

输出

如果存在满足要求的逃离方案，输出最少的用时，如果不存在，输出“Back to jail”。

样例输入

样例输出

```

3
3
1 3 10
2 1 20
3 2 50
0

```

80

分析

简单来说，题目给定了一个加权无向图，要求从指定点出发，找到两条边不重合路径到达指定终点，并计算最小边权和。可以将其建模为 MCMF 问题加以解决。对于本题来说，由于同一条街道只能经过一次，那么在构造流量网络时，街道对应的边的流量为 1，其费用为经过此街道所需要花费的时间。为了便于处理，可以添加一个虚拟的源点 s ，从源点 s 向表示监狱的顶点连接一条边，其容量为 2，费用为 0，最后求从源点 s 开始，到达表示火车站的顶点 n （即汇点）的最小费用最大流。可以应用 SPFA 寻找最小费用增广路，反复增广直到不存在增广路，最后所得即为最小费用最大流。当最大流的流量能够达到 2 时，表明存在两条边不重合路径，从监狱到达火车站且两条路径的用时之和最小，否则不存在满足条件的路径。此处需要注意“费用”的处理，正向弧“费用”表示的是穿越某条街道所花费的时间，为正值，反向弧的“费用”则是花费时间的相反值，为负值。

参考代码

```

const int MAXV = 110, MAXE = 20100, INF = 0x7f7f7f7f;

struct arc { int u, v, capacity, residual, cost, next; } arcs[MAXE];

int idx, source, sink, dist[MAXV], head[MAXV], parent[MAXV], visited[MAXV];
int n, m, fee, flow;

bool spfa() {
    for (int i = 0; i < MAXV; i++)
        dist[i] = INF, parent[i] = -1, visited[i] = 0;
    dist[source] = 0, visited[source] = 1;
    queue<int> q; q.push(source);
    while (!q.empty()) {
        int u = q.front(); q.pop(); visited[u] = 0;
        for (int i = head[u]; ~i; i = arcs[i].next) {
            arc e = arcs[i];
            if (e.residual > 0 && dist[e.v] > dist[u] + e.cost) {
                dist[e.v] = dist[u] + e.cost;
                parent[e.v] = i;
                if (!visited[e.v]) {
                    q.push(e.v);
                    visited[e.v] = 1;
                }
            }
        }
    }
    return dist[sink] < INF;
}

void addArc(int u, int v, int capacity, int cost) {
    arcs[idx] = (arc){u, v, capacity, capacity, cost, head[u]};
    head[u] = idx++;
    arcs[idx] = (arc){v, u, capacity, 0, -cost, head[v]};
}

```

```

        head[v] = idx++;
    }

void mcmf() {
    fee = flow = 0;
    while (spfa()) {
        int delta = INF;
        for (int i = parent[sink]; ~i; i = parent[arcs[i].u])
            delta = min(delta, arcs[i].residual);
        flow += delta, fee += delta * dist[sink];
        for (int i = parent[sink]; ~i; i = parent[arcs[i].u]) {
            arcs[i].residual -= delta;
            arcs[i ^ 1].residual += delta;
        }
    }
}

int main(int argc, char *argv[]) {
    int u, v, t;
    while (cin >> n, n > 0) {
        idx = 0, source = 0, sink = n;
        memset(head, -1, sizeof(head));
        cin >> m;
        for (int i = 1; i <= m; i++) {
            cin >> u >> v >> t;
            addArc(u, v, 1, t);
            addArc(v, u, 1, t);
        }
        addArc(0, 1, 2, 0);
        mcmf();
        if (flow < 2) cout << "Back to jail\n";
        else cout << fee << '\n';
    }
    return 0;
}

```

强化练习：[10594 Data Flow^C](#)，[11301 Great Wall of China^D](#)，[12821 Double Shortest Paths^E](#)。

10.6 边独立集与二部图匹配

设非空无向图 $G=(V, E)$ ，取边集 $E(G)$ 的非空子集 M ，若 M 中任何两条边均不相邻，则称 M 为 G 的边独立集 (edge independent set)。在解题应用中，一般需要求解极大边独立集和最大边独立集，其目标是将图中尽可能多的相互独立的边包含到 M 中，同时使 M 符合边独立集的定义。若在 M 中加入一条属于 $E(G) \setminus M$ 的边后，所得到的集合 M' 不再满足边独立集的定义，则称 M 为极大边独立集。最大边独立集则容易理解，顾名思义，它是指具有最大边数的边独立集。将图 G 中最大边独立集所包含的边数称为边独立数 (edge independent number)，记做 $\alpha(G)$ 。需要注意，极大边独立集不一定是最大边独立集，而最大边独立集一定是极大边独立集。

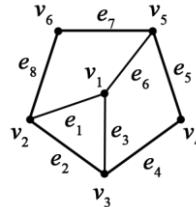


图 10-15 $\{e_4, e_7\}$ 是边独立集但不是极大边独立集, 也不是最大边独立集; $\{e_2, e_5\}$ 是极大边独立集, 但不是最大边独立集, $\{e_1, e_4, e_7\}$ 和 $\{e_3, e_5, e_8\}$ 是最大边独立集同时也是极大边独立集, 由此可知, 图的最大边独立集可能不唯一

通过观察图 10-15, 可以发现边独立集等价于将顶点集 $V(G)$ 中的顶点进行“配对”, “顶点对”之间相互独立, 不会发生重叠, 因此有的文献也将边独立集称为 G 的匹配 (matching), 进而将图 G 中最大边独立集所包含的边数称为匹配数 (matching number), 记做 $\alpha'(G)$ 。还有的文献则从饱和点 (saturated vertex) 的角度来引入匹配和完备匹配的概念。将图 G 中与边集 M 中边关联的顶点称为 M 饱和点, 反之, 称之为 M 非饱和点 (unsaturated vertex)。令顶点集 X 是 $V(G)$ 的非空子集, 若 X 中每个顶点都是 M 饱和点, 则称 M 饱和 X , 若 M 饱和 $V(G)$, 则称 M 为 G 的完备匹配 (perfect matching, 又称完美匹配), 若对 G 的任何匹配 M' 均有 $|M'| \leq |M|$, 则称 M 为 G 的最大匹配 (maximum matching)。显然每个完备匹配都是最大匹配, 但最大匹配不一定是完备匹配。在本书的后续行文中, 均以匹配来指代边独立集。

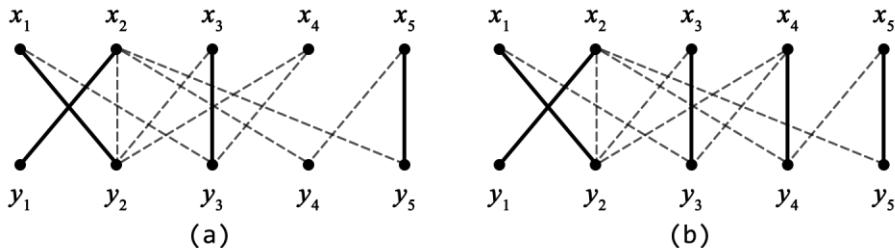


图 10-16 (a) 最大匹配; (b) 完备匹配。其中实线边表示该边属于匹配, 虚线边表示该边不属于匹配

若图 G 的顶点集 V 能划分为两个非空子集 X 和 Y , 使得 X 中任意两个顶点之间无边相连, 并且 Y 中任意两个顶点之间也无边相连, 则称该图为二部图 (bipartite graph), 与此同时, 将 $\{X, Y\}$ 称为二部划分 (bipartition)。如果图 G 不包含奇圈, 则可对图 G 进行二着色, 也就意味着可以对图 G 进行二部划分, 亦即图 G 为二部图。二部划分为 $\{X, Y\}$ 的二部图即为 $(X \cup Y, E)$, 如果 $|X| = |Y|$, 则 $(X \cup Y, E)$ 称为等二部图 (equally bipartite graph)。对于给定的简单二部图 $(X \cup Y, E)$, 如果 X 中的每个顶点和 Y 中的每个顶点均有边相连, 则称该二部图为完全二部图 (complete bipartite graph)。

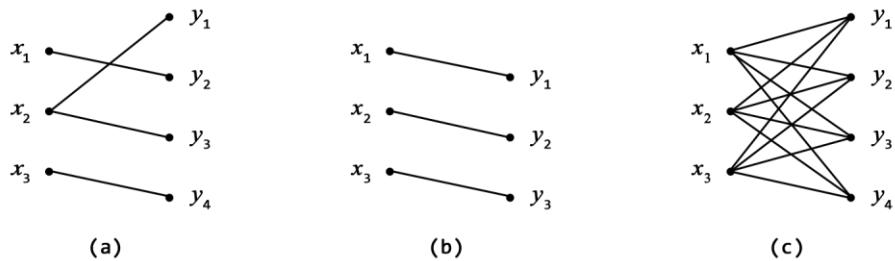


图 10-17 (a) 二部图; (b) 等二部图; (c) 完全二部图

二部图匹配实际上来源于人员安排问题 (personnel assignment problem)，可简要叙述如下：某工厂有 m 名工人 x_1, x_2, \dots, x_m ，每名工人能够从事 n 种工种 y_1, y_2, \dots, y_n 中的若干种，限制每名工人同一段时间只能从事一个工种，为了使尽可能多的工种有工人从事，问如何安排工人和工种之间的对应关系，使得该段时间内有工人从事的工种数量最大。人员安排问题可以转化为图论中的二部图匹配问题，并存在相应的有效算法予以解决。

10.6.1 网络流解法

可以将二部图匹配问题转化为网络最大流问题进行解决。设二部图为 $G=(V, E)$ ，它的顶点集 V 所包含的两个子集为 $X=\{x_1, x_2, \dots, x_m\}$ 和 $Y=\{y_1, y_2, \dots, y_n\}$ ，如果把二部图视为一个容量网络，边 (x_i, y_j) 对应网络中的弧 $\langle x_i, y_j \rangle$ ，则在求最大匹配时要保证从顶点 x_i 出发的边最多只选一条，进入顶点 y_j 的边最多也只选一条。建立源点 s ，将 s 到 x_i 的弧 $\langle s, x_i \rangle$ 的容量设定为 1，这样就能保证从顶点 x_i 出发的边最多只选一条；建立汇点 t ，将 y_j 到 t 的弧 $\langle y_j, t \rangle$ 的容量设定为 1，这样就能保证进入顶点 y_j 的边最多也选一条，弧 $\langle x_i, y_j \rangle$ 的容量也为 1。最后求建立的容量网络的最大流即可。

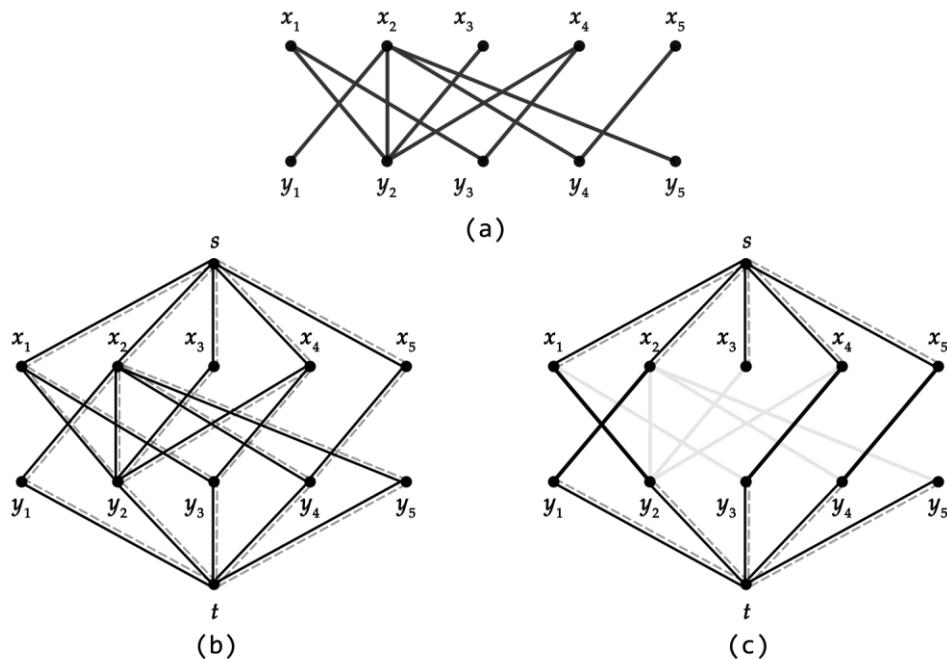


图 10-18 (a) 二部图; (b) 根据二部图得到的流网络, 实线表示正向弧, 虚线表示反向弧, 容量均为 1; (c) 一种可能的最大流, 流量为 4, 图中顶点之间的实线边表示由最大流得到的相应最大匹配, 亦为 4

可将网络流解法具体求解步骤归纳如下:

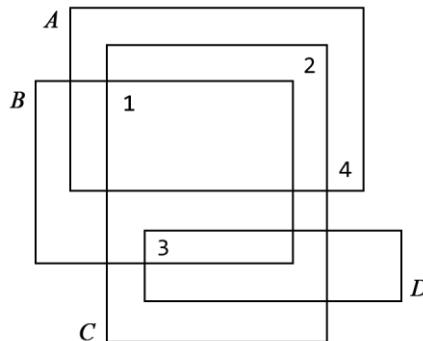
- (1) 建立源点 s 和汇点 t ;
- (2) 从 s 向 X 的每一个顶点 x_i 都建立一条有向弧, 从 Y 的每一个顶点 y_j 向 t 建立一条有向弧;
- (3) 图 G 中的边都改成有向弧, 方向从 X 的顶点 x_i 指向 Y 的顶点 y_j ;
- (4) 将所有有向弧的容量都设定为 1;
- (5) 求构造得到的容量网络 G' 的最大流 F ;
- (6) 在最大流 F 中, 检查从 X 的顶点 x_i 指向 Y 的顶点 y_j 的弧集合, 其中流量为 1 的弧对应着二部图中最大匹配的边, 最大流 F 的流量对应二部图的最大匹配的边数。

上述方法特别适合于求解某些具有特殊条件限制的二部图匹配问题, 因为可以方便地将约束转换为流量网络中的容量限制。需要注意, 在使用网络流算法求解二部图匹配问题时, 一般需要根据题目测试数据的规模来选用不同时间复杂度的最大流算法。

663 Sorting Slides^D (幻灯片排序)

Clumsey 教授今天下午有一堂重要的授课。不巧的是, 教授不是一个非常有条理的人, 他把所有的幻灯片堆成了一大摞, 而在讲课之前, 需要将这些幻灯片按演示顺序进行排列。作为一个最简主义者, 教授想通过最少的步骤来达到此目的。

情况是这样的: 所有的幻灯片上都有一个数字, 该数字表示幻灯片在演示时的序号。由于幻灯片叠在一起而且是透明的, 所以难以直接判断到底哪个数字印在哪张幻灯片上。尽管无法直接分辨数字在哪张幻灯片上, 但可以通过推理来确定数字的归属。如下图所示, 使用字母 A 、 B 、 C 、 D 命名幻灯片, 很显然, D 的序号为 3, A 的序号为 4, C 的序号为 2, B 的序号为 1。



现在要求你根据给定的条件确定幻灯片的命名和序号之间的对应关系。

输入

输入包含多组堆叠的幻灯片描述。每组描述以一个数字 n 开始, 表示该叠幻灯片的数量, 接着 n 行每行包含四个整数 x_{min} , y_{min} , x_{max} , y_{max} , 表示幻灯片矩形的左下角和右上角坐标 (坐标系原点位于屏幕的左下角), 幻灯片按照字母 A 、 B 、 C 、…的顺序命名, 再接着 n 行, 每行包含两个整数, 表示印在幻灯片上的序号数字的 x 坐标及 y 坐标, 第一组坐标表示序号 1 的坐标, 第二组坐标表示序号 2 的坐标, 依此类推。序号数字不会位于幻灯片的边界上。输入以 $n=0$ 的一组幻灯片描述作为结束, 不需处理该组数据。

输出

对于输入中给出的每组幻灯片描述，先输出数据组的编号，然后按字母顺序，依次输出能够唯一确定的数字和幻灯片命名之间的配对。如果不能推断出任何唯一的配对，输出“none”。在每组测试数据后输出一个空行。

样例输入

```
4
6 22 10 20
4 18 6 16
8 20 2 18
10 24 4 8
9 15
19 17
11 7
21 11
0
```

样例输出

```
Heap 1
(A,4) (B,1) (C,2) (D,3)
```

分析

题意要求的是“唯一确定”的数字与幻灯片命名之间的匹配关系，如果一个数字在两个匹配方案中能够匹配不同的幻灯片，那么这样的匹配不是唯一的，就不能将这样的匹配予以输出。可以将问题建模为二部图匹配问题，然后将其转化为网络流问题进行解决。将序号和幻灯片视为图 G 的顶点，表示序号的顶点对应顶点集 X ，表示幻灯片的顶点对应顶点集 Y ，则 $V(G)=X \cup Y$ ，且 X 内顶点无边连接， Y 内顶点同样无边连接， X 内顶点和 Y 内顶点根据序号是否在幻灯片内建立边，构建得到的图 G 符合二部图的定义。建立源点 $source$ ，在 $source$ 和序号间建立容量为 1 的有向边；建立汇点 $sink$ ，在幻灯片和汇点 $sink$ 之间建立容量为 1 的有向边；序号和幻灯片之间，如果某个序号位于某张幻灯片之内，则在该序号和幻灯片之间建立一条容量为 1 的有向边，求构造得到的容量网络的最大流，即为序号和幻灯片之间能够得到的最大匹配数。之后任意选择一条序号和幻灯片之间的有向边予以移除，再次求移除该有向边之后容量网络的最大流，如果最大流的流值相比之前的最大流减少，表明移除的有向边是数字和幻灯片之间的唯一对应边，即为“唯一确定”的匹配关系。由于题目所要求的数据量不大，至多只有 26 张幻灯片，使用标号法求最大流即可，如果数据量较大，可选用 Edmonds-Karp 算法或 Dinic 算法。

强化练习：[753 A Plug for UNIX^C](#)，[10080 Gopher II^A](#)，[11045 My T-Shirt Suits Me^B](#)，[11418 Clever Naming Patterns^C](#)。

10.6.2 Hungarian 算法

1955 年，Kuhn 研究并发表了解决人员安排问题的一种有效方法，因为该方法大部分基于匈牙利数学家 König 和 Egerváry 的工作，因此 Kuhn 将该方法命名为 Hungarian（匈牙利）算法^[120]。在 Kuhn 的论文中，使用的是基于矩阵的方法来描述 Hungarian 算法，在叙述、理解和实现上有诸多不便，此处介绍的是 Edmonds 于 1965 年给出的叙述，它基于交错路的概念和相关结论^[121]。

设 M 和 M' 是 $E(G)$ 的两个不相交的非空真子集， (M, M') 交错路（alternating path）是指其边在 M 和 M' 中交错出现的路。 (M, \bar{M}) 交错路简称 M 交错路，其中 $\bar{M}=E(G) \setminus M$ 。若 M 是 G 的匹配，则将两端点不同且都是 M 非饱和的 M 交错路称为 M 增广路（augmenting path），增广路的特点是非匹配边一定比匹配边多恰好 1 条，将增广路“取反”，即匹配边变为非匹配边，非匹配边变为匹配边，可以得到一个比先前

匹配数增大 1 的匹配, 因此不断 **地** 寻找增广路, 就可以不断地增加匹配。Berge 证明: 若 M 是 G 的匹配, 则 M 是最大匹配的充要条件是 G 中不存在 M 增广路^[122]。若不然, 设 M 是 G 的最大匹配, 并设 $P=x_0e_1x_1e_2\cdots e_mx_m$ 是 G 中 M 增广路, 根据 M 增广路的定义, m 是奇数, 并且 $e_1, e_3, \dots, e_m \notin M$, 而 $e_2, e_4, \dots, e_{m-1} \in M$, 令

$$M' = M \Delta E(P) = (M\{e_2, e_4, \dots, e_{m-1}\}) \cup \{e_1, e_3, \dots, e_m\}$$

则 M' 是 G 的匹配, 并且 $|M'| = |M| + 1$, 与 M 是最大匹配产生矛盾。

Hungarian 算法的正确性即基于上述结论, 其基本思想为: 从 G 的任意匹配 M 开始, 检查每一个 M 非饱和点, 然后从它出发构造可增广路, 沿着增广路进行扩增, 直到不存在增广路时算法结束, 根据 Berge 所证明的结论, 此时的匹配即为最大匹配。

Hungarian 算法

(1) 任取 G 的匹配 M , 若 M 饱和 X , 则停止; 若 M 不能饱和 X , 则取 X 的 M 非饱和点 x , 令 $S = \{x\}$, $T = \emptyset$;

(2) 若 $N(S)^T = T$, 则停止; 此时 G 中无完备匹配, 若 $N(S) \neq T$, 则取 $y \in N(S) \setminus T$;

(3) 若 y 是 M 饱和的, 则存在 $z \in X \setminus S$, 使 $yz \in M$, 用 $S \cup \{z\}$ 替代 S , $T \cup \{y\}$ 替代 T , 转入第 2 步; 若 y 是 M 非饱和的, 则 G 中存在以 x 为起点且以 y 为终点的 M 增广路 P , 用 $M' = M \Delta E(P)$ 替代 M , 转入第 (1) 步。

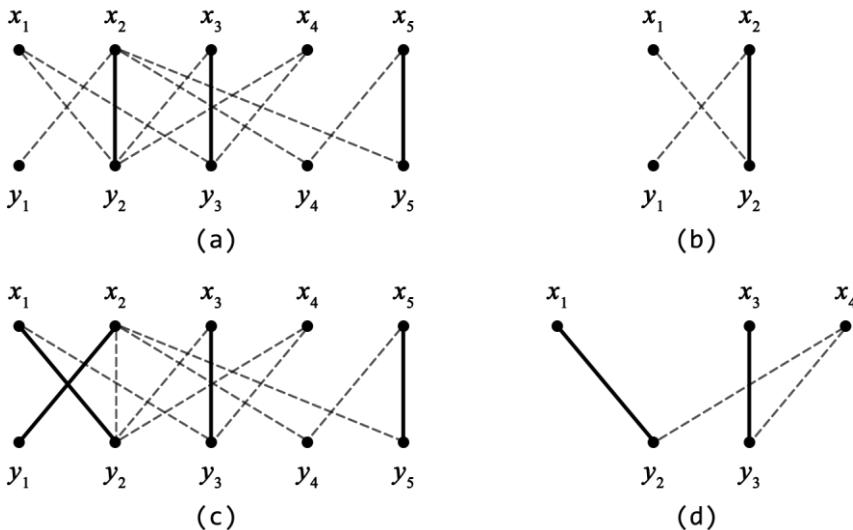


图 10-19 (a) 匹配 M_0 (实线边); (b) M_0 增广路; (c) 匹配 M_1 (实线边); (d) $N(S_2)=T_2$

由于是以图论记号进行算法描述, 初看难以理解, 下面以一个实例来说明算法的工作流程。如图 10-19 所示, 划分为 $\{X, Y\}$ 的二部图 G , 其中 $X = \{x_1, x_2, x_3, x_4, x_5\}$, $Y = \{y_1, y_2, y_3, y_4, y_5\}$, 取初始匹配 $M_0 = \{x_2y_2, x_3y_3, x_5y_5\}$, x_1 是 X 中 M_0 非饱和点, 令 $S_0 = \{x_1\}$, $T_0 = \emptyset$, 因为 $N(S_0) = \{y_2, y_3\} \neq T_0$, 取 $y_2 \in N(S_0) \setminus T_0$ 。

¹ $N(S)$ 是图论记号, 表示 S 在 G 中的邻集。 S 在此处为顶点集合, 则 $N(S)$ 表示与 S 中顶点有边连接的相邻顶点的集合。

y_2 是 M_0 饱和点, $x_2 \in X$ 使 $x_2y_2 \in M_0$ 。令 $S_1 = \{x_1, x_2\}$, $T_1 = \{y_2\}$, 则 $N(S_1) = \{y_1, y_2, y_3, y_4, y_5\} \neq T_1$, 取 $y_1 \in N(S_1) \setminus T_1$, y_1 是 M_0 非饱和点, 所以 $P_0 = x_1y_2x_2y_1$ 是 M_0 增广路, 进一步令

$$M_1 = M_0 \Delta E(P_0) = \{x_1y_2, x_2y_1, x_3y_3, x_5y_5\}$$

用 M_1 替代 M_0 再执行第 (1) 步。 $x_4 \in X$ 是 M_1 非饱和点, 令 $S_0 = \{x_4\}$, $T_0 = \emptyset$, 则 $N(S_0) = \{y_2, y_3\} \neq T_0$, y_2 是 M_1 非饱和点, 并且 $x_1y_2 \in M_1$, 令 $S_1 = \{x_1, x_4\}$, $T_1 = \{y_2\}$, 因为 $N(S_1) = \{y_2, y_3\} \neq T_1$, 取 $y_3 \in N(S_1) \setminus T_1$, y_3 是 M_1 饱和的, 而且 $x_3y_3 \in M_1$, 令

$$S_2 = \{x_1, x_3, x_4\}, T_2 = \{y_2, y_3\} = N(S_2)$$

算法停止, 由于 $|N(S_2)| < |S_2|$, 故 G 无完备匹配。

由于算法的描述使用了较多的图论语言, 理解起来可能较为晦涩难懂, 下面对其作进一步的解释。更为通俗地说, Hungarian 算法就是不断地从 X 侧尚未匹配的某个顶点开始, 试图构建一条由“非匹配边—匹配边—非匹配边—……”所组成的交错路, 这条交错路的最后一条边要求是非匹配边, 因此是一条增广路, 将此交错路“取反”, 即非匹配边变成匹配边, 将匹配边变成非匹配边, 可以得到一个比原有匹配更大的匹配, 其匹配数会增加 1, 持续寻找如上所述的交错路, 直到不能找出这样的交错路, 则此时的匹配就为最大匹配。

在寻找增广路时, 可以使用 DFS 寻找, 也可以使用 BFS 寻找。DFS 寻找增广路的特点是程序结构清晰、易于理解, 适用于边较多的稠密图; 而 BFS 寻找增广路, 适用于边较少的稀疏二部图, 其增广路较短。不过在大多数解题应用中, 使用上述两种不同方式寻找增广路的效率差别不是很大, 因为它们的时间复杂度是相同的。

以下仅给出 Hungarian 算法使用 DFS 寻找增广路的实现。使用 BFS 寻找增广路以及使用非递归的 DFS 寻找增广路的实现, 读者可以作为“思考题”加以完成, 并结合强化练习进行实际运用。

```
-----10.6.2.cpp-----//
const int MAXV = 110;

// tx 表示 X 侧顶点的数量, ty 表示 Y 侧顶点的数量。
int tx, ty;

// g 为邻接矩阵表示的图;
// visited 记录某个顶点是否已经在交错路中;
// cx 记录与某个 X 侧顶点匹配的 Y 侧顶点;
// cy 记录与某个 Y 侧顶点匹配的 X 侧顶点。
int g[MAXV][MAXV], visited[MAXV], cx[MAXV], cy[MAXV];

// 使用深度优先搜索寻找增广路。
int dfs(int u) {
    // 考虑所有与 u 邻接且尚未访问的顶点 v。
    for (int v = 0; v < ty; v++)
        // 检查顶点 u 和 v 之间是否有边连接且顶点 v 尚未访问。
        if (g[u][v] && !visited[v]) {
            visited[v] = 1;
            // 检查 v 是否已经匹配或者从“已与 v 匹配的顶点”出发是否可以找到交错路。
            if (cy[v] == -1 || dfs(cy[v])) {
                // 找到增广路, 将顶点 u 和 v 配对。
                cx[u] = v, cy[v] = u;
                return 1;
            }
        }
}
```

```

// 未能找到增广路，已经是最大匹配。
return 0;
}

// 匈牙利算法求最大匹配数。
int hungarian() {
    // 初始化匹配标记。
    memset(cx, -1, sizeof(cx));
    memset(cy, -1, sizeof(cy));
    int matches = 0;
    for (int i = 0; i < tx; i++)
        // 从 X 侧的每个非饱和点出发寻找增广路。
        if (cx[i] == -1) {
            // 寻找之前需要将访问标记重置为初始状态。
            memset(visited, 0, sizeof(visited));
            // 每找到一条增广路，可使得匹配数增加 1。
            matches += dfs(i);
        }
    // 返回匹配数。
    return matches;
}
//-----10.6.2.cpp-----//

```

如果使用邻接链表来表示图，使用 BFS 或者 DFS 来寻找增广路，则 Hungarian 算法每次寻找增广路的时间复杂度为 $O(E)$ ，总共可能增广的次数为 $O(V)$ ，则算法总的时间复杂度为 $O(VE)$ 。

10615 Rooks^D（车）

给定一个 $N \times N$ 的棋盘，上面放置了一些棋子——车。你需要使用最少的颜色为这些车着色，使得任意一行和任意一列都不包含两个相同颜色的车。

输入

输入的第一行包含整数 S ($0 < S < 10$)，表示测试数据的组数。每组测试数据的格式描述如下：第一行包含整数 N ($0 \leq N \leq 100$)，接下来的 N 行包含棋盘的描述（使用 $N \times N$ 的方阵来表示），空位标记为 ‘.’，车标记为 ‘*’（在一行的标记之间不存在空白字符）。

输出

每组测试数据的输出格式如下：输出的第一行为整数 M ，表示所需的最少颜色数量。接着是使用 N 行输出表示的棋盘，空位使用 ‘0’ 进行标记，车使用 ‘K’ 标记，其中 K 为该棋子的颜色编号（从 1 开始计数）。如果存在多种符合要求的着色方案，输出其中任意一种即可。

样例输入

```

2
2
*.
**
4
*.*.
*.*.
***.
..**

```

样例输出

```

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

```

分析

将棋盘的行视为 X 侧顶点，棋盘的列视为 Y 侧顶点，棋盘上的车视为所在行和列之间的边，可以将题约束建模为一个二部图，同样颜色的车实际上构成了行和列之间的一个匹配，因此得到以下直观的算法：使用 Hungarian 算法不断求最大匹配，匹配的性质可以保证不会有重复的行和列被选择，将每次匹配的行和列的交叉点的车染色，接着删除已经匹配的行和列之间的边，继续使用 Hungarian 算法求最大匹配，一直到最大匹配的数量为 0 为止，则总的所需要的染色数就是求最大匹配过程中成功的次数。由于每次最大匹配至少将一行与一列进行匹配，因此所需匹配的最大次数由某行（或某列）所包含车的最大数量决定。由此得到以下解题方案：

```

int main(int argc, char *argv[]) {
    int n, cases, board[MAXN][MAXN];
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> n;
        // 初始化。
        tx = ty = n;
        memset(g, 0, sizeof(g));
        // 读入数据。
        char c;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cin >> c;
                board[i][j] = 0;
                if (c == '*') g[i][j]++;
            }
        }
        // 使用 Hungarian 算法不断求最大匹配。
        int colors = 0;
        while (hungarian()) {
            colors++;
            for (int i = 0; i < n; i++) {
                if (cx[i] != -1) {
                    // 根据匹配进行染色。
                    board[i][cx[i]] = colors;
                    g[i][cx[i]]--;
                }
            }
        }
        // 输出染色方案。
        cout << colors << '\n';
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                if (j) cout << ' ';
                cout << board[i][j];
            }
            cout << '\n';
        }
    }
    return 0;
}

```

上述代码看上去没有明显的“破绽”，但是对于以下测试输入：

1
3

```
..*
*..
*.*
```

其输出为：

```
3
0 0 1
1 0 0
2 0 3
```

可以容易地看出，结果显然是错误的——着色所需的最少颜色数为 2，一种可行的着色方案为：

```
0 0 2
1 0 0
2 0 1
```

为什么前述算法不能得到正确的结果呢？下面通过观察算法的执行过程来查看一下究竟在何处出现了问题。

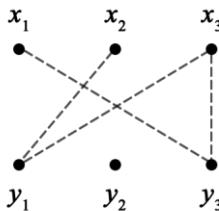


图 10-20 测试数据所对应的二部图。 x_i 表示第 i 行对应的顶点， y_i 表示第 i 列对应的顶点（从 1 开始计数序号）

如图 10-20 所示，这是测试输入所对应的二部图。前述算法在第一次执行 Hungarian 算法求最大匹配时，先将 x_1 和 y_3 、 x_2 和 y_1 匹配，此时位于第一行第三列、第二行第一列的车先染色为‘1’，接着删除 x_1 和 y_3 、 x_2 和 y_1 之间的边；在第二次执行 Hungarian 算法时，只有 x_3 和 y_1 、 x_3 和 y_3 之间有边，此时最大匹配只能是 1，要么选择 x_3 匹配 y_1 ，要么选择 x_3 匹配 y_3 ，“错误算法”选择的是 x_3 匹配 y_1 ，因此将三行第一列的车染色为‘2’，删除 x_3 和 y_1 之间的边；继续执行 Hungarian 算法中，此时最大匹配仍为 1，结果是将 x_3 匹配 y_3 ，使得第三行第三列的车染色为‘3’，最终总的染色数为 3 种。“错误算法”之所以得出了错误的结果，表面原因似乎在于第一次选择了将 x_1 和 y_3 、 x_3 和 y_1 匹配，导致后续无法一次性将剩余的顶点进行匹配，但究其根本原因却在于通过上述方法所构建的二部图并不是一个完全二部图，这样会导致每次进行匹配时得到的可能并不是完备匹配，如果不是每次都是完备匹配，就有可能出现前述所出现的情形。

那么如何避免出现这种情况呢？这就需要应用 Hall 定理的推论¹。基于 Hall 定理，可以得到两个推论：

(1) Forbenius 证明（婚姻定理）：二部划分为 $\{X, Y\}$ 的二部图 G 有完备匹配 $\Leftrightarrow |X| = |Y|$ ，并且对于任何 $S \subseteq X$ （或 Y ）均有 $|N_G(S)| \geq |S|$ 。 (2) König 证明：设 G 是 k ($k > 0$) 正则二部图，则 G 有完备匹配。根据上述两个推论，要使得构建得到的二部图每次都能够得到完备匹配，需要保证给定的二部图是完全二部图，又由于每次完备匹配后会删除已经匹配的边，为了仍能够得到完备匹配，要求在删除边后剩下的二部图仍然

¹ Hall 定理是组合数学中最基本的定理之一，它有各种表达形式，其图论表达形式为：设 G 是二部划分为 $\{X, Y\}$ 的二部图，则 G 有饱和 X 的匹配 $\Leftrightarrow |S| \leq |N_G(S)|, \forall S \subseteq X$ 。

是完全二部图，这就要求原图必须是一个 k 正则二部图，其中 k 为原图中顶点的最大度数。进一步地，如果原图不是 k 正则二部图，需要通过“补边”操作将其“改建”为 k 正则二部图^I。

在具体实现时，需要注意以下两点：(1) 进行补边操作时，如果第 r 行和第 c 列之间已经有边，但是对应的顶点度数均不超过最大度数 D ，则可以一直补边，不需要考虑顶点之间已经存在重复边；(2) 在每次找到完备匹配后，需要根据匹配情况进行染色，如果第 r 行与第 c 列匹配，而位于第 r 行第 c 列的不是车（或者是车但是已经染色）则忽略。

参考代码

```
int main(int argc, char *argv[]) {
    int n, cases, board[MAXN][MAXN];
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> n;
        // 初始化。
        tx = ty = n;
        memset(g, 0, sizeof(g));
        // D 记录顶点的最大度数，dx 记录 X 侧顶点的度数，dy 记录 Y 侧顶点的度数。
        int D = 0, dx[MAXN] = {0}, dy[MAXN] = {0};
        // 读入数据。
        char c;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                cin >> c;
                board[i][j] = 0;
                if (c == '*') {
                    g[i][j]++;
                    dx[i]++;
                    dy[j]++;
                    board[i][j] = -1;
                    D = max(D, max(dx[i], dy[j]));
                }
            }
        }
        // 通过“补边”将二部图构造成 D 正则二部图。
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n && dx[i] < D; j++) {
                while (dx[i] < D && dy[j] < D)
                    g[i][j]++;
            }
        }
        // 使用 Hungarian 算法求完备匹配。
        int colors = 0;
        while (hungarian()) {
```

^I 本题存在更为简单的“贪心”解题方法。由于最少颜色数由某行（或某列）所包含车的最大数量决定，可以考虑为车逐个染色。先将具有最大数量车的一行（或一列）内的车从 1 开始染色，然后从方阵的左上角开始以行优先顺序为尚未染色的车逐个染色。当为某个车染色时，检查该行可用的颜色集合 C_1 以及该列可用的颜色集合 C_2 ，如果两者存在交集 I ，使用交集 I 中的任意一种颜色为该车染色，若两者无交集，取 C_1 中的颜色 A ， C_2 中的颜色 B ，将该车染色为 A ，这会导致该车的颜色与同列上的另外一个车颜色相同，将另外一个车染色为 B ，这时可能又会使得另外一个车与所在行上的第三个车的颜色相同，继续将第三个车染色为 A ，如果导致第三个车所在列颜色存在冲突，同前处理。可以证明，整个过程可以在不影响第一个车染色的情况下终止。为了简便，染色过程可以使用递归实现。

参考代码：<https://github.com/metaphysis/Code/blob/master/UVa%20Online%20Judge/volume106/10615%20Rooks/program2.cpp>。

```

colors++;
for (int i = 0; i < n; i++) {
    if (cx[i] != -1) {
        // 只对未染色的车进行染色，已经染色的车忽略。
        if (board[i][cx[i]] == -1) board[i][cx[i]] = colors;
        // 通过删除边使得原有的 k 正则二部图变成 k-1 正则二部图。
        g[i][cx[i]]--;
    }
}
// 输出染色方案。
cout << colors << '\n';
for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
        if (j) cout << ' ';
        cout << board[i][j];
    }
    cout << '\n';
}
return 0;
}

```

强化练习: [670 The Dog Task^C](#), [11159 Factors and Multiples^C](#), [12159 Gun Fight^D](#), [12644 Vocabulary^D](#)。

扩展练习: [1221* Against Mammoths^E](#), [10804 Gopher Strategy^D](#), [11262 Weird Fence^D](#)。

10.6.3 Hopcroft-Karp 算法

Hopcroft-Karp 算法是对 Hungarian 算法的改进，它采用多增广路的方式进行增广，提高了效率^[123]。回顾 Hungarian 算法，每次 DFS 只能发现一条 M 增广路，因此只能使匹配数增加 1，而 Hopcroft-Karp 算法能够通过一次 DFS 来发现多条增广路，从而增加多个匹配，提高了效率。Hopcroft-Karp 算法先使用 BFS 建立层次网络，之后使用嵌套的 DFS 来实现一次遍历增加多个匹配，从形式上看，该算法和 Dinic 算法非常类似，这并不是偶然，Even 和 Tarjan 指出，Hopcroft-Karp 算法实际上可以认为是 Dinic 算法在二部图这种特殊流量网络上的应用^[124]。

可以证明，在二部图中使用 Hopcroft-Karp 算法寻找匹配，其迭代次数最多为 $2\sqrt{V}$ ，则算法总的时间复杂度为 $O(\sqrt{V}E)$ 。

```

//-----10.6.3.cpp-----//
const int MAXV = 110, INF = 0x7f7f7f7f;

int tx, ty;
int g[MAXV][MAXV], visited[MAXV], cx[MAXV], cy[MAXV], dx[MAXV], dy[MAXV];

int bfs() {
    int dist = INF;
    // 初始化变量。
    memset(dx, -1, sizeof(dx));
    memset(dy, -1, sizeof(dy));
    // 将未饱和点压入队列。
    queue<int> q;
    for (int i = 0; i < ty; i++)
        if (cx[i] == -1) {
            q.push(i);
            dx[i] = 0;
        }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = 0; v < tx; v++)
            if (g[u][v] > 0 && dy[v] == -1) {
                dy[v] = dist;
                if (v == ty)
                    return 1;
                q.push(v);
            }
        dist++;
    }
    return 0;
}

int dfs(int v) {
    if (v == ty)
        return 1;
    for (int u = 0; u < tx; u++)
        if (g[u][v] > 0 && cx[u] == -1 && dy[u] == dy[v] - 1) {
            if (dfs(cx[u] = u, v + 1))
                return 1;
        }
    return 0;
}

int maxflow() {
    int flow = 0;
    while (bfs())
        for (int v = 0; v < tx; v++)
            if (dfs(v, 0))
                flow++;
    return flow;
}

```

```

        }
        // 根据距离分层。
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (dx[u] > dist) break;
            for (int v = 0; v < ty; v++) {
                if (g[u][v] && dy[v] == -1) {
                    dy[v] = dx[u] + 1;
                    if (cy[v] == -1)
                        dist = dy[v];
                    else {
                        dx[cy[v]] = dy[v] + 1;
                        q.push(cy[v]);
                    }
                }
            }
        }
        // 确定从 x 侧顶点是否可以到达 y 侧顶点。
        return dist != INF;
    }

    int dfs(int u) {
        int dist = INF;
        for (int v = 0; v < ty; v++)
            // 沿着分层之间的可行边前进寻找交错路。
            if (g[u][v] && !visited[v] && dy[v] == (dx[u] + 1)) {
                visited[v] = 1;
                if (cy[v] != -1 && dy[v] == dist) continue;
                if (cy[v] == -1 || dfs(cy[v])) {
                    cx[u] = v, cy[v] = u;
                    return 1;
                }
            }
        return 0;
    }

    int hopcroftKarp() {
        int matches = 0;
        memset(cx, -1, sizeof(cx));
        memset(cy, -1, sizeof(cy));
        while (bfs()) {
            memset(visited, 0, sizeof(visited));
            for (int i = 0; i < ty; i++)
                if (cx[i] == -1)
                    matches += dfs(i);
        }
        return matches;
    }
//-----10.6.3.cpp-----//

```

12668 Attacking Rooks^D (攻击的车)

在算法课程中，受国际象棋启发而得到的问题是练习的一个常见来源。从众所周知的 8 皇后问题开始，人们对一些国际象棋相关的问题进行了一般化和变形。其中之一就是 N 车问题，它指的是将国际象棋中的 N 个车放置在一个 $N \times N$ 的棋盘上，使得任意两个车之间不互相攻击。

Anand 教授将 N 车问题介绍给他的学生。因为车能够攻击位于同一行或者同一列的其他棋子，学生们

很快发现，只需将车放置在棋盘的主对角线上就能够轻易的解决 N 车问题。于是，教授决定在棋盘上放置一些兵，使得问题更为复杂。在放置的兵的棋盘上，两个车能够互相攻击，当且仅当它们位于同一行或者同一列且两者之间不存在兵。除此之外，兵所占据的棋盘位置给放置车增加了额外的限制。

给定棋盘的大小以及兵放置的位置，请告诉 Anand 教授在棋盘的空余位置上能够放置的车的最大数量，使得这些车互相之间无法攻击。

输入

输入包含多组测试数据，每组测试数据的格式描述如下。每组测试数据的第一行包含一个整数 N ($1 \leq N \leq 100$)，表示棋盘的行数和列数。接下来的 N 行，每行包含 N 个字符，第 i 行的第 j 列字符表示了棋盘的第 i 行第 j 列方格。给定的字符要么是 ‘.’ (点)，要么是大写字母 ‘X’，分别表示方格内是空的和放有一个兵。

输出

对于每组测试数据输出一行，包含一个整数，该整数表示能够放置的空余的棋盘方格上车的最大数量，这些车互相之间无法攻击。

样例输入

```
5
X....
X....
..X..
.X...
....X
4
.....
.X..
.....
.....
1
X
```

样例输出

```
7
5
0
```

分析

很明显，车只能放置在空的棋盘方格上，如果没有兵相隔，不能位于同一行或者同一列。以样例输入的第 1 组测试数据为例，将每一行被兵隔开且包含空方格的连续区域称做“块”，在一个块之中，最多只能放置一个车，将这些块予以编号，如图 10-21 (a) 所示。类似地，将每一列中被兵隔开且包含空方格的连续区域也进行分块并编号，如图 10-21 (b) 所示。

X	1	1	1	1
X	2	2	2	2
3	3	X	4	4
5	X	6	6	6
7	7	7	7	X

X	9	11	13	14
X	9	11	13	14
8	9	X	13	14
8	X	12	13	14
8	10	12	13	X

(a)

(b)

图 10-21 (a) 行中的空方格构成的块的编号。(b) 列中空方格构成的块的编号

将每行中的块视为二部图中顶点集合 X 中的顶点, 每列中的块视为二部图中顶点集合 Y 中的顶点, 若行和列中的两个块有公共的方格, 则在它们之间连边, 这样, 问题转化为一个二部图, 如图 10-22 所示。

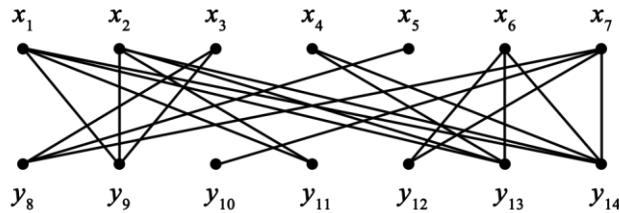


图 10-22 二部图

由于每条边表示一个空的方格, 即一行中的块和一列中的块之间的公共方格, 有冲突的空方格之间必有公共顶点, 问题转化为在二部图中寻找不存在公共顶点的最大边集, 即最大匹配问题。

那么应该如何具体构造二部图呢? 可以设置两个二维数组 xs 和 ys , 分别为行和列上的块进行编号, 然后建边, 详见参考代码。

参考代码

```

const int MAXV = 10100, INF = 0x7f7f7f7f7f;

// 由于图中的顶点可能较多, 使用邻接矩阵的方式在遍历时效率较低, 故使用链式前向星来记录边。
struct edge { int v, nxt; } g[100010];

int cnt, head[MAXV];
int xs[110][110], ys[110][110];
int visited[MAXV], cx[MAXV], cy[MAXV], dx[MAXV], dy[MAXV];
int tx, ty;

int bfs() {
    int dist = INF;
    memset(dx, -1, sizeof(dx));
    memset(dy, -1, sizeof(dy));
    queue<int> q;
    for (int i = 1; i <= tx; i++)
        if (cx[i] == -1) {
            q.push(i);
            dx[i] = 0;
            dist = 0;
        }
    while (!q.empty()) {
        int u = q.front();
        q.pop();
        for (int v = head[u]; v != -1; v = g[v].nxt) {
            int nv = g[v].v;
            if (dy[nv] == -1) {
                dy[nv] = dist + 1;
                if (nv == ty) {
                    return 1;
                } else {
                    q.push(nv);
                    cx[nv] = u;
                }
            }
        }
    }
    return 0;
}

```

```

        }
        while (!q.empty()) {
            int u = q.front(); q.pop();
            if (dx[u] > dist) break;
            for (int i = head[u]; ~i; i = g[i].nxt) {
                int v = g[i].v;
                if (dy[v] == -1) {
                    dy[v] = dx[u] + 1;
                    if (cy[v] == -1) dist = dy[v];
                    else {
                        dx[cy[v]] = dy[v] + 1;
                        q.push(cy[v]);
                    }
                }
            }
        }
        return dist != INF;
    }

    int dfs(int u) {
        int dist = INF;
        for (int i = head[u]; ~i; i = g[i].nxt) {
            int v = g[i].v;
            if (!visited[v] && dy[v] == (dx[u] + 1)) {
                visited[v] = 1;
                if (cy[v] != -1 && dy[v] == dist) continue;
                if (cy[v] == -1 || dfs(cy[v])) {
                    cx[u] = v, cy[v] = u;
                    return 1;
                }
            }
        }
        return 0;
    }

    int hopcroftKarp() {
        int matches = 0;
        memset(cx, -1, sizeof(cx));
        memset(cy, -1, sizeof(cy));
        while (bfs()) {
            memset(visited, 0, sizeof(visited));
            for (int i = 1; i <= tx; i++)
                if (cx[i] == -1)
                    matches += dfs(i);
        }
        return matches;
    }

    char board[110][110]; // 棋盘
    int main(int argc, char *argv[]) {
        int n;
        while (cin >> n) {
            for (int i = 0; i < n; i++)
                for (int j = 0; j < n; j++)
                    cin >> board[i][j];
            int counter = 0;
            memset(xs, 0, sizeof xs);

```

```

    memset(ys, 0, sizeof ys);
    for (int i = 0; i < n; i++) {
        // 块起始标记。
        int block = 0;
        for (int j = 0; j < n; j++) {
            if (board[i][j] == '.') {
                if (!block) counter++;
                xs[i][j] = counter;
                block = 1;
            } else block = 0;
        }
    }
    // X 侧顶点的数目。
    tx = counter;
    for (int j = 0; j < n; j++) {
        int block = 0;
        for (int i = 0; i < n; i++) {
            if (board[i][j] == '.') {
                if (!block) counter++;
                ys[i][j] = counter;
                block = 1;
            } else block = 0;
        }
    }
    // 构建二部图中的边。
    cnt = 0;
    memset(head, -1, sizeof head);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            if (xs[i][j]) {
                g[cnt] = edge{ys[i][j], head[xs[i][j]]};
                head[xs[i][j]] = cnt++;
            }
    cout << hopcroftKarp() << '\n';
}
return 0;
}

```

强化练习：[11138 Nuts and Bolts](#)^C。

扩展练习：[1663* Purifying Machine](#)^E，[10122* Mysterious Mountain](#)^D。

10.6.4 Gale-Shapley 算法

稳定匹配 (stable matching) 问题，又称稳定婚姻 (stable marriage) 问题，由 Gale 和 Shapley 于 1962 年首先提出^[125]，可以将其表述如下：有 n 位男士和 n 位女士，在不允许出现偏好程度相同的情况下，每位男士按照其对每位女士作为配偶的偏爱程度给所有女士排序，类似的，每位女士也按照偏爱程度给所有男士排序。容易知道，这些男士和女士构成完备婚姻匹配的方式有 $n!$ 种。如果某个完备婚姻匹配中存在两位男士 A, B 和两位女士 a, b ，满足下列条件：

- (1) A 和 a 结婚， B 和 b 结婚；
- (2) A 更偏爱 b 而非 a ， B 更偏爱 a 而非 b 。

则称该完备婚姻匹配是不稳定的 (unstable)，否则称之为稳定的 (stable)，人们将求解稳定完备婚姻匹配问题称为稳定婚姻问题。

Gale 和 Shapley 证明, 稳定婚姻问题一定存在解, 可以通过 Gale-Shapley 算法(又称“延迟认可算法”)在多项式时间内得到一组解。

Gale-Shapley 算法

- (1) 初始化所有男士和女士均为单身的;
- (2) 当存在单身男士时:
 - (2.1) 选择一位单身男士 m ;
 - (2.2) 令 w 为 m 的偏好列表中排名最高且 m 尚未向其求婚的女士;
 - (2.3) 如果 w 为单身的, m 和 w 订婚;
 - (2.4) 如果 w 已经和其他男士 m' 订婚, 则检查 w 对 m' 的偏好程度是否大于 m , 如果是, 则 m 仍旧是单身的, 否则 m 和 w 订婚, 此时 m' 将成为单身的。

在 Gale-Shapley 算法的具体实现中, 通过男士去选择女士所得到的解是男士最优的, 反之则是女士最优的。

```
//-----10.6.4.cpp-----//
const int MAXN = 1010;

// 男士和女士的数量。
int n;
// mList 记录男士偏好列表, wList 记录女士偏好列表。
int mList[MAXN][MAXN], wList[MAXN][MAXN];
// mPrefer[i][j] 记录男士 i 对女士 j 的偏好程度。
// wPrefer[i][j] 记录女士 i 对男士 j 的偏好程度。
int mPrefer[MAXN][MAXN], wPrefer[MAXN][MAXN];
// mLast 记录男士向偏好列表中的女士求婚时最后一位被求婚女士在偏好列表中的序号。
// wLast 记录女士向偏好列表中的男士求婚时最后一位被求婚男士在偏好列表中的序号。
int mLast[MAXN], wLast[MAXN];
// mMatched 记录与男士订婚的女士的序号。
// wMatched 记录与女士订婚的男士的序号。
int mMatched[MAXN], wMatched[MAXN];
// 记录男士是否在单身队列中。
int in[MAXN];

// 通过男士来选择女士的 Gale-Shapley 算法。
void galeShapley() {
    // 初始化所有男士为单身。
    for (int i = 1; i <= n; i++) mLast[i] = wMatched[i] = in[i] = 0;
    queue<int> q;
    for (int i = 1; i <= n; i++) {
        q.push(i);
        in[i] = 1;
    }
    while (!q.empty()) {
        int m = q.front(); q.pop();
        in[m] = 0;
        // 从男士 m 上一次求婚的女士之后的一位女士开始, 继续向其他尚未求婚的女士求婚。
        for (mLast[m]++; mLast[m] <= n; mLast[m]++) {
            int w = mList[m][mLast[m]];
            int mm = wMatched[w];
            // 检查该女士是否已经订婚, 如果已经订婚则检查男士 m 是否更佳。
            if (!mm || (wPrefer[w][m] > wPrefer[w][mm])) {

```

```

        if (mm && !in[mm]) {
            q.push(mm);
            in[mm] = 1;
        }
        mMatched[m] = w;
        wMatched[w] = m;
        break;
    }
}
//-----10.6.4.cpp-----//

```

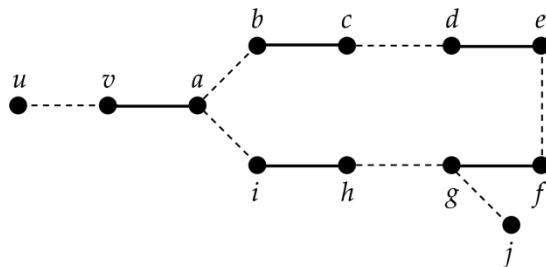
上述实现能够以 $O(n^2)$ 的时间复杂度得到稳定婚姻问题的解。从算法的执行过程可以容易看出，每对一位单身男士进行一次匹配操作，会使得一位女士得到匹配，不会出现某位单身男士已经向所有女士求婚但均被拒绝的情况^I，可以证明，算法最终得到的匹配是稳定的。

强化练习：1175 Ladies' Choice。

10.6.5 Edmonds 算法

对于一般图的最大匹配，起初人们认为这是一个 NP 问题，不存在有效算法。1963 年，Edmonds 首次提出了一个多项式算法，称之为 Edmonds 算法，可以用于解决一般图的最大匹配问题^[117]。Edmonds 算法也是基于增广路定理，即不断地在图中寻找增广路径，直到不存在增广路径。与二部图不同，在一般图中寻找增广路径的过程中，可能会出现“奇圈”，而在二部图中不存在奇圈（可能存在偶圈，但偶圈不影响二部图的划分），正是这个原因给一般图的最大匹配问题带来了难度。

在一般图上应用增广路算法寻找增广路径，如果经过长度为奇数的交错路到达的顶点在之前的增广路搜索中已经被长度为偶数的交错路到达过，那么就发现了一个奇圈，这两条交错路的“并”就称为花（flower），这个奇圈就称为花朵（blossom），两条交错路的最长公共子路称为茎（stem），起点称为根（root），终点称为蒂（tip）或基（base）。



^I 可以通过反证法予以证明这种情形不会出现。假设在算法执行的某个时刻出现了某个单身男士 m ， m 已经向所有女士求婚但均未被接受。对于某位女士 w 来说，一旦 w 接受某个男士的求婚后，她后续的配偶只会越来越好，也就是说女士 w 会一直保持已订婚的状态。“ m 已经向所有女士求婚但均未被接受”意味着所有女士已经订婚，而每位女士只与一位男士订婚，因此所有男士应该都已经订婚，不会出现单身男士的情形，出现矛盾。

图 10-23 从顶点 u 开始寻找增广路。交错路 u, v, a 的长度为 2, 是偶数, 而交错路 $u, v, a, b, c, d, e, f, g, h, i, a$ 的长度为 11, 是奇数, 而且到达了已经经过的顶点 a , 称交错路 $u, v, a, b, c, d, e, f, g, h, i, a$ 为花, 奇圈 $a, b, c, d, e, f, g, h, i$ 为花朵, 路 u, v, a 称为茎, 顶点 u 为根, 顶点 a 为基

由于奇圈的外形与花朵的外形相似, 故而 Edmonds 将之称为“花朵”, 将“花朵”收缩, 使之成为一个超级顶点, 在此新图上进行增广, 可以证明, “缩点”后的新图和原图具有相同的增广路径。每一条增广路径都可以通过把“花朵”展开还原回去, 因为一个奇圈的两段路径必然是一奇一偶, 总能找到一段满足要求的增广路^[126]。

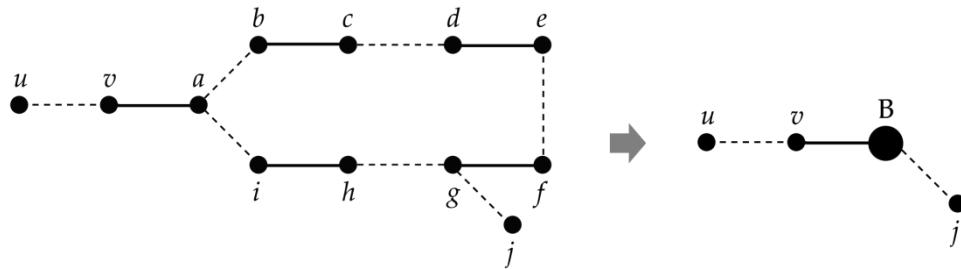


图 10-24 将奇圈 $a, b, c, d, e, f, g, h, i$ 收缩为超级顶点 B

在增广过程中, 一旦发现奇圈, 可以将其收缩为一个顶点, 再继续搜索。如果新图中的增广路经过收缩后的顶点, 可以将“花朵”还原, 之后利用奇圈中两条交错路之一将其还原为原图中的增广路, 从而实现增广——这是 Edmonds 算法的精髓所在。

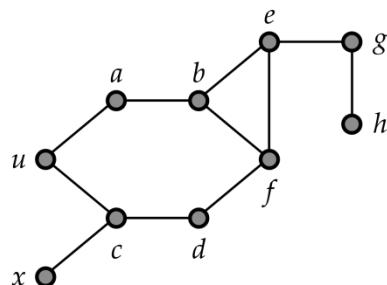
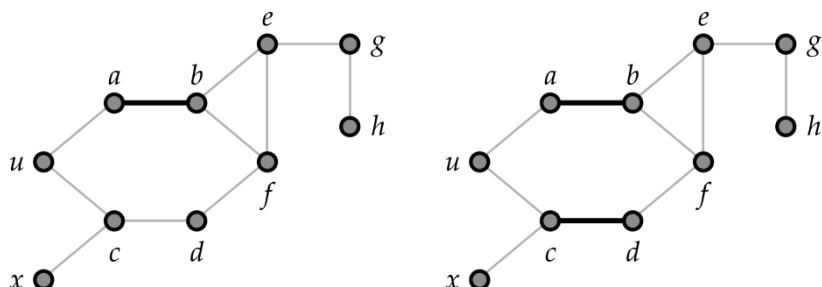
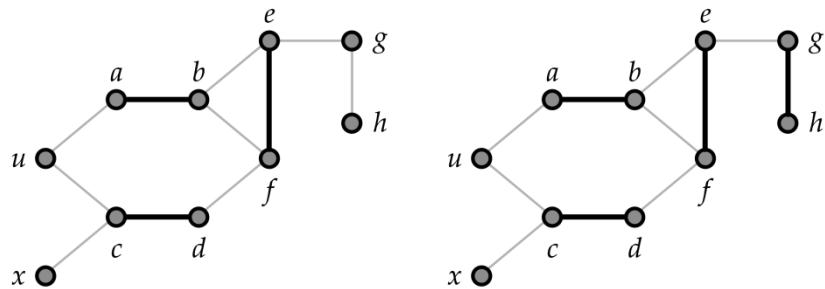


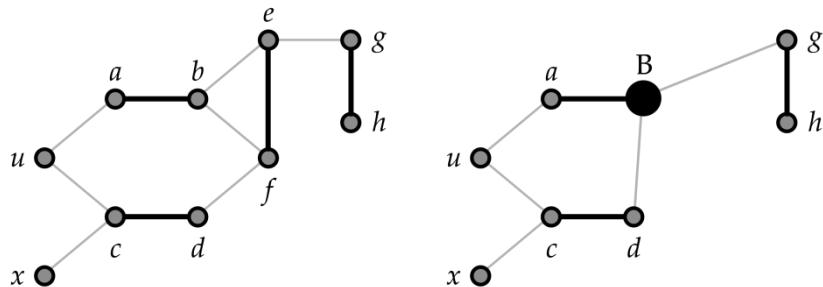
图 10-25 具有 10 个顶点和 11 条边的图, 包含偶圈 (u, a, b, f, d, c) 和奇圈 (b, e, f)



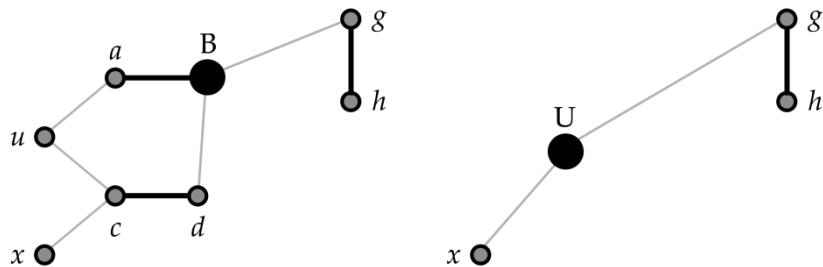
算法执行的第 1 步至第 2 步。第 1 步: 从未饱和的 a 开始搜索, 找到增广路 ab 。第 2 步: 从未饱和的 c 开始搜索, 找到增广路 cd



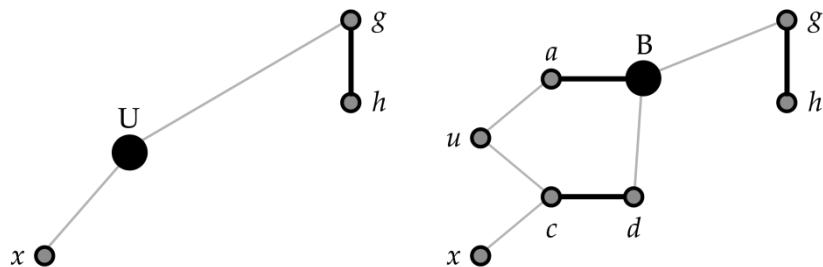
算法执行的第3步至第4步。第3步：从未饱和的 e 开始搜索，找到增广路 ef 。第4步：从未饱和的 g 开始搜索，找到增广路 gh



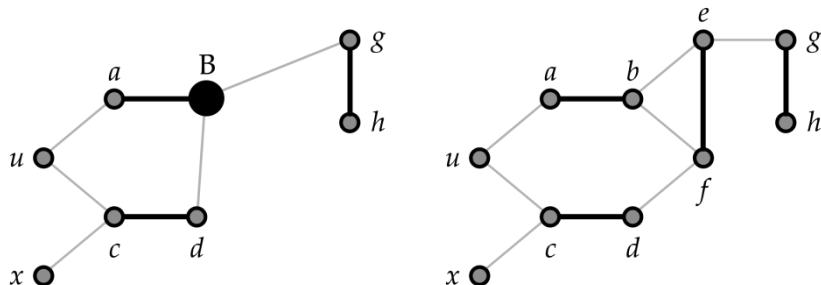
算法执行第5步：从未饱和的 u 开始搜索，找到交错路 $uabefb$ ，此时长为 5 的交错路 $uabefb$ 到达了之前长为 2 的交错路 ab 到达过的顶点 b ，发现花，其中花朵为 bef ，茎为 uab ，基为 b 。收缩 bef 为 B ，继续搜索



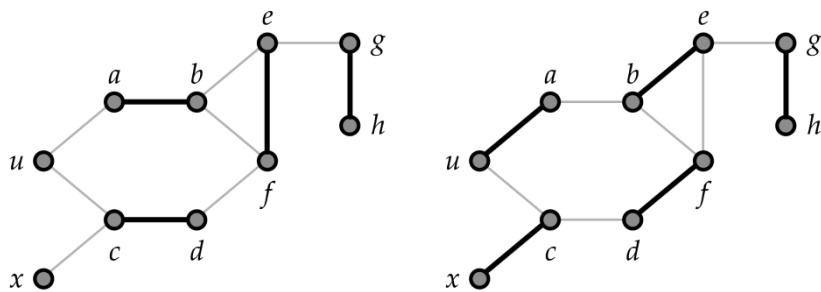
算法执行第6步：从未饱和的 u 开始搜索，找到交错路 $uaBdcu$ ，此时长为 5 的交错路 $uaBdcu$ 到达了之前长为 0 的交错路到达过的顶点 u ，发现花，其中花朵为 $uaBdc$ ，茎为空，基为 u 。收缩 $uaBdc$ 为 U ，继续搜索



算法执行第 7 步：找到增广路 Ux ，涉及收缩顶点 U ，将 U 还原，从关联到 x 的不在当前匹配中的边 cx 起，沿交错路 $uaBdcx$ 退回到基 u



算法执行第 8 步：找到增广路 $uaBdcx$ ，涉及收缩顶点 B ，将 B 还原，从关联到 d 的不在当前匹配中的边 fd 起，沿交错路 bef 退回到基 b



算法执行第 9 步：找到增广路 $uabefd cx$ ，不涉及收缩顶点，将增广路上的边取反，替换进当前匹配，则当前匹配增加 1。由于所有顶点已经匹配，算法结束

图 10-26 Edmonds 算法的执行过程

虽然 Edmonds 算法的思想并不复杂，但是实现起来却有一定难度，因为要考虑“缩点”成“花朵”、“花朵”的展开、寻找最近公共祖先、增广等一系列操作，若想简洁、精巧地实现此算法，必须对算法的核心思想以及实现步骤有非常清晰和深刻的理解。在具体实现时，需要考虑以下细节问题：

(1) 如何区分偶圈和奇圈。可以使用为图进行二着色所使用的方法，在 BFS (或 DFS) 过程中，为顶点交替着色，如果出现已经访问的两个顶点着色相同则表明找到了一个奇圈；

(2) 奇圈的表示。可以考虑使用并查集来表示奇圈，或者使用奇圈的基作为代表来表示奇圈，如果使用后者，需要查找奇圈中两个顶点公共祖先的操作。

Edmonds 算法，如果使用朴素的方法予以实现，其时间复杂度为 $O(n^4)$ ，如果能够高效地处理花的收缩和展开，可以将时间复杂度降低到 $O(n^3)$ 。以下给出 Edmonds 算法的一种参考实现，读者可以结合注释进行理解和运用。除此之外，Gabow 提出了另外一种高效实现 Edmonds 算法的方法^[127]，其时间复杂度亦为 $O(n^3)$ 。

```
//-----10.6.5.cpp-----//
const int MAXN = 210, UNVISITED = -1, WHITE = 0, BLACK = 1;

// n 为图中顶点数目，m 为图中边的数目。
```

```

int n, m;
// 使用边列表表示图。
vector<int> g[MAXN];
// parent 记录顶点的前驱;
// linked 记录顶点匹配;
// base 记录奇圈中各个顶点的代表;
// color 记录顶点的访问状态。
int parent[MAXN], linked[MAXN], base[MAXN], color[MAXN];

void initialize() { for (int i = 0; i <= n; i++) g[i].clear(); }
void addEdge(int u, int v) { g[u].push_back(v), g[v].push_back(u); }

// 查找顶点 u 和 v 的最近公共祖先。
int lca(int u, int v) {
    static int token = 0, key[MAXN] = {};
    for (++token; ; swap(u, v)) {
        if (u == 0) continue;
        if (key[u] == token) return u;
        key[u] = token;
        u = base[parent[linked[u]]];
    }
}

// 将奇圈缩成“花朵”。
void shrink(int u, int v, int p, queue<int> &q) {
    while (base[u] != p) {
        parent[u] = v, v = linked[u];
        if (color[v] == BLACK) q.push(v), color[v] = WHITE;
        base[u] = base[v] = p, u = parent[v];
    }
}

// 使用 BFS 寻找增广路。
int bfs(int u) {
    for (int i = 0; i <= n; i++) base[i] = i;
    memset(color, UNVISITED, sizeof(color));
    queue<int> q;
    q.push(u); color[u] = WHITE;
    while (!q.empty()) {
        u = q.front(); q.pop();
        for (auto v : g[u]) {
            if (color[v] == UNVISITED) {
                parent[v] = u, color[v] = BLACK;
                if (!linked[v]) {
                    for (int prev; u; v = prev, u = parent[v]) {
                        prev = linked[u];
                        linked[u] = v, linked[v] = u;
                    }
                    return 1;
                }
                q.push(linked[v]); color[linked[v]] = WHITE;
            } else if (color[v] == WHITE && base[v] != base[u]) {
                int p = lca(u, v);
                shrink(v, u, p, q);
                shrink(u, v, p, q);
            }
        }
    }
}

```

```

    }
}

return 0;
}

// 使用 Edmonds 算法求一般图最大匹配。
int edmonds() {
    memset(parent, 0, sizeof(parent));
    memset(linked, 0, sizeof(linked));
    int matches = 0;
    for (int i = 1; i <= n; i++) {
        if (!linked[i] && bfs(i))
            matches++;
    }
    return matches;
}

int main(int argc, char *argv[]) {
    while (cin >> n >> m) {
        // 初始化, 读入图数据。
        initialize();
        for (int i = 1, u, v; i <= m; i++) {
            cin >> u >> v;
            addEdge(u, v);
        }
        cout << edmonds() << '\n';
    }
    return 0;
}
//-----10.6.5.cpp-----//

```

除了使用 Edmonds 算法求一般图最大匹配外, 还存在其他的有效算法来解决这个问题, 例如通过高斯消元法来求解一般图的最大匹配问题^[128]^[129]。

扩展练习: [11439* Maximizing the ICPC^D](#)。

10.7 二部图加权完备匹配

如果在二部图中, X 侧的所有顶点均有对应的匹配且 Y 侧的所有顶点也有相应的匹配, 则称该匹配为完备匹配 (perfect matching, 又称完美匹配)。如果为二部图中的边赋予权值, 要求在获得最大匹配的前提下, 所得匹配的边权之和最大 (或最小), 称为二部图加权完备匹配问题。二部图加权完备匹配分最大权完备匹配 (maximum weight perfect matching) 和最小权完备匹配 (minimum weight perfect matching)。对于此问题, 可以尝试使用以下两种方式进行解决¹。

10.7.1 网络流解法

可以将加权完备匹配问题建模为最小费用最大流问题予以解决。其步骤一般为:

- (1) 划分二部图;
- (2) 根据题目的约束条件构建加权容量网络;

¹ 对于本小节给出的每一道习题, 建议读者使用网络流解法和 Kuhn-Munkres 算法分别解决一次以提高解题能力。

(3) 虚拟一个源点 $source$, 一个汇点 $sink$, 从源点 $source$ 向二部图的 X 侧顶点各引一条容量为 1, 费用为 0 的有向弧, 从二部图的 Y 侧顶点向汇点 $sink$ 各引一条容量为 1, 费用为 0 的有向弧;

(4) 在构建得到的加权容量网络上使用 SPFA 不断寻找增广路, 即从源点 $source$ 出发, 能够到达 $sink$ 且具有最短距离的路径, 更新容量;

(5) 无法找到增广路时停止, 此时根据增广路定理, 得到的最大流具有最小费用。

如果所求为最大权完备匹配, 则将初始权值设置为其对应的负值, 求最小费用最大流, 最后将结果取反即可。

强化练习: [10746* Crime Wave The Sequel^C](#), [10888 Warehouse^D](#)。

扩展练习: [1006* Fixed Partition Memory Management^D](#)。

10.7.2 Kuhn-Munkres 算法

除了将加权完备匹配问题转化为网络流解决以外, 也可以使用针对加权完备匹配问题而提出的有效算法——Kuhn-Munkres 算法^[130], 由于该算法经由 Kuhn 和 Munkres 分别独立提出, 故而得名。

为了确定二分图的最大权完备匹配, Kuhn-Munkres 算法引入了顶标 (vertex labelling) 的概念, 顶标指的是为每个顶点所赋予的一个标号值。在顶标的基础上, 可以引出可行顶标 (feasible vertex labelling) 和相等子图 (equality subgraph) 的概念, 从而将求二部图的最大权匹配问题转化为求相等子图的完全匹配问题。令顶点 x_i 的顶标为 $lx[i]$, 顶点 y_j 的顶标为 $ly[j]$, 顶点 x_i 与 y_j 之间的边权为 $weight[i][j]$, 如果对于任一条边 (i, j) , $lx[i] + ly[j] \geq weight[i][j]$ 均成立, 则将该顶标赋值方案称为可行顶标。相等子图由原图构建而来, 它包括原图的所有顶点, 但可能未包括原图中的所有边。相等子图中只包含原图中顶标满足 $lx[i] + ly[j] = weight[i][j]$ 的边 (i, j) , 这些边称为可行边 (feasible edge)。可以证明, 若由二部图中所有满足 $lx[i] + ly[j] = weight[i][j]$ 的边 (i, j) 构成的相等子图具有完备匹配, 那么这个完备匹配就是二部图的最大权匹配。Kuhn-Munkres 算法的正确性即基于以上结论。可以这样来理解此结论: 因为对于二部图的任意一个匹配, 如果它包含于相等子图, 那么它的边权和等于所有顶点的顶标和; 如果它有的边不包含于相等子图, 那么它的边权和小于所有顶点的顶标和, 所以相等子图的完备匹配一定是二部图的最大权匹配。

为了保证 Kuhn-Munkres 算法的正确性, 需要始终保持顶标是可行顶标, 即对于相等子图中的任意边, 均有 $lx[i] + ly[j] = weight[i][j]$ 成立。初始时为了使 $lx[i] + ly[j] \geq weight[i][j]$ 对所有边均成立, 令 $lx[i]$ 为所有与顶点 x_i 关联的边的最大权, 同时令 $ly[j] = 0$, 此时的可行顶标称为平凡顶标 (trivial labelling)。如果当前的相等子图没有完备匹配, 就按下面的方法修改顶标以使得相等子图扩大, 直到相等子图具有完备匹配为止。当前相等子图的完备匹配未能成功, 其原因在于对于某个 X 侧顶点 x_u , 无法找到一条从它出发的增广路, 而在此时根据匹配算法所获得的是一条交错路, 但该交错路的第一条边是未匹配边, 最后一条边是匹配边, 是一条“伪增广路”, 为了将该条“伪增广路”扩展成为增广路, 现在把交错路中 X 侧顶点的顶标全都减少某个值 d , Y 侧顶点的顶标全部增加同一个值 d , 如果将二部图中的边位于 X 侧的顶点称为 x 端, 位于 Y 侧的顶点称为 y 端, 那么就会发现:

(1) x 端和 y 端都在交错路中的边 (x_i, y_j) , 其“顶标和”在修改后为 $(lx[x_i] - d) + (ly[y_j] + d) = lx[x_i] + ly[y_j]$, 即“顶标和”不会发生变化, 也就是说, 它原来属于相等子图, 现在仍然属于相等子图;

(2) x 端和 y 端都不在交错路中的边 (x_i, y_j) , 其顶标 $lx[x_i]$ 与 $lx[y_j]$ 都未发生变化, 也就是说, 它原来属于 (或不属于) 相等子图, 现在仍然属于 (或不属于) 相等子图;

(3) x 端不在交错路中, y 端在交错路中的边 (x_i, y_j) , 它的“顶标和” $lx[x_i] + ly[y_j] + d$ 有所增大, 它原来不属于相等子图, 现在仍然不属于相等子图;

(4) x 端在交错路中, y 端不在交错路中的边(x_i, y_j), 它的“顶标和” $lx[x_i] - d + ly[y_j]$ 有所减少, 它原来不属于相等子图, 现在可能进入了相等子图, 因而可能使得相等子图得到了扩大;

在调整顶标后, 至少会有一条边进入相等子图, 此时再次寻找完备匹配, 如果 X 侧每个顶点至少有一条匹配边, Y 侧每个顶点至少有一条匹配边, 说明最后补充完毕的相等子图存在完备匹配, 根据前述的结论, 这个完备匹配就是该二部图的最大权匹配。那么如何确定 d 值呢? 为了使得 $lx[x_i] + ly[y_j] \geq weight[x_i][y_j]$ 始终成立, 且至少有一条边进入相等子图, 应该使得^I

$$d = \min\{lx[x_i] + ly[y_j] - weight[x_i][y_j], x_i \text{ 在交错路中, } y_j \text{ 不在交错路中}\}$$

根据上述讨论, 可以将 Kuhn-Munkres 算法的基本思路概况为以下四个步骤:

- (1) 初始化所有顶点可行顶标的值;
- (2) 使用 Hungarian 算法寻找完备匹配;
- (3) 若未找到完备匹配则修改可行顶标的值;

(4) 重复 (2) 和 (3) 步骤直到找到相等子图的完备匹配为止, 根据前述给出的结论, 此时相等子图的完备匹配即为原图的最大权完备匹配, 最大权完备匹配的权值为相等子图中各顶点的顶标之和。

如果使用朴素的方法来实现 Kuhn-Munkres 算法, 其时间复杂度为 $O(n^4)$ 。Kuhn-Munkres 算法需要寻找 $O(n)$ 次增广路, 每次增广最多需要修改 $O(n)$ 次顶标, 每次修改顶标需要枚举所有边来求 d 值——这部分的时间复杂度为 $O(n^2)$, 因此总的时间复杂度为 $O(n^4)$ 。通过应用一种优化技巧, 可以将 Kuhn-Munkres 算法的时间复杂度降到 $O(n^3)$ 。具体做法是为每个 Y 侧顶点赋予一个“松弛量”—— $slack[y]$, 每次开始寻找增广路时将其初始化为无穷大, 在寻找增广路的过程中, 当检查边(x_i, y_j)时, 如果它不在相等子图中, 则将 $slack[y]$ 更新为原值与 $lx[x_i] + ly[y_j] - weight[x_i][y_j]$ 两个值中的较小值。在修改顶标时, 取所有不在交错路中的 Y 侧顶点的 $slack[y]$ 值中的最小值作为 d 值即可。

```

//-----10.7.2.cpp-----
const int MAXN = 110, INF = 0x7f7f7f7f;

// n 表示二部图中 X 侧顶点的数量。
int n;
// weight 记录边权值, linky 记录与 Y 侧顶点匹配的 X 侧顶点。
int weight[MAXN][MAXN], linky[MAXN];
// cx 记录 X 侧顶点是否已经匹配, cy 记录 Y 侧顶点是否已经匹配, slack 为松弛量。
int cx[MAXN], cy[MAXN], slack[MAXN];
// lx 记录 X 侧顶点的顶标, ly 记录 Y 侧顶点的顶标。
int lx[MAXN], ly[MAXN];

bool dfs(int x) {
    cx[x] = true;
    for (int y = 0; y < n; y++) {
        if (cy[y]) continue;
        int delta = lx[x] + ly[y] - weight[x][y];
        // 如果顶标之和与边权相等, 表明此边为可行边, 可以进入相等子图用于寻找增广路。
        if (delta == 0) {
            cy[y] = true;
            // 条件: 顶点 y 尚未匹配或者从“已与顶点 y 匹配的顶点”开始能够寻找到增广路。
        }
    }
}
```

^I 此处利用 d 值修改顶标使得相等子图扩大的技巧与“松弛”技巧有异曲同工之妙。

```

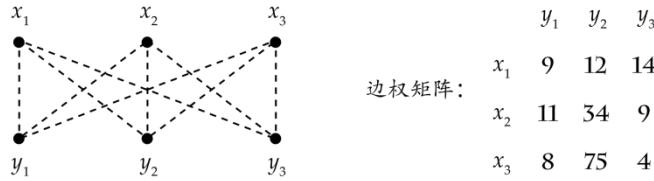
        if (linky[y] == -1 || dfs(linky[y])) {
            linky[y] = x;
            return true;
        }
    }
    // 更新松弛量。
    else slack[y] = min(slack[y], delta);
}
return false;
}

int kuhnMunkres() {
    // 确定初始顶标。
    for (int i = 0; i < n; i++) {
        linky[i] = -1, lx[i] = 0, ly[i] = 0;
        for (int j = 0; j < n; j++)
            lx[i] = max(lx[i], weight[i][j]);
    }
    for (int x = 0; x < n; x++) {
        while (true) {
            memset(cx, 0, sizeof(cx));
            memset(cy, 0, sizeof(cy));
            // 每次匹配前均需要初始化松弛量。
            for (int i = 0; i < n; i++) slack[i] = INF;
            if (dfs(x)) break;
            int delta = INF;
            // 根据松弛量确定顶标的最小变化量 d。
            for (int i = 0; i < n; i++)
                if (!cy[i])
                    delta = min(delta, slack[i]);
            // 调整顶标，扩大相等子图。
            for (int i = 0; i < n; i++) {
                if (cx[i]) lx[i] -= delta;
                if (cy[i]) ly[i] += delta;
            }
        }
    }
    // 统计完备匹配的权值。
    int r = 0;
    for (int y = 0; y < n; y++)
        if (~linky[y])
            r += weight[linky[y]][y];
    return r;
}
//-----10.7.2.cpp-----/

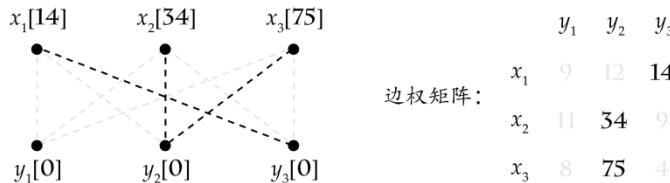
```

Kuhn-Munkres 算法适用于求完全二部图的最大权匹配，如果给定的二部图不是完全二部图，即二部图中 X 侧顶点和 Y 侧顶点数量有差异，可以采用如下技巧将顶点予以“补齐”：假设当前 X 侧顶点数量小于 Y 侧顶点数量，可以在 X 侧“虚拟”若干顶点，使得两侧的顶点数目相同，同时在 X 侧虚拟的顶点和 Y 侧顶点间建立边权值为 0 的边，使得二部图成为完全二部图，然后执行 Kuhn-Munkres 算法求最大匹配即可。

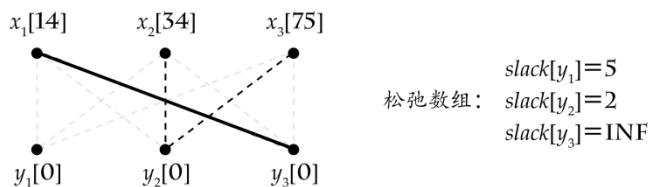
以下是 Kuhn-Munkres 算法的参考实现代码在给定图上的具体执行过程。通过观察该执行过程，读者可以进一步增进对 Kuhn-Munkres 算法的理解。



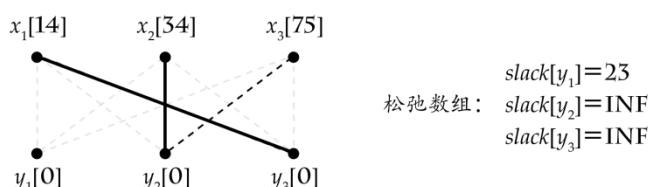
(1) 给定包含 6 个顶点的完全二部图及其边权矩阵, X 侧顶点为 x_1, x_2, x_3 , Y 侧顶点为 y_1, y_2, y_3



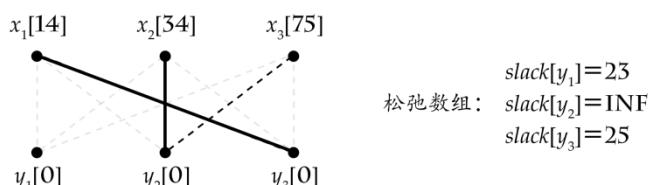
(2) 由边权矩阵获得初始的可行顶标和相等子图, 顶点标记右侧方括号内的数字为顶点的顶标。初始时, 相等子图只包括 3 条可行边: $x_1-y_3, x_2-y_2, x_3-y_3$



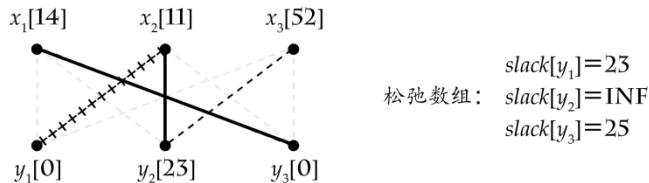
(3) 从 X 侧的顶点 x_1 开始进行匹配。 x_1 依次与 y_1, y_2, y_3 进行匹配。在相等子图中, x_1 与 y_1 不存在边, $slack[y_1] = lx[x_1] + ly[y_1] - weight[x_1][y_1] = 14 + 0 - 9 = 5$ 。 x_1 与 y_2 不存在边, $slack[y_2] = lx[x_1] + ly[y_2] - weight[x_1][y_2] = 14 + 0 - 12 = 2$ 。 x_1 与 y_3 存在边, 且 x_1 和 y_3 均未被匹配, 故可将 x_1 与 y_3 进行匹配, 由于 x_1 与 y_3 已经匹配, $slack[y_3]$ 未计算, 其值为默认值 INF



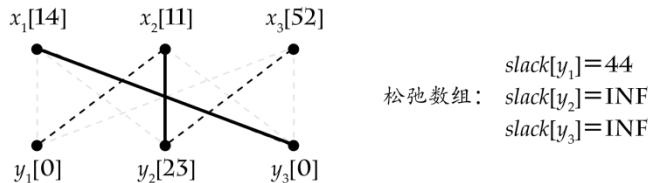
(4) 从 X 侧的顶点 x_2 开始进行匹配。 x_2 依次与 y_1, y_2, y_3 进行匹配。在相等子图中, x_2 与 y_1 不存在边, $slack[y_1] = lx[x_2] + ly[y_1] - weight[x_2][y_1] = 34 + 0 - 11 = 23$ 。 x_2 与 y_2 存在边且均未被匹配, 故可将 x_2 与 y_2 匹配, 由于 x_2 与 y_2 已经匹配, 后续的 $slack[y_2]$ 、 $slack[y_3]$ 未计算, 其值为默认值 INF



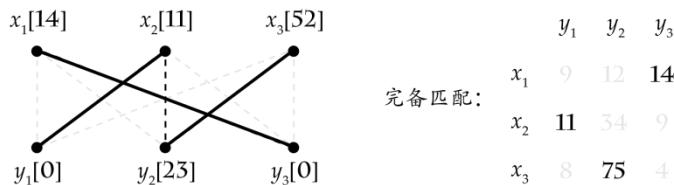
(5) 从 X 侧的顶点 x_3 开始进行匹配。 x_3 依次与 y_1, y_2, y_3 进行匹配。在相等子图中, x_3 与 y_1 不存在边, $slack[y_1] = lx[x_3] + ly[y_1] - weight[x_3][y_1] = 75 + 0 - 8 = 67$ 。 x_3 与 y_2 存在边, 但 y_2 已经与 x_2 匹配, x_2 与其他 Y 侧顶点无法匹配, 最终得到一条交错路: $x_3 - y_2 - x_2$, 但该交错路不是增广路, 故无法从 x_3 出发构建增广路, 需要扩大相等子图。在使用 Hungarian 算法寻找匹配的过程中, 从 x_3 出发, 经过 y_2 , 再到达 x_2 , 由于 x_3 能够与 y_2 匹配, 故 $slack[y_2]$ 未计算, 其值为初始值 INF。Hungarian 算法使用递归寻找增广路, 在此过程中, 从 x_2 出发又再次与 y_1, y_2, y_3 进行匹配, 由于 x_2 与 y_2 已经匹配, 故 $slack[y_2]$ 仍未计算, 其值仍为初始值 INF, 而 $slack[y_1] = \min\{67, lx[x_2] + ly[y_1] - weight[x_2][y_1]\} = \min\{67, 34 + 0 - 11\} = 23$, $slack[y_3] = lx[x_2] + ly[y_3] - weight[x_2][y_3] = 34 + 0 - 9 = 25$



(6) 根据松弛数组, 易知 $d = slack[y_1] = 23$, 将交错路 $x_3 - y_2 - x_2$ 中的 X 侧顶点的顶标减少 d , Y 侧顶点的顶标增加 d , 修改顶标后, 边 $x_2 - y_1$ 进入相等子图, 相等子图得到扩大



(7) 从 X 侧的顶点 x_3 开始再次进行匹配。 x_3 依次与 y_1, y_2, y_3 进行匹配。当前相等子图中, x_3 与 y_1 不存在边, $slack[y_1] = lx[x_3] + ly[y_1] - weight[x_3][y_1] = 52 + 0 - 8 = 44$; x_3 与 y_2 存在边, 但 y_2 已经与 x_2 匹配, 而 x_2 可与 y_1 匹配, 最终可以得到一条交错路: $x_3 - y_2 - x_2 - y_1$, 该交错路为增广路, 可以进行增广。由于 x_3 与 y_2 已经匹配, x_2 与 y_1 又已经匹配, 后续的 $slack[y_2], slack[y_3]$ 未计算, 其值为默认值 INF



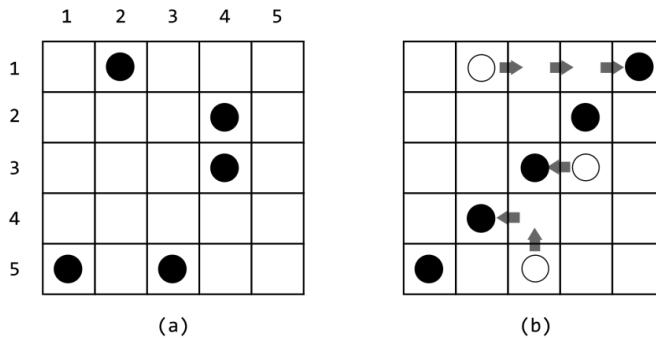
(8) 对增广路 $x_3 - y_2 - x_2 - y_1$ “取反” 以扩大匹配。增广完成后, X 侧和 Y 侧顶点均已匹配: x_1 匹配 y_3 , x_2 匹配 y_1 , x_3 匹配 y_2 。易知, 最大权完备匹配的权值和为 100

图 10-27 Kuhn-Munkres 算法的执行过程

在有关 Kuhn-Munkres 算法应用的题目中, 由于算法本身是“固定不变”的, 只要知道了问题的底层模型是最大(小)权完备匹配, 则构建二部图后应用算法模板即可顺利解决, 因此解题难点在于分析题意并将其建模为完全二部图这个环节。

1045 The Great Wall Game^D (长城游戏)

Hua 和 Shen 自创了一种简单的单人棋盘游戏，他们称之为“长城游戏”。游戏使用 n 颗石子，在一个 $n \times n$ 的网格上进行。石子随机放置在网格的方格中，每个方格最多放置一颗石子。在一步移动中，任意单个石子可以沿着水平或者垂直方向移动一个方格，到达一个尚未被其他石子占据的方格。游戏的目标是创建一堵“墙”——使用最少的步骤数将 n 颗石子排成一条直线，使得这 n 颗石子位于同一行或同一列或同一条对角线。如图 (a) 所示，这是 $n=5$ 时的一个示例。图 (b) 展示了使用 6 步移动将所有石子排列在同一条对角线上的一种方案。不存在少于 6 步移动的方案能够将图 (a) 给定的布局排列成一条直线（不过存在其他的方案，它们也可以通过 6 步移动将 5 颗石子排列成一条直线）。



这里只有一个问题——对于给定的初始布局，Hua 和 Shen 不知道将其排列成一条直线所需的最少移动步数。他们期望你能够编写一个程序，能够将任意一个初始布局作为输入，确定创建一堵墙所需要的最少移动步数。

输入

输入包含多组测试数据，每组测试数据的第一行为一个整数 n ， $1 \leq n \leq 15$ 。接下来的一行包含 n 颗石子的位置，第一和第二个整数表示第一颗石子所在的行和列，第三和第四个整数表示第二颗石子所在的行和列，如此类推。行和列的计数方式如图 (a) 所示。最后一组测试数据后面包含一行，该行只有一个 0，表示输入结束。

输出

对于每组测试数据，输出测试数据的组数 (1, 2, ...), 接着是将 n 颗石子排成一条直线所需的最少步数。按照样例输出的格式输出，在每组测试数据的输出后面打印一个空行。

样例输入

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

样例输出

```
Board 1: 6 moves required.
Board 2: 0 moves required.
Board 3: 1 moves required.
```

分析

以样例输入的第一组测试数据为例，假设将所有石子排列在副对角线上构成一条直线，那么对于某颗石子 s_i 来说，它可以移动到副对角线的任意一个位置，而且每移动到副对角线上的一个特定位置都有着相应的最少移动步数，于是可以将石子视为二部图的 X 侧顶点，副对角线上的每个方格视为二部图的 Y 侧顶点，

单个石子移动到副对角线上的某个特定方格所需的最少移动步数就是顶点间边权的值，那么问题转化为求该完全二部图的最小权完备匹配。

前面介绍了使用 Kuhn-Munkres 算法求二部图最大权完备匹配的方法，那么如何求最小权完备匹配呢？有两种方法：

第一种方法是将边权值“取反”，即将边权值转换成对应的负值，仍然使用前述的求最大权完备匹配的 Kuhn-Munkres 算法，只不过在最后统计完备匹配的权值时再次取反即可；

第二种方法是取边权矩阵中元素最大值 W ，将所有边权值赋值为 $W - weight[i][j]$ ，令新边权下的完全二部图的最大权完备匹配的权值为 w' ，则原边权下的完全二部图的最小权完备匹配的权值为 $w = nW - w'$ ，其中 n 为边权矩阵的阶，即图中 X （或 Y ）侧顶点的数目^I。

由于需要确定将 n 颗石子排列成一条直线的最少移动步数，而排列成一条直线可以有多种选择，可以选择排列成 n 行中的某一行，或者排列成 n 列中的某一列，或者排列成主、副对角线之一，因此要检查所有情形的最小权匹配，取其中具有最小移动步数的方案作为结果输出。

参考代码采用第二种方法进行解题，使用了全局变量 $slack$ 来替代松弛量数组，对于边权值为整数的加权匹配，不会出现问题，但是对于边权值为实数的加权匹配，在遇到某些特殊的测试数据时，可能由于实数的精度问题导致陷入无限循环^{II}。

参考代码

```
const int MAXN = 110, INF = 0x7f7f7f7f7f;

int weight[MAXN][MAXN], linky[MAXN], cx[MAXN], cy[MAXN], slack;
int lx[MAXN] = {0}, ly[MAXN] = {0}, n;

bool dfs(int x) {
    cx[x] = true;
    for (int y = 0; y < n; y++) {
        if (cy[y]) continue;
        int delta = lx[x] + ly[y] - weight[x][y];
        if (!delta) {
            cy[y] = true;
            if (linky[y] == -1 || dfs(linky[y])) {
                linky[y] = x;
                return true;
            }
        }
    }
    // 使用全局 slack 变量而不是松弛量数组。
    else slack = min(slack, delta);
}
return false;
}
```

^I 此方法的正确性基于以下结论：设 a 是 $(K_{n, n}, w)$ 的加权矩阵 $W = (w_{ij})_n$ 中元素的最大值， J_n 是 n 阶全 1 方阵， $W^* = (w_{ij}^*)_n = aJ_n - W$ 是 $(K_{n, n}, w^*)$ 中加权矩阵。则 M^* 是 $(K_{n, n}, w^*)$ 中最大权完备匹配 $\Leftrightarrow M^*$ 是 $(K_{n, n}, w)$ 中最小权完备匹配，而且 $w(M^*) = na - w^*(M^*)$ 。 $K_{n, n}$ 表示二部划分为 $\{X, Y\}$ 的完全二部图且 $|X| = |Y| = n$ 。此结论的证明过程可参阅：徐俊明，《图论及其应用（第 3 版）》，第 222—223 页。

^{II} 例如 1411 Ants，如果使用松弛量数组的方法解题能够获得 Accepted，但如果使用全局变量来表示松弛量，则会得到 Time Limit Exceeded 的提交结果。

```

}

int kuhnMunkres() {
    memset(linky, -1, sizeof(linky));
    for (int i = 0; i < n; i++) {
        lx[i] = -INF, ly[i] = 0;
        for (int j = 0; j < n; j++)
            lx[i] = max(lx[i], weight[i][j]);
    }
    for (int x = 0; x < n; x++)
        while (true) {
            memset(cx, 0, sizeof(cx));
            memset(cy, 0, sizeof(cy));
            slack = INF;
            if (dfs(x)) break;
            for (int i = 0; i < n; i++) {
                if (cx[i]) lx[i] -= slack;
                if (cy[i]) ly[i] += slack;
            }
        }
    int r = 0;
    for (int y = 0; y < n; y++)
        if (~linky[y])
            r += weight[linky[y]][y];
    return r;
}

// 对边权进行调整然后调用 Kuhn-Munkres 算法求最大匹配。
int modifiedKuhnMunkres() {
    int a = 0;
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            a = max(a, weight[i][j]);
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)
            weight[i][j] = a - weight[i][j];
    int r = kuhnMunkres();
    return n * a - r;
}

int main(int argc, char *argv[]) {
    int sx[16], sy[16], cases = 0;
    while (cin >> n, n > 0) {
        for (int i = 0; i < n; i++) cin >> sx[i] >> sy[i];
        int r = INF;
        // 将石子排在某一行上。
        for (int row = 1; row <= n; row++) {
            memset(weight, 0, sizeof(weight));
            for (int cln = 1; cln <= n; cln++)
                for (int si = 0; si < n; si++)
                    weight[si][cln - 1] = abs(sx[si] - row) + abs(sy[si] - cln);
            r = min(r, modifiedKuhnMunkres());
        }
        // 将石子排在某一列上。
        for (int cln = 1; cln <= n; cln++) {
            memset(weight, 0, sizeof(weight));
            for (int row = 1; row <= n; row++)

```

```

        for (int si = 0; si < n; si++)
            weight[si][row - 1] = abs(sx[si] - row) + abs(sy[si] - cln);
        r = min(r, modifiedKuhnMunkres());
    }
    // 将石子排在主对角线上。
    memset(weight, 0, sizeof(weight));
    for (int row = 1, cln = 1; row <= n; row++, cln++)
        for (int si = 0; si < n; si++)
            weight[si][row - 1] = abs(sx[si] - row) + abs(sy[si] - cln);
    r = min(r, modifiedKuhnMunkres());
    // 将石子排在副对角线上。
    memset(weight, 0, sizeof(weight));
    for (int row = n, cln = 1; cln <= n; row--, cln++)
        for (int si = 0; si < n; si++)
            weight[si][cln - 1] = abs(sx[si] - row) + abs(sy[si] - cln);
    r = min(r, modifiedKuhnMunkres());
    // 输出。
    cout << "Board " << ++cases << ":" << r << " moves required.\n";
    cout << '\n';
}
return 0;
}

```

强化练习：1411* Ants^D，11383 Golden Tiger Claw^D。

扩展练习：1349* Optimal Bus Route Design^D。

10.8 点支配集、点覆盖集、点独立集

10.8.1 点支配集

设 G 是无向图， S 是 $V(G)$ 的非空子集，若对于任意顶点 $v \in V(G) \setminus S$ ，存在顶点 $u \in S$ ，在顶点 u 和顶点 v 之间存在无向边 $(u, v) \in E(G)$ ，则称 u 支配 v ，并称 S 为 G 的点支配集 (vertex dominating set)，简称点支配。如果 G 中任何异于 S 的点支配 S' 均有 $|S'| \geq |S|$ ，则称 S 为最小点支配 (minimum vertex dominating)。若对任何 $x \in S$ ， $S \setminus \{x\}$ 均不是点支配，则称点支配集 S 为极小点支配 (minimal vertex dominating)。最小点支配中的顶点数称为点支配数 (vertex dominating number)，记做 $\gamma(G)$ 。

对于一般图来说，求最小点支配为 **NP 完全** 问题，只能通过枚举的方法进行解决，但是对于特殊的图，如树图，存在有效的贪心算法和动态规划算法。

强化练习：10160* Servicing Stations^C。

10.8.2 点覆盖集

设 G 是无向连通图， S 是顶点集 $V(G)$ 的非空子集，若边集 $E(G)$ 中每条边都与 S 中某顶点关联，则称 S 为 G 的点覆盖集 (vertex covering set)，简称点覆盖。如果 G 中任何异于 S 的点覆盖 S' 均有 $|S'| \geq |S|$ ，则称 S 为最小点覆盖 (minimum vertex covering)。对于某个点覆盖集 S ，若对任何 $x \in S$ ， $S \setminus \{x\}$ 均不是点覆盖，则称 S 为极小点覆盖 (minimal vertex covering)。最小点覆盖一定是极小点覆盖，但极小点覆盖不一定是最小点覆盖。

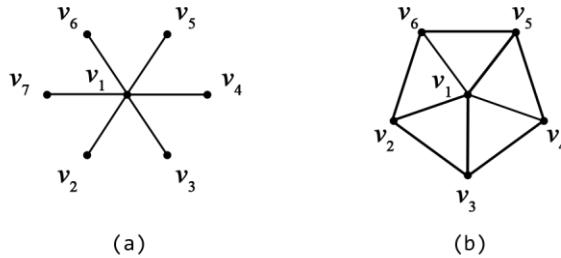


图 10-28 点覆盖。(a) $\{v_2, v_3, v_4, v_5, v_6, v_7\}$ 是极小点覆盖, 但不是最小点覆盖, $\{v_1\}$ 是极小点覆盖, 同时也是最小点覆盖; (b) $\{v_1, v_2, v_4, v_5\}$ 和 $\{v_1, v_2, v_4, v_6\}$ 既是极小点覆盖, 也是最小点覆盖

G 中最小点覆盖的顶点数称为点覆盖数 (vertex covering number), 记为 $\beta(G)$, 在二部图中, G 中最大匹配的边数称为匹配数 (matching number), 记为 $\alpha'(G)$, König 证明: 对任何二部图 G 有 $\beta(G)=\alpha'(G)$, 即求二部图 G 的点覆盖数等价于求其匹配数^[131]。注意, 对于非二部图, 上述结论不成立。

1194 Machine Schedule^D (机器调度)

假设有两台机器 A 和 B , 机器 A 有 n 种工作模式, 分别为 $mode_0, mode_1, \dots, mode_{n-1}$, 机器 B 有 m 种工作模式, 分别为 $mode_0, mode_1, \dots, mode_{m-1}$ 。初始时, A 和 B 的工作模式均为 $mode_0$ 。

给定 k 项作业, 每项作业可以工作在任何一台机器的特定模式下。例如, 作业 0 可以工作在机器 A 的 $mode_3$ 模式或者机器 B 的 $mode_4$ 模式; 作业 1 可以工作在机器 A 的 $mode_2$ 模式或者机器 B 的 $mode_4$ 模式; 等等。因此对于作业 i , 调度中的约束条件可以表述成一个三元组 (i, x, y) , 含义为作业 i 可以工作在机器 A 的 $mode_x$ 模式或者机器 B 的 $mode_y$ 模式。

显然, 为了完成所有作业, 必须不定时切换机器的工作模式, 但不巧的是, 机器工作模式的切换只能通过手动重启机器完成。试编写程序, 通过改变机器的顺序, 给每台机器分配合适的作业, 使得重启机器的次数最少。

输入

输入文件包含多组测试数据, 每组测试数据的第一行包含 3 个整数: n, m ($n, m < 100$) 和 k ($k < 1000$), 接下来的 k 行给出了 k 项作业的约束条件, 每行为一个三元组: i, x, y 。输入文件的最后一行以 0 结束。

输出

对于输入文件的每组测试数据输出一行, 包含一个整数, 表示需要重启机器的最少次数。

样例输入

```
5 5 10
0 1 1
1 1 2
2 1 3
3 1 4
4 2 1
5 2 2
6 2 3
7 2 4
8 3 3
9 4 3
```

样例输出

```
3
```

0

分析

将机器 A 的 n 个模式和机器 B 的 m 个模式视为图的顶点, 如果某项作业可以在 A 的 $mode_x$ 模式或 B 的 $mode_y$ 模式下完成, 则从 $mode_x$ 到 $mode_y$ 连接一条边, 最后得到的是一个二部图。若要使得机器的重启次数最少, 应该使得某个模式能够处理更多的作业, 等价于使顶点覆盖尽可能的多的边, 即题目所求为二部图的最小点覆盖。根据前述的 König 定理, 任意二部图的点覆盖数与匹配数相等, 因此求构造得到的二部图最大匹配即可。

参考代码

```

const int MAXV = 110;

int g[MAXV][MAXV], vx[MAXV], vy[MAXV], cx[MAXV], cy[MAXV], nx, my, kj;

// 使用深度优先搜索寻找增广路。
int dfs(int u) {
    // 设置机器 A 的模式 u 为已访问。
    vx[u] = 1;
    // 考虑所有与模式 u 能够完成同一项作业的模式 v。由于机器的初始状态为模式 0,
    // 不需要重启机器, 故从模式 1 开始寻找增广路。
    for (int v = 1; v < my; v++)
        if (g[u][v] && !vy[v]) {
            // 如果 v 尚未匹配或者 v 已经匹配但是从 v 出发可以找到增广路则匹配成功。
            vy[v] = 1;
            if (!cy[v] || dfs(cy[v])) {
                // 更改匹配。
                cx[u] = v, cy[v] = u;
                return 1;
            }
        }
    // 未能找到增广路, 已经是最大匹配。
    return 0;
}

// 匈牙利算法求最大匹配数。
int hungarian() {
    int matches = 0;
    memset(cx, 0, sizeof(cx));
    memset(cy, 0, sizeof(cy));
    // 机器 A 和机器 B 最初工作在模式 0, 在模式 0 状态下可以完成的作业不需要重新启动机器,
    // 故从模式 1 开始匹配。
    for (int i = 1; i < nx; i++)
        if (!cx[i]) {
            // 注意寻找之前需要将访问标记置为初始状态。
            memset(vx, 0, sizeof(vx)); memset(vy, 0, sizeof(vy));
            // 每找到一条增广路, 可使得匹配数增加 1。
            matches += dfs(i);
        }
    return matches;
}

int main(int argc, char *argv[]) {

```

```

while (cin >> nx, nx > 0) {
    cin >> my >> kj;
    // 根据作业约束建立二部图。
    memset(g, 0, sizeof(g));
    for (int t, x, y, i = 0; i < kj; i++) {
        cin >> t >> x >> y;
        g[x][y] = 1;
    }
    // 使用匈牙利算法求匹配数，从而得到点覆盖数。
    cout << hungarian() << '\n';
}
return 0;
}

```

强化练习：12549 Sentry Robots^D。

确定了二部图的点覆盖数 $\beta(G)$ 后，如何从顶点集中选出 $\beta(G)$ 个顶点来构造最小点覆盖呢？可以使用如下方法：设图 G 二部划分为 $\{X, Y\}$ ，选取 Hungarian 算法结束后 X 侧尚未匹配的顶点（也可以选择 Y 侧尚未匹配的顶点），从这些顶点出发，按照 Hungarian 算法增广路的要求，寻找所有的交错路，这些交错路上的边也是按照“未匹配边—匹配边—未匹配边—…”的交替顺序排列，只不过此时最后一条边一定是匹配边（若是未匹配边则会构成一条增广路，而在之前的 Hungarian 算法执行过程中，已经找到了所有的增广路，因此不会出现是未匹配边的情况），因此是“伪增广路”，将寻找“伪增广路”过程中所经过的顶点予以标记，则最后 X 侧未标记的顶点和 Y 侧已标记的顶点（如果起始时选择从 Y 侧的未匹配顶点开始寻找交错路，则最后选择 X 侧已标记的顶点和 Y 侧未标记的顶点）就构成了最小点覆盖集。

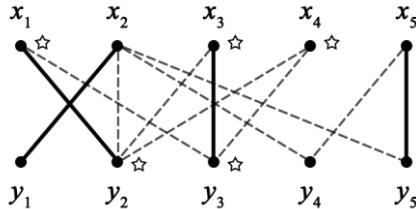


图 10-29 从最大匹配寻找最小点覆盖。设图 G 二部划分为 $\{X, Y\}$ ， $X = \{x_1, x_2, x_3, x_4, x_5\}$ ， $Y = \{y_1, y_2, y_3, y_4, y_5\}$ ，可行的最大匹配如图中实线边所示。从 X 侧未匹配的顶点 x_4 出发，沿着“未匹配—匹配—未匹配—匹配—…”的顺序寻找“伪交错路”，可以找到两条“伪交错路”，一条为 $x_4 \rightarrow y_2 \rightarrow x_1 \rightarrow y_3 \rightarrow x_3$ ，另外一条为 $x_4 \rightarrow y_3 \rightarrow x_3$ ，两条“伪交错路”标记了 X 侧的 x_1, x_3, x_4 ， Y 侧的 y_2, y_3 ，而 X 侧的 x_2, x_5 未标记，则 $\{x_2, x_5, y_2, y_3\}$ 构成了最小点覆盖集。

可使用类似于前述 Hungarian 算法中应用 DFS 过程寻找增广路的方法，沿着“伪增广路”标记交错路上的顶点。具体的代码实现请读者结合强化练习予以完成。

强化练习：11419 SAM I AM^C。

10.8.3 点独立集与最大团

设 G 是无圈图， S 是 $V(G)$ 的非空子集，若 S 中任意两顶点在 G 中均不相邻，则称 S 为 G 的点独立集 (vertex independent set)，简称点独立。如果对任何顶点 $x \in V \setminus S$ ， $S \cup \{x\}$ 都不是点独立集，则称 S 为极大点独立 (maximal vertex independent set)。如果对 G 中任何异于 S 的点独立 S' 均有 $|S'| \leq |S|$ ，则称 S 为 G 的最大点独立 (maximum vertex independent set)。 G 中最大点独立中的顶点数称为 G 的点独立数 (independent

number), 记为 $\alpha(G)$ 。

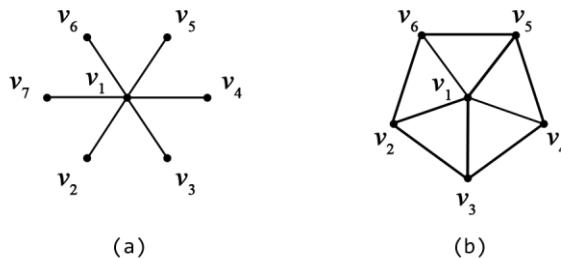


图 10-30 点独立。(a) $\{v_1\}$ 是极大点独立, 但不是最大点独立, $\{v_2, v_3, v_4, v_5, v_6, v_7\}$ 是极大点独立, 同时也是最大点独立; (b) $\{v_2, v_4\}$ 既是极大点独立, 也是最大点独立, 类似的还有 $\{v_3, v_5\}$ 、 $\{v_4, v_6\}$ 等

点独立和点覆盖之间的关系存在如下定理: 设无向图 G 中无孤立顶点, 若顶点集 S 为 $V(G)$ 的子集, 则 S 是 G 的点覆盖的充分必要条件是 $V(G) \setminus S$ 是 G 的点独立, 即 $\alpha(G) + \alpha'(G) = |V|$ 。也就是说, 求最大点独立数和求最小点覆盖数是互补问题。对于二部图, 其点覆盖数等于匹配数, 则二部图的点独立数等于顶点个数减去匹配数。如果二部图中存在独立顶点, 则从等式中扣除独立顶点后等式依然成立。对于一般图来说, 求最大点独立数是 NP 问题, 而对于规模较小的图, 可以使用回溯法进行求解。在求点独立数的题目中, 关键是如何根据题目描述建模得到相应的二部图, 进一步使用相应的算法求最大匹配, 然后间接求得点独立数。

给定无向图 $G = (V, E)$, G 的团 (clique) 是 G 的一个子图 $G' = (V', E')$, 对于 V' 中的任意两个不同顶点 u 和 v , 存在边 $(u, v) \in E$ 。换句话说, 如果图 G' 是图 G 的一个子图, G' 中包含 n 个顶点且这 n 个顶点中的任意两个不同顶点均有一条边相连, 则称 G' 为 G 的团, 亦即 G 的完全子图。如果一个团不被任意其他团所包含, 即它不是其他团的真子集, 则称该团为图 G 的极大团 (maximal clique)。 G 中包含顶点数量最多的团称为 G 的最大团 (maximum clique)。求 G 的最大团等价于求 G 的补图的点独立数。求解极大团和最大团可使用 Bron-Kerbosch 算法^[132]。

强化练习: [10349 Antenna Placement^C](#), [12083 Guardian of Decency^D](#), [12168 Cat vs. Dog^D](#)。

扩展练习: [11065 A Gentlemen's Agreement^D](#), [11069 A Graph Problem^A](#)。

10.9 路径覆盖和边覆盖

路径覆盖 (path covering) 是指图中路径集 P 的一个子集 P' , P' 经过图中所有的顶点。边覆盖 (edge covering) 是指图 G 中边集 E 的一个子集 E' , E' 关联图 G 的所有顶点。

10.9.1 最小路径覆盖

若某个路径覆盖的任意真子集均不是路径覆盖, 则称此路径覆盖为最小路径覆盖 (Minimum Path Covering, MPC)。对于一般图, MPC 问题属于 NP 难问题, 目前尚无有效算法, 但是对于有向无圈图中的 MPC 问题, 存在有效算法。

有向无圈图中的最小路径覆盖可分为两种类型, 一种是最小不相交路径覆盖, 即所有路径经过的顶点均不相同, 另外一种是最小可相交路径覆盖, 即路径可发生相交。对于最小不相交路径覆盖, 可以将其转换为最大匹配问题予以解决, 即将有向图中的每个顶点拆分为两个顶点 v_{in} 和 v_{out} , 如果顶点 x 和顶点 y 之间具有有向边, 则在新图中的顶点 x_{out} 和 y_{in} 间建立一条有向边, 这样得到的新图就是一个二部图, 而原图的最小路径覆盖就等于原图的顶点数减去新图的最大匹配。

如图 10—31 所示, 原给定的有向图为 G_1 , 将其每个顶点进行拆分得到的二部图为 G_2 , 易知 G_2 的最大匹配数为 4, 故原图 G_1 的最小路径覆盖为 2, 即 1—2—3、4—5—6 这两条路径是有向图 G_1 的最小路径覆盖。为了便于理解, 可以做以下推理: 假设原图 G_1 中不存在任何边, 则最小路径覆盖即为 G_1 的顶点个数 $|G_1|$, 当转换得到的二部图 G_2 中每新增 1 对匹配, 则由于两个顶点在同在一条路径上, 则 G_1 的路径覆盖数将减少 1, 若 G_2 有 m 对匹配, 则 G_1 的路径覆盖数将减少 m , 故 G_1 的最小路径覆盖数即为 $|G_1|$ 减去 G_2 的最大匹配数。

对于最小可相交路径覆盖, 可以将其转化为最小不相交路径覆盖解决。对于原图, 使用 Floyd-Warshall 算法求出其传递闭包, 如果顶点 u 和 v 之间具有有向路径, 则在新图中为 u 和 v 添加一条有向边, 这样就能够把原图的最小可相交路径覆盖问题转化为新图的最小不相交路径覆盖问题。

强化练习: [1184 Air Raid^D](#), [1201 Taxi Cab Scheme^D](#)。

10.9.2 最小边覆盖

若某个边覆盖的任意真子集均不是边覆盖, 则称此边覆盖为最小边覆盖 (minimum edge covering)。最小边覆盖中的边数称为 G 的边覆盖数 (edge covering number), 记为 $\beta'(G)$ 。Norman 和 Rabin 证明, 图的最小边覆盖和图的最大匹配实质上为互补问题^[133]。令图 G 的顶点数为 n , 边覆盖数为 $\beta'(G)$, 匹配数为 $\alpha'(G)$, 若图中不存在孤立点, 有

$$\beta'(G) = n - \alpha'(G)$$

对于二部图 G 来说, 令其点独立数为 $\alpha(G)$, 边覆盖数为 $\beta'(G)$, 若图中不存在孤立点, 则有

$$\alpha(G) = \beta'(G)$$

10.10 树的相关问题求解

当给定的图是任意图时, 图的点支配集、点覆盖集、点独立集问题都是 NP 问题。但是对于树, 存在有效的算法。对于树的点支配集、点覆盖集、点独立集问题, 有的可以将其转化为二部图后进一步使用本章介绍的算法予以求解, 有的可以使用贪心算法或树形动态规划算法予以解决。

以下介绍在树中求最小点支配、最小点覆盖、最大点独立的相应算法^I。需要注意的是, 对于三种类型问题的贪心算法, 贪心策略中贪心选择的顺序非常重要, 需要按照 DFS 所得到的遍历序列的反方向进行贪心选择, 这样可以确保对于每个结点来说, 当其子树都被处理过后才会对该结点进行处理, 从而能够保证贪心算法的正确性。

10.10.1 最小点支配

对于图 $G=(V, E)$, 选取 V 中的尽可能少的顶点构成一个集合 V_1 , 使得 $V \setminus V_1$ 中的顶点均与 V_1 中的顶点有边关联, 则称 V_1 为图 G 的最小顶点支配集, 简称最小点支配。

贪心算法^[134]: 以某个结点为根结点, 从此结点开始对树进行深度优先遍历, 按发现结点的先后顺序构建遍历序列, 并记录各个结点的父结点, 之后对得到的遍历序列按照相反的方向进行贪心选择, 对于一个既不属于支配集同时也不与支配集中的结点相连的结点, 如果它的父结点不属于支配集, 将其父结点加入支配

^I 使用动态规划解决树的最小点支配、最小点覆盖、最大点独立问题的方法, 请读者参阅本书第 11 章“动态规划”中第 11.4.3 小节“图论型动态规划”中“树形动态规划”的内容。

集, 标记当前结点、当前结点的父结点和当前结点父结点的父结点。

```
-----10.10.1.cpp-----
const int MAXV = 2048;

vector<int> edges[MAXV], sequence;
int n, visited[MAXV], parent[MAXV], in[MAXV];

// 通过深度优先遍历构建遍历序列。
void dfs(int u) {
    sequence.push_back(u);
    visited[u] = 1;
    for (auto v : edges[u])
        if (!visited[v]) {
            parent[v] = u;
            dfs(v);
        }
}

int greedy() {
    memset(visited, 0, sizeof(visited));
    memset(in, 0, sizeof(visited));
    int r = 0;
    // 根结点满足贪心条件, 需要对根结点进行检查。
    for (int i = n - 1; i >= 0; i--) {
        int u = sequence[i];
        if (!visited[u]) {
            if (!in[parent[u]]) {
                in[parent[u]] = 1;
                r += 1;
            }
            // 当前结点、当前结点的父结点、当前结点父结点的父结点, 均标记为已访问。
            visited[u] = visited[parent[u]] = visited[parent[parent[u]]] = 1;
        }
    }
    return r;
}
-----10.10.1.cpp-----
```

10.10.2 最小点覆盖

最大匹配算法: 求树的最小点覆盖, 可先将其转换为二部图, 求其最大匹配。由于树是可以二着色的, 先使用 BFS (或 DFS) 将树进行二着色, 例如着色为黑色和白色, 那么可将黑色结点作为二部图的 X 侧, 白色结点作为二部图的 Y 侧, 然后根据黑色结点和白色结点间是否存在边构建二部图中的边, 二部图构建完毕后选用求最大匹配的算法求得最大匹配即为最小点覆盖。

贪心算法: 以某个结点为树的根结点, 从此结点开始对树进行深度优先遍历, 按发现结点的先后顺序构建遍历序列, 并记录各个结点的父结点, 之后对得到的遍历序列按照相反的方向进行贪心选择, 如果当前结点和及其父结点都不属于顶点覆盖集合, 则将父结点加入到顶点覆盖集合, 并标记当前结点与其父结点均被覆盖。需要注意的是贪心算法中选择的顺序非常重要, 需要按照深度优先遍历得到遍历序列的反向顺序进行贪心, 这样对于每个结点来说, 可以保证当其子树都被处理完毕后才进行该结点的处理, 从而能够保证贪心选择的正确性。注意, 在最小点覆盖问题中, 由于根结点的父结点为本身, 不满足贪心条件, 因此不能对根结点进行检查, 否则会导致错误的结果。

```
-----10.10.2.cpp-----
int greedy() {
    memset(visited, 0, sizeof(visited));
    memset(in, 0, sizeof(visited));
    int r = 0;
    // 根结点不满足贪心条件, 不做检查, 否则会导致错误的结果。
    for (int i = n - 1; i >= 0; i--) {
        int u = sequeue[i];
        if (!visited[u] && !visited[parent[u]]) {
            in[parent[u]] = 1;
            r += 1;
            visited[u] = visited[parent[u]] = 1;
        }
    }
    return r;
}
-----10.10.2.cpp-----
```

强化练习: 1292 Strategic Game^D, 10243 Fire Fire Fire^C。

10.10.3 最大点独立

最大匹配算法: 求树的最大点独立, 可先将其转换为二部图, 求其最大匹配。由于树是可以二着色的, 先使用 BFS (或 DFS) 将树进行二着色, 例如着色为黑色和白色, 那么可将黑色结点作为二部图的 X 侧, 白色结点作为二部图的 Y 侧, 然后根据黑色结点和白色结点间是否存在边构建二部图中的边, 二部图构建完毕后选用求最大匹配的算法求得最大匹配, 由于二分图的最大匹配数即为最小点覆盖数, 而最小点覆盖数与最大点独立数为互补问题, 因此可以间接求解。

贪心算法: 选择树中的某个结点作为根结点, 按照 DFS 得到遍历序列, 之后按照遍历序列所得到的反向序列的顺序进行贪心选择, 贪心的策略是: 如果当前结点未被覆盖, 则将当前结点加入独立集, 并标记当前结点及其父结点均被覆盖。由于根结点的父结点是其本身, 故需要单独检查根结点是否满足贪心选择的条件。贪心算法的时间复杂度为 $O(V)$ 。

```
-----10.10.3.cpp-----
int greedy() {
    memset(visited, 0, sizeof(visited));
    memset(in, 0, sizeof(visited));
    int r = 0;
    // 根结点满足贪心条件, 需要对根结点进行检查。
    for (int i = n - 1; i >= 0; i--) {
        int u = sequeue[i];
        if (!visited[u]) {
            in[parent[u]] = 1;
            r += 1;
            visited[u] = visited[parent[u]] = 1;
        }
    }
    return r;
}
-----10.10.3.cpp-----
```

10.11 小结

图算法的内容非常丰富, 是编程竞赛考察的一个重点和热点。由于图算法中每种算法都是针对特定的问

题, 因此在学习算法时, 可以按照以下步骤来进行学习: 第一步是了解算法所针对的问题背景, 然后从算法的思想入手, 尽可能对其思想有一个透彻的了解, 然后就是琢磨算法实现, 最后就是通过练习来熟练算法的应用。

图算法中的重点内容包括:

(1) 欧拉回路算法。重点需要掌握 Fleury 算法的思想以及 Hierholzer 算法的实现。作为拓展, 读者可以进一步了解混合图中欧拉回路的求法。

(2) 最小生成树算法。最小生成树算法主要包括 Prim 算法和 Kruskal 算法。Prim 算法和 Kruskal 算法同属贪心算法。总的来说, Prim 算法是以顶点为对象, 挑选与顶点相连的最短边来构成最小生成树。而 Kruskal 算法是以边为对象, 不断地加入新的不构成圈的最短边来构成最小生成树。在掌握最小生成树算法的基础上, 需要掌握其衍生和扩展问题, 包括但不限于次最优最小生成树、单度限制最小生成树、最优比例生成树、最小树形图等。

(3) 单源最短路径算法。如果给定的图中不包含负权边, 则最常使用的是 Moore-Dijkstra 算法。Moore-Dijkstra 使用了广度优先搜索解决加权有向图或者无向图的单源最短路径问题, 算法最终得到一个最短路径树。如果给定的图中包含负边权, 则 Moore-Dijkstra 算法可能不适用, 此时可以使用 Bellman-Ford 算法来求单源最短路径。在掌握最短路径算法的基础上, 可以进一步了解第 k 短路径算法。SPFA 可以看做是 Bellman-Ford 算法的队列优化实现, 在某些情况下可以大幅提高运行效率, 但是往往也可以构造特殊的测试数据使得 SPFA 超时, 这一点需要在实践中予以注意。

(4) 多源最短路径算法。Floyd-Warshall 算法是解决多源最短路径的经典算法, 该算法采用了动态规划中的松弛技巧。

(5) 网络流算法。需要重点理解 Ford-Fulkerson 方法的思想, 理解 Edmonds-Karp 算法对原始的 Ford-Fulkerson 方法的朴素实现的优化, 重点需要掌握的是最大流的 Dinic 算法以及 ISAP 算法的思想。对于学有余力的读者, 建议进一步了解在网络流问题基础上增加限制得到的最小费用最大流问题, 以及本书未予介绍的有上下界限制的最小费用最大流问题。

(6) 二部图匹配算法。匹配算法中最基本的是 Hungarian 算法。在此基础上需要掌握 Hopcroft-Karp 算法以及针对最大权完备匹配问题的 Kuhn-Munkres 算法。

第 11 章 动态规划

Everything should be as simple as possible, but not simpler.

——阿尔伯特·爱因斯坦^I

复杂的问题要善于“退”，足够地“退”，“退”到最原始而不失去重要的地方，是学好数学的一个诀窍。
——华罗庚^{II}

动态规划 (Dynamic Programming, DP, “programming”是“规划”之意而不是指“编程”) ^[135] 作为解决一类问题的有效工具，具有以下两个重要的特点：

(1) 动态规划既考虑了问题的所有可能性，保证了正确性，又在解决问题的时候，通过保存中间结果来避免不必要的重复计算，从而保证了效率。

(2) 动态规划要求能将整个问题的解表示成若干子问题的解，一个重要的步骤是构建问题解的“递推关系式”，又称状态转移方程。

问题具有最优子结构且具有重叠的子问题是应用动态规划的标志，即问题的结构符合最优化原理，同时又具有无后效性的特点^[136]。最优化原理可以这样阐述：一个最优化策略具有这样的性质，不论过去状态和决策如何，对之前决策所形成的状态而言，后继决策必须构成最优策略。简而言之，一个最优化策略的子策略总是最优的。最优化原理是动态规划的基础，任何问题，如果失去了最优化原理的支持，就不可能用动态规划方法求解。无后效性是指“过去的步骤只能通过当前状态影响未来的发展，当前的状态是历史的总结”。这条特征说明动态规划只适用于解决当前决策与过去状态无关的问题。某个状态，出现在策略的任何一个位置，它的地位相同，都可实施同样策略，这就是无后效性的内涵。最优化原理以及无后效性，是动态规划必须符合的两个条件。

动态规划算法的设计一般可以分为以下四个步骤：(1) 描述最优解的结构。(2) 递归定义最优解的值。(3) 按自底向上（或者自顶向下）的方式计算最优解的值。(4) 由计算出的结果构造一个最优解。

11.1 背包问题

11.1.1 01 背包问题

给定一个容量为 C 的背包，现有 n 个物品，每个物品具有容量 C_i 和价值 P_i ， $1 \leq i \leq n$ ，问如何选取放入背包的物品，使得背包内的物品容量之和不超过 C 且价值之和最大。由于物品要么放入背包，要么不放入背包，令 x_i 表示物品在背包中的状态，则有 $x_i \in \{0, 1\}$ ，故称 01 背包问题。

步骤 1：描述最优解的结构

朴素的方法是使用回溯法来解决 01 背包问题，即通过回溯法确定从 n 个物品中选出 1 个物品、2 个物

^I 阿尔伯特·爱因斯坦 (Albert Einstein, 1879—1955)，20 世纪犹太裔理论物理学家。他创立了相对论并且是质能等价公式 ($E=mc^2$) 的发现者。因为对理论物理的贡献，特别是发现了“光电效应”，荣获 1921 年诺贝尔物理学奖。

^{II} 华罗庚 (1910—1985)，中国现代数学家，他是中国解析数论、矩阵几何学、典型群、自守函数论等多方面研究的创始人和开拓者。

品、3个物品、 \cdots 、 n 个物品时所有可能的容量及价值和，从中选取容量小于等于 C 且价值最大的物品组合。很明显，一个物品要么选择，要么不选，使用此种方法的时间复杂度将是 $O(2^n)$ ，当 n 增大时算法不可行。为什么当 n 增大时，回溯法变得低效了？一个重要原因是在回溯过程中出现了许多重复的状态，如果对这些重复状态不加以剪枝会导致搜索时间显著增加。考虑问题所给定的条件，背包的状态由放入的物品数量和背包的容量决定，如果初始时不放入任何物品，那么背包最多只有 $n \times C + 1$ 种状态，只需确定这些状态下背包的最大价值即可，相较于 2^n 级别的状态数，从这一角度考虑问题可以使得问题的状态数大大减少，从而有利于问题的解决。另外使用此种方式考虑问题可以额外带来一个好处，即后续状态可由前置状态递推而得到。

假设在最后的最优结果中总共放入了 x 个物品，那么这 x 个物品放入的先后顺序对最优结果是没有影响的，因此不妨将物品按序号从 1 到 n 的顺序，逐个考虑是否放入背包中。假设当处理到第 i 个物品且背包的容量为 j 时（需要注意此时背包的容量并不一定已经全部使用）所对应的背包最大价值为 $V_{i,j}$ ，使用一个二维数组 V ，以物品的序号 i 和背包容量 j 做下标来表示最大价值 $V_{i,j} = V[i][j]$ ，那么最终背包问题所求为 $V[n][C]$ 的值。

假设已经得到了当处理到物品序号为 $i-1$ 且背包容量为 0 至 j 时的背包价值最大值，即已经确定数组元素 $V[i-1][0]$ 至 $V[i-1][j]$ 的值。当处理物品序号为 i 且背包容量为 j 时的背包最大值时，有两种可能，一种是背包的容量 j 小于物品 i 的容量 C_i ，则物品 i 无法放入背包中，那么 $V[i][j]$ 应该更新为 $V[i-1][j]$ ；反之，若 j 大于物品 i 的容量 C_i ，需要考虑放入物品 i 是否可使 $V[i][j]$ 的值更大，如果能增大背包价值则应该将物品 i 放入背包中，背包的最大价值为 $V[i-1][j-C_i] + P_i$ （需要注意，对于 $V[i][j]$ 来说，考虑放入物品 i ，其子问题中背包的容量为 $j-C_i$ ）。换句话说，如果有 $V[i-1][j-C_i] + P_i$ 大于 $V[i][j]$ ，则 $V[i][j]$ 应该更新为 $V[i-1][j-C_i] + P_i$ ，否则仍为 $V[i-1][j]$ 。

以下通过一个实例来说明最优解的构造过程。设背包的容量为 10，有 5 个物品，其编号、容量、价值如下表所示：

物品	<i>a</i>	<i>b</i>	<i>c</i>	<i>d</i>	<i>e</i>
容量	2	2	4	5	6
价值	2	3	2	4	5

按照解决问题的思路，在求解 $V[1][0]$ 至 $V[1][10]$ 的过程中，需要知道 $V[0][0]$ 至 $V[0][10]$ 的值，为便于问题的处理，可以设置一个虚拟物品 σ 作为初始值，其序号和价值均为 0，在对应的背包容量中其最大价值亦为 0。当背包容量为 0 的时候，无论处理到第几个物品，数组元素 $V[i][0]$ 均为 0。从物品 a 开始逐个处理，可以得到下表：

容量/价值	0	1	2	3	4	5	6	7	8	9	10
σ	0	0	0	0	0	0	0	0	0	0	0
<i>a</i>	0	0	2	2	2	2	2	2	2	2	2
<i>b</i>	0	0	3	3	5	5	5	5	5	5	5
<i>c</i>	0	0	3	3	5	5	5	5	7	7	7

<i>d</i>	0	0	3	3	5	5	5	7	7	7	7
<i>e</i>	0	0	3	3	5	5	5	7	7	9	10

第一行为虚拟物品 σ ，由初始条件， σ 行对应的元素值均为 0。从物品 a 开始处理， a 物品对应行元素是背包容量为 0 到 10 时所能获得的最大价值，当背包容量为 1 时，由于物品 a 的容量为 2，很明显，物品 a 无法放下，背包的最大价值为 σ 行对应元素的值，当背包容量为 2 至 10 时，只有物品 a 可以放入，由于大于 σ 行对应元素的值，即满足条件： $V[\sigma][0] + P_a > V[a][2]$ ，则 $V[a][2]$ 应更新为 $V[\sigma][0] + P_a$ ，即最大价值为 2。按照上述推导过程填表，最后可以得到 $V[e][10] = 10$ ，即为问题所求。

步骤 2：递归定义最优解的值

由第一步，已经得到了最优子结构和使用子问题的解来得到整体问题解的方法，接下来是要得到一个解的递归定义，即解可以由子问题的解来进行表示。该步骤亦为找到问题的递推关系式，或称状态转移方程。

当处理到物品 i 时，有两种可能，一种情况是背包的容量 j 小于物品 i 的容量 C_i ，导致物品 i 无法放入，那么当前背包的最大价值仍然为 $V[i-1][j]$ ，即

$$V[i][j] = V[i-1][j], \quad j < C_i$$

另外一种情况是可以放入，即 $j \geq C_i$ ，若 $V[i-1][j - C_i] + P_i > V[i-1][j]$ ，则应更新 $V[i][j]$ ，有

$$V[i][j] = V[i-1][j - C_i] + P_i, \quad j \geq C_i$$

否则最大值仍为 $V[i-1][j]$ ，那么有递推关系

$$V[i][j] = \max \begin{cases} V[i-1][j] & j < C_i \\ V[i-1][j - C_i] + P_i & j \geq C_i \end{cases}$$

步骤 3：按自底向上的方式计算最优解的值

按照递推关系进行计算，一般都是自底向上进行，保存相应的中间结果。下述实现中，*capacity* 为背包的容量，*volume[i]* 为物品 i 的容量，*price[i]* 为物品 i 的价值。初始时，物品数量为 0 或背包容量为 0 时，背包的最大价值为 0。

```
int v[n][capacity + 1] = {0};

for (int i = 1; i <= n; i++)
    for (int j = 1; j <= capacity; j++) {
        if (j >= volume[i] && v[i - 1][j - volume[i]] + price[i] > v[i - 1][j])
            v[i][j] = v[i - 1][j - volume[i]] + price[i];
        else
            v[i][j] = v[i - 1][j];
    }
}
```

步骤 4：回溯得到最优解

仅仅知道背包的最大价值在某些情况下还不足够，还需确定是放入了哪些物品而得到最大价值，这就需要在动态规划过程中保存相应记录，以便知道在更新背包最大值时使用了哪个物品。可以通过设立一个二维状态数组 *chosen* 记录处理第 i 个物品，背包容量为 j 时是否将第 i 个物品放入背包。如果 *chosen[i][j]* 为 1 表示处理第 i 个物品，背包容量为 j 时放入物品 i ，为 0 则表示不予放入。初始时，数组 *chosen* 的所有元素均设置为 0，即不放入任何物品。在更新背包最大价值时增加记录步骤：

```
if (v[i - 1][j - volume[i]] + price[i] > v[i - 1][j]) {
```

```

v[i][j] = v[i - 1][j - volume[i]] + price[i];
chosen[i][j] = 1;
}
else v[i][j] = v[i - 1][j];

```

当得到 $V[n][C]$ 的值时, 可以通过此状态数组反向查找得到放入的物品序号:

```

indexer = n, capacity = C;
while(indexer > 0) {
    if (chosen[indexer][capacity] == 1) {
        cout << indexer << endl;
        capacity -= volume[indexer];
    }
    indexer--;
}

```

在网络上有关背包问题的讨论给出了 01 背包问题求解的伪代码^[137]:

```

F[0, 0..V] <- 0
for i <- 1 to N
    for v <- Vi to V
        F[i, v] <- max(F[i - 1, v], F[i - 1, v - Vi] + Pi)

```

其中 $F[i, v]$ 表示将前 i 件物品放入容量为 v 的背包中所能得到的最大价值。伪代码并未提及初始化的细节, 如果使用第 i 个物品的容量作为循环的初始值进行递推, 则事先应该将 $F[i-1, 0..V_i-1]$ 的值复制到 $F[i, 0..V_i-1]$, 否则会导致 $F[i, v]$ 的值仍为初始 0 值, 在后续递推中产生错误的结果。而使用优化空间的从后往前进行递推的求解方法时, 可省略此复制初始值的步骤, 因为在递推过程中, 使用的是同一数组, 前一次递推的值仍然保留在数组中。

在前述的 01 背包问题中, 问题要求确定的是“在不超过背包容量情况下所能得到的最大价值”, 如果换一种提法, 要求确定“恰好装满背包时的物品最大价值”, 则在初始化时相应有不同。第一种提法, 如果要求不超过背包容量, 则在任意背包容量时, 选择不放入任何物品都会产生一个合理的解, 即背包的最大价值为 0。而对于第二种提法, 只有当背包容量为 0 时, 选择不放入任何物品能够产生一个合理的解, 对于其他背包容量, 不能确定是否存在放入物品的容量恰为指定背包容量的解, 故其初始值应该设置为一个标记值(例如-1), 表示在此种情况下, 尚不确定是否存在解。在根据递推关系计算时, 需要首先判断是否存在可行解(即判断数组元素是否为初始标记值), 只有在存在可行解时才能进行背包最大价值的更新。

```

// 此处以-1 来表示解未定义。
int v[n + 1][capacity + 1] = {0};
memset(v, -1, sizeof(v));
v[0][0] = 0;
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= capacity; j++)
        if (j >= volume[i] && v[i - 1][j - volume[i]] != -1)
            v[i][j] = max(v[i - 1][j], v[i - 1][j - volume[i]] + price[i]);
        else
            v[i][j] = v[i - 1][j];

```

990 Diving for Gold^B (潜水寻金)

约翰是一名潜水寻宝者, 他刚刚发现了一艘装满宝藏的海盗沉船。约翰乘坐的船上安装了精密的声呐系统, 这套系统可以让约翰确定海底宝藏的位置、深度、包含金币的数量。不巧的是, 约翰忘记了携带 GPS 设备, 因此无法记录沉船的位置以便将来打捞财宝, 只能现在打捞。更糟糕的是, 约翰身边只有一瓶压缩空

气可用。约翰想使用这唯一的一瓶压缩空气潜入水底，将尽可能多的金币打捞上来。现在需要你编写一个程序来帮助约翰，以便让他决定应该打捞哪些宝箱从而使得金币数量最大化。

本问题的限制条件如下：

- (1) 总共有 n 个宝箱， $\{(d_1, v_1), (d_2, v_2), \dots, (d_n, v_n)\}$ ，每个二元组表示宝箱的深度和包含的金币数量。 n 最大为 30；
- (2) 压缩空气瓶只能让潜水者在水下保持 t 秒。 t 最大为 1000；
- (3) 约翰每次潜水最多只能带上来一个宝箱；
- (4) 下潜所用时间 td_i 和宝箱的深度 d_i 线性相关： $td_i = w \times d_i$ ， w 是一个整数常数；
- (5) 上升所用时间 ta_i 和宝箱的深度 d_i 线性相关： $ta_i = 2w \times d_i$ ， w 是一个整数常数；
- (6) 由于设备的限制，所有参数均为整数类型。

输入

输入包含多组数据，每组数据包含若干整数值。每组数据的第一行包含两个整数 t 和 w ，分别表示潜水总时间和整数常数。第二行表示宝箱的数量。接下来的输入每行包含两个整数 d_i 和 v_i ，表示不同宝箱的深度和包含的金币数量。每两组输入数据之间有一个空行。

输出

对于每组测试数据，输出的第一行包含一个整数，表示约翰在给定的条件下能够打捞的金币数量的最大值。第二行表示能够打捞的宝箱数量。接下来的每一行表示打捞上来的宝箱的深度及所包含的金币数。宝箱的输出顺序要与输入时给出的顺序一致。在两组输入数据的输出之间打印一个空行。

样例输入

```
210 4
3
10 5
10 1
7 2
```

样例输出

```
7
2
10 5
7 2
```

分析

本题实质上是 01 背包问题的直接应用，增加了记录每次最佳选择的步骤。此处背包的容量为潜水的总时间，每件物品的容量为打捞一个宝箱时下潜和上升所消耗的总时间，物品的价值为宝箱包含的金币数。使用前述介绍的解决 01 背包问题的方法即可解决。

强化练习：[562 Dividing Coins^A](#)，[10130 SuperSale^A](#)，[10819 Trouble of 13-Dots^A](#)。

扩展练习：[233 Package Pricing^E](#)，[431* Trial of the Millennium^D](#)，[10072* Bob Laptop Woolmer and Eddie Desktop Barlow^D](#)，[10163* Storage Keepers^C](#)，[11341 Term Strategy^C](#)。

11.1.2 完全背包问题

给定一个容量为 C 的背包，有 n 种物品，每种物品都有容量 C_i 和价值 P_i ， $1 \leq i \leq n$ ，假设每种物品的数量不限，问如何选取放入背包的物品，使得背包内的物品容量之和不超过 C 且价值之和最大。完全背包问题和 01 背包问题非常相似，差别仅在于可以放入的每种物品数量不限，而 01 背包问题中每种物品至多只能放入一件。

完全背包问题可以沿用 01 背包问题的思路予以求解。当处理第 i 种物品时，在背包容量仍有剩余的情

况下, 需要考虑是否仍可放入更多的此类物品, 因此其递推关系为

$$V[i][j] = \max \begin{cases} V[i-1][j] & j < C_i \\ V[i][j - kC_i] + kP_i & kC_i \leq j, 1 \leq k \end{cases}$$

完全背包问题和 01 背包问题递推关系的不同之处在于背包剩余容量的处理: 完全背包问题物品为 i 容量为 j 时的最大值可能来自于第 $(i-1)$ 件物品容量为 j 时的最大值, 或者来自于处理到第 i 件物品且容量为减去物品 i 的 k 倍容量后的背包最大值再加上 k 倍的物品 i 的价值。

强化练习: 10980 Lowest Price in Town^C。

扩展练习: [10645* Menu^D](#), [10448* Unique World^D](#)。

11.1.3 多重背包问题

给定一个容量为 C 的背包, 有 n 种物品, 每种物品都有容量 C_i 和价值 P_i , 且有数量上限 Q_i , $1 \leq i \leq n$, 问如何选取放入背包的物品, 使得背包内的物品容量之和不超过 C 且价值之和最大。

多重背包问题的递推关系可由完全背包的递推关系增加数量上限而来

$$V[i][j] = \max \begin{cases} V[i-1][j] & j < C_i \\ V[i][j - kC_i] + kP_i & kC_i \leq j, 1 \leq k \leq Q_i \end{cases}$$

强化练习: [11566 Let's Yum Cha^D](#)。

11.1.4 背包问题扩展

在深刻理解背包问题递推关系式的基础上, 可以沿用其思维方式解决相应的背包问题扩展。这类问题和背包问题在形式上类似, 在解决思路上亦有相似之处。

子集和问题

子集和问题 (subset-sum problem) 是指给定 n 个非负整数, 确定是否可以从中选择若干个整数, 使其和恰为 S 。如果给定整数中存在负数, 可以将其统一增加一个偏移量, 使得所有整数均为非负整数, 便于问题的处理。根据 n 的大小, 可将子集和问题分为以下几种情况¹:

(1) n 较小, 例如 $n \leq 20$ 。此时可以使用位掩码技巧生成所有可能的子集和从而予以确定。

(2) n 较大, S 较大, 例如 $n \geq 100$, $S \geq 1000000$ 。此时仍然可以尝试使用回溯法解题, 不过需要在回溯过程中进行剪枝, 在回溯过程中一旦当前和已经超过 S , 则此回溯分支可以终止 (因为均是非负整数, 之后的和只可能更大)。

(3) n 较大, S 较小, 例如 $n \geq 200$, $S \leq 10000$ 。对于此种情况, 可以容易地给出其对应的背包问题形式描述: 给定 n 个物品 ($n > 0$), 每个物品的价值 P_i 与其容量 C_i 的比值均为 1, 给定容量为 S 的背包, 确定是否存在一种物品选择方案, 使得背包恰好装满 (此时的背包价值最大)。相对于使用回溯法, 可以借鉴 01 背包问题的思路, 得到更为简洁的解决方案。因为所有整数的和不超过 10000, 那么可能的和只有 0 至 10000 共 10001 种, 根据这个特点, 可以设置一个二维数组 F , 如果元素 $F[i][j]$ 为 1, 表示可以通过前 i 个整数获得和 j , 为 0 则表示无法获得。初始时, 所有整数均不考虑, 此时和为 0, 则 $F[0][0] = 1$ 。假设已经得到了前 $i-1$ 个整数的所有不同和是否能够获取的情况, 此时考虑加入第 i 个整数, 如果前 $i-1$ 个整数能够得到和 j , 那么考虑前 i 个整数时, 很显然, 可以得到和 j ; 如果前 $i-1$ 个整数未能得到和 j , 则检查前 $i-1$ 个整数

¹ 实际上, 子集和问题是 NP 问题, 使用动态规划的方法得到的是伪多项式时间 (pseudo polynomial time) 算法。

能否得到 $j - C_i$ ，如果能够得到，则前 i 个整数能够得到 j ，因此递推关系式为

$$F[i][j] = \max\{F[i-1][j], F[i-1][j - C_i]\}, 1 \leq i \leq n, 1 \leq j \leq S$$

可以参照 01 背包问题优化存储空间的做法，将二维数组 F 缩减为一维数组，与此同时，递推的方向需要做相应的改变。对于此种情形，算法的时间复杂度为 $O(nS)$ 。

```
const int MAXN = 10010, MAXC = 110;

int F[MAXN], C[MAXC], N, S;

// 读入数据并初始化。
N = S = 0;
for (int i = 0; i < N; i++) {
    cin >> C[i];
    S += C[i];
}
memset(F, 0, sizeof(F));
F[0] = 1;

// 根据递归关系进行递推。
for (int i = 0; i < N; i++)
    for (int j = S; j >= C[i]; j--)
        F[j] |= F[j - C[i]];
```

如果对构成和的整数个数加以限制，即从 n 个非负整数中选出不超过 m ($m \leq n$) 个数，使其和恰为 S 。解决思路和基本形式的子集和问题相同，只不过在递推时需要对整数的个数加以限制。令 $F[i][j][k]$ 表示从前 i 个数中选取 j 个数的和为 k ，其递推关系式为

$$F[i][j][k] = \max\{F[i-1][j][k], F[i-1][j-1][k - C_i]\}, 1 \leq i \leq n, 1 \leq j \leq i, C_i \leq k \leq S$$

不难看出，其时间复杂度为 $O(nmS)$ 。更进一步，将构成和的整数个数限定为恰为 m 个数，则递推关系式仍然不变，只不过状态含义发生了变化， $F[i][j][k]$ 表示从前 i 个数中恰好选取 j 个数的和为 k ，其时间复杂度仍为 $O(nmS)$ ¹。

平衡划分问题 (balanced partition problem) 是子集和问题的一种扩展。平衡划分问题是将非负整数集合 I 划分为两个部分 I_1 和 I_2 ，其元素和分别为 S_{I_1} 和 S_{I_2} ，使得 $|S_{I_1} - S_{I_2}|$ 最小。此问题可以通过先确定集合 I 所能构成的所有不同和，令 S_I 表示集合 I 的元素和，从 $S_I/2$ 开始，逐个枚举是否能够达到此和，取最先能够达到的值即为所求。

11997 K Smallest Sums^C (K 最小和)

给定 k 个数组，每个数组包含 k 个整数，总共有 k^k 种方法从每个数组中取出一个数并计算它们的和，你的任务是找出所有和中最小的 k 个和。

输入

输入包含多组测试数据，每组测试数据的第一行包含整数 k ， $2 \leq k \leq 750$ ，接下来的 k 行，每行包含 k 个整数，所有整数均不大于 1000000。文件结束符表示输入结束。

输出

¹ 在后续的集合型动态规划中，可以使用位压缩技巧来降低时间复杂度使之达到 $O(nS)$ 。

对于每组测试数据，按照升序输出最小的 k 个和。

样例输入

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

样例输出

```
9 10 12
2 2
```

分析

可以使用前述介绍的求子集和问题的方法予以解决。由于本题的时间限制较严（UVa OJ 上的时间限制为 1 秒），需要对实现进行优化。由于只是求前 k 项最小和，因此每次在确定子集和时并不需要将所有和求出，只需每次求前 k 项即可。具体方法如下：在求和之前将数组按升序排列，假设当前已经求得前 i 个数组的最小的 k 个和，存放在数组 sum 中，现在对第 j 个数组进行求和操作，对于第 j 个数组中的某个数 x ，如果 x 与 $sum[0]$ 的和大于等于 $sum[k-1]$ ，显然没有必要再对后续的数进行求和，在随后的求和过程中，保持数组 sum 一直处于有序状态，可以利用前述性质来避免不必要的更新，提高效率。保持数组 sum 有序可以通过插入排序来实现。

另外还有一种更为巧妙地利用优先队列的解题方法。考虑前两个数组，令其为 A 和 B ，将所有数组元素按升序排列，第 1 个数组的元素为 A_1 至 A_k ，第 2 个数组的元素为 B_1 至 B_k ，有

$$\begin{aligned} A_1 &\leq A_2 \leq \dots \leq A_k \\ B_1 &\leq B_2 \leq \dots \leq B_k \end{aligned}$$

枚举这两个数组中元素的所有和，可以得到

$$\begin{aligned} A_1 + B_1 &\leq A_1 + B_2 \leq \dots \leq A_1 + B_k \\ A_2 + B_1 &\leq A_2 + B_2 \leq \dots \leq A_2 + B_k \\ &\dots \\ A_k + B_1 &\leq A_k + B_2 \leq \dots \leq A_k + B_k \end{aligned}$$

由于数组 A 和 B 都是按升序排列，则对于同一列的和来说，从上到下满足小于等于关系（例如： $A_1 + B_1 \leq A_2 + B_1 \leq \dots \leq A_k + B_1$ ），即 $A_1 + B_1$ 是最小的和，紧接着比 $A_1 + B_1$ 稍大的只可能是 $A_1 + B_2$ 和 $A_2 + B_1$ 中的某一个，也就是说，一旦确定了 $A_i + B_j$ 是较小的，后续紧接最小的值是 $A_i + B_{j+1}$ 和 $A_{i+1} + B_j$ 中的一个。因此可以将当前最小值以及获得当前最小值的下标信息同时记录，使用优先队列来获取当前最小值，同时得到最小值的下标信息，通过下标信息来获取后续紧邻的最小值，同时记录最小值和下标信息，并再次送入优先队列，持续上述过程，直到找到两个数组的前 k 个最小和。按照类似办法，先合并第 1 个和第 2 个数组得到最小的 k 个和，然后使用这 k 个和与第 3 个数组进行合并， \dots ，直到与第 k 个数组合并即为最终结果。

强化练习：[435* Block Voting^B](#)，[10036 Divisibility^A](#)，[10400 Game Show Math^B](#)，[10616 Divisible Group Sums^A](#)，[10664 Luggage^A](#)，[10930 A-Sequence^B](#)，[11658 Best Coalitions^D](#)，[11780 Miles 2 Km^C](#)，[12455 Bars^A](#)，[12563 Jin Ge Jin Qu Hao^C](#)，[12621 On a Diet^D](#)。

扩展练习：[242 Stamps and Envelope Size^D](#)，[430* Swamp County Supervisors^D](#)，[10690 Expression Again^D](#)，[11331 The Joys of Farming^D](#)，[12911* Subset Sum^D](#)。

计数问题

背包问题的另外一种形式是给定若干组整数，从每组整数中取一个整数，求能够获得的“不同和”的种

数 (或者获得指定和的不同方法总数), 称为计数问题。可以看到, 这类问题和后续介绍的兑换问题有相似之处, 但又不完全相同。解决此类问题的关键仍然是得出递推关系式, 如果递推关系式不易求得, 可使用后续介绍的备忘技巧, 借助于自上而下的递归予以解决。

强化练习: [10759 Dice Throwing^B](#), [10910 Marks Distribution^B](#), [10943 How Do You Add^A](#), [11450 Wedding Shopping^A](#)。

扩展练习: [10238 Throw the Dice^D](#), [12063* Zeros and Ones^D](#)。

11.2 备忘

动态规划和分治法 (divide and conquer) 有相似之处, 即都是通过合并子问题的解得到整个问题的解。分治法是将整个问题划分为一些独立的子问题后递归地进行求解, 这些子问题相互之间是不重叠的。动态规划适用于处理子问题重叠的情形, 在求解过程中, 对同样的子问题只进行一次求解, 然后将结果保存在一张表中, 在后续遇到同样的子问题时只需查表即可, 不需要重新计算。这种技巧称为备忘 (memoization), 有的也称为表格式动态规划。子问题既独立又相互重叠, 初看起来是矛盾的, 其实这描述的是问题的两个方面。在对动态规划问题使用自顶向下 (或自底向上) 的方法计算时, 备忘技巧使用得更为频繁。

11.2.1 $3n+1$ 问题

[100 The \$3n+1\$ Problem^A](#) ($3n+1$ 问题)

考虑如下算法:

```

1. input n
2. print n
3. if n = 1 then STOP
4. if n is odd then n = 3n + 1
5. else n = n/2
6. GOTO 2

```

给定输入 $n=22$, 上述算法会产生以下输出:

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

存在以下猜想: 对于给定的任意正整数, 按照上述算法执行, 最后都会终止于 1。尽管算法非常简单, 但是对于所有整数来说, 猜想是否成立尚未得到证明。曾经有人验证, 对于所有整数 n ($0 < n < 1000000$), 猜想均成立。

给定输入 n , 可以统计在执行上述算法过程中输出的整数的个数 (包括最后的 1), 将此个数称为 n 的循环节长度 (cycle-length), 以 $n=22$ 为例, 其循环节长度为 16。

给定任意两个整数 i 和 j , 确定在 i 和 j 之间 (包括 i 和 j) 的所有整数的最大循环节长度。

输入

输入包含多组测试数据, 每组测试数据一行, 包含两个整数 i 和 j 。所有整数均小于 1000000。

输出

对于每组数据输出一行, 包含三个整数, 分别为 i , j 以及在 i 和 j 之间 (包括 i 和 j) 的所有整数的最大循环节长度。

样例输入

样例输出

```
1 10
100 200
201 210
900 1000
```

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

分析

可以将题意重新表述如下：给定一个整数 n ，如果 n 是偶数，则把它除以 2，如果 n 是奇数，把它乘以 3 后加 1，重复这个过程，直到 $n=1$ （这里假定都为正整数，对于负整数同样有效，只不过最后结果收敛到 -1 ）。在得到 1 的过程中会生成一个序列，例如 $n=22$ 时该过程生成的序列为：22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1。对于给定的 n ，该序列的元素个数称为 n 的循环节长度。上述示例中，22 的循环节长度为 16。给定某个整数区间，例如 $[0, 1000000)$ ，要求你确定该区间内整数的循环节长度的最大值^I。

采用最简单的模拟算法，使用一个循环即可计算循环节长度：

```
// steps 表示整数 n 的循环节长度。
steps = 1;
while (n > 1) {
    steps++;
    if (n & 1) n = 3 * n + 1;
    else n = n >> 1;
}
```

但是使用这种方法解题容易出现超时错误^{II}。分析一下整数 22 的循环节序列可知，其循环节长度为整数 11 的循环节长度加 1，如果已经知道了整数 11 的循环节长度，则很容易计算整数 22 的循环节长度。令整数 n 的循环节长度为 $L(n)$ ，则有

$$L(1) = 1, \quad L(n) = 1 + L(x), \quad x = \begin{cases} \frac{n}{2} & \text{若 } n \text{ 为偶数} \\ 3n + 1 & \text{若 } n \text{ 为奇数} \end{cases}, \quad n > 1$$

那么可以使用递归来计算循环节长度：

```
int getCycle(long long n) {
    if (n == 1) return 1;
    if (n & 1) return 1 + getCycle(3 * n + 1);
    return 1 + getCycle(n >> 1);
}
```

但是仅有递归仍然是超时的，因为没有解决重复计算的问题。例如，在计算整数 22 的循环节过程中，仍然会将整数 11 的循环节计算过程重复一遍，就像使用递归计算斐波那契数列一样，中间存在较多的重复计算。可以使用保存中间结果的方法来避免重复计算，以 22 的循环节计算过程为例，其循环节构成为：

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

观察循环节构成易知，如果在计算 22 的循环节过程中，保存了 17 的循环节长度，则计算 34 的循环节长度时，可直接利用 17 的循环节长度，将其加 1 即可得到 34 的循环节长度。那么如何标记某个整数的循环节长

^I $3n+1$ 问题的证明曾经是克雷数学研究所 (<http://www.claymath.org/>, 2020) 百万美元待解决问题之一。

^{II} 截至 2020 年 1 月 1 日，该题目的输入约束已经发生更改，缩小了输入数据的范围，从而降低了问题的难度。输入中所有整数均大于 0 且小于 10000，因此使用模拟算法即可获得 Accepted。

度已经计算并且能够方便地取出其值呢？可以通过设立一个中间结果保存数组 *cache* 来实现，其下标为 *n* 的元素值 *cache[n]* 表示整数 *n* 的循环节长度，初始时数组的所有元素值为 0。在计算循环节的递归过程中，检查下标为 *n* 的数组元素其值是否为 0，如果不为 0 则表示整数 *n* 的循环节已经计算，直接取用其循环节长度 *cache[n]*，否则继续递归求解，并将求解所得的值保存到中间结果数组 *cache* 中^I。

```
const int MAXV = 1000000;

int cache[MAXV];

int getCycle(long long n) {
    // 递归出口。
    if (n == 1) return 1;
    // 执行除以 2 或者乘以 3 后加 1 的操作。
    n = (n & 1) ? (n + (n << 1) + 1) : (n >> 1);
    // 检查整数 n 的值是否已经定义，为否则计算循环节长度并存储到数组中。
    if (n < MAXV) {
        if (!cache[n]) cache[n] = getCycle(n);
        return 1 + cache[n];
    }
    return 1 + getCycle(n);
}
```

强化练习: 371 Ackermann Functions^A, 547 DDF^C, 694 The Collatz Sequence^A, 944 Happy Numbers^C, 10446 The Marriage Interview^B, 10520 Determine It^C, [10651 Pebble Solitaire](#)^A, 10696 f91^A, [10721 Bar Codes](#)^A, [10912 Simple Minded Hashing](#)^B, 11703 sqrt log sin^B。

扩展练习: 249 Bang the Drum Slowly^E, 1261 String Popping^D, 11226 Reaching the Fix-Point^C。

11.2.2 正交范围查询

给定如下图所示的 01 数值矩阵，试回答如下形式的查询：给定矩阵中的一个子矩形，矩形内元素的和

^I (1) 此题的中间计算过程会超出 int 或 long int（如果 int 或 long int 均为 4 字节存储空间）类型数据的表示范围，故需要选择 long long int（8 字节存储空间）类型整数（除非你在做乘法运算时不使用内置的乘法，而是使用替代方法实现原数的三倍加一）；(2) 给定的输入，可能较大的数在前面，较小的数在后面，在输出时需要调整顺序，这个是导致算法正确却 Wrong Answer 的一个重要原因。网络上的解题报告，大多数都忽略了第 (1) 点，在求循环节长度的过程中，选择了 int 或 long int（按 32 位 CPU 来假定，4 字节存储空间）类型的数据，当计算 $(n \times 3 + 1)$ 时会超出 32 位整数的表示范围而得到错误答案，只不过 Programming Challenges 和 UVA OJ 上的测试数据不是很强，所以尽管不完善但都会获得 Accepted。在 1 至 999999 之间共有 41 个数在中间计算过程中会得到大于 32 位无符号整数表示范围的整数，当测试数据包含这些数时，选用 int 或 long int 类型很有可能会得到错误的答案。在中间计算过程中会超过 32 位整数表示范围的整数为（括号内为该数的循环节长度）：159487(184), 270271(407), 318975(185), 376831(330), 419839(162), 420351(242), 459759(214), 626331(509), 655359(292), 656415(292), 665215(442), 687871(380), 704511(243), 704623(504), 717695(181), 730559(380), 736447(194), 747291(248), 753663(331), 763675(318), 780391(331), 807407(176), 822139(344), 829087(194), 833775(357), 839679(163), 840703(243), 847871(326), 859135(313), 901119(251), 906175(445), 917161(383), 920559(308), 937599(339), 944639(158), 945791(238), 974079(383), 975015(321), 983039(290), 984623(290), 997823(440)。

是多少？

0	0	0	0	1	0	1	1
0	1	1	0	1	0	1	0
1	1	0	0	1	1	0	1
0	1	1	1	0	0	1	0
0	1	1	0	1	0	0	0
0	1	0	1	1	1	0	0
0	0	1	1	1	1	0	1
1	0	0	1	0	0	1	1

(a)

0	0	0	0	1	1	2	3
0	1	2	2	4	4	6	7
1	3	4	4	7	8	10	12
1	4	6	7	10	11	14	16
1	5	8	9	13	14	17	19
1	6	9	11	16	18	21	23
1	6	10	13	19	22	25	28
2	7	11	15	21	24	28	32

(b)

图 11-1 (a) 01 数值矩阵, (b) 优势矩阵

由于任何边平行于坐标轴的矩形都可以由左上角(x_l, y_l)和右下角(x_r, y_r)两个点确定（**假定 X 轴水平向右为正, Y 轴垂直向下为正**），朴素的算法是使用嵌套循环把满足 $x_l \leq i \leq x_r, y_l \leq j \leq y_r$ 的所有元素 $m[i][j]$ 相加，不过这样做效率不高，特别是在需要反复进行类似查询的场合。例如，在矩阵元素频繁更改的情况下，要求你确定给定矩阵中具有最大或最小元素和的子矩阵。

正交范围查询 (orthogonal range queries) 是对 $m \times n$ 矩阵网格进行的一种常用操作，它可以用来解决上述问题。该方法是构建一个新的矩阵，称之为优势矩阵 (dominance matrix)，该矩阵中的元素 $d[x][y]$ 表示下标满足 $0 \leq i \leq x$ 且 $0 \leq j \leq y$ 的矩阵元素 $m[i][j]$ 的和，如图 11-1 (b) 所示。

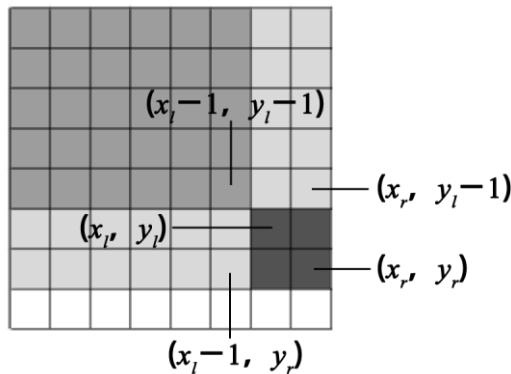


图 11-2 由容斥原理得到子矩形(x_l, y_l, x_r, y_r)的元素和

根据图 11-2 子矩形的关系以及容斥原理，容易得到

$$s(x_l, y_l, x_r, y_r) = d[x_r][y_r] - d[x_l-1][y_r] - d[x_r][y_l-1] + d[x_l-1][y_l-1]$$

那么如何构造优势矩阵呢？由容斥原理，根据同样的思路，可以得到以下递推关系

$$d[x][y] = d[x-1][y] + d[x][y-1] - d[x-1][y-1] + m[x][y]$$

在计算优势矩阵时，可以用行优先顺序填充优势矩阵，**其时间复杂度为 $O(RC)$** ， R 为矩阵的行数， C 为矩阵的列数。

```
const int MAXN = 1000;
```

```

int m[MAXN][MAXN], d[MAXN][MAXN], n;
memset(d, 0, sizeof(d));
// 从 1 开始计数矩阵中存储元素的下标, 这样便于计算。
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j++)
        d[i][j] = d[i][j - 1] + d[i - 1][j] - d[i - 1][j - 1] + m[i][j];

```

强化练习: [108 Maximum Sum^A](#), [836 Largest Submatrix^A](#), [983 Localized Summing for Blurring^C](#), [10502 Counting Rectangles^B](#), [10827 Maximum Sum on a Torus^A](#), [10908 Largest Square^A](#), [11951 Area^C](#)。

11.2.3 最大正方形 (长方形)

给定一个宽为 W , 高为 H 的矩形网格, 黑色方格已被占用, 白色方格未被占用, 要求你计算由白色方格所能构成的最大正方形的边长。

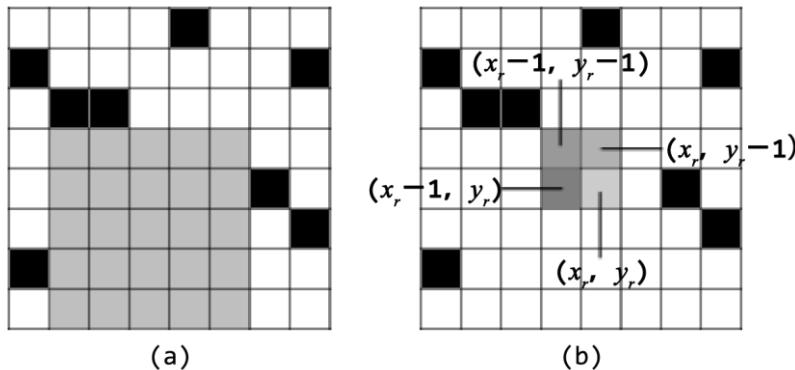


图 11-3 白色方格所能构成的最大正方形。(a) 图中由白色方格构成的最大正方形的边长为 5, 即图中灰色的正方形。(b) 以某个白色方格为右下角的最大正方形与左方、上方、左上方最大正方形边长之间的关系

令 $S(x_r, y_r)$ 表示以白色方格 (x_r, y_r) 为右下角的最大正方形的边长, 观察图 11-3 (b) 可知, 有以下递推关系

$$S(x_r, y_r) = \min\{S(x_r - 1, y_r), S(x_r, y_r - 1), S(x_r - 1, y_r - 1)\} + 1$$

强化练习: [559 Square \(II\)^D](#), [585 Triangles^C](#)。

但是对于求由白色方格构成的最大长方形来说, 却并不具有前述简单的递推关系。当给定的数据范围较小时, 可以使用穷尽搜索, 依次以某个白色方格为长方形的左上角, 枚举可能的右下角, 在此过程中记录能够得到的长方形的面积, 从而获取具有最大面积的长方形。

更为高效的方法是将问题进行适当转换, 然后再予以解决^[138]。首先将给定的网格按照“黑色方格的值为 0, 白色方格的值为 1”的规则将其转换为一个 01 矩阵, 然后在 01 矩阵中计数各矩阵元素向上存在多少个连续的 1, 通过对各列使用简单的动态规划就可以很方便地求出(如果某个方格内的数值为 1, 则向上连续 1 的个数为此方格正上方的方格的相应计数加 1, 否则当前方格的计数为 0)。把网格的每一行都看成一个直方图, 问题就转化为求直方图内最大长方形的问题。

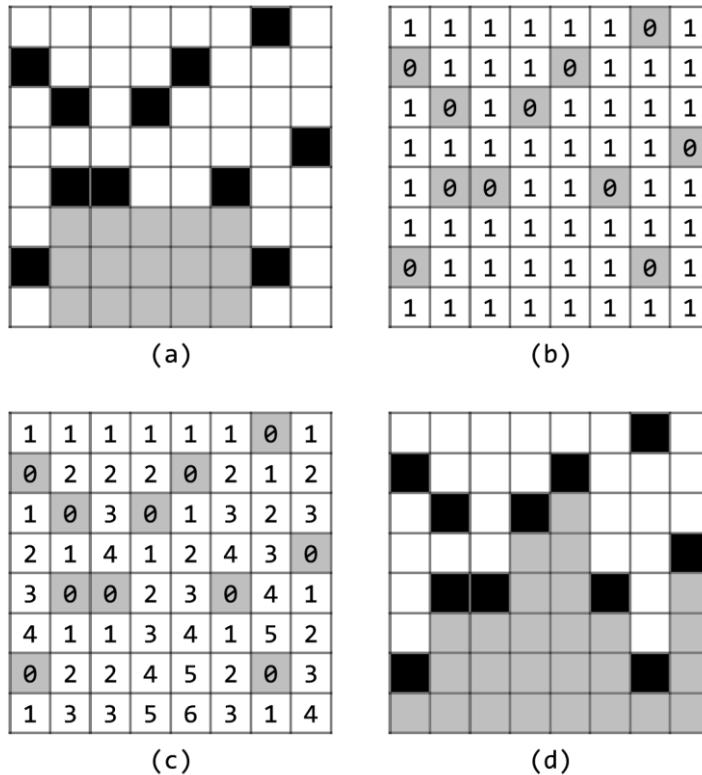


图 11-4 (a) 图中由白色方格构成的最大的长方形面积为 15, 即图中灰色的长方形。(b) 将黑色方格置为 0, 白色方格置为 1。(c) 统计各个方格向上连续白色方格的数量, 形成直方图。(d) 寻找最大长方形相当于在每行的直方图中寻找最大可能的长方形

观察图 11-4 中的直方图, 对于同一行的直方图, 一旦右侧某列直方图的高度小于左侧一列直方图的高度, 就可以确定左侧某些长方形的面积。为了更高效地求解, 可以使用栈来记录局部问题的解。假设栈中记录的是“仍有可能扩张的长方形的信息(记为 *rect*)”, *rect* 记录有两个信息, 一个是长方形的高 *height*, 另一个是其左端的位置 *left*。首先将栈置为空, 接下来对于直方图的各个值 H_i , $i=0, 1, \dots, W-1$, 创建以 H_i 为高, 以其下标 i 为左端位置的长方形 *rect*, 然后进行以下处理:

- 1) 如果栈为空, 将 *rect* 压入栈。
- 2) 如果栈顶长方形的高小于 *rect* 的高, 将 *rect* 压入栈。
- 3) 如果栈顶长方形的高等于 *rect* 的高, 不做处理。
- 4) 如果栈顶长方形的高大于 *rect* 的高:
 - 4a) 只要栈不为空, 且栈顶长方形的高大于等于 *rect* 的高, 就从栈中取出长方形, 同时计算其面积并更新最大值。长方形的长等于当前位置 i 与之前记录的左端位置 *left* 的差值;
 - 4b) 将 *rect* 压入栈, 在压入栈之前, 将 *rect* 的左端位置 *left* 修改为最后从栈中取出的长方形的 *left* 值。

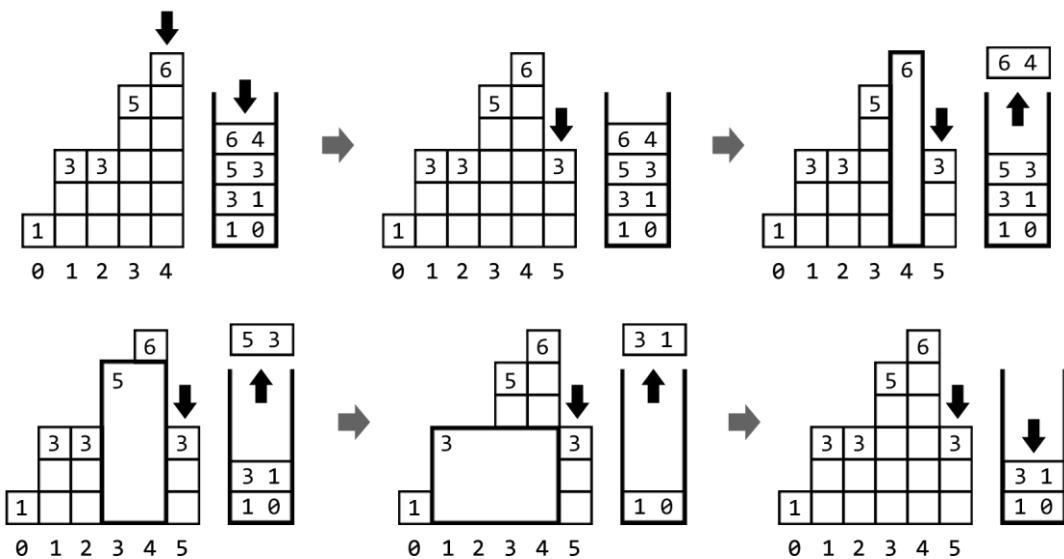


图 11-5 借助栈检测最大长方形

需要注意，在使用栈查找最大长方形的过程中，如果当前处理的矩形高度与栈中的矩形高度相等，此时获取的长方形还不是最大的长方形，按上述算法处理时会将高度相等的矩形重新压入栈中，需要等待下一次遇到高度小于栈顶矩形高度的直方图时，此时计算得到的长方形才是更大的长方形（如图 11-5 中计算高度为 3 的长方形）。除此之外，还会遇到这样的特殊情形——栈中存储的矩形的高度都是递增的，由于未遇到高度小于栈顶的矩形，在处理到最后一列时，栈中的元素均未被处理。为了能够正确应对以上特殊情形，在实现时可以在每行末尾添加一个高度为 0 的直方图，这样就可以保证面积大于 0 的长方形都会找出，从而确保结果的正确性。

```
//-----11.2.3.cpp-----//
struct rectangle { int height, left; };
int m, n, matrix[110][110];

int getMaxArea() {
    int area = 0;
    stack<rectangle> s;
    for (int i = 0; i < m; i++) {
        // 在每行末尾添加一个高度为 0 的直方图，保证结果的正确性。
        matrix[i][n] = 0;
        for (int j = 0; j <= n; j++) {
            rectangle rect = rectangle{matrix[i][j], j};
            if (s.empty() || s.top().height < rect.height) s.push(rect);
            else {
                if (s.top().height > rect.height) {
                    int last = j;
                    while (!s.empty() && s.top().height >= rect.height) {
                        rectangle previous = s.top(); s.pop();
                        area = max(area, previous.height * (j - previous.left));
                        last = previous.left;
                    }
                    rect.left = last;
                }
            }
        }
    }
}
```

```

        s.push(rect);
    }
}
return area;
}
//-----11.2.3.cpp-----//

```

强化练习: 1330 City Game^D, 10074 Take the Land^A, 10667 Largest Block^B。

11.2.4 整数划分

给定正整数 n , 将其表示成若干正整数之和的形式, 即

$$n = m_1 + m_2 + \cdots + m_i, \quad 1 \leq m_i \leq n$$

称 $\{m_1, m_2, \dots, m_i\}$ 为 n 的一个划分, 而整数划分 (integer partition) 所求的就是 n 的不同划分的方法个数。需要注意, 在整数划分中, 将两个加数更改顺序不属于一种新的划分方法。例如当 $n=4$ 时, 只有 5 种划分: $\{4\}$ 、 $\{3, 1\}$ 、 $\{2, 2\}$ 、 $\{2, 1, 1\}$ 、 $\{1, 1, 1, 1\}$, 其中 $\{3, 1\}$ 和 $\{1, 3\}$ 同属一种划分。如果某个划分中 $m_i \leq m$, 则称该划分属于一个 n 的 m 划分。为了方便讨论, 记 $f(n, m)$ 为 n 的 m 划分个数。根据 n 和 m 的关系, 可以将其区分为以下若干情形:

- (1) 当 $n=1$ 时, 只有一种划分, 即 $\{1\}$;
- (2) 当 $m=1$ 时, 只有一种划分, 即 $\{1, 1, \dots, 1\}$, 共 n 个 1;
- (3) 当 $n=m$ 时, 根据划分中是否包含 n , 可以分为两种情况:
 - (3a) 划分中包含 n , 只有一个划分, 即 $\{n\}$;
 - (3b) 划分中不包含 n , 此时划分中最大的数一定比 n 小, 即 n 的所有 $(n-1)$ 划分; 因此 $f(n, n) = 1 + f(n, n-1)$;
- (4) 当 $n < m$ 时, 由于划分中不能出现负数, 因此就相当于 $f(n, n)$;
- (5) 当 $n > m$ 时, 根据划分中是否包含最大值 m , 可以分为两种情况:
 - (5a) 划分中包含 m , 即 $\{m, \{x_1, x_2, \dots, x_i\}\}$, 其中 $\{x_1, x_2, \dots, x_i\}$ 的和为 $(n-m)$, 可能再次出现 m , 因此是 $(n-m)$ 的 m 划分, 所以这种划分个数为 $f(n-m, m)$;
 - (5b) 划分中不包含 m , 则划分中所有值都比 m 小, 即 n 的 $(m-1)$ 划分, 个数为 $f(n, m-1)$, 因此 $f(n, m) = f(n-m, m) + f(n, m-1)$ 。

综上所述, n 的 m 划分个数可以归纳为

$$f(n, m) = \begin{cases} 1 & n = 1 \text{ 或者 } m = 1 \\ f(n, n) & n < m \\ 1 + f(n, n-1) & n = m \\ f(n-m, m) + f(n, m-1) & n > m \end{cases}$$

在编码实现时, 为了避免重复计算, 可以使用备忘技巧记录已经求解的划分数 (需要注意, 在求解前应当将表格的元素初始化为 0 值以便区分检索)。

```

//-----11.2.4.cpp-----//
const int MAXN = 1024;

unsigned long long dp[MAXN][MAXN];

```

```

unsigned long long ip(int n, int m) {
    if (dp[n][m]) return dp[n][m];
    if (n == 1 || m == 1) dp[n][m] = 1;
    else if (n < m) dp[n][m] = ip(n, n);
    else if (n == m) dp[n][m] = 1 + ip(n, m - 1);
    else dp[n][m] = ip(n - m, m) + ip(n, m - 1);
    return dp[n][m];
}
//-----11.2.4.cpp-----

```

11.2.5 博弈树

博弈树 (game tree)，又称决策树 (decision tree)，是指在游戏过程中，由玩家的选择产生的不同后继局面状态所构成的一棵局面状态树。树中的每个结点都表示了游戏过程中的某个状态和此时需要执行操作的玩家，而根结点则表示初始游戏状态和先手玩家。当玩家执行某种游戏操作后，后继局面可能不止两种，因此是一棵多叉树。树中内部结点的子结点表示玩家从当前局面可行的走子选择中选定一种后所得到的局面状态 (与此同时，当前玩家变换为对方玩家)。

从图论的角度来看，由于树本身也是一种图，因此可以将博弈树视为结点对应的是游戏状态的有向图，图中的有向边对应于游戏的某个操作，整棵树包含了游戏的初始状态和所有可达状态。在博弈树中，叶子结点对应没有后继走法的游戏局面，在此种状态下，游戏双方均无可行的游戏操作，因此也称为终止状态。终止状态表示游戏的结束，在此状态下，可能有一方为胜者，或者为平局。

对于无偏博弈来说，由于其可能的游戏状态是有限的，在经过有限数量的游戏步骤之后必定会产生赢家，因此可以使用回溯法 (结合备忘技巧) 遍历所有可能的后继状态，然后确定给定初始状态是否能够获胜。为了便于讨论，此处定义了必胜态和必败态的概念。**如果某个状态至少有一个后继状态能够保证己方玩家必胜，则称此状态为必胜态；如果某个状态的所有后继状态都能使得对方玩家必胜，则称此状态为必败态。**如果延续之前的“PN 态”定义¹，将必胜态称为 N 态，必败态称为 P 态，可以使用下述的框架来判断某个状态是否具有必胜策略。

```

// MAXN 表示最多可能具有的状态数。
const int MAXN = 1 << 20;

// 数组 dp 记录每种状态是否具有必胜策略。
// 如果 dp[x] 为 0 表示状态 x 不存在必胜策略，为 1 表示状态 x 存在必胜策略。
int dp[MAXN];

// 初始时所有状态的必胜策略设置为 -1，表示状态不确定。
// memset(dp, -1, sizeof(dp));

// 根据题目约束确定游戏最终状态是否具有必胜策略。
// dp[0] = 0;

// 使用回溯法，结合备忘技巧遍历所有可能的状态。
int dfs(int x) {
    // 使用备忘技巧，避免重复搜索状态。
    if (~dp[x]) return dp[x];

```

¹ 参见本书第 6 章“组合数学”第 6.9.4 小节“PN 态分析”的内容。

```

// 对于当前状态 x 的每个后继状态 y, 检查其是否为 P 态, 如果是, 则状态 x 为 N 态,
// 即状态 x 具有必胜策略。
for each y in successor(x)
    if (!dfs(y))
        return dp[x] = 1;
// 状态 x 的所有后继状态均为 N 态, 则状态 x 为 P 态, 即状态 x 不具有必胜策略。
return dp[x] = 0;
}

```

对于此种类型的题目, 解题的关键在于确定如何表示状态和根据规则得到当前状态的后继。题目设置者常常在状态的转移环节“做文章”, 倾向于将状态转移规则设置得较为复杂, 使得解题者容易出错。

10111 Find the Winning Move^c (寻找胜着)

4×4 的 Tic-Tac-Toe 游戏是在四行 (从上到下行号为 0 至 3) 四列 (从左到右列号为 0 至 3) 的棋盘上进行。有两名玩家 A 和 B , 分别执 ‘X’ 和 ‘O’, A 和 B 轮流走子, 玩家 A 先行。如果某位玩家先于对方将己方的四个棋子沿着同一行、或同一列、或同一对角线连成直线则获胜。如果棋盘上已经布满棋子, 但是仍然没有玩家获胜则为平局。

假如当前为执 ‘X’ 的玩家 A 行棋, 如果玩家 A 选择某个位置落子后, 无论玩家 B 采取何种落子选择, 玩家 A 都能够获胜, 则称玩家 A 具有必胜策略。玩家 A 具有必胜策略并不意味着后续游戏中玩家 A 可以任意着子, 只是说无论玩家 B 采取某种落子选择, 玩家 A 都能够通过选择相应的落子位置以保证最终的胜利。

给定一个已经部分完成的棋盘状态, 接下来是玩家 A 行棋。编写程序, 确定玩家 A 是否具有必胜策略。你可以假定在给定的棋盘状态中, 每名玩家至少各自已经走了两步, 尚无玩家获胜且棋盘未满。

输入

输入包含多组测试数据, 输入最后一行以美元符号 ‘\$’ 结束。每组测试数据以问号 ‘?’ 开始, 接着 4 行表示棋盘状态, 每行包含 4 个字符, 其中点号 ‘.’ 表示空白位置, 小写字母 ‘x’ 表示玩家 A 的落子, 小写字母 ‘o’ 表示玩家 B 的落子。

输出

对于每组测试数据, 输出一行。如果玩家 A 具有必胜策略, 则输出第一个必胜的落子位置, 以行号、列号的方式输出, 如果玩家 A 无必胜策略, 则输出 “#####”, 具体输出格式参考样例输出。

对于本题来说, 如果玩家 A 有必胜策略, 要求输出的是第一个能够保证获得最终胜利的落子位置, 并不是要求输出获得胜利所需要的着子步数。以行优先顺序逐次检查玩家 A 在位置(0, 0), (0, 1), (0, 2), (0, 3), (1, 0), (1, 1), ..., (3, 2), (3, 3)落子后 (如果该位置可以落子) 能否取得必胜, 如果能, 则输出第一个能取得必胜的位置。在样例输入的第二组测试数据中, 玩家 A 可以在位置(0, 3)或者(2, 0)落子并立即获胜, 但是在位置(0, 1)落子仍能够保证最终获胜 (尽管这不必要地将胜利延迟), 因此位置(0, 1)是第一个能够取得必胜的落子位置。

样例输入

```

?
....
.XO.
.OX.

```

样例输出

```

#####
(0,1)

```

```
....  
?  
O...  
.OX.  
.XXX  
XOOO  
$
```

分析

棋盘为 4×4 共 16 个方格，每个方格有 3 种状态，无棋子、已放置 X 棋子、已放置 O 棋子，则总共有 $3^{16} = 43046721$ 种状态，可以考虑使用三进制数来表示状态，使用一个全局整数数组（或者使用位压缩）表示三进制数所对应的必胜态或必败态。另外一种表示状态的方法是将 X 棋子和 O 棋子的状态用 32 位整数的高 16 位和低 16 位分别表示。由于需要寻找所有可能的初始获胜位置，故需要枚举每一个可以放置棋子的位置，检查其是否可能为必胜态。

参考代码

```
const int X = 0, O = 1, X_WIN = 0, X_LOSE = 1, O_WIN = 2, O_LOSE = 3;

// 将棋盘的可能获胜情形编码为位掩码。
string bits[] = {
    "1111000000000000", "0000111000000000", "0000000011110000",
    "0000000000001111", "1000100010001000", "0100010001000100",
    "0010001000100010", "0001000100010001", "1000010000100001",
    "0001001001001000"
};

int wins[10], mask = 0xffff;
int board, empty[16], used[16] = {}, tot = 0, cnt = 0;

// 使用集合存储必胜态和必败态。
set<int> Ns, Ps;

// 判断当前棋盘状态是否包含表示获胜的位掩码。
inline bool isWin(int key) {
    for (int i = 0; i < 10; i++)
        if ((key & wins[i]) == wins[i])
            return true;
    return false;
}

// 结合备忘技巧，使用递归来确定某个状态是 P 态还是 N 态。
int dfs(int player) {
    // 使用备忘技巧检查当前状态是否已经访问，若已访问则直接返回结果。
    if (player == X) {
        if (Ns.find(board) != Ns.end()) return X_WIN;
        if (Ps.find(board) != Ps.end()) return X_LOSE;
    }
    // 若当前状态不在 P 态或 N 态集合中，检查其是否包含获胜的情形。
    if (player == X && isWin(board & mask)) return X_LOSE;
    if (player == O && isWin(board >> 16)) return O_LOSE;
    // 枚举尚未落子的空白位置。
    for (int i = 0; i < tot; i++)
```

```

if (!used[i]) {
    used[i] = 1;
    if (player == X) board |= (1 << (16 + empty[i]));
    else board |= (1 << empty[i]);
    // 回溯, 对方玩家行棋。
    int next = dfs(1 - player);
    // 更新棋盘状态。
    if (player == X) board ^= (1 << (16 + empty[i]));
    else board ^= (1 << empty[i]);
    used[i] = 0;
    // 记录玩家 A 的必胜态。
    if (player == X && next == O_LOSE) {
        Ns.insert(board);
        return X_WIN;
    }
    if (player == O && next == X_LOSE) return O_WIN;
}
// 记录玩家 A 的必败态。
if (player == X) { Ps.insert(board); return X_LOSE; }
else return O_LOSE;
}

int main(int argc, char *argv[]) {
    // 将表示获胜情形的棋盘状态转换为位掩码。
    for (int i = 0; i < 10; i++) {
        bitset<16> binary(bits[i]);
        wins[i] = (int)(binary.to_ulong());
    }
    // 读入棋盘状态, 记录可以落子的位置。
    char piece;
    while (cin >> piece, piece != '$') {
        if (piece != '?') continue;
        board = cnt = tot = 0;
        while (cnt < 16) {
            cin >> piece;
            if (piece != '.' && piece != 'x' && piece != 'o') continue;
            if (piece == '.') empty[tot++] = cnt;
            else if (piece == 'x') board |= (1 << (16 + cnt));
            else board |= (1 << cnt);
            cnt++;
        }
        // 检查每一个可以落子的位置, 如果可以获得必胜则输出。
        bool flag = false;
        memset(used, 0, sizeof(used));
        for (int i = 0; i < tot; i++) {
            used[i] = 1;
            board |= (1 << (16 + empty[i]));
            if (dfs(0) == O_LOSE) {
                cout << '(' << empty[i] / 4 << ',' << empty[i] % 4 << ")\\n";
                flag = true;
                break;
            }
            board ^= (1 << (16 + empty[i]));
            used[i] = 0;
        }
        if (!flag) cout << "#####\\n";
    }
}

```

```

    }
    return 0;
}

```

强化练习: 682 Whoever Pick The Last One Lose^D, [10536* Game of Euler^C](#), 10578 The Game of 31^C, [11311* Exclusively Edible^C](#), [12846 A Daisy Puzzle Game^E](#)。

扩展练习: 751* Triangle War^D, 1557 Calendar Game^E, 1558 Number Game^E, 1559 Nim^E, 1561* Cycle Game^E, [11587* Brick Game^D](#)。

11.2.6 备忘与递推

在一般的动态规划问题中, 如果状态之间的转移关系容易得出, 那么使用递推方式进行求解是比较容易的, 在此种情形下, 递推相当于以一种自底向上 (bottom-up) 的方式进行解题。与之相对应的备忘式动态规划, 则类似于一种自顶向下 (top-down) 的解题方式。这两种方式有以下的异同点:

(1) 两者都是以表格的形式来存储中间计算结果以避免重复计算, 进而提高效率。递推方式解题是从解决基础的子问题开始, 逐步将多个“较小”的子问题的解合并为“较大”的子问题的解, 并将其填写到表格中。备忘式动态规划则递归地将“较大”的未知问题分解为“较小”的未知问题, 直到这些“较小”的未知问题足够小以至能够容易地进行求解。备忘式动态规划在分解并解决问题的过程中, 将结果保存在备忘录 (实际上也可以看做是一张表格) 中以避免重复计算。

(2) 对于每个可能的子问题都需要解决的动态规划问题, 使用递推的方式效率较高。因为递推会将所有的子问题都解决一遍, 不会产生遗漏, 而填表的方式能够确保不会发生重复。如果动态规划中的每个可能的子问题不一定都会遇到, 则使用递归方式解决效率较高, 这样对于不会出现的子问题不会予以解决, 可以节省一定的时间。

(3) 从控制结构上看, 递推式动态规划使用循环迭代方式对子问题的解决顺序进行控制, 而备忘式动态规划则与回溯法的整体结构类似, 不过其中最大的一点不同在于——典型回溯法中, 某个状态只会访问一次, 而动态规划中, 某个状态会多次访问。显然, 如果每个状态在访问时都要进行计算, 时间消耗会明显增加, 而运用备忘技巧将已访问状态的计算结果予以保存, 在下次访问到该状态时直接返回结果, 这样可以避免重复计算, 从而显著提高程序的运行效率。

(4) 从理论上来讲, 使用备忘方式能够解决的题目, 使用递推方式也能解决, 反之亦然, 只不过在解题便利程度上有所差异。在某些情况下, 题目给定的状态转移关系较为复杂, 如果使用递推的方式解决, 需要在迭代循环结构中处理多个约束条件, 往往不甚方便, 而使用递归的方式则能够较为清晰地表述问题的结构, 同时结合备忘技巧又能够保证计算的高效率, 因此使用备忘方式解题具有优势。使用备忘方式解题的难点主要在于确定递归关系和递归出口。

下面, 以经典的 01 背包问题为例来演示如何应用备忘技巧解决动态规划问题。01 背包问题包含两个状态参数, 一个状态参数是当前能够进入背包的物品种数 N , 另外一个状态参数是背包的容量 V , 动态规划中总的状态数量为 NV 。从其递推关系式可以看出, 对于某件物品 x 来说, 只有两种选择, 选或者是不选。如果不选, 则背包的容量不会变化, 总价值也不会变化。如果选择则有两种可能: 当前背包的容量不足以放下该件物品, 此时背包的容量和总价值仍然不会变化; 另外一种是可以放下当前物品, 则背包的剩余容量及总价值会发生变化, 此时需要取具有最大总价值的选择。将上述过程使用递归的形式予以实现, 非常“自然”而“优美”。

```

//-----11.2.6.cpp-----//
// N 表示物品的总数, V 表示背包的容量。

```

```

int N, V;
// volume 记录各个物品的容量, price 记录各个物品的价值。
int volume[128], price[128];
// dp1 用于备忘方式下记录状态的最优值, dp2 用于递推方式下记录状态的最优值。
int dp1[128][1024], dp2[1024];
// flag 标记物品是否出现在最优选择方案中。
int flag[128];

// 备忘方式进行解题。
int dfs(int n, int v) {
    if (~dp1[n][v]) return dp1[n][v];
    if (n == N) return 0;
    // r1 为不选择物品 n 时的最优值。
    int r1 = dfs(n + 1, v);
    // r2 为能够选择物品 n 时的最优值。
    int r2 = 0;
    if (v + volume[n] <= V) r2 = dfs(n + 1, v + volume[n]) + price[n];
    // 确定哪种情况下具有最优值。
    flag[n] = r1 < r2;
    return dp1[n][v] = max(r1, r2);
}

int main(int argc, char *argv[]) {
    int cases;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> N >> V;
        for (int i = 0; i < N; i++) cin >> volume[i] >> price[i];
        // 使用备忘方式进行解题。
        memset(dp1, -1, sizeof(dp1));
        memset(flag, 0, sizeof(flag));
        cout << dfs(0, 0) << ' ';
        // 使用递推方式进行解题。
        memset(dp2, 0, sizeof(dp2));
        for (int n = 0; n < N; n++)
            for (int v = V; v >= volume[n]; v--)
                dp2[v] = max(dp2[v], dp2[v - volume[n]] + price[n]);
        cout << dp2[V] << '\n';
    }
    return 0;
}
//-----11.2.6.cpp-----//

```

对于以下的输入:

```

1
13 676
1 613
3 98
8 834
10 853
2 781
9 102
3 211

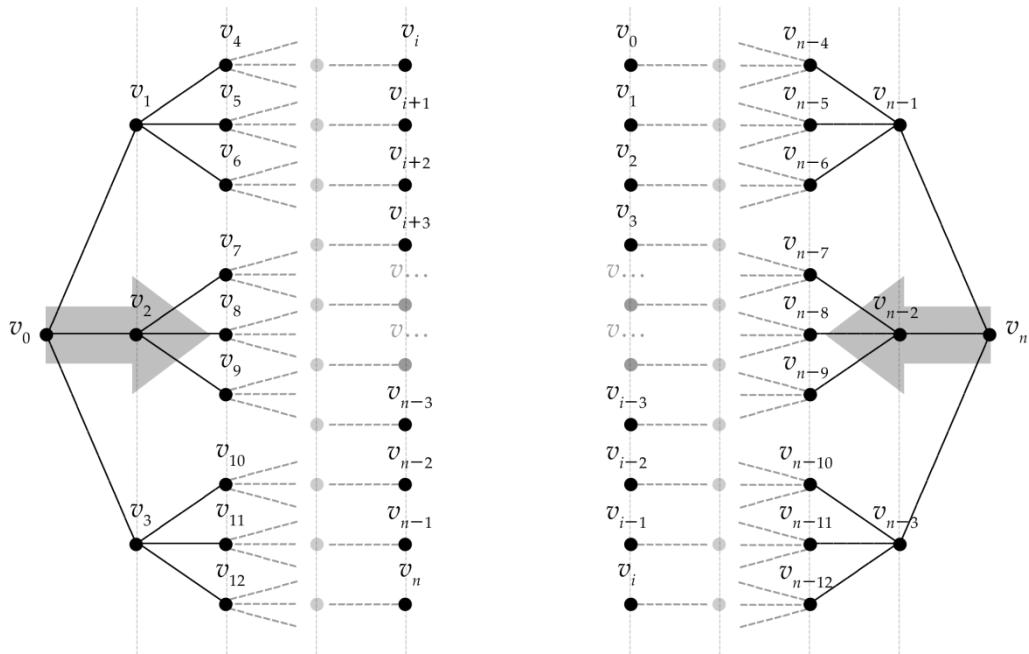
```

9	148
9	459
7	119
1	196
4	731
8	946

其输出为：

6091	6091
------	------

在解题中应用备忘技巧时有两种常见的递归方式，一种是“从前往后”的递归，一种是“从后往前”的递归。如果题目条件给定的是一种初始状态，从初始状态出发可以到达多个后继状态，同时从前置状态到后继状态的转移关系明确且最终需要从初始状态出发能够达到的最优值，那么以“从前往后”的递归形式来应用备忘技巧就较为方便。如果题目给定的是一种结束状态，从结束状态可以根据题目约束条件得到多个前置状态且最终需要确定结束状态的最优值，那么以“从后往前”的递归形式应用备忘技巧较为便利。对于一般的动态规划，在使用递归实现时，既可以使用“从前往后”的递归方式，也可以选择“从后往前”的递归方式，两者不存在本质差别，只不过是在具体编码实现时哪种方式更为便利的问题。也就是说，能够使用“从前往后”式的备忘解决的动态规划问题，使用“从后往前”式的备忘同样能够解决^I。



^I 在“从后往前”的递归中有一种特殊的类型——“区间型”动态规划，在此类题目中，题目的条件给定的是一个总体区间，对于该区间划分为子区间后的子问题会以明确的形式给出，从而使得将较大的问题分解为较小的问题这一步骤较为容易，解题的关键转移到“如何将子区间的结果进行合并来表述总体区间的解”这一步骤上来（参见本章第 11.5 节的“区间型动态规划”）。如果从图论的角度来看，“区间型”动态规划所对应的隐式图是一棵二叉树，而其他类型动态规划所对应的隐式图一般是多叉树。

图 11-6 若顶点 v_0 表示某种初始状态, 从 v_0 开始根据状态转移规则可以得到多个后继状态, 则使用“从前往后”的递归较为合适; 若顶点 v_n 表示某种结束状态, 从 v_n 开始, 根据状态转移规则可以得到多个前置状态, 则适用“从后往前”的递归较为合适

如果从图论的角度来审视此类问题, 则可将状态视为图中的顶点, 将关联状态之间进行转移的代价视为边权, 最终可以将题目给定的约束条件建模为有向无圈图, 题目的解就对应着该有向无圈图中从源点(初始状态)到终点(目标状态)的一条具有最短(或最长)距离的路径。

473 Raucous Rockers^D (破锣摇滚乐队)

你刚刚继承了由破锣摇滚乐队创作的 n 首歌曲的所有权, 这些歌曲之前并未公之于众。你打算从这些歌曲中选出一些, 将其制作成一套包含 m 张 CD 的音乐专辑。每张 CD 最多能够刻录 t 分钟的歌曲, 而且每首歌曲不能跨 CD 刻录(即一首歌曲不能分成多个部分刻录在不同的 CD 上, 只能刻录在一张 CD 上)。因为你是一名古典音乐爱好者, 对这些流行音乐的艺术价值无法作出判断, 因此你决定按照以下规则来选取需要刻录的歌曲: (1) 歌曲将按照创作日期的先后顺序排序并进行刻录; (2) 刻录的歌曲数量应该尽可能地多。

输入

输入包含多组测试数据。输入的第一行包含一个整数, 表示测试数据的组数, 接着是一个空行, 每两组测试数据之间也间隔一个空行。每组测试数据由两行组成, 第一行包含三个整数 n, t, m , 第二行包含一个列表, 表示 n 首歌的长度, t_1, t_2, \dots, t_n , 按照这些歌曲创作日期的先后顺序给出(每首歌的长度 t_i 小于 t , 且有 $\sum_{i=1}^n t_i > m \times t$)。

输出

对于每组测试数据输出一个整数, 表示按照前述给定的选取规则, 在 m 张光盘上能够刻录的最大歌曲数量。在相邻两组测试数据的输出之间打印一个空行。

样例输入

```
2
10 5 3
3, 5, 1, 2, 3, 5, 4, 1, 1, 5
1 1 1
1
```

样例输出

```
6
1
```

分析

如果使用递推的方式解题, 似乎不太容易推导出递推关系式。按照题意, 需要根据歌曲创作日期的先后顺序进行选择, 以决定某首歌曲是否刻录。假设从 0 开始为歌曲和光盘数量进行计数, 使用一维数组 $songs$ 记录每首歌曲的长度, $songs[i]$ 表示第 i 首歌曲的长度, 那么初始时是什么状态呢? 很明显, 初始状态由歌曲的数量、已使用的光盘数量、当前光盘已经使用的时间这三个参数决定, 不仅初始状态是这样, 在问题的任意一个状态, 都可以由前述三个参数决定。因此可以将初始时的状态定义为“已经处理到第 0 首歌曲, 当前为第 0 张光盘, 当前光盘已经使用了 0 分钟”。我们所需求求解的就是在这种初始状态下能够刻录的最大歌曲数量。那么进一步可以假设当前是这样一种状态——“已经处理到第 $nSong$ 首歌曲, 当前为第 $mDisk$ 张光盘, 当前光盘已经使用了 $tMinute$ 分钟”, 对于第 $nSong$ 首歌曲, 有两种选择, 要么略过, 要么刻录。

(1) 如果选择不刻录, 则当前处理的歌曲序号递增 1, 其他两个状态参数不变, 继续进行递归, 递归状态参数更新为“已经处理到第 $nSong+1$ 首歌曲, 当前为第 $mDisk$ 张光盘, 当前光盘已经使用了 $tMinute$ 分钟”。

(2) 如果选择刻录, 有三种情况:

(2a) 第 $mDisk$ 张光盘能够容纳第 $nSong$ 首歌曲且有剩余空间, 则递归状态参数需要更新为“已经处理到第 $nSong+1$ 首歌曲, 当前为第 $mDisk$ 张光盘, 当前光盘已经使用了 $tMinute + songs[nSong]$ 分钟”;

(2b) 第 $mDisk$ 张光盘恰好能够容纳第 $nSong$ 首歌曲, 则递归状态参数需要更新为“已经处理到第 $nSong+1$ 首歌曲, 当前为第 $mDisk+1$ 张光盘, 当前光盘已经使用了 0 分钟”;

(2c) 第 $mDisk$ 张光盘无法容纳第 $nSong$ 首歌曲, 需要将其刻录在第 $mDisk+1$ 张光盘上 (因为每首歌曲的长度 t_i 均不会超过一张光盘的容量 t), 则递归状态参数需要更新为“已经处理到第 $nSong$ 首歌曲, 当前为第 $mDisk+1$ 张光盘, 当前光盘已经使用了 0 分钟”。

递归出口: 如果当前处理的歌曲序号已经大于等于歌曲总数 n 或者已经使用的光盘序号大于等于 m 则返回 0, 表示此种状态下能够刻录的最大歌曲数量已经为 0。

根据上述回溯法解题的思路可以容易得到以下解题关键代码。

```
vector<int> songs;
int n, t, m;

int dfs(int nSong, int mDisk, int tMinute) {
    // 超出可用歌曲或光盘的数量上限后还能够刻录的歌曲数量显然为 0。
    if (nSong >= n || mDisk >= m) return 0;
    // 选择不刻录第 nSong 首歌曲的情况下当前状态能够刻录的最大歌曲数量。
    int r = dfs(nSong + 1, mDisk, tMinute);
    // 选择刻录第 nSong 首歌曲的情况下当前状态能够刻录的最大歌曲数量。分三种情况:
    // (1) 第 mDisk 张光盘能够容纳第 nSong 首歌曲且有剩余空间;
    // (2) 第 mDisk 张光盘恰好能够容纳第 nSong 首歌曲;
    // (3) 第 mDisk 张光盘无法容纳第 nSong 首歌曲, 需要将其刻录在第 mDisk+1 张光盘上。
    if (tMinute + songs[nSong] < t)
        r = max(r, 1 + dfs(nSong + 1, mDisk, tMinute + songs[nSong]));
    if (tMinute + songs[nSong] == t)
        r = max(r, 1 + dfs(nSong + 1, mDisk + 1, 0));
    if (tMinute + songs[nSong] > t)
        r = max(r, dfs(nSong, mDisk + 1, 0));
    // 返回当前状态下能够刻录的最大歌曲数量。
    return r;
}
```

但是仅使用上述普通的回溯显然是不可行的。对于任意一首歌曲, 要么刻录, 要么不刻录, 则总的状态数量为 2^n 级别, 对于 $n \geq 30$ 的情形无法在限定时间内得到解。为什么程序运行效率很低? 观察递归求解的过程, 在每次进入某种状态时, 不管之前是否已经计算, 都要重新开始计算, 这显然会增加时间消耗。那么一个很自然的想法就是使用某种方法将此种状态下的计算结果予以保存, 以待后续使用。因为状态只由三个参数决定, 这不同的状态数量只有 $n \times t \times m$ 种, 只需确定这么多种状态下的计算结果即可, 而不需要反复计算某个已经遇到过的状态的计算结果, 进而能够根据递推关系使用递归结合备忘的方式进行求解。

```
vector<int> songs;
int dp[1024][128][128];
int n, t, m;
```

```

int dfs(int nSong, int mDisk, int tMinute) {
    if (nSong >= n || mDisk >= m) return 0;
    // 使用备忘技巧。
    if (~dp[nSong][mDisk][tMinute]) return dp[nSong][mDisk][tMinute];
    int r = dfs(nSong + 1, mDisk, tMinute);
    if (tMinute + songs[nSong] < t)
        r = max(r, 1 + dfs(nSong + 1, mDisk, tMinute + songs[nSong]));
    if (tMinute + songs[nSong] == t)
        r = max(r, 1 + dfs(nSong + 1, mDisk + 1, 0));
    if (tMinute + songs[nSong] > t)
        r = max(r, dfs(nSong, mDisk + 1, 0));
    // 返回当前状态下能够刻录的最大歌曲数量。
    return dp[nSong][mDisk][tMinute] = r;
}

```

从此题的解题过程可以看出，解题的关键是确定状态是由哪些参数决定，进而能够使用递归加备忘的方式进行求解。实际上，本题所对应的数学模型为一张隐式图，一个状态就相当于有向图中的一个顶点，从某个状态能够到达另外一个状态表明这两个状态所对应的顶点间存在一条有向边。

强化练习：[672 Gangsters^D](#)，[757 Gone Fishing^C](#)，[801 Flight Planning^E](#)，[10261* Ferry Loading^B](#)。

扩展练习：[709* Formatting Text^D](#)，[959* Car Rallying^E](#)，[1250 Robot Challenge^E](#)，[11084* Anagram Division^D](#)，[11514* Batman^D](#)。

607 Scheduling Lectures^C (课程安排)

你正在教授一门课程并且需要讲述 n ($1 \leq n \leq 1000$) 个主题，每次授课的时长为 L ($1 \leq L \leq 500$) 分钟，每个主题需要 t_1, t_2, \dots, t_n ($1 \leq t_i \leq L$) 分钟进行讲述。对于每个主题，你必须决定在哪次授课中予以讲述。这里有两个安排上的限制：

(1) 每个主题必须在单次授课中讲述。它不能被分开在两次授课中讲述。这可以降低在两次授课间的不连续性。

(2) 对于 $1 \leq i < n$ ，主题 i 必须在主题 $i+1$ 之前讲述。否则，学生在理解主题 $i+1$ 时可能不具备必要的前置知识。

在上述限制下，有时候在授课结束时必须剩余一些自由活动时间。假如自由活动时间最多只有 10 分钟，学生们将会很高兴的下课离开。然而，如果自由活动时间剩得更多，学生们则会感到学费有些白花了。因此，我们利用以下公式对一次授课的不满意度指数 (dissatisfaction index, DI) 进行建模：

$$DI = \begin{cases} 0 & \text{如果 } t = 0 \\ -C & \text{如果 } 1 \leq t \leq 10 \\ (t - 10)^2 & \text{其他情形} \end{cases}$$

此处 C 是一个正整数， t 是在授课结束时剩余的自由活动时间。总的不满意度指数为每次授课的不满意度指数的总和。

对于本问题来说，你必须确定在满足上述限制的条件下最少的授课次数。如果存在多种满足最少授课次数的授课安排，则选取具有最小的不满意度指数的授课安排。

输入

输入包含多组测试数据。每组测试数据的第一行包含一个整数 n ，如果 $n=0$ 表示后续已无测试数据。接着的一行包含整数 L 和 C ，后续是 n 个整数 t_1, t_2, \dots, t_n 。

输出

对于每组测试数据，输出三行，分别为测试数据的组数、需要的最少课程节数、对应课程安排的不满意指数。在两组测试数据的输出间打印一个空行。

样例输入

```
6
30 15
10 10 10 10 10 10
10
120 10
80 80 10 50 30 20 40 30 120 100
0
```

样例输出

```
Case 1:
Minimum number of lectures: 2
Total dissatisfaction index: 0

Case 2:
Minimum number of lectures: 6
Total dissatisfaction index: 2700
```

分析

因为题意要求在最少课程节数的情况下所能得到的具有最小不满意度指数的课程安排，因此需要先确定最少的授课次数。由每次课程的长度 L 和每个主题的时间 t_i ，使用贪心策略可以容易确定最少的授课次数，因此难点在于当有多种授课安排满足题目约束时，如何确定具有最小不满意度指数的授课安排方案。令 $dp[i][j]$ 表示 i 次授课讲述 j 个主题时的最小不满意度指数， $di(x)$ 表示一次授课剩余自由活动时间为 x 分钟时的不满意度指数，根据题意，有递推关系

$$dp[i][j] = \min\{dp[i-1][k] + di(t_{k+1} + \dots + t_j), i \geq 1, k < j, t_{k+1} + \dots + t_j \leq L\}$$

也就是说，假设前 $i-1$ 次授课总共讲述了 k 个主题，那么从主题 t_{k+1} 到 t_j 必须在第 i 次授课中讲述，显然 $dp[i][j]$ 需要取所有可能的 k 中的最小值，而且根据题意，从主题 t_{k+1} 到 t_j 的时间不能超过一节课的时间 L 。从递推关系的形式来看，使用备忘中的“从后往前”递归的方式进行求解较为方便，其时间复杂度为 $O(n^2)$ 。

参考代码

```
const int INF = 1 << 30, MAXN = 1024;

int N, L, C, ML, ti[MAXN], DI[MAXN][MAXN], visited[MAXN][MAXN];

// 根据时间 m 确定不满意指数。
int getDI(int m) {
    int t = L - m;
    if (t == 0) return 0;
    if (1 <= t && t <= 10) return -C;
    return (t - 10) * (t - 10);
}

// 使用备忘技巧的动态规划算法。
int dfs(int lectures, int topics) {
    // 如果某个状态已经计算则直接返回结果。
    if (visited[lectures][topics]) return DI[lectures][topics];
    int *di = &DI[lectures][topics];
    if (lectures == 0) return topics ? INF : 0;
    else {
        *di = INF;
        int elapsed = 0;
        // 根据题意确定前置状态的最小不满意指数从而进一步确定当前状态的最小不满意指数。
        for (int t = topics; t >= 1; t--) {
            elapsed += ti[t];
            if (elapsed >= L) break;
            if (di[lectures - 1][t] != INF) {
                *di = min(*di, di[lectures - 1][t] + getDI(elapsed));
            }
        }
    }
}
```

```

        if (elapsed > L) break;
        int best = dfs(lectures - 1, t - 1);
        if (best != INF) *di = min(best + getDI(elapsed), *di);
    }
    visited[lectures][topics] = 1;
    return *di;
}
}

int main(int argc, char *argv[]) {
    int cases = 0;
    while (cin >> N, N > 0) {
        ML = 1;
        cin >> L >> C;
        memset(visited, 0, sizeof(visited));
        for (int i = 1, elapsed = 0; i <= N; i++) {
            cin >> ti[i];
            elapsed += ti[i];
            // 使用贪心策略确定最少的授课节数。
            if (elapsed > L) {
                ML++;
                elapsed = ti[i];
            }
        }
        if (cases++ > 0) cout << '\n';
        cout << "Case " << cases << ":\n";
        cout << "Minimum number of lectures: " << ML << '\n';
        cout << "Total dissatisfaction index: " << dfs(ML, N) << '\n';
    }
    return 0;
}

```

强化练习: 882* The Mailbox Manufacturers Problem^D, 10239 The Book-Shelfer's Problem^D, 10328 Coin Toss^C, 10532 Combination Once Again^C。

扩展练习: 986* How Many^D, 10604* Chemical Reaction^D。

11.3 松弛

松弛 (relaxing) 作为解决优化问题一种技巧, 在图算法中应用广泛。松弛是逐渐放宽条件约束使得欲求的量逐渐达到最优的过程。在某些情形, 直接按照问题的约束解决问题很难, 此时可以考虑将问题约束适当“放宽”, 使得在具有更宽松约束的问题上取得相应进展后, 再利用已有的结论往前推进以便原有问题的解决^[139]。松弛和动态规划的关系密切, 在最优化问题中经常作为常用的两种技巧同时出现。例如, 求单源最短路径的 Moore-Dijkstra 算法, 在存在负权边的图中求最短路径的 Bellman-Ford 算法, 求所有顶点对间最短路径的 Floyd-Warshall 算法……都能看到松弛和动态规划的身影。以下尝试从动态规划和松弛的视角, 对之前介绍的图算法再进行一次回顾以获得更为深刻的理解。

11.3.1 Moore-Dijkstra 算法

单源最短路径 (Single-Source Shortest Paths, SSSP) 问题是图论中的基本问题。给定一个具有 $|V|$ 个顶点和 $|E|$ 条边的有 (无) 向图, 图中所有边权均为非负值, 同时给定一个起点 s , 要求确定从起点 s 到所有其他顶点的最短路径长度和具体的最短路径构成 (即最短路径是由哪些边组成的), 此即 SSSP 问题。Moore-Dijkstra 算法是解决 SSSP 问题的经典方法。算法使用数组 $dist$ 记录各个顶点与起点 s 的当前最短路

径的长度（此时的最短路径可能不是最优的），初始时，起点本身的最短路径长度为 0，所有其他顶点的最短路径长度设为“无穷大”，即

$$dist[s] = 0, \quad dist[v] = \infty, \quad v \neq s$$

此处的“无穷大”并不是指设置一个计算机无法表示的值，只需要将其设置得足够大，使得所有可能的最短路径长度均不会超过此值即可。与此同时，使用数组 *visited* 记录各个顶点是否已经访问（即是否已经经过处理），初始时所有顶点均未被访问，即

$$visited[v] = \text{false}$$

接着算法迭代 $|V|$ 次，每次从未访问的顶点中选择一个与起点具有最短距离的顶点 v ，顶点 v 满足

$$dist[v] = \min_{p: visited[p] = \text{false}} dist[p]$$

最初，起点 s 被选择。在顶点 v 确定后，将其标记为已访问状态，接着进行松弛操作：对于顶点 v 的所有出边 (v, to) ，算法逐一尝试是否能够改进 $dist[to]$ 。令出边的权值为 $weight[v][to]$ ，则松弛操作可以表示为

$$dist[to] = \min\{dist[to], dist[v] + weight[v][to]\}$$

可以将松弛操作视为动态规划中根据递推关系进行状态更新的操作。当顶点 v 的所有出边均处理完毕后，当前轮次的迭代完成。经过 $|V|$ 次同样的迭代后，所有顶点均已被访问，算法终止，此时数组 *dist* 所记录的即是各个顶点与起点 s 的最短路径长度。

在与 Moore-Dijkstra 相关的动态规划题目中，通常は要求在各种约束下确定从某个起始状态到达终止状态的最短距离，处理此类问题的一般步骤为：(1) 定义动态规划的状态参数；(2) 确定状态转移规则；(3) 根据题目约束构建图；(4) 从起始顶点出发，使用类似于 Moore-Dijkstra 算法的队列实现更新到达其他顶点时相应状态的最短距离。

10269 Adventure of Super Mario^C（超级马里奥的冒险）

在拯救了美丽的公主之后，超级马里奥需要找到回家的路——当然了，是和公主一起 :-) 马里奥对“超级马里奥世界”非常熟悉，因此他不需要地图，他只需要一条最佳路径来节省时间。

在超级马里奥世界中，总共有 A 座村庄和 B 座城堡。村庄编号从 1 到 A ，城堡编号从 $A+1$ 到 $A+B$ 。马里奥居住在村庄 1，他需要从城堡 $A+B$ 出发返回村庄 1。在不同的两个地点之间有双向道路连接，两个地点最多只有一条道路连接，不会出现一条道路的两端连接同一处地点的情形。马里奥已经测量了每条道路的长度，但是他并不想一直走路回家，因为每走一单位的距离就要花费他一单位的时间（这可真慢！）。

幸运地是，马里奥在拯救公主的城堡中发现了一双魔法鞋，如果穿上它们，他就能够从一个地方瞬移到另外一个地方，而且不用花费任何时间。（不用担心公主，马里奥已经找到一种安全的方法带着公主和他一起瞬移，但是他是不会告诉你究竟是如何做到的，:--P）

由于城堡中存在陷阱，马里奥在瞬移过程中不会径直穿过某个城堡。如果有城堡在瞬移的路径中，他会在到达城堡时停下来，结束此次瞬移。马里奥总是在村庄或者城堡时开始瞬移或者停止瞬移，不会在两个地点连接的道路中途停止瞬移。不过，由于魔法鞋太旧了，马里奥使用魔法鞋一次最多只能瞬移 L 千米的距离，而且使用魔法鞋的次数总共不能超过 K 次。当返回家中之后，他或许可以修好魔法鞋以便能够再次使用它们。

输入

输入的第一行包含一个整数 T ，表示测试数据的组数 ($1 \leq T \leq 20$)。每组测试数据的第一行包含五个整数： A, B, M, L, K 。 A 表示村庄的数量， B 表示城堡的数量， $1 \leq A, B \leq 50$ ， M 表示道路的数量， L 表示一次瞬移所能经过的最长距离 ($1 \leq L \leq 500$)， K 表示魔法鞋能够使用的次数 ($0 \leq K \leq 10$)。紧接着的 M 行，

每行包含三个整数 X_i , Y_i , L_i , 表示有一条道路连接地点 X_i 和 Y_i , 它们之间的距离是 L_i , 走完该道路的时间也是 L_i ($1 \leq L \leq 100$)。

输出

对于每组测试数据输出一行, 此行包含一个整数, 表示马里奥和公主回家所需花费的最少时间。你可以假定, 对于所有测试数据, 马里奥总是能够找到回家的路。

样例输入

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

样例输出

```
9
```

分析

本题的第一个难点是定义动态规划的状态。将村庄和城堡视为图的顶点, 则马里奥的状态可由三个参数确定: 当前所处顶点的序号 u , 已经使用的瞬移次数 k , 当前瞬移已经移动的距离 $walked$, 可以将其表示为一个三元组(u , k , $walked$)。令 $dist[u][k][walked]$ 表示马里奥到达状态(u , k , $walked$)时所需要花费的最少时间, 则题目所求的是从初始状态($A+B$, 0 , 0)到达目标状态(1 , $[0\dots K]$, $[0\dots L]$)的最短距离。需要注意的是, 由于题目对目标状态无其他约束条件, 因此到达目标状态时, 瞬移的使用次数和当前瞬移已经移动的距离并不是一个固定值, 而是一个范围, 只要在此范围内均视为有效的目标状态。

本题的第二个难点是确定状态间如何发生转移。根据题目的约束条件, 马里奥在前进过程中可以选择使用瞬移, 也可以选择不使用瞬移, 而且在瞬移过程中, 遇到城堡时必须停止瞬移, 因此有以下三种情形:

(1) 如果当前未使用瞬移, 即参数 $walked=0$, 则对于当前所在顶点 u 的所有邻接顶点 v , 当前状态会变更为(v , k , 0), 令 L_{u-v} 表示顶点 u 和 v 之间某条道路的长度, 若 $dist[u][k][walked]+L_{u-v} < dist[v][k][0]$, 则可将状态(v , k , 0)置入队列中继续更新。

(2) 如果当前已经使用了瞬移, 即 $walked>0$, 则有三种情形。(2a) 对于当前所在顶点 u 的某个邻接顶点 v , 如果 v 是城堡, 按照约束条件, 马里奥需要停止瞬移, 则状态变更为(v , k , 0), 若 $dist[u][k][walked]+L_{u-v} < dist[v][k][0]$, 则可将状态(v , k , 0)置入队列中继续更新; (2b) 若 v 是村庄, 且 $walked+L_{u-v} \leq L$, 表明可以继续当前瞬移, 则后续状态变更为(v , k , $walked+L_{u-v}$), 若 $dist[u][k][walked] < dist[v][k][walked+L_{u-v}]$, 则可将状态(v , k , $walked+L_{u-v}$)置入队列中继续更新; (2c) 若 v 是村庄, 但 $walked+L_{u-v} > L$, 表明当前瞬移无法继续, 则后续状态变更为(v , k , 0), 若 $dist[u][k][walked]+L_{u-v} < dist[v][k][0]$, 则可将状态(v , k , 0)置入队列中继续更新;

(3) 无论当前是否已使用瞬移, 只要瞬移的次数未达到上限次数 K , 且 $L_{u-v} \leq L$, 则可以开始一次新的瞬移, 即状态变更为(v , $k+1$, L_{u-v}), 若 $dist[u][k][walked] < dist[v][k+1][L_{u-v}]$, 可将状态(v , $k+1$, L_{u-v})置入队列中继续更新。

建立图的过程较为简单, 可以使用邻接表来表示图, 之后使用类似于 Moore-Dijkstra 算法的队列实现来进行最短距离的更新。

强化练习: 10166 Travel^D, 10356 Rough Roads^C, 10603 Fill^C, 10801 Lift Hopping^A, 11492 Babel^B。

11.3.2 Bellman-Ford 算法

对于存在负权边的图, Moore-Dijkstra 算法可能无法正确计算最短距离, 而 Bellman-Ford 算法能够在图中存在负权边的情况下正确计算最短距离, 只要包含负权边的圈其总的权值不为负值即可。在有负权圈的情况下, Bellman-Ford 算法无法确定在负权圈上的顶点之间的最短距离, 因为沿着负权圈可以使最短距离“越来越负”。Bellman-Ford 算法所采用的技巧也是松弛操作。初始时, 对于每一个顶点 v 都定义一个属性 $d[v]$, 表示从源点 s 到顶点 v 的最短路径上权值的上界, 称为最短路径估计 (shortest-path estimate), $\pi[v]$ 表示源点 s 到顶点 v 的最短路径上位于顶点 v 之前的顶点的编号。在初始化时, 对 $v \in V$, $\pi[v]$ 为空, 即当前尚未确定各个顶点的前驱顶点, 而对于 $v \in V - \{s\}$, 有 $d[s] = 0$ 以及 $d[v] = \infty$ 。在松弛操作中, 需要检查能否通过边 (u, v) 来改进 $d[v]$, 如果可以改进, 则更新 $d[v]$ 和 $\pi[v]$ 。

```
// 对边 (u, v) 进行松弛, 其边权为 w (u, v)。
relax(u, v, w) {
    if (d[v] > d[u] + w(u, v)) {
        d[v] = d[u] + w(u, v);
        π[v] = u;
    }
}
```

每种单源最短路径算法中都会在初始化后重复对边进行松弛, 松弛是改变最短路径和前趋的唯一方式。各种单源最短路径算法间的区别在于对每条边进行松弛操作的次数以及对边执行松弛操作的次序。在 Moore-Dijkstra 算法以及关于有向无圈图的最短路径算法中, 对每条边执行一次松弛操作, 而在 Bellman-Ford 算法中, 每条边要执行多次松弛操作。

```
// 源点为 s。
bool bellmanFord(s) {
    // 对每条边进行松弛操作, 总共进行 |V| - 1 轮松弛操作。
    for (int i = 1; i <= |V| - 1; i++)
        for (edge e : E)
            relax(u, v, w);
    // 检查是否存在负权圈。如果再进行一次松弛能够获得更小的最短距离则表明存在负权圈。
    for (edge e : E)
        if (d[v] > d[u] + w(u, v))
            return false;
    return true;
}
```

从动态规划的角度来看, Bellman-Ford 算法所对应的递推关系式为

$$d[v] = \min\{d[v], d[u] + w(u, v)\}, (u, v) \in E$$

在进行松弛操作时, 总的松弛操作轮数只需 $|V| - 1$ 个轮次。之所以只需进行 $|V| - 1$ 轮松弛操作, 原因在于若存在最短路径, 则最短路径上的顶点数量最多不超过 $|V|$ 个。如果最短路径上的顶点超过 $|V|$ 个, 则必定存在重复的顶点, 将重复的顶点去除从而可以得到更优的最短路径, 产生矛盾, 因此只要两个顶点间存在最短路径, 其路径上的顶点数量必定不会超过 $|V|$ 个。

在动态规划中, 应用较多的是 Bellman-Ford 算法的队列实现形式, 即最短路径加速算法 (SPFA)。出题者一般将解题的关键点设置在状态转移环节, 需要解题者根据题意设计状态, 然后正确地进行状态的转移才能得到正确的解答。总的来说, 解题的一般步骤和应用 Moore-Dijkstra 算法解决相关动态规划问题的步骤是类似的。

11.3.3 Floyd-Warshall 算法

Floyd-Warshall 算法是解决所有顶点对间最短距离问题的一种有效算法，其时间复杂度为 $O(n^3)$ 。在使用邻接矩阵表示图的情形下，其核心代码仅由三重循环构成，非常简洁。尽管核心代码非常简洁，但算法背后所蕴含的动态规划思想却相当地精妙，具有艺术上的美感。

使用动态规划解决最优化问题，其中首要的步骤是定义问题的状态。在 Floyd-Warshall 算法中，将所有顶点从 1 至 n 编号，令 $w[i][j]$ 表示顶点 i 和顶点 j 之间无向边的权值， $d[k][i][j]$ 表示“在只能使用编号为 1 至 k 的顶点作为中间顶点时，从顶点 i 到顶点 j 的最短路径长度”¹，则 $d[0][i][j]$ 表示从顶点 i 到顶点 j 不经过任何中间顶点时的最短路径长度，如果初始时顶点 i 和顶点 j 之间无关联边，则令其距离为无穷大，否则 $d[0][i][j] = w[i][j]$ ； $d[1][i][j]$ 表示从顶点 i 到顶点 j 只经过编号为 1 的顶点作为中间顶点时的最短路径长度； $d[2][i][j]$ 表示从顶点 i 到顶点 j 只经过编号为 1 和编号为 2 的顶点作为中间顶点时的最短路径长度； \dots ， $d[n-1][i][j]$ 表示从顶点 i 到顶点 j 经过编号为 1 至 $n-1$ 的顶点作为中间顶点时的最短路径长度；那么所有点对间最短距离问题的所求即为 $d[n][i][j]$ ，也就是从顶点 i 到顶点 j 经过编号为 1 至 n 的若干顶点作为中间顶点时的最短路径长度。在合理地定义动态规划问题的状态后，接下来的步骤就是根据最优子结构所蕴含的关系得到递推关系式。

递推关系式可以认为是在问题的当前状态和原有状态之间建立联系，是一种状态转移的表示。按照前述状态的定义， $d[k][i][j]$ 的含义是在使用编号从 1 到 k 的顶点中的若干顶点作为中间顶点时，从顶点 i 到顶点 j 的最短路径的长度，那么建立递推关系式的目标就是将 $d[k][i][j]$ 使用 $d[k-1][i][j]$ 来表示，由于从状态 $d[k-1][i][j]$ 到 $d[k][i][j]$ 的差别在于是否使用第 k 号顶点作为中间顶点，因此只有两种情况：(1) 顶点 i 到顶点 j 的最短路径不经过顶点 k ；(2) 顶点 i 到顶点 j 的最短路径经过顶点 k 。对于第(1) 种情况，在不经过顶点 k 的情况下，显然有

$$d[k][i][j] = d[k-1][i][j], \quad 1 \leq i, j, k \leq n$$

对于第(2) 种情况，有

$$d[k][i][j] = d[k-1][i][k] + d[k-1][k][j], \quad 1 \leq i, j, k \leq n$$

综合两种情况，有

$$d[k][i][j] = \min\{d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j]\}, \quad 1 \leq i, j, k \leq n$$

因此有以下的 Floyd-Warshall 算法的初步实现。

```
const int MAXV = 110;

int n, d[MAXV][MAXV], w[MAXV][MAXV];

void floydWarshall() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            d[0][i][j] = w[i][j];
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[k][i][j] = min(d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j]);
```

¹ 需要注意，“中间顶点”并不要求一定是最短路径上的顶点，可以只使用“中间顶点”的若干顶点来构成从顶点 i 到顶点 j 的最短路径。

}

在 Floyd-Warshall 算法中，每次迭代相较前一次迭代，都增加一个顶点作为中间顶点，接着检查是否有可能缩短当前两个顶点的最短距离。当进行 n 次迭代后，所有的顶点都已经作为中间顶点使用，此时获得的最短路径即为最优的。可以看到，每次增加一个顶点，即为“放宽”最短距离路径所能经过的中间顶点条件，因此称之为“松弛”非常形象。

在前述的背包问题中，介绍了使用滚动数组对动态规划问题的空间使用进行优化的技巧。在 Floyd-Warshall 算法中，同样可以使用该技巧来优化空间的使用。观察递推关系式

$$d[k][i][j] = \min\{d[k-1][i][j], d[k-1][i][k] + d[k-1][k][j]\}, 1 \leq i, j, k \leq n$$

在计算 $d[k][i][j]$ 时，需要使用 $d[k-1][i][j]$ 、 $d[k-1][i][k]$ 、 $d[k-1][k][j]$ ，能否省略数组的第一个维度，从而节省算法所使用的运行空间呢？

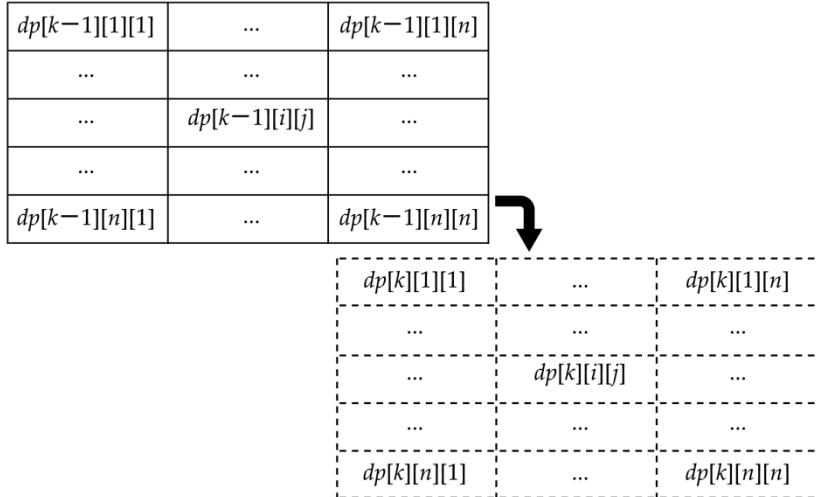


图 11-7 使用滚动数组技巧优化空间使用

答案是肯定的，方法就是使用滚动数组技巧。如图 11-7 所示，将三维数组视为多个二维数组“切片”构成，目前已经确定了第 $k-1$ 层“切片”各个单元格的值，现在需要确定第 k 层“切片”各个单元格的值。 $d[k][i][j]$ 在未更新前其值为无穷大，由之前的递推关系式可知， $d[k][i][j]$ 要更新为 $d[k-1][i][j]$ 和 $d[k-1][i][k] + d[k-1][k][j]$ 之中的较小值，如果将表示“切片”层次的第一维参数 k 去除，使得递推关系式变成

$$d[i][j] = \min\{d[i][j], d[i][k] + d[k][j]\}, 1 \leq i, j, k \leq n$$

可以发现，并不会影响最短距离的计算。因为省略掉第一维参数之后，在未更新之前， $d[i][j]$ 保存的是经过编号为 1 到 $k-1$ 个顶点作为中间顶点的最短路径，在更新 $d[i][j]$ 为经过编号为 1 到 k 的顶点作为中间顶点的最短路径时，可能需要利用 $d[i][k]$ 和 $d[k][j]$ ，此时 $d[i][k]$ 和 $d[k][j]$ 可能已经被更新，即 $d[i][k] \neq d[k-1][i][k]$ 和（或） $d[k][j] \neq d[k-1][k][j]$ ，但是由于递推关系式保证了 $d[i][k] \leq d[k-1][i][k]$ 和 $d[k][j] \leq d[k-1][k][j]$ ，因此使用 $d[i][k]$ 和 $d[k][j]$ 分别替换 $d[k-1][i][k]$ 和 $d[k-1][k][j]$ 进行更新并不会得到“更差”的最短路径，因此省略第一维参数后的递推关系式的正确性是有保证的。

以下是使用滚动数组技巧优化空间使用后的实现代码。

```
const int MAXV = 110;
```

```

int n, d[MAXV][MAXV], w[MAXV][MAXV];

void floydWarshall() {
    for (int i = 1; i <= n; i++)
        for (int j = 1; j <= n; j++)
            d[i][j] = w[i][j];
    for (int k = 1; k <= n; k++)
        for (int i = 1; i <= n; i++)
            for (int j = 1; j <= n; j++)
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

```

强化练习: [821](#) Page Hopping^A, [10543](#) Traveling Politician^C, [10681](#) Teobaldo's Trip^C。

扩展练习: [274](#) Cat and Mouse^D。

11.4 集合型动态规划

利用动态规划思维解决问题, 需要考虑问题的所有可能性, 在某些情况下可能需要一些特殊技巧来达到这个目的, 以便提高效率。在集合型动态规划中, 问题空间的每个状态由多个独立的单元构成, 例如棋盘放置棋子的状态, 如果棋盘的某个方格放置了棋子, 则视方格对应的二进制数位为 1, 否则为 0, 最后棋盘的状态就可以转换为一个二进制数, 对于二进制数, 使用位操作可以显著提高状态比对和状态转移的效率。一般情况下, 题目给定的状态可以使用 `int` 类型的整数进行存储, 特殊情况下可能需要 `long long int` 类型的整数, 但此时总的需要处理的状态数是较少的, 否则无法在规定时间内处理完毕。“状态压缩”是集合型动态规划中经常使用的一种技巧, 即使用位掩码(bit mask)将某种状态转换为一个二进制数来予以表示, 通过这个技巧可以使得解题过程更为简洁。下面通过一道题目的解析来初步了解何为“状态压缩”。

12348 Fun Coloring^D (趣味染色)

给定有限集 U 和一组 U 的子集 $S_1, S_2, S_3, \dots, S_m$, $|S_i| \leq 3$, 确定是否存在映射 $f: U \rightarrow \{\text{RED, BLUE}\}$, 使得对于每一个子集 S_i , 至少有一个元素与同子集的其他元素颜色不同。

输入

对于本问题, 集合 $U = \{x_1, x_2, x_3, \dots, x_n\}$ 。输入的第一行包含一个整数 k , 表示测试数据的组数。相邻两组测试数据间隔一个空行。每组测试数据的第一行包含两个整数 n 和 m , n 表示 U 中元素的个数, m 表示子集的个数。接下来的 m 行每行描述一个子集, 第一行描述的是子集 S_1 , 第二行描述的是子集 S_2, \dots , 第 m 行描述的是子集 S_m , 每行包含若干整数 j , 表示元素 x_j 在子集 S_i 中。 $1 \leq k \leq 13$, $4 \leq n \leq 22$, $3 \leq m \leq 111$ 。

输出

对于每组测试数据, 如果存在这样的映射, 输出 ‘Y’, 否则输出 ‘N’。输出由一行共 k 个字符构成, 每个字符要么是 ‘Y’, 要么是 ‘N’。第一个字符表示第一组测试数据是否存在满足要求的映射, 第二个字符表示第二组测试数据是否存在满足题意要求的映射, 依此类推。最后一个字符表示第 k 组测试数据是否存在满足题意要求的映射。

样例输入

```

2
5 3
1 2 3

```

样例输出

```

YN

```

2	3	4
1	3	5

7	7	
1	2	
1	3	
4	2	
4	3	
2	3	
1	4	
5	6	7

分析

题意要求对于给定大小为 n 的集合 U , 确定是否存在一种颜色的安排方法, 使得若干 U 的子集 S_i 中不包含相同颜色的元素。以样例输入的第一组测试数据为例, 可以为集合 U 中的元素指定如下颜色:

$$U = \{x_1 = \text{RED}, x_2 = \text{BLUE}, x_3 = \text{BLUE}, x_4 = \text{RED}, x_5 = \text{RED}\}$$

令 $R = \text{RED}$, $B = \text{BLUE}$, 则有:

$$S_1 = \{x_1 = R, x_2 = B, x_3 = B\}, S_2 = \{x_2 = B, x_3 = B, x_4 = R\}, S_3 = \{x_1 = R, x_3 = B, x_5 = R\}$$

满足 “ S_i 中至少有一个元素的颜色与其他元素不同” 的条件 (需要注意, 符合要求的颜色指定方案可能不止一种), 因此输出 ‘Y’。对于第二组测试数据, 由于 $S_1 = \{1, 2\}$, $S_2 = \{1, 3\}$, 则 2 和 3 必须安排不同的颜色, 但是 $S_3 = \{2, 3\}$, 使得不可能存在满足要求的颜色指定方案, 因此输出 ‘N’。

由于题目条件中所给的数据规模不大, 直觉上可以使用回溯法生成集合 U 的所有颜色组合, 然后对各个子集进行检查, 如果满足要求则表明符合题意要求的映射存在。但进一步地深入思考可以得知, 由于每个元素只能指定两种颜色的一种, 恰好可以使用二进制数数位来表示颜色。也就是说, 将颜色指定方案压缩为一个二进制数, 为 0 的位表示为此元素指定颜色 RED , 为 1 的位表示为此元素指定颜色 BLUE , 则从 0 到 $2^n - 1$ 的数就表示了所有的颜色指定方案。因此有以下解题步骤¹:

- (1) 读入 n 和 m ;
- (2) 读入子集 S_i , 将子集表示成二进制数 X_{S_i} , 若 S_i 包含 x_j , 则 X_{S_i} 的第 $j-1$ 位为 1;
- (3) 令 X 表示集合 U 的颜色指定方案, 若 X 的第 i 个二进制位为 0, 表示将第 $i+1$ 个元素的颜色指定为 RED , 否则指定为 BLUE , 从 $X=0$ 遍历到 $X=2^n-1$, 将子集 S_i 对应的二进制数 X_{S_i} 和 X 进行 “位与” 运算, 若结果为 X_{S_i} 或 0, 表示此子集中元素为相同颜色, 则对应的颜色指定方案 X 不满足要求;
- (4) 如果能够找到满足条件要求的 X , 则输出 ‘Y’, 否则输出 ‘N’。

强化练习: [11205 The Broken Pedometer^B](#)。

在状态压缩过程中, 需要将状态表示成一个二进制数, 其中为 1 的位表示此元素已经被使用, 为了获取此状态中使用了多少个元素, 需要知道此二进制数中有多少个二进制位为 1 (population count)。在早期的计算机系统中, 一般都设计了一个特定的指令来完成这个工作, 虽然现在计算机的速度得到了极大提高, 但是仍然有某些型号的机器保留了类似的指令可供使用^[140]。对于不提供此类指令的计算机来说, 编程完成这项任务有多种多样的方法, 最朴素的方法是使用右移位运算, 计数最低位为 1 的次数。不过 GCC 提供了内

¹ 截至 2020 年 1 月 1 日, 该题在 UVa OJ 上的评判程序似乎存在问题, 导致正确的解题方案无法获得 Accepted。

建函数 `__builtin_popcount()`，使得完成这项任务变得更为简单，其函数声明为：

```
int __builtin_popcount (unsigned int x);
```

它的作用是返回一个无符号整数的二进制表示中位为 1 的个数，其内部使用汇编调用的方式结合查表实现，效率很高。与之类似的还有：

```
// 返回 x 的二进制表示中从最高位开始的连续 0 的个数 (count of leading zeros).
int __builtin_clz (unsigned int x);
```

```
// 返回 x 的二进制表示中从最低位开始往最高位方向连续 0 的个数 (count of trailing zeros).
int __builtin_ctz (unsigned int x);
```

注意，对于后两个函数，如果给定的参数 x 为 0，则函数的返回值为“未定义状态”。对于这三个函数，有对应的 `unsigned long int` 和 `unsigned long long int` 数据类型版本，其名称稍有差异¹。

强化练习：[12845* The Jolly Friar's Puzzle](#)^E。

从状态压缩的特点来看，集合型动态规划算法适用的题目一般具有以下的特点：

(1) 解法需要保存一定的状态数据（表示一种状态的一个数据值），每个状态数据通常情况下是可以通过二进制来表示的。这就要求状态数据的每个单元只有两种状态，比如说棋盘上的方格，放棋子或者不放，或者是硬币的正反两面。这样用 0 或者 1 来表示状态数据的每个单元，而整个状态数据就是一个一串 0 和 1 组成的二进制数。

(2) 解法需要将状态数据实现为一个基本数据类型，比如 `int` 数据类型等，即所谓的状态压缩。状态压缩的一方面是缩小了数据存储的空间，另一方面是在状态比对和状态整体处理时能够提高效率。这样就要求状态数据中的单元个数不能太大，比如用 `int` 来表示一个状态的时候，状态的单元个数不能超过 32 位二进制数所能表示的范围。

10032 Tug of War^B (拔河)

某地方机构举办的野餐聚会将安排一次拔河比赛。为了使拔河比赛尽可能地公平，所有参与者将被分成两支队伍，且满足下列条件：每个人必须分配到两支队伍中的一支；两支队伍之间的人数之差不能超过 1 人；两支队伍各自的总体重需要尽可能地接近。

输入

输入的第一行包含一个正整数，表示测试数据的组数，每组测试数据的格式描述如下。之后接着是一个空行，每两组测试数据之间有一个空行。每组测试数据的第一行包含一个整数 n ，表示参加野餐聚会的人数，接着是 n 行，第一行表示人员 1 的体重，第二行表示人员 2 的体重，依此类推。体重是一个整数，大小在 1 至 450 之间。野餐聚会至多有 100 人。

输出

对于每组测试数据，输出以空格分隔的两个整数，这两个整数表示两支队伍各自的体重和，较小的体重和排列在前。在相邻两组输出间打印一个空行。

样例输入

样例输出

¹ 参阅：<https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html>，2020。

```
1
3
100
90
200
```

```
190 200
```

分析

使用朴素的回溯法解决显然不可行。考虑到题目形式与子集和问题相类似，不妨从此角度切入思考。可以将问题重新表述为：从 n 个正整数中选出 $m=n/2$ 个数，使得这 m 个数的和与剩余的 $n-m$ 个数之和的差值尽可能小。那么需要确定从 n 个数中选取 m 个数所能得到的所有不同和，令 $F[i][j][k]$ 表示从前 i 个数中选取恰好 j 个数时和能否为 k ，若 $F[i][j][k]=1$ 表示可行， $F[i][j][k]=0$ 表示不可行，那么有以下递推关系式

$$F[i][j][k] = \max\{F[i-1][j][k], F[i-1][j-1][k-W_i]\}, \quad 1 \leq i \leq n, \quad 1 \leq j \leq i, \quad W_i \leq k \leq S$$

边界条件为 $F[0][0][0]=1$ 。根据递推关系式，结合“滚动数组”技巧^I，可以省略三维数组的第一维，从而可得到下述解题核心代码：

```
// N 为总人数, S 为体重和, W 记录人员体重, F 记录状态。
int N, S, W[101], F[64][45010];
F[0][0] = 1;
for (int i = 1; i <= N; i++)
    for (int j = 1; j <= min(i, N / 2); j++)
        for (int k = S / 2; k >= W[i]; k--)
            F[j][k] = max(F[j][k], F[j - 1][k - W[i]]);
```

上述通过递推方式解题的时间复杂度为 $O(n^2S)$ ，由于 UVa OJ 上的测试数据较弱，已经能够获得 Accepted，但是应对更为“严苛”的测试数据时显然会发生超时。能否将递推关系式进行优化以便提高效率？观察递推关系式， $F[i][j][k]$ 表示从前 i 个数中选取恰好 j 个数时其和为 k 的状态，数组第二维元素 j 所起的作用是标记当和为 k 时，“恰好有 j 个数”这种情况是否存在，那么不妨将状态的顺序予以调换，将 $F[i][j][k]$ 解释为“从前 i 个数中选取若干个数，其和为 j 时，恰已选择 k 个数时的状态”，那么从 $F[i][j][0]$ 到 $F[i][j][i]$ 的意义就是“从前 i 个数中选取若干个数，其和为 j ，而所选择的“若干个数”的数量恰为 $0, 1, \dots, i$ 个时的状态”，则递推关系式为

$$F[i][j][k] = \max\{F[i-1][j][k], F[i-1][j-W_i][k-1]\}, \quad 1 \leq i \leq n, \quad W_i \leq j \leq S, \quad 1 \leq k \leq i$$

初始时， $F[0][0][0]=1$ ，其他数组元素的值均为 0。由于 n 最大为 100，那么 $n/2$ 最大为 50，于是可以将前述递推关系式中三维状态数组的最后一维编码为一个 64 位二进制数，即使用 `long long int` 类型的数组元素 $F[i][j]$ 来表示这样的状态：如果 $F[i][j]$ 所对应的二进制数的第 k 个二进制位为 1，表示从前 i 个人中可以选择 k 个人使得体重之和为 j ，第 k 个二进制位为 0 则表示无法在前 i 个人中选择 k 个人使得体重之和为 j ，最终有以下递推关系式

$$F[i][j] = F[i][j] | (F[i-1][j-W_i] \ll 1), \quad 1 \leq i \leq n, \quad W_i \leq j \leq S$$

初始时， $F[0][0]=1$ ，其他数组元素的值均为 0。通过上述改进，可以将解题时间复杂度降低为 $O(nS)$ ^{II}。

^I 参见本章“优化”一节中“空间优化”的相关内容。

^{II} 严格来说，是将原来 $O(n^2S)$ 的时间复杂度降为 $O(knS)$ ，由于“位或”运算可以在常数时间内完成，因此 k 为常数，故优化后的时间复杂度可以认为是 $O(nS)$ 。

10149 Yahtzee^C (Yahtzee 游戏)

Yahtzee 是一个用五个骰子来玩的游戏，共掷 13 轮。同样的，记分卡也包含 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：

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：所有骰子点数之和

三同：若掷出至少三个相同点数的骰子，计所有骰子点数和

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：若掷出至少五个相同点数的骰子，计所有骰子点数和

小顺：若四个骰子成顺 (1, 2, 3, 4 或 2, 3, 4, 5)，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外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，计 0 分。游戏的总分为所有 13 项计分的和。若前六个计分项的得分之和大于或等于 63，则在总分中加上 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

每行输入包含 5 个 1 到 6 之间的整数，表示每轮掷骰子的情况。输入的第 13 行组成了一局完整的游戏。

输出

对于每局游戏，输出一行共 15 个数，表示每个计分项的得分（按题目描述的顺序）、奖励分（0 或 35）以及总分。如果存在多种具有同样最大总分的计分项选择，可以任意输出一种。

样例输入

```
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

```
3 6 9 12 15 30 21 20 26 50 25 35 40 35 327
```

分析

每组色子可以选择 13 种计分方式中的任意一种，已选择的计分方式不能再次选取。如果将每组色子按每种计分方式进行计分，并将分值按顺序排列，则可以得到一个方阵：

$$\begin{bmatrix} S_{1-1} & S_{1-2} & \cdots & S_{1-12} & S_{1-13} \\ S_{2-1} & S_{2-2} & \cdots & S_{2-12} & S_{2-13} \\ \vdots & \vdots & \vdots & \vdots & \cdots \\ S_{12-1} & S_{12-2} & \cdots & S_{12-12} & S_{12-13} \\ S_{13-1} & S_{13-2} & \cdots & S_{13-12} & S_{13-13} \end{bmatrix}$$

其中 S_{i-j} 表示第 i 组色子按照第 j 种计分方式计分的得分 ($i=1, \dots, 13$, 为色子组数; $j=1, \dots, 13$, 为计分种类)。那么题目所求可以转化为下列问题 (暂不考虑奖励分): 从一个大小为 13×13 的方阵中, 每一行和每一列只取一个元素进行求和, 确定能取到的最大值和相应的取法。根据乘法原理, 第 1 行有 13 种取法, 第 2 行有 12 种取法, ……, 第 13 行有 1 种取法, 总共有 $13! = 6227020800$ 种取法, 如果使用朴素的穷尽搜索, 肯定可以找到最大值, 但会超时。为什么会超时? 让我们来看一看穷尽搜索到底发生了什么。以下的伪代码使用嵌套循环来实现穷尽搜索, 用以寻找最大值。

```
int maxSum = 0;
for (int a = 1; a <= 13; a++) {
    // S[i][j] 保存的是第 i 组色子按照第 j 种计分方式计分的得分。
    int sum = S[1][a];
    for (int b = 1; b <= 13; b++) {
        if (b == a) continue;
        sum += S[2][b];
        for (int c = 1; c <= 13; c++) {
            if (c == a || c == b) continue;
            sum += S[3][c];
            // ...
            for (int m = 1; m <= 13; m++) {
                if (m == a || ... || m == 1) continue;
                sum += S[13][m];
                if (sum > maxSum) {
                    maxSum = sum;
                    // 记录取法。
                }
            }
        }
    }
}
```

可以直观地看到, 每完成一种取法的计算, 中间结果就被丢弃, 重新开始另外一种取法的计算, 实际上, 这些中间结果可以换一种方式善加利用以提高效率。

在朴素的穷尽搜索算法中, 由于未能充分利用中间计算结果, 导致了大量的重复计算, 这是程序运行时间大大增加的根本原因, 为了提高效率, 必须减少重复计算。那么如何才能减少重复计算呢? 需要换一个角度思考问题, 从当前取法组合的路走不通, 不妨从计分方式的组合考虑。题目给定了 13 种计分方式, 根据这 13 种计分方式是否选择, 可能的状态总共有 $2^{13} = 8192$ 种, 采用自底向上的递推方式, 对于每种状态确定能够获得的最大值, 应该可以在时间限制内得到结果。假设第一组色子选取了第 1 种计分方式, 分值为 a_1 , 则第二组色子只能在第 2~13 种计分方式中任选一种, 怎样表示这种状态呢? 把各次选择计分方式的状态表示成下列形式

$$c_{13}c_{12}c_{11}c_{10}c_9c_8c_7c_6c_5c_4c_3c_2c_1 \equiv 000000000000000$$

若某种计分方式已经使用, 则对应序号的二进制位为 1, 否则为 0。例如, 第一组色子选择了第一种计分方

式, 第二组色子使用了第二种计分方式, 则 c_1 和 c_2 为 1, 将 c_1 到 c_{13} 的状态转换为一个二进制数 (c_{13} 在高位, c_1 在低位)

$$0000000000011_2$$

该二进制数对应十进制数的 3。若相反, 第一组色子选择第二种计分方式, 第二组色子选择第一种计分方式, 这样得到的二进制数仍然是

$$0000000000011_2$$

那么该状态表示的是不管第一组和第二组色子选择第一或第二种计分方式的顺序如何, 只需比较两种选择下那种选择的分值大, 即可获得此种状态所对应的策略的最大分值。如果设立一个数组 dp 记录计分策略的得分, 将该二进制数即十进制的 3 作为数组的序号, 则该数组元素 $dp[3]$ 所表示的就是当第一和第二组色子选择第一或第二种计分方式时的最大值。同理, 对于以下二进制数

$$0000000000101_2$$

表示的是第一组色子和第三组色子各取第一或第三种计分方式, 同样以该二进制数即十进制下的 5 作为数组序号, 将所得计分和的最大值储存到 $dp[5]$ 中。考虑以下二进制数所表示的状态

$$0000000000111_2$$

前三组色子选择了前三种计分方式, 此二进制数可能为 000000000011_2 与 0000000000100_2 相加而来, 也可以是 0000000000101_2 与 000000000010_2 相加而来, 两种操作的含义: 第一种是当第一组和第二组色子取遍第一种和第二种计分方式得到的最大值与第三组色子取第三种计分方式得到的分值相加; 第二种表示第一组和第三组色子取遍第一种和第三种计分方式所得到的最大值与第二组色子取第二种计分方式得到的分值相加。如果比较两种操作的所得到的分值, 并将较大值储存到二进制数 000000000111_2 即十进制数 7 为序数的数组元素 $dp[7]$ 中, 则 $dp[7]$ 的含义就是前三组色子取前三种计分方式, 不管选取顺序如何, 所得到的最大值。依此类推, 当状态为

$$1111111111111_2$$

即求得了最大值。在求最大值的过程中需要一个数组来保存各个策略状态最大值所采取的计分方式, 以便在最后根据该数组回溯得到各个策略状态所采取的计分方式。

由上述讨论, 我们可以设立一个二维数组 $dp[i][j]$, i 表示前 j 组色子已选计分项的组合, j 表示最后一次是为第 j 组色子选择计分项, $dp[i][j]$ 表示的是为前 j 组色子选择计分项且最后一次是为第 j 组色子选择计分项时的最大总分。 i 的二进制表示中, 从低位到高位, 从 0 开始计数, 序号为 c 的位为 1, 表示计分项 c 已被选择, 为 0 则表示此计分项尚未选择。假设目前已经为前 $j-1$ 组色子确定了计分项, 其最大总分为 $dp[i][j-1]$, 那么当为第 j 组色子选择计分项时, 状态 i 中已经选择的计分项不能再次选择, 通过检查剩余的可能计分项选择就可以得到前 j 组色子选择 j 种计分项时的最大得分。因此, 递推关系可以表示为

$$dp[i | (1 \ll k)][j] = \max\{dp[i | (1 \ll k)][j], dp[i][j-1] + s[j][k]\}, ((i \gg k) \& 1) = 0$$

其中, k 表示尚未选择的计分项序号 (从 0 开始计数), $s[j][k]$ 表示第 j 组色子选择第 k 个计分项的得分。观察递推关系, 由于 j 记录的是最后一次是为第几组色子选择计分项, 而此数值总是和状态 i 的二进制表示中位为 1 的个数相同, 为了简化递推关系, 实际上可以将 j 省略, 转而使用状态 i 的二进制表示中位为 1 的个数来记录最后是为第几组色子分配了计分项, 递推关系可以简化为

$$dp[i | (1 \ll k)] = \max\{dp[i | (1 \ll k)], dp[i] + s[_builtin_popcount(i)][k]\}, ((i \gg k) \& 1) = 0$$

现在来考虑如何处理奖励分。由于是前六项计分大于或等于 63 分时才给予 35 分的奖励分, 在计算过程中, 需要将同策略的不同前六项得分的总分区分开来, 因为大于等于 63 的前六项得分效果是等同的, 只需考虑 0~63 这 64 种情况。如果前六项总分大于等于 63, 则将该总分放在数组序号为 63 的总分元素中, 小

于 63 的则放在相应分数为序号的元素中。在比较时，对前六项总分相同的元素比较总分大小，总分大的替代原来的元素项，那么数组的每一项储存的是前六项分数和等于数组元素序号时的最大总分。如 $dp[1111011100011_2][25]$ 表示的是采用策略 1111011100011₂ 时前六项分数为 25 时的最大总分，可能采用该策略时前六项分数为 25 分的情况并不存在，故初始时数组元素值均赋值为 -1。同样的，需要在替换时记录替换前后的所采用的计分策略和前六项得分以便通过回溯来重建得到解的过程。

与朴素的穷尽搜索算法 $O(n!)$ 的时间复杂度相比，上述动态规划算法时间复杂度为 $O(n2^n)$ ，对于 $n=13$ 的情形，可以在限制时间内获得通过。

强化练习：10123 No Tipping^D，10898 Combo Deal^C，10911* Forming Quiz Teams^A，11088* End up with More Teams^C，11218* KTV^B，11284 Shopping Trip^C。

扩展练习：1076* Password Suspects^D，1240 ICPC Team Strategy^D，1252* Twenty Questions^D，10817* Headmaster's Headache^C，11391* Blobs in the Board^D。

11.5 区间型动态规划

区间型动态规划是线性动态规划的一种扩展，其表现形式一般为：给定直线（或环形）上一组离散的点，每个点均有特定的权值（为整数或浮点数），要求选取若干点作为分界点，以便将这些离散的点划分为若干个部分，选取不同的分界点有不同代价，要求确定一种最优的划分方法，使得代价的总和最优（即使得代价最小或最大）。

11.5.1 矩阵链乘法

由 $m \times n$ 个数 a_{ij} 构成的 m 行 n 列的数表称为 m 行 n 列矩阵，可以记做

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

这 $m \times n$ 个数称为矩阵 \mathbf{A} 的元素，简称为元。矩阵的基本运算包括加法、减法、数乘等。这些运算的规则和四则运算类似。但是两个矩阵的乘法和四则运算中的乘法不同，两个矩阵的乘法仅当第一个矩阵 \mathbf{A} 的列数和另一个矩阵 \mathbf{B} 的行数相等时才有定义。如 \mathbf{A} 是 $m \times n$ 矩阵， \mathbf{B} 是 $n \times p$ 矩阵，它们的乘积 \mathbf{C} 是一个 $m \times p$ 矩阵，所需的标量乘法次数为 $m \times n \times p$ ，矩阵 \mathbf{C} 的某个元素可以通过以下方式计算

$$c_{i, j} = a_{i, 1}b_{1, j} + a_{i, 2}b_{2, j} + \cdots + a_{i, n}b_{n, j} = \sum_{r=1}^n a_{i, r}b_{r, j}$$

矩阵的乘法满足

- (1) 结合律： $(AB)C = A(BC)$ ；
- (2) 左分配律： $(A+B)C = AC + BC$ ；
- (3) 右分配律： $C(A+B) = CA + CB$ ；

但是矩阵乘法不满足交换律，即 AB 不一定等于 BA ，甚至 AB 有定义而 BA 无定义。

采用不同的顺序计算矩阵乘法所需的标量乘法次数可能相差很大。例如有三个矩阵， \mathbf{A} 为 50×10 的矩阵， \mathbf{B} 为 10×20 的矩阵， \mathbf{C} 为 20×50 的矩阵，如果需要计算 ABC ，采用 $(AB)C$ 的顺序，需要 60000 次标量乘法操作，而采用 $A(BC)$ 的计算顺序，只需要 35000 次标量乘法操作。

强化练习：442 Matrix Chain Multiplication^A。

当对一系列矩阵进行乘法操作时，如何确定它们进行相乘时的最少乘法次数呢？最简单的是使用回溯法，

遍历所有可能的相乘顺序，找出其中次数最少的一种相乘顺序。显然这种方法的时间复杂度是指数级别的，对于矩阵数量较少时尚可应用，但当数量增大时，效率会很低而变得不可用。

考虑 n 个矩阵的链乘，有 $(n-1)$ 种方法给这 n 个矩阵加第一组括号。例如 4 个矩阵的链乘 $ABCD$ ，有 3 种方法加第 1 组括号，它们是： $A(BCD)$ ， $(AB)(CD)$ ， $(ABC)D$ 。只要知道 $(n-1)$ 种加括号后的链乘中乘法次数最少的一种，即可得知 n 个矩阵链乘所需的最少乘法次数。观察加括号后的链乘，已经分解成更短长度的矩阵链乘，连续使用以上方法，最终可以将链乘分解为单个矩阵相乘的问题。因此矩阵链乘问题是具有最优子结构的。那么如何使用子问题的解来表示最终的解呢？给定一个长度为 n 的矩阵乘法链，令 $T[i][j]$ 表示对下标从 i 到 j 的矩阵进行链乘所需的最少乘法次数， $cost[i][j]$ 表示对矩阵 i 和 j 进行相乘操作所需要的乘法次数，则有以下递推关系式

$$T[i][j] = \min\{T[i][k] + T[k+1][j] + cost[k][k+1], i < k < j, 1 \leq i < j \leq n\}$$

根据上述递推关系式，可以很容易使用递归进行求解。

```
// 使用递归求 n 个矩阵链乘的最小乘法次数。
// di 存储的是矩阵的维数，第 i 个矩阵的维数为 di[i-1] × di[i]，i=1, 2, ..., n。
const int MAXV = 32, INF = 0x7f7f7f7f7f7f7f7f;

int di[MAXV];

int dfs(int i, int j) {
    // 当矩阵为本身时，乘法次数为 0。
    if (i == j) return 0;
    // 将括号放置在矩阵 i 和矩阵 j 之间的可选位置，递归计算乘法的次数，取最小值。
    int r = INF;
    for (int k = i; k <= j - 1; k++) {
        int cnt = dfs(i, k) + dfs(k + 1, j) + di[i - 1] * di[k] * di[j];
        r = min(r, cnt);
    }
    // 返回最小值。
    return r;
}
```

可以看到，类似于使用递归计算斐波那契数列的过程，在矩阵链乘法中，某些子问题被重复计算。假设有 4 个矩阵，编号为 1 至 4，在使用递归计算的过程中，产生了如图 11-8 所示的分支。

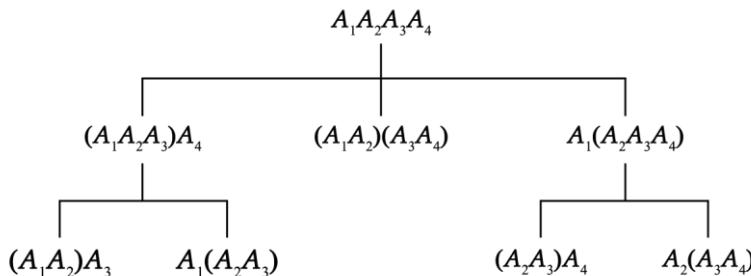


图 11-8 4 个矩阵的链乘法，递归求解时产生了重复计算，其中 A_1A_2 ， A_2A_3 ， A_3A_4 均重复计算了两次

因此矩阵链乘法问题存在重复的子问题，而最优子结构和重复子问题正是应用动态规划的标志。根据动态规划的思想，使用表格存储已经计算的值可以解决重复计算子问题，于是可以得到以下实现。

```

const int MAXV = 32, INF = 0x7f7f7f7f;

int di[MAXV];           // 矩阵的维数
int dp[MAXV][MAXV];    // 记录矩阵乘法的最小次数
int path[MAXV][MAXV];  // 记录最小乘法次数时选择的剖分位置以便重建乘法序列

// 初始化动态规划表格, dp[i][j] 表示第 i 个至第 j 个矩阵相乘时的最小乘法次数。
// memset(dp, -1, sizeof(dp));

// 采用表格式的动态规划, 递归计算矩阵链乘法的最小乘法次数。
int dfs(int i, int j) {
    if (~dp[i][j]) return dp[i][j];
    dp[i][j] = INF;
    if (i == j) dp[i][j] = 0;
    else {
        for (int k = i; k <= j - 1; k++) {
            int cnt = dfs(i, k) + dfs(k + 1, j) + di[i - 1] * di[k] * di[j];
            if (cnt < dp[i][j]) {
                // 记录具有最小乘法次数的括号位置, 以便重建加括号的方法。
                dp[i][j] = cnt;
                path[i][j] = k;
            }
        }
    }
    return dp[i][j];
}

```

在求解问题的同时, 可以使用辅助数据结构记录每次选择具有最小次数的乘法时括号的位置, 这样可以根据该数据结构重建具有最小乘法次数的加括号方法。

```

void printPath(int i, int j) {
    if (i == j) cout << "A" << i;
    else {
        cout << "(";
        printPath(i, path[i][j]);
        cout << " x ";
        printPath(path[i][j] + 1, j);
        cout << ")";
    }
}

```

从矩阵链乘法的动态规划解法不难看出, 区间型动态规划与备忘(记忆化搜索)关系密切。解题的关键是将原始问题的解使用子问题的解来表示, 具体就是将原区间划分为不重叠的子区间, 对子区间递归进行求解, 同时将解使用备忘技巧予以记录以避免重复求解, 最后根据各个子区间的解获得上一级区间的最优解。

强化练习: [348 Optimal Array Multiplication Sequence^A](#)。

扩展练习: [10453 Make Palindrome^B](#), [10739 String to Palindrome^A](#), [11022 String Factoring^C](#)。

11.5.2 石子合并问题

给定从左到右排列的 n 堆石子, 每堆包含 a_i 颗石子, 现在可以进行以下操作: 任选两堆相邻的石子, 将其合并为一堆, 得分为原来两堆石子的数量和。持续进行该操作直到所有石子堆合并为一堆, 求该过程所得分数之和的最小值。约束: $1 \leq n \leq 100$, $1 \leq a_i \leq 10^6$ 。

举个例子, 假设有 3 堆石子, 从左到右依次包含 1 颗、2 颗、3 颗石子, 那么总共有两种合并方法。第

一种方法，先合并包含 1 颗、2 颗石子的石子堆，得分为 3，然后再与包含 3 颗石子的石子堆合并，总得分为 9。第二种方法，先合并包含 2 颗、3 颗石子的石子堆，得分为 5，然后再与包含 1 颗石子的石子堆合并，总得分为 11。很明显，可能的最低得分为 9。

当 $n \leq 2$ 时，属于简单情形，故考虑当 $n \geq 3$ 时的处理。将石子堆从左到右依次编号为 1 至 n ，假想在每两堆石子间均放置一块隔板，在合并相邻两堆石子的时候，把这块假想的隔板抽掉。不难推知，总共可以放置 $n-1$ 块隔板，不妨将其从左到右依次编号为 1 至 $n-1$ 。假设按照某种最优策略进行合并，观察石子堆不断合并且只剩下一块隔板时的情形。不妨令这块隔板的编号为 k ，不难理解，隔板 k 从最初合并到现在并未被移动。那么，按照最优策略从 n 个石子堆开始合并直到只剩下两个石子堆且这两个石子堆之间的隔板编号为 k 的过程，实际上等价于以下过程：按照最优策略将编号 1 到 k 的石子堆合并，按照最优策略将编号 $k+1$ 到 $n-1$ 的石子堆合并。如果令 $dp[i][j]$ 表示将编号 i 至编号 j 的石子堆合并的最小得分值，按照前述的假设，将 n 堆石子进行合并的最小得分值为：

$$dp[1][n] = dp[1][k] + dp[k+1][n] + w$$

其中 w 表示将最后两堆石子合并为一堆石子时的得分。不难推知，最后两堆石子中，左侧的这堆石子包含的石子数目为编号 1 至编号 k （包含 1 和 k ）的石子堆的石子数目总和，右侧的这堆石子包含的石子数目为编号 $k+1$ 至编号 n （包含 $k+1$ 和 n ）的石子堆的石子数目总和，那么有

$$w = \sum_{i=1}^k a_i + \sum_{i=k+1}^n a_i = \sum_{i=1}^n a_i$$

类似的， $dp[1][k]$ 的最小值也一定是将编号 1 到 $k-1$ 的某块隔板作为最后抽取的隔板而获得的，对于 $dp[k+1][n]$ 也有类似的结论。不难看出，这是一个递归的过程。由于我们需要知道 k 的具体值才能计算 $dp[1][n]$ ，但是当前我们并不知道究竟将 1 至 $n-1$ 的哪块隔板作为最后抽取的隔板才是最优策略，那么，我们可以枚举 k 的值，计算 $dp[1][n]$ ，然后取所有可能得分值的最优值即可。类似的，我们也可以对 $dp[1][k]$ 和 $dp[k+1][n]$ 进行相似的操作。不难理解，由于在这个过程中我们枚举了所有的可能，最后肯定能够得到最优值。可以使用递推关系式表示为

$$dp[i][j] = \min_{i \leq k < j} \{dp[i][k] + dp[k+1][j] + w[i][j]\}, \quad 1 \leq i < j \leq n$$

其中 $w[i][j]$ 表示编号 i 至编号 j （包含 i 和 j ）的石子堆的石子数目总和，边界条件为： $dp[i][i] = 0, 1 \leq i \leq n$ 。

观察前述的递推关系式，可以看出它是一种自顶向下的动态规划，在枚举 k 值时，会产生重复的状态，即某个状态可能在自顶向下进行分解的过程中多次遇到。回顾之前“备忘”一节所介绍的内容，区间型动态规划往往与备忘密切相关，因此可以结合备忘技巧对求解过程进行加速。另外，由于在求解过程中需要反复求某个区间的石子堆的石子数目总和，为了提高效率，不妨定义一个前缀和数组 sum ，令 $sum[0] = 0$ ， $sum[i]$ 表示编号 0 至编号 i 的石子堆的石子数目总和，那么易知

$$w[i][j] = sum[j] - sum[i-1], \quad 1 \leq i < j \leq n$$

则前述的递推关系式等价于^I

$$dp[i][j] = \min_{i \leq k < j} \{dp[i][k] + dp[k+1][j] + sum[j] - sum[i-1]\}, \quad 1 \leq i < j \leq n$$

以下是使用自底向上递推和自顶向下递归的参考实现。

^I 对于具有类似递推关系式的区间型动态规划题目，可以应用一种称之为“四边形不等式优化”的技巧来提高运行效率，但需要使用自底向上的迭代式递推进行求解。参见本章第 11.9.6 小节“四边形不等式优化”的内容。

```

//-----11.5.2.1.cpp-----
const int MAXN = 110, INF = 0x7f7f7f7f;

int dp[MAXN][MAXN], n, a[MAXN], sum[MAXN] = {0};

int dfs(int i, int j) {
    if (~dp[i][j]) return dp[i][j];
    if (i == j) return 0;
    int r = INF;
    for (int k = i; k < j; k++)
        r = min(r, dfs(i, k) + dfs(k + 1, j) + sum[j] - sum[i - 1]);
    return dp[i][j] = r;
}

int main(int argc, char *argv[]) {
    bool useBottomUpMethod = true;
    while (cin >> n) {
        for (int i = 1; i <= n; i++) {
            cin >> a[i];
            sum[i] = sum[i - 1] + a[i];
        }
        // 使用自底向上的迭代式递推方法进行求解。
        if (useBottomUpMethod) {
            memset(dp, 0, sizeof(dp));
            for (int L = 2; L <= n; L++)
                for (int i = 1, j = L; j <= n; i++, j++) {
                    dp[i][j] = INF;
                    for (int k = i; k <= j; k++) {
                        int next = dp[i][k] + dp[k + 1][j] + sum[j] - sum[i - 1];
                        dp[i][j] = min(dp[i][j], next);
                    }
                }
            cout << dp[1][n] << '\n';
        }
        // 使用自顶向下的递归结合备忘技巧进行求解。
        else {
            memset(dp, -1, sizeof(dp));
            cout << dfs(1, n) << '\n';
        }
    }
    return 0;
}
//-----11.5.2.1.cpp-----

```

不难看出，上述参考实现的时间复杂度为 $O(n^3)$ ，其中 n 为区间的长度。可以对经典的石子合并问题的条件进行适当改变，从而得到多种变形和扩展问题。

(1) 如果不限定只能合并相邻堆的石子，而是可以任意选择两堆石子进行合并，求所能得到的最低得分。可以按照一种贪心策略来进行操作，即每次合并时，都选择具有最少石子数量的那两堆石子进行合并。可以证明，该种策略是一种最优策略，能够获得最低得分¹。

(2) 在前述的石子合并问题中，石子是从左至右排列成直线的，一般称之为线性区间动态规划。在区

¹ 在此种限制下，最优策略与霍夫曼编码的生成过程类似。参见本章第 11.11.4 小节“霍夫曼编码”的内容。

间型动态规划中有一种特殊的类型，其给定的区间构成一个首尾相连的环，称之为环形区间动态规划。在求解此类环形区间动态规划问题时，可以任意选择环形区间上的某个点 q_i 将其断开，使得环形区间变为线性区间 $Q = q_{i+1} \dots q_n q_1 \dots q_i$ ，然后将整个区间 Q 附加在原区间末尾 q_i ，使得区间成为原始区间的两倍，即 $Q' = Q Q = q_{i+1} \dots q_n q_1 \dots q_i q_{i+1} \dots q_n q_1 \dots q_i$ ，再对 Q' 重新编号得 $Q'' = q_1 \dots q_n q_{n+1} \dots q_{2n}$ ，这样即可使用常规的线性区间型动态规划算法进行求解而不会导致遗漏某个区间。

以石子合并问题为例，假设石子堆围绕着一个圆环排列，即编号为 1 的石子堆和编号为 n 的石子堆相邻，试确定此种情况下的最优得分值。那么按照断环为链的思路，可以得到以下参考实现代码。

```
//-----11.5.2.2.cpp-----
const int MAXN = 210, INF = 0x7f7f7f7f;

int dp[MAXN][MAXN], n, a[MAXN], sum[MAXN] = {0};

int dfs(int i, int j) {
    if (~dp[i][j]) return dp[i][j];
    if (i == j) return 0;
    int r = INF;
    for (int k = i; k < j; k++)
        r = min(r, dfs(i, k) + dfs(k + 1, j) + sum[j] - sum[i - 1]);
    return dp[i][j] = r;
}

int main(int argc, char *argv[]) {
    while (cin >> n) {
        for (int i = 1; i <= n; i++) {
            cin >> a[i];
            sum[i] = sum[i - 1] + a[i];
        }
        for (int i = 1; i <= n; i++) {
            a[n + i] = a[i];
            sum[n + i] = sum[n + i - 1] + a[n + i];
        }
        memset(dp, -1, sizeof(dp));
        int r = INF;
        for (int i = 1; i <= n; i++)
            r = min(r, dfs(i, n + i - 1));
        cout << r << '\n';
    }
    return 0;
}
//-----11.5.2.2.cpp-----
```

可以看到，程序结构与线性区间动态规划差别不大。使用备忘技巧可以利用之前的计算结果使得效率更高，而不是每次都在新的区间上重新计算。

(3) 如果在合并过程中，相邻两堆石子在合并时必须满足某种特定的条件（例如，要求相邻两堆石子的数目必须互质），而且会将合并后的石子堆移走，则在问题处理上有所差别。在典型的石子合并问题中，被合并的石子不会被移除，因此只需考虑边界的左端和右端即可，但是在当前限制下，如果选择两个满足条件的相邻石子堆进行操作后，它们将被移除。更为棘手的是如何处理以下情形：两个不相邻但满足条件的石子堆也可以被移除，只要这两个石子堆之间所包含的那部分石子堆能够被全部移除。那么，似乎就需要枚举区间 $[i, j]$ 内所有可能的满足条件的石子堆进行移除以获得最优值。令 $a[i]$ 表示编号为 i 的石子堆的石子数目， $dp[i][j]$ 表示区间 $[i, j]$ 的最优得分值。假设当前编号为 s 和 t 的石子堆满足条件，为了能够将编号为 s 和 t 的

石子堆移除，需要检查编号在区间 $[s+1, t-1]$ 内的这部分石子堆是否可能被完全移除。为了便于判断，可以先构建前缀和数组 sum ，即将所有石子堆按照从左到右的顺序，从 1 开始编号，令 $sum[i]$ 表示前 i 个石子堆石子数的总和，其中 $sum[0]=0$ 。那么检查区间 $[s+1, t-1]$ 这部分是否可能被完全移除，只需检查 $dp[s+1][t-1]$ 是否等于 $sum[t-1]-sum[s]$ ，即区间 $[s+1, t-1]$ 的最优值能否达到此区间所有石子堆的石子数目总和。如果能够达到，则编号为 s 和 t 的石子堆能够被移除。因此可以得到以下递推关系式

$$dp[i][j] = \max_{1 \leq i < j \leq n, i \leq s < t \leq j} \{ dp[i][s-1] + a[s] + dp[s+1][t-1] + a[t] + dp[t+1][j] \}$$

$$\text{gcd}(a[s], a[t]) = 1, dp[s+1][t-1] = sum[t-1] - sum[s]$$

边界条件：当 $i \geq j$ 时， $dp[i][j]=0$ 。根据上述递推关系式，可以得到以下参考实现。

```
-----11.5.2.3.cpp-----
const int MAXV = 110;

int n, sum[MAXV] = {0}, possible[MAXV][MAXV];
int stones[MAXV], dp[MAXV][MAXV];

int dfs(int i, int j) {
    if (i >= j) return 0;
    if (~dp[i][j]) return dp[i][j];
    int r = 0;
    for (int s = i; s < j; s++)
        for (int t = s + 1; t <= j; t++)
            if (possible[s][t])
            {
                int erased = dfs(s + 1, t - 1);
                if (erased == sum[t - 1] - sum[s])
                {
                    erased += dfs(i, t - 1);
                    erased += dfs(s + 1, j);
                    erased += a[s] + a[t];
                    r = max(r, erased);
                }
            }
    return dp[i][j] = r;
}

int main(int argc, char *argv[]) {
    cin >> n;
    for (int i = 1; i <= n; i++) {
        cin >> a[i];
        sum[i] = sum[i - 1] + a[i];
    }
    memset(possible, 0, sizeof possible);
    for (int i = 1; i <= n; i++)
        for (int j = i + 1; j <= n; j++)
            if (__gcd(a[i], a[j]) == 1)
                possible[i][j] = 1;
    memset(dp, -1, sizeof dp);
    cout << dfs(1, n) << '\n';

    return 0;
}
-----11.5.2.3.cpp-----
```

不难看出，上述实现的时间复杂度是 $O(n^4)$ ，效率较低。从递推关系式来看，不满足应用四边形不等式优化的条件，想从此处着手似乎行不通¹。因为是区间型动态规划，左右两个端点占据了状态的两个维度，只能考虑从切分点的选择这个地方来尝试降低状态维度了。进一步考虑，对于给定的某个区间 $[i, j]$ 来说，如果编号为 i 和 j 的石子堆满足移除的条件，那么只存在两种情况：一种是编号在区间 $[i+1, j-1]$ 内的石子堆能够被全部移除，那么编号为 i 和 j 的石子堆就能够移除，一种是编号在区间 $[i+1, j-1]$ 内的石子堆不能被移除，那么编号为 i 和 j 的石子堆就不可能移除。对于区间 $[i+1, j-1]$ 内石子堆不能被全部移除的这种情况来说，只可能在 $[i+1, j-1]$ 中去寻找某个石子堆作为切分点，检查切分成的两部分的最优值之和是否更优，这种情况对编号为 i 和 j 的石子堆是否能够被移除已经不会构成影响。这意味着可以将其分成两种情况进行处理。一种是中间包含的部分能够被全部移除，一种是中间包含的部分不能被全部移除，对于不能全部移除的情形则继续枚举可能的切分点，类似于经典石子合并问题中的处理方法。

因此递推关系可以更改为

$$dp[i][j] = \begin{cases} dp[i+1][j-1] + a[i] + a[j], & dp[i+1][j-1] = sum[j-1][i] \\ \max_{i \leq k < j} \{dp[i][k] + dp[k+1][j]\}, & dp[i+1][j-1] \neq sum[j-1][i] \end{cases}$$

其中 $1 \leq i < j \leq n$ ，根据上述递推关系式，进而有以下的时间复杂度为 $O(n^3)$ 的优化实现。

```
//-----11.5.2.4.cpp-----
int dfs(int i, int j) {
    if (i >= j) return 0;
    if (~dp[i][j]) return dp[i][j];
    if (possible[i][j]) {
        int erased = dfs(i + 1, j - 1);
        if (erased == sum[j - 1] - sum[i])
            return dp[i][j] = erased + a[i] + a[j];
    }
    int r = 0;
    for (int k = i; k < j; k++)
        r = max(r, dfs(i, k) + dfs(k + 1, j));
    return dp[i][j] = r;
}
//-----11.5.2.4.cpp-----
```

10891 Game of Sum^A (取数游戏)

给定一个包含 n 个整数的序列，玩家 A 和 B 轮流从该序列中取数。每名玩家可以从序列的左端（或者右端）开始取至少一个数，但不能从两端同时取。在取数时，只能从某一端取序列中的连续个整数，不能跳过序列中的某个数不取而取下一个整数。当序列中的所有数被取尽后游戏结束。每名玩家的游戏得分为所取数的和，游戏的目标是尽可能使得己方所取数的和较对方更大，从而得分更多。假如两名玩家均采取最优的游戏策略，且玩家 A 先开始取数，则最终玩家 A 可以比玩家 B 多得多少分？

输入

输入包含多组测试数据，每组测试数据的第一行为一个整数 n ($0 < n \leq 100$)，表示序列中所包含的整数个数，接着一行给出 n 个整数。输入以 $n=0$ 结束。

输出

¹ 参见本章第 11.9.6 小节“四边形不等式优化”的内容。

对于每组测试数据，输出一个整数，表示双方玩家均采用最优策略后，第一名玩家所能获得的分数和第二名玩家获得分数的最大可能差值。

样例输入

```
4
4 -10 -20 7
4
1 2 3 4
0
```

样例输出

```
7
10
```

分析

假设序列为

$$x_1, x_2, x_3, \dots, x_{n-1}, x_n$$

令 x_i 至 x_j 之间（包括 x_i 和 x_j , $1 \leq i \leq j \leq n$ ）的数的和为 $s[i][j]$ ，并设 $dp[i][j]$ 是某个玩家在区间 $[i, j]$ 所能取到的最大和，则 $s[i][j] - dp[i][j]$ 是另外一个玩家在前一个玩家取到最大和的情况下所能取的整数和，则题目所求为 $dp[1][n]$ 。由于是玩家 A 先取数，则玩家 A 可以从序列的左端（或右端）取走任意连续个整数，不妨设其从左端开始取走了 x_i 至 x_k 之间的数 ($1 \leq i \leq k \leq j \leq n$)，其和为 $s[i][k]$ 。此时序列变成

$$x_{k+1}, \dots, x_{j-1}, x_j$$

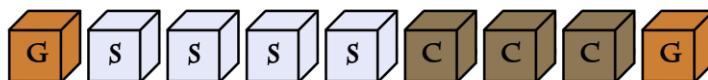
接着轮到玩家 B 取数，由于双方玩家均采取最佳游戏策略，则此时玩家 B 能够取到的最大和为 $dp[k+1][j]$ ，对应的玩家 A 所能取到的剩余最大整数和为 $s[k+1][j] - dp[k+1][j]$ ，为了使得玩家 A 所取的总和尽可能的大，则应该使 $s[i][k] + s[k+1][j] - dp[k+1][j] = s[i][j] - dp[k+1][j]$ 尽可能地大。同理，当玩家 A 从右端开始取走了 x_k 至 x_j 之间的数 ($1 \leq i \leq k \leq j \leq n$) 时，应该使得 $s[k][j] + s[i][k-1] - dp[i][k-1] = s[i][j] - dp[i][k-1]$ 尽可能地大。因此递推关系式为

$$dp[i][j] = \max\{s[i][j] - dp[i][k-1], s[i][j] - dp[k+1][j]\}, i \leq k \leq j$$

为了避免反复求区间 $[i, j]$ 的整数和，可预先将序列的前 i 项求和，令其为 $a[i]$ ，则区间 $[i, j]$ 的整数和 $s[i][j] = a[j] - a[i-1]$ ，其中 $1 \leq i \leq j \leq n$, $a[0] = 0$ 。

10559 Blocks^C (方块)

你可能玩过一种被称为“方块 (Blocks)”的游戏。给定 n 个排成一列的方块，每个方块都涂有某种颜色。例如以下 9 个方块：金色，银色，银色，银色，银色，铜色，铜色，铜色，金色。

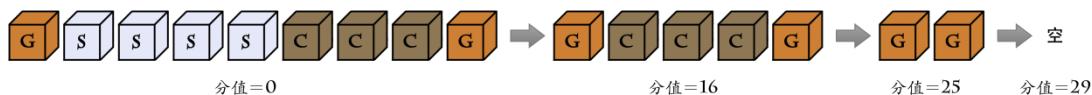


如果相邻连续几个方块都具有相同的颜色，而且在其左侧（如果存在）和右侧（如果存在）的方块具有不一样的颜色，我们将其称为“方块区段”。上图所示的方块构成了 4 个方块区段，它们依次是：金色方块区段，银色方块区段，铜色方块区段，金色方块区段。每个方块区段依次包含 1 个，4 个，3 个，1 个方块。

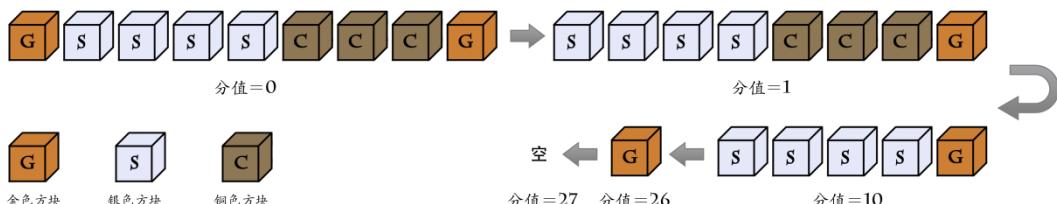
每次你可以点击一个方块，这将使得包含该方块的某个方块区段消失。如果该方块区段包含 k 个方块，你将获得 $k \times k$ 的积分。例如，当你点击银色方块时，银色方块区段将会消失，你将获得 $4 \times 4 = 16$ 的积分。

下图示例了前述给定的游戏的两种可能的玩法，其中第一种玩法是最优的，最高得分为 29 分。

一种可能的玩法：



另外一种可能的玩法：



给定初始的游戏状态，确定你能够得到的最高积分。

输入

输入的第一行是一个数值 t ($1 \leq t \leq 15$), 表示测试数据的组数。每组测试数据包含两行, 第一行包含一个整数 n ($1 \leq n \leq 15$), 表示方块的数目。第二行包含 n 个整数, 表示每个方块的颜色, 表示颜色的整数为不大于 n 的正整数。

输出

对于每组测试数据，输出测试数据的组数以及可能的最高得分。

样例输入

样例输出

2
9
1 2 2 2 2 3 3 3 1
1
1

Case 1: 29
Case 2: 1

分析

由于是多个相同颜色的方块排在一起，为了便于处理，不妨将连续的相同颜色方块看做一个假想的石子堆，该石子堆不仅具有数量还具有颜色。在此题背景下，石子堆可以被移除，计分方式也发生了改变，每次移除石子堆的得分为石子数目的平方。那么是否可以沿用之前介绍的处理石子合并问题的变形的思路呢？答案是否定的。如果按照前述介绍的思路进行处理，无法解决如下的状态：

1
6
1 2 1 1 2 1

由于可以先将颜色为 2 的方块消除，然后再将 4 个颜色为 1 的方块整体消除，最高得分为 18 分。而使用之前介绍的思路进行处理，只能找到这样的最优策略：先将中间两个颜色为 1 的方块消除，得分为 4，然后再将两个颜色为 2 的方块消除，总得分为 8，最后将剩下的两个颜色为 1 的方块消除，总得分为 12 分。动态规划状态设计是否正确，评判的一个重要指标就是分解成的子问题是否产生遗漏。若发生遗漏，就有可能使得最后的结果不是最优解。由于当前的解题思路无法产生这样的子状态——先将两个颜色为 2 的方块消除后

剩下 4 个颜色为 1 的连续方块，因此是无法正确工作的状态设计。是否能够对状态进行修改以便其能表示所有可能的状态呢？答案是肯定的。考虑给定的某个方块序列，对于最右侧的方块区段来说，它要么被消除，要么等待某个时刻和左侧相同颜色的方块连接起来组成一个更长的方块区段后再消除（与不同颜色的方块区段无法合并，因此不会影响得分），这两种情况必占其一，因此通过“最右侧的方块区段是否立即消除”来构建新的状态可以保证不会遗漏子状态。我们可以重新定义状态，令 $dp[i][j][k]$ 表示在区间 $[i, j]$ 右侧还有 k 个与方块区段 j 的颜色相同的方块时的最大得分， $clr[i]$ 表示编号为 i 的方块区段的颜色， $cnt[i]$ 表示编号为 i 的方块区段所包含的方块数目。那么，如果选择将右侧的方块移除，则得分为

$$dp[i][j][k] = dp[i][j-1][0] + (cnt[j] + k)^2$$

如果选择先不移除右侧的方块区段，那么可以往 j 的左侧寻找这样的一个方块区段 q ，方块区段 q 的颜色与方块区段 j 的颜色相同，但方块区段 $q+1$ 的颜色与方块区段 j 的颜色不同，通过移除多个方块区段 $[q+1, j-1]$ ，可以使得右侧尚未移除的方块区段 j 以及 k 个与方块区段 j 同样的方块，能够以一个整体的形式附加在方块区段 q 的右侧，构成一个更长的方块区段，以便后续的消除。在此种情况下，有

$$dp[i][j][k] = \max_{i \leq q < j} \{ dp[i][q][cnt[j] + k] + dp[q+1][j-1][0], clr[q] = clr[j] \text{ 且 } clr[q+1] \neq clr[j] \}$$

综合两种情形，不难得到

$$dp[i][j][k] = \max \begin{cases} dp[i][j-1][0] + (cnt[j] + k)^2 \\ \max_{i \leq q < j} \{ dp[i][q][\text{clr}[j] + k] + dp[q+1][j-1][0], clr[q] = clr[j], clr[q+1] \neq clr[j] \} \end{cases}$$

边界条件： $dp[i][i] = (cnt[i])^2$ ；当 $i > j$ 时， $dp[i][j] = 0$ 。根据上述递推关系式，结合备忘技巧解题即可。

强化练习：[1211 Atomic Car Race^D](#)，[10201 Adventures in Moving: Part IV^B](#)，[10688* The Poor Giant^C](#)，[10954 Add All^A](#)。

扩展练习：[662 Fast Food^C](#)，[970 Particles^D](#)。

11.6 图论型动态规划

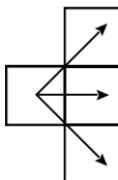
图论型动态规划是指以图为背景的动态规划。在第 10 章“图算法”中，介绍了求单源最短路径的 Moore-Dijkstra 算法以及求所有顶点对之间最短距离的 Floyd-Warshall 算法，这两种算法本质上都是动态规划思想在图上的应用。对于一般的动态规划问题，如果将状态视为图的一个顶点，则最终都可以将问题建模为一个有向无圈图，各个状态根据递推关系式在顶点之间转移，顶点之间的边权就是状态转移的代价。从这个角度来说，动态规划算法的实质就是在图上求函数最优值的过程，并且在此过程中加入了“备忘”技巧以避免重复计算^I。

一般来说，为了增加问题的挑战性，在竞赛中与图论相关的动态规划题目大多给出的是隐式图，需要解题者根据题目约束构建相应的显式图以进一步求解（有时可能并不需要显式地将图予以表示）。隐式图所对应的可能是无向图，也可能是有向图，具体由题目的约束条件所决定。其中较为常见的一类题目是以网格为背景来设置约束条件，因为网格本身就是一种比较特殊的图。如果将网格中的单个方格视为图的顶点，按照行走方式的限制，若从某个方格只能朝上、下、左、右四个方向移动，则每个顶点和其他顶点之间至多存在四条边，若能够沿对角线行走，则最多存在八条边。

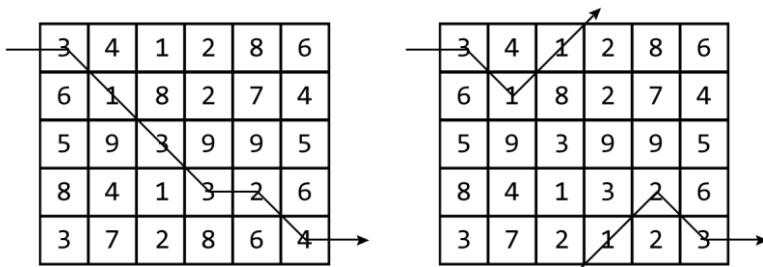
116 Unidirectional TSP^A（单向旅行商问题）

^I 在本章的第 11.3 节“松弛”中介绍了松弛技术在图论算法中的应用。松弛技术和动态规划关系密切。

给定一个 $m \times n$ 的数字矩阵，编写程序找出一条从左至右走过矩阵且权和最小的路径。一条路径可以从数字矩阵第 1 列的任意位置出发，到达第 n 列的任意位置结束。从第 i 列的某行（沿水平或者 45 度斜线）走到第 $i+1$ 列的某行视为移动一步。第一行和最后一行看作是相邻的，即你应当把这个矩阵想象成是一个沿着水平方向卷起来的圆筒。如下是合法的走法：



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 5×6 矩阵（只有矩阵中最下面一行的数不同）的最小权和路径如下图所示。右侧矩阵的最小权和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。每个矩阵描述的第一行为两个数 m 和 n ，分别表示矩阵的行数和列数。接下来的 $m \times n$ 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，依此类推。相邻整数间用一个或多个空格隔开。注意，给定数据可能包含负整数。输入中可能有一个或多个矩阵描述，直到输入结束。每个矩阵的行数在 1 到 10 之间，列数在 1 到 100 之间。路径的权和不会超过 30 位二进制数所能表示的范围。

输出

对每个矩阵输出两行。第一行为具有最小权和的路径，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
```

样例输出

```
1 2 3 4 4 5
16
```

分析

由于在右侧一列的每个方格只能接受来自其左上方、左方、左下方的方格所走过来的路线，若选择这三

条路线中权和最小的路线，则加上右侧的“当前方格”本身的权值，路线的总权和对于到达“当前方格”的路径来说，其权和仍然是当前最优的，因此问题具有最优子结构和重叠子问题的性质。令 $M(i, j)$ 表示第 i 行第 j 列的方格所能得到的最小权值， $g[i][j]$ 表示第 i 行第 j 列的方格中的元素值（从 0 开始计数行和列），则有以下递推关系式

$$M(i, j) = g[i][j] + p(i, j)$$

$$p(i, j) = \min\{M((i-1+m)\%m, j-1), M(i, j-1), M((i+1)\%m, j-1)\}$$

其中 $0 \leq i < m$, $0 < j < n$ 。由于题目附加了额外限制——第一行和最后一行“相邻”且有多种解时输出字典序最小的路径，因此需要注意递推方向的选择。如果从左往右进行递推，在确定了最后一列方格的最小权和后，如果从中选取具有最小权和且行号最小的某一行，这并不能保证输出的一定是字典序最小的路径。例如，给定以下测试数据：

7	7					
1	1	1	9	9	9	9
9	9	9	9	9	1	1
9	9	9	9	1	9	9
9	9	9	1	9	9	9
9	9	1	9	9	9	9
1	1	9	9	9	9	9
9	9	9	1	1	1	1

如果按照从左至右的方向进行递推，并以最后一列具有最小权和且行号最小的“某一行”为准，会得到以下解：

6	6	5	4	3	2	2
7						

这显然是不正确的。而从右往左进行递推，在选择具有最小权和方格的基础上，同时选择最小的行号以保证字典序，则能够得到符合题意的输出。因为在递推结束时，位于第 1 列，此时选择具有最小权和且行号最小的行，必定是满足题意要求的。由此可以得到正确的解：

1	1	1	7	7	7	7
7						

1600 Patrol Robot^c (巡逻机器人)

一个机器人需要围绕某个尺寸为 $m \times n$ 网格（即 m 行 n 列，在为行和列计数时，行从 1 到 m ，列从 1 到 n ）的矩形区域进行巡逻。单元格 (i, j) 表示网格中位于第 i 行第 j 列的方格。每一步，机器人只能从一个单元格移动到相邻的单元格中，例如，从单元格 (x, y) 移动到单元格 $(x+1, y)$, $(x, y+1)$, $(x-1, y)$, $(x, y-1)$ 之中的某一个。某些单元格可能包含障碍。为了移动到包含障碍的单元格中，机器人需要切换到“高速模式”，而在“高速模式”下，机器人连续经过的障碍数量不能超过 k 个。你的任务是编写程序，寻找从单元格 $(1, 1)$ 出发到达单元格 (m, n) 的最短路径（经过最少数量方格的路径），你可以假定出发单元格和终止单元格均不包含障碍。

输入

输入包含多组测试数据。输入的第一行包含一个不大于 20 的正整数，表示测试数据的组数。每组测试数据的格式描述如下。对于每组测试数据，第一行包含两个由空格分开的正整数 m 和 n ($1 \leq m, n \leq 20$)，

第二行包含一个整数 k ($0 \leq k \leq 20$), 接下来的 m 行中, 第 i 行包含 n 个由空格分隔的整数 a_{ij} ($i=1, 2, \dots, m$; $j=1, 2, \dots, n$), 如果单元格(i, j)包含障碍, 则 a_{ij} 为 ‘1’, 否则为 ‘0’。

输出

对于每组测试数据, 假如存在从单元格(1, 1)到达单元格(m, n)的路径, 输出整数 s , 表示机器人需要移动的最少步数, 否则输出 “-1”。

样例输入

```
3
2 5
0
0 1 0 0 0
0 0 0 1 0
4 6
1
0 1 1 0 0 0
0 0 1 0 1 1
0 1 1 1 1 0
0 1 1 1 0 0
2 2
0
0 1
1 0
```

样例输出

```
7
10
-1
```

分析

网格是一种特殊的图, 可以将单元格视为图的顶点从而得到显式的图表示。按题目条件约束, 机器人可以往上、下、左、右四个方向移动, 则图中的顶点至多有四条无向边与其他顶点连接 (处于边界的单元格所对应的顶点只有三条或者两条无向边与其他顶点相连)。从起始顶点出发, 令到达的顶点为 u , 与 u 邻接的顶点为 v , 那么可以容易理解: 到达顶点 u 的最短路径必定经过 u 的某个邻接顶点 v 。也就是说, 到达顶点 v 的最短路径的最优值再增加一步即为到达顶点 u 的最短路径。换句话说, 最短路径问题符合动态规划的最优化原则。由于机器人连续经过的障碍数不能超过 k 个, 那么还需要为最短路径的状态增加一个参数 z 来表示到达某个单元格时已经连续经过的障碍数目。令 $d[x][y][z]$ 为到达单元格(x, y)且已经连续经过 z 个障碍时的最短路径, 那么递推关系式为

$$d[x][y][z] = \min\{d[x_0][y_0][z_0]\}$$

约束条件为

$$1 \leq x_0, x \leq m; 1 \leq y_0, y \leq n; 0 \leq z_0, z \leq k; z = a_{x_2 y_2} \times (z_0 + 1); |x_0 - x| + |y_0 - y| = 1$$

其中的约束条件

$$z = a_{x_2 y_2} \times (z_0 + 1)$$

利用了一个小技巧, 其含义如下: $z_0 + 1$ 表示将此前经过的障碍数增加 1, 若当前方格包含障碍, 则 $a_{x_2 y_2}$ 为 1, 两者相乘即为当前总共已经经过的障碍数; 若当前方格不包含障碍, 则 $a_{x_2 y_2}$ 为 0, 两者相乘为 0, 当前经过的障碍数重置为 0。

当机器人到达目标单元格(m, n)时, 可能已经连续经过了 $0, 1, \dots, k$ 个障碍, 因此最短路径为

$$s = \min\{d[m][n][z_0], 0 \leq z_0 \leq k\}$$

具体实现时可以使用 BFS 来进行最短路径的更新。

强化练习: [590 Always on the Run^B](#), [10047 The Monocycle^B](#), [10702 Travelling Salesman^B](#), [10913 Walking on a Grid^C](#)。

扩展练习: [976 Bridge Building^D](#), [1399* Puzzle^E](#), [11545* Avoiding Jungle in the Dark^D](#), [12030* Help the Winner^D](#)。

11.6.1 路径计数

给定如图 11-9 所示的网格, 以左下角 S 为起点, 右上角 E 为终点, 规定只能向上或者向右行走, 试确定从 S 到 E 的不同路径计数。

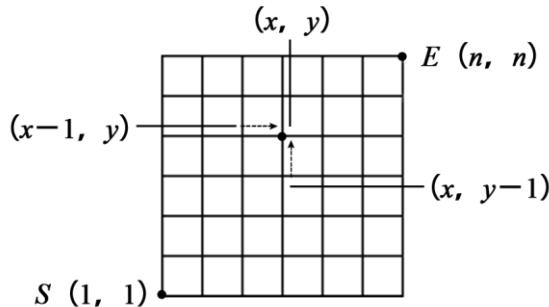


图 11-9 从网格的左下角 S 按照指定规则 (只能向上或者向右, 不能走对角线) 走到右上角 E 的不同路径计数

令 $path(x, y)$ 表示从左下角格点 S 到达任意格点 (x, y) 的不同路径数, 由于每个点只能接受来自左方或者下方的路径, 可以得到递推关系

$$path(x, y) = path(x - 1, y) + path(x, y - 1)$$

对于左边界和下边界上的格点, 它们只能接受来自下方或者左方的路径。如果题目中加以额外限制——在格点的左侧或者下方有障碍不能通过时相应方向的路径数计为 0, 那么, 与正交范围查询的思路类似, 可以使用行优先顺序从网格的左下角开始填充路径计数矩阵, 从而进一步计算路径计数。

类似的, 在图论中有一类问题称为路径计数问题。此类问题要求确定从图中的某个顶点到达另外一个顶点的不同有向 (最短) 路径数量, 两条路径若至少有一条边不同则认为是不同的路径。对于有向图来说, 题目给定的条件一般是无圈图, 要求确定不同的有向路径数, 对于无向图, 一般是要求确定给定的两个顶点间的不同最短路径数。对于此类问题, 可以使用前述介绍的在网格上进行路径计数的方法予以解决。具体来说, 就是沿着构建的有 (无) 向图, 经过顶点 u 的不同有向 (最短) 路径数等于经过其子顶点的 v 的不同有向 (最短) 路径数之和, 即

$$path(v) = \sum_{\substack{u \\ (u, v) \in E}}^u path(u)$$

在实际求解时, 需要应用备忘技巧来提高效率。

强化练习: [825 Walking on the Safe Side^A](#), [926 Walking Around Wisely^C](#), [10544* Numbering the Paths^D](#), [10564 Paths Through the Hourglass^C](#), [11067 Little Red Riding Hood^C](#), [11133 Eigensequence^D](#), [11569 Lovely Hint^D](#), [12967 Spray Graphs^D](#)。

扩展练习: [910 TV Game^C](#), [950 Tweedle Numbers^E](#), [10401* Injured Queen Problem^B](#), [10917 A Walk Through the Forest^C](#), [11125* Arrange Some Marbles^D](#), [11432* Busy Programmer^D](#), [11487* Gathering Food^D](#),

11655 Water Land^D, 11957 Checkers^C。

给定一个有向无圈图 D , 试确定从指定顶点 u 开始到达顶点 v 长度为 k 的不同路径计数。令 $f(u, v, k)$ 表示 D 中从顶点 u 到顶点 v 长度为 k 的不同路径计数, w 为某个中间顶点, 根据路径的性质, 可以得到以下递推关系式:

$$f(u, v, k) = \begin{cases} 1 & \text{若 } u = v \text{ 且 } k = 0 \\ 0 & \text{若 } u \neq v \text{ 且 } k = 0 \\ \sum_{(w, v) \in E} f(u, w, k-1) & k \geq 1 \end{cases}$$

如果直接按照上述递推关系式进行计算, 则由于存在递归, 时间复杂度为 $O(kV^3)$ 。为了加快求解速度, 可以应用类似于快速幂的技巧, 使时间复杂度下降到 $O(V^3 \log k)$ 。令 A 和 B 均为 $|V| \times |V|$ 的矩阵, $A[i][j]$ 表示图 D 中从顶点 i 到顶点 j 长度为 k 的不同路径计数, $B[i][j]$ 表示图 D 中从顶点 i 到顶点 j 长度为 $k+1$ 的不同路径计数, M 为有向图 D 的邻接矩阵, 有

$$B = A \times M = M^k$$

使用矩阵快速幂技巧, 可以先递归计算 $M^{k/2}$, 然后相乘得到 M^k 。

12796 Teletransport^D (超时空传送)

银河联盟在宇宙飞船上安装了新的传送系统。每艘飞船都接收了一个传送仓, 仓内的操作面板有四个按钮。四个按钮依次使用字母 A、B、C、D 进行标记, 每个按钮还关联了一个数字, 该数字表示飞船的编号, 按下某个按钮, 用户将被立即传送到按钮所关联编号的飞船上 (总所周知, 联盟的宇宙飞船使用 1 到 N 的编号来进行区分)。

为了使用该传送系统, 用户必须为其进行的每次旅程买票 (亦即每按一次按钮就相当于一次旅程), 注意到面板上的按钮数量相对于联盟的飞船数量来说是很小的, 因此用户可能需要购买能够进行 L 次旅程的联票以便从编号为 S 的飞船传送到编号为 T 的飞船。

例如, 对于如下图所示的三艘宇宙飞船来说, 如果位于编号为 3 的飞船内的用户按下按钮 B, 那么他将被传送到编号为 2 的飞船上, 如果他有联票并且按下编号为 2 的飞船上的按钮 B, 那么他将被传送到编号为 1 的飞船上。

对于本问题来说, 你的任务是给定编号为 S 的起始飞船和编号为 T 的目标飞船, 旅程次数为 L 的联票, 确定从飞船 S 到达飞船 T , 长度为 L 的不同按钮按下的序列数量。例如, 对于上述所示的三艘飞船, 总共有 4 种不同的长度为 2 的按钮按下序列, 使得用户能够从编号为 3 的飞船到达编号为 1 的飞船: CD, DA, AB, BB。

输入

输入包含多组测试数据。每组测试数据的第一行包含两个整数 N ($1 \leq N \leq 100$) 和 L ($0 \leq L \leq 2^{30}$), 分别表示飞船的数量和联票所能进行的旅程次数。每组测试数据的第二行包含两个整数 S 和 T ($1 \leq S, T \leq N$), 分别表示起始飞船和终止飞船的编号。接下来的 N 行描述了每艘飞船传送仓面板上按钮的状态。第 i ($1 \leq i \leq N$) 行包含四个整数 A, B, C, D ($1 \leq A, B, C, D \leq N$), 表示编号为 i 的飞船内的四个按钮依次所关联的飞船的编号。

输出

对于输入中的每组数据输出一行，包含一个整数，该整数表示从编号为 S 的飞船，通过 L 次旅程，最终到达编号为 T 的飞船不同的路径序列数量模 10^4 的值。

样例输入

```
3 2
3 1
1 2 2 2
2 1 3 2
2 2 3 1
```

样例输出

```
4
```

分析

以样例输入为例，总共有三艘飞船，将从某艘飞船出发，经过 1 次旅程到达其他飞船的不同路径数表示成一个矩阵 M ，则有：

$$M = \begin{bmatrix} 1 & 2 & 0 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix}$$

其含义是：矩阵 M 的第 i 行第 j 列元素的值表示从编号为 i 的飞船出发，经过 1 次旅程，到达编号为 j 的飞船的不同路径数。如果将矩阵 M 与自身相乘，那么所得结果为：

$$M^2 = M \times M = \begin{bmatrix} 1 & 2 & 0 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 2 & 0 \\ 1 & 2 & 1 \\ 1 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 3 & 6 & 2 \\ 4 & 8 & 3 \\ 4 & 8 & 3 \end{bmatrix}$$

此时矩阵的第 i 行第 j 列元素的值表示从编号为 i 的飞船出发，经过 2 次旅程，到达编号为 j 的飞船的不同路径数。同理，将矩阵 M 自乘 L 次，所得的结果矩阵的第 i 行第 j 列元素的值表示从编号为 i 的飞船出发，经过 L 次旅程，到达编号为 j 的飞船的不同路径数。如果 L 等于 0，则不能到达其他飞船，只能停留在起始飞船上，因此有一个单位矩阵来表示此种状态，对于样例输入来说，该矩阵为：

$$I = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

扩展练习：[11486 Finding Paths in Grid^D](#)。

11.6.2 树形动态规划

由于树结构的特殊性，可以很方便地应用动态规划的思维——父结点的最优值取决于其子结点的最优值，令 $dp[u]$ 表示结点 u 的最优值， $w[u]$ 表示选择结点 u 的代价，那么树形动态规划的递推关系式一般为

$$dp[u] = \max_{\text{parent}[v]=u} \text{或} \min\{dp[v] + w[u]\}$$

对于简单类型的树形动态规划，结合基本的递推关系式和相应的约束条件，一般都能够顺利解决。

强化练习：[12186 Another Crisis^C](#)，[12862* Intrepid Climber^D](#)。

扩展练习：[1222* Bribing FIPA^D](#)，[11307* Alternative Arborescence^D^{\[141\]}](#)，[11782* Optimal Cut^D](#)。

对于一般图来说，最小点支配、最小点覆盖、最大点独立均为 NP 问题，无有效算法。但是对于树这种特殊类型的图，存在有效的动态规划算法。

最小点支配

对于图 $G=(V, E)$ ，从 V 中选取若干顶点构成 V 的一个子集 V_1 ，使得 $V \setminus V_1$ 中所有顶点均与 V_1 中的顶点有边相连，则称 V_1 为 G 的顶点支配集，简称点支配。若 V_1 是图 G 的一个顶点支配集，则对于图中任意

一个顶点 u , 要么 u 属于 V_1 , 要么 u 与 V_1 中的某个顶点 v 有边关联。若在 V_1 中任意移除一个顶点后, V_1 不再构成顶点支配集, 则 V_1 是极小顶点支配集, 简称极小点支配。将 G 的所有顶点支配集中顶点个数最少的支配集称为最小顶点支配集, 简称最小点支配。

在树的最小点支配问题中, 对于给定的结点 u , 可能存在三种状态:

- (a) 结点 u 属于支配集, 则与 u 有边相连的结点 v 均被 u 所支配;
- (b) 结点 u 不属于支配集, 但与 u 有边相连的结点 v 中至少有一个结点属于支配集, 则 u 已被支配;
- (c) 结点 u 不属于支配集, 且与 u 有边相连的结点 v 均不属于支配集, 即结点 u 及其邻接结点 v 均被其他结点所支配。

根据上述三种可能的情形, 设计以下三种状态:

- (a) $dp[u].in$ 表示结点 u 属于支配集且以 u 为根的子树均被支配的情况下最小点支配所包含的结点个数;
- (b) $dp[u].selfOut$ 表示结点 u 不属于支配集但 u 被其中不少于一个子结点支配的情况下最小点支配所包含的结点个数;
- (c) $dp[u].selfChildOut$ 表示结点 u 不属于支配集且 u 的任意一个子结点也不属于支配集, 但以 u 为根的子树均被支配的情况下, 其最小点支配中所包含的结点个数。

对于第一种状态, 由于结点 u 属于支配集, 则 $dp[u].in$ 为每个子结点三种状态最小值的总和再增加 1, 即只要每个以 u 的儿子为根的子树均被支配, 再加上当前结点 u , 即为所需要的最少结点个数, 递推关系式如下:

$$dp[u].in = 1 + \sum_{parent[v]=u} \min\{dp[v].in, dp[v].selfOut, dp[v].selfChildOut\}$$

其中 $parent[v]=u$ 表示 v 的父结点为 u , 即 v 是 u 的子结点。

对于第二种状态, 如果结点 u 无子结点, 则 $dp[u].selfOut=INF$; 否则, 需要保证它的每个以 u 的儿子为根的子树均被支配, 那么要取每个子结点的前两种状态的最小值之和, 因为此时 u 不属于支配集, 不能支配其子结点, 所以子结点必须已经被支配, 与子结点的第三种状态无关。如果当前所选的状态中, 每个儿子都没有被选择进入支配集, 即在每个儿子的前两种状态中, 第一种状态都不是所需结点数最小的, 那么为了满足第二种状态的定义, 需要重新选择点 u 的一个子结点 v 的状态为第一种状态, 此时取差值最少的一个点, 即取 $dp[v].in-dp[v].selfOut$ 最小的子结点 v , 强制取其第一种状态, 其他的子结点取第二种状态, 递推关系式为

$$dp[u].selfOut = \begin{cases} INF & u \text{ 无子结点} \\ x + \sum_{parent[v]=u} \min(dp[v].in, dp[v].selfOut) & u \text{ 有子结点} \end{cases}$$

其中

$$x = \begin{cases} 0 & dp[v].in \text{ 已计入 } dp[u].in \\ \min_{parent[v]=u} \{dp[v].in - dp[v].selfOut\} & dp[v].in \text{ 未计入 } dp[u].in \end{cases}$$

对于第三种状态, u 不属于支配集且以 u 为根的子树均被支配, 但由于 u 未被任何子结点支配, 即结点 u 及其任意子结点 v 均不属于支配集, 则结点 u 的第三种状态只与其子结点 v 的第二种状态有关, 其递推关系式为:

$$dp[u].selfChildOut = \begin{cases} 0 & u \text{ 无子结点} \\ \sum_{parent[v]=u} dp[v].selfOut & u \text{ 有子结点} \end{cases}$$

最后所求的是根结点在三种状态下的最小值，令 $root$ 表示树的根结点，由于根结点不属于支配集且根结点未被任意子结点所支配，所以 $dp[root].selfChildOut$ 不符合最小支配集的定义，故只需取根结点前两种状态的最小值，即选择 $dp[root].in$, $dp[root].selfOut$ 中最小的值作为问题的解。

```
-----11.6.2.cpp-----
const int MAXV = 10010, INF = 0x3f3f3f3f;

// 链式前向星表。
struct EDGE {
    int u, v, next;
    EDGE (int u = 0, int v = 0, int next = 0): u(u), v(v), next(next) {}
} edges[MAXV << 1];

// 结点状态。
struct NODE {
    int in, selfOut, selfChildOut;
} dp[MAXV];

int idx, head[MAXV];

// 添加边。
void addEdge(int u, int v) {
    edges[idx] = EDGE(u, v, head[u]);
    head[u] = idx++;
    edges[idx] = EDGE(v, u, head[v]);
    head[v] = idx++;
}

// 使用深度优先遍历进行动态规划，从根结点开始调用 dfs(0, 0)。
void dfs(int u, int father) {
    // 设置初始值。
    dp[u].in = 1;
    dp[u].selfOut = dp[u].selfChildOut = 0;
    int x = INF, hasChild = 0, hasFirstStateIncluded = 0;
    // 遍历当前结点的子结点。
    for (int i = head[u]; ~i; i = edges[i].next) {
        int v = edges[i].v;
        if (v == father) continue;
        dfs(v, u);
        // 标记当前结点具有子结点。
        hasChild = 1;
        // 第一种状态。
        dp[u].in += min(dp[v].in, min(dp[v].selfOut, dp[v].selfChildOut));
        // 第二种状态。
        dp[u].selfOut += min(dp[v].in, dp[v].selfOut);
        if (dp[v].in > dp[v].selfOut) x = min(x, dp[v].in - dp[v].selfOut);
        else hasFirstStateIncluded = 1;
        // 第三种状态，注意避免溢出。
        dp[u].selfChildOut = min(INF, dp[u].selfChildOut + dp[v].selfOut);
    }
}
```

```

// 根据当前结点的状态修正结果。
if (!hasChild) dp[u].selfOut = INF;
else {
    if (!hasFirstStateIncluded) dp[u].selfOut += x;
}
//-----11.6.2.cpp-----//

```

强化练习: 1218* Perfect Service^D。

最小点覆盖

对于图 $G=(V, E)$, 从 V 中选取若干顶点构成 V 的一个子集 V_1 , 使得 E 中所有边均和 V_1 中的顶点相关联, 则称 V_1 为顶点覆盖集, 简称点覆盖。若 V_1 是图 G 的一个顶点覆盖集, 则对于图中任意一条边 (u, v) , 要么 u 属于 V_1 , 要么 v 属于 V_1 。若在 V_1 中任意移除一个顶点后, V_1 不再构成顶点覆盖集, 则 V_1 是极小顶点覆盖集, 简称极小点覆盖。将 G 的所有顶点覆盖集中顶点个数最少的覆盖集称为最小顶点覆盖集, 简称最小点覆盖。

对于树的最小点覆盖问题, 可以给每个结点设计两种状态:

(a) $dp[u].in$ 表示结点 u 属于点覆盖, 且以结点 u 为根的子树中所有边均被覆盖的情况下点覆盖中所包含的最少结点数。

(b) $dp[u].out$ 表示结点 u 不属于点覆盖, 但以结点 u 为根的子树中所有边均被覆盖的情况下点覆盖中所包含的最少结点数;

对于第一种状态 $dp[u].in$, 由于结点 u 属于点覆盖, 则结点 u 的子结点可以属于点覆盖, 也可以不属于点覆盖, 所取的子结点状态应该是所表示的点覆盖个数较小的数值, 其递推关系式为

$$dp[u].in = 1 + \sum_{parent[v]=u} \min\{dp[v].in, dp[v].out\}$$

其中 $parent[v]=u$ 表示 v 的父结点为 u , 即 v 是 u 的子结点。

对于第二种状态 $dp[u].out$, 由于结点 u 不属于点覆盖, 根据点覆盖的定义, 结点 u 的子结点必须属于点覆盖, 因此结点 u 的第二种状态只与其子结点的第一种状态有关, 其递推关系式为

$$dp[u].out = \sum_{parent[v]=u} dp[v].in$$

最后, 由于求的是每个结点在两种状态下的最小值, 在确定问题的解时, 需要取根结点两种状态的较小值, 即选择 $dp[root].in$ 和 $dp[root].out$ 中较小的值作为问题的解。

强化练习: 10859* Placing Lampposts^D。

最大点独立

对于图 $G=(V, E)$, 从 V 中选取若干顶点构成 V 的一个子集 V_1 , 使得这些顶点之间没有边相连, 则称 V_1 为顶点独立集, 简称点独立。若 V_1 是图 G 的一个顶点独立集, 则对于图中任意一条边 (u, v) , u 和 v 不能同时属于集合 V_1 , 甚至 u 和 v 都不属于集合 V_1 。在 V_1 中增加任意不属于 V_1 的顶点后, V_1 不再构成顶点独立集, 则 V_1 是极大顶点独立集, 简称极大点独立。将 G 的所有顶点独立集中顶点个数最多的独立集称为最大顶点独立集, 简称最大点独立。

对于树的最大点独立问题, 可以为每个结点设计两种状态:

(a) $dp[u].in$ 表示结点 u 属于点独立时, 以 u 为根的子树其最大点独立中结点的个数。

(b) $dp[u].out$ 表示结点 u 不属于点独立时, 以 u 为根的子树其最大点独立中结点的个数;

对于第一种状态 $dp[u].in$, 由于结点 u 属于点独立, 根据点独立的定义, 结点 u 的子结点均不属于点独立, 因此结点 u 的第一种状态只与其子结点的第二种状态有关, 其递推关系式为

$$dp[u].in = 1 + \sum_{parent[v]=u} dp[v].out$$

其中 $parent[v]=u$ 表示 v 的父结点为 u , 即 v 是 u 的子结点。

对于第二种状态 $dp[u].out$, 由于结点 u 不属于点独立, 则结点 u 的子结点可以属于点独立, 也可以不属于点独立, 所取的子结点的状态应该是所表示的点独立个数较大的数值, 其递推关系式为

$$dp[u].out = \sum_{parent[v]=u} \max\{dp[v].in, dp[u].out\}$$

最后, 由于求的是每个结点在两种状态下的最大值, 在确定问题的解时, 需要取根结点两种状态的较大值, 即选择 $dp[root].in$ 和 $dp[root].out$ 中较大的值作为问题的解。

强化练习: [1220 Party at Hali-Bula^D](#)。

11.6.3 旅行商问题

在第 10 章“图算法”中, 介绍了旅行商问题 (Traveling Salesman Problem, TSP)。TSP 是指给定连通图 $G=(V, E)$ 及图中所有顶点间的最短距离矩阵 g , 试确定这样一条具有最短距离的路径——该路径的起点和终点相同且经过所有顶点恰好一次。当图中的顶点数量 n 较小时 (例如 $n \leq 10$) 可以使用回溯法构建所有可能的路径并从中选择最优的路径, 适当使用剪枝技巧能够提高解题效率, 其时间复杂度为 $O(n!)$ 。当 $n \geq 16$ 时, 朴素的回溯法实现将难以避免地会导致超时。

假设 TSP 所对应的隐式图共有 12 个顶点, 分别为 A, B, C, \dots, L , 如果使用回溯法解题, 需要考虑从顶点 A 出发的所有路径, 容易知道, 从顶点 A 出发的路径可以分为两类: “ $A-B-C-[剩余 9 个顶点构成的路径]$ ” 和 “ $A-C-B-[剩余 9 个顶点构成的路径]$ ”。不难看出, 两类路径存在共同的子路径—— “[剩余 9 个顶点构成的路径]”, 从动态规划的角度来看, 这表明 TSP 存在重叠的子问题, 因此可以尝试使用动态规划来求解。由于 TSP 的起点和终点相同, 以哪个顶点作为起始顶点对问题的结果不会产生影响, 因此可以考虑总是以编号为 0 的顶点作为路径的起点, 这样在定义状态时只需要考虑两个参数, 一个是已经访问过的顶点集合, 另外一个是最后访问的顶点。令 $dp[mask][u]$ 表示已经访问的顶点集合的位掩码为 $mask$, 最后访问的顶点为 u 时, 距离起始顶点 0 尚剩余的最短路径, 有以下递推关系式

$$dp[mask][u] = \begin{cases} g[u][0], & mask = (1 \ll n) - 1 \\ \min\{g[u][v] + dp[mask \cup (1 \ll v)][v]\}, & v \neq u, mask \cap (1 \ll v) = 0 \end{cases}$$

根据上述递推关系式, 结合备忘技巧, 可以得到以下核心实现代码。

```
const int INF = 0x3f3f3f3f;

// n 为顶点数量, g 为所有顶点间最短距离矩阵。
int n, g[MAXV][MAXV];

// 动态规划算法, 已经访问的顶点集合的位掩码为 mask, 最后顶点为 u。
int dfs(int mask, int u) {
    if (mask == (1 << n) - 1) return g[u][0];
    if (~dp[mask][u]) return dp[mask][u];
    int r = INF;
    for (int v = 0; v < n; v++)
        if (~mask & (1 << v)) r = min(r, g[u][v] + dfs(mask | (1 << v), v));
    dp[mask][u] = r;
    return r;
}
```

```

    if (v != u && !(mask & (1 << v)))
        r = min(r, g[u][v] + dfs(mask | (1 << v)), v);
    return dp[mask][u] = r;
}

```

由于任意选择起点对最后结果不影响, 一般选择序号为 0 的顶点作为起始顶点进行动态规划, 此时位掩码 $mask$ 为 1。使用动态规划算法求解 TSP 的时间复杂度为 $O(n2^n)$, 对于 $n \leq 16$ 的情形可以在时间限制内获得 Accepted (一般命题者会将图中顶点的数量上限控制在 16 个左右)。

强化练习: [10937* Blackbeard the Pirate^D](#), [10944 Nuts for Nuts^C](#), [11405* Can U Win^D](#), [11813 Shopping^D](#), [11795* Mega Mans Missions^C](#)。

扩展练习: [1281 Bus Tour^D](#)。

11.6.4 双调欧几里得旅行商问题

欧几里得旅行商问题是 TSP 的一种, 它指的是给定平面上的 n 个点, 要求确定一条连接这 n 个点的最短闭合旅程。欧几里得旅行商问题的一般形式是 NP 完全的。如果对欧几里得旅行商问题的条件做适当改变, 只考虑双调旅程 (bitonic tour), 即旅程从位于最左的点出发, 严格的从左到右直到抵达最右点, 然后再严格的从最右至左, 最终返回最左点, 在此过程中, 所有点仅访问一次。经过上述限定后的问题称为双调欧几里得旅行商问题 (Bitonic TSP, BTSP), BTSP 存在时间复杂度为 $O(n^2)$ 的算法^[142]。

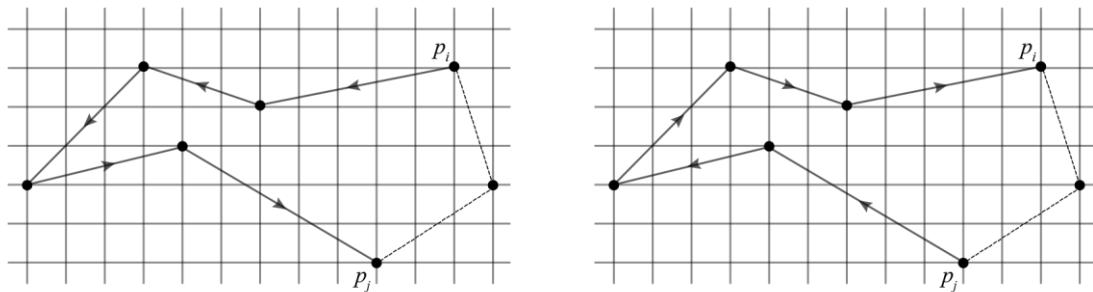


图 11-10 双调欧几里得旅程的对称性

设平面上的 n 个点为 p_1, p_2, \dots, p_n , 令 $d(i, j)$ 表示 p_i 和 p_j 之间的欧几里得距离, $B[i, j]$ 表示以 p_i 为向左旅程的起点, 以 p_j 为向右旅程的终点的双调最短旅程的长度, $1 \leq i < j \leq n$ 。双调旅程具有以下两个“有趣”的性质: (1) 旅程是对称的; (2) 将旅程从最左点开始向右到达最右点的部分称为右向旅程, 从最右点折返到达最左点的部分称为左向旅程, 则旅程的左右两个部分除了在最左和最右两个端点相交外, 在其他内部点不相交。第一个性质容易理解。如图 11-10 所示, 假设确定了以 p_i 为向左旅程起点, p_j 为向右旅程终点的最短旅程, 则反过来, 沿着原旅程的路线, 以 p_j 为向左旅程起点, p_i 为向右旅程终点的旅程, 同样是最短的。

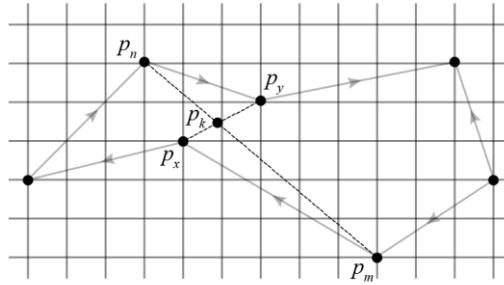


图 11-11 双调欧几里得旅程的内部点不相交

第二个性质可以根据三角形边的不等式得出。如图 11-11 所示，假设最短双调旅程中向左路径的 $p_m p_n$ 和向右路径的 $p_x p_y$ 相交，令交点为 p_k ，根据三角形任意两条边的边长之和大于第三边边长的性质，有

$$|p_m p_n| + |p_x p_y| = |p_m p_k| + |p_k p_n| + |p_x p_k| + |p_k p_y| > |p_m p_x| + |p_n p_y|$$

再结合双调旅程的对称性质，可知新的旅程将比原有旅程具有更短的距离，这与原有旅程是最短旅程相矛盾，因此最短旅程的左向路径和右向路径除了在最左和最右两个端点相交外，在内部点是不相交的。

以上述两个性质为基础，可以使用动态规划算法在 $O(n^2)$ 的时间复杂度确定双调最短旅程。为了简化问题的处理，假设给定的 n 个点其横坐标均不相同，同时根据双调旅程的对称性，为了便于算法的介绍，将“从位于最右侧的点出发到达最左侧的点的旅程”称为左向子路径，“从位于最左侧的点出发到达最右侧的点的旅程”称为右向子路径。算法的第一步先将给定的 n 个点按照横坐标升序排列。假设排序后，点从左至右依次为 p_1, p_2, \dots, p_n ，令 $d(i, j)$ 表示 p_i 和 p_j 之间的欧几里得距离， $B[i, j]$ 表示以 p_i 为左向路径的起点，以 p_j 为右向路径的终点的双调最短旅程的长度（即以 p_i 为起点不断向左，到达 p_1 ，然后不断向右，到达 p_j ，经过 p_1 至 p_j 的所有点一次且仅一次的最短旅程长度）， $1 \leq i < j \leq n$ 。现在来考虑子问题，根据双调旅程的定义，旅程 $B[i, j]$ 访问了 p_1 到 p_j 之间的所有点，则点 p_{j-1} 必定位于双调旅程上，此时有两种可能： p_{j-1} 要么位于右向子路径，要么位于左向子路径。如果 p_{j-1} 位于右向子路径，由于 p_j 是当前的最右侧点，则 p_{j-1} 必定位于 p_j 之前，此时有 $i < j-1$ （ p_i 是左向路径的起点，而 p_{j-1} 和 p_j 均属于右向路径， p_{j-1} 和 p_j 是序号相邻的点，中间不存在其他点，由于已经限定 $i < j$ ，则 p_i 只可能是除 p_{j-1} 和 p_j 以外的其他点，故有 $i < j-1$ ），假设已经确定 $B[i, j-1]$ ，则有递推关系

$$B[i, j] = B[i, j-1] + d(j-1, j), \quad 1 \leq i < j-1 \quad (11.1)$$

如果 p_{j-1} 位于左向子路径，则 p_{j-1} 必定是左向子路径的最右侧点，此时有 $i = j-1$ （请读者自行思考为何此论断成立）。对于右向子路径来说， p_j 是其最右侧点，则 p_i 必定有一个前驱点，令其为 p_k ，其中 $k < j-1$ ，假设已知 $B[j-1, k]$ ，则有递推关系

$$B[i, j] = \min\{B[j-1, k] + d(k, j), \quad 1 \leq k < j-1\}, \quad i = j-1 \quad (11.2)$$

但 $B[j-1, k]$ 尚未计算，因此是未知的，不过根据双调旅程的对称性，有

$$B[j-1, k] = B[k, j-1]$$

而 $B[k, j-1]$ 可以根据递推关系 (11.1) 预先计算，则递推关系 (11.2) 等价于

$$B[i, j] = \min\{B[k, j-1] + d(k, j), \quad 1 \leq k < j-1\}, \quad i = j-1 \quad (11.3)$$

边界情形为只有两个点的情况，根据问题约束，此时 p_1 必定位于 p_2 之前，有

$$B[1, 2] = d(1, 2)$$

最后, 由于在求解过程中, 递推关系要求 $i \leq j-1$, 因此不能直接求得 $B[n, n]$, 而需要通过递推关系 (11.1) 间接计算, 即

$$B[n, n] = B[n-1, n] + d(n-1, n)$$

根据上述讨论, 可归纳得到求解双调欧几里得旅程的伪代码如下, 其中 $r[i, j]$ 用于记录前驱点信息, 以便构建具体的旅程。

```
// 双调欧几里得旅行商问题的动态规划算法。
BTSP()
    // 边界情形。
    B[1, 2] = d(1, 2)
    // 从左至右逐个考虑每个点作为子路径的端点。
    for j = 3 to n
        // 假定点  $p_{j-1}$  位于右向子路径, 根据递推关系 (11.1) 计算  $B[i, j]$ 。
        for i = 1 to j - 2
            B[i, j] = B[i, j - 1] + d(j - 1, j)
            r[i, j] = j - 1
        // 假定点  $p_{j-1}$  位于左向子路径, 根据递推关系 (11.3) 计算  $B[i, j]$  (亦即  $B[j-1, j]$ )。
        B[j - 1, j] = INF
        for k = 1 to j - 2
            if B[k, j - 1] + d(k, j) < B[j - 1, j] then
                B[j - 1, j] = B[k, j - 1] + d(k, j)
                r[j - 1, j] = k
            end if
        // 间接计算  $B[n, n]$ 。
        B[n, n] = B[n - 1, n] + d(n - 1, n)
```

根据动态规划算法过程中记录的前驱点信息 $r[i, j]$, 可以构建得到具体的旅程。

```
// 输出双调旅程的具体路径。
printPath(i, j)
    if i < j then
        k = r[i, j]
        print  $p_k$ 
        if k > 1 then
            printPath(i, k)
        end if
    else
        // 在进行动态规划时仅记录了  $r[i, j]$  而未记录  $r[j, i]$  且需要满足  $i < j$  的约束。
        k = r[j, i]
        if (k > 1) then
            printPath(k, j)
        end if
        print  $p_k$ 
    end if

// 输出双调旅程。
printTour(n)
    print  $p_n$ 
```

```

print p_{n-1}
k = r[n - 1, n]
printPath(k, n - 1)
print p_k

```

强化练习: [1347 Tour^C](#)。

扩展练习: [1096* The Islands^D](#)。

11.7 概率型动态规划

概率型动态规划是指将动态规划与概率论有机结合的题目。欲要顺利解决此种类型的题目，首先需要对概率论的相关概念和公式达到非常熟悉的程度，例如条件概率、期望、贝叶斯公式、全概率公式、全期望公式等。“非常熟悉”是指对概念和公式达到意义上的真正领会而不只是字面意义上的理解，并且能够顺利完成纯概率论的相关课后练习，只有具有上述基础之后，再结合动态规划的思想进行解题才会较为顺利，否则在大多数情况下将会出现毫无解题思路的困境。

常见的动态规划类型题目很容易稍加改变就能转换成概率型动态规划题目。在常规的动态规划题目中，一旦确定某个状态的转移，则转移是以概率 1 发生，如果将转移的概率定为一个变量 p ，即从某个状态转移到另一个状态时不再是确定的而是具有一定概率，这样所求结果就变成了达到预期目标的概率，而不再是达到预期目标的某个确定值。

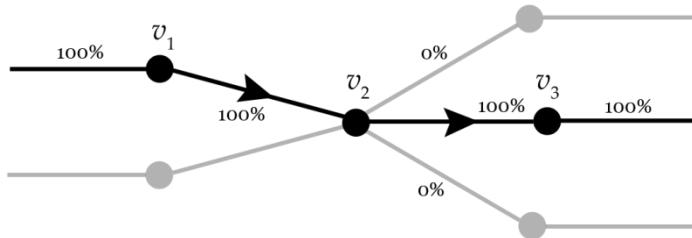


图 11-12 在典型动态规划中，从状态 v_1 转移到状态 v_2 以及从状态 v_2 转移到状态 v_3 的过程均为“确定性”事件，即只能选择后续状态中的某一种，一旦选定，则该后续状态发生的概率就认为是 100%

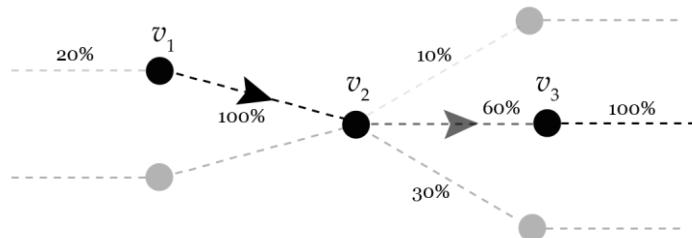


图 11-13 在概率型动态规划中，从状态 v_1 转移到状态 v_2 以及从状态 v_2 转移到状态 v_3 的过程均为“不确定性”事件，每次的状态转移都有一定的概率，可以以一定概率转移到多个后续状态，类似于量子物理中的“量子态”——量子的状态是多个可能状态的叠加且每种状态均有特定的概率

对概率型动态规划的外延作进一步扩展，可以将其视为一个“概率一期望系统”^[143]。概率一期望系统是一个带权有向图，图中的顶点表示某个事件，如果顶点 u 和顶点 v 之间具有一条权值为 p 的有向边，表

示顶点 u 所代表的事件发生后, 后续顶点 v 所代表的事件发生的概率为 p 。在概率一期望系统中, 如果对应的图是有向无圈图, 则可以通过递推或者备忘的方式进行解决, 或者将每个状态发生的概率视为一个未知数, 根据有向图的关系转换成一个多元一次方程组, 进而使用高斯消元法进行求解。如果有向图包含圈, 则可以通过限制递归的深度予以解决, 即在备忘过程中, 当深度达到某个限定值时即认为概率为零, 后续不再予以计算。

11176 Winning Streak^D (连胜)

Mikael 喜欢博彩, 正如你所知, 当今你可以在任何事情上下注。最近, 一件特别的事引起了 Mikael 的兴趣, 那就是在个赛季中某个队伍的最长连胜 (即连续赢得比赛的最大场数)。为了更明智地下注, Mikael 请求你编写一个程序, 用来计算他最喜欢的队伍能够取得的最长连胜的期望值。

一般来说, 一支队伍赢得比赛的概率依赖于许多不同的因素。例如, 是否为主场作战, 是否有主力队员受伤, 等等。然而, 对于程序的第一个原型来说, 我们简化了相关因素。假定所有的比赛具有相同的获胜概率 p , 而且一场比赛的结果不会影响后续比赛获胜的概率。

最长连胜局数的期望值是指: 在一个赛季中, 以全部比赛的每种可能结果发生的概率作为权值, 所有可能的比赛结果中最长连胜局数的平均值。举个例子, 假设某个赛季由 3 场比赛组成, 每场比赛获胜的概率 $p = 0.4$, 总共有 8 种不同的比赛结果, 我们可以使用一个由字母 ‘W’ 和 ‘L’ 构成的字符串来予以表示, 其中字母 ‘W’ 表示赢得比赛, 字母 ‘L’ 表示输掉比赛 (例如, “WLW” 表示队伍赢得了第一场和第三场比赛, 但是输掉了第二场比赛), 则此赛季的可能比赛结果为:

结果	LLL	LLW	LWL	LWW	WLL	WLW	WWL	WWW
可能性	0.216	0.144	0.144	0.096	0.144	0.096	0.096	0.064
连胜	0	1	1	2	1	1	2	3

在这个例子中, 最长连胜局数的期望值为

$$0.216 \cdot 0 + 0.144 \cdot 1 + 0.144 \cdot 1 + 0.096 \cdot 2 + 0.144 \cdot 1 + 0.096 \cdot 1 + 0.096 \cdot 2 + 0.064 \cdot 3 = 1.104$$

输入

输入包含多组测试数据 (最多 40 组), 每组包含一个整数 $1 \leq n \leq 500$, 表示在一个赛季中比赛的场数, 一个浮点数 $0 \leq p \leq 1$, 表示获胜的概率。输入以 $n=0$ 的一组测试数据作为结束标记, 不需处理此组数据。

输出

对于每组测试数据, 输出最大连胜局数的期望值。输出以浮点数表示, 绝对误差小于 10^{-4} 。

样例输入

```
3 0.4
10 0.75
0 0.5
```

样例输出

```
1.104000
5.068090
```

分析

此题难点在于动态规划状态的设计和递推关系的推导。按照一般的思路, 需要记录三个参数: 当前已经进行的比赛场数 i , 最长的连赢场数 j , 从最后一场比赛开始往前连赢的场数 k , 即状态 $dp[i][j][k]$ 表示“前 i 场比赛中不超过 j 场比赛连赢且最后 k 场比赛连赢的概率”, 其中 $0 \leq k \leq j < i \leq n$ 。初始时除 $dp[0][0][0]=1$ 外, 其余 $dp[i][j][k]=0$ 。假定已经得到 $dp[i-1][j][k]$, 若第 i 场比赛胜利, 则连赢场数变为 $k+1$ 场, 按照题

意，单场比赛获胜为独立事件，其概率为 p ，此时有

$$\begin{aligned} dp[i][j][k+1] &+= (dp[i-1][j][k] \times p), \quad k+1 \leq j \\ dp[i][j+1][k+1] &+= (dp[i-1][j][k] \times p), \quad k+1 > j \end{aligned}$$

若第 i 场比赛失利，则连赢场数变为 0 场，有

$$dp[i][j][0] = \sum_{k=0}^j (dp[i-1][j][k] \times (1-p))$$

根据上述递推关系式进行计算会导致时间复杂度为 $O(n^3)$ 的算法，由于 n 最大可为 500，显然会超时，需要考虑优化状态的表示，也就是检查是否可将三个参数表示的状态简化为两个参数表示的状态，从而使得状态的复杂度降低。考虑到第三个参数可以暗含在第二个参数当中，即“从最后一场比赛开始往前连赢的场数 k ”不会超过“最长的连赢场数 j ”，那么是否可以直接将其省略呢？答案是肯定的。也就是说，将状态 $dp[i][j]$ 定义为“前 i 场比赛中不超过 j 场比赛连赢的概率”，那么 n 场比赛后，连赢 j 场的概率为 $dp[n][j] - dp[n][j-1]$ 。根据上述状态定义，正向推导计算 $dp[i][j]$ 存在困难，而从反向角度计算 $dp[i][j]$ 则相对容易，其关键是应用补集的概念：所有可能事件发生的总概率为 1，从中减去不符合要求的事件的概率即为所求事件的概率。假设当前进行了 $i-1$ 场比赛，现在进行第 i 场比赛，如果第 i 场比赛结果为输，则此种情形不会影响 $dp[i][j]$ 的结果；如果第 i 场比赛结果为赢，若前 $i-1$ 场比赛最后为连赢 j 场比赛，此时若再赢一场，则 i 场比赛中会出现连赢 $j+1$ 场比赛的情形，这种情形不符合状态的定义，将这种情形的概率减去即为 $dp[i][j]$ 的正确结果，那么这种情形发生的概率是多少呢？由于前 $i-1$ 场比赛末尾是连赢 j 场，则连赢 j 场之前必须是结果为输的 1 场比赛（或者是连赢 j 场比赛之前不存在其他比赛），否则将会出现连赢 $j+1$ 场的情形，因此其概率为 $dp[i-(j+1)-1][j] \times (1-p) \times p^{j+1}$ （若连赢 j 场比赛前不存在其他比赛则概率为 p^{j+1} ），即除去输掉的 1 场比赛和后续连赢的 $j+1$ 场比赛后，前面的 $i-(j+1)-1$ 场比赛中连赢不超过 j 场比赛的概率 $dp[i-(j+1)-1][j]$ 乘以输掉 1 场比赛的概率 $(1-p)$ 再乘以连赢 $j+1$ 场比赛的概率 p^{j+1} 即为不符合要求的事件发生概率（若连赢 j 场比赛前不存在其他比赛则为连赢 $j+1$ 场比赛的概率 p^{j+1} ），即递推关系式为

$$dp[i][j] = \begin{cases} dp[i-1][j] - p^{j+1} & j+1 = i \\ dp[i-1][j] - dp[i-(j+1)-1][j] \times (1-p) \times p^{j+1} & j+1 < i \end{cases}$$

其中 $0 \leq j < i < n$ 。易知 $dp[0][j] = 1$ ， $0 \leq j \leq n$ 。

按照题目给定的计算方式，在得到相应的概率之后，最大连胜局数的期望值为

$$E = \sum_{j=1}^n ((dp[n][j] - dp[n][j-1]) \times j)$$

强化练习：1456* Cellular Network^D，10091 The Valentine's Day^D，[10648* Chocolate Box^D](#)，11755 Table Tennis^E。

扩展练习：888* Donkey^E，10207* The Unreal Tournament^D，11468* Substring^D。

在概率型动态规划中，最为常见的是与期望有关的题目，之所以期望类型的动态规划题目最为常见，一个重要的原因是和期望的定义有关。期望的定义是所有可能取值的加权平均，其权值是取相应值的概率，只需在普通类型的动态规划题目中为每种状态转移规则设定一个概率，则状态相应的值即对应期望。因此，熟练掌握前述介绍的各种动态规划类型对顺利解决与期望相关的概率型动态规划题目至关重要。解决期望相关的动态规划题目，其关键仍然是推导递推关系。在获得递推关系的过程中，一个常用的技巧是将欲求的期望设为一个未知数，检查是否可以通过取条件来计算期望，从而得到初始期望的一个一元方程，通过解这个一

元方程得到目标期望值。在解题过程中经常联合使用的是备忘技巧。

11427 Expect the Expected^D (意料之中)

我喜欢玩纸牌接龙，每天我都会玩一局。我每局有概率 p 能够赢，有概率 $1-p$ 会输。游戏会保存我已玩游戏的统计资料——我赢得游戏的局数百分比。如果我玩游戏的局数足够多，那么我的胜局百分比将始终在 $p \times 100\%$ 附近徘徊。但我想赢得更多。以下是我的计划：每天我都会玩一局游戏，如果我赢了，我会立马高高兴兴地去睡觉直到天亮；如果我输了，我会一直玩，直到我赢的局数占当天所玩局数的百分比大于 p 时才停止，此时，我会宣布我已胜利然后去睡觉。正如你所见，在每天结束的时候，我保证总是能够将自己的获胜百分比保持在期望值 $p \times 100\%$ 之上，我将打败概率论！

假如你的直觉告诉你上述计划可能有什么地方出错了，那么你是对的。我并不能一直照着计划执行来维持我的获胜百分比，因为每天我所能够玩的游戏局数是有限的。假设一天我最多能玩 n 局游戏，在我的计划失败前，我能期望它持续多少天呢？注意，期望的天数至少为 1，因为我至少需要一天的时间来完成一局游戏，不管是赢还是输。

输入

输入的第一行给出了测试数据的组数 N ，每组测试数据一行，每行包含两个数——分数 p 和整数 n 。 $1 \leq N \leq 1000$ ， $0 \leq p < 1$ ， $1 \leq n \leq 100$ ，分数 p 的分母最大为 1000。

输出

对于每组测试数据，以 ‘Case # x : y ’ 的形式输出，其中 y 为期望的天数，输出时向下取整。

样例输入

```
4
1/2 1
1/2 2
0/1 10
1/2 3
```

样例输出

```
Case #1: 2
Case #2: 2
Case #3: 1
Case #4: 2
```

分析

由于每天玩游戏相互独立，定义事件 A 为“第一天计划成功”，令 X 为计划持续成功的天数， Y 为事件 A 的示性函数，根据全期望公式，通过取条件计算期望，有

$$E[X] = E[E[X|Y]] = E[X|Y = A]P\{Y = A\} + E[X|Y = A^c]P\{Y = A^c\}$$

若事件 A 发生，则第二天问题的状态与第一天问题的状态相同，因此有

$$E[X|Y = A] = 1 + E[X]$$

若事件 A 不发生，持续天数为 1，有

$$E[X|Y = A^c] = 1$$

故有

$$E[X] = (1 + E[X]) \times P(A) + 1 \times (1 - P(A))$$

整理可得

$$E[X] = \frac{1}{1 - P(A)} = \frac{1}{P(A^c)}$$

那么 $P(A)$ 如何求呢？由于当赢的局数占当天所玩局数的百分比大于 $p \times 100\%$ 即停止，定义事件 A_i 为“已经玩 i 局游戏时获胜局数占已玩局数的百分比大于 $p \times 100\%$ ”，则有

$$P(A) = \sum_{i=1}^n P(A_i)$$

不难看出, 利用上式正向计算 $P(A)$ 较为繁琐, 因为构成“第一天计划成功”的状态较多, 不便于确定, 不妨考虑计算 $P(A^c)$, 即“第一天计划失败”的概率。那么“第一天计划失败”这个事件包括哪些状态呢? 由于每局游戏相互独立, 可以将 n 局游戏视为 n 次独立试验, 假设我未达到“获胜局数占已玩局数百分比大于 $p \times 100\%$ ”的目标, 那么我会一直玩到第 n 局游戏结束, 如果当前所赢局数 x 与 n 的比值仍然小于等于 p , 则此时的状态意味着计划失败。令 $dp[i][j]$ 表示“在第 i 局游戏时已经赢得 j 局游戏的概率”, 则

$$P(A^c) = dp[0][n] + dp[1][n] + \dots + dp[x][n], \quad 1 \leq x \leq n, \quad \frac{x}{n} \leq p$$

而

$$dp[i][j] = dp[i-1][j] * (1-p) + dp[i-1][j-1] * p, \quad 1 \leq i \leq n, \quad 0 \leq \frac{j}{i} \leq p$$

边界条件为

$$dp[0][0] = 1, \quad dp[i][j] = 0, \quad i \neq 0 \text{ 且 } j \neq 0$$

注意

$$dp[i][0] = dp[i-1][0] * (1-p), \quad i \geq 1$$

强化练习: 12457 Tennis Contest^D。

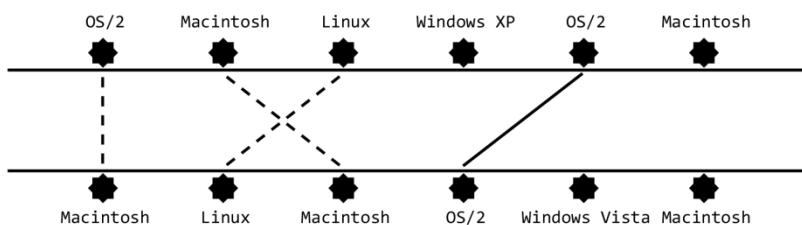
扩展练习: 10529* Dumb Bones^D, 11762* Race to 1^D, 12585 Poker End Games^D, 12723* Dudu the Possum^E, 13030* Brain Fry^E。

11.8 非典型动态规划

此种类型动态规划不同于典型的动态规划, 其难点主要在于以下几个方面: (1) 状态需要仔细斟酌并加以定义才能用于解题从而得出正确的答案; (2) 递推关系不易得出; (3) 状态转移规则复杂, 编写代码容易出错; (4) 在定义状态时需要使用某些技巧, 例如状态偏移。若要解决此类动态规划题目, 需要在经历一定量的练习之后, 在积累了较多解题经验的基础上, 加上一定的想象力才能顺利完成。一般来说, 往往在推导得出递推关系后解题就变得非常简单。

1172 The Bridges of Kølsberg (Kølsberg 之桥)

国王 Beer 有一块非常棘手的辖区需要管理, 该辖区由沿着 Kølsberg 河南北两岸分布的众多城市组成。这些城市拥有不同的操作系统信仰, 并且具有发达的贸易水平。因为河流很宽而且危险, 使得这些城市在经济上相互隔离。



实线所示为符合要求的桥梁建造方式, 虚线为不符合要求的桥梁建造方式

国王 Beer 想建造一些桥梁连通河两岸的城市。有人向他强烈建议, 避免在具有不同操作系统信仰的城

市间搭建桥梁（这些人真心互相厌恶对方），因此他将只在具有相同操作系统信仰的城市之间建造桥梁（尽管这可能会导致完工的桥梁长度过长且外观怪异）。不过，由于技术原因，无法建造与其他桥梁发生交叉的桥梁。某座桥的经济价值使用它所连接的两个城市的贸易总额来衡量。国王希望在建造尽量少的桥的条件下，使得所建桥的经济价值尽可能地大。给定两组城市的描述，确定所建桥梁的最大可能价值以及最少需要建造的桥梁数量^I。

输入

输入的第一行为一个整数，表示测试数据的组数。对于每组测试数据，第一行包含一个非负整数 n ，不大于 1000，表示河北岸城市的数量，接着的 n 行，每行以下列形式给出北岸城市的信息：

```
cityname ostype tradevalue
```

这些数据以空格分隔。字符串 *cityname* 和 *ostype*，其长度不超过 10 个字符，*tradevalue* 是一个不大于 10^6 的非负整数。这 n 行数据从前往后依次对应沿着河北岸从左到右的城市信息。接着以同样的格式给出位于南岸的城市信息。

输出

对于每组测试数据，输出一行，该行包含两个整数，第一个整数表示能够建造的所有桥梁的最大经济价值，然后是一个空格，接着是另外一个整数，表示所需要建造的最少桥梁数量。

样例输入

```
1
3
mordor Vista 1000000
xanadu Mac 1000
shangrila OS2 400
4
atlantis Mac 5000
hell Vista 1200
rivendell OS2 100
appleTree Mac 50
```

样例输出

```
1002250 2
```

分析

对于此类需要在特定条件下求解最大值（和/或最小值）的题目，一般使用回溯法或动态规划解决。显然，对于题目所给定的数据规模，使用朴素的穷尽搜索无法在规定时间内获得通过，而需使用动态规划算法予以解决。先定义所需要的状态，假设北岸共有 n_1 个城市，南岸共有 n_2 个城市，令 $V[i][j]$ 表示在北岸的前 i 个城市和南岸的前 j 个城市间建立桥梁后所能得到的最大经济价值， $B[i][j]$ 表示对应的最少需要架设的桥梁数量， nt_i 表示北岸第 i 个城市的操作系统类型， st_i 表示南岸第 i 个城市的操作系统类型， nv_i 表示北岸第 i 个城市的贸易额， sv_i 表示南岸第 i 个城市的贸易额，则 $V[n_1][n_2]$ 和 $B[n_1][n_2]$ 即为所求。那么 $V[i][j]$ 如何通过递推得到呢？从直觉上不易得到递推关系式，在这种情况下，一般的做法是通过一组数据量较少但典型的测试数据来进行归纳总结，以期得出递推关系式。

给定如下的测试数据：

^I 每个城市至多建造一座桥梁与对岸城市相连，位于河岸同侧的城市之间不需架设桥梁——作者注。

```

1
3
CITY1 OS1 10000
CITY2 OS2 1000
CITY3 OS3 100
3
CITY2 OS2 1000
CITY3 OS3 100
CITY1 OS1 10000

```

如图 11-14 所示, 将上述测试数据以图形的方式进行表示。为了便于说明, 将河岸的方向更改为竖直方向。在此种表示方式下, 左侧为河的北岸, 右侧为河的南岸, 同时在河的两岸添加了一对虚拟的城市, 其操作系统类型均为 OS0, 而其经济价值为 0。

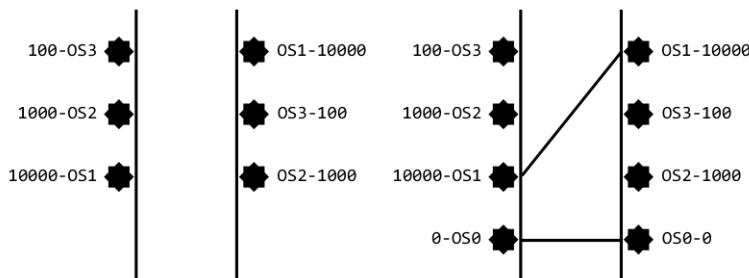


图 11-14 测试数据所对应的示意图

根据分别位于两岸的一对城市之间其操作类型是否相同来构建递推关系。如果北岸的城市 i 和南岸的城市 j 两者的操作系统类型不同, 即 $nt_i \neq st_j$, 则不能在两者之间假设桥梁, 于是 $V[i][j]$ 的值应为 $V[i-1][j]$ 、 $V[i][j-1]$ 、 $V[i-1][j-1]$ 三者中的最大值, 即

$$V[i][j] = \max\{V[i-1][j], V[i][j-1], V[i-1][j-1]\}, \quad nt_i \neq st_j, \quad 1 \leq i \leq n_1, \quad 1 \leq j \leq n_2$$

如果北岸的城市 i 和南岸的城市 j 两者操作系统类型相同, 即 $nt_i = st_j$, 则可在两者之间假设一座桥梁, 此时需要检查新建桥梁后是否可能达到更大的经济总价值, 亦即取 $V[i][j]$ 和 $V[i-1][j-1] + nv[i] + sv[j]$ 的较大值, 即

$$V[i][j] = \max\{V[i][j], V[i-1][j-1] + nv[i] + sv[j]\}, \quad nt_i = st_j, \quad 1 \leq i \leq n_1, \quad 1 \leq j \leq n_2$$

如果 $V[i][j]$ 和 $V[i-1][j-1] + nv[i] + sv[j]$ 相等, 则比较 $B[i][j]$ 和 $B[i-1][j-1] + 1$ 的大小, 取两者的较小值, 即

$$B[i][j] = \min\{B[i][j], B[i-1][j-1] + 1\}, \quad \text{当 } V[i][j] = V[i-1][j-1] + nv[i] + sv[j]$$

边界条件为 $V[i][0] = V[0][j] = B[i][0] = B[0][j] = 0, \quad 0 \leq i \leq n_1, \quad 0 \leq j \leq n_2$ 。

强化练习: 10081 Tight Words^B, 10086 Test the Rods^C, 10128 Queue^B, 10271 Chopsticks^B, 10337 Flight Planner^B, 10918* Tri Tiling^B, 11258 String Partition^B, 11420 Chest of Drawers^B, 11555 Aspen Avenue^D, 11472 Beautiful Numbers^C, 13141 Growing Trees^D。

扩展练习: 909 The BitPack Data Compression Problem^E, 10722* Super Lucky Numbers^D, 11026 A Grouping Problem^C, 11285* Exchange Rates^D, 11552* Fewest Flops^C, 11908 Skyscraper^D, 12654 Patches^D, 12951* Stock Market^D。

状态偏移

存在某些动态规划题目，其给定的状态值是负数，而 C++ 中数组的下标只能是非负整数。为了正确地表示状态，需要对其进行适当处理——使用状态偏移技巧将负值状态调整为非负值状态。例如，在子集和问题中，若给定的整数有正数也有负数，则在递推求解的过程中，和可能为负数，如果此时仍然使用数组来表示状态，则需要将和调整为正数。如果不使用数组来表示状态，则可以使用标准类库中的 `set` 容器类来保存状态，不过这样做的效率可能会比数组表示要稍低一些。

强化练习：323 Jury Compromise^D，11002 Towards Zero^D，11832 Account Book^D。

扩展练习：1238* Free Parentheses^D。

11.9 动态规划的优化

在解决动态规划问题的过程中，可以使用某些优化技巧来便于解题或者提高程序的空间/时间效率。

11.9.1 空间优化

空间优化是指在动态规划过程中，根据递推关系式的特点来缩减保存状态的数组的维数，使用更少的内存空间来完成计算，提高内存空间的使用效率。

空间优化的一种常用技巧是使用滚动数组对状态进行更新。回顾前述的 01 背包问题，在解决该问题的过程中，使用二维数组保存当前的状态，空间复杂度为 $O(VN)$ 。可以进一步优化空间的使用，使用一维数组来保存状态，从而将空间复杂度降为 $O(V)$ 。观察 01 背包问题的递推关系， $V[i][j]$ 取决于 $V[i-1][j]$ 或 $V[i-1][j-C_i]+P_i$ ，第 $i-1$ 行元素在使用后不会再次使用，因此可以只用一维数组来存储中间结果。采用空间优化的方式求解时需要逆向进行，即容量的遍历顺序需要从最大容量到 $volume[i]$ ，这样才能保证取用到正确的值，否则取用的值是被之前计算所覆盖的错误值。

```
int v[capacity + 1] = {0};
for (int i = 1; i <= n; i++)
    for (int j = capacity; j >= volume[i]; j--)
        v[j] = max(v[j], v[j - volume[i]] + price[i]);
```

或者保持遍历顺序从小到大不变，使用模运算技巧，按上述方式优化空间的使用，俗称“滚动数组”。

```
int v[2][capacity + 1] = {0};
for (int i = 1; i <= n; i++)
    for (int j = volume[i]; j <= capacity; j++)
        v[i % 2][j] = max(v[(i - 1) % 2][j], v[(i - 1) % 2][j - volume[i]] + price[i]);
```

类似的，在 Floyd-Warshall 算法中，根据状态转移的特点，通过使用滚动数组技巧可以将初始实现的使用三维数组来表示的状态缩减为二维数组表示的状态。

强化练习：10154 Weights and Measures^B。

11.9.2 状态优化

在某些动态规划题目中，有时并不需要考虑所有给定的状态，而只需要考虑其中的一部分状态即可，亦即部分状态可以予以“丢弃”，不影响最终结果的正确性。还有一种情况是状态的某些参数已经“隐含”于其他参数中，如果将参数全部予以表示，将超出内存的限制，此时可以选择省略掉可以推导得到的参数，从而优化状态表示，同时通过恰当的预处理，进行常数项优化，从而使得解题方案能够在规定时间内获得通过。

702 The Vindictive Coach^C（复仇心重的教练）

在常年经受媒体对其“排兵布阵”战术的恶评后，某个足球队的教练决定向媒体实施“复仇”。他将所有队员按高矮相间的顺序排成一列，迫使媒体的摄像机在拍摄球员的画面时不得不像锯齿一样地上下移动。然而，由于某种特定的原因，球队队长主张教练应该将他排在队列的首位。由于队长想成为媒体的焦点所在，因此队长要求与他相邻的球员的身高必须要比他矮（除非其他队员的身高都比他要高）。如果其他队员的身高都比队长要高，在仍然保证队伍呈锯齿形的前提下，要求与队长相邻的队员与队长的身高之差尽可能地小。

已知所有队员的身高均不相同。教练聘请了一位计算专家（例如，你），根据上述限制条件，确定满足要求的不同排列的总数。众所周知，队员们在战术演示板上以塑料雕像代替，最矮的人编号为 1。当然，队员的数目是任意的，但保证不超过 22 个。

输入

输入包含多组测试数据。每组测试数据一行，包含两个以空格分隔的正整数 N 和 m 。 N (≤ 22) 表示包括队长在内的队员总数， m 表示队长的编号，要求队长总是排在队伍的首位。

输出

输出满足题目条件限制的队伍不同排列方式的总数。

样例输入

```
3 1
3 3
4 1
```

样例输出

```
1
1
1
```

分析

本题实质上是一个有向图上的路径计数问题。初看似乎可以使用集合型动态规划解决。令 $dp[i][j][k]$ 表示已加入队列的队员的位掩码为 i ，队列最末尾的人编号为 j ，当前排列模式为 k 时的不同排列方案总数。 k 可以取 0 和 1 两个值，当 $k=0$ 时，表示下一个作为队列末尾的人要比编号为 j 的队员的身高要高；当 $k=1$ 时，表示下一个作为队列末尾的人要比编号为 j 的队员的身高要矮。结合备忘技巧有以下实现：

```
const int HIGHER = 0, LOWER = 1;

int N, m, ONES;
long long dp[1 << 22][22][2];

long long dfs(int mask, int last, int mode) {
    if (~dp[mask][last][mode]) return dp[mask][last][mode];
    if (mask == ONES) return 1LL;
    long long r = 0;
    if (mode == HIGHER) {
        for (int bit = last + 1; bit < N; bit++) {
            if (mask & (1 << bit)) continue;
            r += dfs(mask | (1 << bit), bit, LOWER);
        }
    } else {
        for (int bit = 0; bit < last; bit++) {
            if (mask & (1 << bit)) continue;
            r += dfs(mask | (1 << bit), bit, HIGHER);
        }
    }
    return dp[mask][last][mode] = r;
}
```

当 N 较小时, 例如 $N \leq 16$, 可以很快得到结果, 但是当 $N=22$ 时, 由于状态数量较大, 出现超时。

进一步考察使用集合型动态规划的实现, 可以发现它对重复状态的利用率较低, 每次得到的基本上都是一个新的状态。假设当前已经排好了 x 名队员, 最高的队员序号为 x_{high} , 最矮的队员序号为 x_{low} , 队尾的队员序号为 x_{last} , 令其为队伍 A , 还需要将剩余的 y 名队员接在这 x 名队员之后形成一个锯齿形的队列, 令其为队伍 B 。若要求 B 的第一名队员比 A 的最后一名队员要矮, 则由于 B 中的 y 个人身高均不相同, 问题转化为从 y 名队员中挑出一个比序号为 x_{last} 的队员要矮的人放在 B 的队首且所能得到的锯齿形队伍的数量(若要求 B 的第一名队员比 A 的最后一名队员要高, 则由于 B 中的 y 个人身高均不相同, 问题转化为从 y 名队员中挑出一个比序号为 x_{last} 的队员要高的人放在 B 的队首且所能得到的锯齿形队伍的数量)。不难看出, 问题是相似的。但是使用前述的状态定义, 需要知道比队尾序号为 x_{last} 的队员要矮的人中有多少人已经加入了队列, 还有哪些人没有加入队列, 在当前状态定义的基础上不便于回答这个问题。

注意到给定的 N 个人其身高均是不相同的, 且题目要求排列得到的队伍外形呈现锯齿形, 那么可以考虑优化状态的表示。即令 $f[i][j]$ 表示“在满足队伍的外形呈锯齿形的情况下, 将 i 个人中的第 j 个人排列在队伍的最前面, 且队伍前面两名队员的身高递增的排列总数”, $g[i][j]$ 表示“在满足队伍的外形呈锯齿形的情况下, 将 i 个人中的第 j 个人排列在队伍的最前面, 且队伍前面两名队员的身高递减的排列总数”, 则可以得到以下递推关系式

$$f[i][j] = \sum_{k=j}^{i-1} g[i-1][k], \quad g[i][j] = \sum_{k=1}^{j-1} f[i-1][k]$$

根据上述递推关系式, 如果令 $dp[i][j][0]$ 表示 $f[i][j]$, $dp[i][j][1]$ 表示 $g[i][j]$, 结合备忘技巧, 可以有以下时间复杂度为 $O(N^3)$ 的解题方案, 能够有效应对 N 较大时的情形。

参考代码

```
const int HIGHER = 0, LOWER = 1;

int N, m;
long long dp[32][32][2];

long long dfs(int i, int j, int mode) {
    if (i == 1 && j == 1) return 1;
    if (~dp[i][j][mode]) return dp[i][j][mode];
    long long r = 0;
    if (mode == HIGHER) {
        for (int k = j; k < i; k++)
            r += dfs(i - 1, k, LOWER);
    } else {
        for (int k = 1; k < j; k++)
            r += dfs(i - 1, k, HIGHER);
    }
    return dp[i][j][mode] = r;
}

int main(int argc, char *argv[]) {
    memset(dp, -1, sizeof(dp));
    while (cin >> N >> m) {
        if (N <= 2) { cout << "1\n"; continue; }
        long long r = 0;
        if (m == 1) r = dfs(N - 1, 2, LOWER);
        else {

```

```

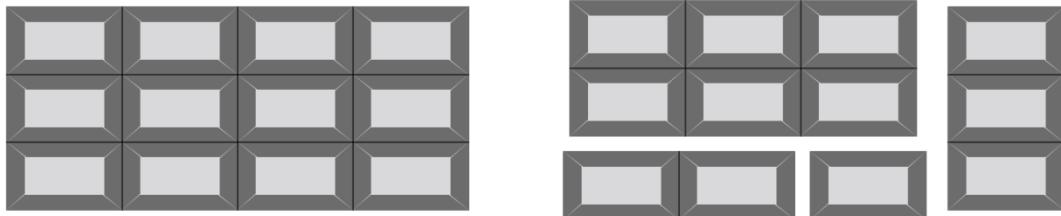
        for (int i = 1; i < m; i++)
            r += dfs(N - 1, i, HIGHER);
    }
    cout << r << '\n';
}
return 0;
}

```

1099 Sharing Chocolate^C (分享巧克力)

给定一块尺寸为 $x \times y$ 的矩形巧克力，它由相同大小的 $x \times y$ 块单位矩形巧克力小块构成。现在你需要将这块矩形巧克力与 n 个朋友们分享，不过你的朋友们都很挑剔，并且有着不同的爱好：有人希望你给他的巧克力多一些，有人希望少一些。为了分享巧克力，你要把一整块大的分成两块小的，从行或者列之间断开，重复地分开巧克力，直到将巧克力分成每一个朋友所要求的大小，而且数量不多不少，恰好能够分享给所有的朋友。编写程序，确定朋友们的要求是否能够实现。

比如说，下图显示了把一个 3×4 的巧克力分成 4 个部分，每个部分分别包括 6 格、3 格、2 格、1 格，分了 3 次（对应于样例输入的第一组数据）。



输入

输入包含多组测试数据，每组测试数据表示一块要被分享的巧克力。每组测试数据首先给出一个整数 n ($1 \leq n \leq 15$)，表示要分成的块数；然后给出两个整数 x 和 y ($1 \leq x, y \leq 100$)，表示这块巧克力的大小；接下来的一行给出 n 个正整数，表示被分成的 n 块巧克力每块具有多少格。输入以包含一个 0 的一行表示结束。

输出

对于每组测试数据，首先输出测试数据的编号，然后输出能否按照要求分开巧克力。如果可能，输出“Yes”，否则输出“No”。

样例输出

```

4
3 4
6 3 2 1
2
2 3
1 5
0

```

样例输出

```

Case 1: Yes
Case 2: No

```

分析

令 $F[i][j][k]$ 表示尺寸为 $i \times j$ 的矩形巧克力能否切分为若干块供 k 所表示的若干朋友所分享。其中 i 和 j

表示巧克力块的长和宽, k 是一个二进制压缩状态, 从序号 0 开始为朋友编号, 每个二进制位与朋友的序号一一对应, 为 1 的二进制位表示该状态下可以切分出对应序号的朋友所要求大小的巧克力块, 那么问题的解即为确定 $F[x][y][2^n-1]$ 的值是否为真。但根据题目给定的约束条件, 可能的最大状态数为 $x \times y \times (2^n-1) = 100 \times 100 \times (2^{15}-1) \approx 328 \times 10^6$, 不仅会超时, 而且可能会超出内存限制。考虑到 n 个朋友所要求的巧克力块大小已经给出, 则 k 所表示的二进制状态实际上已经隐含了需要切出的面积 (对二进制数中为 1 的位所对应朋友的巧克力大小求和即可), 那么只需要知道长度, 就可以推算出宽度, 反之亦然, 因此第二维是多余的, 可以舍去, 则最终的状态数约为 3.3×10^6 , 进行适当地优化, 可以在限定时间和内存条件下解决该问题。

在具体实现时, 可以考虑使用备忘技巧, 并加入一个常数优化: 先把每个面积的状态预处理一次, 将所有可能产生同一面积的状态排列成一个链表 L , 这样可以去除不必要的状态枚举。令 $A[i]$ 表示第 i 个朋友所要求的巧克力块的大小 (从 0 开始计数), 则状态的初始值为:

$$F[0][0] = 1, F[j][2^i] = 1, 0 \leq i < n, 1 \leq j \leq \lfloor \sqrt{A[i]} \rfloor, A[i] \% j = 0$$

在已知巧克力面积为 C 的情况下, 可通过下述方法判断能否从一边长为 x 的巧克力中切出状态 k (另一边长为 $y = C/x$), 即计算 $F[x][k]$:

(1) 枚举 x 方向上的每个可能的切割位置 i , 其中 $1 \leq i \leq \lfloor x/2 \rfloor$;

(2) 枚举链表 $L[i \times y]$ 中的每个状态 j , 若 j 为 k 的子状态, 且沿 i 切分能够产生子状态 j (对应边长为 $\min(i, y)$, 面积为 $i \times y$) 和子状态 $k-j$ (其对应边长为 $\min(x-i, y)$, 面积为 $C - i \times y$), 则返回 $F[x][k] = 1$;

(3) 用同样的方法处理 y 方向的切割;

(4) 若枚举了 x 方向和 y 方向的每一种可能切割, 仍无法满足要求, 则 $F[x][k] = 0$ 。

在记录状态时, 由于可以从面积和长度推导出宽度, 也可以由面积和宽度推导出长度, 因此第一维可选择长度和宽度的较小值作为实际参数值。根据状态的定义, 显然 $F[\min(x, y)][2^n-1]$ 即为问题的解。

强化练习: 10118 Free Candies^C, 10482 The Candyman Can^C, 10626 Buying Coke^C, 10934* Dropping Water Balloons^C, 12324* Philip J. Fry Problem^D。

扩展练习: 1231* ACORN^D, 1244* Palindromic Paths^D。

11.9.3 二进制优化

对于背包问题, 有时题目所给的条件中同类物品的数量较大, 如果仍旧按照逐个物品枚举的方式进行递推容易造成超时。此时可以应用二进制优化技巧将同类物品进行“打包”, 以尽量减少所需遍历物品的个数, 从而提高递推效率。二进制优化的核心思想是将“同类多件物品”转换为“多类单件物品”。在二进制数系统中, 任意一个非负整数均可以使用二进制数进行表示。例如, $78_{10} = 1001110_2 = 0111111_2 + 0001111_2$, 即 78 可以拆分为 1, 2, 4, 8, 16, 32, 15。类似于兑换零钱的过程, 可以通过 1, 2, 4, 8, 16, 32, 15 这 7 个数的适当组合表示 1 到 78 之间的任意整数, 例如 $35 = 4 + 16 + 15$, $62 = 2 + 4 + 8 + 16 + 32$ 。在未进行优化时, 以单件物品进行递推, 需要从 1 遍历到 78, 进行二进制优化后, 仅使用 7 件物品进行递推, 只需从 1 遍历到 7, 时间复杂度降低为原来的约 $1/\log 78$, 当物品的数量 n 越大时, 效率提升越明显。二进制优化的正确性之所以能够保证, 原因在于给定的是同类的多件物品, 将其合并后并不会改变最终的结果。

强化练习: 711 Dividing Up^C。

11.9.4 单调队列优化

对于递推关系式形如

$$f(i) = \max \text{ 或 } \min\{g(j) | b(i) \leq j \leq i\} + h(i)$$

的动态规划问题，若问题约束满足以下条件：

- (1) $b(i)$ 满足单调性，即 $b(i)$ 随着 i 的递增不递减；
- (2) $g(j)$ 是一个与 $f(j)$ 和 j 有关的函数；
- (3) $h(i)$ 是一个只与 i 有关的函数；

则可以使用一种称为单调队列优化的技巧来提高状态转移的效率。根据递推关系式并利用 $b(i)$ 的单调性，能够推导得到问题在进行状态转移时具有如下的性质：如果存在两个决策点 j_1 和 j_2 ，使得 $j_1 < j_2$ 且 $g(j_2)$ 比 $g(j_1)$ 更优，则决策 j_1 是毫无用处的。利用这个性质，可以维护一个队列，使得队列中的元素满足决策的单调性。使用聚合分析，由于每个决策只会进出队列各一次，所以转移复杂度均摊是 $O(1)$ ，最后就把原来 $O(n^2)$ 的时间复杂度优化到了 $O(n)$ 。如果动态规划的递推关系式比较复杂，可以采用变量分离法，将递推关系式中依赖于 i 和依赖于 j 的项进行分离，如果分离成功，则可以考虑使用单调队列优化^[144]。

例如，若动态规划的递推关系式为

$$f[i] = \min\{f[j] | i - L \leq j < i\} + h[i]$$

其中 L 为已知常数， $h[i]$ 为一个只与 i 有关的函数。如果存在 $j_1 < j_2$ ，而且 $f[j_2] < f[j_1]$ ，那么对于 j_2 之后的状态来说， j_1 一定不会被作为最优值。因此，只要维护一个在 j 递增的同时 $f[j]$ 也递增的队列，在更新 $f[i]$ 前查询满足条件的 $f[j]$ 的最小值时，只需将队列前端不满足下标限制的决策点删除，直到遇到一个满足下标限制的决策点，将该决策点的 $f[j]$ 值用于更新 $f[i]$ 即可，而在更新 $f[i]$ 后将 $f[i]$ 插入单调队列时，只需将队列尾端不满足单调性的决策点删除即可^[145]。

1169 Robotruck^D（机器人运货车）

本问题和工厂里的机器人运货车有关，该设备用于将邮件包裹分发到工厂中的各个地点。机器人位于收发室传送带的末端，等待包裹被装载到它的载货区中。机器人具有最大装载量限制，这意味着它可能需要若干次往返才能将所有包裹递送完毕。在不超过机器人最大装载量的情况下，它可以在任意时刻停止传送带并开始已装载包裹的递送。包裹必须按照装载的先后顺序进行递送。

在工厂网格中，一个往返的距离按如下方式进行计算：收发室的位置为 $(0, 0)$ ，机器人从收发室出发，依次递送包裹，往返的距离等于下列三项步数之和——从收发室到第一个包裹需要递送的位置所移动的步数，每两个包裹之间所需要移动的步数，从最后一个包裹的递送位置回到收发室所需要移动的步数。在网格中，机器人可以沿水平或者垂直方向每次移动一步。例如，考虑四个包裹的情形，假设它们将被递送到位置 $(1, 2)$ ， $(1, 0)$ ， $(3, 1)$ 和 $(3, 2)$ 。将这些包裹分为两个往返进行递送，每次递送两个包裹，则第一个往返所移动的步数为 $3+2+1=6$ ，第二个往返所移动的步数为 $4+0+4=8$ 。注意，由于最后两个包裹的位置相同，因此在递送这两个包裹之间需要移动的步数为 0。

给定一个包裹的序列，计算机器人递送所有包裹必须移动的最小距离。

输入

输入包含多组测试数据。输入的第一行包含一个整数，表示测试数据的组数。在每组测试数据之前有一个空行。每组测试数据的第一行为一个正整数 C ，不大于 100，表示机器人的最大装载量，接下来的一行包含一个正整数 N ，不大于 100000，表示在传送带上需要装载的包裹总数量。接着的 N 行，每行包含一个包裹的描述：两个非负整数，表示在工厂网格中包裹的送达位置；一个正整数，表示包裹的重量。每个包裹的重量总是小于机器人的最大装载量。输入中包裹的顺序即为包裹在传送带上出现的顺序。

输出

对于每组测试数据输出一行，包含一个整数，表示机器人为了递送所有包裹所需要移动的最小步数。在

每两组测试数据的输出之间打印一个空行。

样例输入

```
1
10
4
1 2 3
1 0 3
3 1 4
3 1 4
```

样例输出

```
14
```

分析

令 $dp[i]$ 为递送前 i 个包裹所需要移动的最小步数, $sw[i]$ 表示第 i 个包裹的重量, $d_1[i]$ 表示第 $i-1$ 个包裹和第 i 个包裹之间的最短距离, $d_2[i]$ 表示第 i 个包裹距离起始地点的最短距离, 定义

$$sw[i] = \sum_{j=0}^i w[j], \quad w[0] = 0, \quad i \geq 1$$

$$sd_1[i] = \sum_{j=0}^i d_1[j], \quad d_1[0] = 0, \quad i \geq 1$$

根据题意, 有以下递推关系式

$$dp[i] = \min\{dp[j] + d_2[j+1] + (sd_1[i] - sd_1[j+1]) + d_2[i]\}, \quad sw[i] - sw[j] \leq C, \quad j \geq 0, \quad i \geq 1$$

边界条件: $dp[0] = 0$ 。由递推关系式可以得到以下的解题方案。

参考代码

```
const int MAXN = 100010, INF = 0x7f7f7f7f;

int main(int argc, char *argv[]) {
    int cases, C, N, dp[MAXN], sw[MAXN], sd1[MAXN], d2[MAXN];
    dp[0] = sw[0] = sd1[0] = 0;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> C >> N;
        // xx 和 yy 表示上一个包裹的位置, x 和 y 表示当前包裹的位置。
        for (int i = 1, xx = 0, yy = 0, x, y, w; i <= N; i++) {
            cin >> x >> y >> w;
            // 更新相应变量。
            sw[i] = sw[i - 1] + w;
            sd1[i] = sd1[i - 1] + abs(x - xx) + abs(y - yy);
            d2[i] = x + y;
            xx = x, yy = y;
            // 根据递推关系式计算最优值。
            dp[i] = INF;
            for (int j = i - 1; sw[i] - sw[j] <= C && j >= 0; j--)
                dp[i] = min(dp[i], dp[j] + d2[j + 1] + (sd1[i] - sd1[j + 1]) + d2[i]);
        }
        if (cs > 1) cout << '\n';
        cout << dp[N] << '\n';
    }
    return 0;
}
```

}

由于 UVa OJ 上的测试数据较弱, 上述时间复杂度为 $O(CN)$ 的算法已经能够在限定时间内获得 Accepted。如果 C 较大, 显然容易超时, 是否存在效率更高的解题方法呢? 答案是肯定的。对递推关系式进行变量分离操作, 可得到如下的递推关系式

$$dp[i] = \min\{dp[j] + d_2[j + 1] - sd_1[j + 1] + sd_1[i] + d_2[i], sw[i] - sw[j] \leq C, j \geq 0, i \geq 1\}$$

定义

$$g(j) = dp[j] + d_2[j + 1] - sd_1[j + 1], j \geq 0$$

$$h(i) = sd_1[i] + d_2[i], i \geq 1$$

可以发现, 对于某个固定的 i 值, 若 j_1 和 j_2 满足 $sw[i] - sw[j_1] \leq C, sw[i] - sw[j_2] \leq C$, 且有 $j_1 < j_2$ 和 $g(j_2) < g(j_1)$, 那么根据递推关系式可知, 决策 j_2 要优于决策 j_1 。由于机器人存在装载量上限 C , 每个包裹具有固定的重量 w_i , 在递推关系式中, 随着 i 的递增, j 的取值下限是一个不递减的函数。因此, 进行分离变量操作后的递推关系式满足单调队列优化的条件, 那么可以根据决策的单调性来构建一个单调队列来对决策进行“筛选”。具体方法是应用类似于双端队列的数据结构, 根据每个决策 x 所对应的函数值 $g(x)$ 以及 x 的大小关系来进行决策的“筛选”。

参考代码

```
const int MAXN = 100010;

int dp[MAXN], sw[MAXN], sd1[MAXN], d2[MAXN];

int G(int j) { return dp[j] + d2[j + 1] - sd1[j + 1]; }

int main(int argc, char *argv[]) {
    int cases, C, N;
    dp[0] = 0, sw[0] = 0, sd1[0] = 0, d2[0] = 0;
    cin >> cases;
    for (int cs = 1; cs <= cases; cs++) {
        cin >> C >> N;
        int xx = 0, yy = 0, x, y, w;
        for (int i = 1; i <= N; i++) {
            cin >> x >> y >> w;
            sw[i] = sw[i - 1] + w;
            sd1[i] = sd1[i - 1] + abs(x - xx) + abs(y - yy);
            d2[i] = x + y;
            xx = x, yy = y;
        }
        // 为了简便, 使用 STL 提供的双端队列来实现单调队列优化。
        deque<int> dq;
        dq.push_front(0);
        for (int i = 1; i <= N; i++) {
            // 根据装载量的限制从队列前端“剔除”不符合要求的候选值。
            while (!dq.empty() && sw[i] - sw[dq.front()] > C) dq.pop_front();
            dp[i] = G(dq.front()) + sd1[i] + d2[i];
            // 根据决策的单调性从队列尾端“剔除”不符合要求的候选值。
            if (i < N) while (!dq.empty() && G(dq.back()) >= G(i)) dq.pop_back();
            dq.push_back(i);
        }
        if (cs > 1) cout << '\n';
        cout << dp[N] << '\n';
    }
}
```

```

    }
    return 0;
}

```

由于每个决策都仅进入队列一次，平摊后的决策选择时间复杂度为 $O(1)$ ，最终经过单调队列优化的算法总的时间复杂度为 $O(N)$ ，可以有效地应对规模较大的测试数据。

强化练习：1427 Parade^E，12170 Easy Climb^D。

11.9.5 斜率优化

对于递推关系式形如

$$dp[i] = \max \text{ 或 } \min \{dp[j] + f(i, j)\} \quad (11.1)$$

的动态规划问题，无法直接使用前述介绍的单调队列优化技巧来提高状态转移的效率。朴素的方法是使用时间复杂度为 $O(n^2)$ 的方式进行状态转移，显然当 n 较大时会发生超时。考虑递推关系式 (11.1) 的最优值取最小值的情形，令 $k < j < i$ ，当更新 $dp[i]$ 时，如果 $dp[j] + f(i, j)$ 比 $dp[k] + f(i, k)$ 更优，则有以下不等式

$$dp[j] + f(i, j) < dp[k] + f(i, k) \quad (11.2)$$

如果能够将不等式 (11.2) 转化为以下的形式

$$\frac{Y(j) - Y(k)}{X(j) - X(k)} < f(i) \quad (11.3)$$

则可以使用一种被人们称为斜率优化的技巧来提高状态转移的效率。观察不等式 (11.3)，将 $(X(j), Y(j))$ 和 $(X(k), Y(k))$ 视为平面直角坐标系上两个点 p_j 和 p_k 的坐标，则不等式 (11.3) 的几何意义可以理解为：当直线 p_jp_k 的斜率小于 $f(i)$ 时，从状态 j 转移到 i 比从状态 k 转移到 i 要更优。令 $slope(i, j)$ 表示直线 p_jp_i 的斜率，假设有三个决策点 j, k, l ，且有 $slope(j, k) < slope(k, l)$ 。那么有两种可能：若 $slope(j, k) < f(i)$ ，则从状态 j 转移到 i 比从状态 k 转移到 i 更优；若 $f(i) < slope(j, k)$ ，则 $f(i) < slope(k, l)$ ，那么从状态 l 转移到 i 比从状态 k 转移到 i 更优。由此可知，当 $slope(j, k) < slope(k, l)$ 时，不论何种情况，从状态 k 转移到 i 都不是最优决策，因此决策点 k 可以“丢弃”。这种情形正好对应图 11-15 (a) 所示的几何图像。

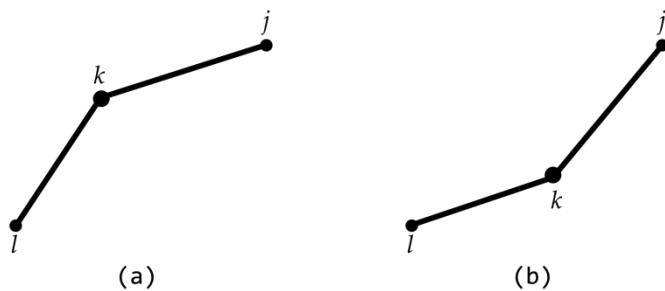


图 11-15 斜率优化。(a) 三个决策点 j, k, l 构成上凸包；(b) 三个决策点 j, k, l 构成下凸包

如果将不可能成为最优的决策点删除，那么由最优决策点形成的几何图像将会是形如图 11-13 (b) 所示的某个凸包的下半部分。在下凸包中，连续三个决策点满足斜率关系： $slope(k, l) < slope(j, k) < f(i)$ ，即对于决策点 k 和 l 来说，从 k 转移到 i 要比从 l 转移到 i 更优，对于决策点 j 和 k 来说，从 j 转移到 i 要比从 k 转移到 i 更优，那么可以维护一个斜率单调不降的队列，在保持相邻两个决策点的斜率小于 $f(i)$ 的情况下，能构成更大斜率的直线且位于更右侧的决策点将更优。

下面, 让我们通过一道题目的解析来更为直观地了解斜率优化在解题中的具体应用方法^I。

USACO 2003 March Green — Best Cow Fences

给定 N 个正整数构成的序列 a_1, a_2, \dots, a_N 和整数 F , 定义

$$A(i, j) = \frac{\sum_{k=i}^j a_k}{j - i + 1}, \quad 1 \leq i \leq j \leq N$$

确定

$$M = \max\{A(i, j) \mid 1 \leq i \leq j \leq N, F \leq j - i + 1\}$$

即求一段长度至少为 F 且平均值最大的子串, 约束条件 $1 \leq F \leq N \leq 10^5$ 。

分析

如果使用朴素的穷尽算法, 需要枚举每一个满足 $F \leq j - i + 1$ 的区间 $[i, j]$, 取其最大值。容易看出, 穷尽算法的时间复杂度为 $O(N^2)$, 由于本题中 N 较大, 显然会超时, 需要另辟蹊径来提高效率^[146]。令

$$S(i) = \sum_{j=0}^i a_j, \quad a_0 = 0, \quad i \geq 0$$

则有

$$A(i, j) = \frac{S(j) - S(i-1)}{j - (i-1)}$$

再令 $Y(i) = S(i)$, $X(i) = i$, 则 $A(i, j)$ 恰为经过二维平面上两个点 $p_{i-1}(i-1, S(i-1))$ 和 $p_j(j, S(j))$ 的直线的斜率。于是可以将原问题转化为以下问题: 给定二维平面上的点集 $p_m(m, S(m))$, m 为整数, 要求从点集中选取两个点 p_i 和 p_j , 其中 $i \leq j$, 使得 $F \leq j - i + 1$ 且直线 $p_i p_j$ 的斜率最大。

对于给定的点 p_i , 需要检查的决策点 p_j 满足条件 $1 \leq i \leq j - F + 1$, 令

$$G_j = \{p_i \mid 1 \leq i \leq j - F + 1\}$$

特别地, 当 $j < F$ 时, G_j 为空集。对于 G_j 中的三个决策点 p_i, p_k, p_l , $i < k < l < j$, 存在这样的一个性质: 如果在状态转移中决策点 p_i, p_k, p_l 先后均被用于更新最优值, 则三个决策点构成的折线段必定是某个凸包的下半部分的局部。

^I 题目描述不是原题的直接译文而是题意的抽象概括。题目来源: the United States of America Computing Olympiad (USACO: <http://www.usaco.org/index.php>, 2020) 2003 March Green — Best Cow Fences。

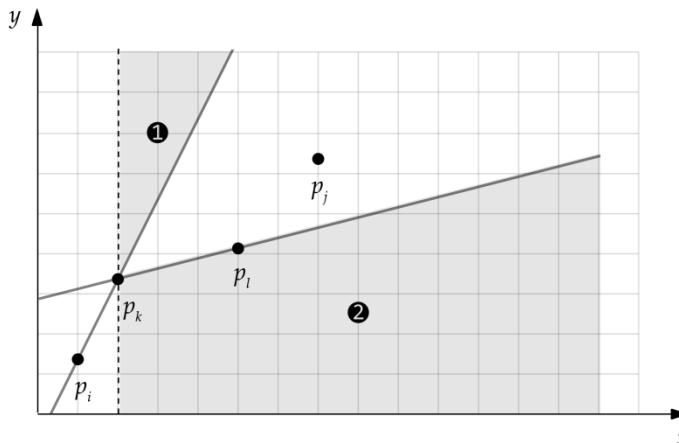


图 11-16 三个决策点 p_i, p_k, p_l 构成上凸包的局部，则 p_k 不可能成为最优决策点

如图 11-16 所示, 假设 p_i , p_k , p_l 构成的并不是下凸包的局部而是一个上凸包的局部, 可以证明 p_k 必定不会是最优决策点。使用反证法予以证明。假设 p_k 是三个决策点中的最优决策点, 那么直线 $p_k p_l$ 的斜率必须大于直线 $p_k p_i$ 的斜率, 要求 p_i 必须位于直线 $p_k p_l$ 的上方区域, 由于存在 $i < k < l < j$ 的约束, 则 p_i 只能位于 1 号阴影区域, 又由于 p_k 优于 p_l , 那么直线 $p_k p_l$ 的斜率必须大于直线 $p_k p_i$ 的斜率, 要求 p_i 必须位于直线 $p_k p_l$ 的下方区域, 由于存在 $i < k < l < j$ 的约束, 则 p_i 只能位于 2 号阴影区域, 但从图 11-16 可以看到, p_i 只可能位于两个阴影区域之一而不可能同时在两个阴影区域中, 产生矛盾, 因此当 p_i , p_k , p_l 构成的并不是下凸包的局部而是一个上凸包的局部时, p_k 不可能是三个决策点中的最优决策点, 要么 p_i 比 p_k 更优, 要么 p_l 比 p_k 更优。因此对于构成上凸包的三个决策点, 可以将位于中间的决策点安全地删除, 使得最优决策点构成的图形总是下凸包, 如图 11-17 所示。

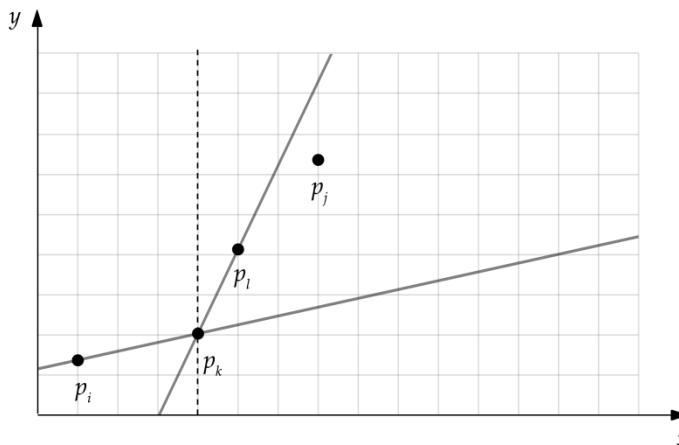


图 11-17 三个决策点 p_i, p_k, p_l 构成下凸包的局部，则 p_k 会成为最优决策点

因此, 为了提高状态转移的效率, 只需在选择与 p_j 构成最大斜率的决策点时保证已有的决策点始终构成下凸包的局部。可以使用类似于 Andrew 合并法的方式来维护下凸包^I, 由于决策点已经按照 x 坐标从左到右的顺序排列, 省去了排序的过程, 因此时间复杂度为 $O(n)$ 。那么如何寻找下凸包上与 p_j 有最大斜率的决策点呢? 可以根据下凸包上的决策点与 p_j 构成直线的斜率的单调性来确定。当 p_j 处理完毕后, 需要将 p_j 作为一个决策点加入下凸包中, 此时可能导致原有的下凸包右侧的某些决策点不再满足下凸的性质, 需要将其剔除, 如图 11-17 所示, 当 p_j 加入后, p_k , p_l , p_j 构成顺时针旋转, 即上凸, 因此 p_l 需要予以剔除。

以下是使用斜率优化技巧进行解题的参考实现代码^{II}。

参考代码

```
//-----11.9.5.cpp-----//
const int MAXN = 100010;

struct point { long long x, y; } P[MAXN];

// 外积。
long long cp(point &a, point &b, point &c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// 如果外积小于 0, 则从点 a 向点 b 望去, 点 c 位于线段 ab 的右侧, 即两条连续线段 ab 和 bc 构成右转。
bool cw(int a, int b, int c) {
    return cp(P[a], P[b], P[c]) < 0;
}

// 如果外积大于 0, 则从点 a 向点 b 望去, 点 c 位于线段 ab 的左侧, 即两条连续线段 ab 和 bc 构成左转。
bool ccw(int a, int b, int c) {
    return cp(P[a], P[b], P[c]) > 0;
}

int main(int argc, char *argv[]) {
    point maxK, k;
    int N, F, ai, si, head, rear, Q[MAXN];
    P[0].x = P[0].y = 0;
    while (cin >> N >> F) {
        // 读入数据。
        si = 0;
```

^I 读者可以参阅本书第 14 章“计算几何”中第 14.5.3 小节“Andrew 合并法”的内容。

^{II} 参考实现代码于 2019 年 7 月 31 日在 Peking University Online Judge (<http://poj.org/problem?id=2018>, 2020) 提交获得 Accepted。本题亦可使用时间复杂度为 $O(n \log n)$ 的二分搜索方法解题。很明显, 具有最大平均值的子串其平均值 M 不会小于序列中最小的元素值, 也不会大于序列中最大的元素值, 假设 m 为最大平均值, 将所有序列元素减去 m 后, 问题转化为在序列中是否能够找到一个子串, 其长度不小于 F 且和至少为 0, 可以通过动态规划在 $O(n)$ 的时间复杂度内予以判定。令 $dp(i, j)$ 表示区间 $[i, j]$ 内子串的最大平均值, 则有以下递推关系式

$$dp(i, j) = \begin{cases} 0 & \text{如果 } j - i < F \\ A(i, j) & \text{如果 } j - i = F \\ \max\{dp(i, j-1) + a_j, dp(j-F+1, j)\} & \text{如果 } j - i > F \end{cases}$$

使用二分搜索解题的参考实现代码: <https://github.com/metaphysis/Code/blob/master/Books/PCC1/11/11.9.5.1.cpp>。

```

for (int i = 1; i <= N; i++) {
    cin >> ai;
    si += ai;
    P[i].x = i, P[i].y = si;
}
maxK.x = 1, maxK.y = 0;
head = 0, rear = 0;
for (int i = 0; i + F <= N; i++) {
    // 保持已有决策点的下凸性。
    while (head + 2 <= rear && cw(Q[rear - 2], Q[rear - 1], i)) rear--;
    Q[rear++] = i;
    // 根据单调性确定最优决策点。
    while (head + 2 <= rear && ccw(Q[head], Q[head + 1], i + F)) head++;
    // 更新最大平均值。
    k.x = i + F - Q[head], k.y = P[i + F].y - P[Q[head]].y;
    if (maxK.y * k.x < maxK.x * k.y) maxK = k;
}
// 题目要求输出 1000 乘以最大平均值的整数部分。
cout << 1000 * maxK.y / maxK.x << '\n';
}
return 0;
}
//-----11.9.5.cpp-----//

```

强化练习：1451* Average^D。

11.9.6 四边形不等式优化

在区间型动态规划中，对递推关系式形如

$$B_{i, j} = \begin{cases} 0 & i = j \\ \min_{i < k \leq j} \{B_{i, k-1} + B_{k, j}\} + w(i, j) & i < j \end{cases}$$

的问题，通常的做法是使用时间复杂度为 $O(n^3)$ 的算法进行解题，当 n 较大时很容易超时。如果递推关系式中的代价函数 w 满足某些特定的性质，则可以使用四边形不等式优化（Knuth-Yao quadrangle-inequality speedup）对递推过程进行优化^I，从而使得时间复杂度从 $O(n^3)$ 下降至 $O(n^2)$ ^{[147][148][149]}。

如果 w 满足

$$w(i, j) + w(i', j') \leq w(i', j) + w(i, j'), \quad i \leq i' \leq j \leq j'$$

则称 w 满足四边形不等式（quadrangle inequality）。

若 w 满足

$$w(i, j) \leq w(i', j'), \quad i' \leq i < j \leq j', \quad \text{即 } [i, j] \subseteq [i', j']$$

则称 w 满足区间包含单调性。

令

^I 四边形不等式优化最初由 Knuth 获得并用于最优二叉查找树构建问题的优化。Yao 对 Knuth 原有较为复杂的论证进行了改进，使得证明易于理解并给出了将四边形不等式优化应用于其他问题的具体条件。Bein 等人进一步论证得出四边形不等式优化实际上是全局单调性（total monotonicity）推论的具体应用，并指出可将 SMAWK 算法应用于可进行四边形不等式优化的问题。

$$K_B(i, j) = \min \left\{ k: B_{i, j} = B_{i, k-1} + B_{k, j} + w(i, j) \right\}$$

即 $K_B(i, j)$ 为区间 $[i, j]$ 的最佳决策点 (使得 $B_{i, j}$ 取最小值的下标 k), 可以证明以下两个结论:

(1) 如果 w 满足四边形不等式和区间包含单调性, 则最优代价函数 B 亦满足四边形不等式;

(2) 当结论 (1) 成立时, 有 $K_B(i, j-1) \leq K_B(i, j) \leq K_B(i+1, j)$, $i \leq j$ 。

应用四边形不等式优化的步骤是首先证明 w 满足四边形不等式和区间包含单调性, 为了节省时间, 亦可在 $O(n^3)$ 的算法中同步验证应用四边形不等式优化的条件是否满足, 如果符合则可考虑使用优化技巧。

10304 Optimal Binary Search Tree^B (最优二叉查找树)

给定 n 个不同元素的集合 $S = \{e_1, e_2, \dots, e_n\}$, S 中的元素满足: $e_1 < e_2 < \dots < e_n$ 。考虑由 S 中所有元素构成的一棵二叉查找树, 人们期望该二叉查找树具有这样的性质: 查询频率越高的元素越靠近根结点。从一棵树中查找 S 中某个元素 e_i 的代价 $cost(e_i)$ 定义为从根结点出发到达包含此元素的结点所经过的路径的边数。给定集合 S 中所有元素各自的查询频率, $f(e_1), f(e_2), \dots, f(e_n)$, 我们定义一棵二叉查找树的总代价为

$$f(e_1) * cost(e_1) + f(e_2) * cost(e_2) + \dots + f(e_n) * cost(e_n)$$

按照上述定义, 具有最小总代价的二叉树是对应集合 S 的最佳二叉查找树表示, 故将其称之为最优二叉查找树。

输入

输入包含多组测试数据, 每组测试数据一行。每行起始为一个整数 n , $1 \leq n \leq 250$, 表示集合 S 的大小, 接着是 n 个非负整数, 表示 S 中各个元素的查询频率, $f(e_1), f(e_2), \dots, f(e_n)$, $0 \leq f(e_i) \leq 100$ 。以文件结束符表示输入终结。

输出

对于每组测试数据输出一行, 输出最优二叉查找树的总代价。

样例输入

1 5
3 10 10 10
3 5 10 20

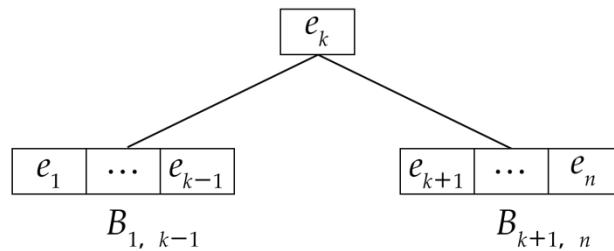
样例输出

0
20
20

分析

构建最优二叉查找树是一个经典问题, 此处给出的题目是该问题的一种变形^I。由于题目已经给定 $e_1 < e_2 < \dots < e_n$, 为了保持二叉查找树的性质, 直观的解题思路是尝试将 n 个键的每一个键 e_k 作为根结点, 这样会将给定区间以 e_k 为界分成两个子区间, 通过获取子区间 $e_1 \dots e_{k-1}$ 和 $e_{k+1} \dots e_n$ 的最优二叉查找树, 可以综合得到整体区间 $e_1 \dots e_n$ 的最优二叉查找树。

^I 关于经典的最优二分查找树 (Optimal Binary Search Tree, OBST) 问题, 读者可以参阅: Thomas H. Cormen 等著, 潘金贵等译, 《算法导论 (第二版)》, 第 212—216 页。

图 11-18 以 e_k 为根将问题分解为两个子问题

令 $B_{i, j}$ 表示 e_i 到 e_j 的元素所构成的最优二叉查找树的总代价，当 e_i 到 e_{k-1} 的元素和 e_{k+1} 到 e_j 的元素以左侧最优二叉查找子树和右侧最优二叉查找子树的形式“挂接”在以 e_k 为根的二叉树下时，由于 e_k 已经被选定为子树的根结点，除 e_k 以外的其他结点的深度都增加了 1 层，故总代价需要加上除 e_k 以外 e_i 至 e_j 的和，故有

$$B_{1, n} = B_{1, k-1} + B_{k+1, n} + e_1 + \dots + e_{k-1} + e_{k+1} + \dots + e_n$$

令 $sum[i]$ 为前 i 个键值的和，即

$$sum[i] = \sum_{j=1}^i e_j$$

定义 $sum[0]=0$ ，则有以下递推关系式

$$B_{i, j} = \begin{cases} 0 & i \geq j \\ \min_{i \leq k \leq j} \{B_{i, k-1} + B_{k+1, j} - e_k\} + sum[j] - sum[i-1] & 0 \leq i < j < n \end{cases}$$

由此可得到以下通过递推方式进行解题的关键实现¹，易知其时间复杂度为 $O(n^3)$ 。

```
int e[256], sum[256], dp[256][256];
memset(dp, 0, sizeof(dp));
// 确定长度从 1 到 n 的所有区间的最小代价，最后所求即为 dp[1][n]。
for (int L = 2; L <= n; L++)
    // 设置区间的起始下标 i 和结束下标 j。
    for (int i = 1, j = L; j <= n; i++, j++) {
        dp[i][j] = INF;
        // 以区间 [i, j] 内的每一个元素作为根结点，确定最优二叉查找树的代价。
        for (int k = i; k <= j; k++) {
            int next = dp[i][k-1] + dp[k+1][j] - e[k] + sum[j] - sum[i-1];
            dp[i][j] = min(dp[i][j], next);
        }
    }
}
```

观察递推关系式中的代价函数

$$w(i, j) = sum[j] - sum[i-1]$$

容易验证代价函数 w 满足四边形不等式和区间包含单调性。

¹ 为了便于四边形不等式优化的说明，采用自底向上的方式进行递推，使用递归方式的自顶向下递推可能更为直观和易于理解。

e_i	\dots	$e_{i'}$	$e_{i'+1}$	\dots	\dots	e_{j-1}	e_j	\dots	$e_{j'}$
-------	---------	----------	------------	---------	---------	-----------	-------	---------	----------

图 11-19 代价函数 w 满足四边形不等式: $w(i, j) + w(i', j') \leq w(i', j) + w(i, j')$, $i \leq i' \leq j \leq j'$

如图 11-19 所示, 给定 $i \leq i' \leq j \leq j'$, 有

$$\begin{aligned}
 w(i, j) + w(i', j') &= (sum[j] - sum[i-1]) + (sum[j'] - sum[i'-1]) \\
 &= (e_i + e_{i+1} + \dots + e_{j-1} + e_j) + (e_{i'} + e_{i'+1} + \dots + e_{j'-1} + e_{j'}) \\
 &= (e_{i'} + e_{i'+1} + \dots + e_{j-1} + e_j) + (e_i + e_{i+1} + \dots + e_{j'-1} + e_{j'}) \\
 &= (sum[j] - sum[i'-1]) + (sum[j'] - sum[i-1]) \\
 &= w(i', j) + w(i, j')
 \end{aligned}$$

$e_{i'}$	\dots	e_i	e_{i+1}	\dots	\dots	e_{j-1}	e_j	\dots	$e_{j'}$
----------	---------	-------	-----------	---------	---------	-----------	-------	---------	----------

图 11-20 代价函数 w 满足区间包含单调性: $w(i, j) \leq w(i', j')$, $i' \leq i < j \leq j'$

如图 11-20 所示, 给定 $i' \leq i < j \leq j'$, 有

$$\begin{aligned}
 w(i, j) &= sum[j] - sum[i-1] \\
 &= e_i + e_{i+1} + \dots + e_{j-1} + e_j \\
 &\leq e_{i'} + \dots + e_i + e_{i+1} + \dots + e_{j-1} + e_j + \dots + e_{j'} \\
 &= sum[j'] - sum[i'-1] \\
 &= w(i', j')
 \end{aligned}$$

由于递推关系式中的代价函数 w 满足应用四边形不等式优化的约束条件, 递推过程可以进一步优化为:

```

int e[256], sum[256], dp[256][256], K[256][256];
memset(dp, 0, sizeof(dp));
for (int i = 1; i <= n; i++) K[i][i] = i;
for (int L = 2; L <= n; L++) {
    for (int i = 1, j = L; j <= n; i++, j++) {
        dp[i][j] = INF;
        for (int k = K[i][j-1]; k <= K[i+1][j]; k++) {
            int next = dp[i][k-1] + dp[k+1][j] - e[k] + sum[j] - sum[i-1];
            if (next < dp[i][j]) {
                dp[i][j] = next;
                K[i][j] = k;
            }
        }
    }
}

```

回顾之前介绍的典型的石子合并问题, 由于其递推关系式为

$$dp[i][j] = \min_{i \leq k \leq j} \{dp[i][k] + dp[k+1][j]\} + sum[j] - sum[i-1], \quad dp[i][i] = 0, \quad 1 \leq i < j < n$$

若定义代价函数

$$w(i, j) = sum[j] - sum[i-1], \quad sum[0] = 0, \quad 1 \leq i < j < n$$

由前述论证易知 $w(i, j)$ 满足四边形不等式和区间包含单调性, 因此典型的石子合并问题可以应用四边形不等式优化技巧提高求解效率, 从而能够使得时间复杂度从 $O(n^3)$ 降为 $O(n^2)$ 。需要注意, 如果使用备忘技巧结合递归的方式求解石子合并问题, 则在递归过程中不能使用四边形不等式优化, 会产生错误, 只能在自底向上使用迭代方法递推的时候使用该优化技巧。究其原因, 在于递归求解时会导致优化数组的状态不一致而出

现错误。

强化练习: [10003* Cutting Sticks^A](#)。

11.10 子序列和子串问题

11.10.1 最短编辑距离

最短编辑距离(minimum edit distance, 又称 Leviathan 距离)是指通过删除(deletion)、插入(insertion)、替换(substitution)三种操作, 将源字符串 S 变换为目标字符串 T 所需最少的操作步骤数。设源字符串 S 为“INTENTION”, 目标字符串 T 为“EXECUTION”, 则将 S 变换为 T 具有最少操作步骤数的一种方案是:

- S : INTENTION 将源字符串第一个字符 ‘I’ 替换为目标字符串第一个字符 ‘E’
- ENTENTION 将当前字符串第二个字符 ‘N’ 替换为目标字符串第二个字符 ‘X’
- EXTENTION 删除当前字符串的第三个字符 ‘T’
- EXENTION 将当前字符串第四个字符 ‘N’ 替换为目标字符串第四个字符 ‘C’
- EXEC*TION 在当前字符串第五个字符位置 (‘*’ 号处) 插入目标字符串第五个字符 ‘U’

T : EXECUTION 后面字符相同, 不需操作, 变换结束, 总共五个步骤。

由上例可以观察到, 在将源字符串变换为目标字符串的过程中, 是按从左至右的顺序进行的, 每次字符的操作位置递增 1, 这个规律不是特例, 实际上对源串的任何修改都可以改写成从左到右的顺序, 使得每次的字符操作都发生在已处理字符串的“最末位置”(此处的最末位置不是指整个字符串真正的最末位置, 而是指操作的最末位置)。可将上述过程归纳如下: 在从源串变换为目标串的过程中, 要么删除源串的最后一个字符, 要么在源串最后插入一个字符, 要么将源串最后一个字符替换成目标串的最后一个字符(如果两者相同则不需替换操作), 由此可知, 在计算 S 和 T 的最短编辑距离时, 需要计算 S 和 T 的子字符串的最短编辑距离。约定使用形如 $A[p..q]$ 的符号表示字符串 A 中从下标 p 到下标 q 的连续子串(从下标 1 开始计数)。

令 $D(i, j)$ 表示 $S[1..i]$ 和 $T[1..j]$ 的最短编辑距离, 那么此问题的递推关系式为

$$D(i, j) = \min\{D(i-1, j) + 1, D(i, j-1) + 1, D(i-1, j-1) + c(i, j)\}$$

如果 $S[i] \neq T[j]$, $c(i, j) = 1$, 否则 $c(i, j) = 0$

其中 $D(i-1, j)$ 表示 $S[1..(i-1)]$ 变换为 $T[1..j]$ 的最短编辑距离, 因为 $S[1..i]$ 需要移除最末一个字符到达 $S[1..(i-1)]$, 然后再到达 $T[1..j]$, 对应删除操作, 操作步骤数增加 1; $D(i, j-1)$ 表示 $S[1..i]$ 变换为 $T[1..(j-1)]$ 的最短编辑距离, 因为 $S[1..i]$ 通过此方式到达了 $T[1..(j-1)]$, 仍然需要增加一个字符才能到达 $T[1..j]$, 因此相当于在 $S[1..i]$ 末尾增加一个字符, 对应插入操作, 操作步骤数增加 1; $D(i-1, j-1)$ 表示 $S[1..(i-1)]$ 到达 $T[1..(j-1)]$ 的最短编辑距离, 如果 $S[i]$ 和 $T[j]$ 不等, 只需将 $T[j]$ 替换 $S[i]$ 即可, 操作步骤数增加 1, 否则无需替换, 操作步骤数不变化。

由以上递推关系, 可以通过基于表格的动态规划解决最短编辑距离问题。最初时, 将 $S[1..i]$ 变换为 $T[0]$ 的操作步骤数为 i , 即进行 i 次删除操作; 将 $S[0]$ 变换为 $T[1..j]$ 的操作步骤为 j , 即进行 j 次插入操作, 故有 $D(i, 0) = i$, $D(0, j) = j$ 。

```
//-----11.10.1.cpp-----
int dp[1024][1024];
int med(string &S, string &T) {
    dp[0][0] = 0;
    int M = S.length(), N = T.length();
    for (int i = 1; i <= M; i++) dp[i][0] = i;
    for (int j = 1; j <= N; j++) dp[0][j] = j;
```

```

for (int i = 1; i <= M; i++)
    for (int j = 1; j <= N; j++) {
        int deleted = dp[i - 1][j] + 1, inserted = dp[i][j - 1] + 1;
        int replaced = dp[i - 1][j - 1];
        if (S[i - 1] != T[j - 1]) replaced = dp[i - 1][j - 1] + 1;
        dp[i][j] = min(min(deleted, inserted), replaced);
    }
return dp[M][N];
}
//-----11.10.1.cpp-----

```

利用动态规划可以得到最少的操作步骤，具体的操作序列可以从后往前反向查找进行构建。利用递归可以将解正向输出。

164 String Computer^B (字符串处理机)

Extel 刚刚购进了一台最新的计算机——命名为 X9091 的字符串处理机，它总共只有三条指令，均用于对字符进行特定操作：

- (1) 删除 (Delete) 字符串指定位置字符的指令；
- (2) 将字符插入 (Insert) 到字符串指定位置的指令；
- (3) 将字符串指定位置字符更改 (Change) 为其他字符的指令；

为此计算机编写的程序以机器码进行表示，每条指令具有形如 ZXdd 的形式——Z 表示指令所代表的操作码（删除——D，插入——I 或者更改——C），X 表示操作的字符，dd 表示一个两位整数。程序以一个特定的停止指令结束，该指令为“E”。注意，每条指令在执行时对内存中的字符串都起作用。

下面以一个例子来说明指令的工作方式。假如需要将字符串“abcde”变换为“bcgfe”，一种方式是通过一系列的更改 (Change) 命令来实现，但是这样的操作步骤数不是最小化的，以下的操作步骤更好：

Da01	abcde	
Cg03	bcde	% 注意，指令中的字符 ‘a’ 是必需的，硬件会进行相应的检查
If04	bcge	
E	bcgfe	% 程序终止

编写程序，读入两个字符串（输入字符串和目标字符串），生成具有最少操作步骤数的 X9091 程序，以便将输入字符串变换为目标字符串。如果有多种解决方案，只需给出一种即可。任何满足上述规则的解决方案都可以接受。

输入与输出

输入包含多行。每行包含两个字符串，中间以一个空格分隔。每个字符串由不超过 20 个小写英文字母组成。输入以只包含 '#' 字符的一行结束。

输出由多行组成，每行输出对应一行输入。每行输出包含了以 X9091 语言编写的转换程序。

样例输入

```
abcde bcgfe
#
```

分析

直接使用前述介绍的动态规划算法解题即可。需要注意的是，由于题目要求输出相应的变换操作及其序

样例输出

```
Da01Cg03If04E
```

号, 在进行更改 (Change) 和插入 (Insert) 操作时, 其序号位置都是相对于目标串来说, 不会发生变化, 直接输出即可。进行删除 (Delete) 操作是相对于源字符串的, 其序号会因为之前的删除和插入操作而发生变化, 因此需要记录删除和插入操作的次数, 根据次数对后续的序号进行适当调整。

参考代码

```

const int NONE = -1, DELETE = 0, INSERT = 1, CHANGE = 2, MATCH = 3;

// 定义动态规划表格单元。
struct cell { int cost, operation; };

cell cells[25][25];
string S, T, operationCode = "DIC";
int M, N, deletions, insertions;

// 显示操作步骤, 注意删除操作其序号会因为已有的删除和插入操作而发生变化。
void displayPath(int i, int j) {
    if (cells[i][j].operation >= DELETE && cells[i][j].operation <= CHANGE) {
        cout << operationCode[cells[i][j].operation];
        if (cells[i][j].operation == CHANGE) {
            cout << T[j];
            cout << setw(2) << setfill('0') << j;
        }
        else if (cells[i][j].operation == DELETE) {
            cout << S[i];
            cout << setw(2) << setfill('0') << (i + insertions - deletions);
            deletions++;
        }
        else if (cells[i][j].operation == INSERT) {
            cout << T[j];
            cout << setw(2) << setfill('0') << j;
            insertions++;
        }
    }
}

// 利用递归构建操作步骤。
void findPath(int i, int j) {
    if (cells[i][j].operation != NONE) {
        if (cells[i][j].operation == DELETE)
            findPath(i - 1, j);
        else if (cells[i][j].operation == INSERT)
            findPath(i, j - 1);
        else
            findPath(i - 1, j - 1);
    }
    displayPath(i, j);
}

void med() {
    // 为每个字符串起始位置增加一个空格, 将字符串序号和表格序号对齐, 方便处理。
    S = ' ' + S;
    T = ' ' + T;
    M = S.length() - 1;
    N = T.length() - 1;
}

```

```

// 初始化动态规划表格。
cells[0][0] = (cell){0, NONE};
for (int i = 1; i <= M; i++) cells[i][0] = (cell){i, DELETE};
for (int j = 1; j <= N; j++) cells[0][j] = (cell){j, INSERT};

// 自底向上动态规划求解。
for (int i = 1; i <= M; i++) {
    for (int j = 1; j <= N; j++) {
        cells[i][j] = (cell){cells[i - 1][j].cost + 1, DELETE};
        if (cells[i][j].cost > (cells[i][j - 1].cost + 1))
            cells[i][j] = (cell){cells[i][j - 1].cost + 1, INSERT};
        if (S[i] == T[j]) {
            if (cells[i][j].cost > cells[i - 1][j - 1].cost)
                cells[i][j] = (cell){cells[i - 1][j - 1].cost, MATCH};
        } else {
            if (cells[i][j].cost > (cells[i - 1][j - 1].cost + 1))
                cells[i][j] = (cell){cells[i - 1][j - 1].cost + 1, CHANGE};
        }
    }
}

// 反向构建操作步骤。
deletions = insertions = 0;
findPath(M, N);
cout << "E" << endl;
}

int main(int argc, char *argv[]) {
    while (cin >> S, S != "#" && cin >> T) med();
    return 0;
}

```

强化练习：[526 String Distance and Transform Process^B](#)，[671 Spell Checker^C](#)，[1207 AGTC^C](#)，[10069 Distinct Subsequences^A](#)。

扩展练习：[963* Spelling Corrector^E](#)。

11.10.2 最长公共子序列

一个给定序列的子序列就是将该序列去掉零个或者多个元素所形成的序列。更为形式化的定义是：给定一个序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ ，若称另一个序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ 是 X 的一个子序列，则存在 X 的一个严格递增的下标序列 $\langle i_1, i_2, \dots, i_k \rangle$ ，使得对所有的 $j = 1, 2, \dots, k$ ，有 $x_{i_j} = z_j$ 。例如， $Z = \langle B, C, D, B \rangle$ 是 $X = \langle A, B, C, B, D, A, B \rangle$ 的一个子序列，相应的下标序列为 $\langle 2, 3, 5, 7 \rangle$ 。

给定两个序列 X 和 Y ，如果 Z 既是 X 的一个子序列又是 Y 的一个子序列，则称序列 Z 是 X 和 Y 的公共子序列。在所有公共子序列中，具有最大长度的公共子序列称为最长公共子序列（Longest Common Subsequence，LCS）。

最长公共子序列问题可以使用动态规划予以解决。令序列 $Y = \langle y_1, y_2, \dots, y_n \rangle$ ， Z 是 X 和 Y 的一个最长公共子序列，如果有 $x_m = y_n$ ，那么必有 $z_k = x_m = y_n$ ，可以推出 Z_{k-1} 是 X_{m-1} 和 Y_{n-1} 的一个最长公共子序列；如果 $x_m \neq y_n$ 且 $z_k \neq x_m$ ，则可推出 Z 是 X_{m-1} 和 Y 的一个最长公共子序列；如果 $x_m \neq y_n$ 且 $z_k \neq y_n$ ，则可推出 Z 是 X 和 Y_{n-1} 的一个最长公共子序列。设 $L(i, j)$ 表示序列 X_i 和 Y_j 的最长公共子序列长度，当序列 X_i 或序列 Y_j 的长度为 0 时，最长公共子序列长度为 0；若已知 $L(i-1, j-1)$ ，如果有 $x_i = y_j$ ($i, j > 0$)，那么可以将此元素附加在目前得到的最长公共子序列末尾形成一个新的最长公共子序列，长度为 $L(i-1, j-1) + 1$ ；若 x_i

$\neq y_j$ ($i, j \geq 0$), 则此元素不能作为一个公共元素看待, 不能增加当前得到的公共子序列长度, 应转而检查 $L(i, j-1)$ 和 $L(i-1, j)$, 找到两者最大值作为新的最长公共子序列长度。因此有以下递推关系式:

$$L(i, j) = \begin{cases} 0 & i = 0 \text{ 或 } j = 0 \\ L(i-1, j-1) + 1 & i > 0, j > 0 \text{ 且 } x_i = y_j \\ \max\{L(i, j-1), L(i-1, j)\} & i > 0, j > 0 \text{ 且 } x_i \neq y_j \end{cases}$$

那么可以根据以上递推关系计算最长公共子序列的长度。如果需要得到构成最长公共子序列的字符, 需要设立一个数组记录 $L(i, j)$ 由何处更新而来, 由此数组回溯得到具体的公共子序列。以下代码确定两个给定字符串的最长公共子序列长度, 并输出任意一种最长公共子序列。

```
-----11.10.2.cpp-----
const int IMINUS_JMINUS = 1, IMINUS = 2, JMINUS = 3;

// length 表示 LCS 的长度, from 表示此长度从何种方式更新而来, 用于重建 LCS。
struct state { int length, from; };

void lcs(string &s, string &t) {
    state dp[s.length() + 1][t.length() + 1] = {};
    // 根据递推关系确定 LCS。
    for (int i = 1; i <= s.length(); i++)
        for (int j = 1; j <= t.length(); j++)
            if (s[i - 1] == t[j - 1]) {
                if (dp[i][j].length < dp[i - 1][j - 1].length + 1) {
                    dp[i][j].length = dp[i - 1][j - 1].length + 1;
                    dp[i][j].from = IMINUS_JMINUS;
                }
            } else {
                if (dp[i][j].length < dp[i - 1][j].length)
                    dp[i][j].length = dp[i - 1][j].length, dp[i][j].from = IMINUS;
                if (dp[i][j].length < dp[i][j - 1].length)
                    dp[i][j].length = dp[i][j - 1].length, dp[i][j].from = JMINUS;
            }
    // 输出 LCS 的长度。
    cout << "LCS: length = " << dp[s.length()][t.length()].length;
    // 根据更新过程中的记录重建 LCS。
    string subsequence;
    int endi = s.length(), endj = t.length();
    while (dp[endi][endj].from) {
        if (dp[endi][endj].from == IMINUS_JMINUS) {
            subsequence.push_back(s[endi - 1]);
            endi -= 1, endj -= 1;
        } else {
            if (dp[endi][endj].from == IMINUS) endi -= 1;
            else endj -= 1;
        }
    }
    reverse(subsequence.begin(), subsequence.end());
    cout << " subsequence = " << subsequence << '\n';
}
-----11.10.2.cpp-----
```

强化练习: 363 Approximate Matches^E, 531 Compromise^A, 10066 The Twin Towers^A, 10100 Longest Match^A, 10192 Vacation^A, 10405 Longest Common Subsequence^A。

扩展练习: 10635 Prince and Princess^A, 11151 Longest Palindrome^A。

11.10.3 最长公共子串

在最长公共子序列问题中, 所求子序列的元素下标不需要是连续的, 而在最长公共子串 (Longest Common Substring, LCS) 中, 要求元素的下标是连续的。

给定两个非空字符串 X 和 Y , 朴素的方法是从 X 的每一个字符开始, 在 Y 中找到相同的字符后开始往后扫描寻找公共子串, 获取其中长度最长的公共子串作为结果。该方法由于重复扫描字符串, 效率较低, 时间复杂度为 $O(m^2n^2)$, 其中 m 为字符串 X 的长度, n 为字符串 Y 的长度。

令 $X = \text{“abcdbc”}$, $Y = \text{“dbcd}\mathbf{b}$ ”, 将字符串中字符的匹配情况以矩阵的形式表示为

	a	b	c	d	b	c
d	0	0	0	1	0	0
b	0	1	0	0	1	0
c	0	0	1	0	0	1
d	0	0	0	1	0	0
b	0	1	0	0	1	0

观察矩阵, 可以得到如下结论: 求最长公共子串等价于求该矩阵对角线上连续 1 的最大长度。但是将匹配情况表示成上述方式仍然不够便利, 如果规定当矩阵的某个元素值为 1 时, 若它的左上角元素值不为 0, 则此元素的值为左上角的元素值加 1, 那么可以将矩阵变换为

	a	b	c	d	b	c
d	0	0	0	1	0	0
b	0	1	0	0	2	0
c	0	0	2	0	0	3
d	0	0	0	3	0	0
b	0	1	0	0	4	0

最终最长公共子串问题可以转化为求此矩阵中的最大元素值, 这显然方便得多, 因为矩阵可以在 $O(mn)$ 的时间内构建得到, 最大值只需在求解过程中用一个变量记录即可。从动态规划的角度考虑, 由于公共子串中各元素是相邻的, 如果 $X_i = Y_j$, 则该元素可以附加在目前得到的最长公共子串后形成一个更长的公共子串, 否则最长公共子串不变。令 $L(i, j)$ 表示以序列 X_i 和 Y_j 的最末元素结尾的最长公共子串长度, 当序列 X_i 或序列 Y_j 的长度为 0 时, 最长公共子串长度为 0; 若已知 $L(i-1, j-1)$, 如果有 $x_i = y_j$, $i, j \geq 0$, 那么可以将此元素附加在目前得到的最长公共子串末尾形成一个新的最长公共子串, 长度为 $L(i-1, j-1) + 1$; 若 $x_i \neq y_j$, $i, j \geq 0$, 则以序列 X_i 和 Y_j 的最末元素结尾的公共子串长度为 0, 即 $L(i, j) = 0$ 。因此有递推关系:

$$L(i, j) = \begin{cases} 0 & i = 0 \text{ 或 } j = 0 \\ L(i-1, j-1) + 1 & i > 0, j > 0 \text{ 且 } x_i = y_j \\ 0 & i > 0, j > 0 \text{ 且 } x_i \neq y_j \end{cases}$$

在下述参考实现中, 如果两个字符串具有多个长度相同的公共子串, 只是得到在字符串中起始序号最小的公共子串。此算法的时间复杂度为 $O(mn)$, m 和 n 分别为两个字符串的长度。

```
//-----11.10.3.cpp-----
pair<int, int> lcs(string &s, string &t) {
    if (s.length() == 0 || t.length() == 0) return make_pair(-1, -1);
    int maxStart = 0, maxLength = 0;
    int dp[s.length() + 1][t.length() + 1] = {};
```

```

        for (int i = 1; i <= s.length(); i++)
            for (int j = 1; j <= t.length(); j++) {
                if (s[i - 1] == t[j - 1]) dp[i][j] = dp[i - 1][j - 1] + 1;
                else dp[i][j] = 0;
                if (dp[i][j] > maxLength) maxStart = i - dp[i][j], maxLength = dp[i][j];
            }
        return make_pair(maxStart, maxLength);
    }
//-----11.10.3.cpp-----

```

11.10.4 最长递增子序列

给定一个由小写字母组成的字符串 $s = \text{"apbtcdzefbcg"}$, 要求去掉若干个字符形成一个长度为 n 的新字符串 s' , 而 s' 中按照字典序前一个字符总是小于后一个字符, 即对于 $0 \leq i < j < n$, 有 $s'[i] < s'[j]$, 求满足此要求的字符串 s' 的最大长度。类似于前述的问题, 给定有 n 个数的序列, 要求从此序列中按从前到后的顺序取出若干个数排成一个新序列, 保持原有的相对位置不变, 且要求新序列中前一个数要严格小于后一个数, 求能够得到的最长序列长度……诸如此类问题, 均可以归结为求最长递增子序列 (Longest Increasing Subsequence, LIS) 问题。

LIS 问题更为形式化的定义是: 给定一个序列 $S = \langle s_1, s_2, \dots, s_n \rangle$, 若存在一个严格递增的下标序列 $\langle i_1, i_2, \dots, i_k \rangle$, 使得对所有的 $j = 1, 2, \dots, k-1$, 有 $s_{i_j} < s_{i_{j+1}}$, 则称序列 $S' = \langle s_{i_1}, s_{i_2}, \dots, s_{i_k} \rangle$ 为序列 S 的一个递增子序列, 具有最大长度的序列 S' 称为最长递增子序列。如前例给出的字符串 “apbtcdzefbcg”, 它的最长递增子序列是 “abc~~defg~~”, 长度为 7。更改前后元素大小所要满足的条件, 可以得到其他类型子序列的定义: 当满足 $s_{i_j} > s_{i_{j+1}}$ 时, 称为最长递减子序列 (Longest Decreasing Subsequence, LDS); 当满足 $s_{i_j} \leq s_{i_{j+1}}$ 时, 称为最长不递减子序列; 当满足 $s_{i_j} \geq s_{i_{j+1}}$ 时, 称为最长不递增子序列。

LIS 问题可通过动态规划解决。首先介绍时间复杂度为 $O(n^2)$ 的算法。令 $L(i)$ 表示以第 i 个元素作为最末元素的递增子序列的最大长度。初始时, 各个元素本身构成了一个长度为 1 的递增子序列, 因此 $L(i) = 1$ 。假设已经知道了以第 $1, 2, \dots, i-2, i-1$ 个元素作为最末元素时所能得到的递增子序列的最大长度 $L(j)$, $1 \leq j \leq i-1$, 现在来处理第 i 个元素。如果第 i 个元素大于第 j 个元素, 则可将第 i 个元素附加在长度为 $L(j)$ 的递增子序列最后形成一个新的递增子序列, 其长度为 $L(j) + 1$, 否则能够得到的递增子序列最大长度仍为 $L(j)$ 。对已有的以前 $i-1$ 个元素作为最末元素的递增子序列逐一比较, 选取长度最大者即为以第 i 个元素作为最末元素的递增子序列的最大长度。由此可以得出递推关系:

$$L(i) = \max \begin{cases} L(j), & s_i \leq s_j \\ L(j) + 1, & s_i > s_j \end{cases} \quad i \geq 2, 1 \leq j < i$$

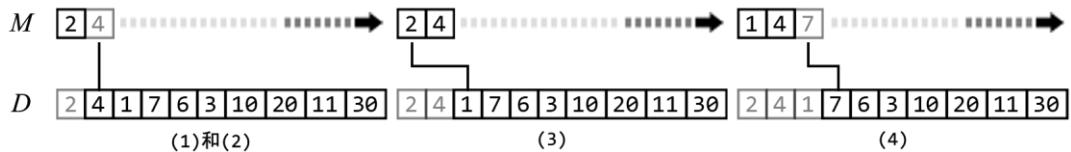
对于其他类型的子序列, 可以类似地推导出对应的递推关系式。

在上述时间复杂度为 $O(n^2)$ 的算法中, 由于保存的是以第 i 个元素作为最末元素时递增子序列的最大长度, 每当处理一个新的元素时, 需要对以之前元素作为最末元素的递增子序列进行比较后才能决定最后的最大长度, 每一次都有 $i-1$ 次比较, 各次比较次数形成一个项差为 1 的等差数列, 其总的操作步骤是 n^2 级别, 故时间复杂度为 $O(n^2)$ 。如果能够减少此步骤的时间, 那么可以将算法效率进一步提高。由此产生了时间复杂度为 $O(n \log n)$ 的算法。

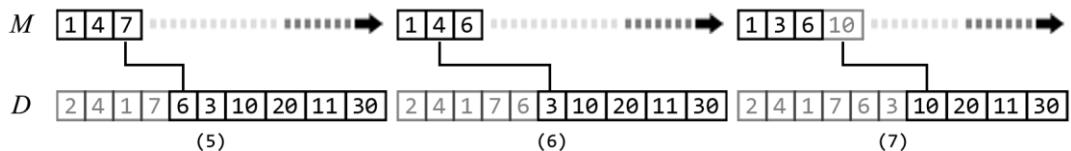
$O(n \log n)$ 的算法思想如下: 令 $M(i)$ 存放的是长度为 i 的递增子序列最末元素中的最小值, M 具有单调递增的性质, 即对于 a, b , 如果 $1 \leq a < b \leq i$, 有 $M(a) < M(b)$ 。当处理第 j 个元素时, 如果第 j 个元素大于 $M(i)$

-1), 则可将第 j 个元素附加在 $M(i-1)$ 之后, 形成一个长度为 i 的递增子序列, $M(i)=s_j$ 。如果第 j 个元素小于等于 $M(i-1)$, 根据 M 具有的单调递增性质, 不需要逐一进行比较, 只需使用二分查找法, 找到最小的长度 k , 满足 $s_j < M(k)$, 使用 s_j 来更新 $M(k)$ 即可, 最后 M 的大小即为最长递增子序列的长度。

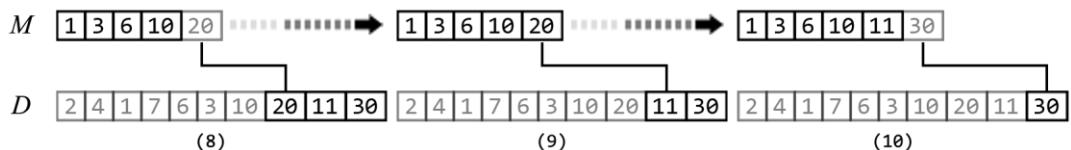
为了使读者能够更好地理解时间复杂度为 $O(n \log n)$ 的算法, 下面通过一个实例来观察算法的执行过程, 以便更为直观地理解算法的步骤。如图 11-21 所示, 设有整数序列 $D = \langle 2, 4, 1, 7, 6, 3, 10, 20, 11, 30 \rangle$, 需要确定其最长递增子序列, 以下是 $O(n \log n)$ 算法的执行过程。



算法执行的第 1 步至第 4 步。第 1 步: 初始时 $M(1)=2$; 第 2 步: 处理第二个元素 4, 由于 4 大于 $M(1)=2$, 可将 4 附加到 $M(1)=2$ 后形成长度为 2 的递增子序列, 此时有 $M(2)=4$; 第 3 步: 处理第三个元素 1, 由于 1 小于 $M(2)=4$, 无法附加在 $M(2)=4$ 之后构成长度为 3 的递增子序列, 此时需要更新当前的 LIS 所包含的元素, 从后往前查找可知第一个大于 1 的为 $M(1)=2$, 那么长度为 1 的递增子序列最小的最末元素需要更新为 1, 此时有 $M(1)=1$; 第 4 步: 处理第四个元素 7, 由于 7 大于 $M(2)=4$, 可将 7 附加到 $M(2)=4$ 后形成长度为 3 的递增子序列, 此时有 $M(3)=7$



算法执行的第 5 步至第 7 步。第 5 步: 处理第五个元素 6, 由于 6 小于 $M(3)=7$, 无法附加在 $M(3)=7$ 之后构成长度为 4 的最长递增子序列, 此时需要更新当前的 LIS 所包含的元素, 从后往前查找可知第一个大于 6 的为 $M(3)=7$, 那么长度为 3 的递增子序列最小的最末元素需要更新为 6, 此时有 $M(3)=6$; 第 6 步: 处理第六个元素 3, 按照前述过程, 得到 $M(2)=3$; 第 7 步: 处理第七个元素 10, 同理得到 $M(4)=10$



算法执行的第 8 步至第 10 步。第 8 步: 对于第八个元素 20, 产生 $M(5)=20$; 第 9 步: 对于第九个元素 11, 产生 $M(5)=11$; 第 10 步: 对于第十个元素 30, 产生 $M(6)=30$, 处理结束, M 的大小为 6, 则 LIS 的长度为 6。需要注意, 此时的 $M(1) \sim M(6)$ 构成的序列为 $\langle 1, 3, 6, 10, 11, 30 \rangle$, 并不是实际的 LIS, 实际的 LIS 是 $\langle 2, 4, 7, 10, 11, 30 \rangle$ 或者 $\langle 2, 4, 6, 10, 11, 30 \rangle$ 。要得到真正的 LIS, 需要在算法更新长度的过程中保存构成当前 LIS 的相应元素值

图 11-21 在求 LIS 的过程中数组 M 的更新

由于在序列 M 的生成和更新过程中, 需要维护序列 M 递增的性质, 当某个新增的元素 x 不能与原有的 M 序列的最末元素构成更长的递增子序列时, 需要使用 x 来更新序列 M 中的某个元素值 $M(j)$ 。如果在查找 $M(j)$ 的过程中使用时间复杂度为 $O(\log n)$ 的二分查找算法来寻找元素的更新位置, 那么就能够使得算法最终

的时间复杂度为 $O(n \log n)$ 。应用 $O(n \log n)$ 算法的一个难点是编写正确的二分查找实现，由于此处的二分查找不是查找一个具体的数，而是查找满足特定需求的一个序号，不应使用算法库中的二分查找函数 `binary_search`，而应使用 `lower_bound` 函数或 `upper_bound` 函数来进行查找¹。

111 History Grading^A (历史成绩判分)

考虑一次历史课的考试，在该次考试中，学生需要将一些历史事件按照时间顺序进行排序。将所有事件正确排序的学生会获得满分，但是那些只将部分历史事件正确排序的学生该如何判分呢？

可行的判分方法包括：

- (1) 对排序正确的历史事件，每个给 1 分；
- (2) 对学生给出的符合时间先后顺序的最长事件序列（事件不需要连续），每个事件给 1 分。

例如，给定四个历史事件，正确的先后顺序是 1 2 3 4，则排序 1 3 2 4 按照第一种判分方法的得分为 2 分（事件 1 和事件 4 的排序正确），按第二种判分方法得分为 3 分（事件序列 1 2 4 和 1 3 4 都有正确的相对顺序）。

在本问题中，要求你编程使用第二种方法为学生的答案判分。

给定 n 个事件 1, 2, …, n ，序列 c_1, c_2, \dots, c_n , $1 \leq c_i \leq n$ ，表示历史事件 i 在正确的排序中其对应的位置序号；序列 r_1, r_2, \dots, r_n , $1 \leq r_i \leq n$ ，表示历史事件 i 在学生提交的答案中的对应排序位置序号，确定在学生提交的答案中能够找到的最长（不一定连续）且具有正确相对时间顺序的事件序列长度。

输入

输入的第一行包含一个整数 n ，表示历史事件的数量， $2 \leq n \leq 20$ 。第二行包含 n 个整数，给出 n 个历史事件的正确时间排序。在接下来的输入行中，每行包含 n 个整数，表示学生对 n 个历史事件的时间排序。除第一行以外，其他输入行每行均包括 n 个整数，这 n 个整数均在 $[1, n]$ 的范围内，每行 1 至 n 的整数只出现一次，两个整数之间以一个或多个空格分隔。

输出

对于输入中每个学生的历史事件排序，输出其对应的分数。每个学生的分数各占一行。

样例输入 1

```
4
4 2 3 1
1 3 2 4
3 2 1 4
2 3 4 1
```

样例输入 2

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

样例输出 1

```
1
2
3
```

样例输出 2

```
6
5
10
9
```

¹ 关于这两个库函数的介绍，读者可参考本书第 4 章“排序与查找”第 4.9 节“算法库函数”的内容。

分析

注意题意的描述，输入中给出的是每个事件在排序好的事件序列中的位置序号，如样例输入 2 中的第二行“3 1 2 4 9 5 10 6 8 7”，它表示第一个历史事件应当排在第 3 位，而不是指第三个历史事件排在第 1 位（第三个历史事件的实际位置应该排在第 2 位）。由于数据量不大，使用时间复杂度为 $O(n^2)$ 的算法即可顺利获得通过。以下给出的是时间复杂度为 $O(n \log n)$ 的解题方案。

参考代码

```

vector<int> order, events;

int getScores() {
    vector<int> M; M.push_back(events.front());
    for (auto it = events.begin() + 1; it != events.end(); it++)
        if (*it > M.back()) M.push_back(*it);
        else {
            auto location = upper_bound(M.begin(), M.end(), *it);
            *location = *it;
        }
    return M.size();
}

int main(int argc, char *argv[]) {
    int n, index;
    string line;
    cin >> n;
    order.resize(n);
    events.resize(n);
    for (int i = 1; i <= n; i++) {
        cin >> index;
        order[index - 1] = i;
    }
    cin.ignore(1024, '\n');
    while (getline(cin, line)) {
        istringstream iss(line);
        for (int i = 1; i <= n; i++) {
            iss >> index;
            events[index - 1] = find(order.begin(), order.end(), i) - order.begin();
        }
        cout << getScores() << '\n';
    }
    return 0;
}

```

强化练习：[103 Stacking Boxes^A](#)，[231 Testing the CATCHER^A](#)，[481 What Goes Up^A](#)，[497 Strategic Defense Initiative^A](#)，[1196 Tiling Up Blocks^C](#)，[10131 Is Bigger Smarter^A](#)，[10534 Wavio Sequence^A](#)，[11003 Boxes^B](#)，[11790 Murcia's Skyline^A](#)。

扩展练习：[11240* Antimonotonicity^D](#)，[11456* Trainsorting^A](#)，[12002* Happy Birthday^D](#)。

11.10.5 最长不重复子串

最长不重复子串（longest substring without repeating characters）是字符串的一个子串，该子串中的字符互不相同，且在所有满足要求的子串中长度最大。需要注意，给定字符串中可能包含多个最长不重复子串。朴素的方法是以字符串中的每个字符作为起始字符向后扫描，直到遇到重复的字符时停止，计数不重复

字符的个数，然后取所有不重复子串的最大长度。该方法容易实现，但是效率不高，时间复杂度为 $O(n^2)$ 。

以下介绍时间复杂度为 $O(n)$ 的算法。令以第 i 个字符结尾的最长不重复子串长度为 $L(i)$ ， $i \geq 0$ 。显然，对于非空字符串来说， $L(0)=1$ 。当处理到第 i 个字符时 ($i \geq 1$)，若此字符未在以第 $i-1$ 个字符结尾的最长不重复子串中出现，则第 i 个字符可以附加在其后，构成一个更长的不重复子串；若第 i 个字符与以第 $i-1$ 个字符结尾的最长不重复子串中的某个字符 x 相同，那么以第 i 个字符结尾的最长不重复子串只能从字符 x 所处的位置往后一位开始计算。

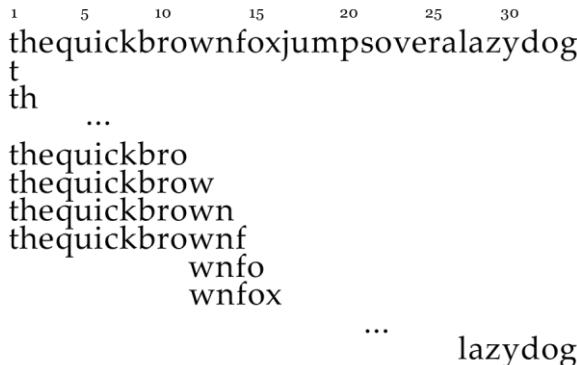


图 11-22 以某个字符作为结尾的最长不重复子串

在此过程中，关键是要知道第 i 个字符在之前的字符串中是否出现以及出现的位置，通过这些信息即可得到以第 i 个字符结尾的最长不重复子串长度。可以通过设立一个位置标记数组来记录某个字符在从前往后的处理过程中最后一次出现的位置，由于题目需要处理的字符集一般为 ASCII 字符集，可以假设需要处理的不同字符为 256 个，即数组大小设为 256。

```
-----11.10.5.cpp-----
// 求给定字符串的最长不重复子串。返回的是第一次遇到的最长不重复子串。
string lswrc(string source) {
    // 字符串长度为 0，则最长不重复子串为空。
    if (source.length() == 0) return "";

    // 设立标记数组，表示某个字符在字符串中的最后一次出现位置，如果未出现则值为 -1。
    vector<int> latest(256);
    fill(latest.begin(), latest.end(), -1);

    // 初始时，最长不重复子串的长度为第一个字符构成的不重复子串，其长度为 1，起始位置为 0。
    int currentMaxLength = 1, maxLength = 1, maxLengthStartAt = 0;
    for (int i = 1; i < source.size(); i++) {
        // 如果第 i 个字符在此前未出现，则第 i 个字符可以附加在之前的最长不重复子串后
        // 构成新的不重复子串，或者该字符出现在以第 i - 1 个字符结尾的最长不重复子串
        // 之前，那么也可以附加在其后构成新的不重复子串。
        if (latest[source[i]] == -1 || latest[source[i]] < (i - currentMaxLength))
            currentMaxLength++;
        // 最长不重复子串的长度需要重新计算。
        else currentMaxLength = i - latest[source[i]];
        // 更新字符的最后一次出现位置。
        latest[source[i]] = i;
        // 通过比较获取最长不重复子串的长度和起始位置。
    }
}
```

```

        if (currentMaxLength > maxLength) {
            maxLength = currentMaxLength;
            maxLengthStartAt = i - currentMaxLength + 1;
        }
    }
    // 截取最长不重复子串。
    return source.substr(maxLengthStartAt, maxLength);
}
//-----11.10.5.cpp-----//

```

11.10.6 最长回文子串

给定某个非空字符串 $X=x_0x_1\dots x_n$, 令字符串中任意两个不同位置的字符下标为 i 和 j , $0 \leq i \leq j \leq n$, 只要下标满足条件 $i+j=n$ 就有 $x_i=x_j$, 那么该字符串为回文字符串¹。例如字符串“ABCCBA”、“acdca”等。如果某个回文字符串 X 是另一字符串 Y 的一个子串, 则称 X 为 Y 的回文子串, 如果在 Y 的所有回文子串中, X 的长度最长, 则称 X 为 Y 的最长回文子串 (Longest Palindrome Substring, LPS)。注意, 最长回文子串是相对于单个字符串来说的, 且给定字符串的最长回文子串可能不唯一。

强化练习: [401 Palindromes^A](#), [10945 Mother Bear^A](#), [11309 Counting Chaos^B](#)。

朴素的方法是从给定字符开始, 向前后两个方向进行比对, 如果相应位置的字符相同则构成回文字符串, 该方法的效率为 $O(n^2)$ 。

```

//-----11.10.6.1.cpp-----//
// 寻找非空字符串中的最长回文子串。返回第一次遇到的最长回文字符的起始位置和长度。
pair<int, int> findLongestPalindrome(string text) {
    int startIndex = 0, maxLength = 0, currentLength, left, right;
    for (int i = 0, length = text.length(); i < length; i++) {
        // 从指定字符向前后两个方向寻找。
        currentLength = 1, left = i - 1, right = i + 1;
        while (left >= 0 && right < length && text[left] == text[right])
            left--, right++, currentLength++;
        if (currentLength > maxLength) {
            startIndex = i - currentLength + 1;
            maxLength = 2 * currentLength - 1;
        }
    }

    // 当连续两个字符相同时, 采用同样的方法向前后比较, 如果不处理这种情况, 会漏掉
    // 形如“qwertyuiooiuytrewq”的回文子串。
    if (i > 0 && text[i] == text[i - 1]) {
        currentLength = 1, left = i - 2, right = i + 1;
        while (left >= 0 && right < length && text[left] == text[right])
            left--, right++, currentLength++;
        if (currentLength > maxLength) {
            startIndex = i - currentLength;
            maxLength = 2 * currentLength;
        }
    }
    return make_pair(startIndex, maxLength);
}

```

¹ 下标从 0 开始, 避免当字符串长度为奇数时出现条件不一致的情形。

```
//-----11.10.6.1.cpp-----//
```

强化练习: 353 Pesky Palindromes^A, 1239 Greatest K-Palindrome Substring^D, 11584 Partitioning by Palindromes^B, 11888 Abnormal 89's^C。

由于回文子串是左右对称的, 如果将原字符串逆转, 则最长回文子串必定是原字符串和逆转后的字符串的某个最长公共子串, 因此可以利用前述求最长公共子串的方法来求最长回文子串, 此种方法的效率仍然为 $O(n^2)$ 。

下面介绍时间复杂度为 $O(n)$ 的 Manacher 算法^[150]。该算法由 Manacher 于 1975 年发现, 最初是为了列出给定字符串从第一个字符开始的所有回文子串, 后经他人予以扩展, 发现可以用来寻找给定字符串的所有最长回文子串^[151]。算法为了克服回文子串长度奇偶不同时需要分别处理的不便之处, 先将原字符串 S 进行预处理, 在 S 中每隔一个字符插入一个“间隔符”(要求原字符串中不能存在该字符, 如原字符串为 “AABCDDCBCDE”, 可选择 ‘#’ 符号, 则经预处理后的字符串变为 “#A#A#B#C#D#D#C#B#C#D#E#”), 变成预处理后的字符串 T 。接下来, 算法定义一个辅助数组 P , 其中的元素 P_i 表示字符串 T 中以第 i 个字符为中心的回文子串的半径, 计算得到的数组 P 具有一个性质, 在原始字符串 S 中, 以指定“位置”为中心的回文子串的长度恰为 P_i 。之所以给“位置”加上引号, 是因为在原始字符串中, 数组 P 的元素所对应的位置可能是两个字符的中间位置, 而并不是刚好对应一个字符。例如, 按照 Manacher 算法对字符串 “AABCDDCBCDE” 进行处理可得到以下结果:

原始字符串 S :	A A B C D D C B C D E
预处理后的字符串 T :	#A#A#B#C#D#D#C#B#C#D#E#
辅助数组 P 元素值:	01210101016101050101010

为了便于理解, 下面先给出 Manacher 算法的实现, 然后再详细予以解释。

```
//-----11.10.6.2.cpp-----//
```

```
// 定义辅助数组 P。
int P[10240] = {};
```

```
// Manacher 算法求字符串中的回文子串。
void manacher(string &line) {
    // 为字符串增加分隔符。
    string modified = {'#'};
    for (int i = 0; i < line.length(); i++)
        modified.push_back(line[i]), modified.push_back('#');

    // center 为当前的“中心点”, rightmost 为当前的“右边界”。
    // 变量 low 和 high 为扩展当前得到的回文子串长度而设置。
    int center = 0, rightmost = 0, low = 0, high = 0;

    // 第一个字符是分隔符, 故从第二个字符开始处理。
    for (int i = 0; i < modified.length(); i++) {
        // 根据右边界和当前字符序号的大小关系来得到数组 P 的值。
        // 这里既是算法的精髓所在, 也是理解算法的难点之处。
        if (rightmost > i) { // 01
            int j = 2 * center - i; // 02
            if (P[j] < (rightmost - i)) { // 03
                P[i] = P[j]; // 04
                high = low = -1; // 05
            }
        } // 06
    }
}
```

```

        else {                                // 07
            P[i] = rightmost - i;             // 08
            high = rightmost + 1;             // 09
            low = 2 * i - high;              // 10
        }                                     // 11
    }                                     // 12
    else {                                // 13
        P[i] = 0;                           // 14
        low = i - 1;                        // 15
        high = i + 1;                       // 16
    }                                     // 17

    // 扩展当前得到的回文子串。
    while (low >= 0 && high < modified.length() &&
           modified[low] == modified[high]) {
        P[i]++;
        low--;
        high++;
    }

    // 根据得到的结果更新“中心点”和“右边界”。
    if ((i + P[i]) > rightmost) {
        center = i;
        rightmost = i + P[i];
    }
}
//-----11.10.6.2.cpp-----//

```

虽然代码不是很多，但是理解起来却不是那么容易。下面重点解释标注了序号的代码。

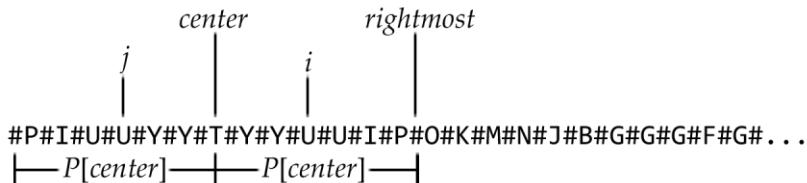


图 11-23 计算数组 P 的过程中的某个中间步骤

图 11-23 描述了计算数组 P 的过程中的某个中间步骤，以下为各个变量的含义。

$center$: 与当前右边界对应的中心位置；

$P[center]$: 以 $center$ 所在字符为中心的最长回文子串的半径；

$rightmost$: 当前的右边界位置， $rightmost = i + P[center]$ ；

i : 当前处理的字符位置；

j : 位置 i 以 $center$ 为中心的对称点，由于 $center - j = i - center$ ，故 $j = 2 \times center - i$ 。

当右边界大于当前处理的字符位置时，即 i 落在 $center$ 和 $rightmost$ 之间时，先查看 i 关于 $center$ 的对称点 j 为最长回文子串的半径 $P[j]$ ，因为对回文字符串以其中心位置进行左右翻转后仍为回文字符串，则以字符 i 为中心的回文子串半径 $P[i]$ 在当前必定等于以字符 j 为中心的回文子串半径 $P[j]$ ，若有 $P[j] < rightmost - i$ ，则以 i 为中心的最长回文子串半径 $P[i]$ 必定落在 $rightmost$ 的左侧，且不必继续对得到的最长回文子串进行扩展，因为这是能够得到的最长回文子串。若 $P[j] \geq rightmost - i$ （实际上， $P[j]$ 最大可能与

$rightmost - i$ 相等, 而不会大于它, 如果大于它, 则以 $center$ 为中心的回文子串半径必将增大, 从而导致与之前得到的 $P[center]$ 是以 $center$ 为中心的回文子串最大半径的结果相矛盾), 表明以 j 为中心的最长回文子串半径不小于 i 与右边界 $rightmost$ 之间的距离, 则 i 到右边界 $rightmost$ 之间的字符必定属于以 i 为中心的回文子串的一部分, 但是由于大于右边界 $rightmost$ 的字符尚未进行匹配, 尚不能断定它们是以 i 为中心的回文子串的组成部分, 故当前以 i 为中心的回文子串半径暂定为 $rightmost - i$, 待后续对当前得到的回文子串进行进一步匹配, 检查是否能够扩展其半径。

当右边界小于当前处理的字符位置时, 无法获得关于当前字符回文子串半径的有效信息, 只有以该字符为中心, 前后逐个匹配以查找最大的回文子串半径。

该算法的巧妙之处在于充分利用了当前得到的最长回文子串半径这个信息, 减少了后续求数组 P 的计算量, 也就是减少了重复匹配的计算量, 从而提高了效率。

注意到在确定了右边界后, 只要当前字符位置 i 小于右边界, 则 $P[i]$ 至少为 $P[2 \times center - 1]$, 而之所以使用变量 low 和 $high$ 来扩展回文子串, 是由于在前期的预处理过程中, 原始字符串的首尾增加的是相同的分隔符, 在扩展过程中, 如果不对字符数组下标进行检查, 可能会发生下标引用越界的情况而引起运行时错误。通过适当改变预处理过程, 可以消除变量 low 和 $high$ 的使用, 具体方法是预处理时在字符串首尾添加与分隔符不同的字符, 假如分隔符设置为 ‘#’, 则字符串首部添加 ‘\$’, 字符串尾部添加 ‘|’, 在扩展回文子串的过程中, 不必再检测下标引用是否越界, 因为当扩展到字符串的首尾边界位置时, 匹配条件必定不再满足, 从而使得扩展过程终止。

```
//-----11.10.6.3.cpp-----
int P[10240] = {};

void manacher(string &line) {
    string modified = {'$'};
    for (int i = 0; i < line.length(); i++)
        modified.push_back(line[i]), modified.push_back('#');
    modified.back() = '|';

    int center = 0, rightmost = 0;
    for (int i = 0; i < modified.size(); i++) {
        int x = 2 * center - i, y = rightmost - i;
        P[i] = (rightmost > i) ? (P[x] < y ? P[x] : y) : 1;
        while (modified[i - P[i]] == modified[i + P[i]]) P[i]++;
        if (i + P[i] > rightmost) {
            center = i;
            rightmost = i + P[i] - 1;
        }
    }
}
//-----11.10.6.3.cpp-----
```

需要注意, 在后一种实现中, 得到的数组 P 的元素值与第一种实现方法得到的数组 P 的元素值含义稍有差异。仍以字符串 “**AABCDDCBCDE**” 为例, 后一种实现中数组 P 的元素为:

原始字符串 S :	A A B C D D C B C D E
预处理后的字符串 T :	\$A#A#B#C#D#D#C#B#C#D#E
辅助数组 P 元素值:	11221212127212161212111

数组 P 中的值是相对于经过预处理之后的字符串 T 而获得, 不是相对于原始字符串 S 获得, 亦即将中心分

隔符也考虑在半径之内，而在前一种实现中，中心分隔符未计入半径，因此在求原字符串以指定位置字符为中心的最长回文子串长度时需要进行适当地变换处理。

强化练习：257 Palinwords^D，689 Napoleon's Grumble^D。

扩展练习：10617 Again Palindromes^B，11475 Extend to Palindromes^B。

11.10.7 最大连续子序列和（积）

给定一个序列 $X = \langle x_1, x_2, \dots, x_m \rangle$ ，若称另一个序列 $Z = \langle z_1, z_2, \dots, z_k \rangle$ 是 X 的一个连续子序列，则存在 X 的一个严格递增的下标序列 $\langle i_1, i_2, \dots, i_k \rangle$ ，使得对所有的 $j = 1, 2, \dots, k$ ，均有 $x_{i_j} = z_j$ ，且下标序列满足 $i_k = i_{k-1} + 1$ 。例如， $Z = \langle B, C, B, D \rangle$ 是 $X = \langle A, B, C, B, D, A, B \rangle$ 的一个连续子序列，相应的 X 下标序列为 $\langle 2, 3, 4, 5 \rangle$ ，而序列 $Z' = \langle B, C, A, B \rangle$ 是 X 的子序列，但不是连续子序列。序列 X 中的单个元素可以认为是长度为 1 的连续子序列。

强化练习：10324 Zeros and Ones^A。

如果序列中的元素均为整数（可以为负数），则可以对连续子序列进行求和操作，那么如何确定连续求和的最大值（Maximum Contiguous Subsequence Sum，MCSS）呢？

最直接的是使用暴力法，穷举所有可能的连续子序列，求和然后取最大值（以下代码假设给定的整数之和均在 `int` 数据类型所能表示的范围内），时间复杂度为 $O(n^3)$ 。

```
//+++++11.10.7.cpp+++++/
int maximumSum(int data[], int n) {
    int maxSum = numeric_limits<int>::min();
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            int sum = 0;
            for (int k = i; k <= j; k++)
                sum += data[k];
            maxSum = max(maxSum, sum);
        }
    }
    return maxSum;
}
```

如果给定的序列较长，使用上述方法解题肯定会超时。尽管可以对上述代码进行优化，使时间复杂度降低到 $O(n^2)$ ，但是对于时间要求较高的竞赛环境仍不足够（将 $O(n^3)$ 的算法优化成 $O(n^2)$ 的算法请读者作为练习加以完成）。

考虑到 MCSS 要么出现在序列的左半部分，要么出现在序列的右半部分，要么横跨左右两个部分，则有基于分治法的时间复杂度为 $O(n \log n)$ 的算法。

```
int maximumSum(int data[], int left, int right) {
    // 递归出口。
    if (left > right) return 0;
    if (left == right) return data[left];

    // 分治。
    int middle = (left + right) / 2;

    // 求最大连续子序列和的左半部分。
    int leftMax = numeric_limits<int>::min(), leftSum = 0;
    for (int i = middle; i >= left; i--) {
        leftSum += data[i];
        leftMax = max(leftMax, leftSum);
    }

    // 求最大连续子序列和的右半部分。
    int rightMax = numeric_limits<int>::min(), rightSum = 0;
    for (int i = middle + 1; i <= right; i++) {
        rightSum += data[i];
        rightMax = max(rightMax, rightSum);
    }

    // 求横跨左右两个部分的最大值。
    int crossSum = leftSum + rightSum;
    if (leftMax + rightMax > crossSum)
        return max(leftMax + rightMax, crossSum);
    else
        return max(leftMax, rightMax);
}
```

```

    leftMax = max(leftMax, leftSum);
}

// 求最大连续子序列和的左半部分。
int rightMax = numeric_limits<int>::min(), rightSum = 0;
for (int i = middle + 1; i <= right; i++) {
    rightSum += data[i];
    rightMax = max(rightMax, rightSum);
}

// 递归求解。
return max(leftMax + rightMax,
    max(maximumSum(data, left, middle), maximumSum(data, middle + 1, right)));
}

```

如果考虑到 MCSS 必定以给定序列中的某个元素结尾，则有时间复杂度为 $O(n)$ 的动态规划解法：设序列共有 n 个元素， $E(i)$ 表示以第 i 个元素作为结尾元素的 MCSS， $M(i)$ 表示至第 i 个元素为止，目前能够得到的 MCSS，则有递推关系

$$E(i) = \max\{S_i, S_i + E(i-1)\}, \quad 2 \leq i \leq n$$

$$M(i) = \max\{E(i), M(i-1)\}, \quad 2 \leq i \leq n$$

即对于第 i 个元素来说, 当前的 MCSS 要么是第 i 个元素本身 (它构成了长度为 1 的连续子序列), 要么是以第 $i-1$ 个元素结尾的 MCSS 加上第 i 个元素, 要么是前 $i-1$ 个元素中的 MCSS。由于只需要取最大值的结果, 可以不需保存 $E(i)$ 及 $M(i)$ 之前的计算结果。若需获取 MCSS 的起始和结束位置, 则需要另外使用辅助变量进行记录。

```
int maximumSum(int data[], int n) {
    int maxSum = data[0], currentSum = data[0];
    for (int i = 1; i < n; i++) {
        currentSum = max(data[i], currentSum + data[i]);
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

对于浮点数来说，同样可以应用上述给出的动态规划算法。与 MCSS 类似的还有最大连续子序列积，即将求和操作改为求乘积操作后所能得到的最大值。可以参考求 MCSS 的动态规划解法，从而得到该问题时间复杂度为 $O(n)$ 的算法。以下给出实现代码，其递推关系请读者自行思考并进行推导。

```
long long maximumProduct(long long data[], int n) {
    long long product, currentMax, currentMin, nextMax, nextMin;
    product = currentMax = currentMin = data[0];
    for (int i = 1; i < n; i++) {
        nextMax = currentMax * data[i], nextMin = currentMin * data[i];
        currentMax = max(data[i], max(nextMax, nextMin));
        currentMin = min(data[i], min(nextMax, nextMin));
        product = max(product, currentMax);
    }
    return product;
}
```

强化练习: 507 Jill Rides Again^A, 10684 The Jackpots^A, 11059 Maximum Product^A, 12640 Largest Sum

Game^C。

扩展练习: [787 Maximum Sub-Sequence Product^B](#), [10755 Garbage Heap^B](#), [11078 Open Credit System^A](#)。

11.11 贪心算法

贪心算法 (greedy algorithm) 和动态规划算法关系密切。在动态规划算法过程中, 需要根据递推关系不断进行选择, 在选择时需要根据最终问题的解来决定最佳选择, 但是对于某些问题来说, 可以采取一种更加简洁的选择策略——只选择当前最佳的, 即每次通过局部的最优解来构造一个全局的最优解。

区间调度 (interval scheduling) 是一个可以使用贪心算法解决的典型问题。区间调度是指给定 n 个区间 $[a_i, b_i]$, $1 \leq i \leq n$, $a_i < b_i$, 要求确定区间的一个子集, 该子集所包含的区间互不重叠且区间的数量最大。从动态规划的角度思考, 表示状态至少需要两个域: 一个域表示互不重叠区间的最大数量, 另外一个域表示位于最右侧区间的右端点位置。令 $dp[i].size$ 表示前 i 个区间互不重叠区间的最大数量, $dp[i].rightMost$ 表示从前 i 个区间中选取的具有最大数量互不重叠区间的子集中位于最右侧区间的右端点位置, $left[i]$ 表示第 i 个区间的左侧端点, $right[i]$ 表示第 i 个区间的右侧端点。为了便于区间的选择, 首先将所有区间按照左侧端点升序排序, 如果左侧端点相同, 则按右侧端点升序排列。假设已经确定了状态 $dp[0], dp[1], \dots, dp[i-1]$ 的值, 对于第 i 个区间来说, 需要检查其是否能够附加在之前得到的子集之后形成更大的子集, 即第 i 个区间的左侧端点要大于或等于 $dp[j].rightMost$, 此时有

$$dp[i].size = \max\{dp[j].size + 1\}, 0 \leq j < i, left[i] \geq dp[j].rightMost$$

在状态转移过程中, 如果对于某个 j 来说, $dp[j].size + 1$ 要优于 $dp[i].size$, 不仅要将 $dp[i].size$ 更新为 $dp[j].size + 1$, 还需将 $dp[i].rightMost$ 更新为 $right[i]$, 如果 $dp[j].size + 1 = dp[i].size$, 则需检查 $right[i]$ 是否优小于 $dp[i].rightMost$, 如果小于 $dp[i].rightMost$, 应该将 $dp[i].rightMost$ 更新为 $right[i]$, 因为具有较小的右侧端点, 则在后续过程中更有可能进行区间的扩展。按照上述递推关系式进行计算, 可以得到时间复杂度为 $O(n^2)$ 的算法, 但通过进一步地仔细分析, 可以得到时间复杂度为 $O(n)$ 的贪心算法。观察递推关系式, 在区间的选取过程中, 对于两个待选区间 k 和 l , 如果区间 k 附加在 $dp[j]$ 之后能够获得更小的右侧端点则区间 k 要优于 l 。那么可以使用贪心选择策略, 将所有区间按照右侧端点升序排列, 在选择第一个具有最小的右侧端点的区间后, 后续的区间如果与前一个区间不发生重叠则可将其附加在前一区间后构成更长的区间子集, 可以证明, 按照贪心策略得到的区间子集必定是最优的^I。

求无向图中最短路径的 Dijkstra 算法以及求无向图最小生成树的 Prim 算法、Kruskal 算法均为贪心算法。存在贪心算法的问题一定可以使用动态规划予以解决, 但是反过来就不一定成立。这是因为在贪心算法中, 将动态规划所需要进行的选择简化成了一次性选择, 排除了贪心算法选择之外的其他选择, 因此简化了解决问题的过程。一般来说, 证明贪心算法的正确性往往不是那么直接和容易, 在解题时使用贪心算法大多来源于平时解题的经验和直觉, 总之需要“大胆假设, 小心求证”, 可以事先手工设计若干测试数据来验证贪心算法的正确性。

410 Station Balance^B (空间站平衡)

在国际空间站的实验室里有许多离心机。每台离心机的离心腔都包含 C 个腔室, 每个腔室最多可放置 2

^I 参见: Jon Kleinberg 和 Éva Tardos 所著《Algorithm Design (影印版)》, 北京: 人民邮电出版社, 2019, 第 118—121 页。

个样本。现在要求你编写一个程序对 S 个样本做出安排，在将这些样本放入离心机后，使得每个腔室包含的样本数不超过限制，同时使得以下表示离心机“不平衡度”的表达式的值最小，即

$$IMBALANCE = \sum_{i=1}^C |CM_i - AM|$$

其中 CM_i 表示放置在离心机腔室 i 的样本质量之和； AM 表示离心机腔室所有样本的平均质量，该数值由所有样本的质量之和除以腔室的数量 C 得到。

输入

输入包含多组测试数据。每组测试数据的第一行包含两个整数，第一个整数 C ($1 \leq C \leq 5$) 表示离心机所包含的腔室数量，第二个整数 S ($1 \leq S \leq 2C$) 表示该组数据中样本的数量。每组测试数据的第二行包含 S 个整数，这 S 个整数表示样本各自的质量。每个样本的质量数值在 1 到 1000 之间，数值以若干个空格分隔。

输出

对于每组测试数据，先输出测试数据组的编号（从 1 开始），按照格式“**Set #X**”输出，其中 ‘X’ 为数据组的编号。接下来输出 C 行，每行的格式及意义如下：第 1 列输出腔室的编号，第 2 列输出一个冒号，分配给该腔室的样本的质量在第 4 列开始输出，样本质量数值以一个空格分隔。接着输出一行，格式为“**IMBALANCE = X**”，其中 ‘X’ 表示能够得到的最小不平衡度，精确到小数点后 5 位。在每组输出的最后打印一个空行。

样例输入

```
2 3
6 3 8
```

样例输出

```
Set #1
0: 6 3
1: 8
IMBALANCE = 1.00000
```

分析

题目所给出的数据范围较小，可以使用穷尽搜索来计算所有可能的排列，然后取其中最小不平衡度的组合。更为简便的方法是采用如下策略：如果样本量 S 小于 $2C$ ，则添加质量为 0 的样本，直到样本数量达到 $2C$ ，然后对样本按质量大小排序，得到

$$m_1, m_2, m_3, \dots, m_{2n-2}, m_{2n-1}, m_{2n}$$

接着按照对称原则将首尾两个样本进行组合放置到同一个腔室中，即将 m_1 和 m_{2n} , m_2 和 m_{2n-1} , \dots , m_n 和 m_{n+1} 分别组合放置在一个腔室，可以证明，最终得到的组合一定是具有最小不平衡度的组合。

强化练习：[1062 Containers](#)^B, [10020 Minimal Coverage](#)^A, [10276 Hanoi Tower Troubles Again](#)^A, [10440 Ferry Loading II](#)^B, [10656 Maximum Sum \(II\)](#)^A, [10785 The Mad Numerologist](#)^B, [10821 Constructing BST](#)^C, [11100 The Trip 2007](#)^B, [11157 Dynamic Frog](#)^B, [11292 Dragon of Loowater](#)^A, [11330 Andy's Shoes](#)^D, [11389 The Bus Driver Problem](#)^A, [11583* Alien DNA](#)^D, [11900 Boiled Eggs](#)^A, [12405 Scarecrow](#)^A, [12516 Cinema-Cola](#)^D, [12834 Extreme Terror](#)^D, [12861 Help Cupid](#)^D, [13031 Geek Power Inc.](#)^D, [13054 Hippo Circus](#)^D, [13082 High School Assembly](#)^D。

扩展练习：[456 Robotic Stacker](#)^E, [668 Parliament](#)^C, [1153 Keep the Customer Satisfied](#)^D, [10714 Ants](#)^A, [11230 Annoying Painting Tool](#)^D, [11368 Nested Dolls](#)^D, [11890 Calculus Simplified](#)^D。

11.11.1 部分背包问题

部分背包问题是指如下的问题：给定一个容量为 C 的背包，有 n 个物品，每个物品都有容量 C_i 和价值 P_i ， $1 \leq i \leq n$ ，物品可以拆分且可取全部或一部分放入背包，问如何选取放入背包的物品，使得背包内的物品容量之和不超过 C 且价值之和最大。

与本章前述的 01 背包、完全背包、多重背包问题不同，此处给出的条件是物品可以进行拆分，只放入一种物品的一部分。例如，使用背包来装入金粉而不是金币的情形。贪心算法的策略很简单，将所有物品按其价值和容量的比值进行排序，即单位容量的物品具有的价值从大到小进行排列，每次在选择时，总是选择单位价值最大的物品放入背包，直到背包被填满或者该种物品已经全部放入背包，若是第二种情况，则继续选择下一种具有最大单位价值的物品放入背包。

强化练习：[12955 Factorial^D](#)。

11.11.2 纸币找零问题

动态规划

每个国家在设计纸币的面值时，一方面会使得面值的种类尽量少，同时又兼顾找零的便利性。如人民币有 1 角、2 角、5 角、1 元、2 元、5 元等多种面值，这样在找 4 元 3 角的零钱时，可以找 4 张 1 元，3 张 1 角，总共 7 张纸币。那么如何在纸币的面值限定的情况下，使得找零时纸币张数最少呢？如在上例中，具有最少纸币张数的找零方案是找 2 张 2 元，1 张 2 角，1 张 1 角，总共 4 张纸币。

最少找零问题可以通过动态规划予以解决。给定一组面值 $D = \langle d_1, d_2, \dots, d_n \rangle$ ，面值按递增排列（不是必须的，只是为了描述问题方便）， $d_i < d_{i+1} (1 \leq i < n)$ ， d_1 是该纸币系统的“最小单元”，即对于任意零钱 X ，都可以通过有限个 d_1 进行找零（如人民币中的 1 分面值），否则将出现某些特定零钱无法找零的情况（如果人民币不存在 1 分的零钱，则找 1 分零钱时存在困难）。设找零钱 X 的最少张数为 $C(X)$ ，若要找的零钱数为 M ，则所求为 $C(M)$ ，因为最后找的一张零钱必定是给定面值中的一种，那么只要知道了 $C(M - d_1)$ ， $C(M - d_2)$ ， \dots ， $C(M - d_i)$ ，其最小值再加 1 即为 $C(M)$ ，而只要知道了 $C(M - d_1 - d_1)$ ， $C(M - d_1 - d_2)$ ， \dots ， $C(M - d_1 - d_i)$ ，其最小值再加 1 即为 $C(M - d_1)$ ……。故此问题的递推关系为

$$C(M) = \min\{C(M - d_i) + 1\} \quad 1 \leq i \leq n, \quad C(0) = 0$$

很显然，在初始化的时候，找 0 元零钱需要的最少纸币张数为 0，因此有 $C(0) = 0$ 。

Automatic exChange Machine (自动兑换机)^I

A 城市的地铁公司决定采取一项新措施——勿需购票，投币上车。有传闻说此举是为了减少乘客购票的排队时间。地铁运营商找到了本市计算机协会（Association for Computing Machinery, ACM）旗下的自动收款机（Automated Checkout Machine, ACM）公司，要求开发一款自动兑换机（Automatic exChange Machine, ACM）来满足乘客的需求。他们雇用你来担任首席程序员为此机器编写程序。自动兑换机内部存放有各种面值的硬币，当乘客将纸币放入机器时，机器会自动根据当前可用的硬币面值将乘客的纸币兑换成等值的硬币。当然，乘客不愿意口袋里面装着一大堆硬币去挤地铁，因此兑换成的硬币数量越少越好。如果现有的硬币面值无法完成兑换要求，应该输出一行信息，提示乘客需要寻求人工窗口的服务。

输入

^I 这是由作者本人拟制的题目，收录于洛谷主题库，题号为 P7396。

输入包含多组测试数据。第一行一个整数 T ($1 \leq T \leq 400$), 表示数据组数。每组测试数据一行, 第一个为整数 c ($1 \leq c \leq 100$), 表示硬币面值的种类, 接下来是 c 个整数, 每个整数 d_i ($1 \leq i \leq c$, $1 \leq d_i \leq 400$) 表示一种硬币的面值, 以美分为单位, 最后是一个实数 m ($0 < m \leq 1000$), 表示乘客需要兑换的纸币的总面值, 以美元为单位。

输出

对于每行输入均输出一行, 如果不存在兑换方案, 输出 **No solution.**, 否则, 按以下格式输出具有最少硬币数量的兑换方案: 首先输出方案中硬币的总个数, 然后一个空格, 接着按照样例输出的格式打印兑换序列, 即依照面值从小到大的顺序, 将方案使用的各个面值及其对应的硬币个数予以输出。如果存在多种具有最少硬币数量的兑换方案, 输出字典序 (即按 ASCII 序) 最小的兑换方案。

样例输入

```
6
6 1 2 5 10 20 50 25.31
5 1 2 2 5 10 0.18
5 1 2 10 9 5 0.18
6 2 5 10 20 50 100 0.03
11 173 151 214 211 238 167 385 179 5 235 112 46.1
13 95 180 285 205 164 82 122 52 362 260 166 364 189 6.55
```

样例输出

```
53 1*1+10*1+20*1+50*50
4 1*1+2*1+5*1+10*1
2 9*2
No solution.
14 112*2+151*1+385*11
4 122*1+164*1+180*1+189*1
```

分析

此题可用应用前述介绍的动态规划算法进行解决。由于需输出最少硬币方案的具体构成, 因此在自底向上进行动态规划时需要记录每一次选择时的相关参数, 以便重建选择路径时使用。

参考代码

```
-----11.11.2.cpp-----
// 定义一个最大值, 将数组元素初始化为此值, 表示某些数量的纸币无法完成兑换。
const int INF = 0x3f3f3f3f;

// n 为面值的种类数量;
// denom 数组存储硬币面值;
// coins 数组存储不同纸币面值所对应的最少兑换硬币数量;
// parent 数组存储各纸币数量选择路径上的“父”兑换纸币数量;
// idx 数组存储的是选择的面值在 denom 数组中的序号;
// cnt 数组用于在程序最后输出时统计各面值出现的次数。
int n;
int denom[110];
int coins[10010], parent[10010], idx[10010], cnt[110];

// 利用递归来重建选择路径。参数 money 表示纸币的数量。
```

```

void findPath(int money) {
    if (money > 0) {
        cnt[idx[money]]++;
        findPath(parent[money]);
    }
}

// 确定是否存在指定的兑换方案。
void findMiniumCoins(int money) {
    // 初始化相关数组元素。
    fill(coins, coins + 10010, INF);
    fill(cnt, cnt + 110, 0);

    // 设置初始值，纸币数量为 0 时最少硬币数量为 0。然后自底向上进行动态规划选择。
    coins[0] = 0;
    for (int m = 1; m <= money; m++) {
        int minCoins = INF, minIdx = INF;
        // 注意选择条件：当前纸币数量要大于硬币的面值（才可使用此面值的硬币）,
        // 而且从纸币数量减去某个面值时的兑换方案必须存在，而且兑换方案的硬币
        // 数量加一后比当前兑换方案硬币数量要少。
        for (int d = 0; d < n; d++) {
            if (m >= denom[d] && coins[m - denom[d]] != INF &&
                minCoins > (coins[m - denom[d]] + 1))
                minCoins = coins[m - denom[d]] + 1, minIdx = d;
        }
        if (minIdx != INF) {
            coins[m] = minCoins;
            parent[m] = m - denom[minIdx];
            idx[m] = minIdx;
        }
    }
}

// 根据结果进行输出。
if (coins[money] == INF) cout << "No solution.\n";
else {
    // 输出总硬币数量。
    cout << coins[money];
    // 重建选择路径。
    findPath(money);
    // 输出各种面值及其对应硬币数量。
    int plusPrinted = 0;
    for (int i = 0; i < n; i++)
        if (cnt[i] > 0) {
            cout << (plusPrinted++ ? "+" : " ");
            cout << denom[i] << "*" << cnt[i];
        }
    cout << '\n';
}

// 读入面值及纸币数量，将纸币数量转换为整数以便于处理。
int main(int argc, char *argv[]) {
    double money;
    while (cin >> n, n) {
        for (int i = 0; i < n; i++) cin >> denom[i];
        // 去除重复的面值。
}

```

```

        sort(denom, denom + n);
        n = unique(denom, denom + n) - denom;
        cin >> money;
        findMinumCoins((int)(money * 100.0 + 0.5));
    }
    return 0;
}
//-----11.11.2.cpp-----//

```

强化练习: 147 Dollars^C, 266 Stamping Out Stamps^E, 11407 Squares^B。

贪心算法

对于某些具有特殊设计的面值系统, 可以简化动态规划的某些步骤, 采取更简单直接的选择策略。如人民币面值系统的找零, 应用贪心算法能产生最优解, 此时可以通过每次都选择尽可能大的找零面值来减少硬币的数量。可以证明, 当可找零的硬币面值是整数 c 的幂, 即 $c^0, c^1, \dots, c^k, c > 1, k \geq 1$, 贪心算法总是可以产生一个最优解。更一般的, 假设有硬币面值序列 $D = \langle d_1, d_2, \dots, d_n \rangle$, 其中 $d_i (1 \leq i \leq n)$ 为整数, 面值按递增排列, 且 $d_1 = 1$, 如果有 $d_i/d_{i-1} \geq 2 (1 < i \leq n)$, 则贪心算法总是可以产生一个最优解。因为在任何一次动态规划的选择步骤中, 如果当前可选 d_i 而未选, 则 d_i 这部分面值只能由更小面值的硬币凑成, 但是根据前述限定的 $d_i/d_{i-1} \geq 2 (1 < i \leq n)$ 关系, 至少需要 2 枚以上的硬币来凑成 d_i , 所以任何其他的选择都将导致硬币总数量比选择 d_i 大, 故总是选择较大的 d_i 的贪心算法可以得到最优解。

166 Making Change^A (找零)

给定数量 (几乎) 不限的硬币, 我们知道将一定数量的纸币兑换成硬币有多种方式。在购买物品付账后找零的过程中, 产生了一个更令人感兴趣的问题。在当今钱包容量普遍有限的情况下, 我们为购物付款时凑硬币的方法大受限制——假如我们能够先行一步将款项凑齐的话, 不过我们要说的是另外一个问题。

我们关心的问题是在店主的硬币数量足够的情况下, 如何在交易过程中将易手的硬币数量最小化 (新西兰货币系统中硬币面值有 5 分, 10 分, 20 分, 50 分, 1 元, 2 元)。假如我们需要付 55 分钱的购物款, 但手上没有 50 分的硬币, 那么我们可以用 2×20 分 + 10 分 + 5 分的方式组成 55 分钱来付款, 易手的硬币数量一共是 4 枚。如果我们给付 1 元, 那么店主找零为 45 分 (2×20 分 + 5 分), 易手的硬币数量也是 4 枚, 但如果我们付 1.05 元 (1 元 + 5 分), 店主找零 50 分, 则易手的硬币数量是 3 枚。

编写程序读入可用的硬币数量以及款项, 确定最小的易手硬币数量。

输入

输入由多行组成, 每行设定了不同的情形。每行的前 6 个整数表示你可用的不同面值的硬币数量, 按前述给出的货币系统中面值的组成排列, 接着是一个实数, 表示交易所涉及的款项, 此款项总是小于 5.00 元。输入以 6 个零结束 (0 0 0 0 0 0)。你手上的硬币数量总是能够让你支付所需款项, 而且款项总是 5 分钱的整数倍。

输出

输出包含多行, 每行对应输入的一种情形。每行输出由表示最小易手硬币数量的数字组成, 以宽度 3 右对齐输出。

样例输入

```
2 4 2 2 1 0  0.95
2 4 2 0 1 0  0.55
```

样例输出

```
2
3
```

0 0 0 0 0 0

分析

此题正向思考似乎无从下手，不妨使用反向思维。顾客的硬币数量有限，那么所能凑的钱有一个上限值 M ，顾客付钱后，如果所付钱数大于购物款 G ，店主会找相应的零钱 C 给顾客，店主有足够硬币，总是可以找零，对于店主来说，可以找 $0 \sim (M - G)$ 之间的任意零钱，而对顾客来说，不一定能够凑齐 $G \sim M$ 之间的所有零钱，那么可以从 0 分开始，每次将店主需要返还给顾客的零钱数 C 增加 1 分，让店主找零，让顾客凑钱，寻找两者硬币数之和的最小值。由于顾客硬币数量有限制，在某些情况下可能无法凑出指定的钱。本题中的面值设定满足使用贪心算法找最少硬币的条件，可以直接使用贪心算法而不必求助于动态规划。

强化练习：[10670 Work Reduction^B](#)。

11.11.3 硬币兑换问题

找零问题是在限定面值的情况下，要求所找的零钱数量最少。硬币兑换问题（coin change）所求的是在限定面值的情况下，能够得到的不同找零方案总数。如果一个面值系统有 1 分，5 分，10 分的硬币，那么将 17 分钱兑换成这 3 种面值的硬币共有 6 种不同的兑换方法（分别为：{全为 1 分}，{12 个 1 分，1 个 5 分}，{7 个 1 分，2 个 5 分}，{2 分 1 分，3 个 5 分}，{7 个 1 分，1 个 10 分}，{2 个 1 分，1 个 5 分，1 个 10 分}）。解决这类问题的方法仍然是动态规划。

11137 Ingenuous Cubrency^A（巧妙的立方币）

给定一个立方币面值系统，其中的硬币面值都是立方数，从 1^3 至 21^3 ，即 1, 8, 27, ..., 9261。计算指定数额的钱币有多少种不同的兑换方法。例如，对于数额共 21 的纸币来说，共有 3 种不同的兑换方法，第一种是使用 21 枚面值为 1 的硬币，第二种是使用一枚面值为 8 的硬币和 13 枚面值为 1 的硬币，第三种是使用两枚面值为 8 的硬币和 5 枚面值为 1 的硬币。

输入

输入包含多行，每行包含一个表示钱币数额的整数，其大小不超过 10000。

输出

对于每行输入输出一行，包含一个整数，表示对于指定的钱币数额有多少种不同的兑换方法。

样例输入

10
21
77
9999

样例输出

2
3
22
440022018293

分析

此类兑换问题和完全背包问题类似，可以沿用其解题思路来解决本问题。设立一个二维数组 W ，令 $W[i][j]$ 表示在只有前 i 种硬币可供兑换的情况下，总额为 j 的钱币的不同兑换方法数， $coins[i]$ 表示第 i 种硬币的面值。假设已经确定只有前 $i-1$ 种硬币可供兑换的情况下，总额为 j 的钱币的不同兑换方法数 $W[i-1][j]$ ，那么，当条件进一步“松弛”时，即在前 i 种硬币可用的情况下，总额为 j 的钱币具有多少种不同兑换方法呢？不难得出，此时的不同兑换方法数

$$W[i][j] = W[i-1][j] + W[i-1][j - coins[i]]$$

为什么是这样呢？理解此递推关系的关键在于包含第 i 种硬币的兑换方法和不包含第 i 种硬币的兑换方法是截然不同的，这一点是很自然的。既然两者是不同的，那么总的方法数就是以下两种方法数的总和：(1) 不包括第 i 种硬币的情况下，将数额为 j 的纸币兑换成硬币的方法数；(2) 包括第 i 种硬币，将数额为 $j - coins[i]$ 的纸币兑换成硬币的方法数，也就是说，一种只用前 $i-1$ 种硬币将数额为 $j - coins[i]$ 的纸币兑换为硬币的方法，只要再加上一枚面值为 $coins[i]$ 的硬币，就能得到数额为 j 的纸币兑换方法。同样的，类似于背包问题优化空间使用的做法，可以将递推关系改写成

$$W[j] = W[j] + W[j - coins[i]]$$

从而使得只需要一维数组来表示最终结果。需要注意，本题中的结果范围较大，需要使用 `long long int` 数据类型来存储不同的兑换方法数。

强化练习：[357 Let Me Count The Ways^A](#), [674 Coin Change^A](#), [10306 e-Coins^A](#), [10313 Pay the Price^B](#), [10465 Homer Simpson^A](#), [11264 Coin Collector^B](#), [11517 Exact Change^A](#)。

扩展练习：[1213 Sum of Different Primes^B](#), [11259* Coin Changing Again^D](#)。

11.11.4 霍夫曼编码

在传输数据时，为了减少数据的传输量，可能需要将数据进行压缩。霍夫曼编码（Huffman encoding）是通过对数据进行重新编码达到压缩数据的一种方式，属于可变长编码（variable length coding）。

霍夫曼编码的具体步骤如下：先统计待编码文件中各字符的出现频度，接着使用贪心策略，选择频度最小的两个字符，分别为其分配编码 0 和 1，然后将两个字符的频度相加作为一个组合字符放入优先队列中，接下来继续使用贪心策略，选择频度最小的两个字符，分别为其分配编码 0 和 1，继续此过程，直到最后只剩下一个字符，最后，将编码按逆序进行输出即为各字符的最终编码。由于每次均需要选择频度最小的两个字符，使用最小优先队列来实现霍夫曼编码非常方便。

给定如下的 ASCII 文件（假定每个字符使用一个字节表示）：

ABCDEABCCDEAEEEAAAABBCCCCDDAAAAAAACDAAAACDDAACCDDDDDD

各字符的频度为： $A(22)$, $B(5)$, $C(14)$, $D(14)$, $E(6)$ 。如果使用定长编码，需要 3 位编码来表示，一种可行的编码方案为： $A(000)$, $B(001)$, $C(010)$, $D(011)$, $E(100)$ ，编码后文件长度为 183 位，平均码长为 3。如果使用霍夫曼编码，编码过程如下：

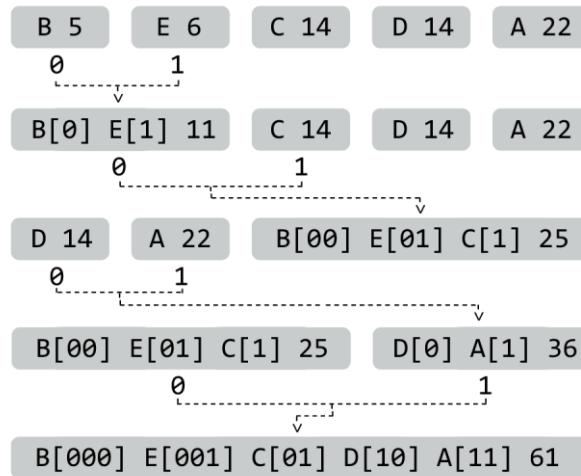


图 11-24 霍夫曼编码过程

最后编码方案为: $A(11)$, $B(000)$, $C(01)$, $D(10)$, $E(001)$, 编码后文件为 133 位, 平均码长为 $133/61 \approx 2.18$ 。

下面给出霍夫曼编码的一种解题用实现。

```
-----11.11.4.cpp-----//
// 定义结构表示字符, 频度, 编码。
struct letter {
    char ascii;
    int frequency;
    string code;
    bool operator<(letter x) const { return ascii < x.ascii; }
};

// 定义一个符号结构体, 方便编码的实现。
struct symbol {
    int frequency;
    vector<letter> letters;
    bool operator<(symbol x) const { return frequency > x.frequency; }
};

void huffman(string line) {
    // 统计各个字符出现的次数。
    map<char, int> counter;
    for (int i = 0; i < line.length(); i++)
        counter[line[i]]++;
    // 将字符放入最小优先队列中。
    priority_queue<symbol> symbols;
    for (auto it = counter.begin(); it != counter.end(); it++) {
        letter l;
        l.ascii = (*it).first;
        l.frequency = (*it).second;
        symbol s;
        s.frequency = (*it).second;
        s.letters.push_back(l);
        symbols.push(s);
    }
    while (symbols.size() > 1) {
        symbol s1 = symbols.top();
        symbols.pop();
        symbol s2 = symbols.top();
        symbols.pop();
        symbol s3;
        s3.frequency = s1.frequency + s2.frequency;
        s3.letters.push_back(s1.letters[0]);
        s3.letters.push_back(s2.letters[0]);
        symbols.push(s3);
    }
    symbol s = symbols.top();
    string code;
    for (letter l : s.letters)
        l.code = code;
    cout << code;
}
```

```

        symbols.push(s);
    }
    // 使用贪心策略将具有最小频度的两个字符进行合并。
    while (symbols.size() > 1) {
        int sumOfFrequency = 0;
        vector<letter> merge;
        for (int i = 0; i < 2; i++) {
            symbol s = symbols.top();
            symbols.pop();
            sumOfFrequency += s.frequency;
            for (int j = 0; j < s.letters.size(); j++) {
                s.letters[j].code.insert(s.letters[j].code.begin(), '0' + i);
                merge.push_back(s.letters[j]);
            }
        }
        // 合并后的字符放入优先队列中。
        symbol s;
        s.frequency = sumOfFrequency;
        s.letters = merge;
        symbols.push(s);
    }
    // 输出编码。
    if (symbols.size()) {
        symbol s = symbols.top();
        symbols.pop();
        sort(s.letters.begin(), s.letters.end());
        for (int i = 0; i < s.letters.size(); i++)
            cout << s.letters[i].ascii << " " << s.letters[i].code << endl;
    }
}
//-----11.11.4.cpp-----//
```

强化练习：240 Variable Radix Huffman Encoding^D。

11.11.5 最优策略选择

在有关动态规划的题目中，有一类题目是给定若干限制条件并要求在这些限制条件下找出最优的选择策略，使得费用、时间、长度等某个变量最优化。一般来说，此类题目的解题关键是应用贪心策略进行选择。如果题目约束条件中给定的是两个相关的变量，且相应的任务之间有先后顺序，可以尝试按某种优先级为任务建立先后顺序，按先后顺序来完成任务，即可得到问题的最优解。

10026 Shoemaker's Problem^A (鞋匠的难题)

鞋匠有 N 项工作（来自顾客的订单）必须完成。鞋匠每天只能进行一项工作。对于第 i 项工作， T_i ($1 \leq T_i \leq 1000$) 表示以天为单位，鞋匠完成该项工作的时间。鞋匠每延迟一天开始第 i 项工作，他将支付 S_i ($1 \leq S_i \leq 10000$) 美分的罚金。编写程序来帮助鞋匠，找到具有最少罚金的工作序列。

输入

输入的第一行包含一个正整数 T ，表示测试数据的组数。接着是一个空行。每组测试数据的第一行包含

一个整数 N ($1 \leq N \leq 1000$), 接下来的 N 行, 每行包含两个整数, 按顺序给出了每项工作的完成时间和罚金。相邻两组测试数据由一个空行分隔。

输出

对于每组测试数据输出一行, 包含具有最少罚金的工作序列, 每项工作由其在输入中的序号表示。如果有多种工作序列满足要求, 输出字典序最小的工作序列。相邻两组输出之间打印一个空行。

样例输入

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

样例输出

```
2 1 3 4
```

分析

“鞋匠每天只能进行一项工作”的含义是一旦鞋匠选择开始某项工作, 需要将此项工作完成后才能开始其他工作。假设当前已经得到了具有最少罚金的工作序列, 任取序列中的两项工作 x 和 y , 令其完成时间和罚金分别为 t_x 和 t_y , s_x 和 s_y , 由于除两项工作 x 和 y 之外的工作已经固定, 那么只需考虑是先完成工作 x 还是先完成工作 y , 如果先完成工作 x , 则损失为 $t_x \times s_y$, 如果先完成工作 y , 后完成工作 x , 需支付罚金 $t_y \times s_x$, 如果 $t_x \times s_y < t_y \times s_x$, 应该先完成工作 x , 如果 $t_x \times s_y > t_y \times s_x$, 应该先完成工作 y , 当 $t_x \times s_y = t_y \times s_x$ 时, 按照题意, 需要选择具有较小序号的工作。推而广之, 对于该序列中任意两项工作均有此结论。

强化练习: 434 Matty's Blocks^C, 812 Trade on Verwegistan^C, 945 Loading a Cargo Ship^D, 10037 Bridge^A, 10747* Maximum Subsequence^D, 11054 Wine Trading in Gergovia^A, 11269 Setting Problems^D, 11729 Commando War^A。

扩展练习: 980 X-Express^E, 1093* Castles^D, 1205* Color a Tree^D^[152]^[153]。

11.12 小结

动态规划作为解题问题的一种思维方式和技巧, 与图算法一样, 一直是近来各种编程竞赛的考察重点。应用动态规划的标志是问题具有最优子结构和无后效性这两个特征。

最优子结构是指不论过去状态和决策如何, 对前面的决策所形成的状态而言, 余下的诸决策必须构成最优策略。简而言之, 一个最优化策略的子策略总是最优的。子问题最优时母问题通过优化选择后一定最优的情况叫做“最优子结构”。

无后效性是指已经做出的最优决策体现在当前状态上, 当前状态是之前所有最优决策的总结, 从此状态出发继续进行最优决策所得到的结果可以保证是最优的。也就是说, 不管之前是如何到达当前状态的, 只要当前状态相同, 从此状态继续进行决策所得到的最优解都是一样的, 当前状态之前的决策不再对后续的决策和最优结果产生影响, 只会通过当前状态产生影响。某阶段的状态一旦确定, 则此后过程的演变不再受此前各种状态及决策的影响, 简单的说, 就是“未来与过去无关”, 当前的状态是此前历史的一个完整总结, 此前的历史只能通过当前的状态去影响过程未来的演变。

最优子结构的一个表现是给定的问题存在重叠的子问题。重叠子问题是将原问题进行分解后, 可以得到一系列的子问题, 这些子问题相互之间是独立的, 但是不同的分解方式能够得到相同的子问题, 这似乎有些矛盾。子问题怎么会互相独立又有重叠呢? 以区间型动态规划为例, 给定一个整数区间 $[L, R]$, 每次从区

间中选取一个整数点位置将其分为两部分，一直分解下去，最终得到的是长度为 1 的子区间，在分解的过程，两个区间都是独立的，不会发生重叠，这对应着子问题分解的独立性。在第一次分解时，选择整数点 x 和整数点 y 作为第一次分解的边界将导致完全不同的两种分解方式，这两种分解方式在各自的分解过程中，子区间是互不相同的，但两种分解方式会产生相同的子区间，这对应着重复的子问题。本质上，动态规划通过只解决一次子问题，然后在此基础上通过子问题的解得到更大问题的解来提高解决问题的效率。

解决动态规划问题的一般步骤：(1) 建立模型，确认状态；(2) 确定递推关系式，亦即找出状态转移方程。动态规划由于需要通过较小的子问题来得到更大子问题的解，因此存在一种递推关系（或者称之为状态转移方程），在解题过程中，最为关键的就是要找出较小子问题和较大子问题之间存在的递推关系，之后解题过程就相对变得简单。(3) 找出初始条件。

理解动态规划，可以从最基础的动态规划学起，01 背包问题即是一种典型动态规划，建议读者反复揣摩背包问题，在理解基本的 01 背包问题的基础上，进一步理解多重背包问题和完全背包问题。动态规划属于一种思维技巧，它可以和其他的问题进行有机结合，创造出更为复杂的问题，例如与概率论结合。本章还有一些动态规划类型尚未覆盖，例如数位型动态规划、基于连通性状态压缩的动态规划（又称插头型动态规划）等，在读者已经掌握本章所介绍内容的基础上，理解这些类型的动态规划应该不难。

第 12 章 网格

I have not failed. I've just found 10000 ways that won't work.

——Thomas Alva Edison¹

关于网格（grid）的题目时有出现，大多和模拟、坐标变换、按指令在网格上行走有关。网格有多种类型，有矩形网格、三角形网格、六边形网格、地球经纬网格等等。

12.1 矩形网格

矩形网格（rectangular grid）是指在横纵坐标方向上相邻格点之间的距离都是单位距离的网格。矩形网格常见的操作是根据特定的指令在网格上行走，判断到达的位置，在此过程中，可能要求计算或记录指定的量，并随之衍生出各种题目形式。

12.1.1 网格行走

在矩形网格上一般可区分八个方向，按照地图方向“上北下南左西右东”的惯例，如果将书籍按封面朝向上的状态进行放置，读者坐在封面下侧所在的一方，**以读者的位置为参考**，一般将书籍纸张从底端往顶端走的方向设为北方，即箭头“↑”所指方向为北方，书籍从纸张左侧向右侧走的方向设为东方，即箭头“→”所指方向为东方，那么有向上为北（north），向右上为东北（northeast），向右为东（east），向右下为东南（southeast），向下为南（south），向左下为西南（southwest），向左为西（west），向左上为西北（northwest）。

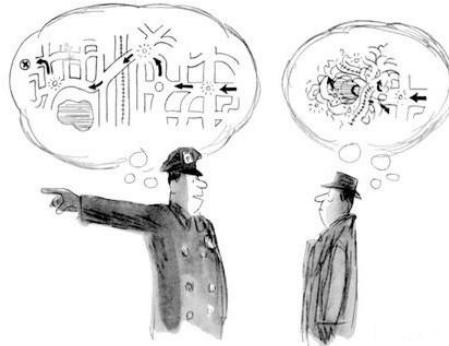


图 12-1 在日常生活中，问路和指路有时并不是一件容易完成的事

当给定的网格可以按对角线行走时，结合可能的八种方向变化：向前走（forward），右半转弯（half right），向右转（right），右急转弯（sharp right），向后退（backward），左急转弯（sharp left），向左转（left），左半转弯（half left），可将转向后的坐标偏移值表示为二维数组，便于在代码中引用而无需每次都加以判断和选择。

```
//++++++++++12.1.1.cpp+++++++/
```

¹ 托马斯·阿尔瓦·爱迪生（Thomas Alva Edison，1847—1931），美国发明家，商人。

```

// 定义方向常量，按照顺时针排序。
const int NORTH = 0, NORTH_EAST = 1, EAST = 2, SOUTH_EAST = 3, SOUTH = 4,
    SOUTH_WEST = 5, WEST = 6, NORTH_WEST = 7;
// 不同方向的数量。
const int CNT_DIRECTIONS = 8;
// 定义转向常量。
const int FORWARD = 0, HALF_RIGHT = 1, RIGHT = 2, SHARP_RIGHT = 3,
    BACKWARD = 4, SHARP_LEFT = 5, LEFT = 6, HALF_LEFT = 7;
// 进行各种转向后横向及纵向坐标的偏移值。
int offset[8][2] = {
    {0, 1}, {1, 1}, {1, 0}, {1, -1}, {0, -1}, {-1, -1}, {-1, 0}, {-1, 1}
};

```

按照上述方法安排方位和转向，获取转向后的方位时非常方便，只需将方位常数加上转向常数，然后模 8，余数即为转向后的方位常数，而转向后的坐标值偏移可以通过转向后的方位得到，如前述代码 offset 二维数组所示。

```

// 按照顺时针（逆时针）定义方向后，给定一个初始方向，当向左（右）转时，
// 可通过模运算获得后续的方向。
int x, y, direction = NORTH, turn = HALF_RIGHT;

if (turn == HALF_RIGHT) direction = (direction + 1) % CNT_DIRECTIONS;
else if (turn == RIGHT) direction = (direction + 2) % CNT_DIRECTIONS;
else if (turn == HALF_LEFT) direction = (direction + 7) % CNT_DIRECTIONS;
else if (turn == LEFT) direction = (direction + 6) % CNT_DIRECTIONS;

x += offset[direction][0], y += offset[direction][1];
//++++++12.1.1.cpp+++++++

```

在网格行走中，一个常见的操作是遍历给定方格的紧邻方格，可以使用如下方式进行遍历：

```

// m 为矩阵的行数，n 为矩阵的列数，r 为当前方格所在行，c 为当前方格所在列，从 0 开始编号。
for (int i = -1; i <= 1; i++)
    for (int j = -1; j <= 1; j++) {
        int rr = r + i, cc = c + j;
        if (rr >= 0 && rr < m && cc >= 0 && cc < n) {
            // 后续处理。
        }
    }

```

关于网格行走的题目绝大部分都和模拟有关，要求在解题时理解清楚题意，注意实现时的细节。此外，网格行走类的题目经常与深度优先遍历相结合。

10189 Minesweeper^A (扫雷)

给定一个 $n \times m$ 的字符矩阵，如果字符为 ‘*’，表示方格内是地雷，如果字符为 ‘.’ 表示方格内不包含地雷。需要你根据字符矩阵输出一个 $n \times m$ 矩阵，如果字符矩阵中方格是地雷，则原样输出字符 ‘*’，否则输出该方格周围相邻的 8 个方格中的地雷总数。

输入

输入包含若干个矩阵，对于每一个矩阵，第一行包含两个整数 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。安全方格用 ‘.’ 表示，有地雷的方格用 ‘*’ 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’ 是数据序号，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
*...
.....
.*...
.....
0 0
```

样例输出

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

分析

解题思路应该是很直接的——读取地雷阵，计算非地雷周围的地雷数量，然后按要求输出。需要注意在遍历 **安全方格** 周围坐标点时，枚举的坐标点要在地雷阵之内，因此需要进行范围检查，这也是所有类似网格问题中都需要进行的边界判断操作。注意输出的要求：在每两个矩阵之间输出一个空行，而不是每一个矩阵之后输出一个空行。

强化练习：114 Simulation Wizardry^B, 118 Mutant Flatworld Explorers^A, 155 All Squares^A, 201 Squares^B, 227 Puzzle^A, 320 Border^B, 411 Centipede Collisions^D, 556 Amazing^B, 587 There's Treasure Everywhere^A, 824 Coast Tracker^C, 10102 The Path in the Colored Field^A, 10116 Robot Motion^A, 10161 Ant on a Chessboard^A, 10279 Mine Sweeper^B, 10360 Rat Attack^A, 10377 Maze Traversal^B, 10452 Marcus^A, 10500 Robot Maps^C, 10642 Can You Solve It^A, 10961 Chasing After Don Giovanni^D, 10963 The Swallowing Ground^A, 11831 Sticker Collector Robot^A, 11975 Tele-Loto^D, 12498 Ant's Shopping Mall^D。

扩展练习：135* No Rectangles^C, 163 City Directions^D, 260 Il Gioco dell'X^A, 10964 Strange Planet^D, 11664* Langton's Ant^D, 12070* Invite Your Friends^E。

12.1.2 Flood-Fill 算法

Flood-Fill 算法，中文翻译有多种，有洪泛法、满水法、水流式填充法等，个人倾向于洪泛法的翻译，和英文原意贴近且有书面语的意味。Flood-Fill 算法实质上是图遍历在网格上的一种应用形式——从给定的任意一个方格开始，如果当前方格符合要求，向周围符合要求的其他方格继续进行搜索。遍历可以采用 DFS 或者 BFS，DFS 相对于 BFS 编写更为简洁，而且方便在递归过程中对满足条件的方格进行计数。

由于采用 Flood-Fill 算法的题目几乎都具有类似的解题步骤，可以将其“公式化”概括如下：

- (1) 题目给定的一般是一个网格，每个方格 **包含一个字符（或数字）**，因此首先需要确定网格的大小，即行数和列数，之后使用一个二维数组来表示整个网格；
- (2) 按题目给定的输入格式将每个方格的 **字符（或数字）** 读入到二维数组中；
- (3) 确定特征 **字符（或数字）**，即需要被替换的 **字符（或数字）**；
- (4) 使用 DFS 过程进行遍历，在遍历过程中将特征 **字符（或数字）** 予以替换，同时计数特征 **字符（或数字）** 的 **数量**；
- (5) 根据需要输出结果。

在 DFS 过程中，根据题目的设定，有的可能指定某个方格只和上下左右四个方格构成相邻关系，有的则会指定某个方格周围八个方向的方格均为相邻关系，需要适当予以调整，最为简便的方法是预先将其表示

成偏移数组，根据需要进行剪裁。

```
-----12.1.2.cpp-----
const int MAXN = 100;
char grid[MAXN][MAXN];
int rows, columns, total = 0;
// 如果只需遍历方格上下左右 4 个方向的相邻方格，取前 4 项偏移量即可。
int offset[8][2] = {
    {-1, 0}, {1, 0}, {0, -1}, {0, 1}, {-1, -1}, {-1, 1}, {1, -1}, {1, 1}
};

// 网格有 rows 行 columns 列，从 0 开始计数，每个方格有 4 个相邻的方格。
// old 表示特征字符，replaced 表示替换字符，可以将其表示为全局变量以避免在递归中进行传递。
void dfs(int i, int j, char old, char replaced) {
    // 范围检查，确保遍历不出网格范围。
    if (i >= 0 && i < rows && j >= 0 && j < columns && grid[i][j] == old) {
        // 计数。
        total++;
        grid[i][j] = replaced;
        for (int k = 0; k < 4; k++)
            dfs(i + offset[k][0], j + offset[k][1], old, replaced);
    }
}

// 另外一种写法，先进行范围检查，后进行递归调用。
void dfs(int i, int j, char old, char replaced) {
    total++;
    grid[i][j] = replaced;
    for (int k = 0; k < 4; k++) {
        int nexti = i + offset[k][0], nextj = j + offset[k][1];
        if (nexti >= 0 && nexti < rows && nextj >= 0 && nextj < columns)
            if (grid[nexti][nextj] == old)
                dfs(nexti, nextj, old, replaced);
    }
}
-----12.1.2.cpp-----
```

785 Grid Colouring^B (网格染色)

在二维网格上有一组使用字符表示的轮廓线，轮廓线由除了空格和下划线以外的任意可打印字符构成。在样例输入中，这个可打印字符设定为‘X’。在网格上的其他格点被空格或称为“标记”的字符所占据。

一个网格“区域”定义为位于轮廓线以内的一组网格格点，位于某个区域内的任意两个网格格点可以通过一条不穿越轮廓线的路径连接起来，注意这条路径只能由横向或纵向的线段组成。如果一个区域内部包含同样的“标记”字符（这些标记字符不能是空格或者用来绘制轮廓线的那些字符），那么该区域的状态称为“已标记”，注意，同一个区域中不能包含不同的标记字符。所有的轮廓线均使用相同的字符绘制而成，但是不同区域的“已标记”程度却并不一致。假如某个区域内部只包含空格，那么这个区域的状态称为“未标记”。网格中的任意一个区域要么是已标记的，要么是未标记的，而且标记字符只能出现在区域内部。

编写程序，从输入文件读取网格，找到其中的区域并填充已经标记的区域。如样例输出中所示，所有已标记的区域均已用标记字符填充完毕。

输入

输入文件中，每个网格由单独一行下划线组成的字符作为网格的结束标记。每个网格最多 30 行，每行

最多 80 个字符。网格每行的长度不定。

输出

按要求填充网格中已标记的区域并输出。输出时按照读入时的格式进行显示，包括分隔行，空行以及可能的前导或尾随空白字符。

样例输入

```
XXXXXXXXXXXXXXXXXXXXXX
X      X      X
X # # XXXXXXXX / X
X      X      X
XXXXXXXXXXXXXXXXXXXXXX
```

样例输出

```
XXXXXXXXXXXXXXXXXXXXXX
X#####X//////////X
X#####XXXXXXXXX///X
X##########X///X
XXXXXXXXXXXXXXXXXXXXXX
```

分析

首先确定构成轮廓线的字符，然后对于不是轮廓线字符也不是空格的字符，视其为标记字符，以标记字符为起点进行洪泛填充即可。根据题意，只要按照行优先顺序扫描，首先遇到的非空字符就是组成轮廓线的字符。

参考代码

```
char maze[35][85];
int offset[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};

// 洪泛填充。
void dfs(int i, int j, char old, char replaced) {
    if (i >= 0 && i < 35 && j >= 0 && j < 85 && maze[i][j] == old) {
        maze[i][j] = replaced;
        for (int k = 0; k < 4; k++)
            dfs(i + offset[k][0], j + offset[k][1], old, replaced);
    }
}

int main(int argc, char *argv[]) {
    string line;
    while (getline(cin, line)) {
        memset(maze, ' ', sizeof(maze));
        int rows = 0; char wall = 0;
        // 读入网格，确定构成轮廓线的字符。
        do {
            for (int i = 0; i < line.length(); i++) {
                maze[rows][i] = line[i];
                if (wall == 0 && line[i] != ' ') wall = line[i];
            }
            maze[rows++][line.length()] = '\n';
        } while (getline(cin, line), line.front() != '\n');
        // 寻找既不是轮廓线字符也不是空格字符的其他字符，这些字符是标记字符，
        // 以标记字符为起点进行洪泛填充。
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < 85; j++) {
                if (maze[i][j] == '\n') break;
                if (maze[i][j] != wall && maze[i][j] != ' ') {
                    char replaced = maze[i][j];
                    dfs(i, j, maze[i][j], replaced);
                }
            }
        }
    }
}
```

```

        maze[i][j] = ' ';
        dfs(i, j, ' ', replaced);
    }
}

// 按输入格式输出已经填充的网格。
for (int i = 0; i < rows; i++) {
    for (int j = 0; j < 85; j++) {
        cout << maze[i][j];
        if (maze[i][j] == '\n') break;
    }
    cout << line << '\n';
}
return 0;
}

```

强化练习: [352 Seasonal War^A](#), [469 Wetlands of Florida^A](#), [572 Oil Deposits^A](#), [601 The PATH^C](#), [657 The Die is Cast^A](#), [722 Lakes^C](#), [758 The Same Game^D](#), [776 Monkeys in a Regular Forest^C](#), [782 Contour Painting^C](#), [784 Maze Exploration^A](#), [830* Shark^D](#), [852 Deciding Victory in Go^C](#), [871 Counting Cells in a Blob^B](#), [10267 Graphical Editor^A](#), [10336 Rank the Languages^A](#), [10946 You Want What Filled^A](#), [11094 Continents^A](#), [11110 Equidivisions^B](#), [11244 Counting Stars^A](#), [11561 Getting Gold^B](#), [11585* Nurikabe^D](#), [11953 Battleships^A](#)。

扩展练习: [312 Crosswords \(II\)^D](#), [705 Slash Maze^B](#), [1103* Ancient Messages^C](#), [10592* Freedom Fighter^D](#), [10707* 2D-Nim^C](#)。

12.1.3 国际象棋棋盘

国际象棋棋盘 (chessboard) 属于有限网格的一种特殊形式。它是 8×8 的正方形网格，黑白相间，横向以字母 A 至 H 编号，纵向以 1 至 8 编号。

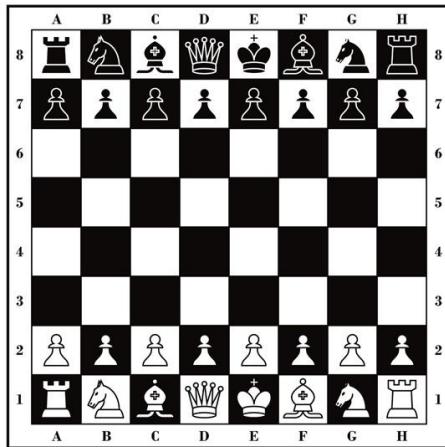


图 12-2 国际象棋棋盘。棋盘底端的两行白棋，一行为兵，最下端的一行从左至右依次为：车、马、象、后、王、象、马、车，黑棋和白棋的位置关于棋盘上下对称

棋盘置于对局者之间，己方棋盘的右下角必须为白格。国际象棋的棋子有王、后、象、马、车、兵。各种棋子的走子规则如下：

王 (king): 每次移动一格, 可横向、纵向、斜向移动到不被对方攻击的另外一方格中, 如果发生王车易位 (castling)¹, 可以横向移动两格。

后 (queen): 后可走到它所在的直线, 横线或斜线上的任何方格。不能越过棋子移动。

象 (bishop): 象可走到它所在斜线上的任何方格。不能越过棋子移动。

马 (knight): 马的走法由两个不同步骤组成, 先沿横线或直线走一格, 然后沿斜线方向再前进一格, 在走第一格时即使该格已有棋子占据也仍可行走, 与中国象棋中的马有“蹩脚”不同, 国际象棋中的马可以越过棋子移动。

车 (rook): 车可走到它所在的直线和横线上任何方格。不能越过棋子移动。

兵 (pawn): 兵只能朝前走。除吃子以外, 兵可从原始位置起沿所在直线向前走一格或两格 (所占据方格必须是空格), 以后每次只能沿直线向前走一格。吃子时, 只能吃它斜前方一格的棋子。当己方兵处于可攻击 “对方兵从原始方格一次走两格所经过的方格” 时, 可以把后者走两格当作走一格而吃掉它, 这种吃法只能在对方以该方式走兵后立即进行, 称为 “吃过路兵”。

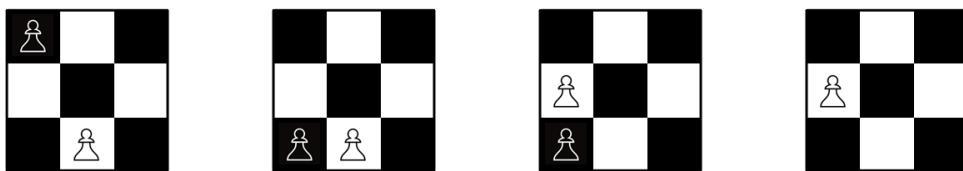


图 12-3 吃过路兵。黑棋兵第一次行棋, 往前走两格, 到达紧贴白棋兵的位置, 黑棋所经过的方格为白棋兵所能攻击的位置, 白棋兵走到该位置后, 即可将黑棋兵拿掉, 此即“吃过路兵”

兵一旦到达底线, 必须立即变换为与它相同颜色的后、车、马、或象, 这种变换仍被视作同一着, 变换何种棋子由棋手选择, 不必考虑棋盘上是否还有同类的其他棋子, 这种由兵变换为别的棋子的走法称为升变 (promotion), 升变的棋子立即生效。

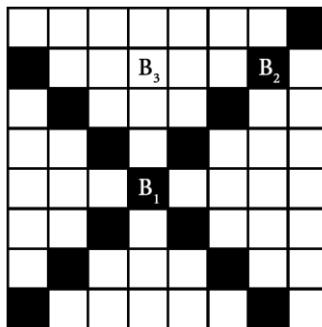
强化练习: 10196 Check the Check^A, [10284 Chessboard in FEN^B](#), 10849 Move the Bishop^B, [11231 Black and White Painting^B](#), 11494 Queen^A。

扩展练习: 286* Dead Or Not-That Is The Question^E, [10426 Knights' Nightmare^D](#)。

861 Little Bishops^A (棋盘上的象)

国际象棋中的象在棋盘上总是沿对角线方向移动, 如果两个象位于对方可以到达的位置, 就能够相互攻击。如下图所示, 黑色的方格表示象 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 间同样不能互相攻击。

¹ 在每一局棋中双方各有一次机会, 可以同时移动自己的王和一个车, 作为一步棋, 叫做王车易位。具体走法为王向一侧车的方向走两格, 再把车越过王放在王的旁边。



给定两个整数 n 和 k ，求出将 k 个象放置在一个 $n \times n$ 的棋盘上，并保证他们相互不能攻击的方案数。

输入

输入包含多组数据，每组数据包含一行共两个整数 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

对于每组测试数据输出一行，表示在指定大小的棋盘上放置指定个数的象且相互不能攻击的方案总数。你可以假定结果不超过 10^{15} 。

样例输入

```
8 6
4 4
0 0
```

样例输出

```
5599888
260
```

分析

根据观察和推理不难得出，在题目给定约束条件下在，在 $n \times n$ 的棋盘上最多能够放置 $2n-2$ 个象而不互相攻击（其中 1×1 的棋盘为特例，可以放置 1 个象）。

国际象棋的棋盘分为白色和黑色区域，在白色区域的象是无法攻击黑色区域内的象的，将黑白方格相间的棋盘顺时针旋转 45 度，则原来呈斜线的主、副对角线成为垂直和水平状态，此时象的走法和车的走法一致，问题转换为在这样的 $n \times n$ 棋盘上放置 k 个车有多少种方法。假设这样的棋盘第 i 行的方格数为 $r[i]$ ，用 $t[i][j]$ 表示在前 i 行放置 j 个车而互不冲突的方法，可以得到以下的递推关系

$$t[i][j] = t[i-1][j] + t[i-1][j-1] \times (r[i] - (j-1))$$

边界条件为

$$t[i][0] = 1, 0 \leq i \leq n; t[0][j] = 0, 1 \leq j \leq k$$

递推关系的意义可以这样理解：因为每一行只能放置一个车，则 j 个车要么全在前 $i-1$ 行，要么第 i 行有一个车； j 个车全在前 $i-1$ 行的放置方法为 $t[i-1][j]$ ；第 i 行放置一个车，前 $i-1$ 行放置 $j-1$ 个车，那么前 $i-1$ 行在放置 $j-1$ 个车时已经占用了第 i 行的 $j-1$ 个方格，剩余的方格数为 $r[i] - (j-1)$ ，则根据乘法原理，第二种放置方法是两者的乘积。又根据加法原理，总的放置方法为第一种和第二种方法数量的和。

边界情形也容易理解，前 i 行放置 0 个车的方法有 1 种，前 0 行放置至少 1 个车的方法有 0 种。由于将棋盘分成了两个区域，故在最后计算总的放置数时，应该是两个区域的累积。

强化练习：[278 Chess^A](#)，[639 Don't Get Rooked^A](#)，[696 How Many Knights^A](#)，[1589 Xiangqi^C](#)，[10237 Bishops^D](#)。

扩展练习: [10477 The Hybrid Knight^D](#), [10748 Knight Roaming^D](#), [10751* Chessboard^C](#), [11202* The Least Possible Effort^D](#), [11352 Crazy King^B](#)。

12.1.4 骑士周游问题

骑士周游问题 (knight's tour problem) 是指如下的问题: 在国际象棋棋盘上确定一条路径, 使得马根据行棋规则能够沿着这条路径访问每个方格恰好一次。如果起始方格和终止方格不同, 称该路径为开放骑士周游路径 (open knight's tour), 若起始方格和终止方格相同, 则称为闭合骑士周游路径 (closed knight's tour)。

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

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

图 12-4 8×8 棋盘上的骑士周游路径。左侧为开放骑士周游路径, 起始方格为(1, 1), 终止方格为(8, 1), 起始方格和终止方格不能通过一步到达; 右侧为闭合骑士周游路径, 起始方格为(1, 1), 终止方格为(3, 2), 起始方格和终止方格可以通过一步到达

骑士周游问题本质上是求无向图的哈密顿路 (回), 属于 NP 问题, 目前尚无有效算法, 只能通过回溯予以解决。不过对于 8×8 的国际象棋棋盘来说, 可以采用以下两个优化技巧使得解可以更快地得以确定。

(1) 对于棋盘的每个方格 (i, j) , 令 $cnt[i][j]$ 表示方格 (i, j) 的“后继方格”——马在方格 (i, j) 内通过一步跳跃能够到达的其他方格 (i', j') ——的数量¹。在回溯过程中, 将从当前方格出发马能够到达的所有方格按“后继方格”数量升序排列, 每次选择候选方格进行回溯时, 选取尚未走过的具有最小“后继方格”数量的方格作为下一个位置, 这样做能够尽量减少搜索空间, 提高出解效率。

(2) 对于较小的棋盘, 在表示已访问方格的状态时可以使用位掩码技巧来进行加速。例如, 对于 8×8 的棋盘, 可以按行优先顺序, 将棋盘的每个方格对应于无符号 64 位整数 $mask$ 的一个二进制位, 初始时, $mask=0$, 假如位于第二行第三列的方格已经访问, 则置 $mask$ 的二进制表示中从低位 (即最右侧) 数起第 11 位为 1, 此时 $mask=10000000000_2$ 。

¹ 可以看到, 按照“后继方格”数量进行下一步位置的选择实际上是后续介绍的 Warnsdorff 启发式规则的一种不完全应用, 即获取的“后继方格”数量是非实时的。也就是说, 在回溯过程中有些方格已经访问, 这些方格不应该被计入“后继方格”之内, 但由于“后继方格”数量是预先计算得到的, 在回溯过程中未给予及时更新, 获取的“后继方格”数量是“过时”的。读者可以进一步思考以下问题: 如何在回溯过程中实时地获取“后继方格”数量而不是预先计算“后继方格”的数量?

以下是使用上述优化技巧的骑士周游问题回溯法解题参考实现。

```

//-----12.1.4.1.cpp-----//
typedef unsigned long long ULL;

// 定义点数据结构以保存位置。
struct point {
    int x, y;
    point (int x = 0, int y = 0): x(x), y(y) {}
};

// 位置偏移量。
const int offset[8][2] = {
    {-2, -1}, {-2, 1}, {-1, 2}, {1, 2},
    {2, 1}, {2, -1}, {-1, -2}, {1, -2}
};

// SUCCEED 表示回溯成功的标记, NR 表示棋盘的行数, NC 表示棋盘的列数。
ULL SUCCEED;
int NR, NC, cnt[8][8];
vector<point> path;

// 用于为候选位置排序的函数。
bool cmp(point &a, point &b) { return cnt[a.x][b.y] < cnt[b.x][b.y]; }

// 回溯。
bool dfs(int r, int c, ULL mask) {
    // 位掩码为指定标记时表示所有方格已经访问。
    if (mask == SUCCEED) {
        path.push_back(point(r, c));
        return true;
    }
    // 计数从当前方格能够到达的其他方格的数量。
    int tot = 0;
    point ps[9];
    for (int i = 0; i < 8; i++) {
        int nr = r + offset[i][0], nc = c + offset[i][1];
        if (nr < 0 || nr >= NR || nc < 0 || nc >= NC) continue;
        if (mask & (1ULL << (nr * NC + nc))) continue;
        ps[tot++] = point(nr, nc);
    }
    if (!tot) return false;
    // 对可达方格按优化技巧指定的规则排序, 依次选择进行递归。
    sort(ps, ps + tot, cmp);
    for (int i = 0; i < tot; i++) {
        if (dfs(ps[i].x, ps[i].y, mask | (1ULL << (ps[i].x * NC + ps[i].y)))) {
            path.push_back(point(r, c));
            return true;
        }
    }
    return false;
}

// W 为棋盘的宽度, H 为棋盘的高度, SR 为起始位置所在行, SC 为起始位置所在列 (均从 1 开始计数)。
void knightTour(int W, int H, int SR, int SC) {
    NC = W, NR = H;
    SUCCEED = (1ULL << (NR * NC - 1)) + ((1ULL << (NR * NC - 1)) - 1ULL);
}

```

```

// 统计能够到达当前方格的其它方格数量。
for (int r = 0; r < NR; r++) {
    for (int c = 0; c < NC; c++) {
        cnt[r][c] = 0;
        for (int i = 0; i < 8; i++) {
            int nr = r + offset[i][0], nc = c + offset[i][1];
            if (nr < 0 || nr >= NR || nc < 0 || nc >= NC) continue;
            cnt[r][c]++;
        }
    }
}
// 回溯。
path.clear();
dfs(SR - 1, SC - 1, 1ULL << (SR * SC - 1));
// 由于递归的性质，在输出时需要将跳过的方格逆序。
if (path.size()) reverse(path.begin(), path.end());
}

int main(int argc, char *argv[]) {
    // W 为棋盘宽度，H 为棋盘高度，SR 为起始位置所在行，SC 为起始位置所在列（均从 1 开始计数）。
    int cases = 0, W, H, SR, SC;
    while (cin >> W >> H >> SR >> SC) {
        if (cases++) cout << '\n';
        knightTour(W, H, SR, SC);
        if (path.size()) {
            int board[H][W] = {0};
            // 为了输出的可读性，将步数显示在方格内。
            for (int i = 0; i < H; i++)
                for (int j = 0; j < W; j++)
                    board[path[i * W + j].x][path[i * W + j].y] = i * W + j + 1;
            for (int i = 0; i < H; i++) {
                for (int j = 0; j < W; j++) {
                    if (j) cout << ' ';
                    cout << setw(2) << right << board[i][j];
                }
                cout << '\n';
            }
        }
        else cout << "No solution.\n";
    }
    return 0;
}
//-----12.1.4.1.cpp-----//

```

对于以下输入：

```

8 8 1 1
5 5 1 1
5 5 2 2

```

其输出为（某个方格内的数字为其步数的序号，在第一组测试数据对应的输出中，第一行第一列为‘1’，表示从此方格开始，第二行第三列为‘2’，表示马第二步跳到此方格内，依此类推）：

```

1 46 13 34 3 20 15 18
60 35 2 45 14 17 4 21
47 12 59 36 33 44 19 16
58 61 48 53 50 37 22 5

```

```

11 28 57 62 43 32 51 38
56 63 54 49 52 39 6 23
27 10 29 42 25 8 31 40
64 55 26 9 30 41 24 7

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

```

No solution.

从输出可以看出，对于某些尺寸的棋盘，特定的起始位置可能并不存在骑士周游问题的解决方案。

可以证明，对于 $m \times n$ ($1 \leq m \leq n$) 的棋盘来说，如果 m 和 n 满足下列三个条件之一，则该 $m \times n$ 的棋盘不存在闭合骑士周游路径^[154]，这三个条件为：(1) m 和 n 都是奇数；(2) $m=1, 2, 4$ ；(3) $m=3$ 且 $n=4, 6, 8$ 。如果 m 和 n 两者的较小值至少为 5，则该 $m \times n$ 的棋盘总是存在开放骑士周游路径^{[155][156]}。

对于较小的 m 和 n 来说 ($m \leq n \leq 8$)，使用上述介绍的回溯法从任意位置寻找闭合骑士周游路径能够在较短时间内获得解，但当 m 和 n 较大时，无法在短时间内得到解。此时可以应用 Warnsdorff 启发式规则在 m 和 n 不太大时 ($m \leq n \leq 200$) 以近似 $O(n)$ 的时间复杂度生成开放骑士周游路径^[157]。令马的当前位置为 P ，从 P 能够到达的其他位置 Q 构成集合 S ，Warnsdorff 启发式规则要求在选择当前位置 P 的下一个位置时，尽量从 S 中选择这样的位置 Q ：从 Q 出发能够到达的其他的尚未访问的可行位置最少，从图论的角度来说就是要求位置 Q 所对应的顶点具有最小的度。如果有多个位置 Q 均满足条件则有三种方法予以选择：(1) 随机选择一个可行位置；(2) 选择距离棋盘中心的欧几里得距离最远的位置；(3) 根据 m 和 n 的大小和当前已到达的方格采用一种预先固定的选择顺序^[158]。以下是根据 Warnsdorff 启发式规则寻找开放骑士周游路径的参考实现¹。

```

//-----12.1.4.2.cpp-----
const int MAXN = 64;

// 棋盘的大小为 N×N, (SR, SC) 表示马的起始方格。
int N, SR, SC, board[MAXN][MAXN];

// 马在跳跃时的位置偏移。
static int dr[8] = { 1, 1, 2, 2, -1, -1, -2, -2 };
static int dc[8] = { 2, -2, 1, -1, 2, -2, 1, -1 };

// 边界检查。
inline bool isInBounds(int r, int c) {
    return ((r >= 0 && c >= 0) && (r < N && c < N));
}

// 检查目标位置是否可达，即目标位置在棋盘边界内且尚未被访问。
inline bool isEmpty(int r, int c) {
    return isInBounds(r, c) && (!board[r][c]);
}

```

¹ 参阅：<https://www.geeksforgeeks.org/warnsdorffs-algorithm-knights-tour-problem/>, 2020。

```

// 获取指定位置(r, c)的度。
inline int getDegree(int r, int c) {
    int cnt = 0;
    for (int i = 0; i < 8; i++)
        if (isEmpty(r + dr[i], c + dc[i]))
            cnt++;
    return cnt;
}

// 根据 Warnsdorff 启发式规则获取马的下一个位置。
bool nextMove(int *r, int *c) {
    int minDegIdx = -1, degree, minDeg = 8, nr, nc;
    // 当有多个位置可选时随机选择一个可行位置。
    int si = rand() % 8;
    for (int i = rand() % 8, j = 0, k; j < 8; j++) {
        k = (i + j) % 8;
        nr = *r + dr[k], nc = *c + dc[k];
        // 获取具有最小可达度的位置作为下一步的候选位置。
        if ((isEmpty(nr, nc)) && (degree = getDegree(nr, nc)) < minDeg) {
            minDegIdx = k;
            minDeg = degree;
        }
    }
    if (minDegIdx == -1) return false;
    nr = *r + dr[minDegIdx];
    nc = *c + dc[minDegIdx];
    board[nr][nc] = board[*r][*c] + 1;
    *r = nr, *c = nc;
    return true;
}

// 输出骑士周游路径。
void render() {
    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++)
            cout << setw(5) << right << board[r][c];
        cout << '\n';
    }
}

// 寻找开放骑士周游路径。
bool findOpenKnightTour() {
    int r = SR - 1, c = SC - 1;
    memset(board, 0, sizeof(board));
    board[r][c] = 1;
    for (int i = 2; i <= N * N; i++)
        if (!nextMove(&r, &c))
            return false;
    return true;
}

int main(int argc, char *argv[]) {
    int cases = 0;
    while (cin >> N >> SR >> SC) {
        if (cases++) cout << endl;

```

```

    if (N < 5) {
        cout << "No Knight's Tour.\n";
        continue;
    }
    while (!findOpenKnightTour()) {}
    render();
}
return 0;
}
//-----12.1.4.2.cpp-----//

```

如果需要寻找闭合骑士周游路径，则可以使用如下的方法：先使用上述代码得到开放骑士周游路径，之后检查起始方格和终止方格，查看两者是否能够通过一步移动到达。如果能，则表明寻找得到的开放骑士周游路径同时也是一条闭合骑士周游路径。计算机实验得到的结论指出，如果随机选择起始位置，通过一次运行就能得到闭合骑士周游路径的概率，要比选择固定起始位置时得到闭合骑士周游路径的概率大得多。

```

// 输出骑士周游路径，按设定的起始位置对表示移动顺序的序号进行调整。
void render() {
    int delta = board[SR - 1][SC - 1], mod = N * N;
    for (int r = 0; r < N; r++) {
        for (int c = 0; c < N; c++) {
            cout << setw(5) << right << (board[r][c] - delta + mod) % mod + 1;
            cout << '\n';
        }
    }
}

// 检查两个位置是否能够通过一步移动到达。
bool neighbour(int er, int ec, int sr, int sc) {
    for (int i = 0; i < 8; i++) {
        if ((er + dr[i]) == sr && ((ec + dc[i]) == sc))
            return true;
    }
    return false;
}

// 寻找闭合骑士周游路径。
bool findClosedKnightTour() {
    // 随机选择起始位置以提高程序运行成功的概率，输出时对表示移动次序的序号进行相应调整即可。
    srand(time(NULL));
    int sr = rand() % N, sc = rand() % N;
    int r = sr, c = sc;
    memset(board, 0, sizeof(board));
    board[r][c] = 1;
    for (int i = 2; i <= N * N; i++) {
        if (!nextMove(&r, &c))
            return false;
    }
    // 检查起始位置和终止位置是否在一步以内可达。
    if (!neighbour(r, c, sr, sc)) return false;
    return true;
}

```

可以在常规的骑士周游问题基础上衍生出许多问题。例如，如果 $m \times n$ 的棋盘上不存在闭合骑士周游路径，那么从 $m \times n$ 的棋盘上需要最少移除多少个方格才能够使得剩余的棋盘存在闭合骑士周游路径？又或者，在 $m \times n$ 的棋盘上需要最少增加多少个方格才能够使得棋盘存在闭合骑士周游路线？在这一类扩展问题中，比较常见的是如下的问题：给定一组共 k 个位置，要求确定马从棋盘的任意一个位置出发，经过所有这些感

兴趣的位置至少一次所需要的最少跳跃步数。可以将此问题转化为旅行商问题予以解决。如果使用回溯法求解，则时间复杂度为 $O(k!)$ ，如果使用集合型动态规划，则可使时间复杂度降为 $O(k2^k)$ ，对于 $k \leq 16$ 的情形，可以在限定时间内获得通过。

强化练习：10255 The Knight's Tour^D。

扩展练习：11643* Knight Tour^D。

12.2 三角形网格

三角形网格的边一般为正三角形。题目形式一般是按指定的方式给格点编号，求格点之间的距离，距离分两种，一种是直线距离，另外一种是按编号方式得到的距离。如图 12-5 所示：

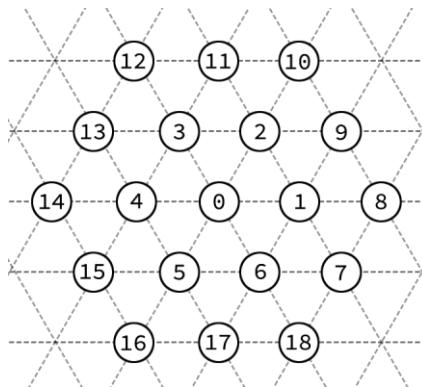


图 12-5 按指定规则编号的三角形网格

一般先按照编号规则求出各个格点在直角坐标系中的对应坐标，然后计算直线距离。坐标的转换需要从题目设定的编号规则中寻找规律来获取。例如上图给定的编号规则，每完成一圈向右移动一个单位距离，每条边的格点增加一个，规律性很明显，可以利用这个特点来获取坐标值。以下代码设定格点之间的单位距离为 2，这样方便计算，避免了小数，使用格点距离原点的横纵坐标来表示，横坐标的单位为 1，其中纵坐标的单位为 $\sqrt{3}$ ，这样做在求格点直角坐标的过程中不需将其转换成小数，可以避免精度问题导致的误差。

```

//-----12.2.cpp-----//
// 需要计算对应直角坐标的格点数量，从 0 开始计数。
const int MAXV = 10000001;

// 表示直角坐标，其中单位距离为 2，横坐标的单位为 1，纵坐标的单位为 sqrt(3)。
struct point { int x, y; };

// 保存格点直角坐标的数组。
point grid[MAXV];

// 计算指定数量格点的直角坐标。
void setCoordinate() {
    // 定义横纵坐标值的偏移量，沿着格点形成的六边形行走，有固定的偏移量。
    int offset[6][2] = {{1, 1}, {-1, 1}, {-2, 0}, {-1, -1}, {1, -1}, {2, 0}};

    // 定义每次行走所经过的格点数，注意是从原点开始，第一步是向右行走。
    int steps[6] = {0, 1, 1, 1, 1, 1};
}

```

```

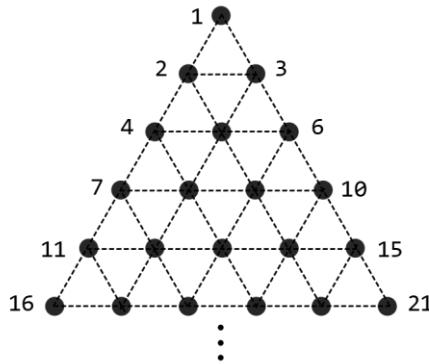
// 设置最后一次经过的格点的坐标。
int index = 0;
point last = (point){0, 0};
grid[index++] = last;

// 根据编号规律计算格点的直角坐标。
while (true) {
    last.x += 2, grid[index++] = last;
    if (index >= MAXV) return;
    for (int i = 0; i < 6; i++) {
        for (int j = 0; j < steps[i]; j++) {
            last.x += offset[i][0], last.y += offset[i][1];
            grid[index++] = last;
            if (index >= MAXV) return;
        }
        // 每完成一圈，各边上的格点数增加 1。
        steps[i]++;
    }
}
//-----12.2.cpp-----//

```

209 Triangular Vertices^c (三角顶点)

给定如下所示的无限等边三角形网格：



如果从左到右，从上到下对格点进行标号，某些格点的编号组合恰好构成特定几何图形的顶点。例如，格点{1, 2, 3}和{7, 9, 18}为三角形的顶点，格点{4, 6, 11, 13}和{2, 7, 9, 18}为菱形的顶点，格点{4, 5, 9, 13, 12, 7}为正六边形的顶点。

编写程序根据给定的格点编号组合，分析确定它们是否是下列可选的图形之一：三角形、菱形、正六边形。这些图形必须满足以下两个条件：

- (1) 图形的每条边必须与网格的边重合。
- (2) 图形的各条边长度相等。

输入

每组数据中给定的格点数量不定，单组数据独占一行，一组数据至多有 6 个格点，所有格点的编号均在 1 至 32767 的闭区间内。

输出

对于输入中的每组格点编号，你的程序需要从格点的数量和编号推断出它们表示的是何种几何图形。注意：如果给出6个格点，则只可能表示正六边形或者不构成几何图形，即所有点都必须为顶点，如果格点在图形的某一边上，则认为不构成任何图形。在输出时，先按原样输出给定格点的编号，接着按样例输出的格式给出你的分析结果。

样例输入

```
1 2 3
11 13 29 31
26 11 13 24
4 5 9 13 12 7
1 2 3 4 5
47
11 13 23 25
```

样例输出

```
1 2 3 are the vertices of a triangle
11 13 29 31 are not the vertices of an acceptable figure
26 11 13 24 are the vertices of a parallelogram
4 5 9 13 12 7 are the vertices of a hexagon
1 2 3 4 5 are not the vertices of an acceptable figure
47 are not the vertices of an acceptable figure
11 13 23 25 are not the vertices of an acceptable figure
```

分析

第一种解法，将给定的点编号按照从小到大的顺序进行排序，将其转换为直角坐标系下的坐标，计算边长，若边长满足要求，则输出所对应的几何图形。具体方法是将相邻顶点间的边长定为2个单位长度，设编号为1的顶点为坐标原点， x 坐标向右为正， y 坐标向上为正，其坐标值为 $(0, 0)$ ，则编号为2的顶点坐标值为 $(-1, -\sqrt{3})$ ，编号为3的顶点坐标值为 $(1, \sqrt{3})$ ，同处一行的编号，相邻右侧编号其横坐标值比左侧编号横坐标值增加2，纵坐标不变。对于处于每行最左端的顶点，每向下增加一行，顶点的横坐标值减1，纵坐标值减 $\sqrt{3}$ ，因此可以先求得所有顶点的坐标，然后根据几何图形的边长条件来判断。注意几何图形的边需要和网格中的边重合这个条件的判断。

第二种解法，不计算直角坐标系下的坐标值，而是通过编号规律解题。观察编号的规律，可以发现编号所处的“行”不同，如编号1所在行为1，编号2、3所在行为2，编号4、5、6所在行为3等，以此类推。知道了某个编号所处行，通过行的差值可以获知编号构成几何图形的边长值，例如编号4的行为2，编号11的行为4，则编号4和编号11作为边的菱形其边长为两者行数之差——2。其次，可以观察到编号1, 3, 6, 10, …, 36, 45, …在同一条“主对角线”上，这条对角线从左上至右下，类似的还有编号2, 5, 9, …, 35, 44, …形成的对角线，4, 8, 13, 19, …, 34, 43, …形成的对角线等；编号1, 2, 4, 7, …, 39, 37, …在同一条“副对角线”上，这条对角线从右上至左下，类似的还有编号3, 5, 8, 12, …, 30, 38, …形成的对角线，6, 9, 13, 18, …, 31, 39, …形成的对角线等。对于所有符合要求的几何图形来说，它们的边必定位于这些对角线上。那么可以采用类似于并查集的表示方法，将编号所在的“主对角线”和“副对角线”使用一个数字来表示，这样在判断两个编号是否同处于一条对角线上时将非常方便。具体解题方法类似于第一种解法，先将编号从小到大排序，然后获取其所在行，通过各个顶点所在行是否处于同一对角线上这一条件，判断其是否符合指定类型的几何图形要求。例如对于三角形的判断，三角形要么顶角向上，如{1,

$2, 3\}$ 所构成的三角形，要么顶角向下，如 $\{7, 9, 18\}$ 所构成的三角形。将顶点编号按升序排列，设为 v_1, v_2, v_3 ，要么 v_2 和 v_3 位于同一行，且 v_1 和 v_2 同处一条对角线上， v_1 和 v_3 同处一条对角线上；要么 v_1 和 v_2 位于同一行，且 v_1 和 v_3 同处一条对角线上， v_2 和 v_3 同处一条对角线上。对于菱形和正六边形还需增加边长相等的判断。

强化练习：[10022* Delta-wave^C](#)，[10233 Dermuba Triangle^C](#)，[11092 IIUC HexWorld^D](#)。

12.3 六边形网格

蜂巢的截面呈现规则的正六边形^I，之所以蜂类会选择正六边形作为基本结构来建筑蜂巢，人们认为，可能是因为正六边形是覆盖二维平面的最佳拓扑结构。如果选择正三角形，其内部所包围的空间有限，不便于利用，而正方形又由于力学结构太弱容易变形而不是一个好的选择。由六边形作为基础结构建成的蜂窝具有重量轻，强度高，节省材料等诸多优点，因此得到蜂类的青睐。日常使用的手机网络一般采用的也是蜂窝式结构，每个基站覆盖的范围是一个近似圆形的区域，而圆心则位于六边形网格中每个正六边形的中心。

一般所讨论的六边形网络，每个方格都是一个正六边形。在有关六边形网络的问题中，常见的主题是坐标变换和确定两个方格间的最短距离。坐标变换是指给定一组规则，将六边形网格从一种编号规则转换为另外一种编号规则。由于正六边形具有规则的外形，以网格中某个正六边形的中心为原点，只要给出其他正六边形中心相对于原点的坐标，即可求得正六边形与原点在横向和纵向上相差的正六边形个数。

给定两个正六边形的编号，要求确定经过正六边形的中心到达对方的最短路径。对于规模较小的网格，可以通过将单个正六边形视为图的顶点，使用 BFS 确定最短路径，但对于规模较大的网格却不适用，此时需要将正六边形的坐标予以适当变换，转化为直角坐标系坐标，从而有利于问题的解决。

强化练习：[808 Bee Breeding^B](#)，[10182 Bee Maja^A](#)。

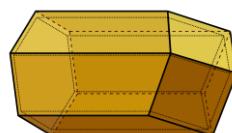
扩展练习：[317 Hexagon^D](#)，[360 Don't Get Hives From This One^E](#)，[10159 Star^D](#)。

12.4 经度与纬度

在地球球面上，与赤道平行的东西方向的环线称为纬线（lines of latitude）。赤道的纬度为 0 度，而北极和南极分别是北纬 90 度和南纬 90 度。一般北纬的角度范围定义为 $[0, 90]$ 度，南纬的角度范围定义为 $[-90, 0]$ 度。经过北极和南极的大圆线称为经线（lines of longitude），以经过格林威治（Greenwich）天文台的经线为 0 度经线（又称本初子午线，该经线是人为选取的），向东为东经，范围为 $[0, 180]$ 度，向西为西经，范围定义为 $[-180, 0]$ 度，实际效果来看，西经 -180 度和东经 180 度是同一条经线。

地球上的测地线（geodetic line）是地球曲面上两点间的最短球面曲线，计算测地线的长度有固定公式。如果将计算测地线的长度问题看做大圆距离（great-circle distance）问题，则有以下计算公式。设地球上点 p 的纬度和经度为 $(plat, plong)$ ，点 q 的纬度和经度为 $(qlat, qlong)$ ，地球的半径为 R ，则 p 和 q 之间的球面

^I 对于单个蜂房而言，它的一端是一个平面的正六边形（下图的左端），而另一端却不是平面的正六边形，而是由三个菱形构成的三维结构组成（下图的右端），其中每个菱形的钝角约为 $109^{\circ}28'$ ，锐角约为 $70^{\circ}32'$ 。



距离 $D(p, q)$ 为

$$D(p, q) = R \times \cos^{-1}(\sin(lat) \sin(qlat) + \cos(lat) \cos(qlat) \cos(plong - qlong))$$

需要注意的是，经纬度的单位均为弧度，如果给定经纬度的单位为“度分秒”格式，需要将其转换为弧度单位后才能应用上述公式。大圆距离公式形式简单，容易记忆，使用代码表示也很方便。

```
//++++++12.4.cpp+++++++
double gcd
(double R, double plat, double plong, double qlat, double qlong) {
    double r = 0;
    r = R * acos(sin(plat) * sin(qlat) + cos(plat) * cos(qlat) * cos(plong - qlong));
    return r;
}
```

强化练习：[10316](#) Airline Hub^D，[10897](#) Travelling Distance^D，[11817](#) Tunnelling the Earth^D。

大圆距离公式使用余弦函数，当两点的经纬度相差较小时，误差可能较大，如果题目的输出精度要求较高，一种方法是对于经纬度相同的两个地点，直接输出距离为零，以避免计算误差；另外一种方法是采用半正矢公式（Haversine formula），该公式在经纬度差值较小的情况下，其计算得到的距离数值精度仍然较高。

```
double haversine
(double R, double plat, double plong, double qlat, double qlong) {
    double lon = qlong - plong, lat = qlat - plat, a = 0, c = 0;
    a = pow((sin(lat / 2)), 2) + cos(plat) * cos(qlat) * pow(sin(lon / 2), 2);
    c = 2 * atan2(sqrt(a), sqrt(1 - a));
    return R * c;
}
//++++++12.4.cpp++++++
```

强化练习：[535](#) Globetrotter^C。

扩展练习：[10075](#) Airlines^C。

12.5 小结

网格一般是作为一种问题背景出现在题目中。

常见的网格分为以下几类：(1) 矩形网格，尤以正方形网格最为常见；(2) 三角形网格或者六边形网格；(3) 经纬度网格。

深度优先遍历与矩形网格密切相关，比如本章介绍的洪泛算法就是深度优先遍历在网格上的应用。深度优先遍历在矩形网格上一般是用于统计连通块的数量。由于棋盘也是一种矩形网格，因此与国际象棋相关的题目也会时常出现，此类题目经常与搜索或者计数有关。

六边形网格实际上是由多个三角形构成的，因此两种网格之间有一定的联系。在这两种网格上，主要会出现一些与几何问题。

经纬度网格主要与计算地球上两点之间的球面距离有关，需要使用相应的计算公式。

第 13 章 几何

Let no one who is ignorant of geometry enter here.

不习几何者不得入内。

——柏拉图学园^I

几何 (geometry) 是一门古老的学科, 是数学的一个基本分支, 主要研究有关形状、大小、图形的相对位置、空间的属性等问题^{II}。正如“几何”的含义——是“多少”的问题, 在编程竞赛中常见的是点、直线、三角形、圆等有关的距离或角度计算的主题。由于不能使用浮点数精确地表示所有实数, 在计算过程中需要特别注意浮点数运算的精度问题。

13.1 点

在表示点 (point) 时, 因为使用的坐标系不同, 一般有两种常用的表示方法。一种是在直角坐标系 (Cartesian coordinate system, 又称笛卡尔坐标系) 中, 使用一个有序实数对(x, y)来表示某个点相对于原点(0, 0)在 X 轴和 Y 轴上的偏移量。可使用结构体表示为:

```
struct pointOfCartesian {  
    double x, y;  
};
```

另外一种是在极坐标系 (polar coordinate system) 中, 使用相对于原点的距离 d 和角度 a 来表示一个点:

```
struct pointOfPolar {  
    double d, a;  
};
```

表示角度的 a 可使用一般的角度单位度或者弧度单位。两种坐标系可相互转换, 极坐标转换为直角坐标:

$$x = d \cos(a), \quad y = d \sin(a)$$

直角坐标转换为极坐标:

$$d = \sqrt{x^2 + y^2}, \quad a = \tan^{-1}\left(\frac{y}{x}\right)$$

其中, \tan^{-1} 为反正切。在 $x=0$ 的情况下, 若 y 为正数, 则 a 等于 90 度 ($\pi/2$ 弧度), 若 y 为负数, 则 a 为 -90 度 ($-\pi/2$ 弧度)。极坐标在计算点连续移动后的最终位置时非常有用, 给定点移动的距离和转角, 使用坐标转换公式即可得到点相对于初始位置在 X 轴和 Y 轴的位移 (有正负), 将初始坐标加上位移即为最终位置的坐标。

如果给定两个点的坐标, 根据直角坐标系中两点间的距离公式, 可以很容易计算其距离。有时在解题时并不需要计算其实际距离, 而只要比较距离大小, 那么可以使用两点距离的平方来代替实际距离进行比较。

```
// 距离的平方。
```

^I Platonic Academy, 又称柏拉图学院, 是由古希腊哲学家柏拉图 (Plato, 约前 427 年—前 347 年) 在约前 387 年创立于雅典的学校。

^{II} <https://en.wikipedia.org/wiki/Geometry>, 2020。

```

double squareOfDistance(point a, point b)
{
    return pow(a.x - b.x, 2) + pow(a.y - b.y, 2);
}

// 两点间实际距离。
double distanceOfCartesian(point a, point b)
{
    return sqrt(squareOfDistance(a, b));
}

```

强化练习: 142 Mouse Clicks^B, 920 Sunny Mountains^B, 1595 Symmetry^C, 10242 Fourth Point^A, 10310 Dog and Gopher^A, 10357 Playball^D, 10585 Center of Symmetry^C, 11519 Logo 2^D。

扩展练习: 10687 Monitoring the Amazon^C。

13.2 直线

13.2.1 直线的表示

两个不重合的点确定一条直线, 因此很自然的, 可以使用以下的方法定义直线:

```
struct line { point a, b; };
```

此表示方法一般为处理问题的基础, 因为大多数情况下都是先给出直线上两点的数据, 然后由此得到直线的其他表示形式。亦可用“斜截式”——直线的斜率 k 和在 Y 轴上的截距 b 来表示一条直线:

```

struct line {
    double k, b;
    line (double k = 0, double b = 0): k(k), b(b) {}
};
```

此时直线的方程为:

$$y = kx + b$$

但对于与 X 轴垂直的直线, 无法使用“斜截式”表示, 因为“垂线”的斜率为无穷大, 故一般使用以下形式表示与 X 轴垂直的直线:

$$x = C$$

其中 C 为一个常数。对于与 X 轴平行的直线, 仍可以使用“斜截式”表示, 此时直线的斜率为 0。

为了克服“斜截式”的不足, 可以使用“一般式”来表示直线。直线的一般式方程为:

$$ax + by + c = 0$$

在代码中可以使用一般方程的系数来表示直线:

```

struct line {
    double a, b, c;
    line (double a = 0, double b = 0, double c = 0): a(a), b(b), c(c) {}
};
```

当给定两个不同点时, 可以由其确定一条直线:

```

const double EPSILON = 1E-7;

// 根据两点得到一条直线, 使用一般形式表示。
line getLine(double x1, double y1, double x2, double y2)
{
```

```

        return line(y2 - y1, x1 - x2, y1 * (x2 - x1) - x1 * (y2 - y1));
    }

line getLine(point p, point q)
{
    return getLine(p.x, p.y, q.x, q.y);
}

```

若给定是直线的斜率和直线上的一点，则可按如下方式进行转换：

```

// 将使用斜率与直线上一点来表示直线的方式转换为一般形式。
line getLine(double k, point u)
{
    // 直线与 X 轴不垂直，系数 b 规定为 1。
    return line(k, -1.0, u.y - k * u.x);
}

```

有时为了程序处理上的便利，可以使用直线与 X 轴之间的角度以及两个点来表示直线。这种表示形式可以按需要进行极角排序，例如在半平面交的排序增量算法中就利用了这种形式来表示直线，这样便于算法的实现。

```

struct line {
    point a, b;
    double angle;
};

```

强化练习：184 Laser Lines^B，905 Tacos Panchita^E，10167 Birthday Cake^B。

13.2.2 直线间关系

平面上的两条直线之间的关系有三种：相交、平行、重合。首先考察平行和重合的情况。若两条直线平行，则其斜率相等，但是截距不等。若两条直线重合，则其标准形式的三个系数均相等。由于判断实数相等时因浮点数表示的缘故可能存在误差，需要采用“系数差值的绝对值小于某个阈值即认为相等”的方式进行判断。

```

const double EPSILON = 1e-7;

// 判断两条直线是否平行。
bool parallel(line p, line q)
{
    return (fabs(p.a * q.b - q.a * p.b) < EPSILON);
}

// 判断两条直线是否重合。
bool same(line p, line q)
{
    return parallel(p, q) && (fabs(p.c - q.c) < EPSILON);
}

```

当两条直线相交时，经常需要求其交点。在求交点之前，预先判断两条直线是否平行或重合，如果平行，则不存在交点，如果重合，则处处是交点。再具体解题时，需要根据特定情况进行取舍。

```

struct point {
    double x, y;
    point (double x = 0, double y = 0): x(x), y(y) {}
}

```

```

};

// 求两条不同直线的交点。
bool getIntersection(line p, line q, point &pi)
{
    // 判断两条直线是否平行, 如果平行则无交点。
    if (fabs(p.a * q.b - q.a * p.b) < EPSILON) return false;
    pi.x = (q.c * p.b - p.c * q.b) / (p.a * q.b - q.a * p.b);
    pi.y = (q.c * p.a - p.c * q.a) / (p.b * q.a - q.b * p.a);
    return true;
}

```

强化练习: 217 Radio Direction Finder^E, 303 Pipe^D, 378 Intersecting Lines^A, 11068 An Easy Task^B。

扩展练习: 609 Metal Cutting^D, 10566 Crossed Ladders^B。

13.2.3 相互垂直的两条直线交点

给定点 p 和一条不过点 p 的直线 L , 要求确定经过点 p 与直线 L 垂直的直线 L' 与直线 L 的交点 p' 。若直线 L 与 X 轴平行 (斜率为 0) 或与 X 轴垂直时 (斜率为无穷大), 属于特殊情形, 交点的坐标容易计算。若直线 L 的斜率不为 0 或不为无穷大, 则直线 L' 的斜率与直线 L 的斜率乘积为 -1 。若直线 L 表示为

$$y = mx + b$$

则直线 L' 可表示为

$$y = -\frac{x}{m} + b'$$

```
const double EPSILON = 1E-6;
```

```

// 使用一般形式来表示直线。
struct line {
    double a, b, c;
    line (double a = 0, double b = 0, double c = 0): a(a), b(b), c(c) {}
};

// 求经过点 po 与直线 p 垂直的直线 q 与直线 p 的交点。
void getIntersection(point po, line p, point &pi)
{
    // 若 p 为平行于 X 轴的直线。
    if (fabs(p.a) <= EPSILON) { pi.x = po.x, pi.y = -p.c; }
    // 若 p 为平行于 Y 轴的直线。
    else if (fabs(p.b) <= EPSILON) { pi.x = -p.c, pi.y = po.y; }
    // 其他情形。
    else getIntersection(p, getLine(p.b / p.a, po), pi);
}

```

在后续章节会介绍使用投影的概念来求解垂直直线的交点问题, 因为此时的交点即为某条直线上的一点在另外一条直线上的投影。

13.3 坐标和坐标系变换

对于常用的直角坐标系, 如果将给定的坐标 (系) 进行平移、旋转、缩放操作, 得到的新坐标 (系) 和原坐标 (系) 具有特定的数学关系, 可以用于计算进行指定操作后的坐标值^[159]。

13.3.1 平移

坐标平移相当于赋予原坐标一个偏移分量(t_x, t_y), 其中 t_x 表示 X 轴方向的偏移, t_y 表示 Y 轴方向的偏移, 假设原坐标为(x, y), 变换后的坐标为(x', y'), 有

$$x' = x + t_x, \quad y' = y + t_y$$

坐标平移操作可以叠加, 最后的偏移分量为各次偏移分量的和。

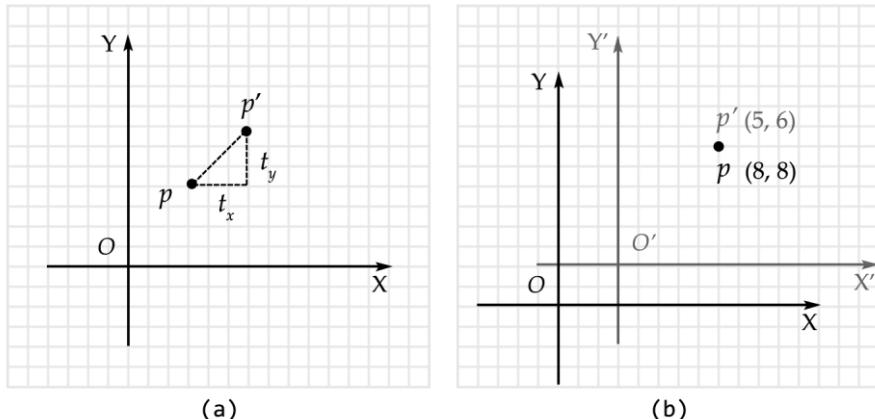


图 13-1 (a) 坐标平移, 点 p 通过平移, 移动到 p' , 偏移分量为(t_x, t_y); (b) 坐标系平移, 新坐标系 O' 是原坐标系 O 的原点移动到点(3, 2)得到, 在原坐标系中的点 $p(8, 8)$ 在新坐标系中的坐标为 $p'(5, 6)$

如果是坐标系平移, 令旧坐标系下的坐标为(x, y), 新坐标系下的坐标为(x', y'), 初始时, 新旧坐标系的原点重合, 此时两个坐标系下的坐标值是相同的, 如果新坐标系的原点平移到旧坐标系的点(t_x, t_y), 以点(t_x, t_y)为原点的新坐标系的坐标(x', y')与旧坐标系坐标(x, y)之间的关系为:

$$x' = x - t_x, \quad y' = y - t_y$$

$$x = x' + t_x, \quad y = y' + t_y$$

若令

$$T = \begin{bmatrix} 1 & 0 & -t_x \\ 0 & 1 & -t_y \\ 0 & 0 & 1 \end{bmatrix}, \quad T^{-1} = \begin{bmatrix} 1 & 0 & t_x \\ 0 & 1 & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

则可以使用齐次矩阵将坐标变换关系表示为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = T \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = T^{-1} \times \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

其中的 T 和 T^{-1} 互为逆矩阵。

强化练习: 10466 How Far^C, 11314* Hardly Hard^D, 11498 Division of Nlogonia^A。

13.3.2 旋转

坐标的旋转一般相对于坐标系原点进行, 以逆时针方向给出旋转的角度 θ , 变换后的坐标为

$$x' = x\cos\theta - y\sin\theta, \quad y' = x\sin\theta + y\cos\theta$$

若坐标的旋转不是相对于原点, 可以先将旋转中心平移到原点, 进行坐标旋转操作后再将旋转中心平移到起始的坐标, 在此过程中, 对需要变换的坐标进行相应的平移操作即可。坐标旋转操作可以叠加, 最后的旋转

角度为各次旋转角度的和。

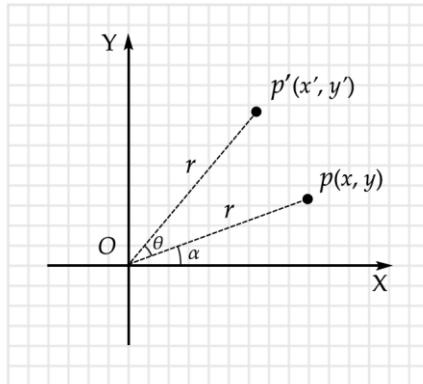


图 13-2 坐标旋转后原有坐标和当前坐标之间关系的推导。从极坐标的角度，可以得到以下的关系：

$$x' = r \cos(\alpha + \theta)$$

$$y' = r \sin(\alpha + \theta)$$

根据三角函数恒等式：

$$\cos(\alpha + \theta) = \cos \alpha \cos \theta - \sin \alpha \sin \theta$$

$$\sin(\alpha + \theta) = \sin \alpha \cos \theta + \cos \alpha \sin \theta$$

易得：

$$x' = r \cos(\alpha + \theta) = r \cos \alpha \cos \theta - r \sin \alpha \sin \theta = x \cos \theta - y \sin \theta$$

$$y' = r \sin(\alpha + \theta) = r \sin \alpha \cos \theta + r \cos \alpha \sin \theta = x \sin \theta + y \cos \theta$$

如果是坐标系旋转，令旧坐标系下的坐标为 (x, y) ，新坐标系下的坐标为 (x', y') ，初始时，新旧坐标系的 X 轴和 Y 轴重合，若新坐标系的 X 轴绕原点逆时针旋转角度 θ ，则新旧坐标系坐标之间的关系为

$$x' = x \cos \theta + y \sin \theta, \quad y' = -x \sin \theta + y \cos \theta$$

$$x = x' \cos \theta - y' \sin \theta, \quad y = x' \sin \theta + y' \cos \theta$$

若令

$$R = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad R^{-1} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

则可以使用齐次矩阵将坐标变换关系表示为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = R \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = R^{-1} \times \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

其中的 R 和 R^{-1} 互为逆矩阵。

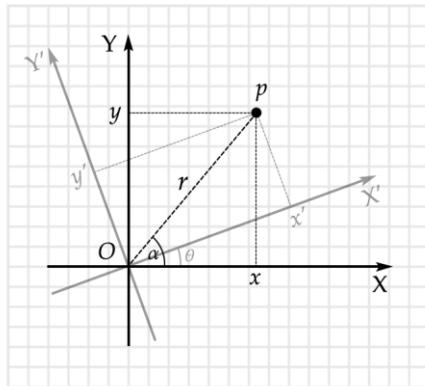


图 13-3 坐标系旋转后原有坐标和当前坐标之间关系的推导。从极坐标的角度，可以得到以下的关系：

$$\begin{aligned}x' &= r \cos(\alpha - \theta) \\y' &= r \sin(\alpha - \theta)\end{aligned}$$

根据三角函数恒等式：

$$\begin{aligned}\cos(\alpha - \theta) &= \cos \alpha \cos \theta + \sin \alpha \sin \theta \\\sin(\alpha - \theta) &= \sin \alpha \cos \theta - \cos \alpha \sin \theta\end{aligned}$$

易得：

$$\begin{aligned}x' &= r \cos(\alpha - \theta) = r \cos \alpha \cos \theta + r \sin \alpha \sin \theta = x \cos \theta + y \sin \theta \\y' &= r \sin(\alpha - \theta) = r \sin \alpha \cos \theta - r \cos \alpha \sin \theta = y \cos \theta - x \sin \theta\end{aligned}$$

在正方形网格上的坐标旋转有一个非常有用 的结论：给定正方形网格上的一个三角形，三角形的三个顶点均位于格点上，以三角形的任意一个顶点作为圆心进行旋转操作，只有当转过的角度值为 90 度的整数倍时，旋转后的三角形的三个顶点才可能仍然位于格点上。

强化练习：11505 Logo^C，11507 Bender B. Rodriguez Problem^A，11894 Genius MJ^D。

曼哈顿距离与切比雪夫距离

给定直角坐标系中的两个点 $A(x_a, y_a)$ 和 $B(x_b, y_b)$ ，两者之间的曼哈顿距离（Manhattan distance）定义为

$$d_M = |x_a - x_b| + |y_a - y_b|$$

而切比雪夫距离（Chebyshev distance）定义为

$$d_C = \max(|x_a - x_b|, |y_a - y_b|)$$

有时在题目中会要求进行如下的查询：给定直角坐标系中的某个点，需要确定与该点曼哈顿距离不大于 L 的所有点中，某个点所关联的量的最大（小）值。由于与某个点曼哈顿距离不大于 L 的所有点构成的区域在直角坐标系中所构成的是一个菱形区域，在表示上并不方便，为了处理上的便利，一般将其旋转 45 度，将其转换为一个矩形区域便于处理。

扩展练习：1105 Coffee Central^D，12074 The Ultimate Bamboo Eater^E。

13.3.3 缩放

如果是坐标缩放，令缩放前的坐标为 (x, y) ，缩放后的坐标为 (x', y') ，在 X 轴上的缩放比率为 s_x ，在 Y 轴方向上的缩放比率为 s_y ，则缩放前后的坐标关系为

$$x' = x \cdot s_x, \quad y' = y \cdot s_y$$

缩放可以叠加，最后的缩放比率为各次缩放比率的积。

如果是坐标系缩放，令旧坐标系下的坐标为 (x, y) ，新坐标系下的坐标为 (x', y') ，新坐标在 X 轴方向缩放比率为 s_x ，在 Y 轴方向缩放比率为 s_y ，则新旧坐标系坐标之间的关系为

$$x' = x \cdot \frac{1}{s_x}, \quad y' = y \cdot \frac{1}{s_y}$$

$$x = x' \cdot s_x, \quad y = y' \cdot s_y$$

若令

$$S = \begin{bmatrix} \frac{1}{s_x} & 0 & 0 \\ 0 & \frac{1}{s_y} & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad S^{-1} = \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

则可以使用齐次矩阵将坐标变换关系表示为

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = S \times \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = S^{-1} \times \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix}$$

其中的 S 和 S^{-1} 互为逆矩阵。

329 PostScript Emulation^E (PostScript 模拟)

PostScript 是一种被广泛用于激光打印机的页面描述语言。在 PostScript 中，进行度量的基本单位是点 (point)，默认每英寸等于 72 点。在 PostScript 程序刚开始运行的时候，它使用的默认坐标系为常见的笛卡尔坐标系，坐标系原点（横坐标和纵坐标均等于 0 的点）位于页面的左下角。

在本问题中，需要你识别 PostScript 语言的一个子集，具体来说，以下列表中所给出的命令必须能够得到识别。所有命令均以小写字母给出。如果命令中包含单词 “number”，表示该命令包含一个数值，该数值会以浮点小数的形式给出，可能包含正负号或者小数点。如果命令中出现两个数值，第一个数值表示 X 坐标，第二个数值表示 Y 坐标。简便起见，每条命令占据一行，在命令的要素之间以空格分隔。每条命令之后紧接着一个行结束符，如果某行的第一个字符是 ‘*’，表示输入结束。

你需要按照输入的顺序来处理所有命令。

number rotate

number 表示角度的大小。该命令要求将当前坐标系以当前原点为中心逆时针旋转 number 所指定的角度。该命令不会影响已经绘制在页面上的图形。

例如：‘90 rotate’ 表示将当前坐标系以当前原点为中心逆时针旋转 90 度，之后 Y 轴的正方向将指向左侧，X 轴的正方向将指向上方。

number number translate

将坐标系的原点移动到指定的坐标点(number, number)，坐标值相对于当前坐标系的原点给出。

例如：‘612 792 translate’ 表示将当前坐标系的原点移动到页面的右上角。进行此操作后，在可见页面显示的点所对应的坐标将具有负的 X 坐标值和负的 Y 坐标值（假定页面的大小为 8.5×11 英寸且页面方向为纵向）。

number number scale

将 X 坐标和 Y 坐标分别缩放指定的比率。该命令的实际效果相当于将物体在 X 坐标上和 Y 坐标上的大小分别乘以相应的缩放因子。缩放会影响之后在此方向上的任何操作，不管坐标系在后续过程中是否继续进行了旋转操作。

例如：‘3 2 scale’ 的效果将使得从(0, 0)到(72, 72)绘制的线段“变形”为一起点为页面的左下角原点，终点为距离页面左侧 3 英寸同时距离页面底部 2 英寸的点——的线段。假定在开始执行该命令前使用的是默认的初始坐标系。

number number moveto

将当前点移动到指定的坐标点。

例如：‘72 144 moveto’ 将会把当前点移动到距离页面左侧 1 英寸、距离页面底部 2 英寸的点。假定当前坐标系为默认的初始坐标系。

number number rmoveto

此命令类似于‘moveto’，不同之处在于命令‘rmoveto’中给出的坐标是新的当前点坐标与旧的当前点坐标的差值。

例如：‘144 -36 rmoveto’ 将会把前一个示例中设置的当前点坐标移动到距离页面左侧 2 英寸且在页面底边下方二分之一英寸的点，当前点的坐标会变成(216, 108)。注意，命令中的相对坐标值可能为负值！

number number lineto

从当前点坐标到指定坐标(*number, number*)绘制一条线段。指定坐标(*number, number*)将成为新的当前点坐标。

例如：‘216 144 lineto’ 将会在当前点坐标和指定坐标(216, 144)之间绘制一条线段，并将坐标(216, 144)设置为新的当前点坐标。如果使用前一示例所设定的当前点坐标，将会绘制一条从(216, 108)到(216, 144)的线段，或者说一条半英寸长的垂直线段。

number number rlineto

此命令与‘lineto’类似，不同之处在于给定的是相对坐标值。在绘制结束后，线段的终点将成为新的当前点坐标。

例如：‘0 144 rlineto’ 将会从当前点坐标出发，沿着 X 轴方向向右 2 英寸作为终点绘制一条线段。如果使用前一个示例所设定的当前点坐标，这将会从(216, 144)到(216, 288)绘制一条线段并将当前点坐标设置为(216, 288)。

输入与输出

你的任务是读入一个小型的 PostScript 程序，然后在不使用‘rotate’、‘translate’、‘scale’命令的情况下，重新生成一个 PostScript 程序，这个新的程序能够产生和原有程序一样的绘制效果。换句话说，对于输入中出现的每条‘moveto’、‘rmoveto’、‘lineto’、‘rlineto’命令，你需要按照它们在输入中出现的顺序将其显示在输出中（很可能需要将命令中的数值予以相应调整以保证绘制效果相同），但是不能使用原输入中出现的‘rotate’、‘translate’、‘scale’命令来达到同样的绘制效果，而且不能使用输入中未出现的其他命令。假定在程序开始执行时使用的是默认的坐标系。在生成的具有同等效果的程序中，命

令所使用的数值至少需要精确到小数点后两位。

样例输入

```
300 300 moveto
0 72 rlineto
0 0 rlineto
0 0 rlineto
0 rotate
2 1 scale
36 0 rlineto
1 -4 scale
0 18 rlineto
1 -0.25 scale
0.5 1 scale
300 300 translate
90 rotate
0 0 moveto
0 72 rlineto
2 1 scale
36 0 rlineto
1 -4 scale
0 18 rlineto
*
```

样例输出

```
300 300 moveto
0 72 rlineto
72 0 rlineto
0 -72 rlineto
300 300 moveto
-72 0 rlineto
0 72 rlineto
72 0 rlineto
```

分析

本题相当于编写一个“迷你版”的PostScript语言解析器。题目给定的三种变换均属于坐标系变换，每进行一次坐标系变换，原有坐标和当前坐标都可以使用一个系数矩阵予以关联。进行多次的坐标系变换，相等于将这些系数矩阵进行矩阵乘操作。以原有坐标为基准，每变换到一个新坐标系，将相应变换所对应的系数矩阵与原有坐标相乘后即可得到新坐标。例如，令初始坐标为 (x_0, y_0) ，假设进行了三种坐标系变换，它们对应的系数矩阵分别为 A 、 B 、 C ，则当前坐标和原有坐标的关系依次为

$$\begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = A \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = B \begin{bmatrix} x_A \\ y_A \\ 1 \end{bmatrix} = BA \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}, \quad \begin{bmatrix} x_C \\ y_C \\ 1 \end{bmatrix} = C \begin{bmatrix} x_B \\ y_B \\ 1 \end{bmatrix} = CBA \begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix}$$

根据可逆矩阵的定义，有

$$\begin{bmatrix} x_0 \\ y_0 \\ 1 \end{bmatrix} = (CBA)^{-1} \begin{bmatrix} x_C \\ y_C \\ 1 \end{bmatrix}$$

而

$$(CBA)^{-1} = A^{-1}B^{-1}C^{-1}$$

因此可知，将各个系数矩阵的逆矩阵按序相乘即可得到将当前坐标还原为初始坐标的系数矩阵¹。

参考代码

```
// 将当前坐标还原为初始坐标的系数矩阵。
double im[3][3] = {
    {1, 0, 0},
```

¹ 也可以先求系数矩阵的积，然后再求其逆矩阵。但求矩阵的逆矩阵涉及求矩阵对应行列式的值及其伴随矩阵，较为繁琐，不如直接利用逆矩阵的积来获得结果来得简便。

```

    {0, 1, 0},
    {0, 0, 1}
};

// 矩阵乘法。
void multiply(double rm[3][3])
{
    double tmp[3][3];
    for (int i = 0; i < 3; i++)
        for (int j = 0; j < 3; j++) {
            tmp[i][j] = 0;
            for (int k = 0; k < 3; k++)
                tmp[i][j] += im[i][k] * rm[k][j];
        }
    memcpy(im, tmp, sizeof(tmp));
}

// 将当前坐标还原为初始坐标。
pair<double, double> restore(double x, double y)
{
    double nx = im[0][0] * x + im[0][1] * y + im[0][2];
    double ny = im[1][0] * x + im[1][1] * y + im[1][2];
    return make_pair(nx, ny);
}

const double PI = 2.0 * acos(0);

int main(int argc, char *argv[])
{
    string line, parameter;
    double cx = 0, cy = 0;
    while (getline(cin, line), line != "*") {
        vector<string> cmd;
        istringstream iss(line);
        while (iss >> parameter) cmd.push_back(parameter);
        // 坐标系平移变换。
        if (cmd.back() == "translate") {
            double tx = stod(cmd[0]);
            double ty = stod(cmd[1]);
            double rm[3][3] = {
                {1, 0, tx},
                {0, 1, ty},
                {0, 0, 1}
            };
            multiply(rm);
            cx -= tx, cy -= ty;
        // 坐标系旋转变换。
        } else if (cmd.back() == "rotate") {
            double alpha = stod(cmd.front()) * PI / 180.0;
            double rm[3][3] = {
                {cos(alpha), -sin(alpha), 0},
                {sin(alpha), cos(alpha), 0},
                {0, 0, 1}
            };
            multiply(rm);
            double nextx = cx * cos(alpha) + cy * sin(alpha);

```

```

        double nexty = -cx * sin(alpha) + cy * cos(alpha) ;
        cx = nextx, cy = nexty;
// 坐标系缩放变换。
} else if (cmd.back() == "scale") {
    double sx = stod(cmd[0]);
    double sy = stod(cmd[1]);
    double rm[3][3] = {
        {sx, 0, 0},
        {0, sy, 0},
        {0, 0, 1}
    };
    multiply(rm);
    cx /= sx, cy /= sy;
// 以绝对坐标进行移动和绘制线段。
} else if (cmd.back() == "moveto" || cmd.back() == "lineto") {
    cx = stod(cmd[0]);
    cy = stod(cmd[1]);
    pair<double, double> r = restore(cx, cy);
    cout << fixed << setprecision(6) << r.first << ' ';
    cout << fixed << setprecision(6) << r.second << ' ';
    cout << cmd.back() << '\n';
// 以相对坐标进行移动和绘制线段。
} else if (cmd.back() == "rmoveto" || cmd.back() == "rlineto") {
    pair<double, double> r1 = restore(cx, cy);
    cx += stod(cmd[0]);
    cy += stod(cmd[1]);
    pair<double, double> r2 = restore(cx, cy);
    cout << fixed << setprecision(6) << (r2.first - r1.first) << ' ';
    cout << fixed << setprecision(6) << (r2.second - r1.second) << ' ';
    cout << cmd.back() << '\n';
}
}
return 0;
}

```

强化练习：316 Stars^D，979 The Abominable Triangleman^E。

扩展练习：197 Cube^D，10206 Stars^E。

13.4 三角形

三角形是由三条直线段构成的几何图形，它有三个顶点，对应的有三个内角，内角和为 180 度。给定三角形的三条边长 a , b , c ，可以证明，如果满足条件

$$a + b > c, \quad a + c > b, \quad b + c > a$$

则边长为 a , b , c 的三角形存在且唯一。根据三角形边长的关系，可以将三角形分为等边三角形 (equilateral triangle)，等腰三角形 (isosceles triangle)，不等边三角形 (scalene triangle)。等边三角形的三个内角相等，均为 60 度，等腰三角形的两个底角相等。

强化练习：11479 Is This the Easiest Problem^A，11936 The Lazy Lumberjacks^A。

有两种常见的单位来表示角的大小，一种是弧度 (radians)，另外一种是角度 (degree)。在平面上，角的范围是 0 到 2π 弧度，或者说是 0 到 360 度。在角度中，比度更小的单位分别是分 (minutes，表示 1/60 度) 和秒 (seconds，表示 1/60 分或 1/3600 度)。角度与弧度可以通过圆周率 π 为中介进行相互转换。

```
const double PI = 2.0 * acos(0.0);
```

```
// 角度转换为弧度。
double degreeToRadians(double degree) { return degree / 180.0 * PI; }
// 弧度转换为角度。
double radiansToDegree(double radians) { return radians / PI * 180.0; }
```

扩展练习：849 Radar Tracking^E。

13.4.1 勾股定理

如果三角形的一个内角为直角 (right angle)，则称此三角形为直角三角形。按约定，对应于直角的边称为斜边，其他两条边称为对边和邻边。

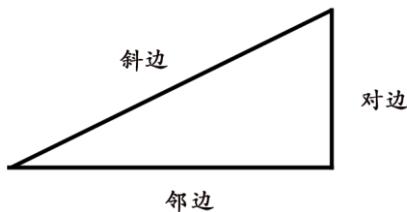


图 13-4 直角三角形的三条边

直角三角形满足勾股定理 (Pythagorean theorem)^I。令斜边，对边，邻边的边长分别为 c, a, b ，则各边的长度有以下关系

$$a^2 + b^2 = c^2$$

当任意给定直角三角形的两条边的长度，通过勾股定理可以计算另外一条边的长度。

106 Fermat vs. Pythagoras^A (费马与毕达哥拉斯)

给定正整数 n ，编写程序统计满足等式

$$x^2 + y^2 = z^2$$

且均小于 n 的正整数 x, y, z 相关的两个量。一个量是三元组 (x, y, z) 的数量，要求 $x < y < z$ ，且 x, y, z 互素，即它们的最大公约数为 1。另外一个量是在小于等于 n 的正整数 p 中，不属于前述的任意一个三元组中 x, y, z 的正整数总数。

输入

输入包含一系列的正整数 n ，每行一个。 n 不大于 1000000。文件结束表示输入结束。

输出

对于输入中的每个整数 n ，输出一行，打印以空格间隔的两个整数。第一个整数表示互素且满足等式的三元组数量 (三元组中的整数要求小于等于 n)。第二个整数表示小于等于 n 的正整数中，不属于前述任意一个三元组的整数数量。

样例输入

样例输出

^I 根据现有记载，西方最早证明此定理的为公元前 6 世纪左右古希腊的毕达哥拉斯 (Pythagoras)，他用演绎法证明了直角三角形斜边平方等于两直角边平方之和，因此在西方，“勾股定理”称为“毕达哥拉斯定理”。我国之前亦将“勾股定理”称为“毕达哥拉斯定理”，这是当时随着西方数学传入时翻译所采用的名称。20 世纪 50 年代，学术界曾展开过关于这个定理命名的讨论，最后决定使用“勾股定理”这个名称，得到教育界和学术界的普遍认同。

10
25
100

1 4
4 9
16 27

分析

本题的要求是当 $x, y, z \in \mathbb{N}$, 给定一个数 n , 找出所有的 $\max\{x, y, z\} \leq n$, 使得 $x^2 + y^2 = z^2$ 成立。如果使用朴素的穷尽搜索, 生成 1000000 以内所有的勾股数而后进行筛选, 显然会超出时间限制, 故需要考虑其他更为高效的方法。如果方程存在一个通解, 那么根据通解生成 x, y, z 则效率要高很多, 有没有这样的通解公式呢? 答案是肯定的。下面进行通解公式的推导。

先假定 x, y, z 互素, 若不互素, 则可令 $x=w \times x_0, y=w \times y_0, z=w \times z_0$, 其中 w 为三者的最大公约数, 将其转化为互素的情形后讨论。由于 x, y, z 互素, 则 x, y 中至少有一个是奇数。下面用反证法证明 x 和 y 中有且只有一个奇数。

假定 x, y 都为奇数, 令

$$x = 2a + 1, y = 2b + 1, a \geq 0, b \geq 0$$

则有

$$x^2 + y^2 = (2a + 1)^2 + (2b + 1)^2 = 4(a^2 + b^2 + a + b) + 2 = z^2$$

也就是说, z^2 必定是偶数, 若 z^2 为偶数, 则 z 必为偶数, 那么 z^2 必能被 4 整除, 但上式中 z^2 除以 4 后余数为 2, 产生矛盾, 因此假设不成立, 即 x, y 中只有一个奇数。

假设 x 为奇数, y 为偶数, 由于奇数的平方是奇数, 偶数的平方是偶数, 则 z^2 必为奇数, 故 z 为奇数, 那么 $z+x$ 和 $z-x$ 都是偶数。不妨设 $z+x=2u, z-x=2v$, 解得

$$z = u + v, x = u - v$$

而且由于 x, y, z 互素, 则 u, v 也必定互素, 若不互素, 则可设 $u=w \times u_0, v=w \times v_0$, 则 z 和 x 有大于 1 的公约数 w , 与前提条件产生矛盾。

将原方程两边同除以 4 得

$$\frac{x^2}{4} + \frac{y^2}{4} = \frac{z^2}{4}$$

移项, 将 u, v 代入, 得

$$\left(\frac{y}{2}\right)^2 = \left(\frac{z}{2}\right)^2 - \left(\frac{x}{2}\right)^2 = \frac{(z+x)}{2} \times \frac{(z-x)}{2} = u \times v$$

也就是说 $u \times v$ 是一个平方数, 又因为 u, v 互素, 所以 u 和 v 本身都是平方数^I。令 $u=a^2, v=b^2$, 则 a, b 同样也是一奇一偶且互素的两个数^{II}。代入 u 和 v , 解得

$$x = u - v = a^2 - b^2, y = 2ab, z = u + v = a^2 + b^2$$

其中 a 与 b 互素, $a > b$, 且一奇一偶。

题目要求统计 (x, y, z) 三元组的数量时只统计 x, y, z 互素的情形, 用上述的通解公式即可解决。

^I 由于 u 和 v 互素, 且 $u \times v$ 为平方数, 设 $u \times v = k^2$, 则由于 $\gcd(u, v) = 1$, 所以 $u = u \times \gcd(u, v) = \gcd(u^2, u \times v) = \gcd(u^2, k^2) = (\gcd(u, k))^2$, 同理 $v = (\gcd(v, k))^2$ 。

^{II} 因为 u 和 v 互素, 则必有一个奇数, 又由于 y 为偶数, 则 u 和 v 不能同为奇数, 故必是一奇一偶。由于奇数的平方是奇数, 偶数的平方是偶数, 则 a 和 b 也是一奇一偶, 若 a 和 b 不互素, 可推出 u 和 v 不互素, 产生矛盾。

对于统计 p 的数量，可采用下述方法：所有非互素的 x_0, y_0, z_0 构成的三元组均可由一组互素的 x, y, z 乘以系数得到，利用通解公式，可以预生成所有的勾股数并标记已经使用的整数，最后统计未使用的整数即为 p 的数量。

强化练习：[10773 Back to Intermediate Math^A](#), [11854 Egypt^A](#), [12843 Disputed Claims^E](#)。

13.4.2 三角函数

常用的三角函数有正弦 (sine)、余弦 (cosine)、正切 (tangent)。正弦和余弦的取值在一 1 和 1 之间。它们的定义如下：

$$\sin(A) = \frac{|\text{对边}|}{|\text{斜边}|}, \quad \cos(A) = \frac{|\text{邻边}|}{|\text{斜边}|}, \quad \tan(A) = \frac{|\text{对边}|}{|\text{邻边}|}$$

三角函数有自身的反函数，例如：反正弦 (arcsin)、反余弦 (arccos)、反正切 (arctan)。它们的作用是将给定的三角函数值映射为相应的角度值。由于三角函数库中的各个三角函数并不是数值稳定的，因此角度 A 的正弦值再取反正弦不一定和 A 相等，即 A 和 $\arcsin(\sin(A))$ 不一定精确相等。

以下是若干在解题中常用的三角恒等公式。

$$\begin{aligned} \sin(\alpha \pm \beta) &= \sin \alpha \cos \beta \pm \cos \alpha \sin \beta, \quad \cos(\alpha \pm \beta) = \cos \alpha \cos \beta \mp \sin \alpha \sin \beta \\ \tan(\alpha \pm \beta) &= \frac{\tan(\alpha) \pm \tan(\beta)}{1 \mp \tan(\alpha) \tan(\beta)}, \quad \tan(\alpha) \pm \tan(\beta) = \frac{\sin(\alpha \pm \beta)}{\cos(\alpha) \cos(\beta)} \\ \sin(\alpha) + \sin(\beta) &= 2 \sin \frac{(\alpha + \beta)}{2} \cos \frac{(\alpha - \beta)}{2}, \quad \sin(\alpha) - \sin(\beta) = 2 \cos \frac{(\alpha + \beta)}{2} \sin \frac{(\alpha - \beta)}{2} \\ \cos(\alpha) + \cos(\beta) &= 2 \cos \frac{(\alpha + \beta)}{2} \cos \frac{(\alpha - \beta)}{2}, \quad \cos(\alpha) - \cos(\beta) = -2 \sin \frac{(\alpha + \beta)}{2} \sin \frac{(\alpha - \beta)}{2} \end{aligned}$$

正弦余弦的 n 倍角公式：

$$\sin(n\alpha) = n \cos^{n-1} \alpha \sin \alpha - C_n^3 \cos^{n-3} \alpha \sin^3 \alpha + C_n^5 \cos^{n-5} \alpha \sin^5 \alpha - \dots$$

$$\cos(n\alpha) = \cos^n \alpha - C_n^2 \cos^{n-2} \alpha \sin^2 \alpha + C_n^4 \cos^{n-4} \alpha \sin^4 \alpha - \dots$$

$$\cos(n\alpha) = \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} (-1)^m C_n^{2m} \cos^{n-2m}(\alpha) \sin^{2m}(\alpha)$$

$$\cos(n\alpha) = \sum_{m=0}^{\lfloor \frac{n}{2} \rfloor} (-1)^m C_n^{2m} \cos^{n-2m}(\alpha) \left(\sum_{k=0}^m (-1)^k C_m^k \cos^{2k}(\alpha) \right)$$

$$\sin(n\alpha) = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} C_{n-1-k}^k (-1)^k 2^{n-1-2k} \cos^{n-1-2k}(\alpha) \sin(\alpha)$$

$$\cos(n\alpha) = \sum_{k=0}^{\lfloor \frac{n}{2} \rfloor} (C_{n-k}^k + C_{n-1-k}^{k-1}) (-1)^k 2^{n-1-2k} \cos^{n-2k}(\alpha)$$

强化练习：[203 Running Lights Visibility Calculator^E](#), [11909 Soya Milk^B](#), [12901 Refraction^E](#)。

扩展练习：[11170 Cos\(NA\)^D](#)。

13.4.3 正弦定理

正弦定理 (law of Sines) 描述的是三角形三条边和它们对角之间的关系。令三角形的三个角分别为 A, B, C ，其对边分别为 a, b, c ，则有以下关系

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

需要注意, 给定两个角和其中一个角的对边, 可以根据正弦定理计算另外一个角的对边, 但是给定两条边和一条边的对角, 使用正弦定理计算另外一条边的对角时, 有可能会得出错误的结果, 因为 $\sin(x) = \sin(\pi - x)$, 同一正弦值可能对应两个角度, 应该使用后续介绍的余弦定理计算。

强化练习: [10286 Trouble with a Pentagon^A](#)。

13.4.4 余弦定理

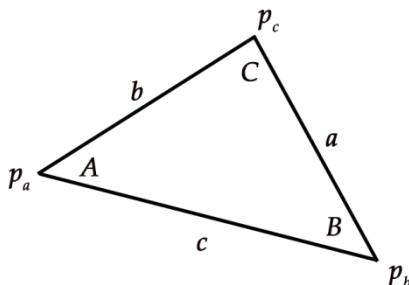


图 13-5 三角形的顶点和边、角的约定命名方式

如图 13-5 所示, 令三角形的三个顶点为 p_a , p_b , p_c , 相应的三个角为 A , B , C , 角的对边分别为 a , b 和 c 。余弦定理 (law of Cosines) 指出, 存在以下等式

$$a^2 = b^2 + c^2 - 2bc\cos A, \quad b^2 = a^2 + c^2 - 2ac\cos B, \quad c^2 = a^2 + b^2 - 2ab\cos C$$

余弦定理在计算三角形的夹角时非常有用, 也可以很容易确定转角是锐角还是钝角, 在某些情况可以简化问题的处理。

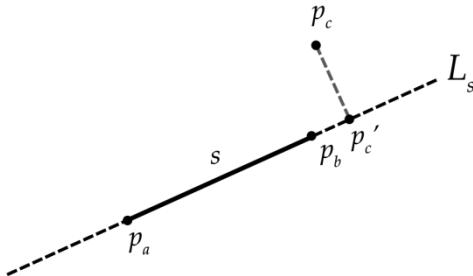


图 13-6 判断点的投影是否在线段上

例如, 如图 13-6 所示, 给定一条线段 s , 其端点分别为点 p_a 和点 p_b , 直线 L_s 经过线段 s , 点 p_c 为直线 L_s 外的一点, 那么点 p_c 在直线 L_s 上的垂直投影点 p_c' 是否在线段 s 上?

朴素的方法是先由线段 s 的两个端点求出直线 L_s 的方程, 然后求出经过点 p_c 与 L_s 垂直的直线 L_p 与直线 L_s 的交点 p_c' , 然后判断交点 p_c' 是否在线段 s 上。方法虽然直观, 但是代码量却不少, 根据题意, 实际上只需要判断 $\angle p_a p_b p_c$ 和 $\angle p_b p_a p_c$ 是否均为锐角 (或两者之一为直角) 即可。

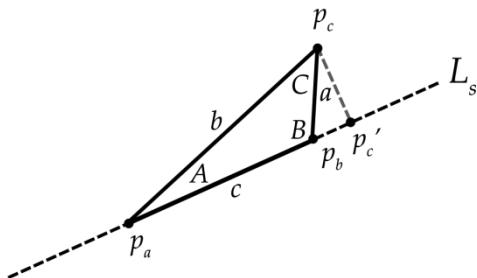


图 13-7 利用余弦定理判断点的投影是否在线段上

如图 13-7 所示, 以 p_a , p_b , p_c 为三角形的三个顶点, 令点 p_a 所在角为 A , 对边为 a , 点 p_b 所在角为 B , 对边为 b , 点 p_c 所在角为 C , 对边为 c , 则 $\angle p_a p_b p_c$ 即为角 B , $\angle p_b p_a p_c$ 即为角 A , 根据余弦定理

$$\cos A = \frac{b^2 + c^2 - a^2}{2bc}, \cos B = \frac{a^2 + c^2 - b^2}{2ac}$$

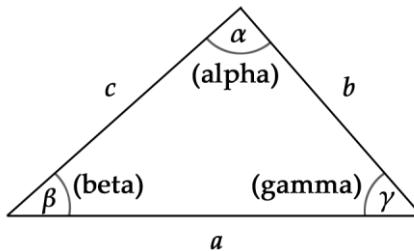
只要满足 $\cos A \geq 0$ 而且 $\cos B \geq 0$, 则 $\angle p_a p_b p_c$ 和 $\angle p_b p_a p_c$ 均为锐角 (或两者之一为直角), 亦即

$$(b^2 + c^2 - a^2) \geq 0, (a^2 + c^2 - b^2) \geq 0$$

换句话说, 只需要计算各点间的距离即可进行判定。

194 Triangle^D (三角形)

三角形在平面几何中是一种基本图形。它由三条边和三个夹角构成。下图所示是常见的给边和角命名的方式:



翻开任何一本关于几何的书, 都会看到存在许多关于三角形的等式, 如

$$\alpha + \beta + \gamma = \pi, \frac{a}{\sin \alpha} = \frac{b}{\sin \beta} = \frac{c}{\sin \gamma}$$

$$a = b \cos \beta + c \cos \gamma, a^2 = b^2 + c^2 - 2bc \cos \alpha$$

$$\frac{a - b}{a + b} = \tan \frac{\alpha - \beta}{2} / \tan \frac{\alpha + \beta}{2}$$

给定 a , b , c , α , β , γ 的值, 这六个参数完全定义了一个三角形, 只要给定足够多的参数, 其他的参数就能够通过公式推导得出。

现在要求你编写程序通过给定的六个参数的子集计算缺失的参数。对于某些参数组合来说, 由于给定的参数数量太少导致无法计算其他参数, 或者给定的参数将导致一个非法的三角形。一个合法的三角形的边长大于 0, 内角大于 0 且小于 π 。你的程序需要能够检测参数非法的情形并且输出 “**Invalid input.**” 当给定的参数数量多于计算需要, 但是给定参数和计算得到的其他参数之间互相矛盾时也应输出 “**Invalid input.**”, 例如, 当所有三个内角均已给出, 但是内角和大于 π 。

某些参数组合可能导致出现多种解，但是解的数量有限，对于这种情况，你的程序应该输出“**More than one solution.**”。

对于所有其他的情况，你的程序应该计算缺失的参数，然后输出所有六个参数。

输入

输入的第一行包含一个整数提示参数的组数。接下来的每一行包含六个实数，以单个空格分隔。给定的数是分别表示六个参数 a , α , b , β , c , γ 的值。参数按照题图所标记。如果参数的值为 -1 表示相应的参数尚未定义，需要通过计算得到。所有的浮点数至少包含 8 位有效数字。

输出

你的程序应该为输入中的每一组参数输出一行。如果能够得到唯一的合法三角形解，你的程序应该输出六个参数 a , α , b , β , c , γ ，相互间隔一个空格。否则输出“**More than one solution.**”或者“**Invalid input.**”。

在进行输出时，每个参数至少应该包含 6 位有效数字，因此你的程序在计算过程中至少应该精确到小数点后 6 位（比如说，相对误差 0.000001 是允许的）。

样例输入

```
4
47.9337906847 0.6543010109 78.4455517579 1.4813893731 66.5243757656 1.0059022695
62.72048064 2.26853639 -1.00000000 0.56794657 -1.00000000 -1.00000000
15.69326944 0.24714213 -1.00000000 1.80433105 66.04067877 -1.00000000
72.83685175 1.04409241 -1.00000000 -1.00000000 -1.00000000 -1.00000000
```

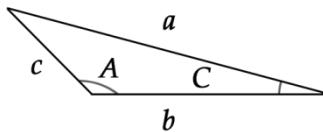
样例输出

```
47.933791 0.654301 78.445552 1.481389 66.524376 1.005902
62.720481 2.268536 44.026687 0.567947 24.587225 0.305110
Invalid input.
Invalid input.
```

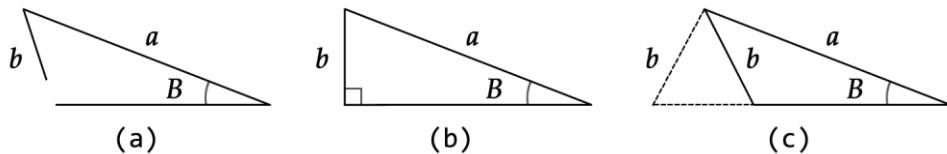
分析

该题是对三角形边长关系、内角和关系、正弦定理、余弦定理的综合运用。以下逐一分析参数和解的情况。

- (1) 当给定的参数小于等于两个时，无法计算其他参数。
- (2) 当给定三条边时，可以唯一确定一个三角形，此时需要检查边长是否满足“任意两条边长之和大于第三边”的性质，如果满足，则可以应用余弦定理继续计算其他参数，否则是一个非法三角形。
- (3) 当给定三个内角时，如果三个内角之和与 π 之间的差值大于指定的误差，则认为是一个非法三角形，否则若只给定了三个内角而没有给定至少一条边，存在无限多的三角形满足要求，当给定了至少一条边时，可通过正弦定理计算其他两条边的边长（若给定了至少两条边长，可通过余弦定理计算剩余一条边的边长）。
- (4) 当给定两条边的边长和其夹角时，可以由余弦定理计算其他参数。注意在计算得到第三条边长后，不能应用正弦定理来计算其他内角，因为在 C++ 的三角函数库中反正弦函数的值域为 $[-\pi/2, \pi/2]$ ，对计算得到的正弦值取反正弦，无法得到钝角。如图 13-8 所示，角度有可能为钝角，所以可能会得出错误结果。

图 13-8 给定边长 a 和 b 以及夹角 C , 可以通过余弦定理计算边长 c , 但是此时不能通过正弦定理计算角度 A

(5) 当给定两条边和一条边的对角时, 首先判断所给角是否小于 π 弧度, 若小于 π 弧度, 分三种情况进行判断: 无解、唯一解、两种解, 具体如图 13-9 所示。

图 13-9 给定边长 a 和 b 以及 b 的对角 B , 可能有三种情形。(a) 无解; (b) 唯一解; (c) 两种解

计算得到其他参数后, 与输入的参数进行比对, 如果误差大于 10^{-6} 则表明不符合要求, 是非法三角形, 否则按照要求输出计算得到的六个参数。

强化练习: 132 Bumpy Objects^C, 267 Of(f) Course^E, 376 More Triangles THE AMBIGUOUS CASE^D, 10195 The Knights of the Round Table^A, 12301 An Angular Puzzle^D。

扩展练习: 10734* Triangle Partitioning^D。

13.4.5 三角形面积

令三角形的三个顶点分别为 $a(a_x, a_y)$, $b(b_x, b_y)$, $c(c_x, c_y)$, 则三角形的有向面积 $A(T)$ 可以使用三阶行列式表示为

$$A(T) = \frac{1}{2} \begin{vmatrix} a_x & a_y & 1 \\ b_x & b_y & 1 \\ c_x & c_y & 1 \end{vmatrix} = \frac{1}{2} (a_x b_y - a_y b_x + b_x c_y - c_x b_y + c_x a_y - c_y a_x)$$

请注意, 有向面积具有符号, 在返回三角形的实际面积前需要取其绝对值。

若给定了三角形的三条边长, 亦可通过海伦公式 (Heron's formula) 计算其面积^I。令 a , b , c 为三条边的边长, 三角形面积 S 可以表示为

$$S = \sqrt{p(p-a)(p-b)(p-c)}, \quad p = \frac{a+b+c}{2}$$

10347 Medians^A (中线)

给定三角形的三条中线的长度, 确定原三角形的面积大小。三角形的中线是指连接某个顶点和其对边的中点所得到的直线段。三角形共有三条中线。

输入

共有 1000 行输入数据, 每行输入包含三个数值, 表示中线的长度, 这些数值均不超过 100, 文件结束

^I 参阅: https://en.wikipedia.org/wiki/Heron%27s_formula, 2020。

即为输入结束。

输出

对于每行输入输出一个数值，表示三角形的面积大小。如果使用给定的中线长度无法构成三角形，则输出‘-1’。如果存在满足要求的三角形，输出其正的面积数值，四舍五入到小数点后三位。

样例输入

3 3 3

样例输出

5.196

分析

第一种解法：三角形的中线交点具有一个性质，即交点和底边中点的距离为该底边中线长度的 $1/3$ 。

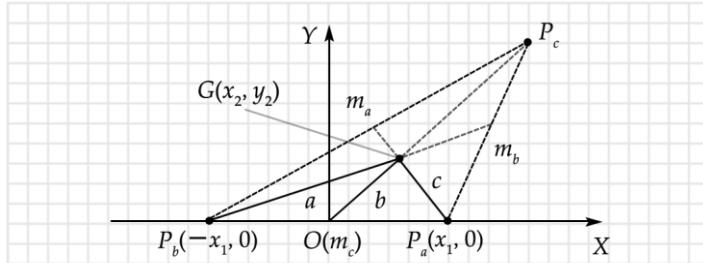


图 13-10 题目约束示意图

如图 13-10 所示，设三条中线的交点为 $G(x_2, y_2)$ ，则有以下方程成立

$$(x_2 + x_1)^2 + y_2^2 = a^2, \quad x_2^2 + y_2^2 = b^2, \quad (x_2 - x_1)^2 + y_2^2 = c^2, \quad x_1 > 0, \quad y_1 > 0$$

联立三个方程可解得

$$x_1 = \sqrt{\frac{a^2 - 2b^2 + c^2}{2}}, \quad x_2 = \frac{a^2 - c^2}{4x_1}, \quad y_2 = \sqrt{b^2 - x_2^2}, \quad a^2 - 2b^2 + c^2 > 0, \quad b^2 - x_2^2 > 0$$

由于 G 是中线 P_cO 的三分点，可得顶点 P_c 的坐标为

$$P_c \cdot x = 3x_2, \quad P_c \cdot y = 3y_2$$

进一步可由顶点 P_a 和 P_c 的坐标得到中点 m_b 的坐标为

$$m_b \cdot x = \frac{x_1 + P_c \cdot x}{2}, \quad m_b \cdot y = \frac{P_c \cdot y}{2}$$

则中线 $P_b m_b$ 的长度为

$$|P_b m_b| = \sqrt{(m_b \cdot x + x_1)^2 + (m_b \cdot y)^2}$$

同理可得中线 $P_a m_a$ 的长度为

$$|P_a m_a| = \sqrt{(m_a \cdot x - x_1)^2 + (m_a \cdot y)^2}$$

最后，根据题目所给条件，判断由上述方法计算得到的中线长度 $P_b m_b$ 和 $P_a m_a$ 和给定的中线长度是否相等（误差在指定范围内）。如果相等，则可构成合法三角形，计算面积即可。

第二种解法：设中线长度分别为 a, b, c ，可以证明，如果三条中线长度的 $2/3$ 能够构成三角形，即满足

$$a + b > c, \quad a + c > b, \quad b + c > a$$

则给定的中线能够成为某个三角形的合法中线，且这个三角形的面积是三条中线长度的 $2/3$ 所构成的三角形

的三倍。

另外，需要注意特殊的测试数据，如三条中线中有若干条中线的长度为 0 的情形。

强化练习：1249 Euclid^D，10522 Height to Area^C，11164 Kingdom Division^D。

扩展练习：11579 Triangle Trouble^C。

13.4.6 三角函数库

C++标准库提供了相应的三角函数用以计算。在使用这些函数前，需要包含头文件<cmath>。

```
#include <cmath>

double sin(double x);           // 返回弧度 x 的正弦值
double cos(double x);           // 返回弧度 x 的余弦值
double tan(double x);           // 返回弧度 x 的正切值
double asin(double x);          // 返回正弦值 x 所对应的角度弧度值
double acos(double x);          // 返回余弦值 x 所对应的角度弧度值
double atan(double x);          // 返回正切值 x 所对应的角度弧度值
double atan2(double y, double x); // 返回正切值 y/x 所对应的角度弧度值
```

asin 与 acos

由于正弦和余弦的值域为[−1, 1]，当给定的值绝对值大于 1 时，对其取反正切或反余弦会得到非数值（Not A Number, NAN）。因此，在解题过程中对值运用三角函数 asin 或 acos 之前，需要检查给定的值是否在相应的三角函数的值域范围内，否则很可能会导致错误的计算结果。

atan 与 atan2

atan 与 atan2 均为反正切函数，但有不同。atan 接受一个正切值，返回其角度值，单位为弧度，返回值范围为[− $\pi/2$, $\pi/2$]。atan 不区分角所在的象限，若在第一象限和第三象限各有一个角，它们的正切值均为正，使用 atan 函数返回的角度值都在第一象限。atan2 接受两个参数，表示某点坐标的 y 值和 x 值，返回该点在对应极坐标系中的极角，单位亦为弧度，返回值范围为[− π , π]。atan2 返回的角度值考虑了点所在的象限，位于第一象限和第三象限的角虽然其正切都为正，但是返回角度值不在同一象限。注意当 y 和 x 同时为 0 时，正切值无确切定义，因此某些标准库实现中 atan2 的两个参数不能同时为 0，否则会报运行时错误。以下是使用 GCC 5.4.1 编译运行的结果，此编译器的三角函数库实现中 atan2 允许两个参数同时为 0，其返回的角度值为 0 弧度。

```
-----13.4.6.cpp-----//
int main(int argc, char *argv[])
{
    cout << fixed << setprecision(6);
    cout << atan(0.0) << endl;           // 0.000000
    cout << atan(1E20) << endl;          // 1.570796
    cout << atan(-1E20) << endl;         // -1.570796
    cout << atan2(0, 1) << endl;          // 0.000000
    cout << atan2(1, 0) << endl;          // 1.570796
    cout << atan2(-1, 0) << endl;         // -1.570796
    cout << atan2(0, -1) << endl;          // 3.141593
    cout << atan2(-1E-10, -1) << endl;    // -3.141593
    cout << atan2(0, 0) << endl;          // 0.000000
    return 0;
```

```

}
//-----13.4.6.cpp-----//

```

强化练习: 206 Meals on Wheels Routing System^D, 10792 The Laurel-Hardy Story^D, [10927 Bright Lights^C](#), [11326 Laser Pointer^D](#)。

扩展练习: [1709* Amalgamated Artichokes^C](#), [10372* Leaps Tall Buildings^D](#), [11437 Triangle Fun^C](#), [11574* Colliding Traffic^D](#)。

13.4.7 桌球碰撞问题

给定长为 $2a$ 宽为 $2b$ 的台球桌面, 以台球桌面的中心为原点, 在坐标 (x, y) 处有一颗母球, 当时刻 0 时, 以角度 A (以原点为端点的向右水平射线与击球方向在逆时针方向上的夹角) 向右击球, 其初速度为 v , 加速度为 s , 试确定母球在时刻 t 时已经撞击台球横边和纵边的次数。为简化问题, 将母球视为一个质点, 碰撞均为弹性碰撞, 即母球不会损失能量, a, b, x, y, A, v, s, t 均为整数, 且 $a > 0, b > 0, -a < x < a, -b < y < b, 0 \leq A < 360, v > 0, s > 0, t \geq 0$ 。

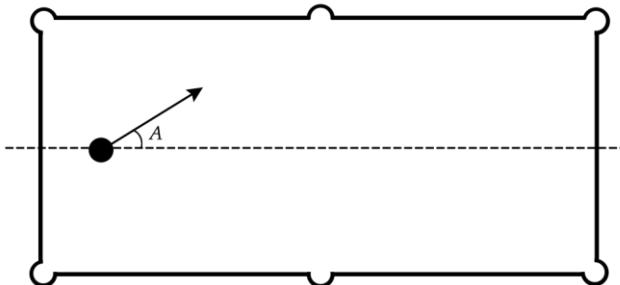


图 13-11 桌球碰撞问题

由弹性碰撞的性质可知, 母球在撞击台球桌边缘时, 入射角等于出射角, 可以利用对称性并结合三角函数来计算撞击次数。将台球行进的路线延长, 同时将台球的实际撞击路线位移到延长线上, 可以看到, 行进方向的位移在 X 轴上的投影恰为台球在水平方向上移动的距离, 在 Y 轴上的投影恰为台球在垂直方向上移动的距离。

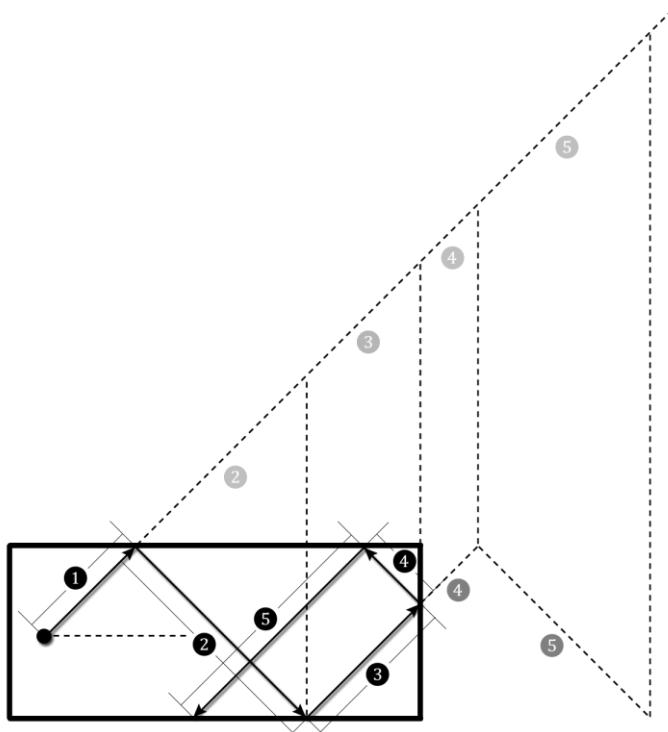


图 13-12 将桌球的移动轨迹通过对称变换移动到以出发点为端点的射线上可以简化问题的处理。也就是说，桌球与台球桌进行多次弹性碰撞后，在 X 轴和 Y 轴方向所移动的距离可以通过桌球沿出发方向移动的距离在 X 轴和 Y 轴上的投影获得

由此可以得到如下的求解代码：

```
-----13.4.7.cpp-----
const double PI = 2 * acos(0);

pair<int, int> collide(int a, int b, int x, int y, int A, int v, int s, int t)
{
    // 根据均变速运动的位移公式得到台球在击球方向的移动距离。
    double d = (v + s * t) * t / 2.0;
    // 确定投影在 X 轴和 Y 轴方向上的移动距离。
    double dx = fabs(d * cos(A * PI / 180.0)), dy = fabs(d * sin(A * PI / 180.0));
    // ch 为撞击横边的次数, cv 为撞击纵边的次数。
    int ch = 0, cv = 0;
    if (A >= 0 && A <= 90) {
        ch = (y + b + dy) / (2 * b);
        cv = (x + a + dx) / (2 * a);
    }
    else if (A > 90 && A <= 180) {
        ch = (y + b + dy) / (2 * b);
        cv = (a - x + dx) / (2 * a);
    }
    else if (A > 180 && A <= 270) {
        ch = (b - y + dy) / (2 * b);
        cv = (a - x + dx) / (2 * a);
    }
}
```

```

    }
    else if (A > 270 && A < 360) {
        ch = (b - y + dy) / (2 * b);
        cv = (x + a + dx) / (2 * a);
    }
    return make_pair(ch, cv);
}
//-----13.4.7.cpp-----

```

强化练习: 10387 Billiard^B, [11130 Billiard Bounces^C](#)。

扩展练习: 10881* Piotr's Ants^B。

13.5 多边形

13.5.1 矩形

矩形是常见的几何对象, 正方形则是特殊的矩形。正方形的边长相等, 对角线和边长的比为 $\sqrt{2}$ 。可以使用以下结构体来表示矩形:

```

struct rectangle {
    // l 为矩形的长度, w 为矩形的宽度。
    double l, w;
    rectangle (double l = 0, double w = 0): l(l), w(w) {}
    double perimeter() { return 2 * (l + w); }
    double area() { return l * w; }
};

```

如果给定矩形的对角顶点坐标, 如何求两个矩形的交 (两个矩形的重叠部分) 呢?

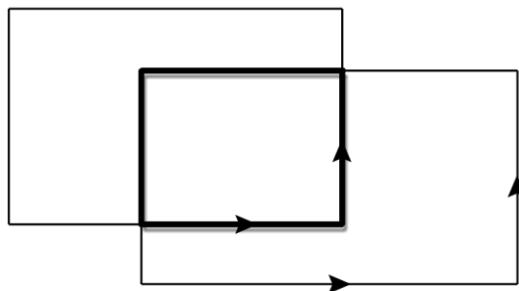


图 13-13 矩形的交

如图 13-13 所示, 将矩形的边按照边所在直线的极角标记方向。可以发现, 如果两个矩形存在公共部分, 则公共矩形各边所在的直线总是箭头所指方向上位于更左侧的那条直线。根据这个性质, 可以容易地求出两个矩形的交。对于长方体的交, 可以按照类似的方法进行。

```

struct point {
    int x, y;
};

struct rectangle {
    point leftLower, rightTop;
};

```

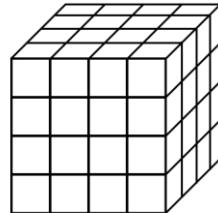
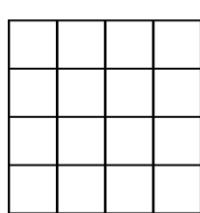
```

rectangle getAnd(rectangle r1, rectangle r2)
{
    int lowx = max(r1.leftLower.x, r2.leftLower.x);
    int lowy = max(r1.leftLower.y, r2.leftLower.y);
    int upx = min(r1.rightTop.x, r2.rightTop.x);
    int upy = min(r1.rightTop.y, r2.rightTop.y);
    if (lowx >= upx || lowy >= upy) return rectangle{point{0, 0}, point{0, 0}};
    return rectangle{point{lowx, lowy}, point{upx, upy}};
}

```

10177 (2/3/4)-D Sqr/Rects/Cubes/Boxes?^B (2/3/4-维立方体?)

以下给出了一个 4×4 的方阵，你能数出其中包含多少个正方形或长方形吗（假定长方形不等同于正方形）？也许你可以用掰手指的方式数出来，但是如果给定的是 100×100 或者是 10000×10000 的方阵呢？如果是在更高的维度，你还能数得过来吗？你能数出一个 $10 \times 10 \times 10$ 的立方体中包含有多少个大小不同的正方体或长方体吗？或者，你能数出一个 $5 \times 5 \times 5 \times 5$ 的四维超立方体中包含多少个超立方体或者超长方体吗？注意，你的程序必须非常高效。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 方阵和 $4 \times 4 \times 4$ 立方体

输入

输入中每行包含一个整数 N ($0 \leq N \leq 1000$)，表示方阵、立方体或超立方体的边长。图示所给定的示例中， $N=4$ ，输入至多包含 100 行。

输出

对于每行输入，输出一行共六个整数 $S_2, R_2, S_3, R_3, S_4, R_4$ ，其中 S_2 表示二维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 对应三维和四维的情形。

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

```

分析

设边长为 n ($n \geq 1$)，则有以下结果：

(1) 正方形数量 S_2 形成的数列后项与前项差为 n^2 ，其通项公式为

$$S_2^n = 1^2 + 2^2 + \dots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{n^3}{3} + \frac{n^2}{2} + \frac{n}{6}$$

(2) 立方体数量 S_3 形成的数列后项与前项差为 n^3 ，其通项公式为

$$S_3^n = 1^3 + 2^3 + \cdots + n^3 = \left(\frac{n(n+1)}{2}\right)^2 = (1+2+\cdots+n)^2$$

(3) 超立方体数量 S_4 形成的数列后项与前项差为 n^4 , 其通项公式为

$$S_4^n = 1^4 + 2^4 + \cdots + n^4$$

(4) 长方形数量 R_2 的通项公式为

$$R_2^n = S_3^n - S_2^n$$

(5) 长方体数量 R_3 的通项公式为

$$R_3^n = S_2^n S_1^n, \quad S_1^n = \frac{(n-1)(n+2)}{2}$$

(6) 超长方体数量 R_4 的通项公式为

$$R_4^n = \left(\frac{n(n+1)}{2}\right)^4 - S_4^n$$

强化练习: 311 Packets^A, 460 Overlapping Rectangles^B, 737 Gleaming the Cubes^B, 1587 Box^B, 10215 The Largest/Smallest Box^C, 10250 The Other Two Trees^B, 11207 The Easiest Way^B, 11345 Rectangles^C, 11639 Guard the Land^C, 13215 Polygonal Park^D。

扩展练习: 308 Tin Cutter^D, 1721* Window Manager^E。

13.5.2 四边形和正多边形

在解题中常见的四边形为菱形; 常见的多边形有正五边形, 正六边形。正 n 边形的内角相等, 对于正 n ($n \geq 3$) 边形, 由于可将其剖分为 $n-2$ 个三角形, 故其内角和为 $(n-2) \times 180$ 度。

强化练习: 10432 Polygon Inside A Circle^A, 11455 Behold My Quadrangle^A, 12300 Smallest Regular Polygon^D。

扩展练习: 11648* Divide the Land^D, 12256* Making Quadrilaterals^D。

13.6 圆

圆 (circle) 是平面上与定点 $c(x, y)$ 距离为 r 的所有点的集合, 可以将圆表示为

```
struct circle {
    double x, y, r;
    circle (double x = 0, double y = 0, double r = 0): x(x), y(y), r(r) {};
```

在几何上, 圆可以使用如下两种方式表示, 一种是使用圆心坐标 (a, b) 及半径 R 来表示, 称之为标准方程, 即

$$(x - a)^2 + (y - b)^2 = R^2$$

另外一种表示方法是将以上表示形式展开, 称之为一般方程, 即

$$x^2 + y^2 + Dx + Ey + F = 0$$

圆的周长和直径有一个固定的比值—— π , 它是一个无理数, 可以通过展开以下泰勒级数得到

$$\pi = 4 \times \sum_{k=1}^{\infty} \frac{(-1)^{k+1}}{2k-1} = 4 \times \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots\right)$$

π 的 32 位有效数字近似值为: 3.1415926535897932384626433832795, 使用该近似值完全可以满足解题需要。如果不想去记忆这个常数, 也可以通过对特定值取反三角函数来得到 π 的值。

```
// const double PI = 4.0 * atan(1.0);
const double PI = 2.0 * acos(0.0);
```

实际上，圆是椭圆的一种特殊形式，而椭圆是圆锥曲线的一种，是圆锥与平面的截线。椭圆可以定义为平面内到定点 f_1 、 f_2 的距离之和等于常数 $2a$ ($2a > |f_1f_2|$) 的动点 p 的轨迹， f_1 、 f_2 称为椭圆的两个焦点。两个焦点的距离 $|f_1f_2| = 2c < 2a$ 称为椭圆的焦距，椭圆截与两焦点连线重合的直线所得的弦为长轴，长为 $2a$ ，椭圆截垂直平分两焦点连线的直线所得弦为短轴，长为 $2b$ ，且

$$c^2 = a^2 - b^2, \quad b = \sqrt{a^2 - c^2}$$

强化练习：[190 Circle Through Three Points^A](#)，[356 Square Pegs And Round Holes^B](#)，[12611 Beautiful Flag^B](#)，[12704 Little Masters^A](#)，[12748 WiFi Access^D](#)。

13.6.1 圆的周长和面积

圆的周长 P_c 和面积 S_c 与圆的半径 r 之间的关系为

$$P_c = 2\pi r, \quad S_c = \pi r^2$$

其中的 π 为圆周率常数。可以使用代码表示为

```
const double PI = 2.0 * acos(0.0);
struct circle {
    double x, y, r;
    double perimeter() { return 2.0 * PI * r; }
    double area() { return PI * r * r; }
    // 圆上扇形的面积, a 为扇形的弧度。
    double areaOfSector(double a) { return r * r * a / 2.0; }
};
```

如图 13-14 所示，弦 (chord) AB 与圆心 O 的距离为 h ，如何计算图中阴影部分的面积呢？

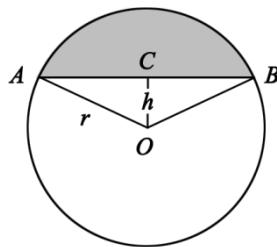


图 13-14 阴影面积的计算

可以根据反三角函数先计算 $\angle AOC$ 的大小，然后再根据阴影面积等于扇形 AOB 面积减去三角形 AOC 面积的两倍关系进行计算。

```
const double PI = 2.0 * acos(0.0);

// 通过反三角函数计算∠AOC, area 表示阴影部分的面积。
double angleOfAOC = acos(h / r), area = 0.0;

// 第一种方式：先计算扇形 AOB 的面积，然后减去两倍的三角形 AOC 面积即为阴影部分的面积。
area = PI * r * r * ((2.0 * angleOfAOC) / (2.0 * PI));
area -= 2.0 * (r * sin(angleOfAOC) * h / 2.0);

// 第一种方式的化简。
```

```

area = r * (r * angleOfAOC - sin(angleOfAOC) * h);

// 略有不同的第二种方式, 利用勾股定理计算直角三角形 AOC 的高 AC, 进而得到面积。
area = r * r * angleOfAOC - sqrt(r * r - h * h) * h;

```

对于椭圆来说, 令其长半轴的长度为 a , 短半轴的长度为 b , 其周长 P_e 和面积 S_e 与半轴的长度 a 和 b 之间的关系为

$$P_e = 2\pi b + 4(a - b), \quad S_e = \pi ab$$

其中的 π 为圆周率常数。可以使用代码表示为

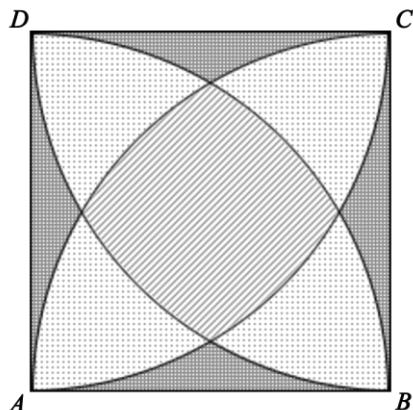
```

const double PI = 2.0 * acos(0.0);
struct ellipse {
    double a, b;
    double perimeter() { return 2.0 * PI * b + 4.0 * (a - b); }
    double area() { return PI * a * b }
};

```

10209 Is This Integration?^A (需要积分吗?)

给定正方形 $ABCD$, 以正方形的四个顶点为圆心, 以其边长 a 为半径绘制弧形, 将正方形区域划分为下图所示的三种不同类型, 试求各种不同类型区域的面积和。



输入

输入中每行包含一个浮点数 a ($0 \leq a \leq 10000$), 表示正方形的边长。输入以文件结束符表示结束。

输出

对于每行输入, 输出三个浮点数, 精确到小数点后 3 位, 分别表示正方形中斜线部分、打点部分、网格部分各自的面积和。

样例输入

0.1

样例输出

0.003 0.005 0.002

分析

设斜线部分的面积为 X , 打点部分的面积为 Y , 网格部分的面积为 Z , 则有以下方程

$$X + \frac{3Y}{4} + \frac{Z}{2} = \frac{\pi a^2}{4}, \quad X + Y + Z = a^2$$

要使方程有定解, 必须还有一个独立方程, 这可以由计算左下角和右下角两个四分之一圆重叠部分的面积来得到, 观察图形表示容易得知, 圆弧 BD 和 AC 的交点与顶点 A 、 B 构成等边三角形, 根据这个性质, 重叠部分的面积容易计算, 因此可以得到方程

$$X + \frac{Y}{2} + \frac{Z}{4} = \frac{\pi a^2}{3} - \frac{\sqrt{3}a^2}{4}$$

联立三个方程可解得

$$X = \left(1 - \sqrt{3} + \frac{\pi}{3}\right)a^2, \quad Y = \left(2\sqrt{3} - 4 + \frac{\pi}{3}\right)a^2, \quad Z = \left(4 - \sqrt{3} - \frac{2\pi}{3}\right)a^2$$

强化练习: 10221 Satellites^A, 10287 Gifts in a Hexagonal Box^C, 10451 Ancient Village Sports^A, 10589 Area^A, 10678 The Grazing Cow^A, 10991 Region^B, 12853 The Pony Cart Problem^D, 12578 10:6:2^A, 12894 Perfect Flag^C。

扩展练习: 1388 Graveyard^C, 10297 Beavergnaw^A, 10668 Expanding Rods^C, 11646 Athletics Track^C。

13.6.2 圆的切线

圆心在 (a, b) , 半径为 r 的圆方程为

$$(x - a)^2 + (y - b)^2 = r^2$$

设 A 为圆弧上的一个点, 其坐标为 (x_A, y_A) , 可以证明, 过 A 点与圆相切的直线方程为

$$(x_A - a)(x - a) + (y_A - b)(y - b) = r^2$$

设 T 为圆外的一个点, 其坐标为 (x_T, y_T) , 过点 T 与某个圆心为 $O(a, b)$ 半径为 r 的圆作切线, 可以作两条切线, 切线的交点可通过圆方程以及切线方程联立解得, 但在编程竞赛中, 由于方程解的表达式复杂, 一般不使用此种方法求切点坐标, 而是使用其他更为简便的方法^I。

415 Sunrise^D (日出)

本题要求计算在日出后的某个时刻, 你所能看到的太阳面积占整个太阳圆盘面积的百分比。本问题属于一颗行星和一颗恒星之间的双体问题。假设地球是一个理想球体, 半径为 3950 英里, 不考虑地球大气的折射对解题的影响。在离地心 92900000 英里的地方是一颗明亮的四等星, 也就是太阳, 忽略太阳上的大气以及行星绕恒星圆周运动的影响, 只需将太阳视为一个平面上的圆盘。太阳圆盘与连接地球中心和太阳中心的直线相垂直, 其半径为 432000 英里。假设地球以均匀的速度自转, 自转一周需要 24 小时。在地球绕太阳公转的过程中, 可以认为太阳的中心始终在地球赤道的上方。任意选定地球赤道上的一个参考点, 从第一缕阳光照到该参考点开始计时, 对于输入中给定的时刻, 计算太阳圆盘上能够照射到该参考点的盘面面积百分比。

输入

输入的时间是以秒为单位的浮点数。时间数据保证不小于 0 且不大于 600。每行包含一个浮点数, 在读入时应该使用 `float` 或 `double` 数据类型, 连续处理输入直到遇到文件结束符。

输出

对于每行输入均应生成一行输出。输出包含一个浮点数, 表示自第一缕阳光照到指定参考点后开始计时, 经过指定的秒数后, 太阳上能够照射到该点的圆盘部分占整个太阳圆盘面积的百分比。对于非 0 的答案, 要求误差在 0.1% 以内。如果输出 0, 要求计算得到的值在 $+/-0.001$ 以内。

^I 本书第 14 章“计算几何”第 14.2.13 小节“圆的切点”介绍了求切点坐标的两种常用方法。

样例输入	样例输出
0.0	0.000
600.0	1.000

分析

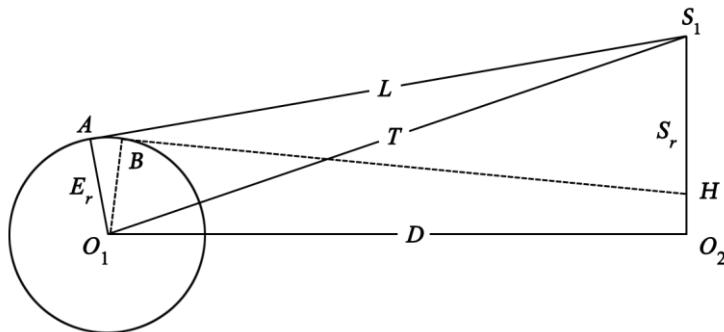


图 13-15 地球和太阳圆盘示意图（只示意了太阳圆盘的上半部分，并未按实际比例绘制）

如图 13-15 所示, 设地球所在圆 O_1 的圆心为坐标系原点, 太阳圆盘的圆心为 O_2 , 太阳圆盘的上边缘为 S_1 , 地球圆心与太阳圆盘圆心间的距离为 D , A 点是起始参考点, B 点是某个时刻, 参考点 A 移动到此处, 若要计算此时能够照射到 B 点的太阳盘面面积, 关键是求得 H 点的纵坐标。由于直线 BH 是圆 O_1 的切线, 故 BH 垂直于 BO_1 , 若能确定直线 BO_1 的斜率, 根据相互垂直直线的斜率乘积为 -1 的关系, 可以得到直线 BH 的斜率, 再根据直线 BH 经过点 B 和点 H , 若能求得 B 点的坐标, 则 H 点的坐标由于已经知道了其横坐标为 D , 其纵坐标必定可求。令 B 点的坐标为 (x_B, y_B) , 则有

$$x_B = E_r \cos \angle BO_1 O_2, \quad y_B = E_r \sin \angle BO_1 O_2$$

关键是求得 $\angle BO_1O_2$ ，由示意图可知

$$\angle A O_1 O_2 = \angle A O_1 B + \angle B O_1 O_2$$

而 $\angle AOB$ 是经过指定的时间 t 之后地球所转过的角度，根据地球自转一周 (2π 弧度) 为 24 小时，则转动 t 时间的角度为

$$\angle A O_1 B = \frac{t \times 2\pi}{24h \times 3600s/h}$$

那么, 只需求得 $\angle AO_1O_2$ 即可得到 $\angle BO_1O_2$, 而观察图中边角关系可得

$$\angle A O_1 O_2 = \angle A O_1 S_1 + \angle S_1 O_1 O_2 = \tan^{-1} \frac{L}{E_r} + \tan^{-1} \frac{S_r}{D}$$

而

$$L = \sqrt{T^2 - E_r^2} = \sqrt{D^2 + S_r^2 + E_r^2}$$

令 H 点的坐标为 (D, h) , 由于圆 O_1 的方程为

$$x^2 + y^2 = {E_r}^2$$

则过圆 O_1 上点 B 的切线方程为

$$x_B x + y_B y = E_r^2$$

此切线同时经过点 $H(D, h)$, 将坐标代入得

$$x_B D + y_B h = E_r^2$$

则有

$$h = \frac{E_r^2 - x_B D}{y_B} = \frac{E_r - D \cos \angle BO_1 O_2}{\sin \angle BO_1 O_2}$$

得到 h 的值之后，即可按照前述介绍的计算圆上扇形面积的方法来计算太阳圆盘的面积。

参考代码

```
const double pi = 2 * acos(0);

int main(int argc, char *argv[])
{
    double Sr = 432000, Er = 3950, D = 92900000;
    double L = sqrt(D * D + Sr * Sr - Er * Er);
    double alpha = atan(L / Er) + atan(Sr / D);
    double seconds;

    while (cin >> seconds) {
        double h, beta = alpha - 2.0 * pi * seconds / 3600.0 / 24.0;
        h = (Er - D * cos(beta)) / sin(beta);
        if (h >= Sr) {
            cout << "0.000000\n";
            continue;
        }
        if (h <= -Sr) {
            cout << "1.000000\n";
            continue;
        }
        double percentage = acos(fabs(h) / Sr) / pi - fabs(h) *
            sqrt(Sr * Sr - h * h) / (pi * Sr * Sr);
        if (h < 0) percentage = 1.0 - percentage;
        cout << fixed << setprecision(6) << percentage << '\n';
    }

    return 0;
}
```

强化练习：10180 Rope Crisis in Ropeland^C。

扩展练习：313* Intervals^D，10136 Chocolate Chip Cookies^D。

13.6.3 三角形的内切圆与外接圆

三角形的内切圆（inscribed circle）是圆心位于三角形内且圆心与三条边的距离相等的圆。设内切圆的半径为 r ，三角形的面积为 S ，边长分别为 a, b, c ，由于内切圆的圆心位于三个内角平分线的交点上，根据海伦公式及面积关系有

$$r = \frac{2S}{a+b+c} = \frac{S}{p} = \frac{\sqrt{p(p-a)(p-b)(p-c)}}{p}, \quad p = \frac{a+b+c}{2}$$

设三角形顶点坐标为 $P_a(x_a, y_a), P_b(x_b, y_b), P_c(x_c, y_c)$ ，其内切圆的圆心坐标 (x, y) 为

$$x = \frac{ax_a + bx_b + cx_c}{a+b+c}, \quad y = \frac{ay_a + by_b + cy_c}{a+b+c}$$

其中 a, b, c 为三角形顶点 P_a, P_b, P_c 的对边长。

强化练习：375 Inscribed Circles and Isosceles Triangles^C。

扩展练习：11524 In-Circle^D。

三角形的外接圆 (escribed circle) 是指同时经过三角形三个顶点的圆。设外接圆的半径为 R , 由于外接圆的圆心位于三条边的垂直平分线的交点上, 根据海伦公式有

$$R = \frac{abc}{4S} = \frac{abc}{4\sqrt{p(p-a)(p-b)(p-c)}}, \quad p = \frac{a+b+c}{2}$$

设三角形顶点坐标为 $P_a(x_a, y_a)$, $P_b(x_b, y_b)$, $P_c(x_c, y_c)$, 由于外接圆的圆心 (x, y) 距离三个顶点的距离相等, 有

$$(x - x_a)^2 + (y - y_a)^2 = (x - x_b)^2 + (y - y_b)^2 = (x - x_c)^2 + (y - y_c)^2 = R^2$$

可化简为

$$2(x_b - x_a)x + 2(y_b - y_a)y = -x_a^2 - y_a^2 + x_b^2 + y_b^2$$

$$2(x_c - x_a)x + 2(y_c - y_a)y = -x_a^2 - y_a^2 + x_c^2 + y_c^2$$

解得圆心坐标为 (为了显示的简洁, 使用行列式来表示解)

$$x = \frac{1}{2} \times \frac{\begin{vmatrix} 1 & x_a^2 + y_a^2 & y_a \\ 1 & x_b^2 + y_b^2 & y_b \\ 1 & x_c^2 + y_c^2 & y_c \end{vmatrix}}{\begin{vmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{vmatrix}}, \quad y = \frac{1}{2} \times \frac{\begin{vmatrix} 1 & x_a & x_a^2 + y_a^2 \\ 1 & x_b & x_b^2 + y_b^2 \\ 1 & x_c & x_c^2 + y_c^2 \end{vmatrix}}{\begin{vmatrix} 1 & x_a & y_a \\ 1 & x_b & y_b \\ 1 & x_c & y_c \end{vmatrix}}$$

其中三阶行列式的定义为

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix} = a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{21}a_{32}a_{13} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

以下为代码实现:

```
//-----13.6.3.cpp-----
struct point {
    double x, y;
};

struct circle {
    double x, y, r;
    double distTo(point a) { return sqrt(pow(x - a.x, 2) + pow(y - a.y, 2)); }
};

circle getCircleFromTriangle(point &a, point &b, point &c)
{
    double A1 = a.x - b.x, B1 = a.y - b.y;
    double A2 = c.x - b.x, B2 = c.y - b.y;
    double C1 = (a.x * a.x - b.x * b.x + a.y * a.y - b.y * b.y) / 2;
    double C2 = (c.x * c.x - b.x * b.x + c.y * c.y - b.y * b.y) / 2;

    circle cc;
    cc.x = (C1 * B2 - C2 * B1) / (A1 * B2 - A2 * B1);
    cc.y = (A1 * C2 - A2 * C1) / (A1 * B2 - A2 * B1);
    cc.r = cc.distTo(a);

    return cc;
}
//-----13.6.3.cpp-----
```

强化练习: [438 The Circumference of the Circle^A](#), [10577 Bounding Box^B](#), [11152 Colourful Flowers^A](#),

11281 Triangular Pegs in Round Holes^D, 12302 Nine-Point Circle^D。

扩展练习: 11406* Best Trap^E, 11761 Super Heronian Triangle^E。

13.6.4 圆与圆的位置关系

圆与圆的位置关系可以分为四种: 重合、相离、相切、相交。如果给定两个圆相应的坐标和半径的三元组 (x_1, y_1, r_1) 、 (x_2, y_2, r_2) , 根据两个圆的坐标、圆心距离、半径和之间的关系, 可以区分以下几种情形判断两个圆的位置关系。

(1) 重合。如果两个圆的圆心距离为零, 可以认为两个圆的圆心重合。若两个圆的半径相等, 则可以认为两个圆相同。有一种特殊情形——圆心重合同时半径为零, 需要根据具体题目进行判断, 可以认为两个圆重合或者认为两个圆相交于圆心这一点。

// 误差阈值的设定需要根据具体题目进行设定。

```
const double epsilon = 1e-7;
// 圆心距离。
double d = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
if (d < epsilon && fabs(r1 - r2) < epsilon)
    cout << "THE CIRCLES ARE THE SAME" << endl;
```

(2) 相离。可能有两种情形: 一种是两个圆的圆心距大于两个圆的半径之和 (外离), 另外一种是半径较小的圆包含在半径较大的圆内 (内含)。

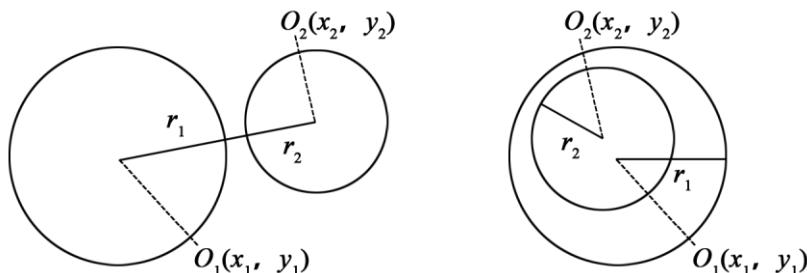


图 13-16 圆相离的两种可能情形: 外离和内含

此种情形可以使用下述代码进行判断:

```
const double epsilon = 1e-7;
double d = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
// 假设 r1 对应的总是大圆的半径, 如果在初始时, r1 小于 r2, 可以事先将其交换。
if (r2 > r1) swap(x1, x2), swap(y1, y2), swap(r1, r2);
if (d > r1 + r2 + epsilon || r1 > d + r2 + epsilon)
    cout << "NO INTERSECTION" << endl;
```

(3) 相切。两个圆相切, 可以分为外相切和内相切两种情形。

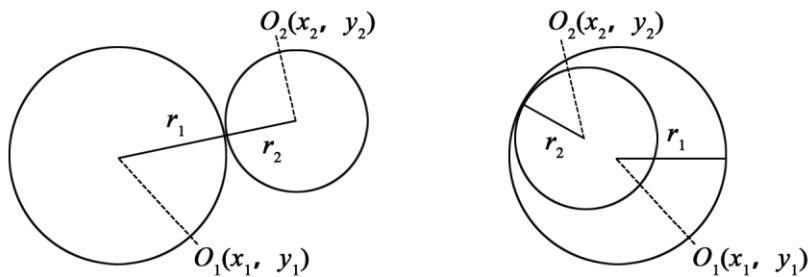


图 13-17 圆相切的两种可能情形

```

const double epsilon = 1e-7;
double d = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
// 两圆相切, 确定唯一交点的坐标。
double x3, y3;
// 外相切和内相切, 交点坐标的计算方式不同, 假设 r1 为大圆的半径, r2 为小圆的半径。
if (fabs(d - (r1 + r2)) < epsilon) {
    x3 = x1 + r1 / (r1 + r2) * (x2 - x1);
    y3 = y1 + r1 / (r1 + r2) * (y2 - y1);
}
else if (fabs(d - fabs(r1 - r2)) < epsilon) {
    x3 = x1 + r1 / d * (x2 - x1);
    y3 = y1 + r1 / d * (y2 - y1);
}

```

(4) 相交。两个圆相交, 有两个交点。

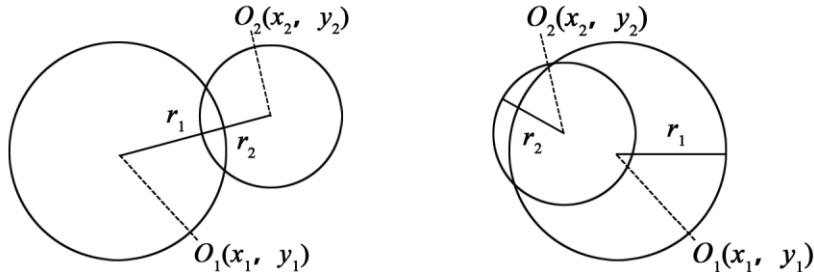


图 13-18 圆相交的两种可能情形

如果两个圆的圆心距离小于半径之和而且大于半径之差的绝对值即可认为两个圆相交。可根据相应的三角关系计算两个交点的坐标^I。

^I 本书第 14 章“计算几何”第 14.2.12 小节“圆和圆的交点”介绍了使用向量和余弦定理来求交点坐标的方法。

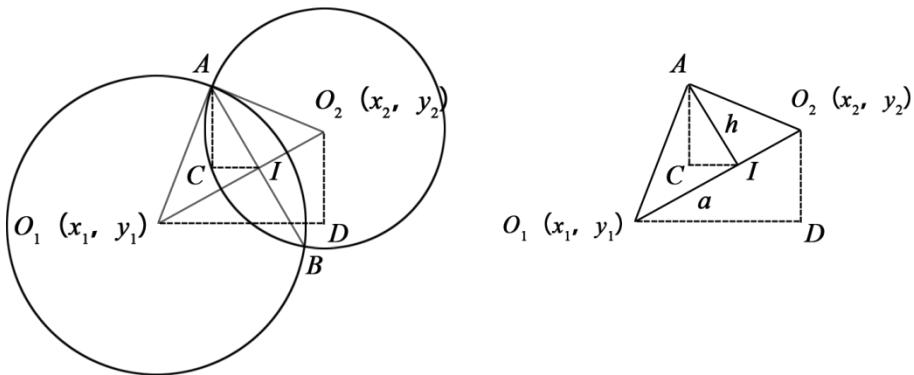


图 13-19 圆相交时交点坐标的计算

设两圆相交于 A 和 B , 线段 AB 和两个圆心的连线 O_1O_2 相交于 I 。首先计算交点 I 的坐标。根据余弦定理有

$$\overline{AO_2}^2 = \overline{AO_1}^2 + \overline{O_1O_2}^2 - 2\overline{AO_1}\overline{O_1O_2} \cos \angle A O_1 O_2$$

其中 AO_1 为圆 O_1 的半径, AO_2 为圆 O_2 的半径, O_1O_2 为两个圆心间的距离, O_1I 为圆 O_1 的圆心到交点 I 的距离, AI 为两圆交点 A 到两线段交点 I 的距离, 不妨设 $AO_1=r_1$, $AO_2=r_2$, $O_1O_2=d$, $O_1I=a$, $AI=h$, 则有

$$r_1 \cos \angle A O_1 O_2 = \frac{r_1^2 + d^2 - r_2^2}{2d} = \overline{O_1I} = a, \quad h = \sqrt{r_1^2 - a^2}$$

那么交点 $I(x_3, y_3)$ 的坐标为

$$x_3 = x_1 + \frac{a(x_2 - x_1)}{d}, \quad y_3 = y_1 + \frac{a(y_2 - y_1)}{d}$$

由于三角形 ACI 和三角形 O_1DO_2 为相似三角形, 则交点 $A(x_4, y_4)$ 的坐标为

$$x_4 = x_3 - \frac{h(y_2 - y_1)}{d}, \quad y_4 = y_3 + \frac{h(x_2 - x_1)}{d}$$

同理可求交点 $B(x_5, y_5)$ 的坐标为

$$x_5 = x_3 + \frac{h(y_2 - y_1)}{d}, \quad y_5 = y_3 - \frac{h(x_2 - x_1)}{d}$$

当圆相交的位置与上述情况不同时, 例如圆 O_2 处于圆 O_1 的左方或者下方, 或者两圆水平相交, 以上计算公式仍然是正确的。可将上述计算过程实现为以下代码。

```
const double epsilon = 1e-7;
double d = sqrt(pow(x1 - x2, 2) + pow(y1 - y2, 2));
double r1, r2, a, h, x3, y3, x4, y4, x5, y5;
// 圆心距离大于半径差的绝对值而且小于半径之和则两圆相交, 求两个交点的坐标。
if (d > (fabs(r1 - r2)) + epsilon && (d + epsilon) < (r1 + r2)) {
    a = (r1 * r1 + d * d - r2 * r2) / (2 * d);
    h = sqrt(r1 * r1 - a * a);
    x3 = x1 + a * (x2 - x1) / d;
    y3 = y1 + a * (y2 - y1) / d;
    x4 = x3 - h * (y2 - y1) / d;
    y4 = y3 + h * (x2 - x1) / d;
    x5 = x3 + h * (y2 - y1) / d;
    y5 = y3 - h * (x2 - x1) / d;
}
```

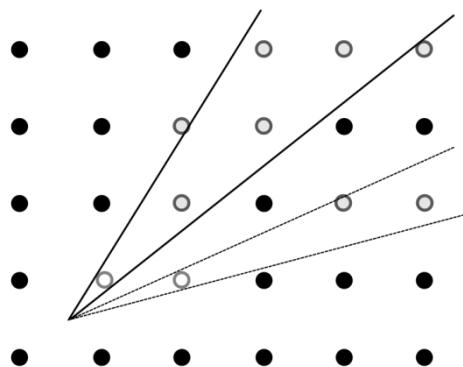
}

强化练习：[10301* Rings and Glue^B](#)，[11515 Cranes^D](#)。

149 Forest^D (森林)

俗话说“见树不见林”，不过这句话不仅是老生常谈，而且不太准确，实际情况是“因林难见树”。假如你站在树林中，由于树之间会相互遮挡，你实际能分辨清楚的树是非常少的。特别是当树按行列对齐进行栽种时（如人工林），这种效应更为明显，因为它们横竖都成直线，更容易发生互相遮挡的情况。本题的目标如下：在一片人工林中（假定人工林无限大），从任意一点向四周望去，确定你能分辨的树的数量。

如果某棵树的树干没有被更近的树所遮挡——要求树干的两侧都能被看见，即树干和更靠近你的树之间存在“可见间隙”——你才能看清这棵树。显然，当树离得太远而显得“太小”时你没法分辨清楚。严格来说，“不太小”和“可见间隙”表示树或间隙的视角需要大于0.01度（你可以假定本题只用一只眼睛进行观察）。因此下图中标记为白色圈的树遮挡了标记为灰色圈的树。



给定树的直径和观察点的坐标，编写程序，在上述条件下确定可见树木的数量。因为网格无限大，原点的位置并不重要。所有的坐标值均在0和1之间。

输入

输入包含多行，每行包括三个形如 $0.nm$ 的实数，第一个数表示树干的直径（diameter）——你可以假定树干均为圆柱形，树干中心恰好位于单位距离的网格格点上。后两个数为观察者的 x 坐标和 y 坐标。为了避免可能的问题，比如说观察者离树太近，输入保证直径满足条件： $diameter \leq x, y \leq (1 - diameter)$ 。为了避免树太小的问题，你可以假定 $diameter \geq 0.1$ 。输入以包含三个0的一行表示结束。

输出

输出由多行组成，每行输出对应一行输入。每行输出包含了在给定的树木尺寸和观察点下可见树木的数量。

样例输入

```
0.10 0.46 0.38
0 0 0
```

样例输出

```
128
```

分析

此题关键是确定两棵树是否互相遮挡。

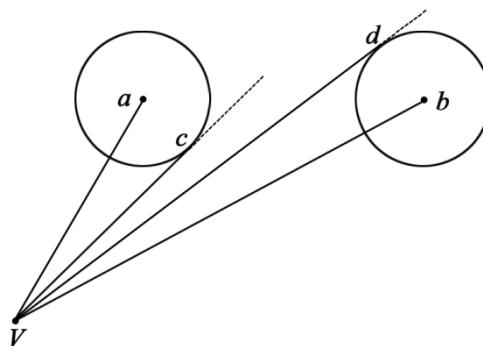
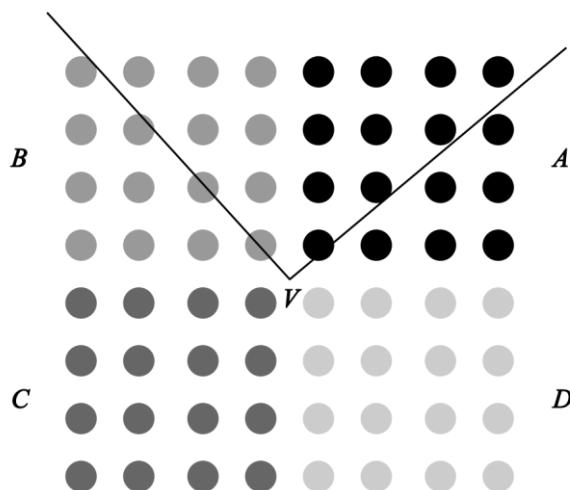


图 13-20 确定两棵树之间的是否有遮挡

如图 13-20 所示, 从观察点 V 望去, 如果有 $\angle aVb > \angle aVc + \angle dVb$, 则两棵树之间互不遮挡。 $\angle aVb$ 可由余弦定理求出, $\angle aVc$ 和 $\angle dVb$ 可由正弦定理求出。

图 13-21 以观察点 V 为中心将树木分为四个区域

如图 13-21 所示, 如果以观察点 V 为中心将围绕观察者的树分为四个区域: A 、 B 、 C 、 D , 由于观察点和树干直径有相互限制, 在确定 A 区域的树是否遮挡时, 不需要考虑其他区域的树干会遮挡 A 区域的树干。每个区域的树按照离观察点的远近再细分为内外层, 由于内层的树不可能被外层的树遮挡, 所以从内层往外层, 逐棵树判断位于内侧的树对其是否遮挡, 这样可以减少计算量, 加快速度¹。

在根据余弦定理计算角度的过程中, 由于余弦值大于 1.0 时无法取反余弦, 而是会得到一个无效的非数值, 且题目中提到当角度大于 0.01 度时人眼才能区分, 因此需要设定一个阈值来判断是否可取反余弦 (此

¹ 根据题意, 视角要大于 0.01 度才能看清树干, 按树干为 0.1 时计算能看清的树最远距离为: $(0.1/2)/\sin(0.01*\pi/180.0) \approx 16414$ 单位距离, 但是相隔如此多的树还能看见是不可能的。根据多次提交测试, 只需计算最多 10 层树木即可通过 UVa OJ 的测试数据。

处是关键，如果未加注意，很可能获得错误的结果）。由于题目给定的坐标均为小数，为了便于编码，在具体实现时可将各坐标值扩大 100 倍以方便处理。

强化练习：121 Pipe Fitters^A, 453 Intersecting Circles^D, 915 Stack of Cylinders^E, 10283 The Kissing Circles^B, 10573 Geometry Paradox^A, 11834 Elevator^C。

扩展练习：10012 How Big Is It^B, 10382 Watering Grass^B, 10947 Bear With Me Again^C。

13.6.5 最小圆覆盖

最小圆覆盖（smallest enclosing circle）问题是指给定 n ($n \geq 2$) 个点，要求确定一个具有最小半径的圆 C ，圆 C 能够将所有 n 个点包含在内（如果点在圆 C 上也计为在圆内）。由于不共线的三个点可以确定一个圆，朴素的做法是枚举三个点，以此确定一个圆 c ，如果圆 c 能够覆盖所有点，则将其列为参考的最小圆，取所有这样的圆 c 中的半径最小者即为解。不难看出，朴素方法的时间复杂度为 $O(n^4)$ 。

可以使用一种称为“随机增量算法”的方法，在 $O(n)$ 的期望复杂度内求得给定 n 个点的最小圆覆盖。算法步骤如下：

(1) 首先将所有点随机排列，使得期望复杂度能够降低到 $O(n)$ 。

(2) 按顺序逐个点加入构建最小圆覆盖，即逐步求前 i 个点的最小圆覆盖，每加入一个点就转入算法步骤 (3)。

(3) 如果发现第 i 个点在当前最小圆的外面，那么说明点 i 一定在前 i 个点的最小覆盖圆边界上，则转到算法步骤 (4) 来进一步确定这个圆，否则前 i 个点的最小覆盖圆与前 $i-1$ 个点的最小覆盖圆一致，不需更新，返回算法步骤 (2)。

(4) 此时已经确认点 i 一定在前 i 个点的最小覆盖圆的边界上，那么我们可以把当前圆的圆心设为第 i 个点，半径为置为 0，然后重新把前 $i-1$ 个点加入这个圆中（类似上面的步骤，只不过这次我们提前确定了点 i 在圆上，其目的是逐步求出包含点 i 的前 j 个点的最小覆盖圆），每加入一个点就转入算法步骤 (5)。

(5) 如果发现第 j 个点在当前的最小圆的外面，那么说明点 j 也一定在前 j 个点（包括第 i 个点）的最小覆盖圆边界上，我们转到算法步骤 (6) 来再进一步确定这个圆，否则前 j 个点（包括第 i 个点）的最小覆盖圆与前 $i-1$ 个点（包括第 i 个点）的最小覆盖圆一致，不需更新，返回算法步骤 (4)。

(6) 此时已经确认点 i, j 一定在前 j 个点（包括第 i 个点）的最小覆盖圆的边界上了，那么我们可以把当前圆的圆心设为第 i 个点与第 j 的点连线的中点，半径为到这两个点的距离（就是找一个覆盖这两个点的最小圆），然后重新把前 $j-1$ 个点加入这个圆中（还是类似于上面的步骤，只不过这次我们提前确定了两个点在圆上，目的是求出包含点 i, j 的前 k 个点的最小覆盖圆），每加入一个点就转入算法步骤 (7)。

(7) 如果发现第 k 个点在当前的最小圆的外面，那么说明点 k 也一定在前 k 个点（包括 i, j ）的最小覆盖圆边界上，由于三个点能确定一个圆，此时能够根据这三个点求出一个圆，否则前 k 个点（包括 i, j ）的最小覆盖圆与前 $k-1$ 个点（包括 i, j ）的最小覆盖圆一致，不需更新。

以下是“随机增量算法”求最小圆覆盖的参考实现。

```
//-----13.6.5.cpp-----//
const double EPSILON = 1e-7;

// 点。
struct point { double x, y; };

// 圆。
class circle {
    public:
        double x, y, r;
        circle() { x = y = r = 0; }
        circle(double x, double y, double r) { this->x = x; this->y = y; this->r = r; }
        void move(double dx, double dy) { x += dx; y += dy; }
        void shrink() { r -= EPSILON; }
        void grow() { r += EPSILON; }
        void setCenter(double x, double y) { this->x = x; this->y = y; }
        void setRadius(double r) { this->r = r; }
        void setPoint(point p) { x = p.x; y = p.y; }
        void setCircle(circle c) { x = c.x; y = c.y; r = c.r; }
        void setLine(point p1, point p2) { x = (p1.x + p2.x) / 2; y = (p1.y + p2.y) / 2; r = sqrt((p1.x - x) * (p1.x - x) + (p1.y - y) * (p1.y - y)); }
        void setCircle(circle c1, circle c2) { x = (c1.x + c2.x) / 2; y = (c1.y + c2.y) / 2; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3) { x = (c1.x + c2.x + c3.x) / 3; y = (c1.y + c2.y + c3.y) / 3; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4) { x = (c1.x + c2.x + c3.x + c4.x) / 4; y = (c1.y + c2.y + c3.y + c4.y) / 4; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5) { x = (c1.x + c2.x + c3.x + c4.x + c5.x) / 5; y = (c1.y + c2.y + c3.y + c4.y + c5.y) / 5; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x) / 6; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y) / 6; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x) / 7; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y) / 7; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x) / 8; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y) / 8; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x) / 9; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y) / 9; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x) / 10; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y) / 10; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x) / 11; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y) / 11; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x) / 12; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y) / 12; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x) / 13; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y) / 13; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x) / 14; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y) / 14; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x) / 15; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y) / 15; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x) / 16; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y) / 16; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x) / 17; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y) / 17; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x) / 18; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y) / 18; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x) / 19; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y) / 19; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x) / 20; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y) / 20; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x) / 21; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y) / 21; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x) / 22; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y) / 22; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x) / 23; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y) / 23; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x) / 24; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y) / 24; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x) / 25; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y) / 25; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x) / 26; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y) / 26; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x) / 27; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y) / 27; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x) / 28; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y) / 28; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x) / 29; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y) / 29; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x) / 30; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y + c30.y) / 30; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30, circle c31) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x + c31.x) / 31; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y + c30.y + c31.y) / 31; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30, circle c31, circle c32) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x + c31.x + c32.x) / 32; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y + c30.y + c31.y + c32.y) / 32; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30, circle c31, circle c32, circle c33) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x + c31.x + c32.x + c33.x) / 33; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y + c30.y + c31.y + c32.y + c33.y) / 33; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30, circle c31, circle c32, circle c33, circle c34) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x + c31.x + c32.x + c33.x + c34.x) / 34; y = (c1.y + c2.y + c3.y + c4.y + c5.y + c6.y + c7.y + c8.y + c9.y + c10.y + c11.y + c12.y + c13.y + c14.y + c15.y + c16.y + c17.y + c18.y + c19.y + c20.y + c21.y + c22.y + c23.y + c24.y + c25.y + c26.y + c27.y + c28.y + c29.y + c30.y + c31.y + c32.y + c33.y + c34.y) / 34; r = sqrt((c1.x - x) * (c1.x - x) + (c1.y - y) * (c1.y - y)); }
        void setCircle(circle c1, circle c2, circle c3, circle c4, circle c5, circle c6, circle c7, circle c8, circle c9, circle c10, circle c11, circle c12, circle c13, circle c14, circle c15, circle c16, circle c17, circle c18, circle c19, circle c20, circle c21, circle c22, circle c23, circle c24, circle c25, circle c26, circle c27, circle c28, circle c29, circle c30, circle c31, circle c32, circle c33, circle c34, circle c35) { x = (c1.x + c2.x + c3.x + c4.x + c5.x + c6.x + c7.x + c8.x + c9.x + c10.x + c11.x + c12.x + c13.x + c14.x + c15.x + c16.x + c17.x + c18.x + c19.x + c20.x + c21.x + c22.x + c23.x + c24.x + c25.x + c26.x + c27.x + c28.x + c29.x + c30.x + c31.x + c32.x + c33.x + c34.x + c35.x) / 35; y = (c1.y + c2.y + c3.y + c4.y + c
```

```

struct circle {
    double x, y, r;
    double distTo(point a) { return sqrt(pow(x - a.x, 2) + pow(y - a.y, 2)); }
};

// 根据三个点确定外接圆的圆心和半径。
circle getCircleFromTriangle(point &a, point &b, point &c)
{
    double A1 = a.x - b.x, B1 = a.y - b.y;
    double A2 = c.x - b.x, B2 = c.y - b.y;
    double C1 = (a.x * a.x - b.x * b.x + a.y * a.y - b.y * b.y) / 2;
    double C2 = (c.x * c.x - b.x * b.x + c.y * c.y - b.y * b.y) / 2;
    circle cc;
    cc.x = (C1 * B2 - C2 * B1) / (A1 * B2 - A2 * B1);
    cc.y = (A1 * C2 - A2 * C1) / (A1 * B2 - A2 * B1);
    cc.r = cc.distTo(a);
    return cc;
}

// 随机增量方法确定最小圆覆盖。
double getMinCoverCircle(point v[], int n)
{
    // 利用库函数将点随机排列。
    random_shuffle(v, v + n);
    // 将初始的最小覆盖圆定为圆心在第一个点，半径为 0。
    circle c;
    c.x = v[0].x, c.y = v[0].y, c.r = 0;
    // 逐个点加入，更新最小覆盖圆。
    for (int i = 1; i < n; i++)
        if (c.distTo(v[i]) >= c.r + EPSILON) {
            c.x = v[i].x, c.y = v[i].y, c.r = 0;
            for (int j = 0; j < i; j++)
                if (c.distTo(v[j]) >= c.r + EPSILON) {
                    c.x = (v[i].x + v[j].x) / 2, c.y = (v[i].y + v[j].y) / 2;
                    c.r = c.distTo(v[i]);
                    for (int k = 0; k < j; k++)
                        if (c.distTo(v[k]) >= c.r + EPSILON)
                            c = getCircleFromTriangle(v[i], v[j], v[k]);
                }
            }
        return c.r;
}
//-----13.6.5.cpp-----//

```

强化练习：10005 Packing Polygons^B。

13.7 小结

几何是计算几何的基础，在编程竞赛中，一般不会作为一道题目出现，而是作为题目的背景或者处理问题的一个环节。本章先介绍了点、直线的表示方法、直线交点的求法，由于是使用浮点数直接表示，在计算过程中容易产生误差，与之相比较，在计算几何中，一般使用向量来表示几何中的基本元素，这样不仅代码的健壮性更强，误差也更小。接下来介绍了坐标和坐标系的变换，坐标系变换使用矩阵形式表示最为简便，因此会涉及矩阵求逆。三角函数以及三角形的性质是几何中的一个重点内容，其中正弦定理和余弦定理需要熟练掌握。对于多边形来说，正多边形的性质需要了解。圆的标准方程和一般方程、圆的周长和面积、圆与

圆位置关系的判断、圆的切线、三角形的内切圆和外切圆、最小圆覆盖是与圆有关需要掌握的内容。

第 14 章 计算几何

易曰：“君子慎始。差若毫厘，谬以千里。”
——《礼记·经解》

计算几何学是计算机科学的一个分支，专门研究用来解决几何问题的算法。在现代工程与数学中，计算机图形学、机器人学、VLSI (Very Large Scale Integration, 超大规模集成电路) 设计、计算机辅助设计以及统计学等领域中，都要用到计算几何学。本章主要介绍有关点、直线、线段、圆、多边形的计算问题。

14.1 基本概念

14.1.1 线段

线段 (segment)，在其两端各有一个点 $p_1(x_1, y_1)$ 和 $p_2(x_2, y_2)$ ，称为端点 (end)。从直观上描述，线段 p_1p_2 可以看成经过点 p_1 和 p_2 的直线位于 p_1 和 p_2 之间（包括 p_1 和 p_2 ）点的集合。更为形式化的定义是使用凸组合 (convex combination) 的概念^I。给定两个不同点 p_1 和 p_2 ，两者的凸组合是点 $p_3(x_3, y_3)$ ，满足

$$x_3 = ax_1 + (1 - a)x_2, \quad y_3 = ay_1 + (1 - a)y_2, \quad 0 \leq a \leq 1$$

线段 p_1p_2 即为 p_1 和 p_2 凸组合的集合。

很自然的，可以使用线段的两个端点来表示线段本身。有时为了形式上的统一和应用上的简便，也可以直接套用表示线段的结构体来表示直线。

```
struct segment {
    point p1, p2;
};
typedef segment line;
```

强化练习：837 Light and Transparencies^B。

扩展练习：972 Horizon Line^D，1193 Radar Installation^D，12321 Gas Stations^D。

14.1.2 多边形

多边形 (polygon) 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，包括非相邻线段也不能自交，不论是否只是在端点处相交。

为了方便处理，一般使用顺序 (逆时针或顺时针方向) 列出多边形各个顶点的方式来予以表示。具体代码实现时，使用一个整数来存储顶点数量，一个数组存储顶点的坐标，或者直接使用 `vector` 来存储多边形的顶点。

```
const int MAXV = 1000;
struct polygon {
    int number = 0;
```

^I 凸组合是一类特殊的线性组合。假设 x_1, x_2, \dots, x_n 是一组对象 (可以是实数、点等对象)， a_1, a_2, \dots, a_n 是 n 个常数 ($a_i \geq 0, i=1, 2, \dots, n$)，并且满足 $a_1 + a_2 + \dots + a_n = 1$ ，那么 $a_1x_1 + \dots + a_nx_n$ 就称为 x_1, x_2, \dots, x_n 的凸组合。

```

    point vertices[MAXV];
};

typedef vector<point> polygon;

```

一个多边形 P 为凸多边形当且仅当 P 的任意两个不相邻顶点的连线完全位于 P 的内部。凸多边形的任意内角均小于 180 度 (π 弧度)。如果一条直线将凸多边形分为两部分 (直线不与凸多边形的任意一条边重合), 则分成的两部分仍为凸多边形。

14.2 几何对象间的关系

在解题时, 常常需要将几何对象使用相应的代码予以表示。为了便于代码的重复利用和强壮性, 一般都会将其实现为一个几何代码库 (library), 将一些基本操作 “封装” 在库中以便随时使用, 免去了重复编写代码的不便。如果使用常规的定义来处理几何对象间的关系, 大多数场合会显得繁琐, 而使用向量, 则会使代码更为简洁和可靠, 因此先引入向量的概念, 并使用向量来作为库的基础部件。

14.2.1 向量、内积和外积

在数学中, 标量是指只有大小的量, 如体积、重量等, 而将既有大小又有方向的量, 称为向量 (vectors)。

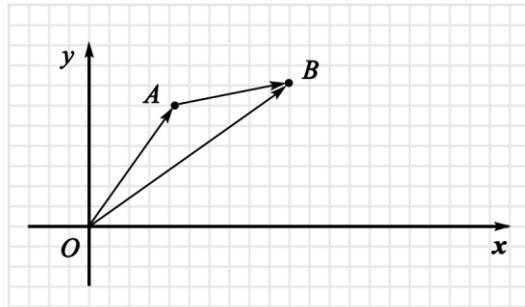


图 14-1 向量 OA , OB , AB

如图 14-1 所示, 在直角坐标系中, 原点为 $O(0, 0)$, 设有点 $A(x_1, y_1)$, $B(x_2, y_2)$, 则向量 \overrightarrow{AB} 可以表示为

$$\overrightarrow{AB} = \overrightarrow{OB} - \overrightarrow{OA} = (x_2 - x_1, y_2 - y_1)$$

其方向为从 A 指向 B 。

向量 a 的大小, 称为向量的模 (modulus, 又称长度), 一般记做 $|a|$ 。向量的模是一个非负实数, 令 a 为 (x, y) , 有

$$|a| = \sqrt{x^2 + y^2}$$

除此之外, 还有表示向量大小平方的概念, 称为范数 (norm)。

```

double norm(point a) { return a.x * a.x + a.y * a.y; }
double abs(point a) { return sqrt(norm(a)); }

```

使用坐标表示向量, 假设有向量 $u(x_u, y_u)$ 和 $v(x_v, y_v)$, u 和 v 之间的夹角为 θ , 根据余弦定理, 有

$$|u - v||u - v| = |u||u| + |v||v| - 2|u||v| \cos \theta$$

即

$$(x_u - x_v)^2 + (y_u - y_v)^2 = x_u^2 + y_u^2 + x_v^2 + y_v^2 - 2|u||v| \cos \theta$$

化简得

$$\cos \theta = \frac{x_u x_v + y_u y_v}{|\mathbf{u}| |\mathbf{v}|}$$

这样，就可以根据向量 \mathbf{u} 和 \mathbf{v} 计算两者之间的夹角。

定义向量 \mathbf{u} 和 \mathbf{v} 的内积 (dot product, 又称点积、数量积) 为

$$\mathbf{u} \cdot \mathbf{v} = x_u x_v + y_u y_v$$

注意，内积是一个标量。于是上面的 $\cos \theta$ 可改写成

$$\cos \theta = \frac{\mathbf{u} \cdot \mathbf{v}}{|\mathbf{u}| |\mathbf{v}|}$$

当 $\mathbf{u} \cdot \mathbf{v} = 0$ (即 $x_u x_v + y_u y_v = 0$) 时，向量 \mathbf{u} 和 \mathbf{v} 垂直；当 $\mathbf{u} \cdot \mathbf{v} > 0$ 时， \mathbf{u} 和 \mathbf{v} 之间的夹角为锐角；当 $\mathbf{u} \cdot \mathbf{v} < 0$ 时， \mathbf{u} 和 \mathbf{v} 之间的夹角为钝角。

```
double dot(point a, point b) { return a.x * b.x + a.y * b.y; }
```

定义两个向量 \mathbf{u} 和 \mathbf{v} 的外积 (cross product, 又称叉积、向量积) 仍然是一个向量，记作 $\mathbf{u} \times \mathbf{v}$ ，它的长度规定为

$$|\mathbf{u} \times \mathbf{v}| = |\mathbf{u}| |\mathbf{v}| \sin \theta$$

其中 θ 为两个向量间的夹角。外积的方向与向量 \mathbf{u} 和 \mathbf{v} 所在平面垂直，并满足右手螺旋法则^I。

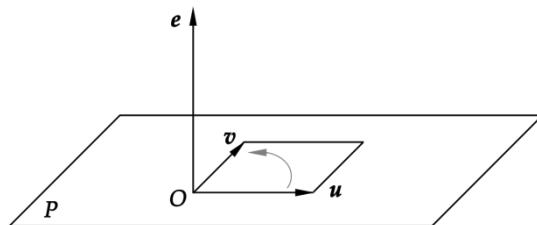


图 14-2 向量 \mathbf{u} 和 \mathbf{v} 外积的几何意义。假定已经用单位向量 \mathbf{e} 规定了平面 P 的定向，对于平面 P 上的定向平行四边形，可以给它的面积定义一个正负号；如果它的定向与 P 的定向一致，则规定它的面积为正的；如果不一致，则规定它的面积为负。该面积称为定向平行四边形的有向面积（或称定向面积）

如图 14-2 所示，外积 $\mathbf{u} \times \mathbf{v}$ 的几何意义是由向量 $\mathbf{O}(0, 0)$, \mathbf{u} , \mathbf{v} , $\mathbf{u} + \mathbf{v} = (x_u + x_v, y_u + y_v)$ 在坐标系中对应的点所确定的平行四边形的有向面积 $A_{\mathbf{u} \times \mathbf{v}}$ ，可以表示为

$$A_{\mathbf{u} \times \mathbf{v}} = x_u y_v - y_u x_v$$

如果 $A_{\mathbf{u} \times \mathbf{v}}$ 的值为正数，则 \mathbf{u} 在 \mathbf{v} 的顺时针方向上；如果 $A_{\mathbf{u} \times \mathbf{v}}$ 的值为负数，则 \mathbf{u} 在 \mathbf{v} 的逆时针方向上；当 $A_{\mathbf{u} \times \mathbf{v}}$ 的值为 0 时， \mathbf{u} 和 \mathbf{v} 共线，但方向可能相同或者正好相反^{II}。

```
double cross(point a, point b) { return a.x * b.y - a.y * b.x; }
```

可以将任意向量的起点平移到原点，使得向量的终点可以用一个点来表示，确定了此点即确定了向量的

^I 将右手除大拇指以外的四个手指展开，指向向量 \mathbf{A} 的方向，然后把四个手指弯曲，弯曲的方向由向量 \mathbf{A} 转向向量 \mathbf{B} (转的角度须小于 π 弧度)，此时大拇指立起的方向，就是向量 \mathbf{A} 和向量 \mathbf{B} 外积的方向。

^{II} 如果沿着顺时针方向旋转 \mathbf{u} ，转过的角度不大于 π ，即可使得 \mathbf{u} 的方向与 \mathbf{v} 的方向相同，则称 \mathbf{u} 在 \mathbf{v} 的逆时针方向， \mathbf{v} 在 \mathbf{u} 的顺时针方向。

大小和方向。为了更为方便地应用向量来进行计算，后续均使用点来表示一个向量。以下代码片段定义了向量及其基本运算。

```
struct point {
    double x, y;
    point (double x = 0, double y = 0): x(x), y(y) {}
    point operator + (point p) { return point(x + p.x, y + p.y); }
    point operator - (point p) { return point(x - p.x, y - p.y); }
    point operator * (double u) { return point(x * u, y * u); }
    point operator / (double u) { return point(x / u, y / u); }
};
```

10089 Repackaging^D (重新打包)

现有三种不同尺寸的咖啡杯（编号分别为规格 1，规格 2，规格 3），均由杯子制造协会（Association of Cup Makers, ACM）下属的工厂所生产，并且以不同的包装尺寸出售。每种包装以三个整数 (S_1, S_2, S_3) 予以标记，其中 S_i ($1 \leq i \leq 3$) 表示规格 i 的杯子在包装中的数量。对于任意一种包装，不存在 $S_1 = S_2 = S_3$ 的情形。

最近，客户对于包含同样数量三种规格杯子的包装需求量大幅增长。作为对需求的一种应急措施，ACM 决定将库存中尚未出售的（无限）包装拆开并进行重新打包，使得包装中三种规格的杯子数量相同。例如，假设 ACM 库存中有以下四种类型的包装：(1, 2, 3), (1, 11, 5), (9, 4, 3) 和 (2, 3, 2)，则可以拆开三个(1, 2, 3)的包装，一个(9, 4, 3)的包装和两个(2, 3, 2)的包装，将它们重新打包为十六个(1, 1, 1)的包装，或者八个(2, 2, 2)的包装，或者四个(4, 4, 4)的包装，或者两个(8, 8, 8)的包装，或者一个(16, 16, 16)的包装，或者上述包装的组合，只要同一个包装中不同规格杯子的数量相同。注意，所有从分拆的包装中获得的杯子在重新打包的过程中必须全部使用。

ACM 雇用你来编写程序，确定能否从库存中选取若干包装，将其拆开后重新打包，使得新的包装中不同规格的杯子数量相等且分拆得到的杯子全部使用。

输入

输入包含多组测试数据，每组测试数据的第一行包含一个整数 N ($3 \leq N \leq 1000$)，表示库存中不同包装的数量，接着 N 行，每行包含三个正整数，表示某个包装中规格 1，规格 2，规格 3 的杯子数量，同组测试数据中不会出现相同的包装。输入以 N 为 0 的一行表示结束。

输出

对于每组测试数据，假如能够按照要求进行重新打包，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
0
```

样例输出

```
Yes
```

分析

令第 i 种包装的数量为 a_i ，第 i 种包装中规格 j 的杯子数量为 S_{ij} ， $1 \leq i \leq n$ ， $1 \leq j \leq 3$ ， $0 \leq a_i$ ，根据题目约束，有

$$\begin{cases} a_1S_{11} + a_2S_{21} + \dots + a_nS_{n1} = k \\ a_1S_{12} + a_2S_{22} + \dots + a_nS_{n2} = k, \quad k > 0, \\ a_1S_{13} + a_2S_{23} + \dots + a_nS_{n3} = k \end{cases}$$

由上式不难得到

$$\begin{cases} a_1(S_{12} - S_{11}) + a_2(S_{22} - S_{21}) + \dots + a_n(S_{n2} - S_{n1}) = 0 \\ a_1(S_{13} - S_{11}) + a_2(S_{23} - S_{21}) + \dots + a_n(S_{n3} - S_{n1}) = 0 \end{cases}, \quad \sum_{i=1}^n a_i > 0, \quad a_i \in \mathbb{Z}_0^+$$

令 $x_i = S_{i2} - S_{i1}$, $y_i = S_{i3} - S_{i1}$, 则上式可转化为以下二维“向量和”问题:

$$a_1(x_1, y_1) + a_2(x_2, y_2) + \dots + a_n(x_n, y_n) = (0, 0), \quad \sum_{i=1}^n a_i > 0, \quad a_i \in \mathbb{Z}_0^+$$

考虑最简单的情形, 当只有两个向量时, 令 $V_1 = (x_1, y_1)$, $V_2 = (x_2, y_2)$, 由于 a_i 为非负整数且至少有一个 a_i 不为 0, 不难得出, 要使得 $a_1V_1 + a_2V_2 = (0, 0)$, 则向量 V_1 与 V_2 之间的方向必定相反, 即夹角为 π 弧度。若两者夹角大于 π 弧度, 则不可能在约束条件下得到零向量。进一步地, 对所有 n 个向量按极角进行排序后, 如果有任何两个相邻向量 V_i 和 V_j 之间的夹角大于 π 弧度, 则不可能在约束条件下得到零向量, 即无解, 反之, 则有解。

强化练习: 10832 Yoyodyne Propulsion Systems^D, 11473 Campus Roads^D, 11800 Determine the Shape^C。

扩展练习: 12165* Triangle Hazard^E。

14.2.2 点和直线的关系

设经过原点 $O(0, 0)$ 和点 $m(x_m, y_m)$ 的直线为 L , 则可得到直线 L 的方程为

$$\begin{vmatrix} x_m & y_m \\ x & y \end{vmatrix} = x_m y - y_m x = 0$$

给定直线 L 上不同的两个点 p_a 和 p_b , 令 p_a 为起点, p_b 为终点, 从 p_a 向 p_b 望去, 位于右侧的半平面定义为直线 L 的顺时针区域, 位于左侧的半平面定义为直线 L 的逆时针区域。设有一点 $p(x_p, y_p)$, 不难推出, 点 p 与直线 L 的位置关系只有三种: 位于直线上、位于直线的顺时针区域、位于直线的逆时针区域。

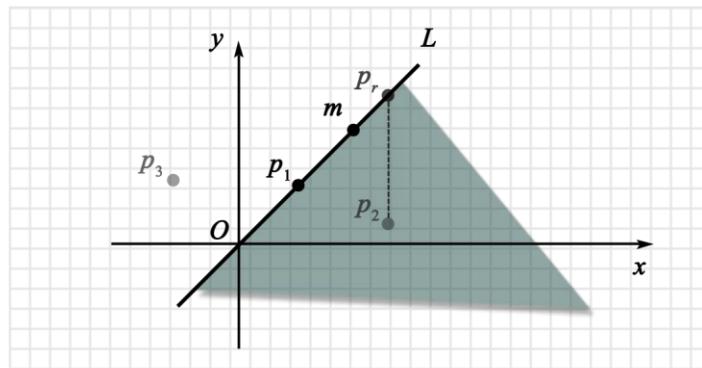


图 14-3 点与直线可能的位置关系。定义直线 L 的方向为从 O 到 m , 点 p_1 位于直线上, 点 p_2 位于直线的顺时针区域, 点 p_3 位于直线的逆时针区域。注意, 直线的顺时针区域和逆时针区域只是象征性地予以表示, 并未完全绘出

可由向量 \overrightarrow{Om} 与向量 \overrightarrow{Op} 的外积确定点 p 与直线 L 的关系。根据外积的几何意义, 向量 p 和 m 构成的平行四边形的有向面积 $A_{p \times m}$ 为

$$A_{p \times m} = x_p y_m - y_p x_m$$

当点 p 位于直线上, 根据直线方程有

$$x_m y_p - y_m x_p = 0$$

则

$$A_{p \times m} = x_p y_m - y_p x_m = -(x_m y_p - y_m x_p) = 0$$

故只要点 p 位于直线 L 上, 有向面积 $A_{p \times m}$ 为零。当点 p 位于直线 L 的顺时针区域时, 设与点 p 具有相同横坐标且在直线 L 上的点为 $p_r(x_r, y_r)$, 由于 p_r 在直线 L 上, 故有

$$\begin{vmatrix} x_m & y_m \\ x_p & y_r \end{vmatrix} = x_m y_r - y_m x_p = 0$$

由于 $y_r > y_p$, 则有

$$x_m y_p - y_m x_p < 0$$

则

$$A_{p \times m} = x_p y_m - y_p x_m = -(x_m y_p - y_m x_p) > 0$$

即当点 p 位于直线 L 的顺时针区域时, 有向面积 $A_{p \times m}$ 为正。同理可知当点 p 位于直线 L 的逆时针区域时, 可得到有向面积 $A_{p \times m}$ 为负的结果。

以上只考虑了直线经过第一、第三象限的情况, 同理可证明当直线与 x 轴平行、直线与 y 轴平行、直线经过第二、四象限时均有以下结论: 当点 p 位于直线上时, 有向面积 $A_{p \times m}$ 总为零, 当位于顺时针区域时, 有向面积 $A_{p \times m}$ 总为正, 当位于逆时针区域时, 有向面积 $A_{p \times m}$ 总为负。由于点 p 与直线的位置关系只可能为三种情况之一, 故可得当有向面积 $A_{p \times m}$ 为正时, 点 p 位于向量 m 所在直线的顺时针区域, 当有向面积 $A_{p \times m}$ 为负时, 点 p 位于向量 m 所在直线的逆时针区域, 当有向面积 $A_{p \times m}$ 为零时, 点 p 与向量 m 所在直线共线。

强化练习: [10865 Brownie Points I^c](#), [11227 The Silver Bullet^c](#)。

14.2.3 确定线段转动方向

为了确定两条连续的线段 p_0p_1 和 p_1p_2 是向左转还是向右转, 即确定一个给定的角 $\angle p_0p_1p_2$ 的转向, 可以使用外积, 将其转化为相对于公共端点 p_0 的向量 $\overrightarrow{p_0p_1}$ 和 $\overrightarrow{p_0p_2}$ 的位置关系问题, 只需要把 p_0 作为原点, 计算有向面积

$$A_{(p_1-p_0) \times (p_2-p_0)} = (x_1 - x_0)(y_2 - y_0) - (y_1 - y_0)(x_2 - x_0)$$

使用代码表示为

```
double cp(point p0, point p1, point p2) {
    // return (p1.x - p0.x) * (p2.y - p0.y) - (p1.y - p0.y) * (p2.x - p0.x);
    return cross(p1 - p0, p2 - p0);
}
```

如果该有向面积的值为正, 则 $\overrightarrow{p_0p_1}$ 在 $\overrightarrow{p_0p_2}$ 的顺时针方向上, 经过点 p_1 处要向左转; 如果为负, 则 $\overrightarrow{p_0p_1}$ 在 $\overrightarrow{p_0p_2}$ 的逆时针方向上, 经过点 p_1 时要向右转; 如果有向面积的值为零, 则表示共线。在编程实践中, 由于坐标不一定是整数, 所以在判断有向面积的值是否为零时, 一般使用一个阈值来予以控制, 当有向面积的绝对值小于该阈值时, 即可认为有向面积为零。由此, 可将外积演化为判断线段转向的三个函数: 顺时针旋转 cw (clockwise)、逆时针旋转 ccw (counter clockwise)、共线 collinear (collinear)。

```
//-----14.2.3.cpp-----/
const double EPSILON = 1E-7;
```

```

// 判断谓词: cw (clockwise), 顺时针旋转。
// 从点 a 向点 b 望去, 如果点 c 位于线段 ab 的右侧, 返回 true, 否则返回 false。
bool cw(point a, point b, point c) { return cp(a, b, c) < -EPSILON; }

// 判断谓词: ccw (counterclockwise), 逆时针旋转。
// 从点 a 向点 b 望去, 如果点 c 位于线段 ab 的左侧时, 返回 true, 否则返回 false。
bool ccw(point a, point b, point c) { return cp(a, b, c) > EPSILON; }

// 当三点共线时, 返回 true。
bool collinear(point a, point b, point c) { return fabs(cp(a, b, c)) <= EPSILON; }

// 判断三点是否构成右转或共线。
bool cwOrCollinear(point a, point b, point c) {
    return cw(a, b, c) || collinear(a, b, c);
}

// 判断三点是否构成左转或共线。
bool ccwOrCollinear(point a, point b, point c) {
    return ccw(a, b, c) || collinear(a, b, c);
}

//-----14.2.3.cpp-----//

```

强化练习: 270 Lining Up^B, 319 Pendulum^D, 833 Water Falls^C。

扩展练习: [11008 Antimatter Ray Clearcutting^C](#)。

14.2.4 确定线段是否相交

对于线段来说, 经常需要进行的操作是判断给定点是否在线段上, 而采用“包围盒测试”可以方便地得到结果。

```

// 测试点 p 是否在线段 ab 上。
bool pointInBox(point a, point b, point p) {
    double minx = min(a.x, b.x), maxx = max(a.x, b.x);
    double miny = min(a.y, b.y), maxy = max(a.y, b.y);
    return p.x >= minx && p.x <= maxx && p.y >= miny && p.y <= maxy;
}

struct segment {
    point p1, p2;
    bool contains(point &p) { return pointInBox(p1, p2, p); }
};

```

而对于确定两条线段是否相交则稍复杂, 有以下两种常用方法。

直线求交法

直观的方法是将线段转换为直线, 求直线的交点, 然后判断交点是否在线段上, 从而进一步得知两条线段是否相交。此方法由于需要求交点, 在求交点时可能会存在误差导致结果产生偏差。

```

//-----14.2.4.1.cpp-----//
const double EPSILON = 1e-7;

struct segment {
    point p1, p2;
    bool contains(const point &p) { return pointInBox(p1, p2, p); }
};

```

```

struct line {
    double a, b, c;
    line (double a = 0, double b = 0, double c = 0): a(a), b(b), c(c) {}
};

line getLine(double x1, double y1, double x2, double y2) {
    return line(y2 - y1, x1 - x2, y1 * (x2 - x1) - x1 * (y2 - y1));
}

line getLine(point p, point q) {
    return getLine(p.x, p.y, q.x, q.y);
}

// 求两条不同直线的交点。需要事先判断两条直线是否平行，如果平行则无交点。
point getIntersection(line p, line q) {
    point pi;
    pi.x = (q.c * p.b - p.c * q.b) / (p.a * q.b - q.a * p.b);
    pi.y = (q.c * p.a - p.c * q.a) / (p.b * q.a - q.b * p.a);
    return pi;
}

double cp(point a, point b, point c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

// 判断两条线段是否相交。
bool isIntersected(segment s1, segment s2) {
    // 构造直线。
    line p = getLine(s1.p1, s1.p2), q = getLine(s2.p1, s2.p2);

    // 判断线段是否重合或平行。
    if (fabs(p.a * q.b - q.a * p.b) < EPSILON) {
        if (fabs(cp(s1.p1, s1.p2, s2.p1)) < EPSILON)
            return s1.contains(s2.p1) || s1.contains(s2.p2);
        return false;
    }

    // 求直线的交点。
    point pi = getIntersection(p, q);
    return s1.contains(pi) && s2.contains(pi);
}
//-----14.2.4.1.cpp-----//

```

上述“直线求交法”是根据直线一般方程的系数来计算交点。除此之外，也可以使用外积来计算交点，而且更为简便。

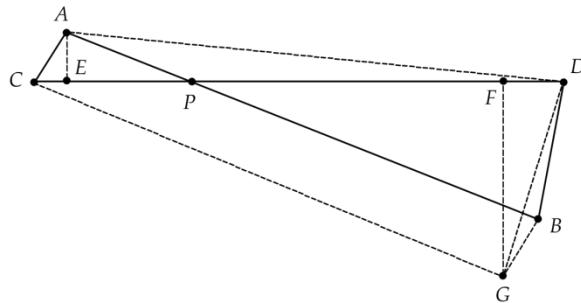


图 14-4 利用外积求直线的交点

如图 14-4 所示, 由于三角形 AEP 和 GFC 相似, 可得

$$\frac{AE}{GF} = \frac{AP}{GC} = \frac{AP}{AB}$$

而 AE 是三角形 ACD 的高, GF 是三角形 GCD 的高, 且三角形 ACD 和 GCD 具有共同的底边 GC , 只需确定三角形 ACD 和 GCD 的面积, 就能确定线段 AP 与 AB 的比值。令线段 AP 与 AB 的比值为 k , 则根据向量长度之间的关系, 有

$$\mathbf{P} = \mathbf{A} + (\mathbf{B} - \mathbf{A}) * k$$

不难推知, 三角形 ACD 的面积可由向量 \mathbf{CA} 和 \mathbf{CD} 的外积得到, 三角形 GCD 的面积可由向量 \mathbf{AB} 和 \mathbf{CD} 的外积得到。在具体实现时, 需要注意坐标点 A 和 C 的选择, 两者不能重合, 否则除数将为零。

```
//-----14.2.4.2.cpp-----
const double EPSILON = 1e-6;

struct point {
    double x, y;
    point (double x = 0, double y = 0): x(x), y(y) {}
    point operator + (point p) { return point(x + p.x, y + p.y); }
    point operator - (point p) { return point(x - p.x, y - p.y); }
    point operator * (double k) { return point(x * k, y * k); }
    point operator / (double k) { return point(x / k, y / k); }
    bool operator==(point &p)
    {
        return fabs(x - p.x) < EPSILON && fabs(y - p.y) < EPSILON;
    }
};

double cross(point a, point b) { return a.x * b.y - a.y * b.x; }

struct line { point a, b; };

point getIntersection(line p, line q) {
    if (p.a == q.a) return p.a;
    double k = cross(p.a - p.b, q.a - q.b) / cross(p.a - q.a, q.a - q.b);
    return p.a + (p.b - p.a) * fabs(k);
}
//-----14.2.4.2.cpp-----
```

相对方位法

相比较于“直线求交法”判断线段相交，更为“巧妙”的方法是根据线段在平面上的相对位置关系，结合外积进行判断。

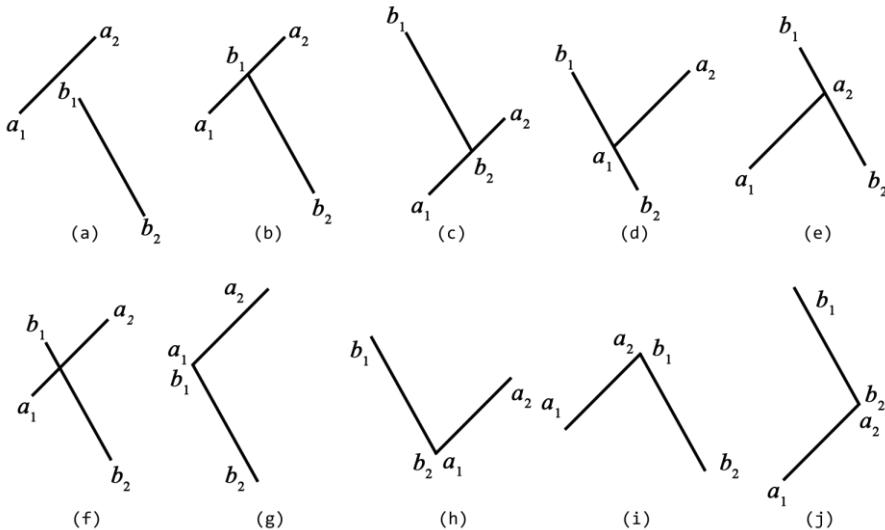


图 14-5 两条线段的相对位置的各种情形（仅列出部分情形，如两条线段共线并部分重叠的情形未列出）

如图 14-5 所示，将这些不同的位置关系定义为相对方位，可以通过线段的相对方位来判断是否相交。相对于线段 a_1a_2 的两个端点，另一线段 b_1b_2 的两个端点分别与之构成转角 $\angle a_1a_2b_1$ 和 $\angle a_1a_2b_2$ ，约定逆时针转角为负，顺时针转角为正，转换为使用外积来表示则转角为正时外积为正，转角为负时外积为负，如果转角 $\angle a_1a_2b_1$ 和 $\angle a_1a_2b_2$ 对应的外积一个为正一个为负则称线段 a_1a_2 跨越（straddle）了另一条线段 b_1b_2 所在的直线。若两条线段相交，则至少满足以下两个条件之一：

- (1) 每条线段都跨越了包含另一线段的直线。
- (2) 一条线段的某一端点位于另一线段上。

由此可以计算线段的每个端点相对于另一线段的相对方位 d_i 。如果所有的相对方位 d_i 都非 0，则可以通过测试相对方位数值的符号判断线段是否相交。如果出现相对方位 d_i 为 0 的情况，表明一条线段的一个端点与另一条线段共线，只需测试该端点是否在另一条线段上，为是则表明两条线段相交，可以通过前述的包围盒测试来进行。

定义 $cp_1 = cp(a_1, a_2, b_1)$, $cp_2 = cp(a_1, a_2, b_2)$, $cp_3 = cp(b_1, b_2, a_1)$, $cp_4 = cp(b_1, b_2, a_2)$ 。按照图 14-5 的各种情形，可以得到如下列表：

序号/外积	cp_1	cp_2	cp_3	cp_4
a	>0	>0	>0	<0
b	$=0$	>0	>0	<0
c	<0	$=0$	>0	<0
d	<0	>0	$=0$	<0
e	<0	>0	>0	$=0$
f	<0	>0	>0	<0

g	$=0$	>0	$=0$	<0
h	<0	$=0$	$=0$	<0
i	$=0$	>0	>0	$=0$
j	<0	$=0$	>0	$=0$

从上表可知：

- (1) 当两条线段不共线且不相交时 (如情形 a), cp_1 、 cp_2 同号的同时 cp_3 、 cp_4 异号, 或者 cp_3 、 cp_4 同号的同时 cp_1 、 cp_2 异号, 且均不为 0。
- (2) 当两条线段跨越式相交时 (如情形 f), cp_1 、 cp_2 异号, cp_3 、 cp_4 异号, 且均不为 0;
- (3) 当两条线段相交于一条线段的端点, 且交点位于另一线段的中部时 (如情形 b, c, d, e), cp_1 、 cp_2 、 cp_3 、 cp_4 中只有一个为 0;
- (4) 当两条线段相交于端点时, 且端点重合时 (如情形 g, h, i, j), cp_1 、 cp_2 必有一个为 0, cp_3 、 cp_4 必有一个为 0, 但不同时为 0;
- (5) 当两条线段共线时, cp_1 、 cp_2 、 cp_3 、 cp_4 为 0。

```
//-----14.2.4.3.cpp-----//
const double EPSILON = 1e-7;

// 使用外积来表示线段的相对方向。
double cp(point a, point b, point c) { return cross(b - a, c - a); }

// 判断两条线段是否相交。
bool isIntersected(segment s1, segment s2) {
    double cp1 = cp(s1.p1, s1.p2, s2.p1), cp2 = cp(s1.p1, s1.p2, s2.p2);
    double cp3 = cp(s2.p1, s2.p2, s1.p1), cp4 = cp(s2.p1, s2.p2, s1.p2);
    if ((cp1 * cp2 < 0) && (cp3 * cp4 < 0)) return true;
    if (fabs(cp1) <= EPSILON && s1.contains(s2.p1)) return true;
    if (fabs(cp2) <= EPSILON && s1.contains(s2.p2)) return true;
    if (fabs(cp3) <= EPSILON && s2.contains(s1.p1)) return true;
    if (fabs(cp4) <= EPSILON && s2.contains(s1.p2)) return true;
    return false;
}
//-----14.2.4.3.cpp-----//
```

当给定的坐标点均为整数坐标时, 可以直接将外积与 0 比较, 而不需使用外积绝对值和阈值比较的形式。

强化练习: [191 Intersection^A](#), [273 Jack Straws^C](#), [866 Intersecting Line Segments^C](#), [10902 Pick-Up Sticks^C](#), [11343 Isolated Segments^C](#), [11783 Nails^D](#)。

扩展练习: [393 The Doors^C](#), [1342 That Nice Euler Circuit^D](#)。

14.2.5 点的投影

从点 p 向直线 (或线段) $s=p_1p_2$ 引一条垂线, 令交点为 x , 点 x 就称为点 p 在直线 (线段) s 上的投影 (projection)。

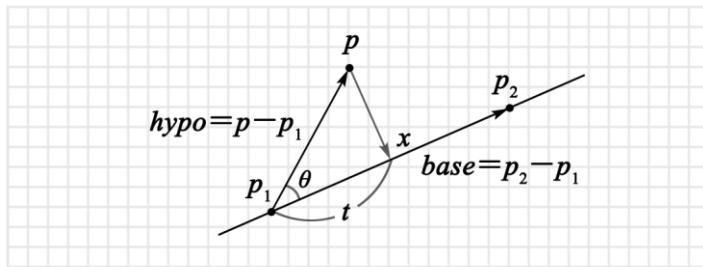


图 14-6 点的投影

如图 14-6 所示, 令向量 $base = p_2 - p_1$, $hypo = p - p_1$, 点 p_1 与点 x 的距离为 t , $hypo$ 与 $base$ 的夹角为 θ , 则有

$$t = |hypo| \cos \theta, \quad hypo \cdot base = |hypo| |base| \cos \theta$$

于是有

$$t = \frac{hypo \cdot base}{|base|}$$

由于

$$\frac{x - p_1}{base} = \frac{t}{|base|}$$

可得

$$x = p_1 + base \frac{t}{|base|} = p_1 + base \frac{hypo \cdot base}{|base|^2}$$

使用代码表示为

```
point project(point p, segment s) {
    point base = s.p2 - s.p1;
    double r = dot(p - s.p1, base) / norm(base);
    return s.p1 + base * r;
}
```

强化练习: [10263 Railway^B](#)。

给定线段 $s=p_1p_2$ 上的点 p , 过点 p 引一条垂线, 将点 p 沿着垂线朝线段 s 的逆时针方向移动距离 d 后成为点 x , 试求点 x 的坐标。可按照求点的投影相反的思路来求解。

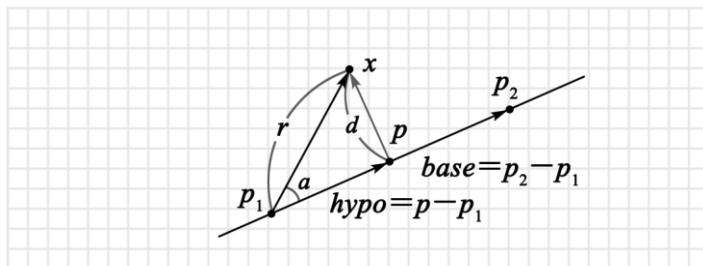


图 14-7 点沿着法线方向移动

观察图 14-7 可知, 只要求出向量 p_1p 以及 p_1x 与 p_1p 的夹角 a , 则根据坐标旋转即可求得 x 。需要注意,

如果通过取夹角 α 的反正切 $\text{atan}(d/|\text{hypo}|)$ 来得到 α ，可能会出现退化的情况。例如，当点 p 和 p_1 重合时， $|\text{hypo}|$ 为 0，无法取得反正切。为了避免这种情况，可以通过直角三角形的边来计算 $|p_1x|$ ，然后通过 α 的反弦 $\text{asin}(d/|p_1x|)$ 来得到 α 。另外，如果点 p 位于线段 s 的端点 p_1 的左侧，则向量 p_1x 与 p_1p_2 的夹角 α 将是 $\pi - \text{asin}(d/|p_1x|)$ 。

```
point rotate(point p, double t) {
    return point(p.x * cos(t) - p.y * sin(t), p.x * sin(t) + p.y * cos(t));
}

point shift(point p, double d, segment s) {
    point hypo = p - s.p1, base = s.p2 - s.p1;
    double r = sqrt(d * d + norm(hypo));
    double a = asin(d / r);
    return p1 + rotate(base * r / abs(base), a);
}
```

利用类似的思路，可以容易地求得将三角形某个角的顶点沿着角平方线移动距离 d 后的坐标，这在处理凸多边形的缩放时非常有用。

强化练习：877 Offset Polygons^E，10406 Cutting Tabletops^D。

14.2.6 点的映像

以线段（或直线） $s=p_1p_2$ 为对称轴，与给定点 p 构成线对称的点 x 称为点 p 的映像（reflection）。

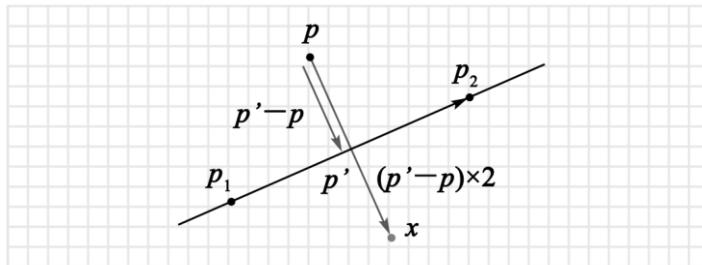


图 14-8 点的映像

求点 p 的映像 x 的坐标时，首先求出点 p 在线段 p_1p_2 上的投影点 p' ，然后将 p 到 p' 的向量 $p' - p$ 扩大至标量 2 倍，最后给起点 p 加上这个向量即为点 x 的坐标。使用代码表示为

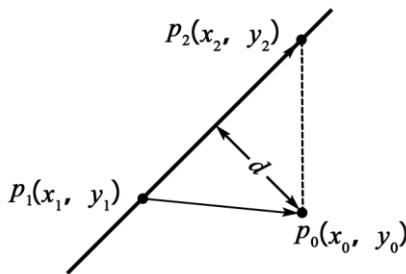
```
point reflect(point p, segment) {
    return p + (project(p, s) - p) * 2.0;
}
```

14.2.7 点和直线间距离

给定点 p 和不经过点 p 的直线 L ，如下定义点 p 和直线 L 之间的距离：令经过点 p 且与直线 L 垂直的直线为 L' ，直线 L' 与直线 L 的交点为 p' ，则点 p 和 p' 之间的距离即为点 p 和直线 L 的距离。

如果两条直线互相垂直，其中一条直线与 X 轴平行或者垂直，则属于特殊情形，交点的坐标容易求得。若任意一条直线均不与 X 轴平行或垂直，那么两条直线斜率的乘积为 -1 。可以根据这个关系，先求出直线 L' 的方程，然后根据第 13 章中求两条直线交点的方法来求出点 p' 的坐标，然后再计算距离。

更为简便的方法是根据外积的性质来计算点和直线间的距离。

图 14-9 给定直线上的两个点 p_1 和 p_2 , 求直线外一点 p_0 和直线的距离

如图 14-9 所示, 外积在几何上是向量所围成的平行四边形的面积。图中三角形 $p_0p_1p_2$ 的面积为平行四边形面积的一半, 而 p_0 和直线间的距离恰为此三角形的高, 根据外积的定义和面积关系有

$$(p_0 - p_1) \times (p_2 - p_1) = 2 * S_{p_1p_2p_0} = |p_2 - p_1| * d$$

展开移项得到

$$d = \frac{|(x_2 - x_1)(y_1 - y_0) - (x_1 - x_0)(y_2 - y_1)|}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

使用代码表示为

```
double getDistPL(point p, line l) {
    return fabs(cross(l.p2 - l.p1, p - l.p1)) / abs(l.p2 - l.p1));
}
```

如果直线 L 给定的是一般方程, 则有

$$d = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}$$

强化练习: [1111 Trash Removal](#)^D, [10210 Romeo & Juliet](#)^C。

14.2.8 点和线段间距离

如图 14-10 所示, 给定线段 $s=p_1p_2$, 令经过线段 s 的直线为 L , 给定点 p , p 在直线 L 上的投影为 p' 。如果点 p' 在线段 s 上, 则点 p 和线段 s 的距离为 p 和 p' 的距离, 若 p' 不在线段 s 上, 则点 p 和 s 的距离为 p 和 s 的两个端点的距离的较小值。

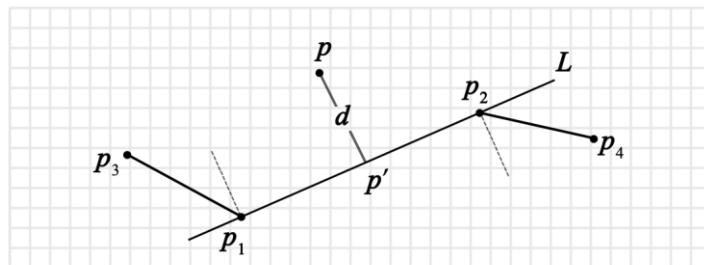


图 14-10 点和线段间距离的定义。当点 p 的投影位于线段 p_1p_2 上时, 点 p 和线段 p_1p_2 的距离为点 p 和投影点 p' 间的距离。当点 p 的投影位于线段外时, 则选择与较近的端点的距离作为点和线段的距离, 如图中的点 p_3 和点 p_4 , 点 p_3 距离线段的端点 p_1 较近, 则 p_3 与线段的距离为线段 p_3p_1 的长度, 同理, 点 p_4 和线段 p_1p_2 的距离为线段 p_4p_2 的长度

判断点的投影是否在线段 s 上有两种方法, 第一种方法是检查角 $\angle pp_1p_2$ 和角 $\angle pp_2p_1$ 是否均不大于 90 度, 如果满足此条件则表明 p 的投影在线段 s 上。在检查角度时, 可以不需要直接计算角度值, 而是使用余弦定理确定角度余弦值的符号, 根据余弦值的符号来判断角度是否小于等于 90 度。

```
double getDistPS(point p, segment s) {
    double cosa = norm(p - s.p1) + norm(s.p1 - s.p2) - norm(p - s.p2);
    double cosb = norm(p - s.p2) + norm(s.p1 - s.p2) - norm(p - s.p1);
    // 点 p 与端点的构成的夹角均不大于 90 度时, 点 p 的投影在线段上。
    if (cosa >= 0 && cosb >= 0) return getDistPL(p, s);
    return min(abs(p - s.p1), abs(p - s.p2));
}
```

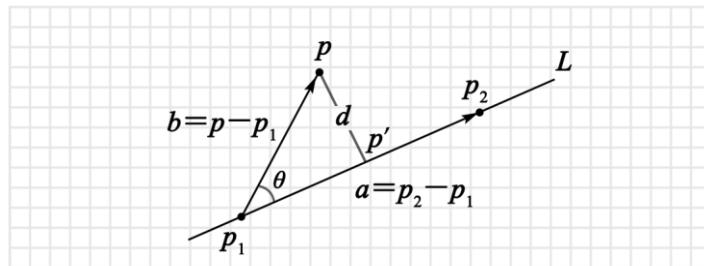


图 14-11 求点到线段间的距离

第二种方法是根据向量间的夹角来判断。如图 14-11 所示, 如果向量 $p_2 - p_1$ 与向量 $p - p_1$ 的夹角 θ 大于 90 度 (或者小于 -90 度), 则点 p 和线段 s 的距离就是 p 到点 p_1 的距离。

```
double getDistPS(point p, segment s) {
    if (dot(s.p2 - s.p1, p - s.p1) < 0.0) return abs(p - s.p1);
    if (dot(s.p1 - s.p2, p - s.p2) < 0.0) return abs(p - s.p2);
    return getDistPL(p, s);
}
```

扩展练习: 1039* Simplified GSM Network^D, [13117 ACIS A Contagious vIruS^D](#)。

14.2.9 线段和线段间距离

线段 s_1 和线段 s_2 的距离是以下四个距离中最小的一个: (1) 线段 s_1 与线段 s_2 的端点 $s_2.p_1$ 的距离; (2) 线段 s_1 与线段 s_2 的端点 $s_2.p_2$ 的距离; (3) 线段 s_2 与线段 s_1 的端点 $s_1.p_1$ 的距离; (4) 线段 s_2 与线段 s_1 的端点 $s_1.p_2$ 的距离。如果线段 s_1 和 s_2 相交, 则两者之间的距离为 0。

```
double getDistSS(segment s1, segment s2) {
    if (isIntersected(s1, s2)) return 0.0;
    return min(min(getDistPS(s2.p1, s1), getDistPS(s2.p2, s1)),
               min(getDistPS(s1.p1, s2), getDistPS(s1.p2, s2)));
}
```

扩展练习: 889* Islands^E, [10514 River Crossing^D](#)。

14.2.10 点和多边形的关系

如果一个多边形为凸多边形，且顶点按照逆时针顺序给出，则可以使用外积判定点 p 是否在在构成凸多边形的所有有向线段 pp_{i+1} 的左侧，如果满足此条件，那么点 p 在凸多边形内^I，但是这条判定原则不适用于任意多边形。

```
//-----14.2.10.1.cpp-----
const double EPSILON = 1E-7;

struct point { double x, y; };

typedef vector<point> polygon;

int cp(point a, point b, point c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

bool ccw(point a, point b, point c) { return cp(a, b, c) > EPSILON; }

// 确定点是否在凸多边形内，要求凸多边形的点按照逆时针顺序给出。
bool isPointInPolygon(point p, polygon pg) {
    bool in = true;
    for (int i = 0; i < pg.size() && in; i++)
        in &= ccw(pg[i], pg[(i + 1) % pg.size()], p);
    return in;
}
//-----14.2.10.1.cpp-----
```

强化练习：143 Orchard Trees^B, [476 Points in Figures: Rectangles](#)^A, 477 Points in Figures: Rectangles and Circles^A, 478 Points in Figures: Rectangles Circles and Triangles^A, 10112 Myacm Triangles^B。

扩展练习：[361 Cops and Robbers](#)^D, [10823 Of Circles and Squares](#)^D。

对于确定点 p 与任意多边形 P 的关系问题，有一个基于 Jordan 曲线定理（Jordan curve theorem）的方法。Jordan 曲线定理指出：每个多边形或其他闭合图形都有一个内部和外部。换句话说，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。那么，可以从给定点 q 出发作一条右水平射线（right horizontal ray） L ，如果该射线穿越边界的次数为偶数，则 q 在 P 的外部。因为射线的最远端必定在多边形外，从射线的最远处朝端点 q 处移动，每两次穿越边界都会重新回到多边形外部，若穿越的次数为奇数，则 q 在 P 的内部。只需要计算多边形每条边与该射线是否存在交点，并判断交点总个数的奇偶性即可确定点与多边形的位置关系。需要注意，在统计交点时，如果射线与多边形的某一条边重合则不算穿越，即不计为有交点。

根据 Jordan 曲线定理，可以将点和多边形的内外关系确定问题转化求线段是否相交的问题，关键在于如何将射线 L 转化为线段进行相交判断。可以确定多边形顶点中位于最右侧顶点的横坐标 x_{max} ，使用端点为 $q(x_q, y_q)$ 和 $e(x_{max}, y_q)$ 的线段来表示射线 L ，通过这条线段逐一与多边形的边进行相交测试并统计交点个数，根据交点个数的奇偶性即可判断点 q 与多边形 P 的内外关系。

强化练习：[1641* ASCII Area](#)^D, [858 Berry Picking](#)^D。

^I 如果顶点按照顺时针顺序给出，则判断点 p 是否在构成凸多边形的所有有向线段 pp_{i+1} 的右侧。

需要注意, 当经过点 q 的右水平射线与多边形的某一顶点相交时, 如何为交点计数。如果简单地认为只要相交就计为交点, 则会得出错误的结论。如图 14-12 (d) 所示, 射线与边 a 、 b 、 c 各有一个交点, 则总交点数为 3 个, 似乎可以得出“点 q 在多边形内”的结论, 但从图示可以看出, 结论显然是错误的。故需要作以下的预设规定以保证正确性:

(1) 当射线与多边形的边重合时, 规定交点个数为 0 个;

(2) 当射线与多边形相交于某一顶点时, 对于在射线上方的线段, 规定其交点个数为 1 个, 对于在射线下方的线段, 规定其交点个数为 0 个(亦可规定在射线下方的线段交点个数为 1 个, 处于上方的线段交点个数为 0 个, 不影响正确性)。

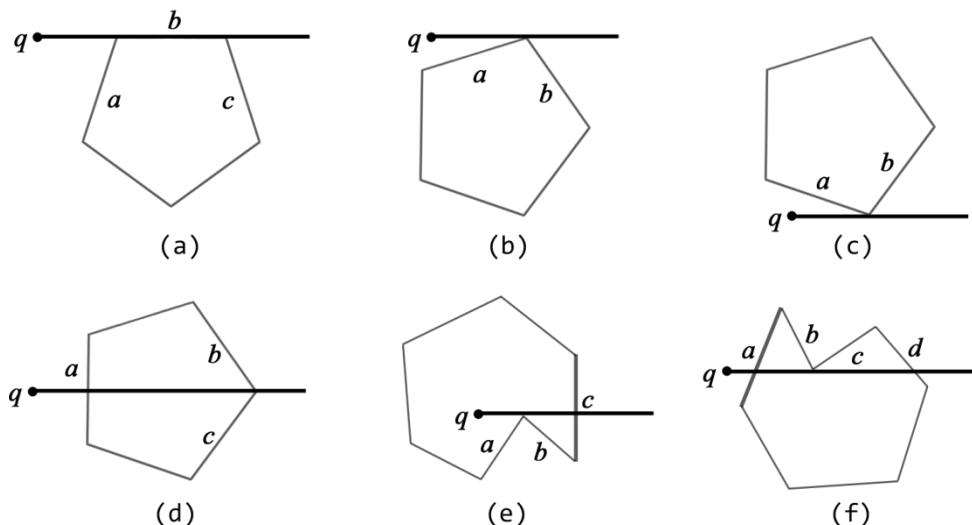


图 14-12 右水平射线与多边形相交的几种特殊情况。根据交点计数规则有: (a) 线段 a 与 q 相交且位于射线下方, 交点个数为 0; 线段 b 与射线 q 重合, 交点个数为 0; 线段 c 与射线 q 相交且位于射线下方, 交点个数为 0; 总交点个数为 0, 为偶数, 故点 q 在多边形外, 结论正确。(b) 根据计数规则, 交点个数为 0, 点 q 在多边形外。(c) 交点个数为 2, 点 q 在多边形外。(d) 交点个数为 2, 点 q 在多边形外。(e) 交点个数为 1, 点 q 在多边形内。(f) 交点个数为 4, 点 q 在多边形外

以下为参考实现代码:

```
//-----14.2.10.2.cpp-----//
const double EPSILON = 1E-7;

typedef vector<point> polygon;

// 判断多边形的边是否在射线上方。
bool isSideAboveRay(segment ray, segment side) {
    return side.p1.y >= ray.p1.y && side.p2.y >= ray.p1.y;
}

// 根据设定的交点数规则判断射线与多边形的边是否相交。
bool segmentsIntersected(segment ray, segment side) {
    double cp1 = cp(ray.p1, ray.p2, side.p1), cp2 = cp(ray.p1, ray.p2, side.p2);
    double cp3 = cp(side.p1, side.p2, ray.p1), cp4 = cp(side.p1, side.p2, ray.p2);
```

```

// 跨越式相交。
if ((cp1 * cp2 < 0) && (cp3 * cp4 < 0)) return true;
// 不相交。
if ((cp1 * cp2 > 0) || (cp3 * cp4 > 0)) return false;
// 共线，不论线段是否重合，均规定为不相交。
if ((fabs(cp1) <= EPSILON && fabs(cp2) <= EPSILON) ||
    (fabs(cp3) <= EPSILON && fabs(cp4) <= EPSILON))
    return false;
// 相交于顶点，判断是否在射线上方。
if (fabs(cp1) <= EPSILON || fabs(cp2) <= EPSILON ||
    fabs(cp3) <= EPSILON || fabs(cp4) <= EPSILON)
    return isSideAboveRay(ray, side);
return false;
}

// 测试点 p 是否在多边形内，如果点 p 在多边形上，不计为在多边形内。
bool isPointInPolygon(point p, polygon pg) {
    // 找到多边形顶点中位于最右边的点的横坐标。
    double rightX = pg[0].x;
    for (int i = 0; i < pg.size(); i++) {
        if (pg[i].x > rightX)
            rightX = pg[i].x;
    }
    // 统计多边形的边与射线的交点数量。
    // 将位于最右边点的横坐标的两倍作为射线右端点的横坐标，避免一些退化情况的发生。
    int numberofIntersection = 0;
    segment ray = (segment){p, (point){2 * rightX, p.y}};
    for (int i = 0; i < pg.size(); i++) {
        segment side = (segment){pg[i], pg[(i + 1) % pg.size()]};
        if (segmentsIntersected(ray, side))
            numberofIntersection++;
    }
    // 测试交点个数奇偶性。
    return ((numberofIntersection & 1) == 1);
}
//-----14.2.10.2.cpp-----//

```

上述实现利用线段相交的方法来确定射线与多边形边是否有交点，并且规定当射线与边重合时交点数为0，射线和边的端点相交时，如果边在射线上方，交点数为1，否则为0。如果结合前述介绍的向量内积和外积，可以有更为简洁的实现。对于构成多边形各边的线段 $g_i g_{i+1}$ ，令 $g_i - p$ 与 $g_{i+1} - p$ 分别为向量 a 和向量 b ，点 p 是否位于 $g_i g_{i+1}$ 上可以通过判断逆时针旋转（前述介绍的判断谓词 ccw ）的方法进行检查，即检查 a 和 b 是否在同一直线上且方向相反。如果 a 和 b 外积大小为0且内积小于等于0，则点 p 位于 $g_i g_{i+1}$ 上。

射线和线段 $g_i g_{i+1}$ 是否相交，可以通过 a 和 b 构成的平行四边形的面积正负，即 a 和 b 外积的大小来判断。首先调整向量 a 和 b ，将坐标中 y 值较小的向量定为 a ，接下来，如果 a 和 b 的外积大小为正（ a 到 b 为逆时针）且 a 和 b 的终点位于射线的两侧，即可确定射线与边交叉。但是需要注意边界情形的处理，即线段与射线相交于线段端点的情形，这个可以通过判断向量 a 和 b 的 y 值以及外积来确定。

```

//-----14.2.10.3.cpp-----//
const int OUT = 0, ON = 1, IN = 2;
const double EPSILON = 1e-7;

// 测试点 p 是否在多边形内，如果点 p 在多边形上，不计为在多边形内。

```

```

int isPointInPolygon(point p, polygon pg) {
    bool in = false;
    for (int i = 0; i < pg.size(); i++) {
        point a = pg[i] - p, b = pg[(i + 1) % pg.size()] - p;
        // 当给定的坐标值均为整数时, 建议使用:
        // if (cross(a, b) == 0 && dot(a, b) <= 0) return ON;
        if (abs(cross(a, b)) < EPSILON && dot(a, b) < EPSILON) return ON;
        if (a.y > b.y) swap(a, b);
        // 当给定的坐标值均为整数时, 建议使用:
        // if (a.y <= 0 && 0 < b.y && cross(a, b) > 0) in = !in;
        if (a.y < EPSILON && EPSILON < b.y && cross(a, b) > EPSILON) in = !in;
    }
    return in ? IN : OUT;
}
//-----14.2.10.3.cpp-----

```

1749 Airport Construction^E (机场建设)

位于热带的 Piconesia 岛国以其秀丽的沙滩、繁茂的植被、可可与咖啡原料的产出以及全年宜人的天气而闻名海外。天堂般美丽的这里将纳入未来举办 ACM-ICPC 世界总决赛的地点考虑名单中（至少也会是执行委员会度假地点的选择之一）。但是还有一个小问题：这个岛真的很难到达。

就目前而言，最快的方式是从最近的机场出发，并借助渔船、油轮、皮划艇和潜艇这些工具，花费三天时间才能到达小岛。为了让大家更轻松地参加 ICPC 世界总决赛，也为了推动小岛的旅游业发展，Piconesia 正在计划修建它的第一个机场。

考虑到越长的起降跑道越有助于对大型飞机的支持，Piconesia 决定在其岛上修建一条最长的起降跑道带。不幸地是，他们无法确定这条跑道应该设置在哪里。也许你可以帮个忙？

对于本题，我们将 Piconesia 的边界抽象成一个边形的模型。根据这个边形，你需要计算出最长的跑道长度（即最长的线段）使得跑道可以完全在小岛上修建。

输入

第一行包含一个整数 n ($3 \leq n \leq 200$)，表示多边形的点数。

接下来 n 行，每行包含两个整数 x 和 y ($|x|, |y| \leq 10^6$)，表示多边形上的顶点 (x, y) ，顶点按照逆时针顺序给出。

给出的多边形是简单多边形，即它的顶点互不相同，并且除了相邻的两条边在公共点相交之外，它的任意两条边都不会相交。此外，相邻的两条边也不会共线。

输出

输出能放进给定多边形内部的最长线段长度，其绝对误差或相对误差不能超过 10^{-6} 。

样例输入

```

7
0 20
40 0
40 20
70 50
50 70
30 50
0 50

```

样例输出

```
76.157731059
```

分析

首先可以明确的是：在简单多边形内的最长线段的两个端点不一定是多边形的顶点，但多边形内最长的线段一定会经过多边形的至少两个顶点。若不然，将该线段顺时针（或者逆时针）旋转，后续得到的线段必定会比原来的线段更长。因此，朴素的做法是以任意两个多边形的顶点作为线段所经过的顶点，确定这两个顶点间的线段是否有部分在多边形外。与此同时，还需确定线段向两端的延长线与多边形的交点，有可能最长的线段的端点是在多边形的边上而不是顶点上。由于测试网站上的数据量较大，朴素的 $O(n^4)$ 算法难以获得通过，需要较为精巧的实现，即需要 $O(n^3)$ 的算法才能获得 Accepted。

强化练习：[634 Polygon^B](#)，[12016 Herbicide^E](#)。

扩展练习：[248* Cutting Corners^E](#)，[10256 The Great Divide^D](#)，[10321* Polygon Intersection^D](#)，[12931* Common Area^E](#)。

14.2.11 直线和圆的交点

直线和圆的关系有三种：不相交、相切、相交。根据圆心和直线的距离 d 以及圆的半径 r 可以容易地判断两者之间的关系。

- (1) 当 $d > r$ 时，直线与圆不相交；
- (2) 当 $d = r$ 时，直线与圆相切，即只有一个交点，交点的坐标恰为圆心在直线上的投影，可以根据前述介绍求点的投影的方法得到切点坐标；
- (3) 当 $d < r$ 时，直线和圆相交，有两个不同的交点，可以按照下述方法求出交点的坐标。

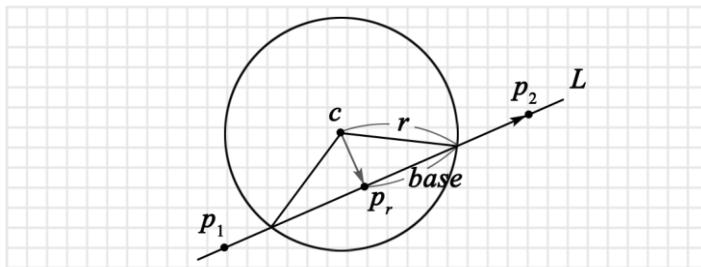


图 14-13 直线和圆的交点

如图 14-13 所示，首先求出圆心 c 在直线 L 上的投影点 p_r ，然后求出直线 L 上的单位向量 e ，单位向量可以根据向量 p_1p_2 和其模的比值得到，即

$$e = \frac{p_2 - p_1}{|p_2 - p_1|}$$

接着，根据半径 r 与向量 p_r 的长度计算出圆内线段的一半，令其为 $base$ ，将单位向量 e 乘以系数 $base$ ，即可得到直线 L 上与 $base$ 同样大小的向量，最后，以投影点 p_r 为起点，向正/负方向加上该向量，即可得到圆与直线的交点坐标。

```
//-----14.2.11.cpp-----/
struct circle { point c; double r; };
pair<point, point> getIntersection(line l, circle c) {
    point pr = project(c.c, l);
    e = (l.p2 - l.p1) / abs(l.p2 - l.p1);
    double base = sqrt(c.r * c.r - norm(pr - c.c));
    return make_pair(pr + e * base, pr - e * base);
}
```

```

}
//-----14.2.11.cpp-----//

```

14.2.12 圆和圆的交点

在第 13 章中, 已经介绍了使用余弦定理求两个圆交点的方法, 这里介绍使用向量和余弦定理求圆交点的方法。

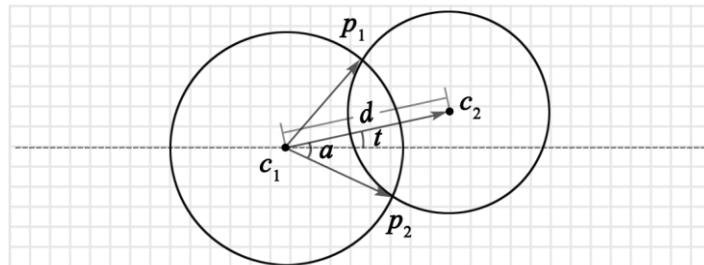


图 14-14 圆和圆的交点

如图 14-14 所示, 由两个圆心和其中一个交点构成的三角形 $c_1c_2p_1$ 的三条边分别为 $c_1.r$ 、 $c_2.r$ 、 d , 根据余弦定理可以求出向量 $c_2.c - c_1.c$ 与 $c_1.c$ 到某个交点的向量的夹角 a , 然后再求出 $c_2.c - c_1.c$ 与 X 轴的夹角 t 。最后所求的交点就是以圆心 $c_1.c$ 为起点, 大小为 $c_1.r$, 与 X 轴正方向夹角为 $t+a$ 和 $t-a$ 的两个向量。

```

//-----14.2.12.cpp-----//
struct circle { point c; double r; };
double atan2(point p) { return atan2(p.y, p.x); }
point polar(double d, double t) { return point(d * cos(t), d * sin(t)); }

pair<point, point> getIntersection(circle c1, circle c2) {
    double d = abs(c2.c - c1.c);
    double a = acos((c1.r * c1.r + d * d - c2.r * c2.r) / (2 * c1.r * d));
    double t = atan2(c2.c - c1.c);
    return make_pair(c1.c + polar(c1.r, t + a), c1.c + polar(c1.r, t - a));
}
//-----14.2.12.cpp-----//

```

14.2.13 圆的切点

过圆外一点可以引两条不同的直线与圆相切, 从而产生两个切点。使用类似于求圆与圆的交点坐标的方法, 可以方便地求得切点的坐标。

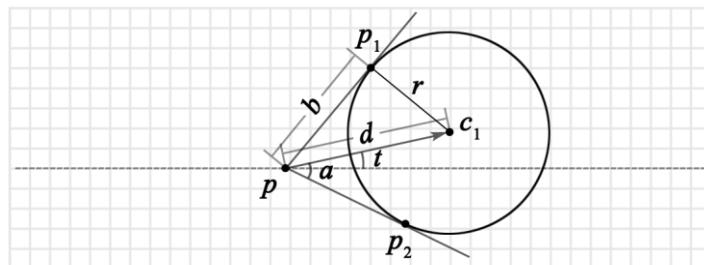


图 14-15 过圆外一点 p 作与圆相切的两条直线, 求切点的坐标

```
//++++++14.2.13.cpp+++++++
struct circle { point c; double r; };
double atan2(point p) { return atan2(p.y, p.x); }
point polar(double d, double t) { return point(d * cos(t), d * sin(t)); }

pair<point, point> getIntersection(point p, circle cc) {
    double d = abs(cc.c - p);
    double a = asin(cc.r / d);
    double t = atan2(cc.c - p);
    double b = sqrt(d * d - cc.r * cc.r);
    return make_pair(p + polar(b, t + a), p + polar(b, t - a));
}
```

但在实际解题过程中，由于上述求切点坐标的运算过程中使用了三角函数，很容易导致精度问题，更好的方法是解题思路不变，转而使用坐标旋转来确定切点坐标，由于计算过程中仅使用四则运算和求平方根运算，引入的误差较前述方法要少，不易因浮点数的精度问题而导致 Wrong Answer。

```
pair<point, point> getIntersection(point p, circle cc) {
    point pv = cc.c - p;
    double d = abs(pv);
    double b = sqrt(norm(pv) - cc.r * cc.r);
    double sina = cc.r / d, cosa = b / d;
    double k = b / d;
    point p1 = point(pv.x * cosa - pv.y * sina, pv.x * sina + pv.y * cosa);
    point p2 = point(pv.x * cosa + pv.y * sina, -pv.x * sina + pv.y * cosa);
    return make_pair(p + p1 * k, p + p2 * k);
}
//++++++14.2.13.cpp+++++++

```

扩展练习：10011* Where Can You Hide^D，10674* Tangents^D。

14.3 扫描线算法

求 n 条线段的交点，朴素的方式是对任意两条线段求交点，算法时间复杂度为 $O(n^2)$ ，当 n 值较大时无法在限制时间内进行处理。如果给定的线段与坐标轴平行，则求这类线段的相交问题特称为曼哈顿几何^I，可以使用扫描线算法（sweep line algorithm）高效地求解。扫描线算法的思路是将一条与 X 轴（或 Y 轴）平行的直线向上（向右）平行移动，在移动过程中寻找交点，这条直线就称为扫描线。

扫描线并不会按照固定的间隔进行扫描，而是在每次遇到线段的端点时停止移动，然后检查该位置上的线段交点。为了便于进行上述处理，需要先将输入的线段端点按照 y 值排序，让扫描线沿 Y 轴正方向移动。

在扫描线移动过程中，算法会将扫描线穿过的垂直线段（与 Y 轴平行）临时记录下来，等到扫描线与水平线段（与 X 轴平行）重叠时，检查水平线段的范围内是否存在垂直线段上的点，然后将这些点作为交点输出。为提高处理效率，可以应用二叉搜索树来保存扫描线穿过的垂直线段。可将线段相交问题的扫描线算法步骤归纳如下：

^I 曼哈顿几何（Manhattan geometry），19 世纪由德国人 Hermann Minkowski 提出，又称出租车几何（taxicab geometry），因位于纽约的曼哈顿岛上横平竖直呈网格状的街道而得名。其距离度规为两点直角坐标系坐标差值绝对值的和，称为曼哈顿距离。在平面上，点 p_1 和 p_2 之间的 L_m 距离由下式给出： $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{1/m}$ ，实际上，欧几里得距离为 L_2 距离，曼哈顿距离为 L_1 距离。

- (1) 将输入线段的端点按 y 坐标升序排列, 添加到线段端点列表 EP 中;
 (2) 将二叉搜索树 BT 置为空;
 (3) 按顺序取出 EP 中的端点 (相当于让扫描线自下而上进行移动), 进行下述处理
 (3a) 如果取出的端点为垂直线段的上端点, 则从 BT 中删除该线段的 x 坐标;
 (3b) 如果取出的端点为垂直线段的下端点, 则将该线段的 x 坐标插入 BT ;
 (3c) 如果取出的端点为水平线段的左端点 (扫描线与水平线段重合), 将该水平线段的两端点
 作为搜索范围, 输出 BT 中包含的值 (即垂直线段的 x 坐标)

如果使用平衡二叉树进行上述操作, 一次搜索操作的时间复杂度为 $O(\log n)$, 则 n 条线段总共所需要操作的时间复杂度为 $O(n \log n)$, 而算法整体的复杂度还与交点数 k 有关, 故扫描线算法求线段交点的时间复杂度为 $O(k + n \log n)$ 。

```
//-----14.3.cpp-----//
const int MAXV = 100000, INF = 0x3f3f3f3f;
const int BOTTOM = 0, LEFT = 1, RIGHT = 2, TOP = 3;

struct EndPoint {
    point p;
    int segIdx, epCode;
    EndPoint(point p, int epIdx, int epCode):
        p(p), epIdx(epIdx), epCode(epCode) {}
    // 保证端点按照下、左、右、上的顺序添加到二叉树中。
    bool operator<(const EndPoint &ep) const {
        if (p.y == ep.p.y) return epCode < ep.epCode;
        return p.y < ep.p.y;
    }
} EP[2 * MAXV];

int manhattanIntersection(vector<segment> S) {
    int n = S.size();
    // 逐条线段进行处理。
    for (int i = 0, k = 0; i < n; i++) {
        if (S[i].p1.y == S[i].p2.y) {
            if (S[i].p1.x > S[i].p2.x) swap(S[i].p1, S[i].p2);
        } else {
            if (S[i].p1.y > S[i].p2.y) swap(S[i].p1, S[i].p2);
        }
        // 注意端点的添加顺序对边界情形的正确处理非常重要。
        if (S[i].p1.y == S[i].p2.y) {
            EP[k++] = EndPoint(S[i].p1, i, LEFT);
            EP[k++] = EndPoint(S[i].p2, i, RIGHT);
        } else {
            EP[k++] = EndPoint(S[i].p1, i, BOTTOM);
            EP[k++] = EndPoint(S[i].p2, i, TOP);
        }
    }
    // 将所有端点按照从下至上的顺序排列, 如果两个端点具有相同的  $y$  坐标则按照从左至右的顺序排列。
    sort(EP, EP + (2 * n));
    // 二叉搜索树。插入 INF 作为哨兵元素以便计数。
    set<int> BT; BT.insert(INF);
    int cnt = 0;
    // 逐个端点处理。
    for (int i = 0; i < 2 * n; i++) {
        if (EP[i].epCode == TOP) {
```

```

        BT.erase(EP[i].p.x);
    } else if (EP[i].epCode == BOTTOM) {
        BT.insert(EP[i].p.x);
    } else if (EP[i].epCode == LEFT) {
        auto begin = BT.lower_bound(S[EP[i].epIdx].p1.x);
        auto end = BT.upper_bound(S[EP[i].epIdx].p2.x);
        cnt += distance(begin, end);
    }
}
// 返回交点的数量。
return cnt;
}
//-----14.3.cpp-----//

```

除了求线段交点外，在执行扫描线算法的过程中，还可以得到一个“副产品”——即这些线段相交所能构成的最大矩形的面积（或者所有矩形面积的并）。

对于一般情形，即线段不一定和坐标轴平行的时候如何处理呢？这时就需要将前述给出的扫描线算法进行适当拓展，使之成为更一般的扫描线算法^[160]。

强化练习：105 The Skyline Problem^A，688 Mobile Phone Coverage^C，10191 Longest Nap^A，11661 Burger Time^A，11683 Laser Sculpture^B。

扩展练习：1382* Distant Galaxy^D。

14.4 坐标离散化

离散化（discretization），原本是指将无限空间中有限的个体映射到有限的空间中去，其目的是提高算法的效率。换句话说，离散化就是在不改变数据相对大小的前提下，将数据进行压缩。坐标离散化^I，沿用的是离散化的思想，其核心是将分布范围较大但数量相对较少的数据进行集中处理，以提高程序的空间使用效率，同时也能加快运行时间速度，其最终结果是在原坐标与新坐标之间建立起一种映射。这个技巧在处理某些特定类型的题目时非常有用。

给定如下的网格，网格中有 n 条垂直或水平宽度为 1 的线段划分了整个网格，要求确定这 n 条线段将网格划分成的区域的个数^{II}。网格的长度 w 和宽度 h 为闭区间 $[1, 1000000]$ 之内的整数，线段使用其端点坐标来表示，其中 (x_1, y_1) 表示线段的左端点或者上端点， (x_2, y_2) 表示线段的右端点和下端点，线段的数量 $n \leq 500$ 。网格的左上角坐标为 $(1, 1)$ ，X 轴的正向向右，Y 轴的正向向下。

^I 从望文生义的角度，坐标本来已经是离散的，坐标离散化似乎是要将单个坐标“离散”为多个坐标，但实际却恰恰相反。

^{II} 题目来源于秋叶拓哉、岩田阳一、北川宜稔著，巫泽俊、庄俊元、李津羽译，《挑战程序设计竞赛，第 2 版》，第 164 页。

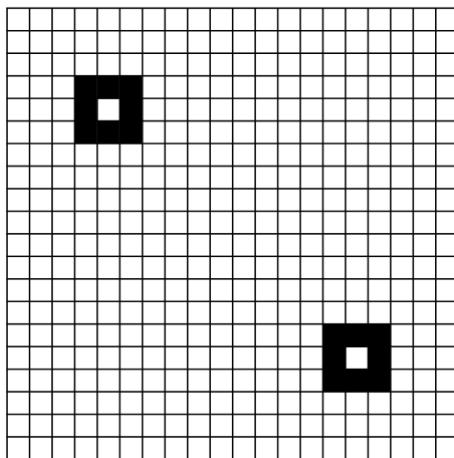


图 14-16 一个 20×20 的网格，其中有若干线段构成的两个正方形区域，将整个网格划分为 3 个区域。朴素的方法是逐一对未占用方格使用洪泛法进行填充，计数填充次数，对于 1000×1000 的网格尚且可行，但是对于 1000000×1000000 的网格在时间和空间限制条件下则明显不可行

通常的做法是使用 $w \times h$ 的二维数组来表示整个网格，然后使用 Flood-Fill 算法来计数区域个数。但是由于 w 和 h 最大可为 1000000 ，创建这样的二维数组显然会超过内存限制。使用坐标离散化，可以将前后没有变化的行列予以消除，这样并不会影响区域的个数（即拓扑结构）。实现过程为：枚举已使用的坐标，对坐标排序，然后去除重复坐标，最后索引坐标离散化后所对应的值。以横坐标 x 为例，将每个使用的 x 坐标的前一列和后一列标记为使用，置入一个数组中，使用 `sort` 进行排序，然后使用 `unique` 将重复的坐标予以去除，则剩余的坐标值是真正有效且被使用的，在此坐标数组中找到原坐标的序号，此序号就构成了原坐标和新坐标之间的一种映射。

```
//++++++14.4.cpp+++++++
// 对坐标数组 x1 和 x2 所包含的坐标进行离散化，返回离散化之后所有新坐标的数量。
int compress(int *x1, int *x2, int w, int n) {
    vector<int> xs;
    for (int i = 0; i < n; i++) {
        // 将原坐标原位偏移一个位置，若在网格范围内则将其添加到已用坐标中。
        for (int d = -1; d <= 1; d++) {
            int tx1 = x1[i] + d, tx2 = x2[i] + d;
            if (0 <= tx1 && tx1 < w) xs.push_back(tx1);
            if (0 <= tx2 && tx2 < w) xs.push_back(tx2);
        }
        // 排序，去除重复的坐标。
        sort(xs.begin(), xs.end());
        xs.erase(unique(xs.begin(), xs.end()), xs.end());
        // 在新坐标中索引原有坐标，因为坐标是有序的，使用 lower_bound 查找可提高效率。
        for (int i = 0; i < n; i++) {
            x1[i] = lower_bound(xs.begin(), xs.end(), x1[i]) - xs.begin();
            x2[i] = lower_bound(xs.begin(), xs.end(), x2[i]) - xs.begin();
        }
        // 返回新坐标的数量。
        return xs.size();
    }
}
```

经过离散化处理后，未使用的“连续成片”区域的坐标被压缩为单个坐标，这样明显减少了表示所需要的坐标数量，从而减少了存储空间需求，同时需要枚举的坐标数量也减少了，运行效率自然得到提高。

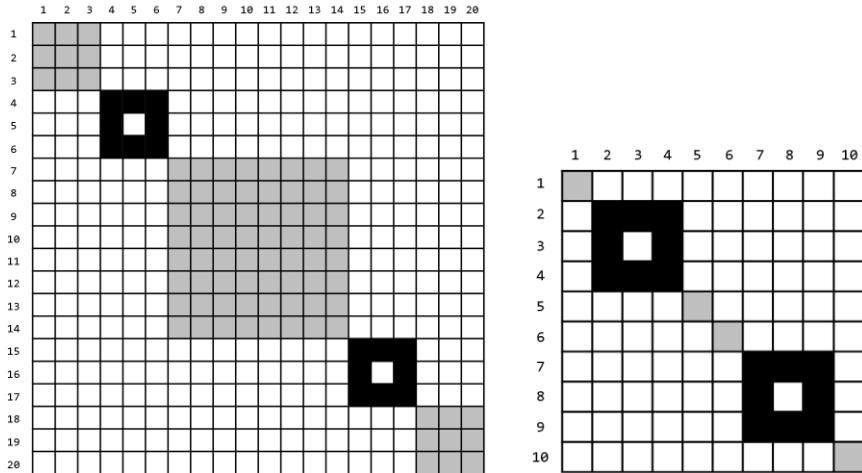


图 14-17 经过坐标离散化处理后，左侧 20×20 的网格被“压缩”为右侧等价的 9×9 网格。当原有的网格越大，“压缩”效果越明显。左侧的三个灰色区域经过压缩后对应右侧的三个灰色区域。离散化处理会对原有图形产生横向和纵向变形作用，但是不会改变原有图形的拓扑结构，且原图中的直线在处理后仍会保持直线形状，只不过比例会发生变化

利用坐标离散化技巧，可以有效处理矩形的面积并问题。如图 14-17 所示，在给定的平面上有 n 个矩形，矩形由其左下角的坐标 (x_1, y_1) 和右上角的坐标 (x_2, y_2) 所决定，坐标值均为整数，求这 n 个矩形所覆盖的总面积，其中 $0 \leq n \leq 100$ ， $-10^8 \leq x_1 < x_2 \leq 10^8$ ， $-10^8 \leq y_1 < y_2 \leq 10^8$ ，注意矩形所覆盖的面积可能会发生重叠。

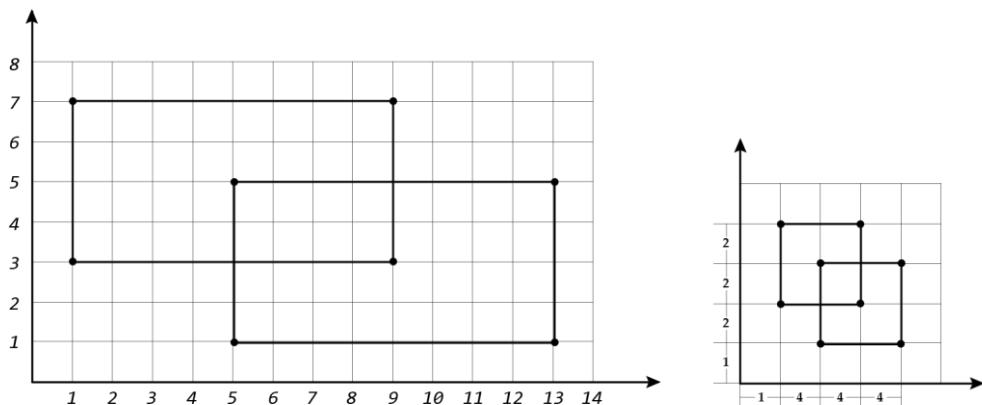


图 14-18 经过坐标离散化处理后，在 X 和 Y 轴上，两个相邻新坐标之间序号的差值虽然为 1，但是其实际对应的距离却很可能不再是 1，而是对应着原坐标中两个坐标之间的实际差值（右图给出了经过离散化处理后相邻两个坐标间的实际差值）

如果使用朴素的方法对其进行填充然后计数覆盖的方格，因为坐标范围在 $[-10^8, 10^8]$ ，显然不可行。可以使用坐标离散化的技巧解决这个问题。在此，可以采用更为“激进”的离散化——将所有矩形的坐标紧密排列，两个坐标间的空白区域压缩为单个间隔。这样可将原来最多 100 个矩形的坐标映射到一个 200×200 的二维数组中，每个原坐标对应 $[0, 2n]$ 中的一个整数。之后使用通常的填充法标记每个矩形覆盖的方格，最后逐个枚举被覆盖的方格，计算总的覆盖面积。

```

const int MAXV = 210;

int main(int argc, char *argv[]) {
    int n;
    int x1[MAXV], y1[MAXV], x2[MAXV], y2[MAXV];
    // 读入数据。
    while (cin >> n) {
        vector<int> xs, ys;
        for (int i = 0; i < n; i++) {
            cin >> x1[i] >> y1[i];
            cin >> x2[i] >> y2[i];
            xs.push_back(x1[i]); xs.push_back(x2[i]);
            ys.push_back(y1[i]); ys.push_back(y2[i]);
        }
        // 排序，去除重复坐标。
        sort(xs.begin(), xs.end());
        sort(ys.begin(), ys.end());
        xs.erase(unique(xs.begin(), xs.end()), xs.end());
        ys.erase(unique(ys.begin(), ys.end()), ys.end());
        // 对坐标重新索引，得到新坐标。
        for (int i = 0; i < n; i++) {
            x1[i] = lower_bound(xs.begin(), xs.end(), x1[i]) - xs.begin();
            x2[i] = lower_bound(xs.begin(), xs.end(), x2[i]) - xs.begin();
            y1[i] = lower_bound(ys.begin(), ys.end(), y1[i]) - ys.begin();
            y2[i] = lower_bound(ys.begin(), ys.end(), y2[i]) - ys.begin();
        }
        // 标记被覆盖的“方格”。
        int g[2 * n][2 * n] = {};
        for (int c = 0; c < n; c++)
            for (int i = x1[c]; i < x2[c]; i++)
                for (int j = y1[c]; j < y2[c]; j++)
                    g[i][j]++;
        // 统计被覆盖“方格”的面积。
        int area = 0;
        for (int i = 0; i < 2 * n - 1; i++)
            for (int j = 0; j < 2 * n - 1; j++)
                if (g[i][j])
                    area += (xs[i + 1] - xs[i]) * (ys[j + 1] - ys[j]);
        cout << area << '\n';
    }
    return 0;
}
//+++++++++++++14.4.cpp+++++*/

```

在有关坐标离散化的题目中，一个明显特征是题目约束中，所给定的图或者网格的长和宽的范围都较大（一般为 10^6 数量级）或者坐标为实数值，这样使得通常利用二维数组表示整个图或网格的方法不再可行，此时可以考虑使用坐标离散化，降低解题的难度。与坐标离散化经常一同出现的是 Flood-Fill 算法。一般解

题思路是应用坐标离散化，将原图形进行“压缩”，从而将题目转化为可使用 Flood-Fill 算法解决的问题。

强化练习：221 Urban Elevations^C, 904 Overlapping Air Traffic Control Zones^D, 934 Overlapping Areas^E。

扩展练习：870 Intersecting Rectangles^D, 1092 Tracking Bio-Bots^D, 12171* Sculpture^D。

14.4.1 最大化矩形问题

给定一个 $w \times h$ 的网格以及网格上 n 个点的坐标， $0 \leq w, h \leq 10^8$, $2 \leq n \leq 1000$, $0 \leq x < w$, $0 \leq y < h$ 。要求确定一个具有最大面积的矩形 R ，要求 R 的水平边和竖直边分别与 X 轴和 Y 轴平行，其内部不能包含任何给定的点，但点可以位于矩形的边界上。

目标是寻找时间复杂度至少为 $O(n^2)$ 的算法。由于矩形面积需要尽可能的大，则满足要求的矩形边界必定经过某个点或者与网格的边界重合，如果不是这样，则可以将这些边进一步向外扩展，直到“碰”到某个点或者网格的边界，此时矩形的面积必定比原有矩形的面积要大，则与原矩形是满足要求的最大面积矩形相矛盾，因此，具有最大面积的矩形的所有边界必定经过某个点或者与网格边界重合。由于 $2 \leq n \leq 1000$ ，可以逐一枚举经过某点的水平线（垂直线）作为矩形的边界，以经过其他点的水平线（垂直线）作为矩形的相对边界，计算面积，获得最优值。为了避免重复计算，达到有序枚举的目的，需要按照枚举的方向分别对坐标按照 X 坐标和 Y 坐标进行排序。

- (1) 将所有点的坐标保存在数组 P 中；
- (2) 对坐标数组 P 按照 X 坐标升序排列；
- (3) 逐一从左至右枚举每一个点 $P[i]$ 作为矩形的左边界，初始时，矩形的下边界设定为直线 $y_1=0$ ，上边界设定为直线 $y_2=h$ ，以后续的点 $P[j]$ 作为矩形的右边界， $j > i$ ，则当前矩形的面积 $A = (P[j].x - P[i].x) \times (y_2 - y_1)$ ；
- (4) 根据位于矩形右边界上的点 $P[j]$ 的纵坐标更新矩形的下边界和上边界；
- (5) 对坐标数组 P 按照 Y 坐标升序排列；
- (6) 逐一从下到上枚举每一个点 $P[i]$ 作为矩形的下边界，初始时，矩形的左边界设定为直线 $x_1=0$ ，右边界设定为直线 $x_2=w$ ，以后续的点 $P[j]$ 作为矩形的上边界， $j > i$ ，则当前矩形的面积 $A = (P[j].y - P[i].y) \times (x_2 - x_1)$ ；
- (7) 根据位于矩形上边界上的点 $P[j]$ 的横坐标更新矩形的左边界和右边界；
- (8) 取两个方向上搜索得到的矩形面积的最优值。

需要注意，在 X 轴方向搜索最大面积矩形，使用位于矩形右侧边界的点 $P[j]$ 更新矩形的下边界和上边界时，若 $P[j]$ 的纵坐标与 $P[i]$ 的纵坐标相同，需要指定只更新下边界或者上边界。如果同时更新上下边界，则矩形的上下边界的距离将缩减为 0，会使得后续搜索得到的矩形面积一直为 0（因为不恰当的更新导致 $y_1 = y_2$ ），从而得到错误的结果。同样的，在 Y 轴方向搜索最大面积矩形时也需要注意上述问题。

强化练习：1312 Cricket Field^D, 10043 Chainsaw Massacre^D。

14.4.2 矩形并的面积

矩形并的面积问题是指给定一组边与坐标轴平行的若干矩形，要求确定这些矩形所覆盖的面积。

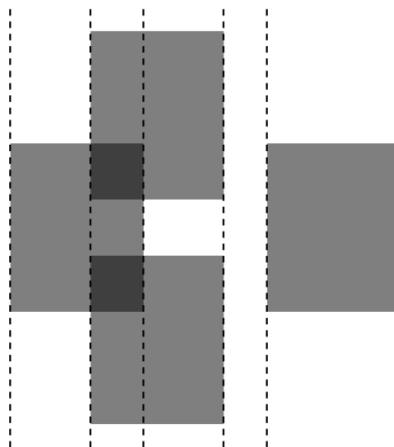


图 14-19 矩形并的面积。将矩形的竖直边延长为扫描线，不难得出，矩形并的面积等于相邻两条扫描线之间所夹的灰色“条带样”区域面积的总和

如果给定的矩形数量 n 较少（例如 $n \leq 100$ ），同时矩形的顶点坐标 (x_i, y_i) 为整数且范围不大 $(-100 \leq x_i, y_i \leq 100)$ ，则可以使用朴素的“填充计数法”来统计所有矩形覆盖的面积，即先将所有网格初始化为未填充，然后将各个矩形所覆盖的网格标记为已填充，最后统计已填充的网格数量，但是当坐标取值范围较大或者坐标值为实数时，“网格计数法”将无法应用。

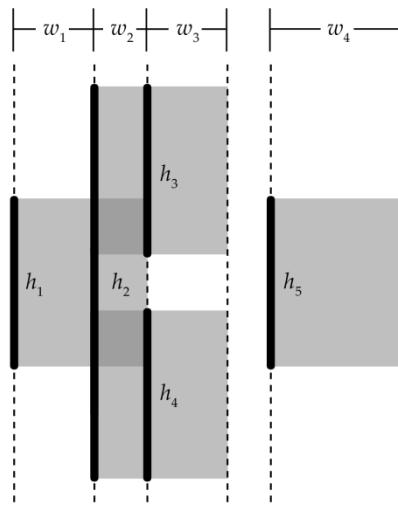


图 14-20 将矩形并的面积拆分为若干子矩形后再进行面积的求和。由于相邻两条扫描线之间的距离 w_i 容易求出，只需知道矩形竖直边覆盖得到的“粗实线”的高度 h_j 即可，而 h_j 可以通过线段树获得

利用扫描线算法和坐标离散化技巧可以巧妙地处理矩形并的面积问题。在扫描线从左至右扫描的过程中，如果遇到矩形的左边界则将其添加到线段树中，若遇到矩形的右边界则将其从线段树中删除。将扫描线遇到矩形定义为事件，一个矩形可以拆分为两个事件，左边的竖直边为添加事件，右边的竖直边为删除事件。定义以下相应的结构体来表示矩形和事件：

```

// 矩形。(x1, y1) 表示矩形的左下角坐标, (x2, y2) 表示矩形的右上角坐标。
struct rectangle {
    int x1, y1, x2, y2;
};

// 事件。
// x 表示竖直边的 X 坐标, y1 和 y2 表示竖直边两个端点的 Y 坐标。
// evtCode 为 1 或 -1, 1 表示添加事件, -1 表示删除事件。
struct event {
    int x, y1, y2, evtCode;
};

```

将所有矩形的竖直边转换为事件后, 将事件根据 X 坐标从小到大排序, 然后依次处理每个事件。使用线段树维护, 如果是添加事件就把 (y_1, y_2) 插入线段树, 删除事件就把 (y_1, y_2) 从线段树中删除。每次发现两个事件的 X 坐标不同, 就把子矩形的面积——当前线段树覆盖的总长度与两个事件 X 坐标差值的乘积——累加到总面积中。如果给定的 Y 坐标范围较大可以使用坐标离散化技巧以便于使用线段树。

```

//-----14.4.2.cpp-----
const int MAXN = 100010, ADD = 1, DELETE = -1;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

struct rectangle { int x1, y1, x2, y2; } rects[MAXN];

struct event {
    int x, y1, y2, evtCode;
    bool operator<(const event &e) const { return x < e.x; }
} evts[MAXN * 2];

struct node { int cnt, height; } st[4 * MAXN] = {};

int n, id[MAXN * 2];

void build(int p, int left, int right) {
    if (left != right) {
        int middle = (left + right) >> 1;
        build(LC(p), left, middle);
        build(RC(p), middle + 1, right);
    }
    st[p].cnt = st[p].height = 0;
}

void pushUp(int p, int left, int right) {
    st[p].height = st[LC(p)].height + st[RC(p)].height;
    if (st[p].cnt) st[p].height = id[right + 1] - id[left];
}

void update(int p, int left, int right, int ul, int ur, int value) {
    if (right < ul || left > ur) return;
    if (ul <= left && right <= ur) st[p].cnt += value;
    else {
        int middle = (left + right) >> 1;
        if (middle >= ul) update(LC(p), left, middle, ul, ur, value);
        if (middle + 1 <= ur) update(RC(p), middle + 1, right, ul, ur, value);
    }
}

```

```

    }
    pushUp(p, left, right);
}

long long getArea() {
    for (int i = 0; i < n; i++) id[i] = rects[i].y1, id[i + n] = rects[i].y2;
    sort(id, id + 2 * n);
    // 将矩形的竖直边转换为事件。
    for (int i = 0; i < n; i++) {
        evts[i].evtCode = ADD, evts[i].x = rects[i].x1;
        evts[i + n].evtCode = DELETE, evts[i + n].x = rects[i].x2;
        // 坐标离散化。
        int ty1 = lower_bound(id, id + 2 * n, rects[i].y1) - id;
        int ty2 = lower_bound(id, id + 2 * n, rects[i].y2) - id;
        evts[i].y1 = evts[i + n].y1 = ty1;
        evts[i].y2 = evts[i + n].y2 = ty2;
    }
    // 对事件按 X 坐标升序排列后逐个处理。
    sort(evts, evts + 2 * n);
    long long area = 0;
    for (int i = 0; i < 2 * n; i++) {
        if (i && evts[i].x > evts[i - 1].x)
            area += (long long) (evts[i].x - evts[i - 1].x) * st[0].height;
        // 使用“区间偏移”技巧进行更新。
        update(0, 0, 2 * n - 1, evts[i].y1, evts[i].y2 - 1, evts[i].evtCode);
    }
    return area;
}
//-----14.4.2.cpp-----//

```

在具体实现时还需注意以下的两个细节：

(1) 在求矩形并的面积时，由于矩形转换为事件后，添加和删除事件必定是对称出现的，所以线段树只需在最开始的时候整体初始化一次即可，在程序运行结束时，线段树会恢复原始状态。在其他应用中，可能需要根据矩形的数量 n 的变化多次初始化线段树。

(2) 在更新线段树的过程中，需要正确处理左边界和右边界相同的情形。由于线段树的叶子结点所表示的区间为点区间，即区间的左右边界是相同的，如果使用一一对应的方式获取得到的覆盖高度将始终为零，会导致不正确的结果。为了解决此问题，可以使用“区间偏移”技巧来处理：在更新时将右界的索引值减去 1。以上述实现代码为例，在更新时，如果原始需要更新区间 $[x, y]$ 则更新 $[x, y - 1]$ ，这样使用 $id[y + 1] - id[x]$ 就能够正确获取区间 $[x, y]$ 所对应的覆盖高度。

扩展练习：11983* Wired Advertisement^D。

14.4.3 矩形并的周长

矩形并的周长是指给定一组边与坐标轴平行的矩形，要求确定这些矩形合并而成的几何图像的周长。该问题与矩形并的面积类似，只不过在具体求解时不需要确定线段树所覆盖的长度，而只需确定所覆盖的线段的“分段数”。

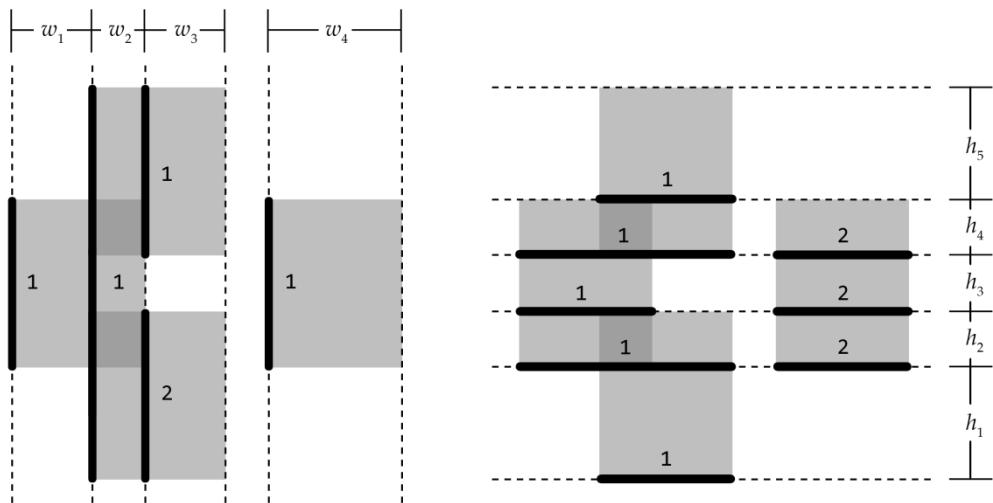


图 14-21 将矩形并的周长拆分为若干子矩形后再进行周长的求和。将周长按照两个方向分别进行求解，先求解水平方向上的周长，然后再使用相同的方法求解垂直方向上的周长，最后将两个方向上的周长相加即为合并后图形的总周长。图中左侧所示为求水平方向上的周长，由于相邻两条扫描线之间的距离 w_i 容易求出，只需知道矩形竖直边覆盖得到的“粗实线”的数量 c 即可，由 $2 \times c \times w_i$ 即可得到相邻两条垂直扫描线间所夹的图形的水平部分周长。同理，如图中右侧所示，可以计算相邻两条水平扫描线间所夹的图形的垂直部分周长。

按照与矩形并的面积类似的解法，可以得到求解矩形并的周长的实现代码。注意，需要将水平方向和垂直方向上的周长累加。

```

//-----14.4.3.1.cpp-----//
const int MAXN = 100010, ADD = 1, DELETE = -1;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

struct rectangle { int x1, y1, x2, y2; } rect[100010];

struct event {
    int x, y1, y2, evtCode;
    bool operator<(const event &e) const { return x < e.x; }
} evt[100010 * 2];

// 线段树的结点。
// cnt 记录该结点表示的区间被覆盖的次数。
// leftCovered 记录该结点表示的区间的左端点是否被覆盖。
// rightCovered 记录该结点表示的区间的右端点是否被覆盖。
// part 记录该结点表示的区间中包含的“分段数”。
struct node { int cnt, leftCovered, rightCovered, part; } st[4 * MAXN] = {};
int n, id[100010 * 2];

void build(int p, int left, int right) {
    if (left != right) {

```

```

        int middle = (left + right) >> 1;
        build(LC(p), left, middle);
        build(RC(p), middle + 1, right);
    }
    st[p].cnt = st[p].leftCovered = st[p].rightCovered = st[p].part = 0;
}

void pushUp(int p, int left, int right) {
    if (st[p].cnt) st[p].part = st[p].leftCovered = st[p].rightCovered = 1;
    else {
        if (left == right)
            st[p].part = st[p].leftCovered = st[p].rightCovered = 0;
        else {
            // 统计区间上线段的“分段数”。
            st[p].part = st[LC(p)].part + st[RC(p)].part;
            // 如果线段重合则总的线段计数需要相应减少。
            st[p].part -= st[LC(p)].rightCovered && st[RC(p)].leftCovered;
            st[p].leftCovered = st[LC(p)].leftCovered;
            st[p].rightCovered = st[RC(p)].rightCovered;
        }
    }
}

void update(int p, int left, int right, int ul, int ur, int value) {
    if (right < ul || left > ur) return;
    if (ul <= left && right <= ur) st[p].cnt += value;
    else {
        int middle = (left + right) >> 1;
        if (middle >= ul) update(LC(p), left, middle, ul, ur, value);
        if (middle + 1 <= ur) update(RC(p), middle + 1, right, ul, ur, value);
    }
    pushUp(p, left, right);
}

int getLength() {
    for (int i = 0; i < n; i++) id[i] = rect[i].y1, id[i + n] = rect[i].y2;
    sort(id, id + 2 * n);
    for (int i = 0; i < n; i++) {
        evts[i].evtCode = ADD, evts[i].x = rect[i].x1;
        evts[i + n].evtCode = DELETE, evts[i + n].x = rect[i].x2;
        // 坐标离散化。
        int ty1 = lower_bound(id, id + 2 * n, rect[i].y1) - id;
        int ty2 = lower_bound(id, id + 2 * n, rect[i].y2) - id;
        evts[i].y1 = evts[i + n].y1 = ty1;
        evts[i].y2 = evts[i + n].y2 = ty2;
    }
    sort(evts, evts + 2 * n);
    int length = 0;
    for (int i = 0; i < 2 * n; i++) {
        if (i && evts[i].x > evts[i - 1].x)
            length += 2 * st[0].part * (evts[i].x - evts[i - 1].x);
        // 使用“区间偏移”技巧进行更新。
        update(0, 0, 2 * n - 1, evts[i].y1, evts[i].y2 - 1, evts[i].evtCode);
    }
    return length;
}

```

```

int main(int argc, char *argv[]) {
    int cases = 0;
    while (cin >> n, n > 0) {
        cout << "Case " << ++cases << ":" ;
        for (int i = 0; i < n; i++)
            cin >> rects[i].x1 >> rects[i].y1 >> rects[i].x2 >> rects[i].y2;
        // 确定水平方向上的周长。
        int length = getLength();
        // 交换矩形的顶点，确定垂直方向上的周长。
        for (int i = 0; i < n; i++) {
            swap(rects[i].x1, rects[i].y1);
            swap(rects[i].x2, rects[i].y2);
        }
        length += getLength();
        cout << length << '\n';
    }
    return 0;
}
//-----14.4.3.1.cpp-----

```

为了提高效率，可以在线段树中额外维护一个值——垂直方向上被覆盖的线段的高度 h ，使得能够通过一次扫描就能获得矩形并的周长。

```

//-----14.4.3.2.cpp-----
const int MAXN = 100010, ADD = 1, DELETE = -1;

#define LC(x) (((x) << 1) | 1)
#define RC(x) (((x) + 1) << 1)

struct rectangle { int x1, y1, x2, y2; } rects[MAXN];
struct event {
    int x, y1, y2, evtCode;
    bool operator<(const event &e) const { return x < e.x; }
} evts[MAXN * 2];

// h 表示当前垂直方向覆盖线段的高度。
struct node { int cnt, leftCovered, rightCovered, part, h; } st[4 * MAXN] = {};
int n, id[MAXN * 2];

void build(int p, int left, int right) {
    if (left != right) {
        int middle = (left + right) >> 1;
        build(LC(p), left, middle);
        build(RC(p), middle + 1, right);
    }
    st[p].cnt = st[p].leftCovered = st[p].rightCovered = st[p].part = 0;
    st[p].h = 0;
}

void pushUp(int p, int left, int right) {
    if (st[p].cnt) {
        st[p].part = st[p].leftCovered = st[p].rightCovered = 1;
        st[p].h = id[right + 1] - id[left];
    } else {

```

```

    if (left == right) {
        st[p].part = st[p].leftCovered = st[p].rightCovered = 0;
        st[p].h = 0;
    } else {
        st[p].part = st[LC(p)].part + st[RC(p)].part;
        st[p].part -= (st[LC(p)].rightCovered & t[RC(p)].leftCovered);
        st[p].leftCovered = st[LC(p)].leftCovered;
        st[p].rightCovered = st[RC(p)].rightCovered;
        st[p].h = st[LC(p)].h + st[RC(p)].h;
    }
}
}

void update(int p, int left, int right, int ul, int ur, int value) {
    if (right < ul || left > ur) return;
    if (ul <= left && right <= ur) st[p].cnt += value;
    else {
        int middle = (left + right) >> 1;
        if (middle >= ul) update(LC(p), left, middle, ul, ur, value);
        if (middle + 1 <= ur) update(RC(p), middle + 1, right, ul, ur, value);
    }
    pushUp(p, left, right);
}

int getLength() {
    for (int i = 0; i < n; i++) id[i] = rects[i].y1, id[i + n] = rects[i].y2;
    sort(id, id + 2 * n);
    for (int i = 0; i < n; i++) {
        evts[i].evtCode = ADD, evts[i].x = rects[i].x1;
        evts[i + n].evtCode = DELETE, evts[i + n].x = rects[i].x2;
        int tyl = lower_bound(id, id + 2 * n, rects[i].y1) - id;
        int ty2 = lower_bound(id, id + 2 * n, rects[i].y2) - id;
        evts[i].y1 = evts[i + n].y1 = tyl;
        evts[i].y2 = evts[i + n].y2 = ty2;
    }
    sort(evts, evts + 2 * n);
    int length = 0, lastH = 0;
    for (int i = 0; i < 2 * n; i++) {
        if (i && evts[i].x > evts[i - 1].x)
            length += 2 * st[0].part * (evts[i].x - evts[i - 1].x);
        update(0, 0, 2 * n - 1, evts[i].y1, evts[i].y2 - 1, evts[i].evtCode);
        // 当前线段覆盖高度与前一次覆盖高度差的绝对值就是垂直方向上新增部分的长度。
        length += abs(st[0].h - lastH);
        lastH = st[0].h;
    }
    return length;
}

int main(int argc, char *argv[]) {
    int cases = 0;
    while (cin >> n, n > 0) {
        cout << "Case " << ++cases << ":" ;
        for (int i = 0; i < n; i++)
            cin >> rects[i].x1 >> rects[i].y1 >> rects[i].x2 >> rects[i].y2;
        int length = getLength();
        cout << length << '\n';
    }
}

```

```

    return 0;
}
//-----14.4.3.2.cpp-----//

```

14.5 凸包

点集 Q 的凸包 (convex hull) 是一个最小的凸多边形 P , 满足 Q 中的每个点或者在 P 的边界上, 或者在 P 的内部。显然, 当点集 Q 中点的数量小于 3 个时, 凸包无确切定义, 以下介绍的凸包算法在点的数量小于等于 3 个时, 认为其均在凸包上, 可根据题目具体应用加以适当修改。

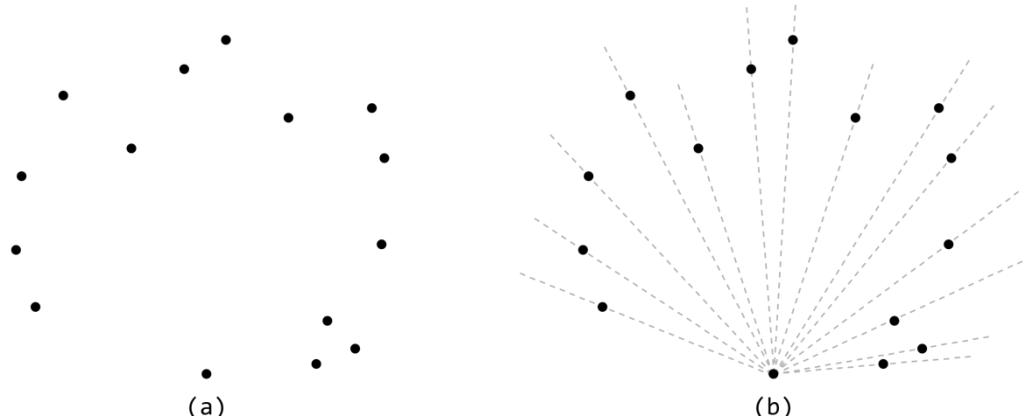
14.5.1 Graham 扫描法

Graham 扫描法 (Graham's scan algorithm) 通过设置一个关于候选点的栈 S 来解决凸包问题^[161]。输入集合 Q 中的每个点都被压入栈一次, 非凸包点最终被弹出栈。当算法终止时, 栈 S 中仅包含凸包的顶点。算法步骤如下:

(1) 去除输入点集 Q 中的重复点, 找到剩余点集 Q' 中的“最低点”, 即 y 坐标最小的点, 如果存在多个“最低点”, 则选取位于最左侧的“最低点”(即 x 坐标最小的点)作为参考点 p_r , 由于参考点是“最低且最左的点”, 必定是凸包的组成部分^I。

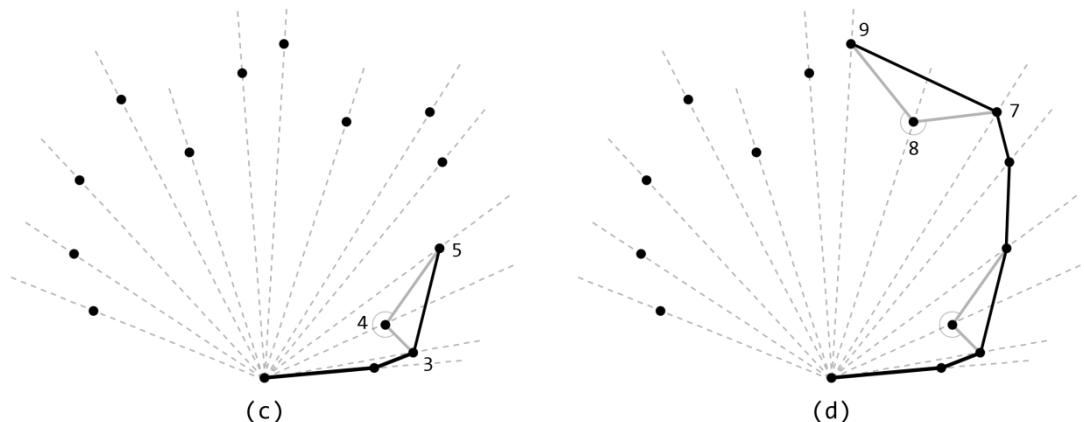
(2) 以参考点 p_r 为准, 按照极角大小对点集 Q' 中除参考点以外的其他点 p 进行排序。

(3) 按照到参考点 p_r 极角从小到大的顺序依次插入各个点。判定的依据是新插入的点与栈 S 上最后两个点形成的角度, 如果形成的角度大于 180 度, 那么栈上最后一个点必须被删除, 因为凸包的任意内角均小于 180 度, 重复此过程, 直到角度小于 180 度, 或者所有点处理完毕。为了避免进行实际的角度计算, 从而减少浮点运算导致的精度问题, 可以使用外积来判定角度是否大于 180 度。

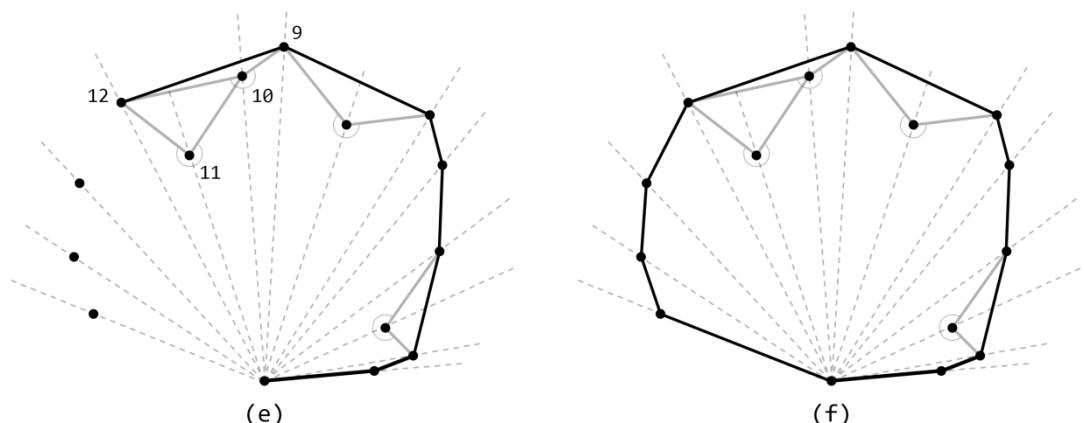


(a) 初始给定的点集, 共有 15 个点。(b) 选择位于最下侧的点 (如果有多个位于最下侧的点, 则选择这些点中位于最左侧的点) 作为参考点, 该点必定是凸包上的点, 将其他点相对于参考点按极角进行排序

^I 在原始版本的 Graham 扫描法中, 共有五个步骤, 第一个步骤所做的工作是选择一个位于凸包内的点作为参考点, Graham 在论文中选择点集中不共线的三个点计算其重心 (centroid) p_c , 然后以 p_c 作为参考点。此处选择了位于最低且最左的点作为参考点计算极角, 不影响算法的正确性且更为简便。



(c) 编号为 3、4、5 的点构成“右转”，故编号为 4 的点不是凸包上的点，需要将其从凸包上剔除。(d) 编号为 7、8、9 的点构成“右转”，故编号为 8 的点不是凸包上的点，需要将其从凸包上剔除



(e) 编号为 10、11、12 的点构成“右转”，故编号为 11 的点不是凸包上的点，需要将其从凸包上剔除，再次进行检查，可以发现编号为 9、10、12 的点构成“右转”，故编号为 10 的点不是凸包上的点，需要将其从凸包上剔除。(f) 最后的凸包

图 14-22 Graham 扫描法工作示例

Graham 扫描算法的时间复杂度为 $O(n \log n)$ ，最后得到的凸包顶点从栈底到栈顶按照逆时针顺序排列。

```
//-----14.5.1.cpp-----//
const double EPSILON = 1e-7;

// 定义表示点的结构体。
struct point {
    double x, y;
    bool operator<(const point &p) const {
        if (fabs(y - p.y) > EPSILON) return y < p.y;
        return x < p.x;
    }
}
```

```

bool operator==(const point &p) const {
    return fabs(x - p.x) <= EPSILON && fabs(y - p.y) <= EPSILON;
}
double distTo(const point &p) { return pow(x - p.x, 2) + pow(y - p.y, 2); }

// 定义多边形。
typedef vector<point> polygon;

// 参考点, 表示给定点中位于最下且最左的点。
point pr;

// 按相对于参考点的极角大小进行排序。当两个点共线时, 按距离参考点的距离大小排序,
// 如果两个点与参考点共线, 与参考点距离较小的排序在前, 这样能够保证在后续选择时,
// 总能够使用距离较大的点替换掉距离较小的点, 从而保证总是距离栈顶顶点最远的点构
// 成凸包而不是距离更近的点构成凸包。
bool cmpAngle(point &a, point &b) {
    if (collinear(pr, a, b)) return pr.distTo(a) <= pr.distTo(b);
    return ccw(pr, a, b);
}

// Graham 凸包扫描算法。
polygon grahamConvexHull(polygon &pg) {
    polygon ch(pg);

    // 按纵坐标和横坐标排序。
    sort(ch.begin(), ch.end());
    // 移除重复点。因为当三个点重复时, 转角的定义不明确, 为了避免这一问题, 事先去除重复点。
    // 使用库函数 unique 可以方便地予以实现。
    ch.erase(unique(ch.begin(), ch.end()), ch.end());

    // 顶点数量小于 3 个, 不构成凸包。
    if (ch.size() < 3) return ch;

    // 按极角排序, 最下且最左的点必定为凸包的一个顶点。
    pr = ch.front();
    // 按相对于参考点的极角大小对点进行排序。
    sort(ch.begin() + 1, ch.end(), cmpAngle);
    // 设置哨兵元素, 将最下且最左点设置为最后一个元素以便扫描时能回到参考点。
    // 漏掉此步骤, 将导致扫描无法回到起点, 从而使得求出的凸包不正确。
    ch.push_back(pr);

    // 根据转角必须小于 180 度的条件求凸包。当某个点与凸包的最后两点的角度构
    // 成一个右转或共线, 表明角度过大, 凸包上的最后一点必须被删除, 回退继续
    // 检查, 直到构成一个左转或者处理完毕。
    int top = 2, next = 2, total = ch.size() - 1;
    while (next <= total) {
        if (cw(ch[top - 2], ch[top - 1], ch[next])) top--;
        // 如果存在三点共线的情况且需要去除共线的凸包顶点则需要判断共线。
        // 将当前凸包的最末点替换为共线但离参考点距离最远的点。因为在预处理
        // 时已经将点按相对于参考点的极角和距离排序, 此时直接将距离较远的点
        // 替换距离较近的点即可。
        else {
            if (collinear(ch[top - 2], ch[top - 1], ch[next]))

```

```

        ch[top - 1] = ch[next++];
    else
        ch[top++] = ch[next++];
    }
}
// 去除非凸包顶点。注意：由于添加了哨兵元素，最后计算所得到的凸包，其起始点和结束点相同。
// 若要求凸包顶点不重复，则需将结尾的哨兵元素删除。
top--;
ch.erase(ch.begin() + top, ch.end());
return pg;
}
//-----14.5.1.cpp-----//

```

10135 Herding Frosh^D (新生集会)

某一天，许多大一新生聚集在校园中心位置的草坪上。为了美化校园环境，一位学长决定用粉色的丝带将所有新生围起来。你的任务是计算学长完成此项任务所需要的丝带长度。

学长将丝带的一端绑在公用电话柱上，然后绕着站满新生的区域走一圈，将丝带拉紧以便围住所有人，最后回到公用电话柱旁。学长所用丝带的长度尽可能地短，但又能够围住所有新生。除此之外，丝带两端各多出一米，以便将其绑在公用电话柱上。

你可以假定公用电话柱的坐标为(0, 0)，坐标的第一个分量为南北方向，第二个分量为东西方向。新生的坐标按照相对于公用电话柱的位置给出，单位为米。总共有不超过 100 名的新生。

输入

输入以一个正整数开始，表示测试数据的组数，之后一行为空行，每两组测试数据间也有一个空行。每组测试数据的第一行指定了一个整数，表示新生数量，接着每行用两个实数来表示一名新生的坐标。

输出

对于每组测试数据输出一行，包含一个实数，表示所用丝带的长度，单位为米，精确到小数点后两位。每两组测试数据的输出之间输出一个空行。

样例输入

```

1
4
1.0 1.0
-1.0 1.0
-1.0 -1.0
1.0 -1.0

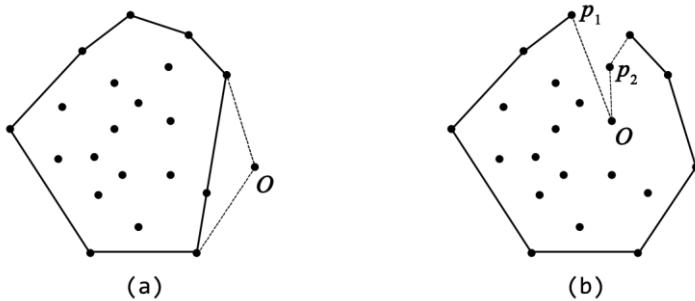
```

样例输出

```
10.83
```

分析

本题的关键在于如何处理公用电话柱。由于多出一个公用电话柱，简单的求凸包周长并不能得到正确的答案，需要分两种情况分别处理。

图 14-23 公用电话柱 O 的位置和凸包的关系

(a) 假设已经求出了所有新生点集所对应的凸包, 如果原点在此凸包外, 那么依照题意, 丝带必须绕过原点, 则只需将原点加入新生点集中求凸包, 凸包的周长加上多余需要的两米即为答案。

(b) 若原点在凸包内, 则丝带可以从任意两个按相对于原点极角排序的相邻点绕到电话柱子上, 此时需要枚举所有可能的长度, 然后取最小值即为结果。先将点按照相对于原点的极角排序, 然后任意选择两个相邻的点, 假设为 p_1 和 p_2 , 按照 Graham 凸包扫描算法的思想开始扫描, 不过这里需要将起始点设置为原点, 最后一点设置为 p_2 , 在扫描结束时再加上点 p_1 , 那么扫描所得到的“包”的形状恰恰就是题目所要求丝带的形状。需要注意, 在预处理时需要去除重复点以免干扰枚举过程。

强化练习: 218 Moth Eradiction^B, 1206* Boundary Points^D, 11096 Nails^C。

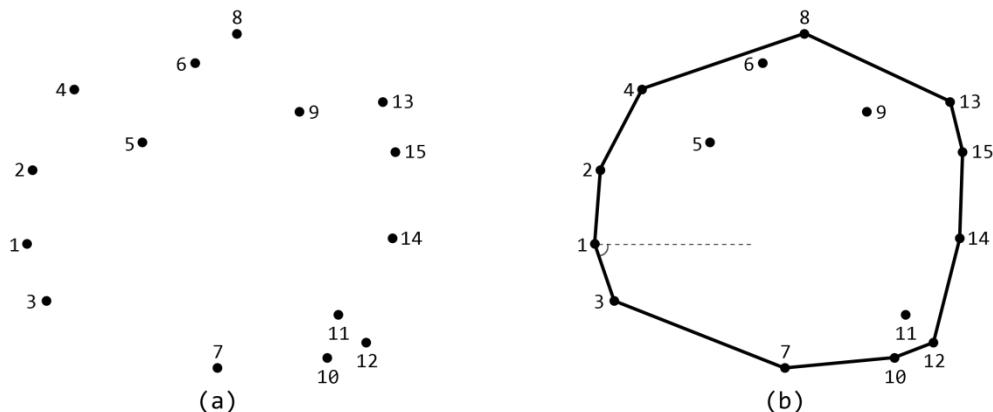
扩展练习: 811 The Fortified Forest^D。

14.5.2 Jarvis 步进法

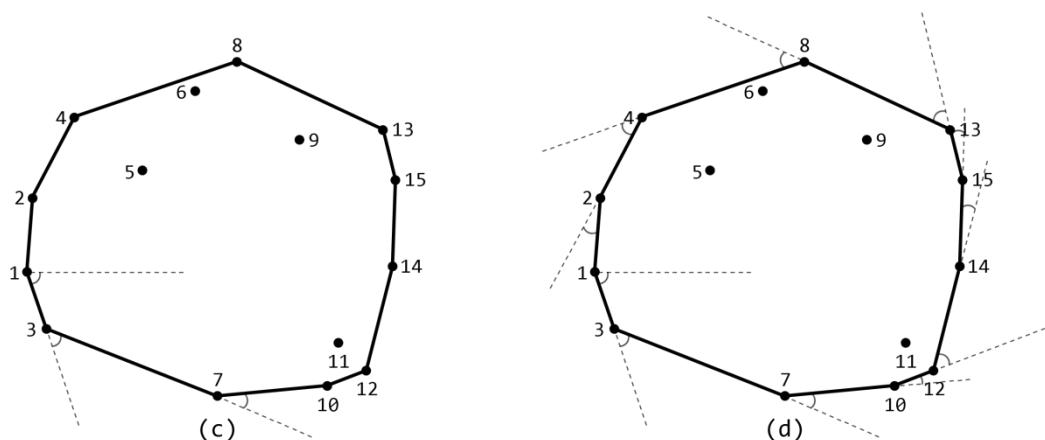
Jarvis 步进法 (Jarvis march algorithm) 使用类似于将礼物打包 (gift wrapping) 的方法来计算一个点集 Q 的凸包 $CH(Q)$ ^[162], 因此又称卷包裹法。算法首先选择位于最左且最下的点为起始凸包顶点, 采用类似于将礼物用彩纸包裹的过程, 逐个凸包顶点进行计算。算法在概念上与 Graham 扫描算法相似, 但是 Graham 扫描算法在每次顶点入栈时, 该顶点可能在后续计算过程中会被丢弃, 而 Jarvis 步进法每次迭代均会确定一个凸包顶点。算法的运行时间为 $O(nh)$, 其中 h 是 $CH(Q)$ 的顶点数, 本算法属于输出敏感算法 (output-sensitive algorithm), 即当结果凸包的顶点越少, 其计算速度越快。当 h 为 $o(\log n)$ 时, Jarvis 步进法在渐进意义上比 Graham 扫描法的速度更快。

Jarvis 步进法选出的第一个顶点是最左且最低的点 p_0 , 下一个顶点 p_1 与其他点相比, 有着相对于 p_0 的最小极角, 接着, p_2 有相对于 p_1 的最小极角, 依此类推。可以将算法概括为以下步骤:

- (1) 找到位于最左的点, 如果有多个最左点, 选择其中位于最下的点 $p_{leftLower}$, 根据凸包的性质, 该点肯定为凸包上的一个点。
- (2) 以点 $p_{leftLower}$ 为起始点, 比较其他点相对于此点的极角, 选取极角最小的一个点 $p_{minAngle}$, 该点为凸包上的点。
- (3) 以得到的凸包顶点 $p_{minAngle}$ 作为起始点重复步骤 (2)。
- (4) 当得到的凸包点重新回到 $p_{leftLower}$ 时算法结束。



(a) 初始给定的点集，共有 15 个点。将所有点按照 x 坐标升序排列，如果多个点具有相同的 x 坐标，则按 y 坐标升序排列，选择排序后的第一个点为参考点，该点必定是凸包上的点。(b) 以参考点向右的水平线为参考线，寻找与点 1 构成最小极角的点，可知点 3 是凸包的一个顶点



(c) 以点 3 为原点，从点 1 到点 3 的延长线为参考线，寻找与点 3 所成极角最小的点，可知点 7 为凸包上的点。同理，寻找与点 7 所成极角最小的点，可知点 10 为凸包上的点。(d) 重复上述过程，寻找与点 2 所成极角最小的点，可知点 1 为凸包上的点，此时已经回到初始的凸包顶点，算法结束。从算法工作示例可以看出，每进行一次迭代就会确定凸包的一个顶点

图 14-24 Jarvis 步进法工作示例

在选取极角最小的点时，使用转向判断谓词比使用三角函数计算实际的转角更为方便。如果需要输出共线的凸包点，则需要适当对后续凸包点的选择条件进行更改，在更新凸包顶点时选择离当前凸包顶点距离最小且不为零的点，同时记录已经选择的顶点，以免在后续过程中重复选择。使用 Jarvis 步进法得到的凸包顶点顺序为逆时针排列。

```
//-----14.5.2.cpp-----//
// 定义表示点的结构体。
struct point {
    double x, y;
```

```

bool operator<(const point &p) const {
    if (fabs(x - p.x) > EPSILON) return x < p.x;
    return y < p.y;
}
bool operator==(const point &p) const {
    return fabs(x - p.x) <= EPSILON && fabs(y - p.y) <= EPSILON;
}
double distTo(const point &p) { return pow(x - p.x, 2) + pow(y - p.y, 2); }
};

// 定义多边形。
typedef vector<point> polygon;

// Jarvis 步进法求凸包。
polygon jarvisConvexHull(polygon &pg) {
    polygon ch;

    // 排序, 得到位置处于最左且最下的点。当有多个 x 坐标最小的点时, 取 y 坐标最小的点。
    // 去除重复点后, 得到的点数组的第一个元素肯定为凸包上的顶点。
    sort(pg.begin(), pg.end());
    pg.erase(unique(pg.begin(), pg.end()), pg.end());

    // 顶点数量小于 3 个, 不构成凸包。
    if (pg.size() < 3) return ch;

    // 当需要输出共线的凸包顶点时, 需要额外使用数据结构记录已经使用的顶点, 否则可能会
    // 重复选择顶点。
    // vector<int> selected(pg.size(), 0);

    // last 为当前凸包上的最后一个顶点。
    int last = 0;
    do {
        // 将第一个点设为候选凸包顶点, 遍历点集获取下一个凸包顶点。
        int next = 0;
        for (int i = 1; i < pg.size(); i++) {
            // 测试序号为 last, next, i 的点是否构成右转或共线。
            // 当构成右转时, 说明点 i 比 next 相对于 last 有更小的极角, 应该将当前的
            // 待选凸包点更新为点 i。当共线时, 选择距离当前凸包点 last 更远的点。
            if (cw(pg[last], pg[next], pg[i]) ||
                (collinear(pg[last], pg[next], pg[i]) &&
                pg[last].distTo(pg[i]) > pg[last].distTo(pg[next])))
                next = i;

            // 需要输出共线的凸包顶点时的判定条件: 当前为非凸包顶点, 且构成右转
            // 或者共线但与当前凸包顶点有不为零的更小距离。
            //if (!selected[i] &&
            //    (cw(pg[last], pg[next], pg[i]) ||
            //    (collinear(pg[last], pg[next], pg[i]) &&
            //    pg[last].distTo(pg[i]) < pg[last].distTo(pg[next]))))
            //    next = i;
        }
        // 将得到的顶点加入凸包。
        ch.push_back(pg[next]);
        // 需要输出共线的凸包顶点时, 需要记录已经成为凸包顶点的点。
        // selected[next] = 1;
    }
}

```

```

    // 更新当前凸包顶点, 如果获取的凸包顶点回到起始凸包点则结束。
    last = next;
} while (last != 0);
return ch;
}
//-----14.5.2.cpp-----//

```

强化练习: 596 The Incredible Hull^D, 675 Convex Hull of the Polygon^D。

14.5.3 Andrew 合并法

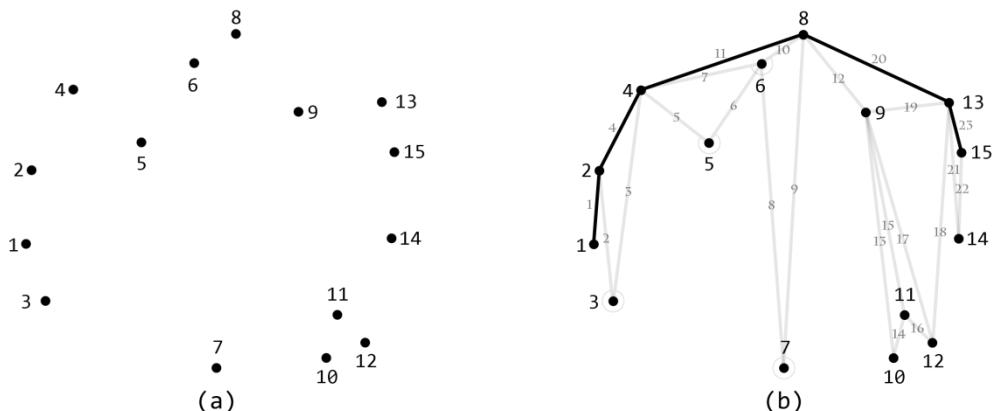
Andrew 合并法 (Andrew's monotone chain algorithm) 采用了类似于 Jarvis 步进法的打包技术来求解凸包^[163], 但该算法不需要按极角排序, 只需对点按 x 坐标和 y 坐标排序, 使用谓词 `ccwCollinear` 判断转向, 对存在三点共线的情况也能正确处理, 同时也不需要去除重复点, 实现起来比较简单。该算法分别求出上凸包和下凸包的顶点, 然后通过合并的方式得到整个凸包。其步骤为:

(1) 对点进行排序。按 x 坐标升序排列, 如果有多个点 x 坐标相同, 则按 y 坐标升序排列。

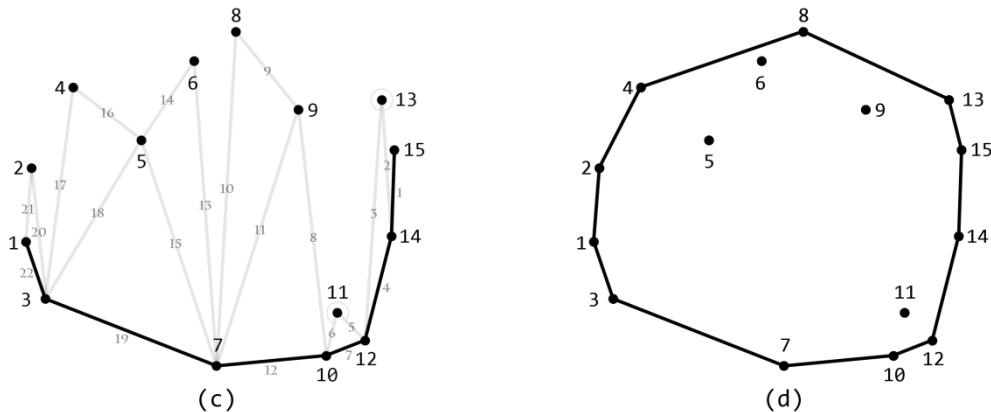
(2) 在排序好的点中, 第一个点肯定为凸包上的点, 因为它处于最下最左的位置, 以此点为基础, 建立扫描栈正向扫描, 当发现栈顶的 3 个点构成一个逆时针旋转时, 表明栈顶点位于更外侧, 需要退栈, 直到栈中元素不足 3 个或者栈顶 3 个点构成的是顺时针旋转; 若栈上的三个点共线, 由于已经将所有点按照坐标从左至右, 从下到上的顺序进行排序, 栈顶的点相较而言具有更远的距离, 因此同样需要退栈。依据此规则确定上凸包的顶点。

(3) 排序好的点中, 最末点肯定也为凸包上的点, 因为它处于最上最右的位置, 以此点为基础, 建立扫描栈反向扫描, 当发现栈顶的 3 个点构成一个逆时针旋转时, 表明栈顶点位于更外侧, 需要退栈, 直到栈中元素不足 3 个或者栈顶 3 个点构成的是顺时针旋转; 对于共线的情况处理方法与求上凸包的方法相同。依据此规则确定下凸包的顶点。

(4) 合并上下凸包的顶点。



(a) 初始给定的点集, 共有 15 个点。将所有点按照 x 坐标升序排列, 如果多个点具有相同的 x 坐标, 则按 y 坐标升序排列, 选择排序后的第一个点为参考点, 该点必定是凸包上的点。(b) 从参考点开始, 建立扫描栈, 从左向右正向扫描, 使用 `ccw` 判断谓词构建上凸包, 图中使用小号数字标注了线段的入栈顺序



(c) 以最右侧点为参考点, 由于该点必定是凸包上的顶点, 建立扫描栈, 从右向左逆向扫描, 使用 ccw 判断谓词构建下凸包, 图中使用小号数字标注了线段的入栈顺序。(d) 将上下两个“半凸包”合并即可得到所求点集的凸包

图 14-25 Andrew 合并法工作示例

Andrew 合并法的时间复杂度为 $O(n \log n)$ 。算法最终得到的凸包顶点是按照顺时针排列的, 如果需要逆时针输出凸包顶点需要适当进行调整¹。

```
//-----14.5.3.cpp-----//
const double EPSILON = 1e-7;

struct point {
    double x, y;
    bool operator<(const point &p) const {
        if (fabs(x - p.x) > EPSILON) return x < p.x;
        return y < p.y;
    }
};

// Andrew 凸包扫描算法。
polygon andrewConvexHull(polygon &pg) {
    polygon ch;
    // 排序。
    sort(pg.begin(), pg.end());
    // 求上凸包。
    for (int i = 0; i < pg.size(); i++) {
        while (ch.size() >= 2 &&
               ccwOrCollinear(ch[ch.size() - 2], ch[ch.size() - 1], pg[i]))
            ch.pop_back();
        ch.push_back(pg[i]);
    }
}
```

¹ 如果要求凸包顶点按逆时针顺序排列, 也可以先求下半凸包, 再求上半凸包, 最后予以合并, 此时需要使用转向判断谓词 cw 。

```

// 求下凸包。
for (int i = pg.size() - 1, upper = ch.size() + 1; i >= 0; i--) {
    while (ch.size() >= upper &&
           ccwOrCollinear(ch[ch.size() - 2], ch[ch.size() - 1], pg[i]))
        ch.pop_back();
    ch.push_back(pg[i]);
}
// 移除重复添加的起始凸包顶点。
ch.pop_back();
return ch;
}
//-----14.5.3.cpp-----//

```

如果需要保留共线的凸包顶点，在判断转向时使用逆时针转向判断谓词 `ccw`，并且在求下凸包时，从排序好的点数组中倒数第二个点开始构建。之所以这样，是因为在构建上半凸包的过程中，已经选择了最后一个点，在后续构建下半凸包的过程中，转向判断谓词 `ccwOrCollinear` 确保了不会再选择最后一个点作为下半凸包的起始顶点，而使用转向判断谓词 `ccw` 构建下半凸包时，会再次选择最后一个点，因此需要从倒数第二个点开始构建。使用此方法得到的凸包顶点，即使有多个顶点共线，也是严格按照顺时针方向排列的，这可以很好地满足某些题目的输出要求。

```

// 求下凸包，要求输出共线的凸包顶点。
for (int i = (int)pg.size() - 2, upper = ch.size() + 1; i >= 0; i--) {
    while (ch.size() >= upper && ccw(ch[ch.size() - 2], ch[ch.size() - 1], pg[i]))
        ch.pop_back();
    ch.push_back(pg[i]);
}

```

强化练习：[109 SCUD Busters^A](#)，[681 Convex Hull Finding^B](#)，[10652 Board Wrapping^C](#)，[11626 Convex Hull^C](#)。

扩展练习：[11072 Points^D](#)。

14.5.4 Melkman 算法

在求凸包时，如果点数据不是一次性全部给出，而是以每次一个点或多个点的方式给出，可以在每次加入新点时重新进行一次求凸包算法（例如使用 Graham 扫描算法），总的运行时间为 $O(n^2 \log n)$ ，如果使用在线凸包算法，可将运行时间缩短为 $O(n^2)$ 。若给定的点集具有特殊性质，则可以使用更为高效的算法。

Melkman 算法^[164]可在线性时间内计算依次给出的点形成的是一个简单多边形的点集的凸包，如果点集形成的是一个有自交的多边形，即非简单多边形，则使用此算法可能会得到错误的结果。由于该算法不需要一次性读入所有点，而是可以一次读入一个点，实时计算出当前的凸包，故可以作为在线凸包算法使用。

该算法使用一个双端队列，可以在队首（也称栈顶）进行压入或者弹出操作，或者在队尾（也称栈底）进行插入或移除操作。在算法中使用了判断谓词 `ccw`，即逆时针旋转。

假定简单多边形的各个顶点以逆时针方向给出，且任意相邻的三个顶点不共线，算法步骤如下：

(1) 从输入中取前面三个点 v_0, v_1, v_2 ，判断 $ccw(v_0, v_1, v_2)$ 是否为真，如果为真，则双端队列中的元素为 $D = \langle v_2, v_0, v_1, v_2 \rangle$ ，否则为 $D = \langle v_2, v_1, v_0, v_2 \rangle$ ，设置 $d_l = d_b = v_2$ ， $t = 2$ ， $b = 0$ 。

(2) 从输入中取下一个点 v_i ，检查 $ccw(d_{l-1}, d_l, v_i)$ 和 $ccw(d_b, d_{b+1}, v_i)$ 是否都为真，如果都为真，则点 v_i 必定在已有凸包内，否则违反简单多边形的限制。

(3) 若 v_i 不满足以上条件，则逐次按以下步骤恢复点集的凸性。队首逐次弹出 d_l ，直到 $ccw(d_{l-1}, d_l,$

v_i 为真, 然后压入 v_i , 队尾逐次移除 d_b , 直到 $\text{ccw}(v_i, d_b, d_{b+1})$ 为真, 然后插入 v_i 。

(4) 跳转到步骤 (2)。

```
-----14.5.4.cpp-----
// Melkman 凸包算法。
polygon melkmanConvexHull(polygon &pg) {
    // 点数小于 3 个, 不构成凸包。
    if (pg.size() < 3) return pg;
    // 使用数组实现双端队列。
    point deque[2 * pg.size() + 1];
    int bottom = pg.size(), top = bottom - 1;
    // 初始化双端队列。
    bool isLeft = ccw(pg[0], pg[1], pg[2]);
    deque[++top] = isLeft ? pg[0] : pg[1], deque[++top] = isLeft ? pg[1] : pg[0];
    deque[++top] = pg[2], deque[--bottom] = pg[2];
    // 检查给定顶点是否符合要求。
    int next = 3;
    while (next < pg.size()) {
        if (ccw(deque[top - 1], deque[top], pg[next]) &&
            ccw(deque[bottom], deque[bottom + 1], pg[next])) {
            next++;
            continue;
        }
        // 移除双端队列两侧不符合要求的顶点。
        while (!ccw(deque[top - 1], deque[top], pg[next])) top--;
        deque[++top] = pg[next];
        while (!ccw(pg[next], deque[bottom], deque[bottom + 1])) bottom++;
        deque[--bottom] = pg[next];
        next++;
    }
    polygon ch;
    for (int i = bottom; i < top; i++) ch.push_back(deque[i]);
    return ch;
}
-----14.5.4.cpp-----
```

14.6 公式及定理应用

14.6.1 Pick 定理

Pick 定理 (Pick's theorem) 描述了格点多边形 P 的面积与其边界/内部格点数之间的关系^[165]。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

$$A(P) = I(P) + \frac{B(P)}{2} - 1$$

如图 14-26 所示的多边形, 位于单位距离的格点矩阵上, 其 $I(P)=22$, $B(P)=11$, 故其面积为 $A(P)=22+11/2-1=26.5$ 个单位面积。

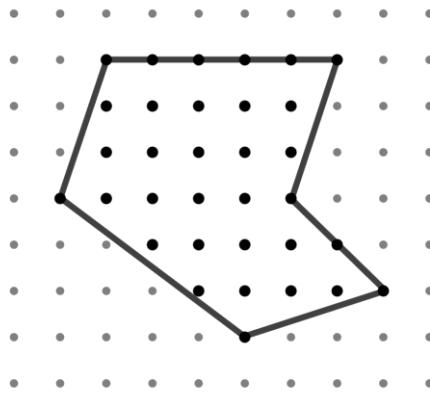


图 14-26 位于单位距离格点矩阵上的多边形

在应用 Pick 定理时, 注意其需要在等距离的格点多边形上使用, 且多边形的顶点均在格点上, 不能自交, 即多边形必须是简单多边形, 否则使用上述面积计算公式会得到错误的结果。格点多边形还有一个有用的性质: 设多边形某条边的起点坐标为 $p_1(x_1, y_1)$, 终点坐标为 $p_2(x_2, y_2)$, 则位于该边上 (不包括端点) 的格点数量为 $\gcd(\text{abs}(x_2 - x_1), \text{abs}(y_2 - y_1)) - 1$ 。

强化练习: [10088 Trees on My Island^B](#)。

扩展练习: [800* Crystal Clear^E](#)。

14.6.2 多边形面积

可以使用有向面积的概念来计算多边形的面积, 多边形的有向面积可以表示为

$$A(P) = \frac{1}{2} \sum_{i=0}^{n-1} (x_i y_{i+1} - x_{i+1} y_i)$$

注意顶点需要按照逆时针或顺时针给出, 否则无法使用该公式进行计算。当顶点不是按照逆时针顺序给出时, 计算得到的有向面积为负值, 需要取绝对值才能得到实际面积, **可以根据此特点判定给定的多边形的顶点顺序是逆时针顺序还是顺时针顺序**。

```
-----14.6.2.cpp-----
typedef vector<point> polygon;

// 利用有向面积公式计算多边形的面积。
double getArea(const polygon &pg) {
    double area = 0.0;
    int n = pg.size();
    // 顶点数量小于三个, 面积假定为 0。
    if (n < 3) return area;
    for (int i = 0, j = 1; i < n; i++, j = (i + 1) % n)
        area += (pg[i].x * pg[j].y - pg[j].x * pg[i].y);
    // 实际面积为有向面积的绝对值的一半。
    return fabs(area / 2.0);
}
-----14.6.2.cpp-----
```

强化练习: [10060 A Hole to Catch a Man^C](#), [10065 Useless Tile Packers^B](#), [11447 Reservoir Logs^D](#)。

扩展练习: 922 Rectangle by the Ocean^E, 10445 Make Polygon^D。

14.6.3 多边形重心

多边形的重心 (centroid) 计算和三角形的重心计算有关, 先介绍三角形重心的求解方法, 然后推广到凸多边形, 再推广到任意多边形。

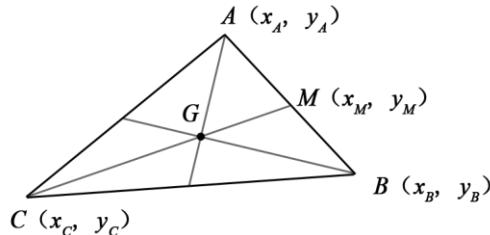


图 14-27 三角形的重心

如图 14-27 所示, 三角形的重心为三条中线的交点, 由于中线是边的平分线, 有

$$x_M = \frac{x_A + x_B}{2}$$

可以证明, 交点所处的位置为中线 CM 靠近顶点 C 的 $2/3$ 处 (中线 CM 靠近边 AB 的 $1/3$ 处), 因此有

$$x_G = x_C + \frac{2(x_M - x_C)}{3} = \frac{x_A + x_B + x_C}{3}$$

同理可得

$$y_G = \frac{x_A + x_B + x_C}{3}$$

对于凸多边形来说, 如果其具有 n 条边 ($n \geq 3$), 那么可将其剖分为 $n-2$ 个三角形, 其中每个三角形的重心为其中线的交点, 而凸多边形的重心坐标为所有三角形重心坐标与各自面积为权的加权算术平均数。

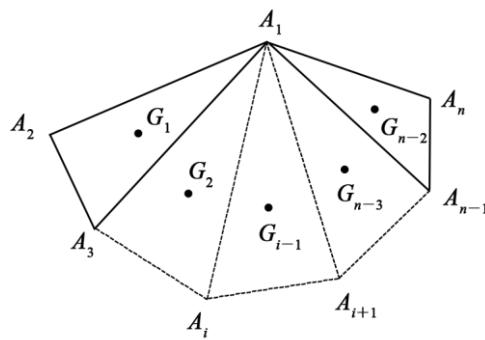


图 14-28 凸多边形重心

如图 14-28 所示, 凸多边形的重心坐标为

$$G_{CH(A)} = \frac{G_1 S_1 + G_2 S_2 + \cdots + G_{n-2} S_{n-2}}{S_1 + S_2 + \cdots + S_{n-2}}$$

注意, 此处的面积为各个剖分得到的三角形的有向面积, 可以使用前述介绍的求多边形的面积公式来得到三角形的有向面积。在求解过程中, 需要按序列举顶点坐标, 但不需考虑凸多边形的顶点是否以逆时针顺序给

出, 因为在最后求凸多边形的重心坐标时会进行除运算, 有向面积的符号会互相抵消。

```
//++++++14.6.3.cpp+++++++
struct point {
    double x, y;
    point (double x = 0, double y = 0): x(x), y(y) {}
    point operator-(point &p) { return point(x - p.x, y - p.y); }
};

double cross(const point &a, const point &b) {
    return a.x * b.y - a.y * b.x;
}

typedef vector<point> polygon;

point getCentroid(polygon &pg) {
    double areaOfPolygon = 0, areaOfTriangle = 0, px = 0, py = 0;
    for (int i = 2; i < pg.size(); i++) {
        areaOfTriangle = cross(pg[i - 1] - pg[0], pg[i] - pg[0]);
        areaOfPolygon += areaOfTriangle;
        px += areaOfTriangle * (pg[0].x + pg[i - 1].x + pg[i].x);
        py += areaOfTriangle * (pg[0].y + pg[i - 1].y + pg[i].y);
    }
    return point(px / (3.0 * areaOfPolygon), py / (3.0 * areaOfPolygon));
}
```

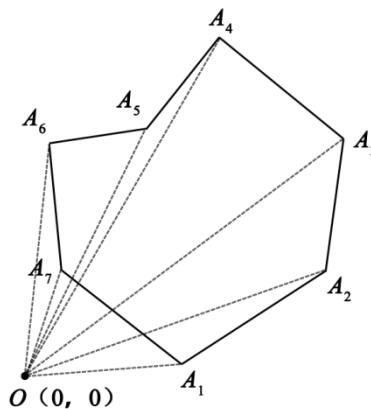


图 14—29 简单多边形的重心

对于任意简单多边形来说, 可以将 n 边多边形中每两个点 (有序, 逆时针顺序或者顺时针顺序) 加上原点共构成 n 个三角形, 将这些三角形看做质点 (质点的位置是三角形的重心 G_1, G_2, \dots, G_n , 质量是有向面积 S_1, S_2, \dots, S_n), 那么多边形可以看成就由这些质点组成, 质点坐标以其质量为权的加权算术平均数即是多边形重心坐标 G_{CH} 。计算方法与前述介绍的求凸多边形重心的方法类似。

```
point getCentroid(polygon &pg) {
    double areaOfPolygon = 0, areaOfTriangle = 0, px = 0, py = 0;
    int n = pg.size();
    for (int i = 0; i < n; i++) {
        areaOfTriangle = cross(pg[i], pg[(i + 1) % n]);
        areaOfPolygon += areaOfTriangle;
    }
}
```

强化练习：10002 Center of Masses^B。

10.6.4 三维几何体的表面积和体积

对于规则的三维几何体，例如圆柱、圆锥，可以利用既有的表面积和体积计算公式计算其表面积和体积，而对于表面不规则的三维几何体，可以尝试通过积分计算其表面积和体积。

球体的表面积和体积: $S = 4\pi r^2$, $V = 4\pi r^3/3$, r 为球体的半径。

圆柱体的表面积和体积: $S = 2\pi r^2 + 2\pi r h$, $V = \pi r^2 h$, r 为圆柱底面半径, h 为圆柱的高。

圆锥的表面积和体积: $S = \pi r^2 + \pi r L$, $V = \pi r^2 h / 3$, r 为圆锥底面半径, L 为圆锥侧面母线长度, h 为圆锥的高。

圆台的表面积和体积: $S = \pi(R^2 + r^2 + RL + rL)$, $V = \pi h(R^2 + Rr + r^2)/3$, L 为圆台的母线长度, R 为圆台下底半径, r 为圆台上底半径, h 为圆台的高。

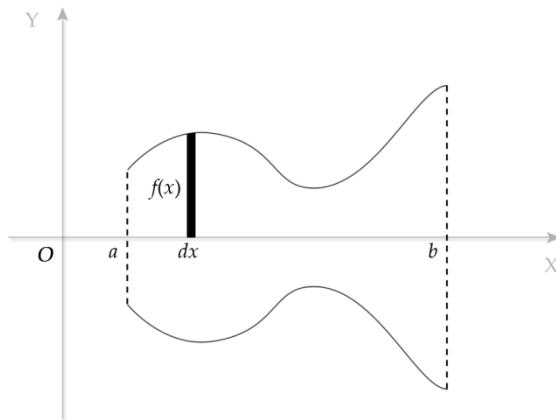


图 14-30 旋转体的体积元素

一般地, 如果旋转体是由连续曲线 $y=f(x)$, 直线 $x=a$, $x=b$ 及 x 轴所围成的曲边梯形绕 x 轴旋转一周而成的立体, 则可按下列步骤计算其体积: 取积分变量为 $x \in [a, b]$, 在 $[a, b]$ 上任取小区间 $[x, x+dx]$, 取以 dx 为高的窄边梯形绕 x 轴旋转而成的薄片的体积为体积元素

$$dV = \pi [f(x)]^2 dx$$

旋转体的体积为

$$V = \int_a^b \pi[f(x)]^2 dx$$

计算定积分一般通过牛顿 - 莱布尼茨公式(Newton-Leibniz formula)进行,即求得 $[f(x)]^2$ 的一个原函数 $F(x)$,则有

$$V = \int_a^b \pi[f(x)]^2 dx = \pi(F(b) - F(a))$$

强化练习: 10499 The Land of Justice^A, 11232 Cylinder^D, 12851 The Tinker's Puzzle^D。

扩展练习: 1280* Curvy Little Bottles^D, 1338 Crossing Prisms^E。

14.7 半平面交问题

14.7.1 凸多边形切分

半平面交问题 (half-plane intersection problem) 是指用一条直线将给定凸多边形切分成两个部分, 如何求被切分成的两部分的顶点坐标。值得注意的是, 凸多边形被直线剖分后仍为凸多边形。

这里介绍两种常用的方法。第一种方法比较直观, 算法步骤如下^[166]:

- (1) 求得直线与凸多边形各边的交点。
- (2) 根据需要, 如果需要确定直线逆时针方向的部分凸多边形, 则对原凸多边形的顶点进行扫描, 使用外积, 得到在直线逆时针方向的顶点, 否则得到顺时针区域的顶点。
- (3) 对得到的顶点、交点求凸包。

```
//-----14.7.1.1.cpp-----//
const double EPSILON = 1e-7;

struct point {
    double x, y;
    bool operator==(const point &p) const {
        return fabs(x - p.x) <= EPSILON && fabs(y - p.y) <= EPSILON;
    }
};

struct line {
    point a, b;
    bool contains(point p) { return pointInBox(p, a, b); }
};

typedef vector<point> polygon;

double cp(point a, point b, point c) {
    return (b.x - a.x) * (c.y - a.y) - (b.y - a.y) * (c.x - a.x);
}

bool cw(point a, point b, point c) {
    return cp(a, b, c) < -EPSILON;
}

bool collinear(point a, point b, point c) {
    return fabs(cp(a, b, c)) <= EPSILON;
}

bool parallel(line p, line q) {
    return fabs((p.a.x - p.b.x) * (q.a.y - q.b.y) -
               (q.a.x - q.b.x) * (p.a.y - p.b.y)) <= EPSILON;
}

point getIntersection(line p, line q) {
    point p = p.a;
```

```

double scale =
    ((p.a.x - q.a.x) * (q.a.y - q.b.y) - (p.a.y - q.a.y) * (q.a.x - q.b.x)) /
    ((p.a.x - p.b.x) * (q.a.y - q.b.y) - (p.a.y - p.b.y) * (q.a.x - q.b.x));
p.x += (p.b.x - p.a.x) * scale;
p.y += (p.b.y - p.a.y) * scale;
return p;
}

vector<polygon> halfPlaneIntersection(polygon pg, line cutline) {
    polygon cutted;
    for (int i = 0; i < pg.size(); i++) {
        point p1 = pg[i], p2 = pg[(i + 1) % pg.size()];
        cutted.push_back(p1);
        line edge = line{p1, p2};
        if (parallel(edge, cutline)) continue;
        if (!collinear(cutline.a, cutline.b, p1)) {
            point p3 = getIntersection(edge, cutline);
            if (edge.contains(p3)) cutted.push_back(p3);
        }
    }
    cutted.erase(unique(cutted.begin(), cutted.end()), cutted.end());
    if (cutted.size() > 0 && cutted.front() == cutted.back()) cutted.pop_back();

    polygon leftHalf, rightHalf;
    for (auto v : cutted.vertices) {
        if (collinear(cutline.a, cutline.b, v)) {
            leftHalf.push_back(v);
            rightHalf.push_back(v);
        }
        else {
            if (cw(cutline.a, cutline.b, v))
                rightHalf.push_back(v);
            else
                leftHalf.push_back(v);
        }
    }
}

vector<polygon> partitions;
if (leftHalf.size() >= 3) partitions.push_back(leftHalf);
if (rightHalf.size() >= 3) partitions.push_back(rightHalf);
return partitions;
}
//-----14.7.1.1.cpp-----//

```

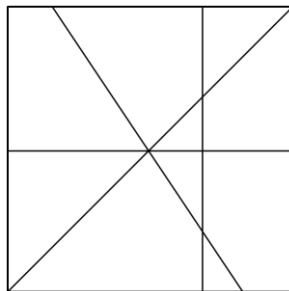
527 The Partition of a Cake^D (蛋糕切分)

给定一块大小为 1000×1000 单位长度的方形蛋糕，如果用刀对蛋糕进行切分，当经过若干次切分后，会分成多少块蛋糕呢？

假定：

1. 每次切分都不会超过 8 次；
2. 每次切分，每块蛋糕的边长均不会小于 1 个单位长度；
3. 表示蛋糕四个顶点的坐标分别为 $(0, 0)$, $(0, 1000)$, $(1000, 1000)$, $(1000, 0)$ ；
4. 切分线与蛋糕边缘的交点总是两个。

如下图所示的蛋糕切分，蛋糕块数总共有 10 块。



输入

输入的第一行为一个整数 M , 接着是一个空行, 然后是 M 组测试数据, 每两组测试数据间有一个空行。每组测试数据的第一行是一个整数, 表示切分的次数, 接着包含了对应次数的切分线信息。每条切分线由 4 个整数确定, 表示切分线与蛋糕边缘交点的坐标。

输出

对于每组测试数据输出一行, 表示经过切分后的蛋糕块数。每两组测试数据的输出间打印一个空行。

样例输入

```
1
3
0 0 1000 1000
500 0 500 1000
0 500 1000 500
```

样例输出

```
6
```

分析

初始给定的蛋糕形状是一个凸多边形, 之后每次切分可能会将已有的凸多边形剖分为更多的凸多边形。使用前述介绍的半平面交问题第一种求解方法即可。

参考代码

```
int main(int argc, char *argv[]) {
    int cases, cuts;
    double x1, y1, x2, y2;

    vector<point> square {
        point{0, 0}, point{1000, 0}, point{1000, 1000}, point{0, 1000}
    };

    cin >> cases;
    for (int c = 1; c <= cases; c++) {
        if (c > 1) cout << '\n';

        vector<polygon> current, next;
        current.push_back(polygon{square});

        cin >> cuts;
        for (int i = 1; i <= cuts; i++) {
            cin >> x1 >> y1 >> x2 >> y2;
            line cutline = line{point{x1, y1}, point{x2, y2}};
        }
    }
}
```

```

        for (auto pg : current) {
            vector<polygon> partitions = halfPlaneIntersection(pg, cutline);
            for (auto partition : partitions) next.push_back(partition);
        }
        current.swap(next);
        next.clear();
    }
    cout << current.size() << '\n';
}

return 0;
}

```

第二种方法为排序增量算法 (sort-and-incremental algorithm)，只需一次扫描即可将剖分后的凸多边形顶点求出^[167]。算法将凸多边形的每条边延长为直线，那么凸多边形可以看作是这些直线相交而成，其顶点即为相邻直线的交点。算法具体步骤为：

- (1) 将多边形的顶点逆时针排列，按逆时针得到凸多边形的各条边所在的直线。
- (2) 将所有直线使用极角和直线上两点的方式进行表示。
- (3) 将直线按极角进行排序，按下述规则去除极角相同的直线：当直线 a 与直线 b 极角相同时，若直线 a 位于直线 b 的逆时针方向，则保留直线 a ，去除直线 b (可以使用 `unique` 函数实现)。
- (4) 设立一个双端队列存放最终结果直线 (可使用数组实现，用两个整数指示凸包的起始直线和结束直线的序号)。
- (5) 对已按极角排序的直线进行扫描，对任意一条待扫描直线，检查双端队列栈顶两条直线的交点是否在待扫描直线的逆时针方向，若为否则栈顶退栈直到满足条件为止，继续检查，如果双端队列栈底两条直线的交点在待扫描直线的逆时针方向，则栈底退栈，直到满足条件为止，最后将待扫描直线压入双端队列的栈顶，成为凸包直线的一部分。
- (6) 所有直线扫描完毕，检查双端队列中栈顶的两条直线交点是否在栈底直线的逆时针方向，为否则栈顶退栈直到满足条件或队列为空，检查栈底的两条直线交点是否在栈顶直线的逆时针方向，为否则栈底退栈直到满足条件或队列为空。
- (7) 逐次求双端队列中相邻直线的交点即为最后凸多边形的顶点。

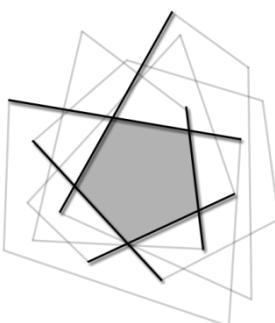


图 14-31 将多个凸多边形的边线转换为有向直线，按照极角进行排序，类似于切蛋糕的过程，这些凸多边形的交肯定是由若干边线构成，只有最靠近灰色“核心”区域的有向直线会被选择

需要注意的是：

- (1) 在对直线排序时，若两条直线极角相同，总是选择位于逆时针方向的那条直线。

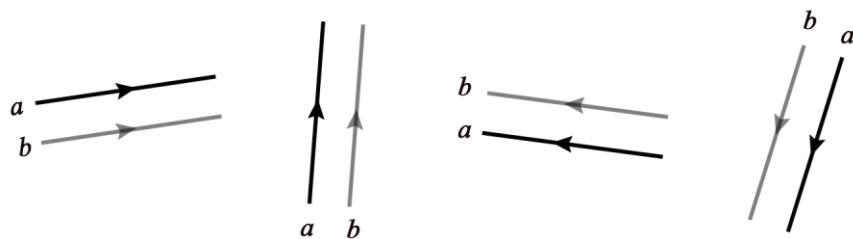


图 14-32 在排序过程中, 如果两条直线的极角相同, 总是选择位于逆时针方向的直线, 即选择直线 a。图中箭头表示直线极角的方向

(2) 使用排序增量算法时, 剖分直线的方向决定最后得到的凸多边形。

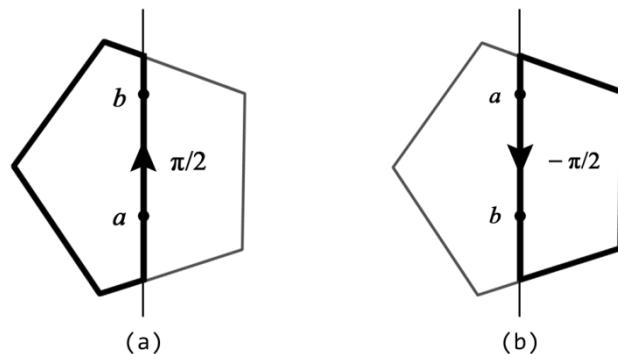


图 14-33 当选取的直线极角为 $\pi/2$ (直线方向为从 a 点指向 b 点), 最后得到的是左边粗线区域的凸多边形。反之, 若选取的直线极角为 $-\pi/2$ (直线方向为从 a 点指向 b 点), 则得到的是右侧粗线区域的凸多边形。因为在对直线按极角排序时, 若两条直线的极角相同, 总是会选择位于逆时针方向的直线, 这条直线更靠近“中心”

```
//-----14.7.1.2.cpp-----//
const int MAXV = 1100;
const double EPSILON = 1E-7;

// 点。
struct point {
    double x, y;
};

// 使用直线的极角和直线上的两个点来表示直线。
struct line {
    point a, b;
    double angle;
};

line pointToLine(point a, point b) {
    line lr;
    lr.a = a, lr.b = b, lr.angle = atan2(b.y - a.y, b.x - a.x);
    return lr;
}

// 多边形。
```

```

typedef vector<point> polygon;

// 将直线按照极角进行排序。若极角相同，选择位于极角逆时针方向的直线。
bool cmpLine(line p, line q) {
    if (fabs(p.angle - q.angle) <= EPSILON) return cw(p.a, p.b, q.a);
    return p.angle < q.angle;
}

// 比较两条直线的极角。
bool cmpAngle(line p, line q) { return fabs(p.angle - q.angle) <= EPSILON; }

// 给定一组直线，求直线的交点得到多边形的顶点。直线按逆时针顺序排列。
polygon halfPlaneIntersection(line *sides, int nLine) {
    polygon pg;
    line deq[MAXV];

    // 将所有有向直线按极角升序排列。
    sort(sides, sides + nLine, cmpLine);
    // 如果两条有向直线的极角相同，则选择沿着极角方向位于内侧的那条有向直线。
    nLine = unique(sides, sides + nLine, cmpAngle) - sides;

    // 将排序后的前两条有向直线压入双端队列。
    int btm = 0, top = 1;
    deq[0] = sides[0], deq[1] = sides[1];

    // 逐条有向直线进行筛选。
    for (int i = 2; i < nLine; i++) {
        // 如果发现栈顶或者栈底的两条有向直线平行，则不可能存在有效的凸多边形交。
        if (parallel(deq[top], deq[top - 1]) || parallel(deq[btm], deq[btm + 1]))
            return pg;
        // 如果位于栈顶的两条直线的交点位于 sides[i] 的右侧，说明栈顶直线不符合要求。
        while (btm < top &&
               cw(sides[i].a, sides[i].b, getIntersection(deq[top], deq[top - 1])))
            top--;
        // 如果位于栈底的两条直线的交点位于 sides[i] 的右侧，说明栈底直线不符合要求。
        while (btm < top &&
               cw(sides[i].a, sides[i].b, getIntersection(deq[btm], deq[btm + 1])))
            btm++;
        // 将 sides[i] 压入栈顶。
        deq[++top] = sides[i];
    }
    // 由于经过初步筛选后，剩余的有向直线可能并不能构成一个有效的凸多边形交，因此需要
    // 按照前述的判断逻辑进行检查，即首尾两端相互检查。
    while (btm < top &&
           cw(deq[btm].a, deq[btm].b, getIntersection(deq[top], deq[top - 1])))
        top--;
    while (btm < top &&
           cw(deq[top].a, deq[top].b, getIntersection(deq[btm], deq[btm + 1])))
        btm++;

    // 如果队列中剩余的有向直线不足三条，则无法构成有效的凸多边形交。
    if (top <= (btm + 1)) return pg;

    // 求相邻两条凸包边的交点获取顶点坐标。
    for (int i = btm; i < top; i++)
        pg.push_back(getIntersection(deq[i], deq[i + 1]));
}

```

```

// 首尾两条直线的交点也是顶点。
if (btm < (top + 1))
    pg.push_back(getIntersection(deq[btm], deq[top]));

// 返回凸多边形的交。
return pg;
}
//-----14.7.1.2.cpp-----//

```

需要注意，上述排序增量算法得到的凸包顶点，其顺序不确定（可能是逆时针排列，也可能是顺时针排列，还可能包含重复点），如果题目需要进一步使用得到的凸包顶点，需要进行适当处理。例如使用 Graham 算法对顶点求一次凸包，既能得到凸包顶点的逆时针排列，又能达到去除重复点的目的。

强化练习：137 Polygons^C，10084 Hotter Colder^D，11122 Tri Tri^D，[11265 The Sultan's Problem^D](#)。

扩展练习：10117* Nice Milk^D。

14.7.2 多边形内核

给定任意简单多边形，其内核（kernel）是指一个区域，该区域内的任意一点与所有顶点连接而成的线段均在该多边形内。任意简单多边形的内核要么不存在，要么是一个点或一条线段或一个凸多边形。如果多边形存在内核，对于多边形内的一点，如果该点不再内核中，则称其为“盲点”，从该点望去，多边形总会有部分无法通视。如果只是要求确定给定的多边形是否包含“盲点”，可以使用求多边形的凸包来解决，如果求得的凸包顶点数与原多边形顶点数相同，表明多边形为凸多边形，其内核就是多边形本身，不存在“盲点”区域，否则存在若干“盲点”区域。若需要求出内核，可以使用求半平面交的方法予以确定。

强化练习：[588 Video Surveillance^C](#)，[1304 Art Gallery^E](#)，[1571 How I Mathematician Wonder What You Are^E](#)，[10078* The Art Gallery^B](#)。

扩展练习：10907* Art Gallery^D。

14.8 最近点对问题

最近点对问题是指给定点集 Q ，求 Q 中最近的一对点之间的距离。如果使用朴素的穷尽搜索算法，需要检查任意一对点的距离，再从中选择距离最近的点，总共需要检查 $n(n-1)/2$ 个点对，运行时间为 $O(n^2)$ ，当点的数量较多时，此算法明显会超出时间限制。对于此问题，存在更为高效的分治算法，其算法的运行时间为 $O(n \log n)$ 。

分治算法

设有点集 Q ，其点的个数为 n ($n \geq 2$)，目标是寻找该点集中的最近点对。

算法的每一次调用的输入为点集 Q 的子集 P 、数组 X 、数组 Y ，数组 X 和 Y 均包含输入子集 P 的所有点。对数组 X 中的点按其 x 坐标单调递增的顺序进行排序，对数组 Y 中的点按其 y 坐标单调递增的顺序进行排序。

输入为 P 、 X 和 Y 的递归调用首先检查是否满足条件： $|P| \leq 3$ ，即子集 P 中点的个数是否小于等于 3 个，如果是，则对所有点对进行检查，返回最近点对的距离，如果点的数量大于 3，则递归调用如下分治模式。

分解

找出一条垂直线 L ，它把点集 P 划分为满足下列条件的两个集合 P_L 和 P_R ， $|P_L| = \lceil |P|/2 \rceil$ ， $|P_R| = \lfloor |P|/2 \rfloor$ ，

P_L 中的所有点在线 L 上或在 L 的左侧, P_R 中的所有点在线 L 上或在 L 的右侧。数组 X 被划分为两个数组 X_L 和 X_R , 分别包含 P_L 和 P_R 中的点, 并按 x 坐标单调递增的顺序进行排序。类似的, 数组 Y 被划分为两个数组 Y_L 和 Y_R , 分别包含 P_L 和 P_R 中的点, 并按 y 坐标单调递增的顺序进行排序。

解决

把 P 划分为 P_L 和 P_R 后, 再进行两次递归调用, 一次是找出 P_L 中的最近点对, 另一次找出 P_R 中的最近点对。第一次调用的输入为子集 P_L , 数组 X_L 和 Y_L ; 第二次调用的输入为子集 P_R , X_R 和 Y_R 。设对于子集 P_L 和 P_R , 返回的最近点对的距离分别为 d_L 和 d_R , 则当前得到的最近点对距离 $d = \min(d_L, d_R)$ 。

合并

最近点对要么是某次递归调用找出的距离为 d 的点对, 要么是 P_L 中的一个点与 P_R 中的一个点组成的点对, 算法接下来还需确定是否存在距离小于 d 的一个点对。事实上, 如果存在这样的一个点对, 则点对中的两个点必定都在距离直线 L 的 d 单位之内。也就是说, 它们必定都处于以直线 L 为中心、宽度为 $2d$ 的垂直带形的区域内。为了找出这样的点对, 算法需要做如下工作:

(1) 建立一个数组 Y' , 它是把数组 Y 中所有不在以直线 L 为中心宽度为 $2d$ 的垂直带形区域内的点去掉后所得到的数组。数组 Y' 与 Y 一样, 是按 y 坐标顺序排列的。

(2) 对数组 Y' 中每个点 p , 算法试图找出 Y' 中距离 p 在 d 单位以内的点。在 Y' 中仅需要考虑紧随 p 后的 7 个点。算法计算从 p 到这 7 个点的距离, 并记录 Y' 的所有点对中, 最近点对的距离 d' 。

(3) 如果 $d' < d$, 则垂直带形区域内, 的确包含比根据递归调用所找出的最近距离更近的点对, 于是返回该点对及其距离 d' 。否则, 就返回递归调用中发现的最近点对及其距离 d 。

正确性

第一, 当 $|P| \leq 3$ 时, 递归调用过程到底, 不会继续进行递归调用 (如果继续进行递归会造成无限循环进而引发错误)。第二, 仅需检查数组 Y' 中紧随每个点 p 后的 7 个点。为什么只需要检查 7 个点呢? 可以通过下图来说明其正确性。

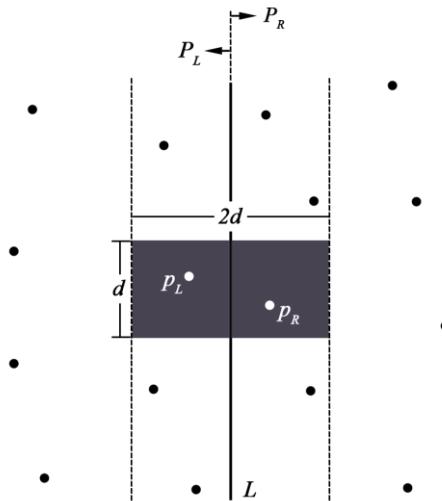


图 14-34 如果 $p_L \in P_L$, $p_R \in P_R$, 且 p_L 和 p_R 间的距离小于 d , 则它们必定位于一个以直线 L 为对称轴的 $d \times 2d$ 的矩形区域内

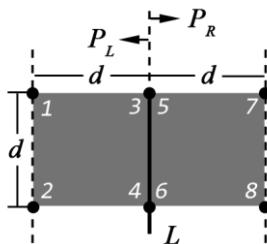


图 14-35 4 个两两距离至少为 d 单位的点是如何位于同一个 $d \times d$ 大小的正方形内的。左边的 4 个点 1、2、3、4 为 P_L 中的点, 右边的 4 个点 5、6、7、8 为 P_R 中的点, 其中点 3、5 互相重合, 点 4、6 互相重合, 且均位于切分直线 L 上。由图可知, 在 $d \times 2d$ 的矩形范围内至多有 8 个点, 它们相互之间的距离至少为 d 单位

下面说明 P 中至多有 8 个点可能处于该 $d \times 2d$ 矩形区域内。考察该矩形左半边的 $d \times d$ 正方形, 因为 P_L 中所有点之间的距离至少为 d 单位, 所以至多有 4 个点可能位于该正方形内, 图 14-35 说明了原因。类似的, P_R 中至多有 4 个点可能位于该矩形右半边的 $d \times d$ 正方形内。因此, P 中至多有 8 个点可能位于该 $d \times 2d$ 矩形内。(注意, 由于直线上的点可能属于 P_L , 也可能属于 P_R , 所以直线上最多可以有 4 个点。如果有两对重合的点, 每对包含一个 P_L 中的点和 P_R 中的点, 一对在直线 L 与矩形上面一条边的交点处, 另一对在直线 L 与矩形下面一条边的交点处, 就会达到上述限制。)

在说明了 P 中至多有 8 个点可能位于该矩形后, 就很容易看出仅需检查数组 Y' 中每个点之后的 7 个点。仍假设最近的点对为 p_L 和 p_R , 并假设在数组 Y' 中, p_L 位于 p_R 之前。那么, 即使 p_L 在 Y' 中尽可能早出现而 p_R 尽可能晚出现, p_R 也一定是跟随 p_L 的 7 个位置中的一个, 因此, 就证明了最近点对算法是正确的。

以下为具体实现, 为了尽量减少浮点数比较的误差, 计算两点间距离的函数 `getDistance` 的返回的是两点间欧几里得距离的平方, 最后计算得到的最近点对距离也是欧几里得距离的平方, 如果需要获取实际距离, 需要取其平方根。

```

//-----14.8.cpp-----
// 点的最大数量。
const int MAXV = 10010;

// “无限大” 距离值, 需要根据具体应用设置。
const double MAX_DIST = 1E20;

struct point { double x, y; };

double getDistance(const point &a, const point &b) {
    return (a.x - b.x) * (a.x - b.x) + (a.y - b.y) * (a.y - b.y);
}

// 记录点的坐标数据。
point dots[MAXV];

// 分治法求最近点对距离。
double closestDistance(int *P, int Pn, int *X, int Xn, int *Y, int Yn) {
    // 递归调用的出口, 当拆分后点数小于等于 3 个时, 使用穷举法计算最近距离。注意初始距离

```

```

// 应设为“无限大”，“无限大”的具体值应该根据具体应用设置。
if (Pn <= 3) {
    double dist = MAX_DIST;
    for (int i = 0; i < Pn - 1; i++)
        for (int j = i + 1; j < Pn; j++)
            dist = min(dist, getDistance(dots[P[i]], dots[P[j]]));
    return dist;
}

// 分解：把点集 P 划分为两个集合 P1 和 Pr。并得到相应的 Xl, Xr, Yl, Yr。
int P1[Pn], P1n, Pr[Pn], Prn;
int Xl[Pn], Xln, Xr[Pn], Xrn;
int Yl[Pn], Yln, Yr[Pn], Yrn;

// 标记某点是否在划分的集合 P1 中。初始时，所有点不在集合 P1 中。
// 当点的数量较大时，如果仍然使用数组来表示点是否在集合 P1 中，则可能使得数组的大小
// 超过运行内存限制。因此为了兼顾空间和查找效率，使用 unordered_map 来实现。
unordered_set<int> inP1;

// 将数组 P 划分为两个数量接近的集合 P1 和 Pr。P1 中的所有点在线 l 上或在 l 的左侧，
// Pr 中的所有点在线 l 上或在 l 的右侧。数组 X 被划分为两个数组 Xl 和 Xr，分别包含的
// P1 和 Pr 中点，并按 x 坐标单调递增的顺序排序。类似的，数组 Y 被划分为两个数组 Yl 和 Yr，
// 分别包含 P1 和 Pr 中的点，并按 y 坐标单调递增的顺序进行排序。对于 Xl, Xr, Yl, Yr,
// 由于参数 X 和 Y 均已排序，只需从中拆分出相应的点即可，并不需要再次排序，拆分后的数组
// 仍保持有序的性质不变，这是获得  $O(n \log n)$  运行时间的关键，否则若再次排序，运行时间
// 将为  $O(n(\log n)^2)$ 。
int middle = Pn / 2;
P1n = Xln = middle;
for (int i = 0; i < P1n; i++) {
    P1[i] = Xl[i] = X[i];
    inP1.insert(X[i]);
}

Prn = Xrn = (Pn - middle);
for (int i = 0; i < Prn; i++)
    Pr[i] = Xr[i] = X[i + middle];

// 根据某点所属集合，划分 Yl 和 Yr。
Yln = Yrn = 0;
for (int i = 0; i < Yn; i++) {
    if (inP1.find(Y[i]) != inP1.end())
        Yl[Yln++] = Y[i];
    else
        Yr[Yrn++] = Y[i];
}

// 解决：把 P 划分为 P1 和 Pr 后，再进行两次递归调用，一次找出 P1 中的最近点对，
// 另一次找出 Pr 中的最近点对。
double distanceL = closestDistance(P1, P1n, Xl, Xln, Yl, Yln);
double distanceR = closestDistance(Pr, Prn, Xr, Xrn, Yr, Yrn);

// 合并：最近点对要么是某次递归调用找出的距离为 minDist 的点对，要么是 P1 中的一个
// 点与 Pr 中的一个点组成的点对，算法确定是否存在其距离小于 minDist 的一个点对。
double minDist = min(distanceL, distanceR);

```

```

// 建立一个数组 Y', 它是把数组 Y 中所有不在宽度为 2*minDist 的垂直带形区域内去掉后
// 所得的数组。数组 Y' 与 Y 一样，是按 y 坐标顺序排序的。
int tmpY[Pn], tmpYn = 0;
for (int i = 0; i < Yn; i++)
    if (fabs(dots[Y[i]].x - dots[X[middle]].x) <= minDist)
        tmpY[tmpYn++] = Y[i];

// 对数组 Y' 中的每个点 p，算法试图找出 Y' 中距离 p 在 minDist 单位以内的点。仅需要考虑
// 在 Y' 中紧随 p 后的 7 个点。算法计算出从 p 到这 7 个点的距离，并记录下 Y' 的所有点对中，
// 最近点对的距离 tmpDist。
double tmpDist = MAX_DIST;
for (int i = 0; i < tmpYn; i++) {
    int top = ((i + 7) < tmpYn ? (i + 7) : (tmpYn - 1));
    for (int j = i + 1; j <= top; j++)
        tmpDist = min(tmpDist, getDistance(dots[tmpY[i]], dots[tmpY[j]]));
}

// 如果 tmpDist 小于 minDist，则垂直带形区域内包含这样的点对——此点对之间的距离比
// 根据递归调用所找出的最近距离的点对之间的距离更小。
return min(minDist, tmpDist);
}

// 排序点时所使用的比较函数。
bool cmpX(int a, int b) { return dots[a].x < dots[b].x; }
bool cmpY(int a, int b) { return dots[a].y < dots[b].y; }

// 参数 number 为点的个数。
double getClosestDistance(int number) {
    // 准备初始条件，注意，数组中保存的只是各个点的序号，而不是点的坐标。这样做，既可以
    // 减少数据复制的时间，提高效率，又不会对算法的正确性产生影响。
    int P[number], Pn, X[number], Xn, Y[number], Yn;
    // 初始化。
    Pn = Xn = Yn = number;
    for (int i = 0; i < number; i++)
        P[i] = X[i] = Y[i] = i;
    // 预排序，按 x 坐标和 y 坐标分别排序。
    sort(X, X + Xn, cmpX);
    sort(Y, Y + Yn, cmpY);
    // 调用分治算法。
    return closestDistance(P, Pn, X, Xn, Y, Yn);
}
//-----14.8.cpp-----//

```

除了使用上述介绍的分治法解决最近点对问题之外，还可以通过构建一种称为 Voronoi 图的几何结构在 $O(n \log n)$ 的时间内解决最近点对问题^[168]，不过在具体实现时该种方法编程复杂度较高，更为常见的是构建 kd 树来进行最近点对的查询。

强化练习：[10245 The Closest Pair Problem^A](#)，[11378 Bey Battle^D](#)。

扩展练习：[152 Tree's a Crowd^A](#)。

14.9 最远点对问题

最远点对问题和最近点对问题恰好相反，其所求为给定点集中具有最远距离的一对点。如果使用朴素的

穷尽搜索，时间复杂度为 $O(n^2)$ 。与最近点对问题不同，最远点对问题并不能直接运用求最近点对问题时的分治策略，而利用凸包的几何性质，可以在 $O(n \log n)$ 的时间内求解该问题。首先给出一个结论：可以证明，给定平面上的 n 个点，其中的最远点对必定位于这 n 个点所对应的凸包上^[169]。根据此结论，可以先求出点集所对应的凸包，然后枚举凸包上两点间的最大距离。一般来说，随机点集所对应凸包的顶点数远少于原有点集中的点数，因此相较于前述的朴素穷尽搜索，可以提高枚举的效率。除此之外，在生产实践中，还可以预先确定一组“极限点”——只有这些极限点才可能位于凸包上，将位于极限点内部的点筛除，然后再求凸包，这样可以减少参与求凸包运算的点数，从而进一步提高效率^[170]。例如，可以先确定点集中位于最左、最上、最右、最下的四个极限点，由此构建一个凸四边形，很明显，位于此凸四边形内部的点不可能是凸包顶点，因此可以将其筛除。

如果点集对应凸包的顶点数超过 10^5 量级，使用枚举的方法确定最远点对效率不高，此时需要使用更为高效的方法。由于凸包上最远点对间的距离正为凸包直径的定义，因此求平面上点集的最远距离问题可以转化为求点集凸包的直径问题。而求凸包的直径，存在有效的算法。以下介绍一种称为“旋转卡壳”(rotating calipers)¹的方法来计算凸包的直径^[171]。为了便于理解旋转卡壳法，首先介绍支撑线(supporting line)和对踵点(antipodal point)的概念。

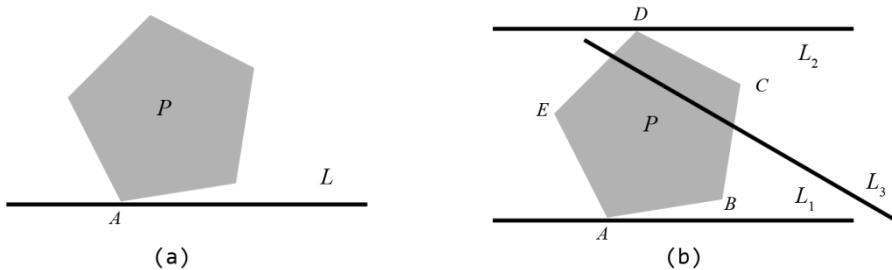
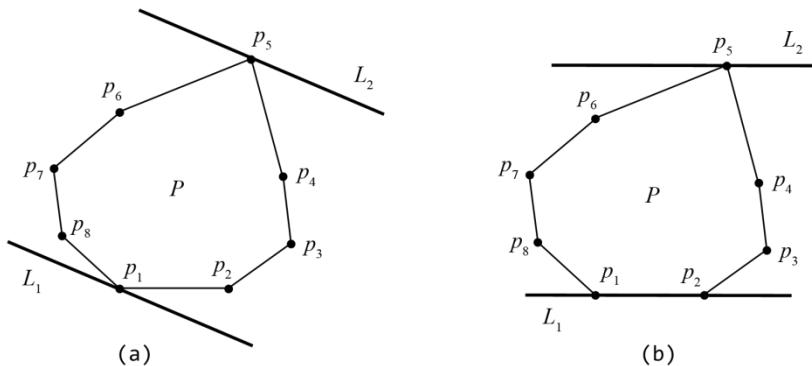


图 14-36 支撑线和对踵点

如图 14-36 (a) 所示，经过凸包 P 上顶点 A 的直线 L 即为凸多边形 P 的一条支撑线。直观上理解，多边形的支撑线是一条直线，该直线经过多边形上至少一个顶点，且多边形的所有顶点均在该直线的同一平面（或在直线上）。例如，图 14-36 (b) 中的直线 L_1 和 L_2 都是凸多边形 P 的支撑线，但 L_3 不是凸多边形 P 的支撑线。给定凸多边形的两条不同支撑线，如果它们互相平行，则称为平行支撑线。可以证明，凸多边形的直径是其平行支撑线间对踵点对的最远距离。什么是对踵点对呢？如果通过凸包上的两个点能够作一对（不重合的）平行支撑线，则这两个点就称为对踵点对。例如图 14-36 (b) 中的顶点对 (A, D) 和 (B, D) 都是对踵点对，但顶点对 (D, E) 不是对踵点对，因为经过 D 和 E 无法作出凸多边形 P 的（不重合的）两条平行的支撑线。

¹ 旋转卡 (qiǎ) 壳 (kē)。也有人建议称之为“旋转卡尺”更合适，一是英文单词“caliper”就有“测径尺”的含义，二是“卡壳”容易引起误读（“卡”和“壳”都是多音字），而“旋转卡尺”既能表达算法的内涵又不容易引起误解。

图 14-37 (a) 顶点 p_1 和 p_5 构成对踵点对; (b) 顶点 p_5 和边 p_1p_2 构成对踵“点一边”对

那么如何找出所有的对踵点对从而得到最远距离呢? 可以证明, 给定凸多边形 P , 若其顶点数为 n , 则对踵点对的数量不超过 $[3n/2]$ 对^I。可以通过一种巧妙的方法在线性时间内找出所有的对踵点对。如图 14-37 (a) 所示, 在得到凸包 P 后, 设 p_1 和 p_5 是凸包上最远的两个顶点, 那么必然可以分别过 p_1 和 p_5 构造一对平行支撑线, 则 p_1 和 p_5 构成对踵点对, 不过此种情形却不易处理。如图 14-37 (b) 所示, 通过旋转这对平行线, 可以让某一条支撑线和凸包上的一条边重合, 这样的情形更容易处理, 称之为对踵“点一边”对。注意到 p_5 是凸包上离 p_1 和 p_2 所在直线最远的点, 那么可以枚举凸包上的所有边, 对每一条边找出凸包上离该边最远的顶点, 计算这个顶点到该边两个端点的距离, 并记录最大的值。直观上这是一个 $O(n^2)$ 的算法, 但是注意到当逆时针枚举边的时候, 最远点的变化也是逆时针的, 这样就可以不必每次都从头计算最远点, 而可以紧接着上一次的最远点继续计算, 于是可以得到 $O(n)$ 的算法。那么如何紧接着上一次的计算结果继续计算呢? 这里需要应用一个结论, 即对于凸多边形的任意一条边来说, 按序 (顺时针或逆时针) 枚举顶点时, 顶点和该边两个端点的距离所构成的函数是一个单峰函数, 可以通过图 14-38 直观地观察出这一性质。

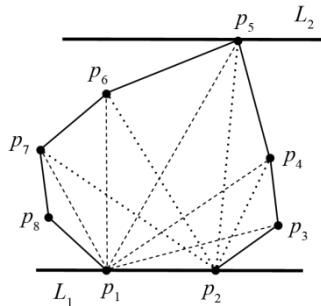


图 14-38 点和边之间的距离构成单峰函数

也就是说, 如果某个顶点距离指定的边越远, 则该顶点和此边的某个端点才会构成对踵点对。可以根据外积的几何意义来判断哪个点距离指定的边更远, 如图 14-38 所示, 对于凸包顶点 p_i 和 p_{i+1} 来说, 只需比较三角形 pp_ip_{i+1} 和 $p_{i+1}p_ip_{i+2}$ 谁的面积更大即可, 因为这两个三角形具有相同的底边, 具有更大面积的三角形

^I $[x]$ 表示对 x 向上取整, 例如 $[2.4]=3$ 。

自然具有更大的高度，高度更大表示顶点距离底边越远，根据前述，最远的顶点和底边构成对踵“点一边”对。比较面积可以转换为向量 p_1p_2 和 p_ip_{i+1} 的外积是否大于 0 来实现。

```
//++++++14.9.cpp+++++++
const double EPSILON = 1e-7;

struct point {
    double x, y;
    point(double x = 0, double y = 0): x(x), y(y) {}
    point operator+(point p) { return point(x + p.x, y + p.y); }
    point operator-(point p) { return point(x - p.x, y - p.y); }
    point operator*(double k) { return point(x * k, y * k); }
    point operator/(double k) { return point(x / k, y / k); }
    double distTo(point p) { return sqrt(pow(x - p.x, 2) + pow(y - p.y, 2)); }
};

typedef vector<point> polygon;
double cross(point a, point b) { return a.x * b.y - a.y * b.x; }

// 注意：凸包顶点需要按照逆时针方向排序。
double rotatingCalipers(polygon pg) {
    double dist = 0.0;
    pg.push_back(pg.front());
    for (int i = 0, j = 1, n = pg.size() - 1; i < n; i++) {
        while (cross(pg[i + 1] - pg[i], pg[j + 1] - pg[j]) > EPSILON)
            j = (j + 1) % n;
        dist = max(dist, max(pg[i].distTo(pg[j]), pg[i + 1].distTo(pg[j + 1])));
    }
    return dist;
}
```

旋转卡壳法除了计算凸包的直径外还可以解决很多其他问题，例如确定两个凸包的最远距离、两个凸包的最近距离、凸包的最小面积外接矩形、凸包的最小周长外接矩形、对凸包进行三角剖分等等^{[172][173]}。下面以求凸包的最小面积外接矩形为例，介绍旋转卡壳法的应用。

可以证明，对于凸多边形的最小面积外接矩形，该矩形必定有一条边与凸包的边重合^[174]。根据此结论，可以枚举凸包的每条边作为外接矩形的一条边，剩下的就是找出三条支撑线，一条与枚举的边平行，另外两条与枚举的边垂直，这四条直线的交点构成一个矩形，枚举所有可能的矩形，取面积最小的矩形即为结果^[175]。

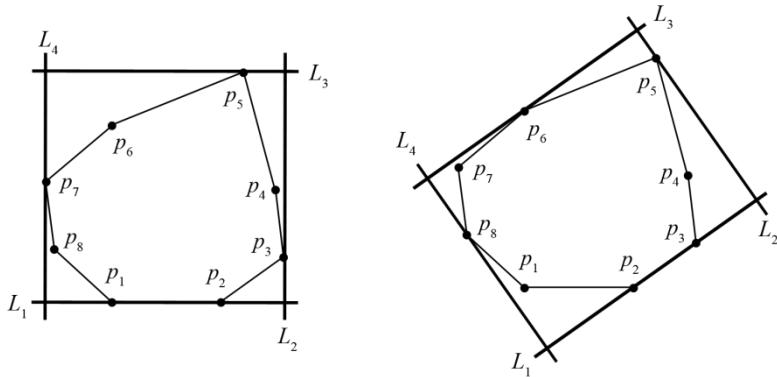


图 14-39 外接矩形的旋转

与确定点对间的最远距离类似，首先找出与直线 p_1p_2 构成对踵“点一边”对的顶点 p_i ，这个步骤可以通过比较各个顶点与底边 p_1p_2 构成的三角形 $p_ip_1p_2$ 的面积大小，找出能够构成最大面积的顶点来确定，而此三角形面积大小的计算可转化为向量 p_1p_2 和 p_ip_1 外积的计算，因为两者的外积在几何意义上是向量构成平行四边形的面积，恰为三角形 $p_ip_1p_2$ 面积的两倍。确定了三角形 $p_ip_1p_2$ 的面积，可以很容易得到矩形的高 H 为

$$H = \frac{2 * S_{p_ip_1p_2}}{|p_1p_2|} = \frac{p_1p_i \times p_1p_2}{|p_1p_2|}$$

那么如何计算外接矩形的宽度 W 呢？从图 14-40 (b) 可以直观地看出， p_r 是位于 p_1p_2 最右侧的凸包顶点， p_l 是位于 p_1p_2 最左侧的凸包顶点，可以通过内积的符号来判断凸包顶点是否继续远离 p_1p_2 。

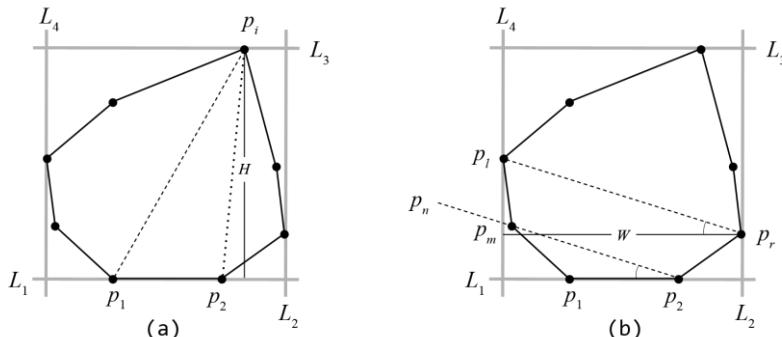


图 14-40 确定外接矩形的高度和宽度

如图 14-40 (b) 所示，将向量 p_ip_1 的起点平移至 p_2 ，有

$$W = |p_r p_m| = |p_r p_l| * \cos \angle p_l p_r p_m = |p_2 p_n| * \cos \angle p_l p_r p_m = |p_2 p_n| * \cos \angle p_n p_2 p_1$$

而

$$p_2 p_n \cdot p_2 p_1 = |p_2 p_n| * |p_2 p_1| * \cos \angle p_n p_2 p_1$$

故

$$|p_2 p_n| * \cos \angle p_n p_2 p_1 = \frac{p_2 p_n \cdot p_2 p_1}{|p_2 p_1|} = \frac{p_r p_l \cdot p_2 p_1}{|p_2 p_1|} = \frac{p_l p_r \cdot p_1 p_2}{|p_1 p_2|} = W$$

以下给出求凸包外接矩形的最小面积和最小周长的参考实现。

```
const double EPSILON = 1e-9;

struct point {
    double x, y;
    point(double x = 0, double y = 0): x(x), y(y) {}
    point operator+(point i) { return point(x + i.x, y + i.y); }
    point operator-(point i) { return point(x - i.x, y - i.y); }
    point operator*(double k) { return point(x * k, y * k); }
    point operator/(double k) { return point(x / k, y / k); }
    double distTo(point i) { return sqrt(pow(x - i.x, 2) + pow(y - i.y, 2)); }
};

typedef vector<point> polygon;

double cross(point a, point b) { return a.x * b.y - a.y * b.x; }
double dot(point a, point b) { return a.x * b.x + a.y * b.y; }
double norm(point a) { return dot(a, a); }
double abs(point a) { return sqrt(norm(a)); }
```

强化练习: 1453 Squares^E, 10173* Smallest Bounding Rectangle^D。

扩展练习: 12307* Smallest Enclosing Rectangle^E, 12311* All-Pair Farthest Points^E。

14.10 三维空间计算几何

三维空间计算几何相对于二维空间的计算几何更为复杂，在竞赛中一般较少出现。这里介绍最为常见的若干主题。

类似于二维计算几何，三维空间中的计算几何也需要一些基础的“元件”来达成目标，这些“元件”包括三维空间的点、线、面。为了定义这些“元件”，需要定义其依附的坐标系统。给定空间中任意三个有序的互不共面的向量 d_1, d_2, d_3 ，将其称为空间中的一组基。对于空间中的任意一个向量 m ，如果存在

$$m = x\mathbf{d}_1 + y\mathbf{d}_2 + z\mathbf{d}_3$$

则将三元有序实数组 (x, y, z) 称为 m 在基 d_1, d_2, d_3 中的坐标。给定空间中的一个点 O 和一组基 d_1, d_2, d_3 , 将这两者的组合称为空间的一个仿射坐标系, 记作 $[O; d_1, d_2, d_3]$, 其中 O 称为原点。进一步地, 如果 d_1, d_2, d_3 互相垂直且均为单位向量, 则将 $[O; d_1, d_2, d_3]$ 称为一个直角标架或者直角坐标系^[176]。给定一个向量, 可以使用直角坐标系中的一个点的坐标来表示向量的位置。

14.10.1 点

与二维的情形类似，也可以定义三维向量的内积、外积，借助三维向量来简化代码，同时能够增强代码的健壮性（robustness）。令向量 $u(x_1, y_1, z_1)$, $v(x_2, y_2, z_2)$ 为空间中的两个三维向量¹，定义 u 和 v 的外积为三维向量 r

¹ 严谨的写法是“令坐标为 $(x_u, y_u, z_u)^\top$ 的向量 u , 坐标为 $(x_v, y_v, z_v)^\top$ 的向量 v 为空间中的两个三维向量”。为了简便,本书使用了非正式的写法。

$$\mathbf{u} \times \mathbf{v} = \mathbf{r} = [y_1 z_2 - z_1 y_2, z_1 x_2 - x_1 z_2, x_1 y_2 - y_1 x_2]$$

\mathbf{r} 的方向和 \mathbf{u} , \mathbf{v} 都垂直, 方向由右手守则确定, 长度是以 \mathbf{u} 和 \mathbf{v} 为边组成的平行四边形的面积, 即

$$|\mathbf{r}| = |\mathbf{u} \times \mathbf{v}| = |\mathbf{u}| \cdot |\mathbf{v}| \cdot \sin \theta$$

其中 $|\mathbf{r}|$ 表示向量的模, 令 \mathbf{r} 为 (x_r, y_r, z_r) , 则有

$$|\mathbf{r}| = \sqrt{x_r^2 + y_r^2 + z_r^2}$$

定义 \mathbf{u} 和 \mathbf{v} 的内积为

$$\mathbf{u} \cdot \mathbf{v} = x_1 x_2 + y_1 y_2 + z_1 z_2 = |\mathbf{u}| \cdot |\mathbf{v}| \cdot \cos \theta$$

注意, 内积的结果是一个实数而不是一个向量。

如果使用坐标来表示向量的位置, 那么可以使用坐标来进行向量的和、差、外积、内积等运算。

```
//++++++14.10.1.cpp+++++++
const double EPSILON = 1e-7;
```

// 三维空间向量。

```
struct point3 {
    double x, y, z;
    point3 (double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {}
    point3 operator+(const point3 p) { return point3(x + p.x, y + p.y, z + p.z); }
    point3 operator-(const point3 p) { return point3(x - p.x, y - p.y, z - p.z); }
    point3 operator*(double k) { return point3(x * k, y * k, z * k); }
    point3 operator/(double k) { return point3(x / k, y / k, z / k); }
};
```

// 判断给定的值是否为零值。

```
bool zero(double x) { return fabs(x) < EPSILON; }
```

// 向量的模。

```
double norm(point3 p) {
    return sqrt(pow(p.x, 2) + pow(p.y, 2) + pow(p.z, 2));
```

// 三维向量的外积。

```
point3 cross(point3 a, point3 b) {
    point3 r;
    r.x = a.y * b.z - a.z * b.y;
    r.y = a.z * b.x - a.x * b.z;
    r.z = a.x * b.y - a.y * b.x;
    return r;
}
```

// 三维向量的内积。

```
double dot(point3 a, point3 b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}
```

混合积

给定向量 \mathbf{a} , \mathbf{b} , \mathbf{c} , 将 $\mathbf{a} \times \mathbf{b} \cdot \mathbf{c}$ 称为向量 \mathbf{a} , \mathbf{b} , \mathbf{c} 的混合积 (mixed product 或 triple product, 又称三重积)。

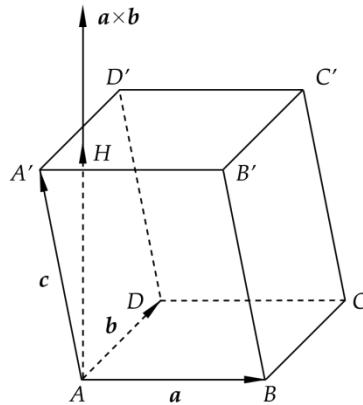


图 14-41 混合积的几何意义。设 $\overrightarrow{AB} = \mathbf{a}$, $\overrightarrow{AD} = \mathbf{b}$, $\overrightarrow{AA'} = \mathbf{c}$, 则平行六面体 $ABCD-A'B'C'D'$ 的底面积为 $|\mathbf{a} \times \mathbf{b}|$, 高为 $|\overrightarrow{AH}|$, 其中 $|\overrightarrow{AH}|$ 是 \mathbf{c} 在方向 $(\mathbf{a} \times \mathbf{b})^\circ$ 上的内射影, 因此

$$|\overrightarrow{AH}| = |\Pi_{(\mathbf{a} \times \mathbf{b})^\circ}(\mathbf{c})|$$

从而平行六面体 $ABCD-A'B'C'D'$ 的体积为

$$\begin{aligned} V &= |\mathbf{a} \times \mathbf{b}| |\Pi_{(\mathbf{a} \times \mathbf{b})^\circ}(\mathbf{c})| \\ &= |\mathbf{a} \times \mathbf{b}| |\Pi_{(\mathbf{a} \times \mathbf{b})^\circ}(\mathbf{c})| \\ &= |\mathbf{c} \cdot (\mathbf{a} \times \mathbf{b})| \\ &= |\mathbf{a} \times \mathbf{b} \cdot \mathbf{c}| \end{aligned}$$

如图 14-41 所示, 在几何上, 向量 \mathbf{a} , \mathbf{b} , \mathbf{c} 的混合积表示以 \mathbf{a} , \mathbf{b} , \mathbf{c} 为棱的平行六面体 $ABCD-A'B'C'D'$ 的有向体积, 其绝对值的六分之一就是向量 \mathbf{a} , \mathbf{b} , \mathbf{c} 所构成的四面体 $ABDA'$ 的体积。

```
// 向量 a, b, c 的混合积。
double mp(point3 a, point3 b, point3 c) {
    return dot(cross(a, b), c);
}
```

强化练习: 11012* Cosmic Cabbages^C。

14.10.2 直线

空间中的两个不重合点可以确定一条直线, 因此直线可以表示为

```
// 三维空间直线。
struct line3 {
    point3 a, b;
    line3 (point3 a = point3(), point3 b = point3()): a(a), b(b) {}
};

// 使用直线来定义线段。
typedef line3 segment3;
```

一个点和一个非零向量也可以确定一条直线, 因此可以得到另外一种常用的直线表示形式: 假设直线 l 经过点 p_0 , 其方向向量为 \mathbf{v} , 令直线 l 上的点为 p , 并设 p_0 和 p 的定位向量分别用 \mathbf{p}_0 , \mathbf{p} 表示, 则直线 l 的向量式参数方程为

$$\mathbf{p} = \mathbf{p}_0 + t\mathbf{v} \quad (14.1)$$

其中 t 称为参数, 它可以取任意实数。参数 t 的几何意义是: 点 p 在直线 l 上的仿射标架 $[p_0; v]$ 中的坐标。

如果将直线的向量式参数方程使用坐标写出, 可得

$$\begin{cases} x = x_0 + tX \\ y = y_0 + tY \\ z = z_0 + tZ \end{cases} \quad (14.2)$$

将 (14.2) 式称为直线的参数方程, 其中参数 t 可取任意实数。

将 (14.2) 式进行适当变换, 可得

$$\frac{x - x_0}{X} = \frac{y - y_0}{Y} = \frac{z - z_0}{Z} \quad (14.3)$$

将 (14.3) 式称为直线 L 的标准方程 (或点向式方程)。

如果已知直线 L 上两点 $p_1(x_1, y_1, z_1)$, $p_2(x_1, y_1, z_1)$, 则 $\overrightarrow{p_1p_2}$ 是直线 L 的一个方向向量, 从而可得 L 的方程为

$$\frac{x - x_1}{x_2 - x_1} = \frac{y - y_1}{y_2 - y_1} = \frac{z - z_1}{z_2 - z_1} \quad (14.4)$$

将 (14.4) 式称为直线 L 的两点式方程。

```
// 判断三点是否共线。
// 如果点 p1, p2, p3 共线, 则向量 p1p2 与向量 p2p3 外积的模为零。
bool collinear(point3 p1, point3 p2, point3 p3) {
    return norm(cross(p2 - p1, p3 - p2)) < EPSILON;
}

// 判断点是否在线段上, 包括在端点上。
bool pointOnSegmentInclude(point3 p, segment3 s) {
    return collinear(p, s.a, s.b) &&
        (s.a.x - p.x) * (s.b.x - p.x) < EPSILON &&
        (s.a.y - p.y) * (s.b.y - p.y) < EPSILON &&
        (s.a.z - p.z) * (s.b.z - p.z) < EPSILON;
}

// 判断点是否在线段上, 不包括在端点上。
bool pointOnSegmentExclude(point3 p, segment3 s) {
    return pointOnSegmentInclude(p, s) &&
        (!zero(p.x - s.a.x) || !zero(p.y - s.a.y) || !zero(p.z - s.a.z)) &&
        (!zero(p.x - s.b.x) || !zero(p.y - s.b.y) || !zero(p.z - s.b.z));
}

// 判断两点是否在线段的同侧。
// 点在线段上返回 false, 如果给定的两点和线段不共面则无意义。
bool sameSide(point3 p1, point3 p2, segment3 s) {
    return dot(cross(s.a - s.b, p1 - s.b), cross(s.a - s.b, p2 - s.b)) > EPSILON;
}

// 判断两点是否在线段的异侧。
// 点在线段上返回 false, 如果给定的两点和线段不共面则无意义。
bool oppositeSide(point3 p1, point3 p2, segment3 s) {
    return dot(cross(s.a - s.b, p1 - s.b), cross(s.a - s.b, p2 - s.b)) < -EPSILON;
}
```

两条直线的位置关系

空间的两条直线有以下三种位置关系: (1) 相交直线, 即两条直线有且仅有一个公共点。(2) 平行直线,

即两条直线在同一平面内且无公共点。(3) 异面直线, 即两条直线不同在任何一个平面且无公共点。

与空间直线相关的概念包括:(1)直线 a, b 是异面直线, 经过空间任意一点 O , 作直线 a', b' , 并使 $a' \parallel a$, $b' \parallel b$, 一般将直线 a' 和 b' 所成的锐角(或直角)称为异面直线 a 和 b 所成的角。(2)如果两条异面直线所成的角是直角, 我们就说这两条异面直线互相垂直。(3)与两条异面直线都垂直相交的直线, 称为这两条异面直线的公垂线。(4)两条异面直线的公垂线在这两条异面直线间的线段(公垂线段)的长度, 称为这两条异面的距离。

```
// 判断两条直线是否平行。
bool parallel(line3 u, line3 v) {
    return norm(cross(u.a - u.b, v.a - v.b)) < EPSILON;
}

// 判断两条直线是否垂直。
bool perpendicular(line3 u, line3 v) {
    return zero(dot(u.a - u.b, v.a - v.b));
}

// 计算直线到直线距离, 即共垂线段的长度。
double lineToLine(line3 u, line3 v) {
    point3 n = cross(u.a - u.b, v.a - v.b);
    return fabs(dot(u.a - v.a, n)) / norm(n);
}

// 计算两条直线夹角的余弦值。
double cos(line3 u, line3 v) {
    return dot(u.a - u.b, v.a - v.b) / norm(u.a - u.b) / norm(v.a - v.b);
}
```

点到直线的距离

求不在直线上的一点 p_3 到直线 p_1p_2 的距离, 在不必知道垂足的情况下, 可以利用面积法快速得到解。

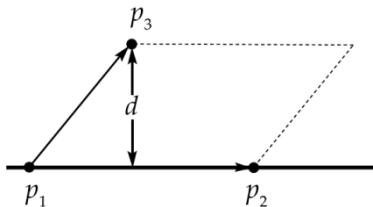


图 14-42 点到直线的距离

如图 14-42 所示, 点 p_3 到直线 p_1p_2 的距离 d 就是以向量 $\overrightarrow{p_1p_3}$ 和向量 $\overrightarrow{p_1p_2}$ 为邻边的平行四边形的底边 p_1p_2 上的高, 因此有

$$d = \frac{|\overrightarrow{p_1p_3} \times \overrightarrow{p_1p_2}|}{|\overrightarrow{p_1p_2}|}$$

```
// 利用面积法计算点到直线的距离。
double pointToLine(point3 p, line3 l) {
    return norm(cross(p - l.a, l.b - l.a)) / norm(l.b - l.a);
}
```

两条直线的交点

在立体几何中, 如果两条直线相交, 则一定在同一平面内且不平行。因此, 可以通过以下步骤来判断空间的两条直线是否相交: 已知两条直线的四个端点, 首先判断这四个端点是否在同一个平面内, 如果不在一个平面内, 则不相交; 再判断这个平面内的两条直线是否平行, 从而判断空间的两条直线是否相交。在确定两条直线相交后, 可以通过面积比的关系, 通过外积来求得交点的坐标^I。

```
// 利用面积比计算两条直线的交点。注意要事先判断两条直线是否共面和平行。
point3 intersection(line3 u, line3 v) {
    double k = cross(u.a - v.a, v.b - v.a).z / cross(u.b - u.a, v.b - v.a).z;
    return u.a + (u.b - u.a) * fabs(k);
}
```

14.10.3 平面

空间平面是指没有高低曲折的面。在相交的两直线上各取一动点, 并用直线连接起来, 所有这些直线就构成了一个平面。空间平面上任意两点的连线都完全落在此面上。任意非平行两平面的交线是一条直线。平面有多种表示方法, 第一种是普通方程

$$Ax + By + Cz + D = 0$$

这个平面的法向量为 $\mathbf{n}(A, B, C)$, 法向量是一个垂直于平面的向量。

第二种表示方法称为点法式方程, 即由平面上的一点 P_1 和平面法向量 \mathbf{n} 所确定。由于平面上任意一点 P 与 P_1 得到的向量都在此平面上, 因此和 \mathbf{n} 垂直, 即有

$$\mathbf{n} \cdot (\mathbf{P} - \mathbf{P}_1) = 0$$

亦即

$$\mathbf{n} \cdot (\mathbf{P} - \mathbf{P}_1) = \mathbf{n} \cdot \mathbf{P} - \mathbf{n} \cdot \mathbf{P}_1 = \mathbf{n} \cdot \mathbf{P} - d = 0 \Leftrightarrow \mathbf{n} \cdot \mathbf{P} = d$$

```
// 三维空间平面。
struct plane3 {
    point3 a, b, c;
    plane3 (point3 a = point3(0, 0, 0), point3 b = point3(0, 0, 0),
             point3 c = point3(0, 0, 0)): a(a), b(b), c(c) {}
};

// 平面的法向量。
point3 normalV(plane3 s) {
    return cross(s.a - s.b, s.b - s.c);
}
```

点和平面的关系

点和平面的关系只有两种: (1) 点在平面外; (2) 点在平面内。可通过内积和外积对点和平面的相互关系进行检测。

```
// 判断四点是否共面。利用内积判断平面 abc 的法向量与直线 ad 是否垂直。
bool coplanar(point3 a, point3 b, point3 c, point3 d) {
    return zero(dot(normalV(plane3(a, b, c)), d - a));
}
```

^I 参见本章“14.2.4 确定线段是否相交”一节的内容。

```

// 判断点是否在空间三角形上，包括在边界上，三点共线则无意义。
bool pointInPlaneInclude(point3 p, plane3 s) {
    return zero(norm(cross(s.a - s.b, s.a - s.c)) -
        norm(cross(p - s.a, p - s.b)) -
        norm(cross(p - s.b, p - s.c)) - norm(cross(p - s.c, p - s.a)));
}

// 判断点是否在空间三角形上，不包括在边界上，三点共线则无意义。
bool pointInPlaneExclude(point3 p, plane3 s) {
    return pointInPlaneInclude(p, s) &&
        norm(cross(p - s.a, p - s.b)) > EPSILON &&
        norm(cross(p - s.b, p - s.c)) > EPSILON &&
        norm(cross(p - s.c, p - s.a)) > EPSILON;
}

```

两条线段的相关位置

对于线段相交的判断，与二维线段相交的判断类似，也可以利用“线段跨越”进行检查。

```

// 判断两点是否在平面同侧，点在平面上返回 false。
bool sameSide(point3 p1, point3 p2, plane3 s) {
    return dot(normalV(s), p1 - s.a) * dot(normalV(s), p2 - s.a) > EPSILON;
}

// 判断两点是否在平面异侧，点在平面上返回 false。
bool oppositeSide(point3 p1, point3 p2, plane3 s) {
    return dot(normalV(s), p1 - s.a) * dot(normalV(s), p2 - s.a) < -EPSILON;
}

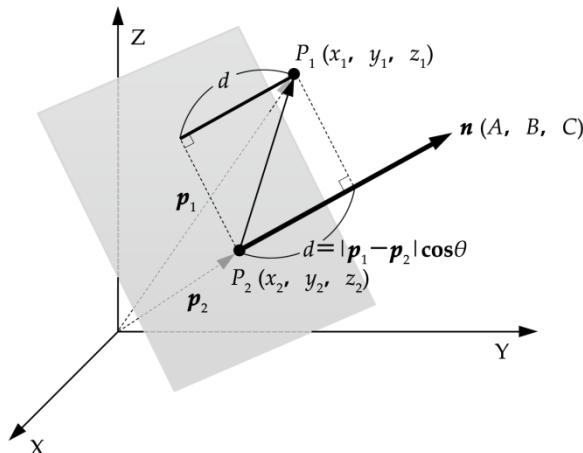
// 判断两条线段是否相交，包括端点相交和部分重合的情形。
bool intersectInclude(line3 u, line3 v) {
    if (!coplanar(u.a, u.b, v.a, v.b)) return false;
    if (!collinear(u.a, u.b, v.a) || !collinear(u.a, u.b, v.b))
        return !sameSide(u.a, u.b, v) && !sameSide(v.a, v.b, u);
    return pointOnSegmentInclude(u.a, v) || pointOnSegmentInclude(u.b, v) ||
        pointOnSegmentInclude(v.a, u) || pointOnSegmentInclude(v.b, u);
}

// 判断两条线段是否相交，不包括端点相交和部分重合的情形。
bool intersectExclude(line3 u, line3 v) {
    return coplanar(u.a, u.b, v.a, v.b) &&
        oppositeSide(u.a, u.b, v) &&
        oppositeSide(v.a, v.b, u);
}

```

点到平面的距离

点到平面的距离又称离差，可以使用下述方法予以求解。

图 14-43 求平面外任意一点 P_1 与平面的距离

如图 14-43 所示, 令点 $P_1=(x_1, y_1, z_1)$, 平面的法向量 $\mathbf{n}=(A, B, C)$ 以及平面上任意点 $P_2=(x_2, y_2, z_2)$, 只需要把线段 P_1P_2 投影到平面法向量 \mathbf{n} 上, 投影之后沿着法向量 \mathbf{n} 方向上的长度即为点到面的距离 d , 即¹

$$d = |\mathbf{p}_1 - \mathbf{p}_2| \cos \theta$$

根据内积的定义有

$$\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_2) = |\mathbf{n}| |\mathbf{p}_1 - \mathbf{p}_2| \cos \theta$$

故

$$d = |\mathbf{p}_1 - \mathbf{p}_2| \cos \theta = \frac{\mathbf{n} \cdot (\mathbf{p}_1 - \mathbf{p}_2)}{|\mathbf{n}|} = \frac{A(x_1 - x_2) + B(y_1 - y_2) + C(z_1 - z_2)}{\sqrt{A^2 + B^2 + C^2}}$$

如果给定的是平面方程 $Ax + By + Cz + D = 0$, 则有

$$d = \frac{Ax_1 + By_1 + Cz_1 + D}{\sqrt{A^2 + B^2 + C^2}}$$

注意, 计算得到的 d 是有向距离, 如果需要实际距离, 可以取其绝对值。有向距离 d 在某些时候非常有用: 如果 $d > 0$, 表明 P_1 位于平面的正面方向——法线指向的那一边; 如果 $d < 0$, 表明 P_1 位于平面的反面方向——法线相反方向的那一边; 若 $d = 0$, 表明 P_1 位于平面上。

```
// 计算点到平面的有向距离。
double pointToPlane(point3 p, plane3 s) {
    return dot(normalV(s), p - s.a) / norm(normalV(s));
}

// 计算点到平面的实际距离。
double realPointToPlane(point3 p, plane3 s) {
    return fabs(pointToPlane(p, s));
}
```

¹ 参考: <http://www.songho.ca/math/plane/plane.html>, 2020。

直线和平面的位置关系

给定一条直线 l 和一个平面 s , 直线 l 和平面 s 的位置关系有三种: (1) 直线 l 与平面 s 平行, 此时直线的方向向量 v 和平面的法向量 n 垂直; (2) 直线 l 与平面 s 相交; (3) 直线 l 在平面 s 内。

```
// 判断直线与平面是否平行。
bool parallel(line3 l, plane3 s) {
    return zero(dot(l.a - l.b, normalV(s)));
}

// 判断直线与平面是否垂直。
bool perpendicular(line3 l, plane3 s) {
    return norm(cross(l.a - l.b, normalV(s))) < EPSILON;
}

// 判断直线是否在平面内。
bool in(line3 l, plane3 s) {
    return coplanar(l.a, s.a, s.b, s.c) && coplanar(l.b, s.a, s.b, s.c);
}

// 计算直线与平面夹角的正弦值。
double sin(line3 l, plane3 s) {
    return dot(l.a - l.b, normalV(s)) / norm(l.a - l.b) / norm(normalV(s));
}
```

判断直线和平面是否相交可通过如下的方法: 若直线与平面不平行且直线不在平面内, 则直线和平面相交。类似的, 可以判断线段和平面是否相交。

```
// 判断线段与空间三角形是否相交, 包括交于边界和(部分)包含的情形。
bool intersectInclude(line3 l, plane3 s) {
    return !sameSide(l.a, l.b, s) &&
        !sameSide(s.a, s.b, plane3(l.a, l.b, s.c)) &&
        !sameSide(s.b, s.c, plane3(l.a, l.b, s.a)) &&
        !sameSide(s.c, s.a, plane3(l.a, l.b, s.b));
}

// 判断线段与空间三角形是否相交, 不包括交于边界和(部分)包含的情形。
bool intersectExclude(line3 l, plane3 s) {
    return oppositeSide(l.a, l.b, s) &&
        oppositeSide(s.a, s.b, plane3(l.a, l.b, s.c)) &&
        oppositeSide(s.b, s.c, plane3(l.a, l.b, s.a)) &&
        oppositeSide(s.c, s.a, plane3(l.a, l.b, s.b));
}

// 利用三棱锥的体积比, 计算直线与平面的交点。注意事先判断是否平行, 并保证三点不共线。
point3 intersection(line3 l, plane3 s) {
    point3 r = normalV(s);
    double t = dot(r, s.a - l.a) / dot(r, l.b - l.a);
    r.x = l.a.x + (l.b.x - l.a.x) * t;
    r.y = l.a.y + (l.b.y - l.a.y) * t;
    r.z = l.a.z + (l.b.z - l.a.z) * t;
    return r;
}
```

两个平面的交线

平面和平面的位置关系只有两种：平行或者相交。如果两平面不平行，则一定相交，所以能够通过判断两个平面是否平行从而得出它们是否相交。在两个平面保证相交的前提下，可以容易地求得两个平面的交线。

任意一条直线可以看出某两个相交平面的交线。设直线 l 是相交平面 P_1 和 P_2 的交线， P_i 的方程为

$$A_i x + B_i y + C_i z + D_i = 0, \quad i = 1, 2$$

它们的一次系数不成比例，则有

$$\begin{cases} A_1 x + B_1 y + C_1 z + D_1 = 0 \\ A_2 x + B_2 y + C_2 z + D_2 = 0 \end{cases} \quad (14.5)$$

将 (14.5) 式称为直线 l 的普通方程。

根据直线 l 的普通方程，可以得到直线的其他形式的方程。先找到直线 l 上的一个点 m_0 ，如果

$$\begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} \neq 0$$

则令 $z=0$ ，解 x, y 的一次方程组，可求得唯一的一组解 $x=x_0, y=y_0$ 。于是 $m_0(x_0, y_0, 0)$ 在 l 上，再确定直线 l 一个方向向量 $v(X, Y, Z)$ 即可确定其他形式的直线方程。由于 v 平行于两个给定的平面，故有

$$\begin{cases} A_1 X + B_1 Y + C_1 Z = 0 \\ A_2 X + B_2 Y + C_2 Z = 0 \end{cases}$$

令

$$X = \begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}, \quad Y = -\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \quad Z = \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix}$$

则

$$A_i X + B_i Y + C_i Z = \begin{vmatrix} A_i & B_i & C_i \\ A_1 & B_1 & C_1 \\ A_2 & B_2 & C_2 \end{vmatrix} = 0, \quad i = 1, 2$$

所以坐标为

$$\left(\begin{vmatrix} B_1 & C_1 \\ B_2 & C_2 \end{vmatrix}, -\begin{vmatrix} A_1 & C_1 \\ A_2 & C_2 \end{vmatrix}, \begin{vmatrix} A_1 & B_1 \\ A_2 & B_2 \end{vmatrix} \right) \quad (14.6)$$

的向量 v 与两个平面 P_i ($i=1, 2$) 平行，由于 (14.6) 式中的三个行列式不全为零，故 v 不为零向量，于是坐标为 (14.6) 的向量 v 就是 l 的一个方向向量，有了 l 上的一点 m_0 和它的一个方向向量 v ，就能够很容易地得到直线其他形式的方程。

如果使用点法式来表示平面，令两个平面的点法式方程为^[177]

$$\mathbf{n}_1 \cdot \mathbf{P} = d_1, \quad \mathbf{n}_2 \cdot \mathbf{P} = d_2$$

则交线 l 的方程可以使用向量式参数方程表示为

$$\mathbf{P} = \lambda \mathbf{n}_1 + \mu \mathbf{n}_2 + t(\mathbf{n}_1 \times \mathbf{n}_2)$$

之所以能够这样表示，是因为两个平面不平行时，它们的法向量也不平行，因此两个法向量所确定的平面和交线 l 必有交点（交线 l 同时垂直于两条法线，因此不可能和它们张成的平面平行，故而必有交点）。由于交线 l 的方向和两条法线都垂直，故交线 l 的方向向量为 $(\mathbf{n}_1 \times \mathbf{n}_2)$ 。为了确定交线 l 的向量式参数方程，还需确定交线 l 与两个法向量所确定平面的交点。不妨令交点为 \mathbf{P}_0 ，因为交点 \mathbf{P}_0 在向量 \mathbf{n}_1 和 \mathbf{n}_2 所确定的平面内，可以令交点 $\mathbf{P}_0 = \lambda \mathbf{n}_1 + \mu \mathbf{n}_2$ ^I。把 \mathbf{P} 的表达式代入两平面方程，可得^I

^I 可以证明：若向量 $\mathbf{a}, \mathbf{b}, \mathbf{c}$ 共面，并且 \mathbf{a} 与 \mathbf{b} 不共线，则存在唯一的一对实数 λ, μ ，使得 $\mathbf{c} = \lambda \mathbf{a} + \mu \mathbf{b}$ 。

$$\begin{aligned}\mathbf{n}_1 \cdot \mathbf{P} &= \lambda \mathbf{n}_1 \cdot \mathbf{n}_1 + \mu \mathbf{n}_1 \cdot \mathbf{n}_2 = d_1 \\ \mathbf{n}_2 \cdot \mathbf{P} &= \lambda \mathbf{n}_1 \cdot \mathbf{n}_2 + \mu \mathbf{n}_2 \cdot \mathbf{n}_2 = d_2\end{aligned}$$

联立解得

$$\lambda = \frac{d_1 \mathbf{n}_2 \cdot \mathbf{n}_2 - d_2 \mathbf{n}_1 \cdot \mathbf{n}_2}{\wedge}, \quad \mu = \frac{d_2 \mathbf{n}_1 \cdot \mathbf{n}_1 - d_2 \mathbf{n}_1 \cdot \mathbf{n}_2}{\wedge}$$

其中

$$\Delta = (\mathbf{n}_1 \cdot \mathbf{n}_1)(\mathbf{n}_2 \cdot \mathbf{n}_2) - (\mathbf{n}_1 \cdot \mathbf{n}_2)^2$$

// 判断两个平面是否平行。

// 如果平面 u 和 v 的法向量的外积的模为零, 表明两个平面的法向量平行, 因此两个平面也是平行的。

```
bool parallel(plane3 u, plane3 v) {
    return norm(cross(normalV(u), normalV(v))) < EPSILON;
}
```

// 判断两个平面是否垂直。

// 如果平面 u 和 v 的法向量的内积为零, 表明两个平面的法向量垂直, 因此两个平面也是垂直的。

```
bool perpendicular(plane3 u, plane3 v) {
    return zero(dot(normalV(u), normalV(v)));
}
```

// 计算两个平面的交线。

```

line3 intersection(plane3 u, plane3 v) {
    line3 r;
    r.a = parallel(line3(v.a, v.b), u) ?
        intersection(line3(v.b, v.c), u) : intersection(line3(v.a, v.b), u);
    r.b = parallel(line3(v.c, v.a), u) ?
        intersection(line3(v.b, v.c), u) : intersection(line3(v.c, v.a), u);
    return r;
}

```

两个平面的夹角

两个相交平面的夹角是指两个平面交成四个二面角中的任意一个。易知，其中两个等于两个平面的法向量 n_1, n_2 的夹角 $\langle n_1, n_2 \rangle$ ，另外两个等于 $\langle n_1, n_2 \rangle$ 的补角。两个平行（或重合）平面的夹角的夹角规定为它们的法向量 n_1, n_2 的夹角或其补角，从而等于 0 或 π 。设在直角坐标系中，两个平面的方程是

$$A_i x + B_i y + C_i z + D_i = 0, \quad i = 1, 2$$

则两个平面的一个夹角 θ 满足

$$\cos \theta = \frac{\mathbf{n}_1 \cdot \mathbf{n}_2}{|\mathbf{n}_1||\mathbf{n}_2|} = \frac{A_1A_2 + B_1B_2 + C_1C_2}{\sqrt{A_1^2 + B_1^2 + C_1^2} \cdot \sqrt{A_2^2 + B_2^2 + C_2^2}}$$

从上式可知, 两个平面垂直的充分必要条件是

$$A_1 A_2 + B_1 B_2 + C_1 C_2 \equiv 0$$

// 计算两个平面夹角的余弦值

```
//计算两个平面法向量的余弦值。  
double cos(plane3 u, plane3 v) {  
    return dot(normalV(u), normalV(v)) / norm(normalV(u)) / norm(normalV(v));  
}  
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++14.10.1.cpp++++++++++++++++++++++++++++++++++
```

¹ 由内积和外积的定义, η_1 与 $t(\eta_1 \times \eta_2)$ 的内积为零, η_2 与 $t(\eta_1 \times \eta_2)$ 的内积亦为零。

强化练习：578 Polygon Puzzler^D。

扩展练习: 503 Parallelepiped Walk^D, 10184* Equidistance^D。

14.10.2 三维凸包

计算三维凸包有多种方法，朴素的方法是穷举法，即枚举任意三个点构建一个面，检查所有其他点是否在该面的一侧，如果满足这个条件则表明该面是三维凸包的一个面。可以使用有向体积的方法判断其他点是否均在该面的一侧，如果所有其他点与该面的三个点所构成的向量的有向体积均为正或者均为负，表明其他点均在该面的一侧。穷举法实现简单，但是时间复杂度为 $O(n^4)$ ，效率不高。需要注意的是，在枚举可行的凸包表面时，避免选择共线的三个点作为凸包表面，此种退化情形可能会造成预想不到的错误。

强化练习：11769 All Souls Night^D。

更为高效的求三维凸包的方法是随机增量法，该方法直观且容易理解。首先将输入的点打乱顺序，然后选择四个不共面的点组成一个初始四面体，如果找不到这样的初始四面体，则凸包不存在。否则，每次加入一个点，不断更新当前的凸包即可。更新的方法是：

- (1) 如果当前点已经在凸包内，则不需更新；

- (2) 如果当前点在凸包之外, 那么找到所有这样的原凸包上的“分界边”——过这条边的两个面一个可以被当前点看到, 另一个不能。以这三个点新建一个面加入凸包中, 这样就得到了一个包含所有点的新的凸包。

以下是具体实现时需要注意的细节：

- (1) 为了便于判断点 p 是否在凸包内, 在保存三维凸包的面时, 使用三个点 p_1, p_2, p_3 表示一个面, 而且点 p_1, p_2, p_3 的排列顺序满足“右手螺旋法则”, 使得构成的面的法线始终指向凸包外侧, 这样就可以使用点 p 和构成面的三个点 p_1, p_2, p_3 所构成向量的有向体积来判断点 p 是否在凸包内, 如果有向体积为正, 表明点 p 位于凸包外, 若有向体积为负, 则点 p 在凸包内。

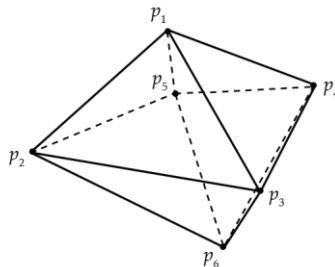


图 14-44 在保存三维凸包的面时, 对于由点 p_1, p_2, p_3 构成的凸包面, 在保存时按照逆时针方向保存, 即 $\{p_1, p_2, p_3\}$, 使得面的法线方向指向凸包外, 这样可以使得在用有向体积判断点 p 是否在能够看见某个面时保持一致性

- (2) 为了能够找到“分界边”——过此边的两个面一个能够被点 p 看到而另外一个不能被点 p 看到，可以将点 p 想象为一个点光源，从其发射光线照射到已经构建的凸包上，如果凸包的某个面能够被光线照射到，说明该面不属于新凸包的面，将该面的三条边予以标记；若某个面不能被点 p 发出的光照射到，说明该面属于新凸包的面。检查不能被点 p 发出的光照射到的面，即原凸包上仍然属于新凸包的面，如果面上的某条边被标记过，说明这条边就是“分界边”，将此边与点 p 构建一个面加入新凸包即可。判断凸包上的面是否能够被点 p 发出的光照射到，可以使用点 p 与构成面的三个点的有向体积进行判断，若有向体积为负，说

明点 p 发出的光能够照射到该面，也就是说从点 p 能够看到该面，否则从点 p 无法看到该面。

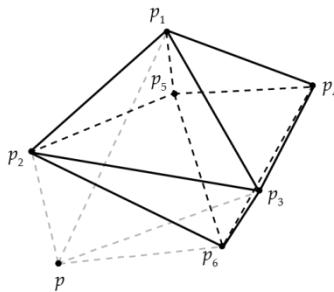


图 14-45 从凸包外一点 p 向三维凸包发射光线，面 $p_1p_2p_3$ 和 $p_2p_6p_3$ 均能被点 p 发出的光线照射，而其他面均不能被点 p 发出的光线所照射，因此面 $p_1p_2p_3$ 和 $p_2p_6p_3$ 不属于新的凸包，将这两个面的边 p_1p_2 , p_2p_3 , p_3p_1 , p_2p_6 , p_6p_3 予以标记，再检查未被 p 点照射的面，例如面 $p_1p_2p_5$ ，发现它的一条边 p_1p_2 已被标记，因此边 p_1p_2 属于“分界边”，将“分界边”与点 p 构成的面 p_1p_2p 加入新的凸包

随机增量法的朴素实现时间复杂度为 $O(n^2)$ 。在下述实现中，平面保存的是点的序号而不是实际的点数据。

```

//-----14.10.2.cpp-----
const int MAXN = 1100;
const double EPSILON = 1e-7;

// 判断给定的值是否为零值。
inline bool zero(double x) { return fabs(x) < EPSILON; }

// 三维空间点。
struct point3 {
    double x, y, z;
    point3 (double x = 0, double y = 0, double z = 0): x(x), y(y), z(z) {}
    point3 operator+(const point3 p) { return point3(x + p.x, y + p.y, z + p.z); }
    point3 operator-(const point3 p) { return point3(x - p.x, y - p.y, z - p.z); }
    point3 operator*(double k) { return point3(x * k, y * k, z * k); }
    point3 operator/(double k) { return point3(x / k, y / k, z / k); }
    bool operator<(const point3 &p) const {
        if (!zero(x - p.x)) return x < p.x;
        if (!zero(y - p.y)) return y < p.y;
        return z < p.z;
    }
    bool operator==(const point3 &p) const {
        return zero(x - p.x) && zero(y - p.y) && zero(z - p.z);
    }
} ps[MAXN];

// 三维空间平面。平面的三个点使用点的序号予以表示。
struct plane3 {
    int a, b, c;
    plane3 (int a = 0, int b = 0, int c = 0): a(a), b(b), c(c) {}
};

// 三维向量的模。
double norm(point3 p) {

```

```

        return sqrt(pow(p.x, 2) + pow(p.y, 2) + pow(p.z, 2));
    }

// 三维向量的外积。
point3 cross(point3 a, point3 b) {
    point3 r;
    r.x = a.y * b.z - a.z * b.y;
    r.y = a.z * b.x - a.x * b.z;
    r.z = a.x * b.y - a.y * b.x;
    return r;
}

// 三维向量的内积。
double dot(point3 a, point3 b) {
    return a.x * b.x + a.y * b.y + a.z * b.z;
}

// 有向体积。
double signedVolume(int p, int a, int b, int c) {
    return dot(ps[a] - ps[p], cross(ps[b] - ps[p], ps[c] - ps[p]));
}

// 有向面积。
double signedArea(int a, int b, int c) {
    return norm(cross(ps[b] - ps[a], ps[c] - ps[a]));
}

// 三维凸包面。
vector<plane3> faces;
// n 为点的数量, cnt 和 visited 为标记。
int n, cnt, visited[MAXN][MAXN];

// 构建初始凸包。
bool initializeConvexHull() {
    for (int i = 2; i < n; i++) {
        // 找到不共线的三个点。
        if (zero(signedArea(0, 1, i))) continue;
        swap(ps[i], ps[2]);
        for (int j = i + 1; j < n; j++) {
            // 找到不共面的四个点。
            if (zero(signedVolume(0, 1, 2, j))) continue;
            swap(ps[j], ps[3]);
            // 将面加入凸包, 此时凸包只有两个面, 而且这两个面是贴合在一起的, 但法线不同。
            faces.push_back(plane3(0, 1, 2));
            faces.push_back(plane3(0, 2, 1));
            return true;
        }
    }
    return false;
}

// 逐个点加入凸包, 注意此处使用的是点的序号而不是实际的点数据。
void addPoint(int p) {
    cnt++;
    // unlighted 保存从点 p 不可见的面。
}

```

```

vector<plane3> unlighted;
for (int i = 0, a, b, c; i < faces.size(); i++) {
    a = faces[i].a, b = faces[i].b, c = faces[i].c;
    // 检查点p和指定面构成的四面体的有向体积，若有向体积为负表明该面从点p可见，
    // 继而标记该可见面的所有边。
    if (signedVolume(p, a, b, c) < 0) {
        visited[a][b] = visited[b][a] = visited[a][c] = visited[c][b] = cnt;
        visited[c][a] = visited[b][c] = visited[c][b] = cnt;
    }
    else unlighted.push_back(faces[i]);
}
faces = unlighted;
// 对于不可见的面，如果面上的边被标记则该边是“分界边”。
for (int i = 0, a, b, c; i < unlighted.size(); i++) {
    a = unlighted[i].a, b = unlighted[i].b, c = unlighted[i].c;
    if (visited[a][b] == cnt) faces.push_back(plane3(b, a, p));
    if (visited[b][c] == cnt) faces.push_back(plane3(c, b, p));
    if (visited[c][a] == cnt) faces.push_back(plane3(a, c, p));
}
}

// 获取三维凸包的表面积。
double getAreaOf3DConvexHull() {
    // 对点排序并去除重复点。
    sort(ps, ps + n);
    n = unique(ps, ps + n) - ps;
    // 将点随机乱序。
    random_shuffle(ps, ps + n);
    faces.clear();
    // 确定初始三维凸包。
    if (initializeConvexHull()) {
        cnt = 0;
        for (int i = 0; i < n; i++)
            for (int j = 0; j < n; j++)
                visited[i][j] = 0;
        // 使用增量法求三维凸包。
        for (int i = 3; i < n; i++) addPoint(i);
        // 返回三维凸包的表面积。
        double area = 0;
        for (int i = 0; i < faces.size(); i++)
            area += signedArea(faces[i].a, faces[i].b, faces[i].c);
        return area / 2;
    }
    return -1.0;
}
//-----14.10.2.cpp-----//

```

扩展练习：1438* Asteroids^E，12308* Smallest Enclosing Box^{E^[178]}。

14.11 小结

由于计算几何需要处理浮点数，如何修正误差以使得计算结果尽量精确是一项基本技巧。为了达到控制误差的目标，引入了向量的概念，并定义了内积和外积，并在此基础上使用向量作为一种基本的“元件”来表示几何中的点、线段、直线，进一步构建各种几何元素之间的关系。使用基于向量的表示方法，一方面能

够使得搭建的“元件”鲁棒性尽可能的强，减少误差，另一方面也使得几何代码库能够统一、易于编码和调整，代码量也较小。

在掌握基本几何元素关系处理的基础上，需要重点掌握二维凸包、多边形面积求法、半平面交、最小圆覆盖。再之后就是熟悉与计算几何相关的算法和技巧，包括扫描线算法、旋转卡壳算法、坐标离散化。对于三维几何部分，主要是熟悉三维几何基本对象之间关系的判断，这与二维几何中集合对象之间关系的判断是类似的，重点需要掌握三维凸包的求法。

计算几何的内容非常丰富，在既往的 ACM-ICPC 中一直是一个重点内容，而在 NOI/CTSC 等竞赛中则不常见。与计算几何相关的题目，一般来说，难度不会很高，但需要解题者需要比较巧妙的思维，并且准备可靠的算法模板，对代码的组织、浮点数运算的精度控制也有一定要求。在实际编码中，为了尽量减少误差，能够使用整数运算代替的地方就使用整数运算。除了本章介绍的内容之外，学有余力的读者可以进一步学习圆的面积并、三维凸包的最小体积外接长方体等较难的内容。

附录

1 ASCII 表

Dec	控制字符	Dec	控制字符	Dec	控制字符	Dec	控制字符
0	空字符	32	(空格)	64	@	96	`
1	标题开始	33	!	65	A	97	a
2	正文开始	34	"	66	B	98	b
3	正文结束	35	#	67	C	99	c
4	传输结束	36	\$	68	D	100	d
5	请求	37	%	69	E	101	e
6	收到通知	38	&	70	F	102	f
7	响铃	39	'	71	G	103	g
8	退格	40	(72	H	104	h
9	水平制表符	41)	73	I	105	i
10	换行键	42	*	74	J	106	j
11	垂直制表符	43	+	75	K	107	k
12	换页键	44	,	76	L	108	l
13	回车键	45	-	77	M	109	m
14	不需切换	46	.	78	N	110	n
15	启用切换	47	/	79	O	111	o
16	数据链路转义	48	0	80	P	112	p
17	设备控制 1	49	1	81	Q	113	q
18	设备控制 2	50	2	82	R	114	r
19	设备控制 3	51	3	83	S	115	s
20	设备控制 4	52	4	84	T	116	t
21	拒绝接收	53	5	85	U	117	u
22	同步空闲	54	6	86	V	118	v
23	传输块结束	55	7	87	W	119	w
24	取消	56	8	88	X	120	x
25	介质中断	57	9	89	Y	121	y
26	替换	58	:	90	Z	122	z
27	退出	59	;	91	[123	{
28	文件分隔符	60	<	92	\	124	
29	分组符	61	=	93]	125	}
30	记录分隔符	62	>	94	^	126	~
31	单元分隔符	63	?	95	-	127	删除

2 C++运算符优先级

优先级	操作符	描述	结合性
1	<code>::</code>	域解析	自左向右
2	<code>a++ a--</code>	后缀自增/后缀自减	
	<code>a()</code>	括号	
	<code>a[]</code>	数组下标	
	<code>.</code>	成员选择 (对象)	
	<code>-></code>	成员选择 (指针)	
3	<code>++a --a</code>	前缀自增/前缀自减	自右向左
	<code>+ -</code>	加/减	
	<code>! ~</code>	逻辑非/按位取反	
	<code>(type)</code>	强制类型转换	
	<code>*a</code>	取指针指向的值	
	<code>&a</code>	取变量的地址	
	<code>sizeof</code>	取数据类型的大小	
	<code>new new[]</code>	动态内存分配/动态数组内存分配	
	<code>delete delete[]</code>	动态内存释放/动态数组内存释放	
4	<code>. * ->*</code>	成员对象选择/成员指针选择	自左向右
5	<code>a*b a/b a%b</code>	乘法/除法/取模	
6	<code>a+b a-b</code>	加号/减号	
7	<code><< >></code>	位左移/位右移	
8	<code><=></code>	三方比对 (c++20 标准支持)	
9	<code>< <=</code>	小于/小于等于	
	<code>> >=</code>	大于/大于等于	
10	<code>== !=</code>	等于/不等于	
11	<code>a&b</code>	按位与	
12	<code>^</code>	按位异或	
13	<code> </code>	按位或	
14	<code>&&</code>	与运算	
15	<code> </code>	或运算	
16	<code>a?b:c</code>	三目运算符	自右向左
	<code>throw</code>	抛出异常	
	<code>=</code>	赋值	
	<code>+= -=</code>	相加后赋值/相减后赋值	
	<code>*= /= %=</code>	相乘后赋值/相除后赋值/取余后赋值	
	<code><= >=</code>	位左移赋值/位右移赋值	
	<code>&= ^= =</code>	位与运算后赋值/位异或运算后赋值/位或运算后赋值	
17	<code>,</code>	逗号	自左向右

3 解题提示

107 The Cat in the Hat。除了可以使用素因子分解的方法解题外，此题也可使用二分搜索或对数解决。使用对数解题需要注意数值精度问题。使用二分搜索解题的方法可参阅：http://algorithmist.com/index.php/UVa_107, 2020。

135 No Rectangles。手工列出当 k 较小时的可行方案，观察寻找规律。

208 Firetruck。由于街角的数量不超过 20，可以通过回溯来确定所有可能的路径。但是由于此题本质是求无向连通图的所有简单通路问题，如果在回溯过程中不剪枝，对于特殊的测试数据，代码很容易超时。例如给定一组共有 20 个街角的测试数据，其中街角 1 和街角 4 相连构成一个连通子图，所有其他街角构成另外一个连通子图，但与街角 1 和街角 4 不连通，对于这样的测试数据，如果不加剪枝，在回溯时大量时间将耗费在枚举不可能的路线上。一种可行的剪枝做法是先使用 Floyd-Warshall 算法确定各个街角之间的连通性，如果两个街角是不连通的，则不需要在此线路上继续进行搜索，从而能够节省时间，提高效率。

248 Cutting Corners。为了达到递送路线距离最短的目的，信使应该从起点出发，不断的从一座建筑的某个“角落”沿直线到达另外一座建筑的某个“角落”，最终到达终点。中途不能穿越其他建筑，可以沿着建筑的边界行进。可以预先处理得到所有可达的“角落”之间的距离，之后使用 Moore-Dijkstra 算法求最短距离。

250 Pattern Matching Prelims。注意浮点数精度问题。

279 Spin。当游戏长度较小时，模拟游戏的进行，列出各种情况的移动步骤，总结发现规律。

286 Dead Or Not-That Is The Question。题目描述中未明确说明“升变”规则如何运用，按照评判程序的设定，当己方兵到达对方棋盘的底线时，仍然将其视为兵而不是其他棋子。

295 Fatman。将障碍物视为图中的顶点，将走廊的上侧边和下侧边也视为障碍，构建无向图，运用二分搜索解题。令胖子的直径为 d ，将相互距离小于 d 的障碍物用无向边连接起来构成无向图，在此无向图中检测上侧边和下侧边对应的顶点是否连通，若不能连通，则表明直径为 d 的胖子能够穿越障碍到达走廊对侧，可以增大 d ；若能够连通，则表明直径为 d 的胖子在穿行过程中必定会碰到一条边，这条边所连接的两个障碍物之间的距离小于 d ，导致胖子无法通过，因此应减少 d 。反复进行上述过程，直到 d 的值满足精度要求。

313 Intervals。截至 2020 年 1 月 1 日，该题并未采用“Special Judge”，避免使用三角函数计算切线与 X 轴的交点以免引入较大的浮点数误差而导致 Wrong Answer。

359 Sex Assignments and Breeding Experiments。根据题意可知有向图需要满足以下约束：(1) 任意一个顶点的入度不超过 2；(2) 有向图不存在圈；(3) 能够为有向图的顶点指定一个性别，使得遗传关系能够满足。第 (2) 条约束可通过 DFS 进行检查；第 (3) 条约束可通过 2-SAT 进行检查。

425 Enigmatic Encryption。为了能够正确编译源代码，需要添加头文件 `<crypt.h>` 或者 `<unistd.h>`（函数 `crypt` 所在头文件 `<crypt.h>` 已经包含在头文件 `<unistd.h>` 中），同时在编译时使用链接选项 `“-lcrypt”`。

430 Swamp County Supervisors。经 `assert` 语句测试，评测数据中的“the minimum number of votes needed for passage of an ordinance”，即“最小通过票数”的最大值不超过 10000。

431 Trial of the Millennium。截至 2020 年 1 月 1 日，UVa OJ 上的评判程序似乎存在 Bug，尝试了多种解题方案均无法获得 Accepted。

435 Block Voting。由于此题的数据量较小，可以通过使用回溯法构造所有可能的子集和并予以计数来解题。与此题类似的是 430 Swamp County Supervisors，该题数据量较大，使用回溯法构造所有子集和的方法效率较低，无法在规定时间内获得 Accepted，使用动态规划解决较为适宜。

475 Wild Thing。给定的模式串中可能包含连续的星号通配符。

571 Jugs。此题给定的是两个容器且两个容器的容量互素，这使得问题容易解决。可以将题目建模为在隐式图中寻找有向路径的问题。感兴趣的读者可以尝试解题 10603 Fill，此题将容器数量增加至三个且取消了容量互素的条件，需要应用图算法才能顺利地解决。

580 Critical Mass。计数结果为 OEIS 序号为 A050231 的数列。

581 Word Search Wonder。当 $s \leq 0$ 时，对于给定的单词均需要输出“NOT FOUND”。可将转换得到的字符矩阵“补齐”以便判定搜索时的“越界”问题。

1641 ASCII Area。朴素的方法是使用 Flood-Fill 算法将多边形外侧的字符 ‘.’ 更改为除 ‘/’、‘\’、‘.’ 以外的其他字符（例如，‘*’），之后统计字符 ‘/’、‘\’、‘.’ 的个数，字符 ‘/’、‘\’ 对面积的贡献为 0.5，字符 ‘.’ 对面积的贡献为 1。为了便于应用 Flood-Fill 算法，可以先将给定的二维字符数组外周加上一圈字符 ‘.’，这样使得多边形外侧的字符 ‘.’ 能够连通以便于处理。在应用 Flood-Fill 算法时需要注意，如果某个方格为字符 ‘.’，若其上侧、下侧、左侧、右侧方格重任意一个方格的字符也为 ‘.’，则视为连通。若左上角方格的字符为 ‘.’，则需要检查上侧和左侧方格字符是否为 ‘\’，如果是，也视为连通，类似地处理右上角、左下角、右下角方格字符为 ‘.’ 的情形。更为巧妙的方法是应用多边形 Jordan 曲线定理。注意到字符 ‘/’、‘\’ 表示多边形的边界，当从左至右逐行扫描给定的字符数组时，统计遇到的边界字符数量，如果遇到过的边界字符为奇数，表明当前处于多边形的内部，此时再遇到的字符 ‘.’ 位于多边形内部，若遇到过的边界字符为偶数，则当前已经位于多边形外侧，此时再遇到的字符 ‘.’ 位于多边形外侧。由于在逐行扫描字符数组时，初始时位于多边形外，最后也位于多边形外，则遇到的边界字符必定为偶数，因此多边形的面积必定为整数。

645 File Mapping。注意边界输入的处理。例如，同一文件夹下虽然不能包含相同名称的文件夹或文件，但是不同文件夹却可以包含名称相同的文件夹或文件，即可能有类似于“ROOT/dir1/dir”和“ROOT/dir2/dir”的文件结构。

652 Eight。由于 8 数码问题搜索空间较小，可以预生成所有布局的走法序列，从而能够以查表的方式来输出解。或者使用 IDA* 搜索实时生成解。

658 It's Not a Bug It's a Feature。本题实质上是加权有向图的最短路径问题。由于题目所对应的隐式图中状态（顶点）较多，而且很多状态（顶点）可能在实际求最短路径的过程中并未使用，因此预生成完整的图会耗费较多时间，可以采取现场计算的方法生成顶点的有向边，这样可以减少运行时间，从而获得通过。可以通过位运算技巧来加速邻接边的生成。

676 Horse Step Maze。将迷宫视为国际象棋棋盘，按照题目给定的行走方式，从黑色方格出发只能进入黑色方格，从白色方格出发只能进入白色方格，亦即坐标差值的绝对值必须具有相同的奇偶性，若不同，则不可达，若相同则肯定可达。当起点坐标和终点坐标相同时，只需输出起点坐标即可。

684 Integral Determinant。在计算行列式 (determinant) 的值时，可以利用其以下性质：(1) 将行列式的任意两行（或两列）互换，行列式的值的符号改变；(2) 将行列式中的一行乘以某个数加到另外一行上（或将一列乘以某个数加到另外一列上），行列式的值不改变；(3) 将行列式变换为对角线行列式（除了对角线元素外其他元素均为 0），或上三角行列式（行列式的主对角线以下的元素均为 0），或下三角行列式（行列式的主对角线以上的元素均为 0），行列式值为对角线元素的乘积；(4) 如果行列式某一行（或某一列）的元素全为 0，则行列式值为 0；(5) 将行列式的一行（或一列）乘以某个数 k ，则行列式值变为原来的 k 倍。可以看到，性质 (2) 和高斯消元法中的消元操作相似，因此可以采用类似于高斯消元的方法将给定的行列式转换为上三角行列式，然后将对角线元素相乘即可得到行列式的值。需要注意，在普通的高斯消元法中，在处理到第 i 行时，需要将第 i 行以外的其他行减去第 i 行的若干倍，使得系数矩阵成为对角线矩阵（即除了对角线元素外其他元素均为 0），而在行列式的操作中，因为性质 (3)，并不需要将其转换为对角线行列式，只需将其转换为上三角行列式即可，因此只需将第 i 行以后的行减去第 i 行的若干倍即可，使得第 i 行以后的所有行中第 i 列的元素变为 0。另外，由于需要保证结果为整数，则在消元时不能使用浮点数除法，而应使用以下方法：假设当前正在处理第 i 行，则需要找到第 i 行（包括第 i 行）以后的所有行中，第 i 列的绝对值最小但又不为 0 的一行，将这一行与第 i 行互换（此时，行列式的值的符号会发生改变），之后再将第 i 行以后的行与第 i 行的若干倍进行减法操作，重复进行上述操作，直到第 i 行第 i 列元素变为 1，或者除了第 i 行第 i 列元素不为 0 外，第 i 行以后的行中第 i 列元素均为 0（类似于在欧几里得算法中使用辗转相除的方法来求最大公约数的过程）。如果在寻找的过程中发现某一行的第 i 列元素为负值，可以将该行的所有元素乘以 -1 ，将其转换为正值以便进行减的操作。

697 Jack and Jill。注意计量单位的转换：1 feet=12 inches。在进行实数大小的比较时需要运用误差控制，否则不易获得 Accepted。

704 Color Hash。由于题目所蕴含的隐式图搜索空间较大，可使用双向搜索降低搜索量以提高效率，否则容易超时。

709 Formatting Text。对于给定一行，当单词长度之和一定时（该行单词长度之和，加上必要的空格——相邻两个单词至少一个空格，不超过一行的宽度限制），欲使得该行的“badness”最小，应该使得相邻单词间的空格尽量平均分布，即任意两个相邻单词之间的空格数量最多相差一个。例如宽度限制为 28，某行单词个数为 5 个，单词长度之和为 19，则由于 5 个单词之间至少需要 4 个空格分隔，则至少需要占用 23 的宽度，剩余的 5 个空格再平均分配给 4 个单词间隔，则具有最小“badness”且符合题意最后的输出要求的空格分配方案为 2, 2, 2, 3（空格分配方案 2, 3, 2, 2 具有同样的“badness”但是不符合输出要求）。

736 Lost in Space。(1) 评测数据包含多组测试数据，每两组测试数据之间有一个空行分隔。(2) 从“N”开始，按顺时针选择方向并按此方向进行精确匹配，中途不能改变方向，需要忽略空格。(3) 匹配到达边界时，不允许绕过边界到达对端继续匹配。

743 The MTM Machine。注意，规则 1 不能递归应用，而规则 2 可以递归应用，即给定输入“33225”，应该输出“25225225225”，而不是“5252525”。

751 Triangle War。如果玩家连线后能够构成三角形则奖励一次连线权，可叠加，即若能再次连线构成三角形，又奖励一次连线权，直到不能构成三角形为止，此时连线权交由对方玩家。

756 Biorhythms。注意边界情况的处理，如“1 1 1 0”这样的测试数据。

790 Head Judge Headache。截至 2020 年 1 月 1 日，该题的输入输出格式仍未明确指定，导致题目本身不难，通过率却较低。以下是题目描述中未明确说明的事项：(1) 输入由多组测试数据构成；(2) 两组相邻的测试数据输出之间有一个空行；(3) 输出中出现的队伍数量要求达到输入中曾经出现过的最大队伍编号数。例如，某组测试数据中只有一条记录“22 A 1:33 Y”，那么在输出中要包含队伍 1 至 22 的排名结果；(4) 如果某支队伍对同一道题目的两次提交时间相同，但结果一次是‘Y’，一次是‘N’，则认为错误的提交在先，需要计算罚时 20 分钟。(5) 输入中可能出现“陷阱”，即某个队伍对同一道题目的提交，结果为‘N’的输入顺序在前，提交时间在后，结果为‘Y’的输入顺序在后，提交时间却在前。如果按照输入顺序处理，可能会误认为结果为‘N’的提交需要计算罚时，而实际上由于结果为‘Y’的在时间上靠前，根据时间计算规则，应该忽略结果为‘N’的提交。

795 Sandorf's Cipher。注意边界情形的处理，例如，给出的输入中一行包含的字符数可能不一定是 36 个，而可能是 72 个或更多，可能出现原始字符串中包含‘#’字符的情形。

800 Crystal Clear。 n 边形的内角和为 $2 \times (n-2) \times \pi$ 弧度。给定的多边形是简单多边形，但有可能是凹多边形，因此需要注意处理这样的情形：尽管晶体的圆心在多边形的边界上，但由于多边形可能为凹多边形，与圆心所在边界相邻的边可能切割晶体从而导致该晶体不符合题意要求，则其面积不能计入有效面积。

830 Shark。截至 2020 年 1 月 1 日，该题在 UVa OJ 上的评判数据仍存在问题。经 assert 语句测试，在至少一组测试数据中，字符矩阵的实际列数比输入中所指定的列数 C 多一列，如果使用逐个字符读取的方式处理输入可能会得到错误的答案（可以借助 `cin.ignore(1024, '\n')` 忽略行末多余的字符来避免此问题）。如果使用 `getline(cin, line)` 先读取一行输入，然后再将其赋值到字符矩阵中，则能够正确处理。

835 Square of Primes。如果不考虑顺序，使用回溯法进行搜索很容易超时。根据题意，最后一行和最后一列的素数必定是数位全部为奇数的素数，则当给定和为偶数时，必定无解。在 10000 至 99999 之间，共有 8363 个素数，而其中全部数位均为奇数的素数共有 608 个。因此按照如下策略来设置搜索顺序可以显著提高搜索效率：预先生成具有指定和且首位数字为给定数字的所有素数，先确定主对角线上的素数，之后枚举位于最后一列和最后一行的素数，接着再确定倒数第二列的素数，之后再确定第一行至第四行的素数，在确定第一行至第四行的素数时，之前的素数已经确定了至少 2 个数位（第一行至第三行已经确定了 3 个数位），因此搜索范围可以进一步缩小。最后只需进行第一列至第三列和副对角线的素数检测，如果通过，则为符合题意的一种素数方阵。

840 Deadlock Detection。题目的输出格式并未明确指定，截至 2023 年 5 月 1 日，使用 DFS 检测圈，按照圈的长度大小从小到大排列输出，对于同一个圈，按照字典序最小的圈进行输出（例如，圈为“a-B-b-A-a”，在输出时应调整顺序，输出“A-a-B-b-A”），可以获得 Accepted。

882 The Mailbox Manufacturers Problem。可以将题目的要求概括如下：使用一种固定的策略对邮箱进行测试，使得不论邮箱的可承受鞭炮当量如何，此种策略所需的鞭炮数量是最少的。以 $k=2, m=10$ 为例，令邮箱分别为 A 和 B ，在此种情况下，选择先放入邮箱 A 中 2 枚鞭炮，如果邮箱 A 爆破，则只需再使用 1 枚鞭炮测试邮箱 B ，总共只需 $2+1=3$ 枚鞭炮；若邮箱 A 未爆破，接着将 7 枚鞭炮放入邮箱 A 中（当前总共使用 $2+7=9$ 枚鞭炮），如果邮箱 A 爆破，则使用剩下的邮箱 B 测试 3、4、5、6 枚的鞭炮当量，共需 $2+7+3+4+5+6=27$ 枚鞭炮；若邮箱 A 仍未爆破，则继续将 9 枚鞭炮放入邮箱 A 中（当前共使用 $2+7+9=18$ 枚鞭炮），如果邮箱 A 爆破，则使用剩下的邮箱 B 测试 8 枚鞭炮，共需 $2+7+9+8=26$ 枚鞭炮；如果邮箱 A 还未爆破，可以继续使用邮箱 A 测试 10 枚鞭炮，则总共需要 $2+7+9+10=28$ 枚鞭炮。综合所有情形，即使在最坏的情况下，使用 2、7、9、10 的鞭炮当量策略，最多需要 28 枚鞭炮。读者可以验证，其他的测试策略不会比前述的测试策略更优。

886 Named Extension Dialing。（1）先按照输入的顺序对分机号进行精确匹配，如果有匹配的分机号则不必再匹配姓名所对应的按键编码。（2）如果没有精确匹配的分机号，则匹配姓名对应的按键编码。题目描述“*If you know your party's name, dial the first letter of the first name followed by the first letters of the last name of your party now*”并没有明确表达出题者的意图。若要获得 Accepted，需要将 First Name 的首字符和 Last Name 的全部字符进行编码，然后在此编码字符串中检查给定输入是否构成该字符串的前缀，如果构成前缀则表示输入匹配该姓名，输出姓名所对应的分机号即可。（3）输出时按照输入给出的先后顺序输出分机号。

888 Donkey。先考虑特殊情形。当 $N=1$ 时，显然第一个玩家获胜的概率是 1。当 $M=0$ 时，若当前轮到第一个玩家玩游戏，则获胜的概率是 1，否则为 0。考虑 $N=2$ 的情形，令 $dp[i][p_1][p_2]$ 表示“第 i 个玩家玩游戏且两个玩家的位置依次在 p_1 和 p_2 时第一个玩家获胜的概率”， $p_1 \neq p_2$ 。由于掷骰子得到 1 到 6 点的概率均为 $1/6$ ，若当前为第一个玩家玩游戏，则递推关系式为

$$dp[1][p_1][p_2] = \sum_{j=1}^6 \frac{dp[2][\min\{x \mid x \geq p_1 + j, x \neq p_2\}][p_2]}{6}$$

类似的，若为第二个玩家玩游戏，递推关系式为

$$dp[2][p_1][p_2] = \sum_{j=1}^6 \frac{dp[1][p_1][\min\{x \mid x \geq p_2 + j, x \neq p_1\}]}{6}$$

根据题目条件约束，两个玩家谁首先跨越第 M 条河谁就获胜，则 $dp[1][\{x \mid x > M\}][p_2] = 1, dp[2][p_1][\{x \mid x > M\}] = 0$ 。类似的，可以得到 $N=3$ 和 $N=4$ 时的递推关系式，结合备忘技巧解题即可。由于题目约束 $1 \leq N \leq 4, N \times M \leq 50$ ，则 M 最大值为 25，使用 5 个二进制位足以表示，则可以将所有玩家的位置以二进制的形式编码为一个整数进行处理，有利于备忘技巧的应用。需要注意，UVa OJ 上的评测数据量非常大（经使用 assert 语句测试，至少有 150000 组测试数据），必须保存已有的计算结果以便查表输出，否则很容易超时。虽然使用位掩码技巧便于状态的统一处理，但是在具体操作时需要从位掩码中解码出所有玩家的位置，在更新当前玩家的位置后又需要将所有玩家的位置重新编码为位掩码，费时较多，且用于保存已有运算结果时需要加上河流的条数 M 以及初始玩游戏的玩家序号 i 这两个状态参数，导致存储计算结果的数组的大小会超出内存限制。由于本题并未采用“Special Judge”，但运算过程中存在浮点数误差，使得解题者的输出与评测数据的输出可能存在细微差异，例如 $1/16=0.0625$ ，在输出时需要四舍五入为 0.063，使用 C++ 的 `setprecision` 操纵子控制输出精度时要为结果加上一个很小的常数 (`1E-8`) 才能使得输出为 0.063，这似乎能够在一定程度上解释为何该题的通过人数和通过率都非常低。

889 Islands。两个简单多边形之间的距离为构成多边形的线段之间距离的最小值。

959 Car Rallying。一个“move”结束之后不管是否改变速度，都从下一个“unit”开始另外一个“move”。到达最终的“unit”之后，还有一个“unit”，此“unit”无速度限制，需要经过此“unit”后比赛才结束。由于需要查询区间的最小值，可以进行预先处理以提高效率，否则容易超时。

963 Spelling Corrector。截至 2020 年 1 月 1 日，在 UVa OJ 的评测数据中，某些输入行的行首或行末或者行内单词之间可能有多个空格，为了获得 Accepted，需要将这些空格原样打印至输出中。

986 How Many。此题难点在于确定动态规划状态间的关系。考虑网格中的某个格点，令其坐标为 (x, y) ，如果 $y > 0$ ，则该格点既能接受来自左上方格点 $(x-1, y+1)$ 的路径也能接受来自左下方格点 $(x-1, y-1)$ 的路径；若 $y = 0$ ，则该格点只能接受来自左上方格点 $(x-1, y+1)$ 的路径。令 $dp[x][y][r][0]$ 表示从左上方格点 $(x-1, y+1)$ 出发并终止于格点 (x, y) 且具有 r 个高度为 k 的峰的路径总数， $dp[x][y][r][1]$ 表示从左下方格点 $(x-1, y-1)$ 出发并终止于格点 (x, y) 且具有 r 个高度为 k 的峰的路径总数，那么只有当满足条件“ $y+1$ 等于 k ”时，从左下方到达格点 $(x-1, y+1)$ 处的路径才能向右下到达格点 (x, y) 时形成一个高度为 k 的峰。可以使用备忘技巧并结合“从后往前”的递归予以解决。读者可以进一步思考如下扩展问题：如果可行的走法还包括水平向右平移一步 $(1, 0)$ ，即可从 (x, y) 到达 $(x+1, y)$ ，其他约束条件不变，那么本问题应该如何求解？

1006 Fixed Partition Memory Management。程序的时限 (turnaround) 是指从最开始 (0 秒) 到程序运行结束的时间，而不是程序在特定大小内存区域运行所需的时间 t_i 。也就是说，如果有程序先于当前程序运行，则当前程序的起始运行时间为先前运行程序的时限。可以将题目约束建模为二部图最小权匹配问题予以解决。解题的难点是如何获得平均最短时限。有两种处理方式，第一种方式是通过多个轮次的最小权匹配来逐步获得所有程序的最短时限，即将程序视为 X 侧顶点，内存区域视为 Y 侧顶点，当内存区域大小满足程序的内存要求时，在程序和该内存区域间连接一条容量为 1 权值为程序运行时间的有向弧，使用最小费用最大流算法得到第一轮最小权最大匹配的结果，如果第一个轮次未能将所有程序运行完毕，则将已经运行的程序剔除，对剩余的程序重新建立容量网络，再次求最小权最大匹配，直到所有程序运行完毕。需要注意，后续在建立容量网络时，有向弧的权值应该是内存区域已经累积执行的程序时间加上需要在该内存区域上执行的程序的运行时间。例如，假设内存区域 1 已经先后运行了 A、B 两个程序，A 程序的运行时间为 10 秒，B 程序的运行时间为 6 秒，则 A 程序的时限为 10 秒，运行时间从 0 秒到 10 秒，B 程序的时限为 16 秒，运行时间从 10 秒到 16 秒，若后续拟运行程序 C，其运行时间为 20 秒，则在容量网络中，程序 C 与内存区域 1 所对应的顶点间连接的有向弧其权值应该为 16 秒 + 20 秒 = 36 秒。不过由于 UVa OJ 上的测试数据规模较大，使用多个轮次的最小费用最大流求最小权最大匹配难以在限定时间内获得通过。第二种方式是将每个程序在内存区域上执行的次序 k 考虑在内，一次性建立容量网络的所有弧，通过一个轮次的最小费用最大流来求得最小权最大匹配。考虑 A、B、C 三个程序先后在内存区域 1 运行，其运行时间分布为 10 秒、6 秒、20 秒，则三个程序的时限分别为 10 秒、16 秒、36 秒，不难看出，在统计总的时限时，相当于 A 程序的运行时间被统计了三次，B 程序的运行时间被统计了两次，C 程序的运行时间被统计了一次，由此可以看出，在建立容量网络时，边权值 w (表示单个程序对总的运行时限的贡献) 和程序在该内存区域上的运行时间 t 的比值正为次序 k 。

1025 A Spy in the Metro。利用两个域来表示 Maria 的状态，即所处的站点编号和当前时间，将状态视为图的顶点，根据题目所给条件在状态之间建立有向边，最后使用 Moore-Dijkstra 算法求最短路径。

1039 Simplified GSM Network。此题的难点在于确定从城市 A 到城市 B 的道路所需转换的 BTS 个数，有两种方法：(1) 将城市 A 到城市 B 的道路视为一条线段 s ，对于某个 BTS，令其为 t ，求出和 s 的距离 d 和最近点 p ，若 p 和除 t 之外的 BTS 之间的距离均大于 d ，表明若沿线段 s 行进，在某个时刻， t 离 s 最近，则 t 在需要转换的 BTS 之列。(2) 将城市 A 到城市 B 的道路视为一条线段 s ，其端点分别为 p_A 和 p_B ，求出与端点 p_A 和 p_B 最近的 BTS，令其为 t_A 和 t_B ，若 t_A 和 t_B 为同一个 BTS，表明该 BTS 在需要转换的 BTS 之列，停止检查；若 t_A 与 t_B 不同，则取 s 的中点 p_M ，构成两条线段，分别为 s_1 和 s_2 ， s_1 的端点为 p_A 和 p_M ， s_2 的端点为 p_B 和 p_M ，继续前述检查。

1040 The Traveling Judges Problem。题目所求为包括裁判所在城市和目标城市在内的一棵最小生成树。枚举城市的所有可能组合，确定满足要求的最小生成树，之后使用 BFS 得到最短路径。需要注意，在输出时，每组测试数据的输出后面都要输出一个空行，这在题目描述中未予明确说明。

1052 Bit Compressor。在解题时，需要准确理解题目描述中的约束条件“Replace any maximal sequence of n 1's with the binary version of n whenever it shortens the length of the message.”。

1057 Routing。解题思路可参阅：<http://apps.topcoder.com/forums/?module=Thread&threadID=662332>, 2020。

1069 Always an Integer。题目描述中指出输入以包含一个‘.’的行作为结束，但是测试数据中最末一行可能包含多余的空格，需要予以注意。可以通过数学归纳法证明，对于给定的多项式 P ，令 k 是多项式 P 中 n 的最高幂次，只需验证 $n=1, 2, \dots, k+1$ 时，多项式 P 是否能够被 D 整除，若能够整除，则对于所有的正整数，多项式的结果均为整数。证明简要描述如下：当 $k=0$ 时， P 中不包含 n 的幂次，只需验证 $P(1)$ 即可；当 $k=1$ 时， P 是 n 的一次多项式，设为 $an+b$ ，则 $P(n+1)-P(n)=a$ ，如果把 $P(n)$ 视为数列的第 n 项，那么 $P(1), P(2), \dots, P(n)$ 构成了一个首项为 $P(1)$ ，公差为 a 的等差数列，如果首项 $P(1)$ 和公差 $a=P(2)-P(1)$ 均能被 D 整除，显然 $P(n)$ 能够被 D 整除，实际相当于验证 $P(1), P(2)$ 是否能够被 D 整除；当 $k=2$ 时， P 是 n 的二次多项式，设为 an^2+bn+c ，则 $P(n+1)-P(n)=2an+a+b$ ，是关于 n 的一次多项式，类似地，可以将其视为首项为 $P(1)$ ，公差为 $2an+a+b$ 的等差数列，只需确定 $2an+a+b$ 是否能够被 D 整除，若能够被 D 整除，则整个数列均能被 D 整除，根据前述讨论，只需验证 n 为 1 和 2 时， $2an+a+b$ 是否能够被 D 整除，当 n 为 1 时， $2a+a+b=P(2)-P(1)$ ，当 n 为 2 时， $4a+a+b=P(3)-P(2)$ ，相当于验证 $P(1), P(2), P(3)$ 是否能够被 D 整除，依此类推，可以根据数学归纳法证明，对于最高次数为 k 的关于 n 的多项式，只需验证 $1, 2, \dots, k+1$ 时该多项式的值是否能够被 D 整除，若均能被整除，则对于任意正整数，该多项式的值均能被 D 整除。

1076 Password Suspects。本质是有向图的路径计数问题，可根据 Aho-Corasick 算法为所有模式建立转移图，在转移图基础上添加必要的有向边以构建完整的解题所需的有向图，动态规划的状态即在有向图的各个顶点间转移。动态规划的状态包含三个域：字符串的长度，有向图中当前所处顶点的编号，已经匹配的关键字组合的位掩码标记。如果可能的密码数量不超过 42 个，则根据动态规划过程中路径计数不为零的状态进行回溯输出。

1079 A Careful Approach。遍历所有可能的飞机降落顺序，对于每一种降落顺序，二分搜索最大可能的“minimum achievable time gap”。

1091 Barcodes。需要检查给定的条形码宽度是否在题目指定的误差范围内，若超出误差范围则为“bad code”。注意检查两个字符条码间的分割线是否符合要求。Code-11 编码的格式参阅：<http://wwwbarcodeisland.com/code11.shtml>, 2020。

1093 Castles。每个城堡在进攻时需要一定数量的士兵 a ，在攻击城堡的过程中会损失一部分士兵 m ，同时还需要若干士兵 g 留在城堡中继续防守。易知，攻占该城堡至少需要的士兵数量为 $s = \max(a, m+g)$ ，实际使用的士兵数量为 $u = m+g$ ，剩余可用士兵数量 $r = s-u$ 。由于给定图为一棵树，若城堡只有单个子结点，则可按上述方法叠加计算，若城堡具有多个子结点，则需要考虑子结点累加的顺序，最优的策略是按 r 值递增的顺序进行累加，这样可以获得最少的士兵数量。此外，需要考虑以每一个城堡作为根结点开始进攻，取所有情形的最小值才为正确的解。

1096 The Islands。由于双调路程在“去程”和“回程”需要分别经过不同的两个特定的岛屿，导致最终具有最短距离的旅程可能出现交叉的情形，使用本节介绍的递推关系式进行计算存在困难，不妨从另外一个角度考虑。题目所求实际上是一条哈密顿回路，回路可以看成是两条从左至右、不重复地覆盖所有点的路径，令 $F[i][j]$ 表示第一条路径走到 i 、第二条路径走到 j 的最短距离，为了避免重复经过同一点的情况，若两条路径的走前状态为 (i', j') ，则走后的状态仅有两种： $(i', \max\{i', j'\}+1)$ 或者 $(\max\{i', j'\}+1, j')$ 。由此得出递推关系式：

$$F[i][\max(i, j) + 1] = \min\{F[i][\max(i, j) + 1], F[i][j] + d(j, \max(i, j) + 1)\}$$

$$F[\max(i, j) + 1][j] = \min\{F[\max(i, j) + 1][j], F[i][j] + d(i, \max(i, j) + 1)\}$$

其中 $d(x, y)$ 表示岛屿 x 和岛屿 y 之间的欧式距离。在具体编码实现时，限制第一条路径经过 b_1 而不经过 b_2 ，限制第二条路径经过 b_2 而不经过 b_1 ，以满足题目约束条件。参阅：吴永辉，王建德等编著，《ACM-ICPC 世界总决赛试题解析（2004—2011 年）》，第 302—305 页。

1103 Ancient Messages。初看似乎无从着手，但是仔细观察给出的符号可以发现，每个符号由黑色轮廓所包围的白色连通区域数是互不相同的，分别为 Ankh (1)、Wedjat (3)、Djed (5)、Scarab (4)、Was (0)、Akhet (2)。因此只需得到符号内所包含的白色“空洞”数，就能确定是哪个符号，“空洞”实际上对应拓扑学中的“亏格”概念。

1153 Keep the Customer Satisfied。按照订单的截止时间递增排序，对于当前订单 J_v ，如果可以在截止时间之前完成就接受，如果无法完成，就从已接受订单中选择一个要求生产钢铁吨数最大的订单，令其为 J_u ，比较订单 J_u 和 J_v 所要求生产的钢铁吨数，若订单 J_u 所要求生产的钢铁吨数 d_u 大于当前订单 J_v 所要求生产的钢铁吨数 d_v ，则将订单 J_u 拒绝，接受订单 J_v 。获取要求生产钢铁吨数最大的订单，可以使用优先队列来实现。

1185 Big Number。由于斯特林公式为近似公式，当 n 较大时可能存在较大误差，使用如下的方式计算阶乘的位数一般更为精确：给定 $n!$ ，其十进制表示下的位数 $d = \text{floor}(\log_{10}(n!)) + 1$ 。考虑对数的性质及精度误差，可得 $d = \text{floor}(\log_{10}(1) + \log_{10}(2) + \dots + \log_{10}(n) + 1e-7) + 1$ 。floor 为向下取整函数。

1205 Color a Tree。本题可以归结为以下问题：给定一棵有根树，树中的每个结点表示一项工作，每项工作都有一个权值（表示该工作的重要性或价值），完成每项工作的所需的单位时间相同。从 0 开始计算工作的完成时间，如果一项工作延迟完成，则给予一定的惩罚，惩罚使用工作延迟完成的单位时间与工作权值的乘积来表示。每项工作只有当它的前置工作被完成后才能够被完成（对于树来说，即父结点表示的工作完成后，子结点所表示的工作才能被完成），要求确定一种完成所有工作的顺序方案，使得总的惩罚最少。Horn 研究了该问题并给出了简单的算法来解决上述问题（读者可以参考标注给出的论文）。Horn 关于该问题的算法虽然在算法步骤上有所不同，但实际效果等效于使用以下最优策略：对于树中权值最大的结点所对应的工作，最优策略是在其父结点所表示的工作完成之后立即完成该工作，这样可以得到最小的总惩罚值。解题思路是不断寻找具有最大权值的结点与父结点合并，计算合并后的平均权值，并将其作为一个新的结点予以考虑，直到最后整棵树合并为一个结点。朴素的实现其时间复杂度为 $O(n^2)$ ，优化的实现其时间复杂度为 $O(n \log n)$ 。

1206 Boundary Points。虽然题目描述中指出给定点的坐标 $x, y \in \mathbb{R}$ ，但输出部分却未明确如何处理坐标的小数部分，不过 UVa OJ 上的评判数据似乎只包含坐标值为整数的点数据。

1209 Wordfish。该题在 UVa OJ 上的评测输入所给出的用户名可能是递变的第一个或者最末一个序列，此时仍然需要生成 21 个连续的递变序列（包括给定用户名在内），并按字典序排列。例如，输入可能是“ABCDEFJ”或者“JFEDCBA”。

1218 Perfect Service。根据题意，所给定的图为树，但任意结点不能同时被两个结点所支配，则在进行状态更新时，结点的第一种状态不能包含子结点的第二种状态，其他状态的更新仍与正常情形一致。注意对于根结点的处理，如果根结点包含多个子结点，且 $dp[u].selfOut < dp[u].in$ ，此时需要进一步进行检查，若 $dp[u].selfOut$ 包含两个（或以上）子结点的状态为 $dp[v].in$ ，则根结点会被两个（或以上）子结点所支配，不符合题意，此时根结点的状态应该选择 $dp[u].in$ 。

1221 Against Mammoths。人类星球在攻打外星人星球时不需要同时出发。

1222 Bribing FIPA。此题可以转化为 01 背包问题。在状态递推时有如下限制：树中的结点不能更新其子孙结点的状态，只能更新非子孙结点的状态。可以通过并查集来判定某个结点是否为当前结点的子孙结点。

1224 Tile Code。计数结果是数列 A090597 的一部分，而数列 A090597 可以使用 Jacobsthal 数（OEIS 序号为 A001045）来表示：当 n 为奇数时， $A(n-3) = (J(n-3) + J((n-3)/2))/2$ ，当 n 为偶数时， $A(n-3) = (J(n-3) + J(n/2))/2$ ， $n \geq 3$ ，其中 $A(n)$ 表示数列 A090597 的第 n 项， $J(n)$ 表示 Jacobsthal 数的第 n 项， $J(n) = J(n-1) + 2*J(n-2)$ ， $J(0) = 0$ ， $J(1) = 1$ 。长度为 n 时的不同瓦片铺设方案计数对应数列 A090597 的第 $n+1$ 项， $n \geq 3$ 。

1231 ACORN。由于并不需要确定松鼠搜集最多橡子的路径，故不需要考虑松鼠当前位于哪一棵橡树，这样在表示状态时能够省略一个维度，从而能够在限定时间内进行递推求解。

1233 USHER。此题实质上是求题目隐含给出的有向图中的最小权值圈。可以使用“顶点拆分”技巧将起始顶点拆分为两个顶点，之后使用 Moore-Dijkstra 算法计算最短路径来获得圈的最小权值。

1238 Free Parentheses。基本思路是使用回溯法确定所能得到的所有可能的不同和。恰当地定义状态是解题的关键，与此同时，使用备忘技巧可以降低时间复杂度以避免超时。要唯一的确定当前的状态，需要考虑“当前所处的位置”、“已经添加的左括号数量”、“当前数字和”这三个参数。根据题意可以得到以下结论：(1) 在表达式的任意位置可以（但并无必要）添加任意个左括号，因为最终可以在表达式的末尾添加对应的右括号使得括号达到平衡，从而使得表达式是合法的；(2) 在表达式中添加右括号时，需要考虑当前已经添加的左括号数量，显然，只有当未匹配的左括号数量大于零时才能添加右括号；(3) 只有在减号后加括号才会改变后续数字的正负性，从而改变原始表达式计算结果，而在加号后加括号是不会改变计算结果的；(4) 当前数字的正负性只与数字原有的符号以及在此数字之前添加的左括号数量有关。

1243 Polynomial-Time Reductions。将问题视为顶点，归约关系视为有向边，题目约束可转化为有向图 D ，如果若干问题在有向图 D 中构成有向圈，记该有向圈中的顶点数为 n ，则位于该有向圈中的问题至多需要 n 个归约关系，多余的归约关系可以予以简化。因此，先对有向图 D 进行“缩点”操作，构建新图 D' ，在此新图 D' 中，通过 Floyd-Warshall 算法确定新图中各顶点间的归约关系，对于能够通过中间顶点实现归约的两个顶点，则原有的归约关系可以简化，否则应予保留。需要注意，进行“缩点”操作后，新图的一个顶点可能对应原图中的一个有向圈，也可能对应原图中的单个顶点。如果对应的是单个顶点，在转换前后，需要累加的归约关系数量为零；如果对应的是一个有向圈，记该有向圈中的顶点数量为 n ，为了能够使该有向圈在新图中存在，新图中必须至少具有 n 个归约关系，因此需要将这 n 个归约关系累加到总的归约关系数量中去。

1244 Palindromic Paths。题目所要求的路 (path) 是指不包含重复顶点的迹 (trail)，可以结合使用备忘和松弛技巧解题。解题的关键是如何控制松弛的过程，使得最长路中不会出现重复的顶点。

1252 Twenty Questions。注意题目描述中的要求，并不是一开始就固定所需询问的问题，而是根据提问的前一个问题的结果来决定下一个提问，这两者之间是有差别的。

1254 Top 10。本题有多种解题方法。(1) 朴素的方法是先将字典内的所有单词按照题意的规则先排序，之后逐一读入待查询的单词 w ，检查 w 是否在字典单词中存在，直到获得前 10 个满足要求的单词序号。初看该方法可能会超时，但是由于题目的条件特殊，运行时间可以接受，且编码相对简单。(2) 使用后缀 Trie 并构建 AC 自动机予以解决，运行效率相对较高，编码难度也相对较高。(3) 构建后缀数组，结合线段树进行查询。先将给定字典中的单词 w 使用字符 ‘\$’ 进行连接，令连接得到的字符串为 T ，求出 T 的后缀数组，由于后缀数组有序，可以使用二分查找确定字典中每个单词 w 的排序。接着构建线段树，线段树的结点存储了每个区间前 10 个满足要求的单词序号。最后进行查询，先确定每个待查询单词 w 在后缀数组中的序号范围 (因为字典中可能有多个单词包含 w ，因此包含 w 的后缀可能有多个，因此需要确定一个序号范围)，接着结合线段树确定满足要求的前 10 个单词的序号。此方法编码难度较大。

1258 Nowhere Money。截至 2020 年 1 月 1 日，按照题目描述中指定的输出格式“*The second line is a series of slot sizes (in descending order) separated by spaces*”进行输出会导致 Presentation Error，如果在每组测试数据对应输出的第二行、第三行的每个项后面输出一个空格，可以获得 Accepted。

1280 Curvy Little Bottles。使用定积分计算体积，二分搜索得到刻度。

1309 Sudoku。由于搜索空间较大，使用位运算优化的回溯法求解会超时，需要使用更为高效的回溯搜索算法。由于本问题实际上能够转化为精确覆盖问题，可以使用舞蹈链 (Dancing Links) X 算法高效求解，该算法为《计算机程序设计艺术》的作者 Donald E. Knuth 所发明，具体细节请读者参考 Knuth 为该算法所撰写的论文。注意，舞蹈链 X 算法本质上仍属于回溯算法，它的优点在于利用舞蹈链这种巧妙的数据结构加速了回溯状态的更改和还原，还能够通过在矩阵中选择包含最少元素 1 的列来减小在回溯的某个层次时的搜索分支数，能够以较高的效率求解精确覆盖问题，因此可以用于高效地求解数独问题和 N 皇后问题。对于本题来说，解题的难点在于如何将问题约束转换为 01 矩阵。易知数独必须满足以下四个约束：(1) 每个方格必须填写 1 至 16 中的一个数字；(2) 每行必须填写 1 至 16 的数字一次且仅一次；(3) 每列必须填写 1 至 16 的数字一次且仅一次；(4) 每宫 (4×4 方格) 必须填写 1 至 16 的数字一次且仅一次。那么转换得到的矩阵需要包含 1024 列：(1) 第 0 列至第 255 列，如果某列 c 包含 1 则表示在第 $c/16$ 行第 $c \% 16$ 列的方格填写了某个数字；(2) 第 256 列至第 511 列，如果某列 c 包含 1 则表示在第 $(c-256)/16$ 行填写了数字 $(c-256) \% 16 + 1$ ；(3) 第 512 列至 767 列，如果某列 c 包含 1 则表示在第 $(c-512)/16$ 行填写了数字 $(c-512) \% 16 + 1$ ；(4) 第 768 列至第 1023 列，如果某列 c 包含 1 则表示在第 $(c-768)/16$ 宫填写了数字 $(c-768) \% 16 + 1$ 。

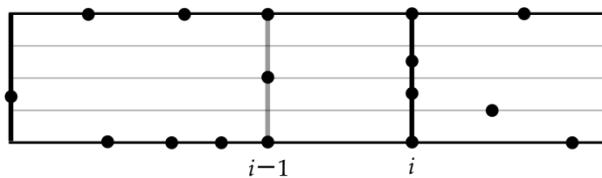
1349 Optimal Bus Route Design。对于图中的任意一个顶点 u 来说，它必有一个后继顶点 v ，反过来，如果有向图中每个顶点 u 都有对应的后继顶点 v ，则这些顶点必定构成一个圈。那么题目约束可以建模为二分图的最小权完备匹配问题，将顶点 i 拆分为两个顶点——“前点” x_i 和“后点” y_i ，如果顶点 u 和 v 之间有一条有向边 (u, v) ，则二部图中顶点 y_u 和 x_v 之间有边，求构造得到的二部图的最小权完备匹配，如果不存在完备匹配则表明符合题意要求的巴士路线不存在。注意，UVa OJ 上的测试数据中两个顶点的有向边可能会存在重复边的情形。

1378 A Funny Stone Game。题意要求从第 i 堆石子中取出一颗石子，然后在第 j 堆和第 k 堆石子中各自放入一颗石子， $i < j \leq k$ ，第 i 堆的石子数目必须大于 0，无法进行下一步操作的玩家输掉游戏。在 Nim 游戏中，从第 i 堆石子中取出一颗石子后，即予以丢弃，而在此游戏中，是在第 j 堆和第 k 堆中各自放入一颗石子，也就是说，一颗第 i 堆的石子，会转化为第 j 堆的一颗石子和第 k 堆的一颗石子，换句话说，从第 i 堆中取出一颗石子的游戏由以下两个放入石子的子游戏构成：子游戏 1——在第 j 堆中放入一颗石子；子游戏 2——在第 k 堆中放入一颗石子。那么可以将第 i 堆的一颗石子视为游戏的一个基本组成单元，令 $sg[i]$ 表示在第 i 堆的单个石子的 Grundy 值，则有

$$sg[i] = \text{mex}\{sg[j] \wedge sg[k], 0 \leq i < j \leq k < n\}$$

显然，当 $i=n-1$ 时，游戏无法继续转移，定义 $sg[n-1]=0$ 。使用动态规划并结合备忘技巧求得 sg 值（或者将石子堆的序号予以逆转，就能够通过递推的方法确定 sg 值）。由于取同在一堆的任意一个石子，其子游戏构成都是相同的，故同在一堆的石子其地位是等同的，也就是说，在同一堆中各个石子的 Grundy 值是相同的，若第 i 堆的石子数目为偶数，该堆内石子的 Grundy 值异或后必然为零，因此只需考虑拥有奇数个石子的石子堆的 Grundy 值，而具有奇数个石子的石子堆中各个石子的 Grundy 值的异或结果恰为该堆内单个石子的 Grundy 值，因此，整个游戏的 Grundy 值异或值等于所有具有奇数个石子的石子堆中单个石子的 Grundy 值的异或值。如果整个游戏的异或值不为 0，则枚举 i, j, k ，确定执行一步游戏操作使得整个游戏的 Grundy 异或值为 0 的位置。

1382 Distant Galaxy。如果在枚举上下边界的同时枚举左右边界，将导致 $O(n^4)$ 的算法，当 $n=100$ 时显然会超时，因此需要寻求 $O(n^3)$ 的算法。假定枚举上下边界，考虑在从左至右扫描右边界的过程中维护相应的信息以避免重复计算，从而提高效率。



如上图所示，令 $left[i]$ 表示边界 i 左侧（不包括边界 i ）位于上下界的星系数量， $on[i]$ 表示边界 i 上且包括上下边界的星系数量， $in[i]$ 表示边界 i 上但不包括上下边界的星系数量，则 $left[i]=7$ ， $on[i]=4$ ， $in[i]=2$ ，且有递推关系

$$left[i] = left[i-1] + on[i-1] - in[i-1], i \geq 1$$

固定上下边界，当扫描到右边界 i 时，最大边界星系数量 g 可以表示为

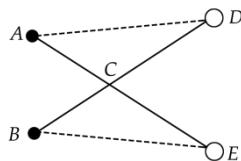
$$g = \max\{left[i] + on[i] - left[k] + in[k]\}, i \geq 1, k < i$$

通过在扫描右边界时维护 $-left[k] + in[k]$ 的最大值，可以得到局部最优值 g ，最后取 g 的最大值即为解。

1399 Puzzle。使用 Aho-Corasick 算法对所有模式建立转移图，其中与输出函数相关联的结点为禁止结点，问题转化为从初始状态出发，沿着转移图寻找一条不经过禁止结点的最长路径。如果转移图中出现圈则表明可以构造任意长度且不包含给定模式的字符串。注意，使用 Aho-Corasick 算法根据所有模式构建的转移图只是初步的有向图，从转移图中的某个状态出发，有些边并未指向其他状态，需要为这些边指定后续的转移状态，以便构成完整的状态转移图。之所以需要这样做是因为构建字符串时所使用的字符在给定模式中可能并不存在，如果不为转移图中的这些边指定后继状态很可能会出现“漏掉”某些解的情况，从而导致得到的解并不是最优解。由于转移图中状态数量可能较大，在动态规划中需要结合备忘技巧解题，否则容易超时。

1408 Count DePrimes。题目描述中的“sum of its prime factors”不包括重复的素因子，例如 $4=2^2$ ， $8=2^3$ ，按照题意，4 和 8 的“素因子和”均为 2，故均为“DePrime”。

1411 Ants。将蚂蚁视为 X 侧顶点，苹果树视为 Y 侧顶点，以蚂蚁和苹果树之间路线的长度（平面欧几里得距离）为权值，求最小权完备匹配，可以通过三角不等式（三角形的任意两条边边长之和大于第三条边的边长）证明，所得到的完备匹配中任意两条路线必定不相交。如下图所示，假设在最小权完备匹配中，路线发生相交，即蚂蚁 A 匹配苹果树 E，蚂蚁 B 匹配苹果树 D，则根据三角不等式，有 $AE + BD = (AC + CD) + (BC + CE) > AD + BE$ ，表明“蚂蚁 A 与苹果树 D 匹配，蚂蚁 B 与苹果树 E 匹配”能够获得更小的边权和，与当前匹配是最小权完备匹配产生矛盾，故假设错误，即以路线长度为权值所得到的最小权完备匹配中，路线必定不相交。



在 Cormen 等人所著的《算法导论（第二版）》中，位于第 33 章“计算几何”的思考题“33-3 Ghostbusters and ghosts”（中文版翻译为“魑魅和鬼问题”，在题目描述中又将“Ghostbuster”翻译为“巨人”，显然“魑魅”或“巨人”与英文“Ghostbuster”所要表达的含义不相符合，可以考虑直译为“猎鬼者”）与本题实质上是同一问题。假设平面上蚂蚁和苹果树不存在三点共线，可以证明，在 $O(n \log n)$ 的运行时间内，可以找到一条通过蚂蚁和苹果树的直线，使得直线一边的蚂蚁和同一边的苹果树数量相等。进一步地，可以在 $O(n^2 \log n)$ 的运行时间内，使其按不会有路线交叉的条件把蚂蚁和苹果树配对。具体方式是通过分治法予以解决。参考使用 Graham 法求凸包的过程，将所有点按照 Y 坐标值排序（若有两个点的 Y 坐标相同，则取 X 坐标较小的点），取排序后的第一个点作为参考点，将其他点按照相对于参考点的极角从小到大排序，设置两个变量 S_{ant} 和 S_{apple} ，分别表示当前类型为蚂蚁和苹果树的点的数量，初始时 $S_{\text{ant}} = S_{\text{apple}} = 0$ ，若参考点类型为蚂蚁则置 $S_{\text{ant}} = 1$ ，否则置 $S_{\text{apple}} = 1$ ，对极角排序后的点按照极角从小到大的顺序进行统计，若点类型为蚂蚁，则 S_{ant} 自增 1，否则 S_{apple} 自增 1，直到 $S_{\text{ant}} = S_{\text{apple}}$ ，那么将参考点和刚进入统计的点进行配对（此时两者的类型一定是相反的），并以参考点和刚进入统计的点的连线作为分界线，将点集划分为两个部分，这两部分中蚂蚁和苹果树的数量一定相等，继续递归解决划分得到的两个部分的配对问题。

1415 Gauss Prime。当 $a \neq 0$ 时，判断对应的范数 $a^2 + 2b^2$ 是否为有理素数，如果是则为高斯素数，当 $a = 0$ 时，显然对于任意的 $b > 0$ ，均不是高斯素数，因为 $b\sqrt{-2}$ 可以分解为 b 和 $\sqrt{-2}$ 的乘积。

1438 Asteroids。容易推知，多面体的重心与自身某个面的距离的最小值就是它与其他多面体的表面能够靠近的最近距离，因此只要求出两个多面体各自重心与自身表面的最小距离然后再求和即可。在计算多面体的重心时，可将二维情形下计算多边形重心的公式推广到三维情形（适用于任何已经将表面剖分为三角形的多面体，包括凹多面体）。首先计算给定点集的三维凸包，以三维坐标系原点 $O(0, 0, 0)$ 为参考点，令 A_i 为多面体的某个面，表示面 A_i 的三个点 a_i, b_i, c_i 按照逆时针方向排列，其法线指向多面体外侧， V_i 表示 A_i 与原点 O 构成的四面体的有向体积，则该四面体的重心 G_i 为

$$G_i = \frac{a_i + b_i + c_i}{4}$$

令多面体的有向体积为 V ，则整个多面体的重心 G 为

$$G = \frac{\sum_{i=1}^n G_i V_i}{\sum_{i=1}^n V_i} = \frac{G_1 V_1 + G_2 V_2 + \dots + G_n V_n}{V_1 + V_2 + \dots + V_n} = \frac{G_1 V_1 + G_2 V_2 + \dots + G_n V_n}{V}$$

1451 Average。由于本题要求具有最大平均值的区间的长度尽可能地短，因此在维护原有决策点的下凸性时需要考虑决策点共线的情形。如果决策点 j 、决策点 k 和待进入队列的决策点 i 共线，且 $j < k < i$ ，则需要删除决策点 k ；在寻找最优决策点时，如果决策点 j 、决策点 k 和待检查点 i 共线，且 $j < k < i$ ，则需要忽略决策点 j ，选择决策点 k 。

1456 Cellular Network。观察可知题目条件具有贪心选择的性质：对于概率较大的区域先处理，平均费用较少。设 $dp[i][j]$ 表示将前 i 个网格划分为 j 个区域时的最小平均费用，将网格内发现手机的概率 p_i 从大到小排序， ps 为排序后的概率前缀和数组，选择第 k 个网格作为分界，将前 $k-1$ 网格划分为 $j-1$ 个区域，将 k 至 i 的网格划分为一个区域，则递推关系式为：

$$dp[i][j] = \min\{dp[i][j], dp[k-1][j-1] + i * (ps[i] - ps[k-1])\}$$

其中 $1 \leq i \leq n$, $1 \leq j \leq w$, $1 \leq k \leq i$, 边界条件 $dp[0][0] = 0$ 。

1482 Playing With Stones。读者可以列出 $a_i \in [1, 100]$ 时的 Grundy 值，尝试发现规律。

1494 Qin Shi Huang's National Road System。首先得到最小生成树，然后枚举使用“魔法道路”连接的两个城市，如果两个城市之间的道路尚未进入最小生成树，则将此道路加入最小生成树会使得最小生成树出现圈，目标是找到此圈中除刚加入的道路以外的最长道路，必须预先进行处理以得到最小生成树中任意顶点对之间的最短路径上的最长边，否则容易超时。

1537 Picnic Planning。将公园所对应的顶点视为 0 号顶点，则题目所求为 0 号顶点的度不大于 s 的最小生成树，将单度限制最小生成树算法稍作修改即可用于解题（先求出最小生成森林，将 0 号顶点与最小生成森林的各个连通分量相连接，然后进行边交换，因为要求是 0 号顶点的度不大于 s ，那么只有当边交换所产生的增量为负时才进行交换，否则可以立即退出）。由于时间限制较为宽松且 n 较小 ($n \leq 20$)，亦可采用以下较为简单的方法解题：令公园为 0 号顶点，将与 0 号顶点有邻接边的顶点“挑选”出来构成顶点集 A ，由于最小生成树中与 0 号顶点相连的其他顶点必定是点集 A 的子集，于是可以逐个枚举 A 的子集 a （对于 $|a| > s$ 的子集不予考虑，在具体编码时可使用位掩码技巧来简化实现），使用 Kruskal 算法生成最小生成树，在合并边时，如果边的一端为 0 号顶点，另一端必须是子集 a 中的顶点时才予以考虑（有可能在此限制下无法得到最小生成树），求能够得到的最小生成树的最优值。

1561 Cycle Game。游戏的状态可由以下三个参数决定：当前所处的顶点，顶点左侧边的数值，顶点右侧边的数值。可以结合回溯法和备忘技巧解题。

1567 A Simple Stone Game。此问题可以视为 Fibonacci 博弈的扩展。当 $k=1$ 时, 若 n 不为 2 的幂, 则为 N 态局势, 否则为 P 态局势, 因为将 n 表示为二进制数后, 若 n 不为 2 的幂, 则先手可以选择取掉最右侧为 1 的位所对应的物品数量, 由于二进制数的高位 1 的权值至少是低位 1 的权值的 2 倍, 当取掉低位 1 所对应数量的物品后, 后手无法一次性取掉位于高位 1 所对应数量的物品, 使得先手总是能够取掉最后的物品。当 $k=2$ 时, 分析过程与 Fibonacci 博弈相同, 若 n 不为 Fibonacci 数, 则为 N 态局势, 否则为 P 态局势。当为 N 态局势时, 玩家 A 的必胜策略是取掉位于最右侧为 1 的位所对应权值数量的物品。从 $k=1$ 和 $k=2$ 的情形可以得到启发, 若能构造一种进制系统, 令该进制系统中各个数位上的权值从小到大依次为 $a[0], a[1], \dots, a[i], i \geq 0$, 在将 n 转换成该进制系统中的数时, 按照总是先取较大的 $a[i]$ 值的原则, 而且要求在转换得到的数中, 任意取两个“相邻”为 1 的位, 位于高位的 1 所对应的权值 $a[x]$ 和位于低位的 1 所对应的权值 $a[y]$ 满足 $a[x]/a[y] > k$, 那么可以证明: 如果 n 不为 $a[i]$ 中的任意一项, 则为 N 态局势, 否则为 P 态局势。若为 N 态局势, 则玩家 A 的获胜策略是取掉最右侧为 1 的数位对应权值数量的物品。以下简述 $a[i]$ 的构造方法。假设已经确定了 $a[0]$ 到 $a[i]$ 的值, 令 $b[i]$ 表示使用 $a[0]$ 到 $a[i]$ 的权值所能表示的最大整数, 则 $b[i]+1$ 由于无法使用 $a[0]$ 到 $a[i]$ 的权值予以表示, 所以在该进制系统中必定需要存在权值 $a[i+1]=b[i]+1$, 接下来, 确定使用 $a[0]$ 和 $a[i+1]$ 所能表示的最大整数, 由于需要满足选取的相邻两个权值之比大于 k , 则在选取比 $a[i+1]$ 小的权值时, 能够选取的权值 $a[y]$ 必须满足 $a[y] \times k < a[i+1]$, 令 y' 是满足约束的最大权值的序号, 则使用 $a[0]$ 至 $a[i+1]$ 所能表示的最大整数为 $b[i+1]=a[i+1]+b[y']$ (若不存在满足 $a[y] \times k < a[i+1]$ 的 $a[y]$, 则 $b[i+1]=a[i+1]$)。以 $k=3$ 为例构造该进制系统, 初始时 $a[0]=1, b[0]=1$ 。由于 $b[0]=1$, 故 $a[1]=b[0]+1=2$, 而满足 $a[y] \times 3 < a[1]$ 的 y 值不存在, 故 $b[1]=a[1]=2$; 由于 $b[1]=2$, 故 $a[2]=b[1]+1=3$, 而满足 $a[y] \times 3 < a[2]$ 的 y 值不存在, 故 $b[2]=a[2]=3$; 由于 $b[2]=3$, 故 $a[3]=b[2]+1=4$, 而满足 $a[y] \times 3 < a[3]$ 的最大 y 值是 0, 故 $b[3]=a[3]+b[0]=5$; 由于 $b[3]=5$, 故 $a[4]=b[3]+1=6$, 而满足 $a[y] \times 3 < a[4]$ 的最大 y 值是 0, 故 $b[4]=a[4]+b[0]=7$, 继续按此规律构造, 直到 $a[i] \geq n$ 。

1605 Building for UN。一种简单的符合题意要求的构造方法: 一共两层, 第一层 n 行 n 列, 每个国家一行, 第二层 n 行 n 列, 每个国家一列。

1642 Magical GCD。容易知道, 随着子序列的延长, 题目所求的 GCD 是不递增的。而且由于 GCD 必定是某个数的因子, 则对于给定的数 a 来说, a 和其他数的不同的 GCD 个数至多只有 $\log a$ 个。因此可以固定子序列的右侧端点 r , 枚举具有不同 GCD 的子序列的左侧端点 l , 确定能够获得的最大子序列 GCD, 再乘以该子序列的长度, 从而得到题目要求的积, 最后取最大积即可。总的时间复杂度可以优化到 $O(n \log A)$, 其中 A 为数列中最大的数。

1663 Purifying Machine。如果两块受感染的奶酪之间仅相差一个位, 则可使用一个操作将病毒清除, 因此目标是尽可能多地找到这样的受感染奶酪配对, 如果确定了受感染奶酪与自身的最大匹配数 M , 则 $M/2$ 即为能够使用一次操作清除两块受感染奶酪的模式数量, 但是由于对未感染的奶酪执行清除病毒操作会导致奶酪变质, 因此某些受感染的奶酪可能未与其他受感染奶酪匹配, 这些受感染奶酪需要使用不包含 ‘*’ 的模式进行匹配, 令这些受感染奶酪的数量为 K , 则总共所需的最少模式数量为 $M/2+K$ 。

1709 Amalgamated Artichokes。本题实质上可以归结为以下问题: 给定 n 个实数构成的数组 A , 对于 A 中第 i 个元素, 确定区间 $[i+1, n]$ 内元素的最小值。如果使用朴素的线性扫描方法, 时间复杂度为 $O(n^2)$, 如果使用线段树或者稀疏表, 时间复杂度可以优化为 $O(n \log n)$ 。令 $dp[i]$ 为第 i 个元素之后的最小元素的值, 则有递推关系: $dp[i]=\min(A[i+1], dp[i+1])$, 使用该递推关系式从后往前扫描一遍数组 A 即可得到第 i 个元素对应的 $dp[i]$, 时间复杂度为 $O(n)$ 。

1721 Window Manager。输入包含多组测试数据但无空行间隔。输出时, 多组测试数据的输出之间不需要打印空行。题意描述并未明确定义“矩形重叠”、“矩形包含某个点”的具体条件, 因此容易造成解题者困扰。经过测试, 使用以下代码来判断矩形是否重叠和获取包含某个点的矩形能够获得 Accepted。

```

// 记录矩形信息的结构体。
struct window { int x, y, w, h; };
// 屏幕的宽度和高度。
int SW, SH;
// 记录矩形信息的向量。
vector<window> windows;
// 判断左上角顶点坐标为 (x, y), 宽为 w, 高为 h 的矩形是否与已有矩形发生重叠。
bool overlapped(int wid, int x, int y, int w, int h) {
    for (int i = 0; i < windows.size(); i++) {
        // 当命令为 "RESIZE" 时, 忽略与矩形本身是否重叠的判断。
        if (i == wid) continue;
        int x1 = max(x, windows[i].x);
        int y1 = max(y, windows[i].y);
        int x2 = min(x + w, windows[i].x + windows[i].w);
        int y2 = min(y + h, windows[i].y + windows[i].h);
        if (x1 < x2 && y1 < y2) return true;
    }
    return false;
}
// 判断左上角顶点坐标为 (x, y), 宽为 w, 高为 h 的矩形是否超出屏幕范围。
bool out(int x, int y, int w, int h) { return x + w > SW || y + h > SH; }
// 获得包含点 (x, y) 的矩形的序号, 如果不存在, 返回-1。
int getId(int x, int y) {
    for (int i = 0; i < windows.size(); i++)
        if (windows[i].x <= x && x < windows[i].x + windows[i].w &&
            windows[i].y <= y && y < windows[i].y + windows[i].h)
            return i;
    return -1;
}

```

1723 Intervals。此题的难点在于如何将题目给定的约束条件转化为有向图中的边。设 $S[i]$ 是集合 Z 中小于等于 i 的元素个数, 即 $S[i] = |\{s \mid s \in Z, s \leq i\}|$ 。集合 Z 中范围在 $[a_i, b_i]$ 中的整数个数 $S[b_i] - S[a_i - 1]$ 至少为 c_i , 则有 $S[b_i] - S[a_i - 1] \geq c_i$, 转换成: $S[a_i - 1] - S[b_i] \leq -c_i$ 。根据 $S[i]$ 的定义, 还存在两个约束条件: $S[i] - S[i - 1] \leq 1$ 和 $S[i] - S[i - 1] \geq 0$ (即 $S[i - 1] - S[i] \leq 0$)。设所有区间右端点的最大值为 max , 所有区间左端点的最小值为 min , 最终要求的是 $S[max] - S[min - 1]$, 令 $M = S[max] - S[min - 1]$, 假设最终求得的各顶点到源点的最短距离长度保存在数组 $dist$ 中, 那么 $M = dist[max] - dist[min - 1]$ 。因为可以将集合 Z 指定为区间 $[min, max]$, 故必定有解。

1738 Ceiling Function。本题并不是判定树的同构问题。朴素的做法: 建立树结构, 选定一种树的遍历方式, 如果能对两棵树进行完全相同的遍历操作则两棵树的类型相同。

10003 Cutting Sticks。可以将木棒切割看做石子合并的反过程, 因此也是一种典型的区间型动态规划。令 $dp[i][j]$ 表示第 i 个切割点起始至第 j 个切割点结束这段木棒的最小切割费用, c_i 表示第 i 个切割点距离与木棒的一端的距离 (统一为木棒的左端), 并设 $c_0 = 0$, 易知有递推关系式

$$dp[i][j] = \min_{i \leq k < j} \{dp[i][k] + dp[k + 1][j]\} + c_j - c_{i-1}, \quad 1 \leq i < j \leq n$$

不难看出, 上述递推关系式中的代价函数 $w(i, j) = c_j - c_{i-1}$ 满足应用四边形不等式优化的条件。

10011 Where Can You Hide。题目所求是从家的位置出发，往任意方向直线前进时，在整个前进过程中都能够不被放射线损伤时能够前进的最大距离。正确理解题意的关键在于“最大安全旅行距离”需要建立在往任意方向前进的基础上，虽然可能往某个方向前进的距离可能比其他方向的距离都要大，但是依据题意，不能取最大的距离，而是应该取最小值。由于前进时不能穿越树本身，因此最多能够到达树的边界，因此“家与树的最大距离”定义为家和树的中心的距离减去树的半径。(1a) 如果放射源恰好位于树的边界，则切点为原点 O 本身，设过切点的直线为 l ，如果家和树在直线 l 的异侧，则家始终位于放射源的照射范围，因此“最大安全旅行距离”为 0.000；(1b) 如果家和树在直线 l 的同侧，则“最大安全旅行距离”为家与直线 l 的距离和“家与树的最大距离”的较小值。如果原点在树外，则过原点 O 可引两条直线与树相切，设切点为 p_1 和 p_2 ，则分三种情况：(2a) 家的位置在 $\angle p_1Op_2$ 内且在树后，此时放射源能够被树阻挡，“最大安全旅行距离”为家和两条切线的距离及“家与树的最大距离”三者的最小值（尽管有可能出现家和某条切线的最短距离所经过的线段与表示树的圆相交而不符合条件，但是此时“家与树的最大距离”以及家和另外一条切线的距离必定会更小，从而不影响最终结果）；(2b) 家的位置在 $\angle p_1Op_2$ 内且在树前，此时家能够被放射源照射，“最大安全旅行距离”为 0.000；(2c) 家的位置在 $\angle p_1Op_2$ 的外侧，此时“最大安全旅行距离”亦为 0.000。

10021 Cube in the Labirint。测试数据中，障碍的给出的顺序可能与示例数据不一致，即可能存在类似以下形式的测试数据，需要调整输入的处理方式以便正确地读取数据。

```
1
3 3
1 1
3 3
h
3 1
v
1 3
h
1 2
```

10022 Delta-wave。本题需要寻找规律。观察给定的图形，将数字 1 所在的三角形视为顶角在上方的三角形，简称为上三角形；将数字 3 所在的三角形视为顶角在下方的三角形，简称为下三角形。将数字 1 所在的三角形定为坐标系的原点，向右为 X 轴的正方形，向下为 Y 轴的正方向，为各个数字所在三角形确定坐标。确定坐标的方式很简单：观察给定的图形，第 1 行只有 1 个数字，第 2 行有 3 个数字，第 3 行有 5 个数字，…，第 n 行有 $2n-1$ 个数字，那么给定数字 m ，其 Y 轴坐标 y_m 为 $\text{ceil}(\sqrt{m})$ ，而其 X 轴坐标 x_m 为 $m - (y_m \times y_m - y_m + 1)$ 。给定两个数字 a 和 b ，可以确定其坐标 (x_a, y_a) 和 (x_b, y_b) ，并且能够确定数字 a 和 b 所在三角形的类型，即是上三角形还是下三角形（为了后续解题的方便，如果 a 大于 b ，可以交换 a 和 b ，保证 a 小于 b ，从而使得数字 a 所在的三角形总是位于数字 b 所在三角形的上方或者左方）。接下来，使用贪心算法得到两个三角形的距离：令当前所在三角形的坐标为 $(x_c, y_c) = (x_a, y_a)$ ，在移动过程中，目标是优先向 Y 轴方向移动，以使得当前坐标的 Y 轴分量 y_c 趋向于终点坐标的 Y 轴分量 y_b ，在此过程中，尽量使得当前坐标的 X 轴坐标分量 x_c 趋向于终点的 X 轴坐标分量 x_b 。如果当前位于一个下三角形，则只能向左或者向右移动一个位置，此时 Y 轴坐标不变，X 轴坐标增加 1 或者减少 1；如果当前位于一个上三角形，则可以直接向下移动一行，使得 Y 轴坐标增加 1，X 轴坐标不变。

```
pair<int, int> getCoordinate(int n) {
    int row = ceil(sqrt(n));
    int col = n - (row * row - row + 1);
    return make_pair(row, col);
}

int main(int argc, char *argv[]) {
```

```

int T; cin >> T;
while (T--) {
    int M, N;
    cin >> M >> N;
    if (M > N) swap(M, N);
    pair<int, int> c1 = getCoordinate(M), c2 = getCoordinate(N);
    int r = 0, up = (M % 2) != (c1.first % 2);
    while (c1.first < c2.first) {
        if (up) {
            if (c1.second < c2.second) c1.second += 1;
            else c1.second -= 1;
            up = 0;
        } else {
            up = 1;
            c1.first += 1;
        }
        r += 1;
    }
    r += abs(c1.second - c2.second);
    cout << r << '\n';
    if (T) cout << '\n';
}
return 0;
}

```

10023 Square Root。此题可以使用多种方法解决。截至 2020 年 1 月 1 日，如果使用 Java 的 BigInteger 结合二分搜索解题，由于 UVa OJ 上的评测数据可能存在问题，尽管算法正确，但不易获得 Accepted。评测数据中可能包含某些测试数据 y ，它们并不是完全平方数，对于这种情况，当使用二分搜索得到 x 后，如果 $x^2 > y$ ，将 $x-1$ 作为结果输出可以获得 Accepted。参阅：https://en.wikipedia.org/wiki/Methods_of_computing_square_roots, 2020。

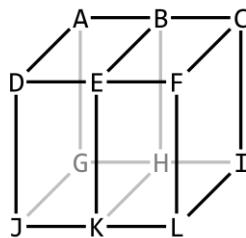
10029 Edit Step Ladders。将题目约束构建为有向无圈图，则题目所求即为该有向无圈图中的最长简单路径，应用动态规划结合备忘技巧可以求解。亦可将此题转化为最长递增子序列问题，然后使用动态规划解决。

10051 Tower of Cubes。与 10029 Edit Step Ladders 类似，将题目约束构建为有向无圈图，则题目所求即为该有向无圈图中的最长简单路径，应用动态规划结合备忘技巧可以求解。亦可将此题转化为最长递增子序列问题，然后使用动态规划解决。

10061 How Many Zero's and How Many Digits。 $n!$ 在十进制下的位数 $d_1 = \lfloor \log_{10}(n!) \rfloor + 1$ 。其中 floor 表示向上取整。类似的，在 B 进制下，其位数 $d_2 = \lfloor \log_B(n!) \rfloor + 1 = \lfloor \log_{10}(n!)/\log_{10}(B) \rfloor + 1$ 。在具体计算时，要注意精度误差（避免使用斯特林公式计算 $n!$ 的位数，因为其存在较大误差，会导致 Wrong Answer）。计算 B 进制下末尾 0 的个数可以参考十进制下末尾 0 的个数计算方式。将 10 分解为素数的乘积为 2×5 ，只要 $n!$ 的素因子分解式中出现一对 2×5 ，末尾就增加一个 0。类似的，将 B 进行素因子分解时，假设 $B = 98 = 2 \times 7^2$ ，那么只要 $n!$ 的素因子分解式中出现一对 2×7^2 ，则对应的九十八进制数末尾就多一个 0，也就是说，只需检查 $n!$ 中包含多少个 2×7^2 的素因子组合即可确定末尾 0 的个数。更进一步地，根据 $n!$ 的性质，令 B 的素因子分解中最大素因子为 q ，指数为 e ，令 $B = p \times q^e$ ，只需确定 $n!$ 的素因子分解式中 q 出现的次数 c ，则末尾 0 的个数 $t = c/e$ （作为练习，请读者思考可以这样做的原因）。

10072 Bob Laptop Woolmer and Eddie Desktop Barlow。此题既可以使用 01 背包问题的解题思路解决，也可以使用图论中的最小费用最大流算法予以解决。对于任意一名候选人员，可以选择其担任 batsman、bowler、all-rounder 三种角色之一，或者不选。需要注意，由于牵涉到小数的四舍五入，取整方法的不同可能导致无法获得 Accepted。以计算候选人员选为 batsman 为例，如果采用避免小数运算的取整方式，即使用 $(8 * bl[i] + 2 * fl[i] + 5) / 10$ 作为评分，会获得 Wrong Answer 的评判，而采用库函数 round 进行四舍五入，即使用 $round(0.8 * bl[i] + 0.2 * fl[i])$ 作为评分，则可以获得 Accepted，有理由推断生成评判答案的程序使用的是后一种取整方式。

10073 Constrained Exchange Sort。题目描述中 “ $l_1 \sim l_2 = d_i$ ” 的含义为 $abs(l_1 - l_2) = d_i$ ，即 $l_1 - l_2$ 的绝对值。初看似乎可以使用双向搜索予以解决，但是由于搜索空间较大，很难在限定时间内获得 Accepted。可将其转化为类似于三维形式的 15 数码问题，使用 IDA* 搜索加以解决。给定字符串 “ABCDEFGHIJKLM”，可将其转换为以下的三维形式（其中 ‘L’ 对应“空滑块”）：



10078 The Art Gallery。此题亦可先求给定点所构成的凸包，比较凸包的面积与原有点构成的多边形两者面积是否相等（或者只比较两者顶点数量是否相等），若相等则表明原有点构成的多边形是凸多边形，则内部不存在“关键点”，否则为凹多边形，存在“关键点”。

10091 The Valentine's Day。注意题目的约束条件：(1) 假定历史上第一个情人节的日期是 470 年 2 月 14 日；(2) 每到一个新的月份才从当前城市迁移到其他城市（或不迁移）。注意特殊日期的处理，例如，465 年 12 月 1 日（距离下一个情人节还有 50 个月），480 年 2 月 15 日（距离下一个情人节还有 12 个月），490 年 2 月 14 日（距离下一个情人节还有 12 个月），500 年 2 月 13 日（距离下一个情人节还有 0 个月）。

10097 The Color Game。此题与 899 Colour Circles 基本相同。

10109 Solving Systems of Linear Equations。由于要求输出精确解，需要使用分数。注意处理以下问题：(1) 在分数运算中，尽管题目描述中声明最终结果的分子和分母均不会超过 64 位整数，但是中间结果可能会超过 long long int 型数据的范围，因此为了尽量避免溢出，在计算时建议先除以各自的最大公约数以尽量减小分子和分母的大小，之后再进行乘运算。(2) 输入中可能包含非常规的分数输入，例如 ‘-6/1’，‘-3/-5’，‘0/-5’ 等。(3) 特殊情形的处理，例如方程数量小于未知数，或者方程数量大于未知数的数量。

10110 Light More Light。当电灯开关被按下奇数次时，电灯是亮的，若为偶数次，则电灯是灭的，换句话说，若某数的因子个数为奇数，则该数对应的电灯最终状态是亮的，若其因子个数为偶数，则该数对应的电灯最终状态是灭的，而只有完全平方数（此数为某个数的平方，例如 $9=3^2$, $121=11^2$ ）的因子个数为奇数，非完全平方数的因子个数为偶数。

10117 Nice Milk。使用回溯法确定所有可能的蘸牛奶方案，使用半平面交算法确定尚未蘸取牛奶的区域，取面积最小的方案，则在此方案下，已蘸取牛奶的区域面积最大。

10120 Gift?!。当 N 大于等于 49 时, 可以证明, 无论 M 为何值 ($2 \leq M \leq N$), 结论均为 “Let me try!”。对于 N 小于 49 的情形, 可以使用广度优先遍历来确定。证明参见: https://algorithmist.com/wiki/UVa_10120_-_Gift%3F!

10122 Mysterious Mountain。令最迟到达时间为 T , 以 T 为限制构建二部图, 二分搜索确定 T , 条件为二部图的最大匹配是否为 M 。

10160 Servicing Stations。该题的本质是求一般图的最小点支配数。由于求一般图的最小点支配数是 **NP 完全** 问题, 为了提高求解效率, 可以采用以下优化技巧: (1) 若原图可以拆分为多个不相连的子图, 则先予拆分, 然后对子图求最小支配集的顶点个数, 最后求和得到原图的相应结果; (2) 对于求两个集合的“并”采用位操作进行加速, 预先将某个顶点的邻接表表示为一个整数以便用“位与”操作来代替集合的并; (3) 枚举时先考虑顶点度较大的顶点。

10163 Storage Keepers。先使用二分搜索确定能够达到的最高安全底线 L , 之后使用 01 背包问题解题思路确定最小费用 Y 。

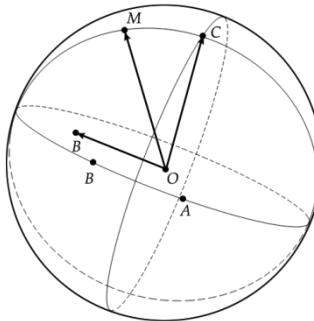
10169 Urn-Ball Probabilities。定义事件 A 为 “ n 次取球中至少有一次取球拿到的两颗球均为红色”, 事件 B_i 为 “第 i 次取球拿到的两颗球均为红色”, 则有 $P(A) = P(B_1 \cup \dots \cup B_n)$, 由于 B_i 和 B_j 属于相容事件, 需要应用容斥原理计算 $P(A)$, 即

$$P(A) = P(B_1 \cup \dots \cup B_n) = \sum_{r=1}^n (-1)^{r+1} \sum_{i_1 < \dots < i_r} P(B_{i_1} \dots B_{i_r})$$

由于题目条件中取球次数 $N < 1000000$, 当 N 较大时无法直接计算 $P(A)$, 因此需要从反向考虑, 计算事件 A 的补事件 A^c ——“ n 次取球中没有一次取球拿到的两颗球为红色”——的概率。第 i 次取球时两颗球都是红球的概率 $p_i = 1/(i \times (i+1))$, 则第 i 次取球时不都是红球的概率 $q_i = 1 - p_i$, 那么 $P(A^c) = q_1 \times q_2 \times \dots \times q_n$, 有 $P(A) = 1 - P(A^c)$ 。 n 次取球每次取出的两颗球均为红色的概率 $Q = p_1 \times p_2 \times \dots \times p_n$, 其中每一项 p_i 对小数点后 0 的个数的贡献为 $\log_{10}(p_i)$ 。

10173 Smallest Bounding Rectangle。可以证明: 对于任意凸多边形的最小面积外接矩形, 此矩形至少有一条边与凸多边形的一条边重合。因此, 可以先确定点集的凸包, 然后枚举凸包的每一条边 L 作为与外接矩形重合的一条边, 确定距离此边最远的一个顶点 p_f , 从而得到矩形的高, 然后再确定矩形的宽 (可先求出 p_f 在 L 上的投影点 p_f' , 再依次求出凸包顶点与直线 pp_f' 的距离, 取直线 pp_f' 两侧的最大距离和即为矩形的宽), 此方法的时间复杂度为 $O(n^2)$ 。对于凸包顶点数较少的测试数据, 此方法可以在时间限制内获得通过, 如果测试数据规模较大, 则需要寻求时间复杂度为 $O(n)$ 的旋转卡壳算法。

10184 Equidistance。令 Alice 所在地点为 A , Bob 所在地点为 B , 过 A 和 B 及球心 O 可作一大圆 C_1 , 在球面上与 A 和 B 具有相等球面距离的点构成另外一大圆 C_2 , 题目所求为确定会面地点 M 与大圆 C_2 的球面距离。以球心为坐标系原点, 根据经纬度可得到给定地点的三维坐标, 然后根据向量 OB 和向量 OM 的内积确定向量 OB 和向量 OM 之间的夹角 $\angle MOB$, 进而确定向量 OM 和过大圆 C_2 的三维平面间的夹角 $\angle COM = |\pi/2 - \angle MOB|$, 最终得到 M 和大圆 C_2 的球面距离。注意, 当 A 和 B 重叠时, 任意大圆均满足题目要求, 此时 M 距离大圆的距离为 0。



10200 Prime Time。注意输出精度的控制。

10207 The Unreal Tournament。结合备忘技巧解题。注意边界输入数据的处理，例如 $i=0, j=0$ 的情形。由于递归调用的次数可能非常大，因此需要应用高精度整数。

10217 A Dinner with Schwarzenegger。题目描述中指定在队列中的第一个人与售票者的生日进行比较，但未明确指明在队列中后续的人是否与售票者的生日进行比较。经过对不同情形进行计算，结合样例输入与输出的结果，推断出题者的本意应该是指位于第一名购票者之后的人其生日也应与售票者的生日进行比较。定义事件 A_i 为“队列中第 i 个人中奖”，事件 B_i 为“队列中前 $i-1$ 个人不中奖”，事件 C_i 为“队列中第 i 个人与队列中前 $i-1$ 个人加上售票者总共 i 个人中的至少一个人生日相同”，则事件 A_i 是事件 B_i 和事件 C_i 的交事件，即 $P(A_i)=P(B_i C_i)$ ，定义事件 D_i 为“队列中前 $i-1$ 个人加上售票者总共 i 个人中任意两个人的生日不同”，则事件 B_i 发生的概率等同于事件 D_i 发生的概率，根据生日问题的结论，有

$$\begin{aligned}
 P(A_1) &= \frac{1}{n} \\
 P(A_2) &= \frac{(n-1)}{n} \cdot \frac{2}{n} \\
 P(A_3) &= \frac{(n-1)}{n} \cdot \frac{(n-2)}{n} \cdot \frac{3}{n} \\
 &\dots = \dots \\
 P(A_i) &= \frac{(n-1)}{n} \cdot \frac{(n-2)}{n} \cdot \dots \cdot \frac{(n-i+1)}{n} \cdot \frac{i}{n} \\
 P(A_{i+1}) &= \frac{(n-1)}{n} \cdot \frac{(n-2)}{n} \cdot \dots \cdot \frac{(n-i+1)}{n} \cdot \frac{(n-i)}{n} \cdot \frac{i+1}{n}
 \end{aligned}$$

按照题意，题目所求为满足下列关系的最大 i 值：

$$\frac{P(A_i)}{P(A_{i+1})} \leq 1 \Leftrightarrow \frac{ni}{(n-i)(i+1)} \leq 1 \Leftrightarrow i^2 + i - n \leq 0$$

函数 $f(i)=i^2+i-n$ 的图像是一条开口向上的抛物线，抛物线位于 X 轴下方的曲线为方程的两个根之间所包围的部分，由于问题的实际意义要求 $i \geq 1$ ，则满足题意的最大 i 值为方程 $i^2+i-n=0$ 的正数根，即

$$i = \frac{-b + \sqrt{b^2 - 4ac}}{2a} = \frac{-1 + \sqrt{1 + 4n}}{2}$$

题目要求输出“最佳实数位置”和“最佳整数位置”，“最佳实数位置”即上式所得到的解，而“最佳整数位置”的定义却存在歧义，题目输出中并未予以明确。在某些情况下，“最佳整数位置”可能有两个，例如当 $n=30$ 时， $i=5$ 和 $i=6$ 都是能够获得最大中奖概率的整数位置，其中奖概率均为 $1/6$ ，若为了获得 Accepted，需要输出较大的 i 值。

10218 Let's Dance。题目要求将 C 颗糖果逐颗随机分给两组人，一组是男士共 M 人，一组是女士共 W 人，在全部糖果分配完毕后，从男士组的人手中回收糖果，要求男士组手中糖果的数量为偶数，以便能够将糖果再次平分给两组人，等同于确定男士组分到偶数颗糖果的概率。注意在分配糖果时是在所有人中随机分配，亦即单个人获得某颗糖果的概率是 $1/(M+W)$ 。

10219 Find the Ways。截至 2020 年 1 月 1 日，此题需要使用 `long double` 数据类型才能获得 Accepted。如果使用 `double` 数据类型会导致 Wrong Answer。

10227 Forest。题目描述中的“*How many different opinions are represented in the input?*”可能会引起误解。题目的本意是要求统计所有人听到树倒下的不同情形。如果有 4 个人，4 棵树，第 1 个人听到树 1 倒下，第 2 个人听到树 2 倒下，第 3 个人也听到树 2 倒下，第 4 个人没有听到树倒下，则不同意见数总共有 3 种，第一种意见是只听到树 1 倒下，第二种意见是只听到树 2 倒下，第三种意见是未听到树倒下。在评测输入中，某人听到树倒下的数据可能会重复给出，但最终只计入一次，即只是 `set` 而不是 `multiset` 之间异同的比较。

10249 The Grand Dinner。此题存在更为简单的贪心算法，但贪心算法的正确性似乎不容易证明。

10261 Ferry Loading。初看使用朴素的穷尽搜索法似乎不可行，因为本题车辆数量可达两百辆之多，指数级的计算时间会导致超时。不过 UVa OJ 上此题的评测数据相对较弱，使用回溯法可以勉强通过。使用回溯法解题的要点是：当车辆的长度小于某一车道的空余空间时，可以将车辆放置于该条车道，则继续回溯。不过使用回溯法解题一定要有效地记录已经访问过的状态，否则会超时，可以通过使用标准模板库中的 `set` 容器类来实现这个目标。

10294 Arif in Dhaka。对于项链 (necklace) 来说，仅需考虑旋转变换，具有 n 个珠子的项链共有 n 个旋转变换，即沿顺时针（或逆时针）方向旋转 $0, 1, 2, \dots, n-1$ 个珠子时构成的变换，旋转 i 个珠子后形成的变换所对应的置换的轮换数为 $\gcd(n, i)$ 。而对于手镯 (bracelet) 来说，不仅需要考虑旋转变换还需考虑翻转变换，在翻转变换中，又会由于给定珠子的数目而导致翻转变换的数目不同。令珠子的数目为 n ，如果 n 为奇数，由于翻转轴会经过每一颗珠子，则共有 n 种翻转方式，每种翻转方式对应的置换的轮换数为 $1+n/2$ ；如果 n 为偶数，翻转时有两种方式，一种是对称轴经过相对的两颗珠子，另外一种是对称轴经过相对的两条边，如果是前者，则对应置换的轮换数为 $2+(n-2)/2$ ，如果是后者，其对应置换的轮换数为 $n/2$ 。

10301 Rings and Glue。注意输出时单复数的差异。当结果为 0 时，需要输出“ $\dots 0$ rings.”，当结果为 1 时，需要输出“ $\dots 1$ ring.”，其他情形输出“ $\dots x$ rings.”。

10321 Polygon Intersection。两个任意多边形相交后，其顶点要么是两个多边形边的交点，要么是在多边形内部的点。

10323 Factorial You Must be Kidding。此题的评判数据在数学逻辑上并不成立，负数并无阶乘的定义。评判程序所使用的解题方法可参考：http://www.algorithmist.com/index.php/UVa_10323, 2020。

10331 The Flyover Construction。利用 Floyd-Warshall 算法确定所有点对间的最短距离信息，然后遍历所有边检查是否在最短路中使用的次数。即，令 $d[i][j]$ 表示顶点 i 和 j 之间的最短距离，对于权值为 w 的边 (u, v) ，检查 $d[i][u]+w+d[u][j]$ 是否与 $d[i][j]$ 相等，若相等，则表明该边是最短路径中的一条边。

10372 Leaps Tall Buildings。二分搜索角度，由角度计算速度，判断抛物线是否和建筑相交。由于浮点数运算误差以及根据角度计算速度的方式不同，尽管算法正确，但仍可能获得 Wrong Answer 的评判，可使用精度阈值控制误差。

10388 Snap。如果使用 C++ 语言解题，在存储玩家的牌堆时，既可以使用字符内置数组，也可以使用向量容器类，或者使用 `string` 类，由于一张牌只需一个字符表示，考虑到后续操作的便利性，使用 `string` 类较为合适。截至 2021 年 12 月 9 日，使用 C++ 语言，在代码中使用 `rand()` 或者 `random()` 生成随机数，提交评测均能获得 Accepted。

10392 Factoring Large Numbers。题目描述中未明确指定数据范围，只需预先筛出小于 10^7 的素数即可。对于 UVa OJ 上的评测数据，给定整数的素因子分解式中只有一个素因子会超过 10^7 。

10401 Injured Queen Problem。截至 2020 年 1 月 1 日，UVa OJ 上的评测输入与题目描述不严格相符，输入中可能包含空行，需要忽略这些空行才能获得通过。

10419 Sum-up the Primes。由于题目约束条件较松，此题可通过单纯的回溯法解决，使用备忘技巧对减少运行时间效果不佳。可以将问题转化为子集和问题进而使用动态规划解决。

10447 Sum-up the Primes (II)。由于此题的时间限制非常紧，需要进行适当地剪枝才能在时间限制内获得 Accepted。一个可行的剪枝技巧是利用奇偶性，假设不能使用素因子 2，则当给定的 N 为奇数但 t 为偶数，或者给定的 N 为偶数 t 为奇数，显然无解。类似于 10419 Sum-up the Primes，可以将问题转化为子集和问题进而使用动态规划解决。

10430 Dear GOD。解题关键是根据题意推导出 X 和 K 之间的关系式。在得到关系式后，既可以使用 C++ 解题，也可以使用后续介绍的 Java 中提供的 `BigInteger` 类来解题，使用后者更为简便。

10448 Unique World。由于从起点到终点的路径唯一，那么可以通过 BFS/DFS 确定从起点到终点的路径以及路径上边的边权，那么问题可以转化为：除了最后一条边只能走一次以外，其他的边都能走任意偶数次，是否可以通过这些边权组合得到指定的权值，实际上就是完全背包问题。

10490 Mr. Azad and His Son。给定正整数 n ，令其所有不同因子依次为 d_1, d_2, \dots, d_k ，如果 $d_1 + d_2 + \dots + d_k = 2n$ ，则称 n 为完美数 (perfect number，或称完全数)。例如，6 的所有不同因子为 1、2、3、6，而 $1+2+3+6=12=2\times 6$ ，故 6 是完美数。

10501 Simplified Shisen-Sho。截至 2020 年 1 月 1 日，此题的题目描述有两处文字错误：“As a side effect, this also means that in horizontal or diagonal.” 应为 “As a side effect, this also means that in horizontal or vertical.”；“Each tile in the board and (n,m) the lower left.” 应为 “Each tile in the board and (n,m) the lower right.”。由于此题的评测数据似乎较弱，不需使用剪枝技巧即可获得 Accepted。

10505 Montesco vs Capuleto。此题在 UVa OJ 上的测试数据中可能包含部分“非常规”的测试数据，即某个人的敌人列表中出现的编号会大于给定的 N 值。处理的方法是在构建图时将这些边忽略，因为题目描述中“暗示”只考虑给定的 N 个人，因此对于编号大于 N 的敌人不予考虑(编号大于 N 的敌人可能会导致受邀请者所在的连通子图无法构成二分图)。

10513 Bangladesh Sequences。此题实际上是八皇后问题的扩展和变形，但是由于评测数据时间限制很紧，如果逐组数据计算，即使应用位运算优化技巧，仍然不能在限定时间内获得 Accepted。根据题目所给定的条件，只有极少部分序列不是 Bangladesh Sequences，例如，取 $n=15$ ，每个位置均为“?”，则不是 Bangladesh Sequences 的序列总数为 32516。因此可以预先计算得到所有的非 Bangladesh Sequences，再根据评测数据所给定的条件进行筛选，这样可以显著减少运行时间，从而获得 Accepted。

10525 New to Bangladesh。此题在题目描述中未明确顶点的数量，取 256 即可。题目所求为具有最短时间的路径，若多条路径同时具有最短时间，则选择具有最短距离的路径，即时间最短是优先选择条件。此外，还需注意给定的两个顶点间可能存在平行边。解题可以使用 Moore-Dijkstra 算法，也可以使用后续介绍的 Floyd-Warshall 算法。若使用后者，在读入边数据时优先选择时间更短的平行边，若平行边的时间相同，则选择长度更短的平行边。

10526 Intellectual Property。在使用后缀数组获得所有公共子串后，需要对结果进行适当的处理，以达到排序和去除重叠子串的目的。

10529 Dumb Bones。设 $dp[x]$ 表示放置连续 x 块骨牌的期望步数。考虑放置第 i 块骨牌时的情形，此时第 i 块骨牌左侧的 j 块骨牌和右侧的 $i-1-j$ 块骨牌均已经放置完毕，对于在第 i 个位置放置骨牌，有三种可能：(1) 成功放置，期望步数增加 1；(2) 骨牌有 p_l 的概率向左边倒下，导致左侧的 j 块和第 i 块骨牌需要重新放置，则需要的期望步数为 $p_l \times (dp[i] - dp[i-1-j])$ ；(3) 骨牌有 p_r 的概率向右侧倒下，导致右侧的 $i-1-j$ 块和第 i 块骨牌需要重新放置，则需要的期望步数为 $p_r \times (dp[i] - dp[j])$ 。于是有

$$dp[i] = dp[j] + dp[i-1-j] + 1 + p_l \times (dp[i] - dp[i-1-j]) + p_r \times (dp[i] - dp[j])$$

化简可得递推关系式

$$dp[i] = \min \left\{ \frac{1 + dp[j] * (1 - p_r) + dp[i-1-j] * (1 - p_l)}{1 - p_l - p_r} \right\}, \quad 0 \leq j < i$$

边界条件 $dp[0]=0$ 。

10536 Game of Euler。假设放入棋盘的不是图钉而是长度为 1 至 3 个方格宽度为 1 个方格的木条，木条可以横向和纵向放置，且木条可以置入棋盘中任意连续的空格内，则此时状态转移规则将发生改变。更进一步地，如果木条可以沿着对角线放置，或者给定的不是木条，而是类似于俄罗斯方块的不规则木块，则状态转移规则又将不同。作为扩展练习，思考在上述条件下应该如何解题。

10541 Stripe。根据题意， N 为总的方格数，设 K 个编码占用了 M 个方格，则剩余 $N-M$ 个方格，分隔编码需要用去 $K-1$ 个方格，则最终剩余 $T=N-M-(K-1)$ 个方格，问题等价于将 T 个方格放置到 $K+1$ 个“间隙”中（包括首尾和编码中间）的不同放置方法数（亦即将 T 表示为 $K+1$ 个非负整数的和的不同方法数，此处的“不同”是指只要顺序不同即视为不同。例如，将 4 表示为 2 个非负整数的和，则 {0, 3} 和 {3, 0} 视为 2 种不同的方法，但 {2, 2} 只计为 1 种方法）。

10544 Numbering the Paths。注意，测试数据中可能包含重复的有向边，需要予以处理，否则会产生错误的结果或者导致超时。

10557 XYZZY。此题要求在有向图中确定是否存在一条从起点到终点的路径：沿着此路径行走时，能量处处为正。如果起点和终点不连通则肯定不存在这样的路径。若从起点到终点的路径上存在正权圈，则可以反复经过此正权圈使得能量值趋于无穷大，只要正权圈上的任意一个顶点和终点存在有向路径，则必定能够以正能量的状态到达终点。需要注意，在从起点到达正权圈中的任意一个顶点时，要求路径上的能量处处为正，而且经过正权圈时，能量也必须处处为正，若正权圈上某处能量不为正，则不符合题意的要求。

10592 Freedom Fighter。需要考虑边界测试数据的处理，例如网格中只包含字符‘.’，‘B’，‘P’，不包含字符‘*’的情形。

10604 Chemical Reaction。首先需要明确题目的子问题，然后确定如何有效地表示状态。截至 2020 年 1 月 1 日，此题目在 UVa OJ 上的测试数据与现实世界稍有差异。在现实世界中，物质 A 和 B 发生反应，在条件一定时，无论是往 A 中添加 B，还是往 B 中添加 A，其结果应该是一致的。但此题的测试数据中可能会出现这样一种情形：物质 A 和 B 反应顺序不同，其最终产物和释放的热量不同。因此需要考虑这样的测试数据才能获得 Accepted。可能是出题者在生成测试数据时并未加以斟酌，以致出现这样与现实世界不相符合的测试数据。

10606 Opening Doors。可参阅 10110 Light More Light。

10624 Super Number。此题在 UVa OJ 上的评测数据较弱，使用单纯的回溯法可以在限制时间内获得通过。似乎没有特别有效的剪枝方法来提高效率，可以采取的措施有：(1) 在进行模运算判断是否能够整除时，先使得被除数尽可能大，再一次性求模，这样可以减少求模运算的次数从而提高效率（具体来说就是只要被除数不超过 long long int 数据类型的表示范围，就暂不进行模运算）；(2) 预先生成每个数位可能包含的数字而不是从 0 到 9 枚举，因为除 1 以外，能够被 i 所整除的数其末位不会取遍 0 到 9。由于题目所给定的数据范围较小，也可以预先生成所有可能的解，然后使用查表的方式来获得 Accepted。

10625 GNU = GNU's Not Unix。需要注意输出中结果的数据范围要求“*The output will always fit in a 64-bit unsigned integer*”。

10643 Facing Problem With Trees。 $P = 1/2 \times (m!)/((m/2)! \times (m/2)!)$ 。此计数和“super ballot numbers”（OEIS 序号为 A001700）有关。当 m 较大时， P 超出 64 位整数表示范围，需要应用高精度整数。

10645 Menu。完全背包问题的变形。连续若干天烹饪同样的食物可以看做是一次性向背包中放入多个同样的物品。结合使用备忘技巧可以降低编码难度。需要注意的是，在记录最佳策略时，不仅需要记录选择的是哪种食物，而且需要记录连续烹饪的天数，这样在构建解时才能“有据可查”。

10646 What is the Card。截至 2020 年 1 月 1 日，该题在 UVa OJ 上的题意描述不够清晰，容易导致误解。对于一行 52 张牌的输入数据，第一张牌对应着底部的那张牌，最后一张牌对应着顶部那张牌。起始时，第一张牌在牌堆的最下方，最后一张牌在牌堆的最上方。先将牌堆上半部分的 25 张牌移走待用，对牌堆的下半部分 25 张牌进行 3 次指定的组合操作（这些操作会移走一部分牌），然后将最初移走的 25 张牌放到组合操作后剩下的牌堆顶部，从这个牌堆中的最底下一张牌数起，确定第 Y 张牌即为所求。

10648 Chocolate Box。令 $dp[i][j]$ 表示将 i 颗糖果放入 m 个盒子中且 j 个盒子有糖果的概率。那么可以得到递推关系式：

$$dp[i][j] = dp[i-1][j] \times \frac{j}{m} + dp[i-1][j-1] \times \frac{m-j+1}{m}$$

边界条件： $dp[1][1]=1$, $dp[1][0]=0$ 。

10650 Determinate Prime。注意输出的要求，如果某个给定区间不能容纳一个完整的连续等间隔素数序列，则不应输出。例如，给定区间“250 268”，在此区间连续等间隔素数序列“251 257 263”满足要求，但是“251 257 263”是更长的连续等间隔素数序列“251 257 263 269”的一部分，而区间“250 268”并不能容纳后者，故在此区间不存在满足输出要求的连续等间隔素数序列。

10666 The Eurocup is Here。根据题目所定义的传递性，给定序号 X ，设比 X 更好的队伍数量为 B ，比 X 更差的队伍数量为 W ，则 X 的 ranking 值最小不能小于 $B+1$ ，最大不能大于 2^N-W-1 。读者可以尝试将样例输入中的 X 转换成二进制数，结合样例输出进行观察以发现规律。

10669 Three Powers。读者可以按照题意，以子集和从小到大的顺序列出给定集合的子集，再结合整数 n 的二进制表示以获得解题思路。

10674 Tangents。需要根据两个圆的相对位置关系的不同分别处理。两个圆可能的相对位置关系有：重合、内含、内切、外切、相交、相离。

10677 Base Equality。截至 2020 年 1 月 1 日，此题的题目描述不够清晰使得通过率较低。以样例输入的第四组数据为例，其对应的样例输出为 9240_{10} ，将其转换为十一进制数为 $9240_{10}=6A40_{11}$ ，对应的十四进制数 $6A40_{14}=18480_{10}$ ，而 $18480_{10}/9240_{10}=2=c$ 。

10679 I Love Strings。由于此题评测数据量较大，使用朴素的字符串查找方法 (`string::find`) 理论上难以获得通过，需要使用高效的字符串匹配算法 (KMP 算法或者后续介绍的 Aho-Corasick 算法)。截至 2020 年 1 月 1 日，如果仅仅是为了获得 Accepted，仍可以利用评测数据存在的以下“缺陷”： T 要么为 S 的前缀，要么在 S 中不存在。

10686 SQF Problems。（1）单词是指由连续的（大写或小写）字母所构成的字符串。（2）问题描述中必须出现特定类别的至少 P 个不同关键词才能将其划入此类别，两个关键词相同则只能算出现一次。

10688 The Poor Giant。截至 2020 年 1 月 1 日，本题的描述仍存在细微错误，可能会造成解题者困扰。出题者的本意：使用一种最优的试吃苹果顺序策略，对于甜苹果所在位置的所有可能情形，确定所吃苹果的最小总重量。本题不要求确定具体的策略，而是要求确定最小总重量。对于 $n=4, k=0$ 的情形，最优的策略是先吃苹果#1，然后再吃苹果#3。根据题意，可以如下计算所需要吃掉苹果的总重量：（1）苹果#1 是甜的，需要吃掉的苹果重量为 1；（2）苹果#2 是甜的，在吃掉苹果#1 后并不能确定哪个苹果是甜的，还需要把苹果#3 吃掉，由于苹果#2 是甜的，按题意限制苹果#3 是酸的，吃掉苹果#3 后可反推出苹果#2 是甜的，因此总共需要吃掉的苹果重量为 4；（3）苹果#3 是甜的，需要吃掉苹果#1 后再吃掉苹果#3，总共需要吃掉的苹果重量为 4；（4）苹果#4 是甜的，吃掉苹果#1 之后需要吃掉苹果#3，由苹果#3 的味道是苦的可以推知苹果#4 是甜的，总共需要吃掉的苹果重量为 4。这样所有情形需要吃掉的苹果总重量为： $1+4+4+4=13$ ，比先吃苹果#2 再吃苹果#2 的策略所得到的总重量 14 要少。题目描述中的 $1+3+3+3=13$ 显然是错误的。

10698 Football Sort。在输出球队名称前，需要对其进行排序，样例输出中是按照忽略大小写的形式进行排序的，题目描述中未明确说明。

10707 2D-Nim。由于块 (piece) 在移除时必须满足连续且在同一行或同一列的限制，因此两个簇 (cluster) 所对应的图同构并不能保证两者能够通过平移、旋转、镜像的操作组合使之重合，因此本题并不是图的同构判定。也就是说，两个簇所对应的图同构只是两者能够通过操作达到重合的必要条件而不是充分条件。可以先应用 Flood-Fill 算法将原图拆分为不连通的簇，然后对簇进行坐标变换 (平移、旋转、镜像)，检查经过坐标变换的簇是否在目标图中存在。如果两个网格中的簇能构成一一对应(映射)，即对于左侧图中的每个簇，通过相应的操作之后都能与右侧图中与之对应的簇重合，则两个图是等同的。注意，由于是网格图，旋转操作只需考虑旋转 90 度的整数倍 (即只考虑旋转 90 度、180 度、270 度)，镜像操作也只需考虑左右镜像和上下镜像，因为只有这些操作才可能保证块的位置在进行变换操作后仍位于网格的格点上。

10722 Super Lucky Numbers。需要注意，题目描述中的 N 是指“数位”，而不是指“数字”。假设 $B=16$, $N=2$ ，以十进制数来表示数位上的数字，相邻数位以空格分隔，则“10 13”(AD_{16})是十六进制下的两位数，而不是十六进制下的四位数。按照题意，给定一个数，若某个数位为 1，紧接着的数位为 3，则该数不是“super lucky number”，因此“13”(13_{16})不是“super lucky number”，而“1 13”($1D_{16}$)是“super lucky number”。

10733 The Colored Cubes。对于正方体来说，共有 24 种面旋转变换，对应的置换群是 S_6 的一个子群，应用 Pólya 计数定理，可得不同的染色方案数为： $(n^6+3n^4+12n^3+8n^2)/24$ 。

10734 Triangle Partitioning。两个三角形相似，则两者对应的内角必定相等。

10746 Crime Wave The Sequel。由于涉及浮点数运算，截至 2020 年 1 月 1 日，对于 UVa OJ 的评测数据，需要对计算结果进行精度修正才能获得 Accepted。

10747 Maximum Subsequence。显然，在能够保证乘积为正的情况下，选择绝对值越大的数会使得最后的积更大，因此可将给定的数按绝对值递减的顺序排列，统计前 K 个数中正数、负数、零的个数，令其分别为 P 、 D 、 Z ，根据 P 、 D 、 Z 的值来分类讨论并进行贪心选择。当 $Z > 0$ 时，最大乘积为 0，则选择 N 个数中最大的 K 个数相乘可以得到最大的积且保证和最大；接着分别按 $Z=0$ 且 $D=0$, $Z=0$ 且 $P=0$, $Z=0$ 且 $P>0$ 且 $D>0$ 这三种情形分别进行最优选择，在具体选择时，需要考虑 K 的奇偶性和 D 的奇偶性。

10751 Chessboard。尽可能多地走对角线以增加路径的长度。

10802 Lex Smallest Drive。此题需要准确理解题目描述“A walk, W , from a to b is lexicographically smallest if there is no other walk from a to b in G that is smaller than W .”的含义，如果从起始顶点出发，可以反复通过图中的圈到达目标顶点，则认为不存在到达目标顶点的“walk”，读者可以结合样例输入中的第二组数据的输出进行理解。

10805 Cockroach Escape Networks。此题要求使用最少的边将所有顶点连接起来，则对应的边集合就是一棵最小生成树，因此题目可以归结为求无向图的最小直径生成树（minimum diameter spanning tree, MDST）问题。Hassin 和 Tamir 证明，求无向图的 MDST 等同于寻找图的绝对 1-中心（absolute 1-center），可以使用 Kariv-Hakimi 算法来确定图的绝对 1-中心，进而直接得出最小直径生成树所对应的直径。在本题的特定条件下，可以将边权认为是 1，进而可以使用下述更为直观的算法：由于图的绝对中心可能在顶点上也可能位于边上，对于图给出的所有边，以边的两个端点同时作为起点，使用 BFS 算法生成一棵最短路径树 T （设边的两个端点为 u 和 v ，初始时 u 和 v 已进入队列并置为已访问，即忽略此边，在最后 BFS 完毕时再将此边添加到以 u 和 v 为根的最短路径树，使得两棵最短路径树相连接而构成原图的一棵最小生成树），求 T 的直径，取最小值即为结果。因为对于连通图来说，通过上述方法生成的最短路径树必定是一棵最小生成树，而对所有边进行上述处理考虑了绝对中心的所有可能性（位于顶点或者位于边的中心），所以能够得到正确结果。如果仅仅是从每个顶点开始进行 BFS 来生成最短路径树，则由于未考虑图的绝对中心位于边的中心的可能性，会得到错误的结果。

10807 Prim Prim。此题要求从给定的边中找出两组互不重叠的边，使得每组边都构成一棵生成树且两棵生成树的权和最小。注意两棵生成树不一定都是最小生成树。该问题无特定算法，可考虑使用回溯法解决，命题者设定的数据规模也暗示了这一点。在具体编码实现时要注意回溯的控制，否则很容易超时。在回溯过程中，将两组边分别构造，同时运用适当的剪枝技巧进行优化，可以显著减少运行时间。

10817 Headmaster's Headache。可使用二进制数中的两个位来表示单个科目的状态，亦可考虑使用三进制数的一个位来表示单个科目的状态。

10843 Anne's Game。此题和 Cayley 公式（OEIS 序号为 A000272）有关：给定具有 n 个标号顶点的完全图，其生成树个数为 n^{n-2} 。

10859 Placing Lampposts。给定的无向图有可能为森林，即由多个不连通子图构成。

10876 Factory Robot。解题思路与 295 Fatman 类似。

10881 Piotr's Ants。两只蚂蚁在相遇后立即转向朝相反方向继续行走，可以视为蚂蚁“相互穿越”，左侧的蚂蚁变成了右侧的蚂蚁，右侧的蚂蚁变成了左侧的蚂蚁。不难推知，假设有 A, B, C, D 四只蚂蚁，从左到右依次排列并向不同方向前进，如果棒的长度无限，那么经过 T 秒后，棒上蚂蚁的顺序从左至右必然仍为 A, B, C, D 。也就是说，蚂蚁的排布顺序随着时间的改变是不会发生变化的，变化的是蚂蚁的位置和朝向。由于可以将蚂蚁相遇视为“相互穿越”，可以很容易得到 T 秒后各个蚂蚁的位置和朝向，将其按位置进行排序后，结合原有蚂蚁的排布顺序，可以确定最终各只蚂蚁的位置的朝向。需要注意，按照样例输出所示，蚂蚁最终恰好位于棒的边缘时不视为已经掉落。

10890 Maze。此题的题目描述中未明确说明进入某个包含宝物的方格是否必须要将其拾取，按照 Accepted 代码的预设，不需要一定将其拾取，而是可以忽略所经路径的某件宝物。

10907 Art Gallery。需要充分利用给定多边形只有一个“凹顶点”的条件。设 p_i, p_j, p_k 是多边形的三个连续的顶点，则可以根据 $p_i p_j p_k$ 是否构成一个“右转”来判断顶点 p_i 是否就是“凹顶点”。确定“凹顶点”后，继而可以确定给定的光源与“凹顶点”两侧的顶点所张角的位置关系，如果光源在此两个顶点张角的范围内，则光源能够照亮整个多边形内部，否则会有一侧的张角光线无法穿过。根据光源所在张角，可以使用半平面交算法求出不能被光照亮的区域。

10909 Lucky Number。首先确定区间[1, 2000000]内的所有幸运数，将其按升序存放在整数列表 L 中，给定 n ，如果 n 为奇数，则无法将其表示为两个幸运数之和（因为幸运数都是奇数，故两个幸运数之和为偶数），如果 n 为偶数，令 $x = \left\lceil \frac{n}{2} \right\rceil$ ，使用二分查找在 L 中找到一个最小的元素 $L[i]$ ，使得 $x \leq L[i]$ ，然后从序号 i 开始，暴力枚举 $L[i]$ ，检查 $n - L[i]$ 是否在 L 中存在，若存在，则 $L_1 = \min(L[i], n - L[i])$, $L_2 = \max(L[i], n - L[i])$ 。由于需要高效地筛选出所有幸运数，需要以下数据结构：给定一个整数列表，它能够高效地完成两种操作，一是定位列表中的第 k 个数，二是删除列表中的第 k 个数。可以使用多种数据结构完成上述任务，例如，线段树、树状数组等。

10911 Forming Quiz Teams。此题存在多种解题方法：(1) 使用回溯法构造所有可能的组合辅以适当剪枝解题；(2) 使用自顶向下的递归辅以记忆化搜索技巧解题；(3) 使用自底向上的递推辅以位掩码技巧解题。读者可以逐一尝试使用上述方法分别解题。

10918 Tri Tiling。此题是 10359 Tiling 的升级版，递推关系式不太直观，需要认真思考后才能得出。

10923 Seven Seas。敌方战舰的移动规则：向距离己方战舰平面欧几里得距离最小的方格移动。需要高效地表示战舰和礁石的位置状态，否则容易超时。

10934 Dropping Water Balloons。如果令 $dp[i][j]$ 表示“使用 i 个气球对 j 个楼层进行测试所需的最少实验次数”，则由于楼层的最大数量可达 2^{63} 级别，显然无法使用限定的内存予以表示，因此需要考虑其他的状态表示方法。如果令 $dp[i][j]$ 表示“使用 i 个气球进行 j 次试验所能判定的楼层数量（注意，不是楼层的高度）”，则对于给定的一个气球来说，任选一楼层 x 将其扔下，它要么破裂，要么不破裂，如果破裂，则其能够判定的楼层数量为 $dp[i-1][j-1]+1$ ，如果它不发生破裂，则还能够判定的楼层数为 $dp[i][j-1]$ ，则有递推关系式

$$dp[i][j] = dp[i-1][j-1] + 1 + dp[i][j-1], \quad i \geq 1, \quad j \geq 1$$

边界条件： $dp[i][0] = 0, i \geq 0$ 。寻找使得 $dp[k][j] \geq n$ 的最小 j 值即可。

10937 Blackbeard the Pirate。注意特殊情况的处理。如 ‘!’ 紧邻 ‘*’ 或者多个 ‘*’ 相邻时对周围方格的影响。

10958 How Many Solutions。将给定的等式进行适当变换：

$$\frac{m}{x} + \frac{n}{y} = \frac{1}{p} \rightarrow x = \frac{pmy}{y - pn} \rightarrow x = pm \frac{y}{y - pn} \rightarrow x = pm \left(1 + \frac{pn}{y - pn}\right) \rightarrow x = pm + \frac{p^2mn}{y - pn}$$

易知 pm 、 p^2mn 、 $y - pn$ 均为整数，若 x 有整数解，则要求 $y - pn$ 必须是 p^2mn 的约数。统计 p^2mn 的约数个数 d ，考虑到可以存在正、负约数且 $x = y = 0$ 不是合法解，则总的解个数为 $2d - 1$ 。

10987 Antifloyd。令 $d[i][j]$ 为顶点 i 和 j 之间的最短距离，如果最短距离不满足三角形不等式，则测量存在问题，否则枚举顶点 i 和顶点 j 之间的最短距离是否存在中间点，即给定顶点 k ，如果 $d[i][k] + d[k][j] = d[i][j]$ ，则顶点 k 可以作为中间点，说明顶点 i 和 j 之间的最短距离可以通过其他路径连接得到，因此顶点 i 和 j 之间可以不需直接边连接，否则需要直接边连接顶点 i 和 j ，其边权为 $d[i][j]$ 。

10997 Medals。在本题的测试数据中，名称为“Canada”的国家不一定会出现，如果在输入数据中，名称为“Canada”的国家未出现，应该输出“Canada cannot win.” 另外，若名称为“Canada”的国家使用适当的向量组合能够获得与其他国家同样的总分，则应该输出“Canada wins!”。只需尝试少量的具有代表性的向量组合即可，不需枚举所有可能的向量组合。

10982 Troublemakers。本题对应图论中的最大截问题 (max-cut problem)，最大截问题为 NP 难问题，目前尚未发现有效算法。有两种解题方法：(1) 随机算法。将学生随机分配到两个班中，最后检查分配方案是否符合要求，如果符合则予以输出。(2) 贪心算法。将学生逐个分配到两个班中，原则是与已分配到某班中的学生构成的配对越少就分配到哪个班，实际上，贪心算法是随机算法使用条件概率去随机化后得到的算法。需要注意，根据最大截定理，对于无向图 $G = (V, E)$ ，至少有 $|E|/2$ 条边是截，因此，在题目给定条件下，总是存在符合要求的分配方案。

参阅：https://en.wikipedia.org/wiki/Method_of_conditional_probabilities, 2020。

10983 Buy One Get the Rest Free。二分搜索解题，参阅：http://algorithmist.com/index.php/UVa_10983, 2020。

11012 Cosmic Cabbages。给定两个空间点 $A (x_i, y_i, z_i)$ 和 $B (x_j, y_j, z_j)$ ，按题意， A 和 B 两点间的最短距离为曼哈顿距离： $d = |x_i - x_j| + |y_i - y_j| + |z_i - z_j|$ 。由于 $x_i - x_j, y_i - y_j, z_i - z_j$ 的符号要么为正，要么为负，则总共只有 8 种情形，假设 $x_i - x_j$ 为负， $y_i - y_j, z_i - z_j$ 均为正，则 $d = (-x_i + y_i + z_i) + (x_j - y_j - z_j)$ ，那么只需要确定对于所有点来说， $-x_i + y_i + z_i$ 的最大值以及 $x_j - y_j - z_j$ 的最大值即可，将两个最大值相加即可得到此种情形下两点间曼哈顿距离的最大值。遍历所有 8 种可能的组合，则最终 8 种情形中的最大值即为解。可能 8 种情形的某些情形并不是符合实际的距离计算值，但是不会影响最终的解，因为最终的解一定包含在这 8 种情形之中，不符合实际情况的距离值会小于真实的距离最大值。

11013 Get Straight。本题的难点是如何简便地判断给定的牌组合能够获得多少赔付。可以将给定的牌组合转换为数值，然后进行全排列，对全排列直接按照给定的规则进行判定，这样可以避免考虑各种其他的情形，因为只要给定的牌能构成有赔付的组合，在全排列中就一定会出现。以下是确定赔付的参考代码：

```
// 数组 ns 存储给定的五张牌所对应的牌面值。
int ns[5];
// 判断给定的两张牌是否连续。
bool adjacent(int i) { return (ns[i] + 1) % 13 == ns[i + 1]; }
// 确定给定牌的赔付。
int getPayment()
{
    // 先将牌面值排序，然后进行全排列，对单个排列直接应用规则进行检查。
    sort(ns, ns + 5);
    int p = 0;
    do {
        if (adjacent(0) && adjacent(1) && adjacent(2) && adjacent(3)) p = max(p, 100);
        if (adjacent(0) && adjacent(1) && adjacent(2)) p = max(p, 10);
        if (adjacent(0) && adjacent(1) && adjacent(3)) p = max(p, 5);
        if (adjacent(0) && adjacent(1)) p = max(p, 3);
        if (adjacent(0) && adjacent(2)) p = max(p, 1);
    } while (next_permutation(ns, ns + 5));
    return p;
}
```

11019 Matrix Matcher。由于测试数据规模较大，限制时间较紧，如果使用 Aho-Corasick 算法解题，可能需要使用位掩码技巧来提高比对的效率。

11028 Sum of Product。此问题对应的数列又称为“dartboard sequence”(OEIS 序号为 A007773)。

参阅：<https://oeis.org/A007773/a007773.pdf>, 2020。

11038 How Many 0's。本题的核心是如何既不重复也不遗漏的计数所有 0 的个数。给定区间 $[m, n]$ ，直接计数该区间内数所包含的 0 个数存在困难，因此需要另辟蹊径。利用前缀和的思想，如果能够比较容易的计算区间 $[0, n]$ 的数包含的 0 个数 $f(n)$ ，那么题目所求即为 $f(n) - f(m - 1)$ 。令 $\overline{a_n a_{n-1} \cdots a_0}$ 表示整数 $a = a_n 10^n + a_{n-1} 10^{n-1} + \cdots + a_0$ ，其中 a_n, a_{n-1}, \dots, a_0 依次表示整数 a 的十进制表示中从左到右的数位值。令 $b = \overline{b_n b_{n-1} \cdots b_0}$ ，对于 $0 \leq k < n$ ，如果将数 b 的第 k 位置为 0，如果能够比较容易地计数第 k 位为 0 时共有多少个数，那么依次将各个数位置为 0，将满足要求的数的个数累加起来，即为从 0 写到 b 时，0 总共出现的次数。举个例子，如果 $n = 1982$ ，将最末一位置为 0，计数形式为 “###0” 的数的个数，可以容易知道，“###” 可以取 1 到 198，即从 0 写到 1982，总共有 198 个这样的数，最末一位是 0（如果加上 0 本身的话，则总共有 199 个数最末一位为 0）。如果将倒数第二位置为 0，即计数形式为 “##0?” 的数的个数，可以容易知道，“##” 可以取 1 到 19，共 19 种情况，“?” 可以取 0 到 9，共 10 种情况，根据乘法原理，共有 19×10 个数，满足倒数第二位为 0，也就是说，从 0 写到 1982，一共有 190 个数，倒数第二位是 0；依此类推……归纳总结一下，可以分两种情况进行统计：如果 $b_k > 0$ ，则形式为 $\overline{a_n a_{n-1} a_{k+1} 0 a_{k-1} \cdots a_0}$ 的数必定小于 b ，这样的数的个数为 $\overline{a_n a_{n-1} a_{k+1}} \cdot 10^k$ ；如果 $b_k = 0$ ，则形式为 $\overline{a_n a_{n-1} a_{k+1} 0 a_{k-1} \cdots a_0}$ 的数的个数为 $(\overline{a_n a_{n-1} a_{k+1}} - 1) \cdot 10^k + (\overline{a_{k-1} \cdots a_0} + 1)$ 。需要注意，最后的计数需要加上 0 本身，此种情况不会被前两种计数所覆盖。

```

long long getCount(long long n) {
    if (n == -1) return 0;
    string nn = to_string(n);
    long long ret = 0, pow = 1;
    for (int i = nn.length() - 1; i >= 1; i--) {
        if (nn[i] - '0') ret += stoll(nn.substr(0, i)) * pow;
        else {
            ret += (stoll(nn.substr(0, i)) - 1) * pow + 1;
            if (i < nn.length() - 1) ret += stoll(nn.substr(i + 1));
        }
        pow *= 10;
    }
    return ret + 1;
}
int main(int argc, char *argv[]) {
    long long m, n;
    while (cin >> m >> n) {
        if (m == -1) break;
        cout << getCount(n) - getCount(m - 1) << '\n';
    }
    return 0;
}

```

11055 Homogeneous Squares。如果给定的方阵是同质的（homogeneous），则将任意一行或者任意一列的每个方格中的数值同时增加（或减少）1，该方阵仍然是同质的。将方阵第一行和第一列的数值通过上述方法全部变为 0，再检查方阵中是否包含非 0 的方格，如果包含则肯定不是同质方阵。

参阅：<http://www.informatik.uni-ulm.de/acm/Locals/2006/html/judge.html>, 2020。

11084 Anagram Division。此题亦可通过回溯法（或者使用标准类库中的全排列函数 `next_permutation`）构建所有排列来进行解题（尽管效率上比动态规划算法差一些，但可在规定时间内获得 Accepted）。

11086 Composite Prime。截至 2020 年 1 月 1 日，此题的题目描述不够清晰，容易使得解题者产生困惑。题目中最后所给定的约束 “ $N \leq 2^{20}$ ”，其含义不是指一组测试数据所包含整数的最大个数，而是指测试数据中整数的取值范围，即给定一组测试数据，此组测试数据中所包含的整数均不大于 2^{20} 。

11088 End up with More Teams。此题和 10911 Forming Quiz Teams 类似。

11090 Going in Cycle。此题实质上是最小平均权值圈问题，对于该问题存在有效算法——Karp 最小平均权值圈算法，其时间复杂度为 $O(VE)$ 。本题亦可使用时间复杂度为 $O(VE\log V)$ 的二分搜索算法予以解决，即将所有边权减去一个常数值 c ，然后使用 Bellman-Ford 算法检测是否存在负权圈，如果存在负权圈，表明常数 c 大于图中圈的最小平均权值，若不存在负权圈，则表明常数 c 小于或等于圈的最小平均权值，不断缩小搜索范围直到常数 c 满足精度要求，此时的常数 c 即可认为是圈的最小平均权值。二分搜索算法的正确性容易理解，因为将每条边的权值均减去 c ，则相当于将所有圈的平均权值减小 c ，只要 c 刚好使得图中不存在负权圈，那么 c 就是最小平均权值圈所对应的平均权值。

11125 Arrange Some Marbles。此题的测试数据组数较多，需要优化状态表示，否则容易超时。可以应用状态压缩技巧以提高效率。注意边界测试数据的处理，例如测试数据“3020”。

11163 Jaguar King。解题的关键是根据题目的约束条件寻找合适的启发式信息估算函数，使得能够较为准确地确定中间状态到目标状态所需的最少步骤数。观察题目所给定的位置变换规则，可以将其转换为与 15 数码问题类似的问题予以解决。由于此题所求的是达到目标状态的最少步骤数，在设置深度限制时，每迭代一次，深度限制增加 1，而不能像 15 数码问题那样，每次迭代后深度限制可能增加不止 1。因为 15 数码问题所求的不是最优解，因此可以在每次迭代时对深度限制的增量放宽。若 15 数码问题所求的是最短走法步骤，则每次迭代后深度限制的增量也需要限制为 1。

11202 The Least Possible Effort。需要考虑棋盘的横向和纵向对称性，对于方形棋盘还需进行特殊处理。

11218 KTV。由此题亦可使用简单的三重循环来构建所有可能的组合情形而不必求助于动态规划，如果组数较多，则使用回溯法或动态规划解题较为适宜。

11240 Antimonotonicity。本题所求序列实际上为抖动序列。对于本题的评测数据规模，如果使用动态规划算法解决，需要结合使用能够高效进行 RMQ 查询的数据结构（如线段树）才能在限定时间内通过。另外，本题也存在非常巧妙的贪心算法，其关键是寻找给定的序列所构成“凸峰”的个数，如果 $a_i < a_{i+1}$ 且 $a_{i+1} > a_{i+2}$ ，则可以称之为构成了一个“凸峰”，最长抖动序列的长度和凸峰的个数密切相关。

11249 Game。首先找出 P 态局势的规律，然后通过预处理得到所有的 P 态局势，查表输出，否则容易超时。

11259 Coin Changing Again。

11267 The Hire-a-Coder Business Model。对于给定的连通图，检查是否可以二着色，若能够二着色则求图的最小生成树。注意，由于求的是损失最小的配对方案，如果给定的某条边未进入生成树，但其边权为负值，则应将其加入，因为这样可以使得损失更小。

11270 Tiling Dominoes。棋盘完美覆盖计数，参阅 OEIS 序号为 A099390、A004003 的数列。在数列 A004003 的页面列出了多篇有关完美覆盖计数的论文，读者可以进一步查阅。本题也可通过集合型动态规划予以解决。

11279 Keyboard Comparison。空格的键入移动距离不需计入统计结果。

11285 Exchange Rates。维护每天通过兑换能够得到的最大加元和美元值。注意浮点数的截断处理。

11297 Census。截至 2020 年 1 月 1 日，经过 `assert` 语句测试，UVa OJ 上的评测数据存在如下“缺陷”：在查询语句‘`q x_1 y_1 x_2 y_2` ’中，会出现 $x_2 > n$ 和（或） $y_2 > n$ 的测试数据，即查询范围超出指定数据范围的情形。可将其调整到给定的数据范围内，不影响 Accepted。

11300 Spreading the Wealth。解题关键在于找到最小硬币交换数量的数学表达式并从中观察得出与中位数的联系。从 0 开始计数，令第 i 个人拥有的硬币数量为 c_i , $0 \leq i < n$, 硬币数量的均值为 M , 依据题意，第 i 个人会将若干硬币传递给左侧和右侧的人，假设第 i 个人传递给右侧的第 $(i+1)\%n$ 个人 a 枚硬币，第 $(i+1)\%n$ 个人传递给左侧的第 i 个人 b 枚硬币，则最终的效果等价于第 i 个人传递给第 $(i+1)\%n$ 个人 $a - b$ 枚硬币。因此，为了简化问题的处理，规定第 i 个人只向右侧的第 $(i+1)\%n$ 个人传递 x_i 枚硬币，如果 x_i 为正，等价于第 i 个人给第 $(i+1)\%n$ 个人 $|x_i|$ 枚硬币，否则等价于第 $(i+1)\%n$ 个人给第 i 个人 $|x_i|$ 枚硬币。那么，有

$$\begin{aligned} c_1 + x_0 - x_1 &= M \\ c_2 + x_1 - x_2 &= M \\ &\dots \\ c_{n-1} + x_{n-2} - x_{n-1} &= M \end{aligned}$$

其通项公式为

$$c_i + x_{(i-1+n)\%n} - x_i = M, \quad 1 \leq i \leq n-1$$

根据上述等式，可以使用 x_0 来表示 x_1, x_2, \dots, x_{n-1} ，即

$$\begin{aligned} x_1 &= c_1 + x_0 - M = x_0 - (M - c_1) \\ x_2 &= c_2 + x_1 - M = c_1 + c_2 + x_0 - 2 \times M = x_0 - (2 \times M - c_1 - c_2) \\ &\dots \\ x_{n-1} &= c_1 + c_2 + \dots + c_{n-1} + x_0 - (n-1) \times M = x_0 - ((n-1) \times M - c_1 - c_2 - \dots - c_{n-1}) \end{aligned}$$

通项公式为

$$x_i = x_0 - [i \times M - (c_1 + c_2 + \dots + c_i)], \quad 1 \leq i \leq n-1$$

令

$$T = |x_0| + |x_1| + |x_2| + \dots + |x_{n-2}| + |x_{n-1}|$$

则

$$T = |x_0| + |x_0 - (M - c_1)| + \dots + |x_0 - ((n-1) \times M - c_1 - c_2 - \dots - c_{n-1})|$$

那么题目所求即为 T 的最小值，观察 T 的表达式，由中位数的性质不难得知， T 的最小值当 x_0 为

$0, M - c_1, 2 \times M - c_1 - c_2, 3 \times M - c_1 - c_2 - c_3, \dots, (n-1) \times M - c_1 - c_2 - \dots - c_{n-1}$ 的中位数时取得。注意数据类型的使用以及当 n 为 0 时的处理。

11307 Alternative Arborescence。此题为树的着色和问题，可以使用动态规划解决。存在 $O(n)$ 的算法，具体方法请读者参阅给出的参考资料。

11311 Exclusively Edible。此题亦可转化为 Nim 游戏加以解决。

11314 Hardly Hard。根据对称原理，将点 A 作关于 X 轴的对称点 A' ，将点 B 作关于 Y 轴的对称点 B' ，然后再根据平面上两点之间直线距离最短的原理，易知四边形的最短周长为线段 AB 和线段 $A'B'$ 的长度之和。

11346 Probability。注意边界输入数据的处理。

11391 Blobs in the Board。可以从 10651 Pebble Solitaire 的解题过程获得启发。利用备忘技巧可以提高程序运行效率。

11393 Tri-Isomorphism。观察当 n 较小时的结果，归纳总结以便发现规律。

11395 Sigma Function。读者可以尝试列出 100 以内因子和为奇数的正整数，观察以发现规律。

11405 Can U Win。题目描述要求确定以下目标是否能够实现：使用己方的马（Knight）在指定的 n 步内将敌方的兵（Pawn）全部吃掉，而且在此过程中不能攻击或者占据其他非兵棋子的位置。题目描述并未要求马在吃掉全部敌方兵后回到原位。

11406 Best Trap。题目中所给定的“房间的大小为 N ”的条件似乎并未使用，如果考虑这个约束反而会得到“Wrong Answer”的评判结果（考虑到捕蚊圈为刚性物体，过于靠近房间的边界会导致捕蚊圈被房间的墙阻挡而无法放置，毕竟从现实角度来说，捕蚊圈无法“穿墙”）。注意当房间内的蚊子数量少于 3 只时的处理。

11423 Cache Simulator。解题思路是高效地模拟缓存的 LRU 机制。令缓存的大小为 s ，假设 CPU 在访问数据 x 后，要求在下一次访问数据 x 时不发生缺失（miss），那么在此期间，CPU 所访问的其他不同数据个数必须小于缓存的大小 s ，如果大于或等于缓存的大小 s ，则根据缓存的 LRU 机制，需要将前一次的数据 x 从缓存中移除以容纳新的数据。因此，我们可以将 CPU 的所有访问请求排成一个队列，逐次检查每个数据在访问时是否会发生缺失。假设 CPU 在第 t_2 次请求时访问数据 x ，要判定是否发生缺失，只需要找到比第 t_2 次访问早一次访问数据 x 的第 t_1 次访问，检查在第 t_1 次访问和第 t_2 次访问之间 CPU 访问的其他不同数据的个数 y ，如果 y 大于或等于缓存的大小 s ，则会发生缺失。可以利用树状数组（抑或线段树）来予以实现。具体做法是为每次访问标记三个信息，一个是访问的数据 x ，另外一个是当前的访问请求序号 c ，再一个是前一次访问数据 x 的访问请求序号 p 。设立一个辅助数组 f ，初始时数组 f 的全部元素值均为 0，对于某次访问 a ，首先将 $f[a.c]$ 置为 1，接着按 $a.p$ 的值分别处理：（1）如果 $a.p$ 为 0，表示是第一次访问数据 x ，因此必定发生缺失，将缺失计数加 1；（2）如果 $a.p$ 大于 0，则确定数组 f 中闭区间 $[a.p, a.c]$ 内值为 1 的元素个数 y ，如果 y 大于或等于缓存的大小 s ，则会发生缺失，将缺失计数加 1，然后将 $f[a.p]$ 置为 0。对于“将 $f[a.c]$ 置为 1”、“将 $f[a.p]$ 置为 0”、“确定数组 f 中闭区间 $[a.p, a.c]$ 内值为 1 的元素个数 y ”等操作，使用树状数组完成即可。需要注意，在每次输出缺失的统计信息后，需要将缺失计数器全部重置，从 0 开始重新计数。

11432 Busy Programmer。此题的时间限制较为严格，需要优化状态表示和初始化环节，否则容易超时。由于题目输入范围不大，亦可预生成所有结果，使用“打表”的方式进行提交。

11439 Maximizing the ICPC。令所有比赛中最低的 ICPC 为 x ，只在比赛双方 ICPC 大于等于 x 的摔跤运动员之间建立无向边，使用一般图最大匹配算法求最大匹配数，若最大匹配数 M 不小于 2^{N-1} 则表明 x 可行，二分搜索 x 的最大值即可。

11451 Water Restrictions。题目的描述并不十分清晰，当某个喷洒器的流量为 0 时，它所浇灌的地块数目为 0。只有当流量不为 0 时，喷洒器所浇灌的地块包含本身所在的地块。

11456 Trainsorting。由于题目要求最后的车厢序列必须是按照重量从大到小的顺序排列，那么考虑符合题目要求的最长车厢序列中第一个进入的车厢为第 i 节，那么很显然，在第 i 节车厢进入序列之后，接着进入序列并位于第 i 节车厢左侧的肯定都是重量大于第 i 节的车厢，位于第 i 节车厢右侧的肯定都是重量小于第 i 节的车厢，令 $weight[i]$ 表示第 i 节车厢的重量， $dp_1[i]$ 表示第 i 节车厢为起始的最长递增子序列， $dp_2[i]$ 表示以第 i 节车厢为起始的最长递减子序列，显然题目所求为

$$M = \max\{dp_1[i] + dp_2[i] - 1\}, \quad 1 \leq i \leq n$$

可以从序列的最后一个元素开始，从后往前进行动态规划，即

$$dp_1[i] = \max\{dp_1[i], dp_1[j] + 1\}, \quad i < j, \quad weight[i] < weight[j]$$

$$dp_2[i] = \max\{dp_2[i], dp_2[j] + 1\}, i < j, weight[i] > weight[j]$$

由于题目约束所有车厢的重量均不相同，除了序列的第一节车厢以外，递增序列和递减序列的其他车厢不会发生重叠的情形。除了使用上述方法解题之外，还可以将问题转化为求最长递增子序列问题。假设给定序列为“1234”，可将序列反向与原序列合并构成回文序列，即构造“43211234”，然后求此序列的最长递增子序列即可。

11466 Largest Prime Divisor。(1) 输入中可能包含负数，需要将其转换为正数进行处理。(2) 题意要求 N 至少要有两个不同的素因子，否则输出“-1”，则对于 $N=32=2^6$ ，由于只包含 2 这个唯一素因子，应输出“-1”，类似的，如果 N 为素数，也同样需要输出“-1”。

11468 Substring。使用 Aho-Corasick 算法对所有模式建立转移图，则问题转化为有向图上的动态规划问题。

11471 Arrange the Tiles。将同类型的滑块合并可以减小搜索空间，从而能够更快地得到解，否则容易超时。

11478 Halum。将题目给定的约束条件转化为差分约束系统，使用二分搜索予以解决。由于操作是可以分开进行的，令 $sum[u]$ 表示对顶点 u 的操作所累积的 d 值， $weight[u][v]$ 表示顶点 u 和 v 之间有向边的权值，如果顶点 a 和 b 之间存在有向边 (a, b) ，令经过调整后最后能得到的边权最小值为 x ，则有

$$weight[a][b] + sum[a] - sum[b] \geq x$$

亦即

$$sum[b] - sum[a] \leq weight[a][b] - x$$

可知其为差分约束形式，二分搜索 x 的最大值即可。对于特定的值 x ，如果转换得到新图中存在负权值圈则表明不可行，否则表示最终经过调整后所有边权均能大于等于 x 。

11487 Gathering Food。此题的题目描述有两处容易导致误解：(1) 每组测试输入的网格大小为 $N \times N$ ，但并不表明其中包含的表示食物的字母也只有 N 个；(2) 当 Yogi 站在有食物的方格上时，它必须将食物拾取，也就是说，为了完成按序拾取食物的目的，在中途它要避开某些不是当前需要拾取的食物方格，例如有食物 A、B、C，在拾取 A 后，继续拾取 B，在此过程中不能经过包含 C 的方格，否则会先拾取 C，不符合要求。此题将无向图的最短路径问题和路径计数问题予以综合，有一定难度。

11511 Frieze Patterns。根据题目给定的约束关系，逐列计算每列的元素，找到循环节，然后确定指定位置的值。在寻找循环节时，只有整列元素重复出现才视为是一个循环，单个元素重复出现并不能认为是出现了循环。实际上，令模式的行数为 n ，循环节的长度为 $n+1$ 。

11514 Batman。超人的每种超能力只能使用一次。

11526 H(n)。选择较小的 n ，例如 100，列出它的除数和商，观察规律。注意当输入为负数时的处理。

11545 Avoiding Jungle in the Dark。结合备忘技巧解题。使用三个参数来表示状态：当前所处位置 p ，已经行走的小时数 h ，当前时间 t 。为了便于判断是否满足题目的约束条件，在动态规划时可以每次只向右侧走一格。

11552 Fewest Flops。每 k 个字符构成的区块 (chunk) 可以视为一个区间，在计数最小区块时，后一个区间只与前一个区间的最后一个字符有关联，因此只需两个参数来表示状态。令 $dp[i][j]$ 表示前 i 个区间以字符为 j 结尾时的最小区块数，根据题目约束可以得到相应的递推关系，使用自底向上的递推方式解题即可。

11574 Colliding Traffic。设两艘船 B_1 和 B_2 之间的距离为 D ，随着时间 t 的变化， B_1 的位置为 $(x_1 + s_1 \times t \times \sin(d_1), y_1 + s_1 \times t \times \cos(d_1))$ ， B_2 的位置为 $(x_2 + s_2 \times t \times \sin(d_2), y_2 + s_2 \times t \times \cos(d_2))$ ，根据两点间距离公式可得 $D^2 = at^2 + bt + c$ ，其中 $a = (s_1 \times \sin(d_1) - s_2 \times \sin(d_2))^2 + (s_1 \times \cos(d_1) - s_2 \times \cos(d_2))^2$ ， $b = 2 \times ((s_1 \times \sin(d_1) - s_2 \times \sin(d_2)) \times (x_1 - x_2) + (s_1 \times \cos(d_1) - s_2 \times \cos(d_2)) \times (y_1 - y_2))$ ， $c = (x_1 - x_2)^2 + (y_1 - y_2)^2$ ，由于 $a \geq 0$ ，可知函数曲线是一条开口向上的抛物线（或者直线）。函数最小值在对称轴 $t = -b/2a$ 处取得，最小值 $D_{\min} = \sqrt{((4ac - b^2)/4a)}$ 。若 $D_{\min} < r$ ，则 B_1 和 B_2 满足相撞的约束条件。此时利用求根公式确定方程 $at^2 + bt + c - r^2 = 0$ 的两个根 $root_1$ 和 $root_2$ ，则相撞时间 $T = \max(0, \min(root_1, root_2))$ 。需要注意，当 a 的绝对值较小时，例如小于 $\text{fabs}(a) < 10^{-8}$ ，则认为 a 为 0，此时函数曲线退化为一条直线，若有 $D_{\min} < r$ ，则根据题意相撞时间 $T = 0$ 。

11583 Alien DNA。由于题目表述的原因，容易使得解题者无法理解题目的含义。以样例输入中的第一组数据为例，一共有 5 个 DNA 片段：as, sd, df, fg, gh，要求使用最少分划分次数将这 5 个 DNA 片段从前往后划分为若干部分，使得每个部分所包含的 DNA 片段中的碱基集至少有一个公共的碱基，使用 2 次划分，划分为 3 个部分，第一部分：as, sd，拥有公共的碱基 s，第二部分：df, fg，拥有公共的碱基 f，第三部分：gh，只有一个片段，不需考虑约束。容易验证，上述划分是最优的一种方案之一（另外一种可行的只需 2 次的划分方案是：(as); (sd, df); (fg, gh))。使用贪心策略从前往后扫描，只要共同碱基数量不为 0 就将其纳入划分，这样可以使得划分次数最少。

11585 Nurikabe。题目包含若干组测试数据，其中阴影方格包含数字，对于此类测试数据，一律输出“not solved”。例如以下测试数据：

```
1
2 3 2
0 0 2
1 2 1
..#
###
```

坐标为 (1, 2) 的方格为阴影方格，但是却为其指定的数字值 1，因此需要输出“not solved”。

11587 Brick Game。测试数据中存在全为“L”的测试数据，对于此类测试数据，由于 S 不能为空集，需要输出 $n+1$ ，其中 n 为游戏胜负结果字符串的长度。

11601 Avoiding Overlaps。由于给定的矩形坐标值均为整数且范围较小，存在更为简洁的解题方法——填充标记法。将给定矩形的坐标增加偏移量 100 以便调整为非负整数，设立一个二维数组 $grid[200][200]$ 记录已填充的方格，对于给定的某个矩形，检查在矩形范围内的方格是否已经填充，若至少有一个方格已经填充，则表明当前矩形与已绘制的矩形产生重叠。若给定矩形范围内的方格均未填充，则将此矩形面积累加，并标记相应的方格为已填充。若给定的矩形坐标为浮点数或者虽为整数但范围较大，则填充标记法将不可行。

11605 Lights Inside a 3D Grid。令单个方格被选中的概率 p ， $on[i]$ 表示第 i 次操作后方格内灯为点亮状态的概率， $off[i]$ 表示第 i 次操作后方格内灯为熄灭状态的概率，有

$$\begin{aligned} on[1] &= p \\ on[i] + off[i] &= 1, \quad i \geq 1 \\ on[i] &= on[i-1] * (1-p) + off[i-1] * p, \quad i \geq 2 \end{aligned}$$

解得

$$on[i] = \frac{1 - (1 - 2p)^i}{2}, \quad i \geq 1$$

11610 Reverse Prime。由于查询操作量较大，且有删除操作，必须使用时间复杂度为 $O(n \log n)$ 的算法才能在限定时间内通过。需要综合运用线性素数筛法、树状数组、二分搜索进行解题。

11627 Slalom。截至 2022 年 11 月 13 日，在线评测的结果很可能存在问题，即使采用了正确的二分搜索算法也无法获得 Accepted。可以使用以下技巧获得 Accepted：在线测试数据的输入实际上被截断了，经过测试，评判数据只包含 8 组测试数据，且最后两组测试数据被截断，也就是说，输入文件中的数据不完整，因此算法正确也无法得到正确输出，由于评判数据是来源于原题目的测试数据，UVa 的用户 Rene Argento (<https://uhunt.onlinejudge.org/id/913936>) 通过搜索获得了原始数据，并指出，对于前 6 组测试数据，正常读入并处理，而对于最后 2 组测试数据，只需按照如下输出即可获得 Accepted。

```
int cases; cin >> cases;
for (int cs = 1; cs <= cases; cs++) {
    // 特别处理
    if (cs == 7) { cout << "186566\n"; continue; }
    if (cs == 8) { cout << "3\n"; break; }
    // 正常处理
}
```

11628 Another Lottery。注意题目所求：确定第 i 个人比其他人能够赢得“更多”钱的概率。

11638 Temperature Monitoring。令初始为 0 时刻，可以确定第 i 个常温区间的起始时间 s_i 和终止时间 t_i ，确定测温装置是否能够记录该常温区间的温度，等价于确定以下不等式是否存在非负整数解（如果是第一个区间，根据题意，左侧的小于号需要更改为小于等于号）：

$$s_i \leq S + x \times M \leq t_i$$

如果存在非负整数解，则能够记录该常温区间的温度，根据报警器的相应设置判断其是否能够触发即可。

11643 Knight Tour。可以将问题转化为旅行商问题并使用动态规划算法解决，读者可参阅本书第 11 章“动态规划”中第 11.6.3 小节“旅行商问题”的内容。由于本题的测试数据规模较大且时间限制较紧，如果每次都重新计算两个马所在位置的最短移动步数则容易超时，需要使用技巧利用之前计算得到的结果从而缩短程序运行时间。

11648 Divide the Land。使用二分搜索解题。

11660 Look-and-Say Sequences。根据题意进行模拟即可。需要注意的是，每次生成数列中的下一个项时，并不需要生成此项所包含的所有数字，这样做容易导致超时，只需生成必要位数的数字即可（“必要”所要求的条件请读者自行思考得出）。

11664 Langton's Ant。解题的关键步骤是将大整数转换成二进制数以得到网格的颜色状态。注意，目标方格 (n, n) 位于网格的右上角，起始方格 (x, y) 给出的是蚂蚁在网格中的直角坐标。

11692 Rain Fall。可以根据题意得到一个一元二次方程，其中未知数为降雨强度，单位为 mm/h （毫米/小时）。

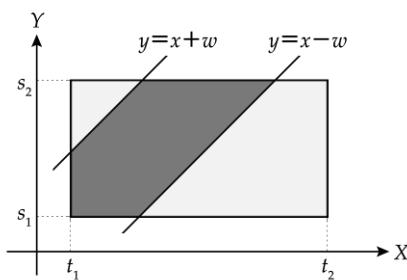
11693 Speedy Escape。劫犯选择的逃跑路线需要满足以下条件：以某个固定的速度到达路线上的某个交叉口的时间总是要小于警察以 160km/h 的速度通过某条具有最短距离的路线到达此交叉口的时间。需要确定劫犯和警察到达各个交叉路口的最短距离，然后计算出劫犯到达某个交叉路口所需的最低速度，之后使用二分搜索来确定最低可能的速度，即设定某个速度，检查是否能够在前述的最低速度限制下到达任意一个出口。注意，劫犯在到达某个交叉路口的过程中，不能经过警察所在的交叉路口。另外，由于输出时精度要求较高，在二分搜索时需要控制迭代的次数，而不是仅控制上限值和下限值之间的差值，若仅控制后者，可能会造成无限循环而导致超时。

11695 Flight Planning。此题需要求出原树的直径，然后沿着直径逐一尝试删除边，这样会将原树分割为两棵树，再使用一条边将两棵树的绝对中心连接起来，确定得到的新树的直径，取最小值即为结果。

11715 Car。此题需要运用物理中匀变速直线运动的速度和位移计算公式。设初速度为 u ，末速度为 v ，加速度为 a （加速度为矢量，其数值可能为负值，表示其方向与速度的方向相反），位移为 s ，则有关系： $v=u+at$, $s=ut+\frac{1}{2}at^2$ 。对于给定 v 、 a 、 s 要求计算 u 、 t 的情形，在某些特定输入下，解方程会得到两个不同的 t 值，总是取能够使得 u 为正数的 t 值。

11719 Gridland Airports。此题和完全二部图生成树计数有关。给定一个左部分为 n 个顶点，右部分为 m 个顶点的完全二部图，其生成树个数为 $m^{n-1} \times n^{m-1}$ 。

11722 Joining with Friend。令你到达的时间为 x ，朋友到达的时间为 y ，有 $t_1 \leq x \leq t_2$, $s_1 \leq y \leq s_2$ ，如果满足 $|x-y| \leq w$ ，则你和朋友能够相遇。问题转化为确定区间 $[t_1, t_2]$ 和 $[s_1, s_2]$ 在直角坐标系上所“围成”的矩形（即矩形在 X 轴上的投影为区间 $[t_1, t_2]$ ，在 Y 轴上的投影为区间 $[s_1, s_2]$ ）在直线 $y=x+w$ 的下方和直线 $y=x-w$ 的上方之间（阴影）部分面积与整个矩形面积的比值。



11754 Code Feat。令 x 为符合题意要求的正整数，因为给定的 X_i ($1 \leq i \leq C$) 满足互素的条件，则不管 x 模 X_i 的值 Y_j ($1 \leq j \leq k_C$) 为何，同余方程组 $x \equiv Y_j \pmod{X_i}$ 必定有解。当余数种类的乘积 $K = k_1 \times k_2 \times \dots \times k_C$ 较小时（例如小于 10000），可以通过中国剩余定理求解此 K 组不同的同余方程组来得到所有情形下的解，之后选取前 S 个值最小的解输出（若不足 S 个，则依次选择已有的解 x ，对其递增 $M = X_1 \times X_2 \times \dots \times X_C$ 得到后续解，直到满足 S 个的要求）。但当 K 较大时，此时解所有情形的同余方程组在规定时间内不可行，需要采用枚举的方法。具体是找到 k_i/X_i 的值最小的某个“线索”，令 $x = t \times X_i + Y_j$, $t \geq 0$ ，检查是否符合条件。为何选择 k_i/X_i 的值最小的“线索”进行枚举呢？因为这样 x 增长得最快，枚举所需要的次数也最少，也就能够在最快的时间内找到满足要求的 S 个值最小的解。

11762 Race to 1。设 X 表示将 D 变为 1 所需步数， Y 为选取的小于等于 D 的某个素数，根据全期望公式，有

$$E[X] = E[E[X|Y]] = \sum_i E[X|Y = p_i] P\{Y = p_i\}$$

对 D 进行素因子分解, 得到

$$D = p_1^{e_1} p_2^{e_2} \cdots p_m^{e_m}$$

令 n 表示小于等于 D 的素数个数, 则当随机选择这 n 个素数中的一个素数 p_i 去整除 D 时, 如果 p_i 恰好是 D 的素因子, 则

$$E[X|Y = p_i] = 1 + E\left[\frac{X}{p_i}\right]$$

若 p_i 不是 D 的素因子, 则

$$E[X|Y = p_i] = 1 + E[X]$$

则有

$$E[X] = \frac{n-m}{n} (1 + E[X]) + \frac{1}{n} \sum_{i=1}^m \left(1 + E\left[\frac{X}{p_i}\right]\right)$$

整理可得

$$E[X] = \frac{n-m + \sum_{i=1}^m \left(1 + E\left[\frac{X}{p_i}\right]\right)}{m}$$

边界条件 $E[1]=0$, 结合备忘技巧求解即可。

11774 Doom's Day。以题目中所给的 $3^2 \times 3^1$ 网格为例 ($n=2, m=1$), 首先按照三进制, 从序号 0 开始, 区分行编号和列标号为每个方格编号。例如, 数字 4 位于第二行第一列, 其编号为 $[01]_R[0]_C$ 。按照题意对方格进行第一次重排后, 数字 4 位于第四行第一列, 其编号为 $[10]_R[0]_C$, 进行第二次重排, 数字 4 位于第一行第二列, 其编号为 $[00]_R[1]_C$, 进行第三次重排, 数字 4 位于第二行第一列, 其编号为 $[01]_R[0]_C$, 回到原位。由于变换的对称性, 其他数字也同样回到了原位, 因此, 对于 $3^2 \times 3^1$ 网格, 其天数周期为 3。观察三进制表示的方格编号, 每次重排, 数位 1 向右移动 2 位, 恰等于 n , 这不是巧合而是规律所在。读者可以在更大的网格上 (例如 $n=3, m=4$) 进行试验, 可以发现同样的规律。一般地, 给定一个 $3^n \times 3^m$ 网格, 从 0 开始使用三进制为方格的行和列进行编号, 则每进行一次重排, 对应三进制数的所有数位循环右移 n 位, 当三进制数的各个数位回到原位时, 则对应网格中的数字也回到原位, 由于转换得到的三进制数数位长度为 $(n+m)$, 那么就是需要确定最小的重排次数 x , 使得 $x \cdot n \equiv 0 \pmod{(n+m)}$, 容易解得 $x = \gcd(n, n+m) = \gcd(n, m)$, 则最小的重排次数为 $(n+m)/x = (n+m)/\gcd(n, m)$ 。此处 \gcd 表示最大公约数。

11782 Optimal Cut。可以结合备忘技巧解题。使用两个参数来表示状态: 当前结点的序号, 以当前结点为根的子树其最优割至多包含的结点数。对于每个结点, 要么选择该结点, 要么选择其左右子树, 如果选择左右子树, 则左右子树各自的最优割所包含的结点数之和不超过 k ($k \leq K$), 逐一枚举所有情形取最大值即可。

11795 Mega Mans Missions。需要注意, 武器可以重复使用。

11806 Cheerleaders。给定 M, N, K , 假设 S 表示在 M 行 N 列矩阵中放置 K 个队员的方法数, A 表示在第一行不放置队员的方法数, B 表示在最后一行不放置队员的方法数, C 表示在第一列不放置队员的方法数, D 表示在最后一列不放置队员的方法数, 那么符合题目要求的方法数为:

$$S - (A + B + C + D) + (AB + AC + AD + BC + BD + CD) - (ABC + ABD + ACD + BCD) + ABCD$$

应用容斥原理结合组合数即可确定。

11816 HST。截至 2020 年 1 月 1 日, 由于题目描述不够明确和特殊的精度要求, 导致此题通过率较低。对于每种物品, 需要征收三种税, 假设税率分别为 $PST=1.2\%$, $GST=2.5\%$, $HST=5.69\%$, 某类物品价格为 \$100.78, 则 PST 税为 1.20936, GST 税为 2.5195, HST 税为 5.734382, 题意要求在各类物品的税收相加之前进行四舍五入, 那么此类物品的 PST 税为 1.21, GST 税为 2.52, HST 税为 5.73, 最终 PST 税和 GST 税之和为 3.73, 与 HST 税的差额为 2.00。为了能够获得 Accepted, 在计算过程中需要避免使用浮点数, 而应将所有数值转换为整数后再进行运算。

11841 Y-Game。如果某个结点是三角形网格的一个顶点, 例如, 坐标为 $(0, 0, n)$ 的结点, 则该顶点认为同时位于 x 边和 y 边。另外, 赢得 Y-Game 的要求的是连通块至少有一个结点在 x 边, 至少有一个结点在 y 边, 至少有一个结点在 z 边。

11860 Document Analyzer。解题的关键是找出所有能够包含全部不同单词的区间。对于给定的区间 $[p, q]$, 只有两种可能: (1) 区间已经包含了全部不同的单词; (2) 区间尚未包含全部不同的单词。对于第 (1) 种情况, 检查是否可以通过删除区间的首元素, 在缩小区间的同时仍能够保持包含全部不同单词的性质。对于第 (2) 种情况, 需要继续扩展区间以便能够包含全部不同的单词。可以只使用 map 来完成解题, 一个 $map<int, string>$ 记录区间内各个序号所对应的单词, 另外一个 $map<string, int>$ 记录当前区间所包含的不同单词及对应单词的个数。第一个 map 的功能用 $queue<pair<int, string>>$ 替代, 第二个 map 使用 $unordered_map$, 效率会更高。

11916 Emoogle Grid。令 r 为障碍所在行的最大值, 则 M 至少应该为 r 。根据题目的约束, 可以分为 $M=r$, $M=r+1$, $M>r+1$ 三种情况求解。对于前两种情况, 可以应用组合计数计算方案数, 检查模 100000007 的值是否为 R 。对于 $M>r+1$, 每增加一行, 总的染色方案数就在原 $r+1$ 行染色方案数的基础上乘以一次 $(K-1)^N$, 令 $M=r+1$ 时的染色方案数为 C , 且令 $X=(K-1)^N$, 则转化为求离散对数 $C \times X^Y \equiv R \pmod{100000007}$, 由于 100000007 是素数, 故 Y 有最小解, 则所求 $M=r+1+Y$ 。

11952 Arithmetic。截至 2020 年 1 月 1 日, 此题的题目描述不够明确使得通过率较低。(1) 在进行进制转换时, 默认一个数字对应一个数位。(2) 对于 1 进制的数需要进行特殊处理, $11 + 11 = 1111$ 是合法的 1 进制加法。(3) 对于评测数据来说, 最大只需尝试到十八进制即可。

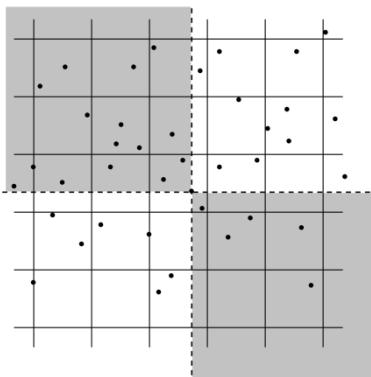
11968 In The Airport。此题可使用分数间的精确比较来判定物品价格与平均价格的关系。注意, 根据题意, 当两种物品的价格与平均价格的差的绝对值相同时要选择价格更小的物品。

11971 Polygon。正向计算能够构成多边形的概率存在困难, 不妨从反向考虑, 先计算不能构成多边形的概率, 然后求得构成多边形的概率。由多边形的性质可知, 任意一条边长必小于其他边长之和。考虑将给定的线段首尾相连构成一个圆, 第一次切割的位置可以为圆上的任意一个位置, 第一次切割后, 圆断开成为线段, 与原有问题等价。令第一次切割的位置为 C_1 , 令 C_1 关于圆心的对称位置为 C_2 , 如果后续切割全部位于 C_2 的一侧 (左侧或右侧), 则切割得到的线段中最长的线段至少为原始线段长度的 $1/2$, 由多边形的边长性质可知, 这将导致切割得到的线段无法构成一个多边形, 因此只要求出这种情况的概率即可得到无法构成多边形的概率, 进而可以得到能够构成多边形的概率。由于第一次切割是将圆断开成为线段, 故选择圆上任意一点作为第一次切割的位置均可, 其概率可以认为是 1, 接着进行 n 次切割, 将线段分割为 $n+1$ 段, 如果其中任意一段的长度至少为原始线段长度的 $1/2$ 则无法构成多边形, 假设其中一条线段 x_i 的长度大于等于原始线段的 $1/2$, $1 \leq i \leq n+1$, 则其他线段的切口需要全部位于线段 x_i 的一侧, 其概率为 $1/2^n$, 由于 x_1 到 x_{n+1} 这 $n+1$ 条线段均有可能是长度大于等于原始长度 $1/2$ 的那条线段, 则总的不能构成多边形的概率是 $(n+1)/2^n$, 最终能够构成多边形的概率为 $1 - [1 \times (n+1)/2^n] = 1 - [(n+1)/2^n]$ 。

11983 Wired Advertisement。题目所求为矩形范围内（包括边界上）所有整数坐标点的数量，可以将给定的矩形右上角的坐标 (x_2, y_2) 更改为 (x_2+1, y_2+1) ，使用求矩形并的面积方法所求得的覆盖至少 k 次的面积即为所求。由于普通的矩形并的面积所求是覆盖至少1次时的面积，与题目所求不符，因此需要为线段树的结点增加信息域 $height[i]$ ——在结点所表示的区间上覆盖至少 i 次的区间长度， $0 \leq i \leq k$ 。

11987 Almost Union-Find。对于第二种操作，设 p 所在集合的代表为 F_p ， q 所在集合的代表为 F_q ，不能简单地将 F_p 的祖先设置为 F_q ，因为 p 可能是 p 所在集合的代表，即 $p=F_p$ ，进行上述操作会将 p 所在的集合与 q 所在的集合合并，不符合第二种操作的预期。

11990 Dynamic Inversion。由于给定的是 $[1, n]$ 内的自然数的排列，则可将序号 i 和数组元素 $a[i]$ 构成的二元组 $(i, a[i])$ 视为二维平面上的一个点。易知，位于 $(i, a[i])$ 左上角区域内的点以及位于 $(i, a[i])$ 右下角区域内的点的纵坐标 $a[j]$ 与 $a[i]$ 构成逆序对，如下图所示（注意，图形并未按本题的实际情况绘制）：



可以考虑使用“二维”根号分块算法，即将平面区域按照类似于一维的情形，沿X轴和Y轴分块，则平面区域将被划分为若干的方格。令 $countSum(x, y)$ 表示左下角顶点为 $(0, 0)$ ，右上角顶点为 (x, y) 的矩形区域所包含的点数，那么将某个数 $a[k]$ 删除，则统计位于 $(k, a[k])$ 左上角和右下角的区域所包含点的总数量 $sum = countSum(k, n) + countSum(n, a[k]) - 2 \times countSum(k, a[k])$ ， sum 即为删除 $a[k]$ 后逆序对数减少的数量。为了提高效率，可以维护每行方格包含点数目的前缀和，在统计包含点数时，直接累加相应的前缀和，在删除 $a[k]$ 后，使用 $O(\sqrt{n})$ 的时间更新前缀和即可。

11992 Fast Matrix Operations。如果使用四叉树的方式来实现二维线段树，必须使用延迟更新技巧才能在限定时间内获得 Accepted。

12002 Happy Birthday。由于序列中存在相同的元素，需要对解题思路进行适当扩展。仍然按照类似于 11456 Trainsorting 的方法求得从第 i 个盘子开始的最长不递减子序列 $dp_1[i]$ 和从第 i 个盘子开始的最长不递增子序列 $dp_2[i]$ ，即

$$dp_1[i] = \max\{dp_1[i], dp_1[j] + 1\}, \quad i < j, \quad weight[i] \leq weight[j]$$

$$dp_2[i] = \max\{dp_2[i], dp_2[j] + 1\}, \quad i < j, \quad weight[i] \geq weight[j]$$

则符合题意的最长子序列可由以下两部子序列构成：从第 i 个盘子开始的最长不递减子序列加上在第 i 个盘子之后且比第 i 个盘子小的第 j 个盘子开始的最长不递增子序列（或者是从第 i 个盘子开始的最长不递增子序列加上在第 i 个盘子之后且比第 i 个盘子大的第 j 个盘子开始的最长不递减子序列），即题目所求为

$$M = \max \begin{cases} \max\{dp_1[i], dp_2[i]\} & weight[i] = weight[j] \\ dp_1[i] + dp_2[j] & weight[i] > weight[j] \quad 1 \leq i < j \leq n \\ dp_2[i] + dp_1[j] & weight[i] < weight[j] \end{cases}$$

12005 Find Solutions。对给定的表达式进行适当变换，进行因式分解可得 $4c-3=(2a-1)(2b-1)$ 。

12022 Ordering T-Shirts。参阅 OEIS 序号为 A000670 的数列。由于 n 较小，本题也可使用回溯法来解题。

12028 A Gift from the Setter。对求和计算式进行适当变换后，可以利用排序和前缀和予以解决。

12030 Help the Winner。此题可以转化为隐式有向图的路径计数问题。恰当的定义状态（已经匹配的服装，当前匹配的服装，是超级匹配还是完全匹配）是关键，同时需要结合备忘技巧、位掩码技巧（状态压缩）进行解题。

12063 Zeros and Ones。当 N 为奇数或者 $K=0$ 时，符合要求的二进制数个数为 0。

12070 Invite Your Friends。使用类似于 BFS 的方法，检查某个朋友在 T 天内所能到达的城市 C ，同时确定到达城市 C 时所需要的最小花费，最后枚举所有城市，取所有朋友花费最小的城市即为解。

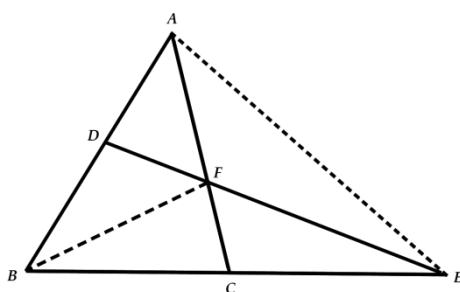
12071 Understanding Recursion。本题可以使用求逆序对数的方法予以解决，但存在更简单的解题方法。观察题目中所给定的代码，实际上统计的是以下数量：对于数组中的第 i 个元素 x ，确定数组中小于 x 的元素个数 c_i ，然后将 c_1 到 c_n 的值累加起来即为结果。将数组进行排序，依次扫描获得 c_i 累加即可。

12090 Counting Zeroes。使用前述介绍的方法求所有约数，由于中间过程会生成重复的约数使得效率不够高，可以使用回溯法根据素因子分解的结果来生成所有约数，则每次生成的约数都是唯一的，相比较而言更为高效。不过对于题目给定的数据规模来说，最大输入整数可能的不同约数个数不会超过 $2^{13}=8192$ 个，因此两种方法的效率差别可能并不是很明显。

12096 The SetStack Computer。此题可以利用映射和 STL 算法库中集合的交(set_intersection)、并(set_union)模板函数来解题。

12143 Stopping Doom's Day。计算 $n \in [1, 50]$ 时的 T 值，观察总结规律。

12165 Triangle Hazard。可以根据梅涅劳斯定理 (Menelaus' theorem) 求解。给定三角形 ABC ，当一条直线与三角形的三条边所在直线 AB 、 BC 、 CA 相交，假设分别交于 D 、 E 、 F 点。



根据三角形面积的关系，有

$$\frac{AD}{DB} \cdot \frac{BE}{EC} \cdot \frac{CF}{FA} = \frac{S_{AEF}}{S_{BEF}} \cdot \frac{S_{BEF}}{S_{CEF}} \cdot \frac{S_{CEF}}{S_{AEF}} = 1$$

上述等式即为梅涅劳斯定理。对于给定题目中给定的三角形，根据梅涅劳斯定理，有以下等式，

$$\frac{AR}{RP} \cdot \frac{PQ}{QB} \cdot \frac{BF}{FA} = 1$$

$$\frac{BP}{PQ} \cdot \frac{QR}{RC} \cdot \frac{CD}{DB} = 1$$

$$\frac{CQ}{QR} \cdot \frac{RP}{PA} \cdot \frac{AE}{EC} = 1$$

亦即

$$\frac{AR}{RP} \cdot \frac{PQ}{PQ + BP} = \frac{m_5}{m_6} = k_1$$

$$\frac{BP}{PQ} \cdot \frac{QR}{QR + RC} = \frac{m_1}{m_2} = k_2$$

$$\frac{CQ}{QR} \cdot \frac{RP}{RP + RA} = \frac{m_3}{m_4} = k_3$$

令

$$\overrightarrow{RA} = x_1 \overrightarrow{PR}, \quad \overrightarrow{BP} = x_2 \overrightarrow{PQ}, \quad \overrightarrow{QC} = x_3 \overrightarrow{RQ}$$

可得三元一次方程组

$$\begin{cases} x_1 = k_1(1 + x_2) \\ x_2 = k_2(1 + x_3) \\ x_3 = k_3(1 + x_1) \end{cases}$$

解得

$$x_1 = \frac{k_1 + k_1 k_2 + k_1 k_2 k_3}{1 - k_1 k_2 k_3}$$

$$x_2 = \frac{k_2 + k_2 k_3 + k_1 k_2 k_3}{1 - k_1 k_2 k_3}$$

$$x_3 = \frac{k_3 + k_3 k_1 + k_1 k_2 k_3}{1 - k_1 k_2 k_3}$$

12256 Making Quadrilaterals。题目给定 $n \geq 4$ 根铁棒，要求任意 4 根铁棒不能构成四边形，求最长铁棒的最小可能长度。根据题意不能有 4 根或 4 根以上的铁棒具有相同的长度，设 n 根铁棒按长度递增的顺序排列为 L_1, L_2, \dots, L_n ，为了使得无法构成四边形，由三角形边不等式可以推出需要满足条件： $L_i \geq L_{i-1} + L_{i-2} + L_{i-3}$ ， $i \geq 4$ 。由此关系可以求得最长铁棒的最小可能长度。

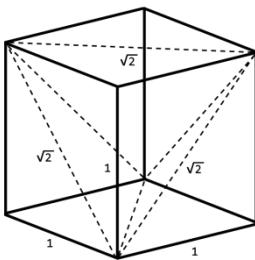
12171 Sculpture。由于给定的长方体可能围成一个“空腔”，该“空腔”未被任何长方体所包含，而其体积计入结果但表面积却不计入结果，因此直接计算存在困难。可以设想在组合体的表面附加一层“空气”，通过 BFS（使用 DFS 可能会因为递归深度较大而导致栈溢出）确定“空气”层的体积，则组合体的体积为总体积减去“空气”层的体积，而组合的表面积为组合体与“空气”层接触面的面积之和。

12269 Lawn Mower。注意题意要求，草坪的横向和纵向都至少除一次草，而不是横向或纵向至少除一次草。

12291 Polyomino Composer。截至 2020 年 1 月 1 日，此题在 UVa OJ 上的评测数据仍存在以下问题：某些测试数据在一行上所包含的字符数小于 n 或 m 所指定的字符数。使用 `getline(cin, line)` 先读入整行字符然后再赋值到二维数组中，不足的部分补充 ‘.’，可以获得正确的输入。否则很有可能因为不正确的输入导致最后的结果错误，尽管算法在逻辑上是正确的。

12307 Smallest Enclosing Rectangle。类似的，可以证明：对于凸多边形的最小周长外接矩形，此矩形至少有一条边与凸多边形的一条边重合。由于此题的数据规模较大，不能使用类似于 10173 Smallest Bounding Rectangle 时间复杂度为 $O(n^2)$ 的算法，而必须使用时间复杂度为 $O(n)$ 的旋转卡壳算法予以解决。

12308 Smallest Enclosing Box。初始猜测最小体积外接长方体的某个面必定与凸多面体的某个面重合，由此得到以下算法：先求给定点集的三维凸包，枚举三维凸包的每个面 P 作为与外接长方体重合的面，将其他凸包顶点投影到平面 P ，记录投影时各点距离平面 P 的最大距离 H ，然后求所有投影点的二维凸包 C ，利用旋转卡壳法确定二维凸包 C 的外接矩形的最小面积 A ，则外接长方体的体积 $V = A \times H$ ，对三维凸包的所有面 P 进行上述操作，取 V 的最小值即为解。但实际上，凸多面体的最小体积外接长方体的面并不一定与多面体的某个面重合。如下图所示，实线表示的是边长为 1 的正方体，虚线表示的是棱长均为 $\sqrt{2}$ 的四面体，四面体的最小外接长方体即为实线所示的棱长为 1 的正方体。不难看出，正方体的所有面均不与四面体的任意一个面重合。



在 O'Rourke 的论文中，证明了凸多面体的最小体积外接长方体至少有两个相邻的面与多面体的边重合，并由此提出了基于高斯球 (Gaussian sphere) 的三维“旋转卡壳”算法，具体细节请读者阅读给出的参考文献（论文可在 O'Rourke 的论文索引网页下载：<http://cs.smith.edu/~jorourke/papers.php>, 2020），不过该算法实现起来较为复杂，可以考虑使用随机寻优近似算法解题，例如模拟退火算法 (simulated annealing algorithm)。

12311 All-Pair Farthest Points。根据题目的数据规模，需要寻求时间复杂度至少是 $O(n \log n)$ 的算法，可以利用凸包的凸性辅助解题。

12324 Philip J. Fry Problem。由于每段旅程最多只能使用一块燃料饼，因此对于任意一段旅程来说，只需考虑有 0 至 n 块燃料饼可用状态下的最短旅行时间。值得一提的是，此题亦可使用简洁的贪心法予以解决，贪心法基于以下事实：每段旅程最多只能使用一块燃料饼，那么将燃料饼使用在时间最长的旅程上所获得的效益也相应是最大的，因此按旅程时间从长到短的顺序来使用燃料饼可以获得最短的旅行时间。

12363 Hedge Maze。找出给定图中的所有割边（桥），能够通过割边连通的顶点之间存在唯一简单路径，使用并查集即可判定。

12366 King's Poker。既可以使用分类讨论方法进行解题，也可以制定排序规则，利用库函数 `sort` 将所有可能的一手牌 (hand) 进行排序，然后寻找比给定的一手牌大的另外一手牌。相较而言，使用分类讨论的方法更为直接和简便。

12379 Central Post Office。假设从起始邮局出发，遍历所有其他邮局且需要回到起始邮局，由于给定的图是树，任意顶点对之间的最短路径唯一，那么所有边都必须经过两次。如果不需要回到起始邮局，应该恰当选择起始邮局和终止邮局，使得两个邮局之间的路径尽可能地长，这样可以接受尽可能多的时间，容易知道，将树的最长路径的一端作为邮局的起始位置最优，题目所求即为所有边经过两次的时间减去经过树中最长路径所需的时间，亦即确定树的直径即可。

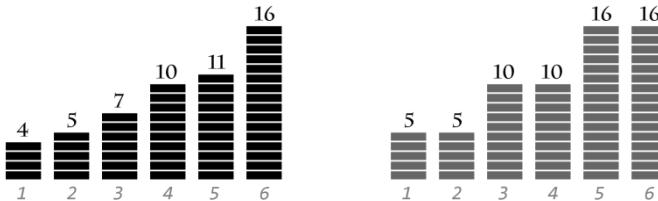
12412 A Typical Homework。需要注意以下几点：(1) 在输出主菜单时，最后一行是一个空行；(2) 选择主菜单的第五项输出统计资料时，最后一行是一个空行；(3) 在查询后输出符合要求的学生信息时，需要按照学生被加入数据库的先后顺序输出，即先添加的学生其信息需要先输出；(4) 学生的排位 (rank) 可能不是连续的；(5) 输出分数平均值时需要加上一个小常数（例如 10^{-5} ）以避免浮点数误差带来的输出差异；(6) 在题目描述中并未明确当学生总数为 0 时平均分应该如何输出，但截至 2021 年 12 月 7 日，评测数据似乎并不包含这样的测试数据，因此输出“0.00”即可，不影响获得 Accepted。

12428 Enemy at the Gates。题目描述中的“critical road”对应图论中的“割边（桥）”。令割边数为 k ，由于每条割边会“占用”一个顶点，余下的 $N-k$ 个顶点至少需要构成 $M-k$ 条边，而题目要求不能出现平行边或者自环，则由余下的 $N-k$ 个顶点构成一个 $N-k$ 阶完全图，能够使得图的边数最大化。容易知道， n 阶完全图中每个顶点与其他顶点均有一条边相连，即每个顶点的度均为 $n-1$ ，根据握手引理， n 阶完全图总的边数为 $n(n-1)/2$ 。二分搜索割边数 k ，检查条件 $(N-k)(N-k-1)/2 \geq M-k$ 是否满足，若满足，表明 k 可行，否则不可行。

12461 Airplane。可以应用数学归纳法证明所求概率是一个常数，请读者自行思考并予以证明。

12464 Professor Lazy Ph.D.。虽然出题者的构思非常巧妙，但由于题目描述上存在一定问题，使得题目整体质量不高。如果按照题目中给定的递推公式计算序列中各项的值，中途会发生除零的情况（虽然题目中明确指出“不存在”这种情况），需要使用另外一种方式来得到“通项公式”。根据递推公式及分数运算，可以依次得到： $Q_0=\alpha$, $Q_1=\beta$, $Q_2=(1+\beta)/\alpha$, $Q_3=(1+\alpha+\beta)/(\alpha\beta)$, $Q_4=(1+\alpha)/\beta$, $Q_5=\alpha$, $Q_6=\beta$, ……循环周期为 5。

12499 I am Dumb 3。将问题转化为 Nim 游戏加以解决。



如上图左侧所示，令 $n=6$, $L=20$ ，初始状态时，硬币堆的数量从左至右依次为 4, 5, 7, 10, 11, 16。按照题目约束，上图右侧所示为“安全态”，当前玩家必败，因为在此状态下，当前玩家只能对偶数编号的硬币堆进行操作，而下一玩家可以采取可逆操作 (reversible moves) 消除当前玩家操作的影响。例如，在“安全态”的基础上，当前玩家在第 2 堆硬币上增加了 3 枚硬币，使得第 1 堆硬币的数量变成 8 枚，那么下一玩家可以立即选择在第 1 堆硬币上增加 3 枚硬币，使得第 1 堆硬币的数量也变成 8 枚，从而使得当前状态仍然是一个“安全态”。那么问题转化为哪个玩家最先使得初始状态变成“安全态”谁就获胜。比较“安全态”和初始状态的差别，容易看出，与紧邻的奇数堆和偶数堆的硬币数量差值相关。将初始状态紧邻的奇数堆和偶数堆的硬币数量差值列出（若硬币堆总数是奇数，可以在最后添加一个硬币数量为 L 的虚拟硬币堆），即可得到一个等价的 Nim 游戏：3 个硬币堆，硬币数量分别为 1, 3, 5。

12545 Bits Equalizer。对字符串 S 和 T 逐个字符进行比较，如果序号 i 的字符相同，则不需要进行操作，否则有以下四种情形：（一） $S[i] = '1'$, $T[i] = '0'$ ，令此种情形的数量为 c_1 ；（二） $S[i] = '0'$, $T[i] = '1'$ ，令此种情形的数量为 c_2 ；（三） $S[i] = '?'$, $T[i] = '0'$ ，令此种情形的数量为 c_3 ；（四） $S[i] = '?'$, $T[i] = '1'$ ，令此种情形的数量为 c_4 。由于 S 中的 ‘1’ 无法直接变为 ‘0’，只能与其他位置的 ‘0’ 进行交换，因此第一种情形只能由第二种情形与之交换而消除。如果第一种情形的数量 c_1 大于第二种情形的数量 c_2 ，则可将第四种情形中的 ‘?’ 变为 ‘0’ 来补充，如果还不够，则无法完成题目所要求的变换，即 $c_2 + c_4 < c_1$ 时，无法按照题意完成转换。当 $c_2 + c_4 \geq c_1$ 时，可以完成转换，需要确定最少的操作次数。对于第三和第四种情形，无论如何均需将 ‘?’ 转换为 ‘0’ 或 ‘1’，因此操作次数至少为 $c_3 + c_4$ 。如果 $c_1 > c_2$ ，则将全部的第二种情形与第一种情形交换外还不够，此时需要将第四种情形中 ‘?’ 更改为 ‘0’ 来补足，总的操作次数是 $c_1 + c_3 + c_4$ 。如果 $c_2 > c_1$ ，则第一种情形只需第二种情形与之交换就足够，多余的第二种情形将 ‘0’ 更改为 ‘1’ 即可，总的操作次数是 $c_2 + c_3 + c_4$ 。因此，当 $c_2 + c_4 \geq c_1$ 时，最少操作次数为： $c_3 + c_4 + \max(c_1, c_2)$ 。

12643 Tennis Rounds。令两个参赛者的编号为 i 和 j , $i < j$, 初始时，轮数 r 为 1，两个参赛者位于同一轮的充分必要条件是： i 是奇数， j 是偶数，且两者相差 1。若两个参赛者的编号不满足上述条件，则将 i 和 j 各自加 1，然后都除以 2，轮数 r 加 1，检查是否满足条件，若满足条件，输出轮数 r ，否则继续操作。

12723 Dudu the Possum。令 $dp[i]$ 为负鼠从冰箱第 i 层出发到第 N 层时所能吸收卡路里的期望值，其递推关系式为

$$dp[i] = \sum_{j=1}^{Q_i} C_{ij} X_{ij} + \sum_{j=1}^K p_j dp[i+j], \quad i \geq 1$$

边界条件：当 $i > N$ 时， $dp[i] = 0$ 。所求即 $dp[1]$ ，结合备忘技巧解题即可。

12730 Skyrk's Bar。本题的难点在于递推关系式的推导。令 $dp[i]$ 表示 i 个连续的小便池能够容纳的人数期望，考虑初始时有 n 个空的小便池可用时的情形，第一个进入厕所的人以概率 $1/n$ 选择 n 个小便池中的任意一个，令其选择的小便池的编号为 j ，则后续能够使用的小便池为左侧的 $j-K-1$ 个小便池和右侧的 $n-j-K-1$ 个小便池（需要满足任意两人之间相隔 K 个小便池），根据条件期望有

$$dp[n] = \sum_{j=1}^n \frac{dp[j-K-1] + 1 + dp[n-j-K-1]}{n}$$

对递推关系式进行展开，可得

$$dp[n] = \frac{(dp[1] + 1 + dp[n-K-1]) + \dots + (dp[n-K-1] + 1 + dp[1])}{n} = 1 + \frac{2 \sum_{j=1}^{n-K-1} dp[j]}{n}$$

令

$$S[n] = \sum_{j=1}^n dp[j]$$

则有

$$dp[n] = 1 + \frac{2S[n-K-1]}{n}, \quad n \geq 1$$

边界条件：当 $1 \leq j \leq K+1$ 时，只能容纳一个人，因此 $dp[j] = 1$ ；当 $j < 1$ 时，无法容纳任何人，因此 $dp[j] = 0$ 。

12792 Shuffled Deck。题目和完美洗牌问题有关，等价于求最小的正整数 n ，使得 $2^n \equiv 1 \pmod{P+1}$ 。由于本题的测试数据规模较小，可以直接暴力枚举确定 n 。

12797 Letters。使用位掩码技巧遍历所有可能的字母组合，利用 BFS 确定最短路径。

12801 Grandpa Pepe's Pizza。如果给定的测试数据存在符合要求的切分方案，一定可以调整切分的位置（将所有的切分位置顺时针统一旋转适当的角度），使得每个切分位置均位于两个整数位置的中间，因此只需枚举少量的切分方案是否可行即可。以样例输入的第一组数据为例，由于 C 是 N 的倍数，切分后每块的弧长 $L=C/N=12/3=4$ 为整数，假定某处切分线位于第一颗橄榄和第二颗橄榄之间，那么只需尝试以下三种区间划分方案：(1) [8.5, 0.5], [0.5, 4.5], [4.5, 8.5]; (2) [9.5, 1.5], [1.5, 5.5], [5.5, 9.5]; (3) [10.5, 2.5], [2.5, 6.5], [6.5, 10.5]。检查上述三种区间划分方案中，是否满足每个区间有且只有一颗橄榄，如果满足，则表明符合要求的切分方案存在，输出‘S’，否则输出‘N’。在具体实现时，可以选择间隔最小的两颗橄榄区间进行尝试，这样需要枚举的切分方案能够达到最小，另外，可以将运算中所涉及的数值都乘以 2 转换为整数以避免使用浮点数。

12840 The Archery Puzzle。本题亦可通过动态规划予以解决。

12845 The Jolly Friar's Puzzle。可以先搜索得到最优的放置方案，然后将给定方案与最优方案逐一对比，确定最少的移动次数。需要注意，在计数“even line”的数量时，非主副对角线的其他对角线，若有偶数颗硬币，也计为“even line”。总共只有 4 种最优放置方案。



12850 Skating Puzzle。需要注意浮点数误差，截至 2023 年 4 月 10 日，对于评测网站上的测试数据，可能需要在结果上加上一个小的常数（例如， $1e-7$ ）才能获得 Accepted。

12862 Intrepid Climber。题目所求为顶点 1 到所有朋友所在顶点的路径之并的边权和，然后减去到达朋友的路径中边权之和最大的那条路径的边权和，最后即为结果。可以使用贪心思维，利用 BFS 确定顶点 1 到所有朋友所在顶点的最短路径的边数及边权和，然后按照最短路径的边数从大到小的顺序，累加边权，由于是求路径并的边权和，已经累加的不能重复累加，最后减去边权和最大的路径的边权和即可，因为最后不需要再返回顶点 1。亦可使用 DFS 结合动态规划思维予以求解。

12869 Zeroes。容易知道，将 $n!$ 进行素因子分解，每出现一对素因子 2 和 5， $n!$ 的结果中就会增加一个末尾 0，因此 $n!$ 的末尾 0 个数就等于 $n!$ 素因子分解后 2 和 5 的对数。进一步观察可知， $n!$ 在进行素因子分解时，素因子 2 的个数大于素因子 5 的个数，因为将 $n!$ 拆开成 1 到 n 分别进行素因子分解时，2 的倍数比 5 的倍数要多，因此 $n!$ 的末尾 0 的个数与 1 到 n 中 5 的倍数直接相关。亦即，1 到 n 中，有多少个数是 5 的倍数，则 $n!$ 的结果末尾就有多少个 0。

12911 Subset Sum。由于本题中 N 最大为 40，暴力生成所有子集和显然不可行。可以将数组分成两个数组 A 和 B ，其大小只相差 1，分别生成 A 和 B 的所有子集和，由于 N 最大为 40，则 A 的大小最大为 20，不同子集和的个数最多为 2^{20} 。然后对于数组 A 的每个子集和 s ，在数组 B 的子集和中查找 $T-s$ 是否存在，若存在，则将数组 A 中子集和 s 的数量 C_s 乘以数组 B 中子集和 $T-s$ 的数量 C_{T-s} 就是对总方案数的贡献，逐个处理数组 A 的所有子集和然后累加即可。可以使用 STL 中的映射 (map) 来存储子集和的个数。上述算法的时间复杂度为 $O(N \times 2^{N/2})$ 。需要注意，子集不能为空。

12931 Common Area。两个简单多边形如果满足以下四个条件中的任意一个（或多个），即可判定为交集不为空，否则给定的两个多边形不存在重叠部分。（1）两个多边形有两条边跨越式相交（非相交于多边形的顶点）；（2）某个多边形有至少一个顶点位于另外一个多边形内；（3）某个多边的一条边完全包含在另外一个多边形内；（4）以多边形的相邻两条边构成的所有三角形中，至少有一个三角形的重心位于两个多边形的内部。

12951 Stock Market。本题难点在于动态规划状态的设计。令 p_i 为第 i 天股票的价格， c 为购入股票的交易费用， f_i 表示第 i 天时拥有股票时的最大收益， g_i 表示第 i 天不拥有股票时的最大收益，则递推关系式为：

$$f_i = \max(f_{i-1}, g_{i-1} - p_i - c)$$

$$g_i = \max(g_{i-1}, f_{i-1} + p_i)$$

初始条件： $f_1 = -\infty$, $g_1 = 0$ 。

12995 Farey Sequence。手工列出当 n 较小 ($n \leq 7$) 时的解，可以发现规律：令 G_n 为满足要求的“成对”分数的数量， F_n 为 n 阶 Farey 序列中分数的个数，有 $G_n = F_n - 2$ ，而 $F_n = 1 + \varphi(1) + \varphi(2) + \dots + \varphi(n)$ ，故 $G_n = -1 + \sum_{i=1}^n \varphi(i)$ 。

参阅：

- (1) https://en.wikipedia.org/wiki/Farey_sequence, 2020;
- (2) <https://oeis.org/A015614>, 2020;
- (3) <https://www.maths.ed.ac.uk/~v1ranick/fareyproject.pdf>, 2020。

13025 Back to the Past。有多种解题方法：(1) 通过查询智能手机的日历解题；(2) 使用 Java 的 `Calendar` 类解题；(3) 根据某个已知日期（例如操作系统的当前日期）是星期几以及已知日期和 2013 年 5 月 29 日之间相差的天数来推算。(4) 由同余知识得到的计算公式进行计算。把 3 月算做这一年的第一个月，4 月算做这一年的第二个月，…，12 月算做这一年第十个月，下一年的 1 月算做这一年的第十一个月，下一年的 2 月算做这一年的第十二个月，在这样的规定下：1991 年 9 月 2 日就要写为“1991”年“7”月 2 日；而 1991 年 1 月 3 日就要写为“1990”年“11”月 3 日，即日期 D 按照如下格式指定：

$$D = "N" \text{ 年 } "m" \text{ 月 } d \text{ 日}$$

对星期几也给定一个数字予以表示：

星期日 = 0, 星期一 = 1, 星期二 = 2, 星期三 = 3, 星期四 = 4, 星期五 = 5, 星期六 = 6

这些代表星期几的数字称之为 **星期数**，令日期 D 的星期数为 W_D ，可以证明（参阅：Kenneth H. Rosen 著，夏鸿刚译，《初等数论及其应用，第 6 版》，第 144—146 页或者潘承洞，潘承彪著《初等数论，第 3 版》，第 507—511 页）：

$$W_D \equiv d + [(13m - 1)/5] + y + [y/4] + [c/4] - 2c \pmod{7}$$

公式中的除法为整除， c 和 y 由下式确定：

$$N = 100 \cdot c + y, \quad 0 \leq y < 100$$

对于本题来说，按照日期表示的约定，2013 年 5 月 29 日应写为：

$$D = "2013" \text{ 年 } "3" \text{ 月 } 29 \text{ 日}$$

所以 $c = 20$, $y = 13$, $m = 3$, $d = 29$, 由计算公式可得：

$$W_D \equiv 29 + [38/5] + 13 + [13/4] + [20/4] - 40 \equiv 29 + 7 + 13 + 3 + 5 - 40 \equiv 3 \pmod{7}$$

则 2013 年 5 月 29 日是星期三。需要注意的是，采用上述公式计算星期几只适用于格列高利历法中自 1582 年 10 月 15 日之后的日期，因为格列高利在对历法进行改革时，将 1582 年 10 月 4 日的下一天定为 1582 年 10 月 15 日，导致日期不连续。

13015 Promotions。对于给定的有向图，构建正向图和反向图，逐个枚举顶点进行判断。在正向图中，从给定顶点 u 所能到达的顶点即为必须将顶点 u 对应的人员提拔后才能提拔的人员，令其数量为 c ，那么 $E-c$ 即为不提拔顶点 u 所对应的人员的情况下，总共还能够提拔的人员总数，若 $E-c$ 小于 A ，表明不提拔 u 所对应的人员，无法达到提拔人数的下限，亦即必须提拔顶点 u 所对应的人员。同理，若 $E-c$ 小于 B ，表明不提拔 u 所对应的人员，无法达到提拔人数的上限，亦即必须提拔顶点 u 所对应的人员。在反向图中，从给定顶点 u 所能到达的顶点即为必须将这些顶点所对应的人员提拔后，才能提拔顶点 u 所对应的人员，令其数量为 d ，如果 d 大于 B ，则表明不需提拔 u 所对应的人员就可以使得提拔的人数达到上限值 B ，因此顶点 u 所对应的人员必定不会被提拔。

13030 Brain Fry。令 $probability[u][t]$ 表示在顶点 u 且已用时间 t 时能成功吃到“Brain Fry”的概率， $elapsed[u][t]$ 表示在顶点 u 且已用时间 t 时返回大学的期望时间，预处理得到餐馆与大学间的最短距离，然后根据题目给定的状态转移规则进行动态规划，同时利用备忘技巧提高效率。

13095 Tobby and Query。考虑到 $0 \leq a_i \leq 9$ ，范围很小，因此可以使用以下简便的方法：令 $dp[i][j]$ 表示区间 $[1, i]$ 内整数 j 出现的次数，暴力统计得到 $dp[i][j]$ ，则给定区间 $[l, r]$ ，该区间内不同整数的个数可以根据 $(dp[r][j] - dp[l-1][j])$ 的值是否为 0 来进行累加，即从 0 到 9 枚举 j ，若 $(dp[r][j] - dp[l-1][j])$ 大于 0，表示在区间 $[l, r]$ 内，整数 j 出现了至少一次，则最终区间 $[l, r]$ 内不同整数个数增加 1。那么如何暴力统计 $dp[i][j]$ 呢？可以使用递推的方法，假定第 i 个整数为 x ，则 $dp[i][0]$ 至 $dp[i][9]$ ，除了 $j=x$ 时， $dp[i][j]$ 比 $dp[i-1][j]$ 增加 1 之外，对于其他的 $j \neq x$ ， $dp[i][j]$ 均等于 $dp[i-1][j]$ 。

```
int dp[100010][10];
int main(int argc, char *argv[]) {
    int n, q, x;
    while (cin >> n) {
        for (int i = 1; i <= n; i++) {
            cin >> x;
            for (int j = 0; j < 10; j++) dp[i][j] = dp[i - 1][j];
            dp[i][x]++;
        }
        cin >> q;
        for (int i = 0; i < q; i++) {
            int l, r, answer = 0;
            cin >> l >> r;
            for (int j = 0; j < 10; j++) answer += (dp[r][j] - dp[l - 1][j]) > 0;
            cout << answer << '\n';
        }
    }
    return 0;
}
```

如果 a_i 的取值范围较大，则可以先将 a_i 离散化，之后使用树状数组予以解决。考虑到以下事实：给定区间 $[1, r]$ ，整数 x 若在此区间内出现，其对不同整数个数的贡献只由最右侧（亦即最后）出现的 x 来决定。举个例子，给定序列 1, 2, 3, 4, 3，如果统计区间 $[1, 5]$ 内不同整数的个数，则整数 3 的贡献只由最后一个 3 来决定。因此，可以先将查询区间排序，按查询区间的右侧端点递增排列，从左向右扫描序列，对于第 i 个整数 x ，将第 i 位标记置为 1，将之前出现的 x 的位置的标记值置为 0，令 $dp[i]$ 为区间 $[1, i]$ 的标记值的和，那么区间 $[l, r]$ 内不同整数的个数即为 $(dp[r] - dp[l-1])$ 。

```
const int MAXN = 100010;

int bit[MAXN], a[MAXN], pos[MAXN], answer[MAXN], n, m;

int query(int x) {
    int sum = 0;
    while (x) {
        sum += bit[x];
        x -= x & -x;
    }
    return sum;
}
```

```

        x -= x & -x;
    }
    return sum;
}

void add(int x, int k) {
    while (x <= n) {
        bit[x] += k;
        x += x & -x;
    }
}

struct interval { int l, r, id; } q[MAXN];

bool cmp(const interval & a, const interval & b) { return a.r < b.r; }

int main(int argc, char *argv[]) {
    while (cin >> n) {
        memset(bit, 0, sizeof bit);
        memset(pos, 0, sizeof pos);
        for (int i = 1; i <= n; ++i) cin >> a[i];
        cin >> m;
        for (int i = 0; i < m; i++) {
            cin >> q[i].l >> q[i].r;
            q[i].id = i;
        }
        sort(q, q + m, cmp);
        int r = 0;
        for (int i = 0; i < m; i++) {
            while (r < q[i].r) {
                r++;
                add(r, 1);
                if (pos[a[r]]) add(pos[a[r]], -1);
                pos[a[r]] = r;
            }
            answer[q[i].id] = query(r) - query(q[i].l - 1);
        }
        for (int i = 0; i < m; i++) cout << answer[i] << '\n';
    }
    return 0;
}

```

13113 Presidential Election。建议将所有数值转换为整数再进行运算以避免浮点数误差。

13151 Rational Grading。注意，题意要求是根据前一行的结果来确定当前行的结果是否正确，当为第一行时，则根据最初的输入来确定输出结果是否正确。例如，给定以下的测试数据：

```

0xFF 4
i++ 256
i-- 257
i 255
i++ 255
0 0

```

给定的初始输入为十六进制数 0xFF，对应的十进制数为 255，因此第一行的输出不正确，应该输出 255。第二行的输出正确，因为按照第一行的输出为 256，则在自增后，变量的值应该为 257。第三行的输出不正确，因为第二行输出 257，则在自减后，变量的值为 256。第四行的输出正确，因为第三行的输出为 255，变量的值不变，因此第四行输出 255。总的得分为 2 分。

13252 Rotating Drum。常规的解题方法是使用 Hierholzer 算法构建目标序列，在遍历顶点的出边时采用从小到大的顺序取用边，这样就能够保证得到的目标序列具有最小的字典序。具体编码时需要使用适当的数据结构来高效地表示图，否则容易超时。此问题还可使用其他更为巧妙的方法来解题，例如利用 Burrows-Wheeler 逆变换和置换的循环来进行求解，具体细节可参阅：https://en.wikipedia.org/wiki/De_Bruijn_sequence，2020。

4 习题索引

- 100 The $3n+1$ Problem, 633
- 101 The Blocks Problem, 163
- 102 Ecological Bin Packing, 417
- 103 Stacking Boxes, 721
- 104 Arbitrage, 551
- 105 The Skyline Problem, 823
- 106 Fermat vs. Pythagoras, 772
- 107 The Cat in the Hat, 368
- 108 Maximum Sum, 637
- 109 SCUD Busters, 844
- 110 Meta-Loopless Sorts, 216
- 111 History Grading, 720
- 112 Tree Summing, 68
- 113 Power of Cryptography, 282
- 114 Simulation Wizardry, 743
- 115 Climbing Trees, 70
- 116 Unidirectional TSP, 675
- 117 The Postal Worker Rings Once, 523
- 118 Mutant Flatworld Explorers, 743
- 119 Greedy Gift Givers, 58
- 120 Stacks of Flapjacks, 213
- 121 Pipe Fitters, 797
- 122 Trees on the Level, 70
- 123 Searching Quickly, 161
- 124 Following Orders, 418
- 125 Numbering Paths, 552
- 126 The Errant Physicist, 274
- 127 Accordian Patience, 40
- 128 Software CRC, 386
- 129 Krypton Factor, 157
- 130 Roman Roulette, 35
- 131 The Psychic Poker Player, 42
- 132 Bumpy Objects, 778
- 133 The Dole Queue, 38
- 134 Loglan - A Logical Language, 165
- 135 No Rectangles, 743
- 136 Ugly Numbers, 50
- 137 Polygons, 856
- 138 Street Number, 285
- 139 Telephone Tangles, 160
- 140 Bandwith, 210
- 141 The Spot Game, 55
- 142 Mouse Clicks, 761
- 143 Orchard Trees, 815
- 144 Student Grants, 47
- 145 Gondwanaland Telecom, 161
- 146 ID Codes, 209
- 147 Dollars, 734
- 148 Anagram Checker, 414
- 149 Forest, 795
- 150 Double Time, 397
- 151 Power Crisis, 38
- 152 Tree's a Crowd, 860
- 153 Permalex, 293
- 154 Recycling, 161
- 155 All Squares, 743
- 156 Ananagrams, 59
- 157 Route Finding, 540
- 158 Calader, 239
- 159 Word Crosses, 157
- 160 Factors and Factorials, 360
- 161 Traffic Lights, 52
- 162 Beggar My Neighbour, 41
- 163 City Directions, 743
- 164 String Computer, 713
- 165 Stamps, 414
- 166 Making Change, 734
- 167 The Sultan's Successor, 406
- 168 Theseus and the Minotaur, 449
- 169 Xenosemantics, 162
- 170 Clock Patience, 41
- 171 Car Trialling, 168
- 172 Calculator Language, 46
- 173 Network Wars, 450
- 174 Strategy, 168
- 175 Keywords, 165
- 176 City Navigation, 456
- 177 Paper Folding, 162
- 178 Shuffling Patience, 41
- 179 Code Breaking, 414
- 180 Eeny Meeny, 38
- 181 Hearts, 240
- 182 Bonus Bonds, 287
- 183 Bit Maps, 162
- 184 Laser Lines, 762
- 185 Roman Numerals, 267

- 186 Trip Routing, 540
187 Transaction Processing, 161
188 Perfect Hash, 385
189 Pascal Program Lengths, 165
190 Circle Through Three Points, 786
191 Intersection, 810
192 Synchronous Design, 502
193 Graph Coloring, 414
194 Triangle, 776
195 Anagram, 210
196 Spreadsheet, 58
197 Cube, 771
198 Peter's Calculator, 46
199 Partial Differential ..., 32
200 Rare Order, 501
201 Squares, 743
202 Repeating Decimals, 271
203 Running Lights Visibility ..., 774
205 Getting There, 414
206 Meals on Wheels Routing ..., 781
207 PGA Tour Prize Money, 162
208 Firetruck, 552
209 Triangular Vertices, 756
210 Concurrent Simulator, 54
211 The Domino Effect, 414
212 Use of Hospital Facilities, 52
213 Message Decoding, 65
214 Code Generation, 46
215 Spreadsheet Calculator, 61
216 Getting in Line, 525
217 Radio Direction Finder, 763
218 Moth Eradication, 839
219 Department of Redundancy ..., 422
220 Othello, 34
221 Urban Elevations, 827
222 Budget Travel, 429
224 Kissin' Cousins, 456
225 Polygons, 414
226 MIDI Preprocessing, 241
227 Puzzle, 743
228 Resource Allocation, 422
229 Scanner, 429
230 Borrowers, 162
231 Testing the CATCHER, 721
232 Crossword Answers, 34
233 Package Pricing, 629
234 Switching Channels, 210
238 Jill's Bike, 540
239 Time and Motion, 300
240 Variable Radix Huffman ..., 738
242 Stamps and Envelope Size, 632
243 Theseus and the Minotaur ..., 450
244 Train Time, 32
245 Uncompress, 66
246 10-20-30, 61
247 Calling Circles, 483
248 Cutting Corners, 819
249 Bang the Drum Slowly, 635
250 Pattern Matching Prelims, 34
253 Cube Painting, 299
254 Towers of Hanoi, 246
255 Correct Move, 61
256 Quirksome Squares, 384
257 Palinwords, 727
258 Mirror Maze, 416
259 Software Allocation, 407
260 Il Gioco dell'X, 743
261 The Window Property, 61
262 Transferable Voting, 162
263 Number Chains, 161
264 Count on Cantor, 269
265 Dining Diplomats, 414
266 Stamping Out Stamps, 734
267 Of(f) Course, 778
268 Double Trouble, 248
269 Counting Patterns, 422
270 Lining Up, 806
271 Simply Syntax, 168
272 TEX Quotes, 19
273 Jack Straws, 810
274 Cat and Mouse, 658
275 Expanding Fractions, 271
276 Egyptian Multiplication, 267
277 Cabinets, 20
278 Chess, 748
279 Spin, 32
280 Vertex, 456
282 Rename, 157
283 Compress, 414
284 Logic, 459

- 285 Crosswords, 34
 286 Dead Or Not-That Is The ..., 747
 288 Arithmetic Operations With ..., 252
 289 A Very Nasty Text Formatter, 66
 290 Palindroms <---> smordnilaP, 250
 291 The House Of Santa Claus, 459
 292 Presentation Error, 169
 293 Bits, 162
 294 Divisors, 368
 295 Fatman, 461
 296 Safebreaker, 416
 297 Quadtrees, 70
 298 Race Tracks, 456
 299 Train Swapping, 213
 300 Maya Calendar, 397
 301 Transportation, 422
 302 John's Trip, 509
 303 Pipe, 763
 304 Department, 52
 305 Joseph, 38
 306 Cipher, 300
 307 Sticks, 419
 308 Tin Cutter, 785
 309 FORCAL, 165
 310 L-System, 61
 311 Packets, 785
 312 Crosswords (II), 746
 313 Intervals, 790
 314 Robot, 456
 315 Network, 478
 316 Stars, 771
 317 Hexagon, 758
 318 Domino Effect, 544
 319 Pendulum, 806
 320 Border, 743
 321 The New Villa, 455
 323 Jury Compromise, 696
 324 Factorial Frequencies, 252
 325 Identifying Legal Pascal ..., 205
 326 Extrapolation Using a ..., 315
 327 Evaluating Simple C ..., 46
 328 The Finite State ..., 163
 329 PostScript Emulation, 767
 330 Inventory Maintenance, 163
 331 Mapping the Swaps, 213
 332 Rational Numbers From ..., 272
 333 Recognizing Good ISBNs, 162
 334 Identifying Concurrent ..., 554
 335 Processing MX Records, 58
 336 A Node Too Far, 552
 337 Interpreting Control ..., 163
 338 Long Multiplication, 162
 339 SameGame Simulation, 34
 340 Master-Mind Hints, 58
 341 Non-Stop Travel, 540
 342 HTML Syntax Checking, 168
 343 What Base Is This, 265
 344 Roman Digititis, 267
 345 It's Ir-Resist-Able!, 277
 346 Getting Chorded, 163
 347 Run Run Runaround Numbers, 414
 348 Optimal Array Multiplication ..., 667
 349 Transferable Voting (II), 162
 350 Pseudo-Random Number, 58
 352 Seasonal War, 746
 353 Pesky Palindromes, 724
 355 The Bases Are Loaded, 265
 356 Square Pegs And Round Holes, 786
 357 Let Me Count The Ways, 736
 358 Don't Have A Cow Dude, 226
 359 Sex Assignments and Breeding ..., 496
 360 Don't Get Hives From This ..., 758
 361 Cops and Robbers, 815
 362 18000 Seconds Remaining, 162
 363 Approximate Matches, 717
 365 Welfare Reform, 162
 367 Halting Factor Replacement ..., 252
 368 Indexing Web Pages, 454
 369 Combinations, 360
 370 Bingo, 162
 371 Ackermann Functions, 635
 372 WhatFix Notation, 72
 373 Romulan Spelling, 162
 374 Big Mod, 387
 375 Inscribed Circles and ..., 790
 376 More Triangles THE AMBIGUOUS ..., 778
 377 Cowculations, 262
 378 Intersecting Lines, 763
 379 Hi-Q, 34
 380 Call Forwarding, 58

- 381 Making the Grade, 161
 382 Perfection, 384
 383 Shipping Routes, 552
 384 Slurpys, 168
 385 DNA Translation, 162
 386 Perfect Cubes, 285
 387 A Puzzling Problem, 414
 388 Galactic Import, 544
 389 Basically Speaking, 265
 390 Letter Sequence Analysis, 240
 391 Mark-Up, 16
 392 Polynomial Showdown, 162
 393 The Doors, 810
 394 Mapmaker, 58
 395 Board Silly, 34
 396 Top Dog, 162
 397 Equation Elation, 162
 398 18-Wheeler Caravans (aka ..., 162
 399 Another Puzzling Problem, 414
 400 Unix ls, 162
 401 Palindromes, 723
 402 M*A*S*H, 38
 403 Postscript, 162
 405 Message Routing, 58
 406 Prime Cuts, 236
 407 Gears on a Board, 454
 408 Uniform Generator, 373
 409 Excuses Excuses, 61
 410 Station Balance, 729
 411 Centipede Collisions, 743
 412 Pi, 373
 413 Up and Down Sequences, 315
 414 Machined Surfaces, 145
 415 Sunrise, 788
 416 LED Test, 414
 417 Word Index, 49
 418 Molecules, 417
 422 Word-Search Wonder, 169
 423 MPI Maelstrom, 552
 424 Integer Inquiry, 250
 425 Enigmatic Encryption, 163
 426 Fifth Bank of Swamp County, 162
 427 Flatland Piano Movers, 227
 428 Swamp County Roofs, 162
 429 Word Transformation, 462
 430 Swamp County Supervisors, 632
 431 Trial of the Millennium, 629
 432 Modern Art, 38
 433 Bank (Not Quite O.C.R.), 163
 434 Matty's Blocks, 739
 435 Block Voting, 632
 436 Arbitrage (II), 552
 437 The Tower of Babylon, 422
 438 The Circumference of the ..., 791
 439 Knight Moves, 456
 440 Eeny Meeny Moo, 38
 441 Lotto, 414
 442 Matrix Chain Multiplication, 665
 443 Humble Numbers, 52
 444 Encoder and Decoder, 162
 445 Marvelous Mazes, 160
 446 Kibbles "n" Bits "n" Bits ..., 65
 447 Population Explosion, 32
 448 OOPS, 163
 449 Majoring in Scales, 162
 450 Little Black Book, 20
 451 Poker Solitaire Evaluator, 41
 452 Project Scheduling, 502
 453 Intersecting Circles, 797
 454 Anagrams, 240
 455 Periodic Strings, 159
 456 Robotic Stacker, 730
 457 Linear Cellular Automata, 31
 458 The Decoder, 153
 459 Graph Connectivity, 124
 460 Overlapping Rectangles, 785
 462 Bridge Hand Evaluator, 161
 464 Sentence/Phrase Generator, 459
 465 Overflow, 15
 466 Mirror Mirror, 34
 467 Synching Signals, 52
 468 Key to Success, 162
 469 Wetlands of Florida, 746
 471 Magic Numbers, 417
 473 Raucous Rockers, 648
 474 Heads/Tails Probability, 285
 475 Wild Thing, 169
 476 Points in Figures: ..., 815
 477 Points in Figures: ..., 815
 478 Points in Figures: ..., 815

- 481 What Goes Up, 721
 482 Permutation Arrays, 32
 483 Word Scramble, 211
 484 The Department of Redundancy ..., 58
 485 Pascal's Triangle of Death, 252
 486 English-Number Translator, 162
 487 Boggle Blitz, 454
 488 Triangle Wave, 155
 489 Hangman Judge, 61
 490 Rotating Sentences, 157
 492 Pig-Latin, 19
 493 Rational Spiral, 269
 494 Kindergarten Counting Game, 205
 495 Fibonacci Freeze, 252
 496 Simply Subsets, 61
 497 Strategic Defense Initiative, 721
 498 Polly the Polynomial, 274
 499 What's The Frequency Kenneth, 148
 500 Table, 162
 501 Black Box, 237
 502 DEL Command, 165
 503 Parallelepiped Walk, 876
 505 Moscow Time, 393
 506 System Dependencies, 58
 507 Jill Rides Again, 728
 508 Morse Mismatches, 156
 509 RAID, 161
 511 Do You Know the Way to San ..., 240
 512 Spreadsheet Tracking, 163
 514 Rails, 46
 515 King, 556
 516 Prime Land, 368
 517 Word, 162
 518 Time, 393
 519 Puzzle (II), 422
 520 Append, 66
 521 Gossiping, 553
 522 Schedule Problem, 558
 523 Minimum Transport Cost, 552
 524 Prime Ring Problem, 407
 525 Milk Bottle Data, 429
 526 String Distance and ..., 715
 527 The Partition of a Cake, 851
 529 Addition Chains, 442
 530 Binomial Showdown, 295
 531 Compromise, 717
 532 Dungeon Master, 456
 533 Equation Solver, 46
 534 Frogger, 555
 535 Globetrotter, 759
 536 Tree Recovery, 72
 537 Artificial Intelligence, 163
 538 Balancing Bank Accounts, 161
 539 The Settlers of Catan, 463
 540 Team Queue, 49
 541 Error Correction, 32
 542 France '98, 329
 543 Goldbach's Conjecture, 361
 544 Heavy Cargo, 555
 545 Heads, 285
 547 DDF, 635
 548 Tree, 72
 549 Evaluating an Equations ..., 418
 550 Multiplying by Rotation, 387
 551 Nesting a Bunch of Brackets, 46
 552 Filling the Gaps, 407
 553 Simply Proportion, 157
 554 Caesar Cypher, 163
 555 Bridge Hands, 240
 556 Amazing, 743
 557 Burge, 336
 558 Wormholes, 547
 559 Square (II), 637
 560 Magic, 429
 561 Jackpot, 331
 562 Dividing Coins, 629
 563 Crimewave, 569
 565 Pizza Anyone, 65
 567 Risk, 552
 568 Just the Facts, 387
 570 Stats, 27
 571 Jugs, 373
 572 Oil Deposits, 746
 573 The Snail, 17
 574 Sum It Up, 417
 575 Skew Binary, 262
 576 Haiku Review, 20
 577 WIMP, 163
 578 Polygon Puzzler, 876
 579 Clock Hands, 399

- 580 Critical Mass, 326
581 Word Search Wonder, 169
583 Prime Factors, 368
584 Bowling, 161
585 Triangles, 637
586 Instant Complexity, 46
587 There's Treasure Everywhere, 743
588 Video Surveillance, 856
589 Pushing Boxes, 462
590 Always on the Run, 679
591 Box of Bricks, 17
592 Island of Logic, 414
593 MBone, 552
594 One Little Two Little Three ..., 11
596 The Incredible Hull, 842
598 Bundling Newspapers, 417
599 The Forrest for the Trees, 124
601 The PATH, 746
602 What Day Is It, 397
603 Parking Lot, 42
604 The Boggle Game, 414
607 Scheduling Lectures, 650
608 Counterfeit Dollar, 416
609 Metal Cutting, 763
610 Street Directions, 478
612 DNA Sorting, 241
613 Numbers That Count, 163
614 Mapping the Route, 459
615 Is It a Tree, 67
616 Coconuts Revisited, 384
617 Nonstop Travel, 285
618 Doing Windows, 417
619 Numerically Speaking, 250
620 Cellular Structure, 168
621 Secret Research, 157
622 Grammar Evaluation, 46
623 500!, 252
624 CD, 422
625 Compression, 157
626 Ecosystem, 32
627 The Net, 456
628 Passwords, 417
630 Anagrams (II), 240
631 Microzoft Calendar, 397
632 Compression (II), 241
633 A Chess Knight, 462
634 Polygon, 819
635 Clock Solitaire, 162
636 Squares (III), 262
637 Booklet Printing, 162
638 Finding Rectangles, 240
639 Don't Get Rooked, 748
640 Self Numbers, 368
641 Do the Untwist, 387
642 Word Amalgamation, 162
644 Immediate Decodability, 159
645 File Mapping, 162
647 Chutes and Ladders, 34
648 Stamps, 418
651 Deck, 325
652 Eight, 442
653 Gizilch, 407
654 Ratio, 269
655 Scrabble, 161
656 Optimal Programs, 462
657 The Die is Cast, 746
658 It's Not a Bug It's a ..., 541
661 Blowing Fuses, 161
662 Fast Food, 675
663 Sorting Slides, 587
665 False Coin, 61
668 Parliament, 730
670 The Dog Task, 595
671 Spell Checker, 715
672 Gangsters, 650
673 Parentheses Balance, 46
674 Coin Change, 736
675 Convex Hull of the Polygon, 842
676 Horse Step Maze, 461
677 All Walks of Length n From ..., 407
679 Dropping Balls, 71
681 Convex Hull Finding, 844
682 Whoever Pick The Last One ..., 645
684 Integral Determinant, 281
685 Least Path Cost, 463
686 Goldbach's Conjecture (II), 366
688 Mobile Phone Coverage, 823
689 Napoleon's Grumble, 727
694 The Collatz Sequence, 635
696 How Many Knights, 748

- 697 Jack and Jill, 273
 698 Index, 58
 699 The Falling Leaves, 70
 700 Date Bugs, 387
 701 The Archeologist's Dilemma, 281
 702 The Vindictive Coach, 696
 703 Triple Ties: The Organizer's ..., 32
 704 Color Hash, 454
 705 Slash Maze, 746
 706 LC-Display, 23
 707 Robbery, 419
 709 Formatting Text, 650
 710 The Game, 416
 711 Dividing Up, 700
 712 S-Trees, 71
 713 Adding Reversed Numbers, 245
 714 Copying Books, 227
 718 Skyscraper Floors, 460
 719 Glass Beads, 179
 721 Invitation Cards, 541
 722 Lakes, 746
 725 Division, 417
 726 Decode, 163
 727 Equation, 43
 729 The Hamming Distance Problem, 210
 732 Anagrams by Stack, 46
 735 Dart-a-Mania, 416
 736 Lost in Space, 169
 737 Gleaming the Cubes, 785
 739 Soundex Indexing, 157
 740 Baudot Data Communication ..., 65
 741 Burrows Wheeler Decoder, 202
 743 The MTM Machine, 168
 748 Exponentiation, 247
 750 8 Queens Chess Problem, 407
 751 Triangle War, 645
 753 A Plug for UNIX, 588
 755 487-3279, 214
 756 Biorhythms, 389
 757 Gone Fishing, 650
 758 The Same Game, 746
 759 The Return of the Roman ..., 267
 760 DNA Sequencing, 200
 762 We Ship Cheap, 462
 763 Fibinary Numbers, 319
 775 Hamiltonian Cycle, 525
 776 Monkeys in a Regular Forest, 746
 782 Contour Painting, 746
 784 Maze Exploration, 746
 785 Grid Colouring, 744
 787 Maximum Sub-Sequence Product, 729
 789 Indexing, 61
 790 Head Judge Headache, 162
 793 Network Connections, 124
 795 Sandorf's Cipher, 163
 796 Critical Links, 478
 800 Crystal Clear, 846
 801 Flight Planning, 650
 808 Bee Breeding, 758
 811 The Fortified Forest, 839
 812 Trade on Verweggistan, 739
 815 Flooded, 32
 816 Abbott's Revenge, 456
 820 Internet Bandwidth, 566
 821 Page Hopping, 658
 824 Coast Tracker, 743
 825 Walking on the Safe Side, 679
 828 Deciphering Messages, 163
 830 Shark, 746
 833 Water Falls, 806
 834 Continued Fractions, 270
 835 Square of Primes, 429
 836 Largest Submatrix, 637
 837 Light and Transparencies, 800
 839 Not so Mobile, 70
 840 Deadlock Detection, 459
 843 Crypt Kicker, 414
 844 Pousse, 34
 846 Steps, 284
 847 A Multiplication Game, 355
 848 Fmt, 162
 849 Radar Tracking, 772
 850 Crypt Kicker II, 169
 852 Deciding Victory in Go, 746
 855 Lunch in Grid City, 216
 856 The Vigenère Cipher, 163
 857 Quantiser, 162
 858 Berry Picking, 815
 859 Chinese Checkers, 456
 860 Entropy Text Analyzer, 285

- 861 Little Bishops, 747
865 Substitution Cypher, 157
866 Intersecting Line Segments, 810
868 Numerical Maze, 414
869 Airline Comparison, 554
870 Intersecting Rectangles, 827
871 Counting Cells in a Blob, 746
872 Ordering, 418
877 Offset Polygons, 812
880 Cantor Fractions, 285
882 The Mailbox Manufacturers ..., 652
884 Factorial Factors, 368
886 Named Extension Dialing, 169
888 Donkey, 691
889 Islands, 814
890 Maze (II), 162
892 Finding Words, 157
893 Y3K Problem, 394
895 Word Problem, 18
897 Anagrammatic Primes, 367
899 Colour Circles, 456
900 Brick Wall Patterns, 317
902 Password Search, 264
903 Spiral of Numbers, 32
904 Overlapping Air Traffic ..., 827
905 Tacos Panchita, 762
906 Rational Neighbor, 269
907 Winterim Backpacking Trip, 227
908 Re-Connecting Computer Sites, 529
909 The BitPack Data Compression ..., 695
910 TV Game, 679
911 Multinomial Coefficients, 296
912 Live From Mars, 165
913 Joana and the Odd Numbers, 15
914 Jumping Champion, 233
915 Stack of Cylinders, 797
918 ASCII Mandelbrot, 273
920 Sunny Mountains, 761
921 A Word Puzzle in the Sunny ..., 417
922 Rectangle by the Ocean, 847
924 Spreading the News, 453
925 No More Prerequisites Please, 554
926 Walking Around Wisely, 679
927 Integer Sequences from ..., 273
928 Eternal Truths, 456
929 Number Maze, 541
930 Polynomial Roots, 274
932 Checking the N-Queens ..., 407
933 Water Flow, 161
934 Overlapping Areas, 827
939 Genes, 70
941 Permutations, 295
942 Cyclic Numbers, 271
943 Number Format Translator, 267
944 Happy Numbers, 635
945 Loading a Cargo Ship, 739
946 A Pile of Boxes, 32
947 Master Mind Helper, 417
948 Fibonaccimal Base, 319
949 Getaway, 456
950 Tweedle Numbers, 679
957 Popes, 52
959 Car Rallying, 650
960 Gaussian Primes, 364
962 Taxicab Numbers, 58
963 Spelling Corrector, 715
964 Custom Language, 163
967 Circular, 367
970 Particles, 675
972 Horizon Line, 800
974 Kaprekar Numbers, 384
976 Bridge Building, 679
978 Lemmings Battle, 52
979 The Abominable Triangleman, 771
980 X-Express, 739
983 Localized Summing for ..., 637
985 Round and Round Maze, 456
986 How Many, 652
988 Many Paths One Destination, 459
989 Su Doku, 407
990 Diving for Gold, 628
991 Safe Salutations, 324
993 Product of Digits, 360
995 Super Divisible Numbers, 248
996 Find the Sequence, 414
997 Show the Sequence, 250
999 Book Signatures, 54
1006 Fixed Partition Memory ..., 608
1013 Island Hopping, 529
1025 A Spy in the Metro, 545

- 1027 Toll, 545
 1039 Simplified GSM Network, 814
 1040 The Traveling Judges Problem, 529
 1045 The Great Wall Game, 612
 1047 Zones, 314
 1051 Bipartite Numbers, 416
 1052 Bit Compressor, 414
 1056 Degrees of Separation, 552
 1057 Routing, 545
 1061 Consanguine Calculations, 165
 1062 Containers, 730
 1064 Network, 52
 1069 Always an Integer, 385
 1076 Password Suspects, 665
 1079 A Careful Approach, 417
 1091 Barcodes, 163
 1092 Tracking Bio-Bots, 827
 1093 Castles, 739
 1096 The Islands, 689
 1098 Robot on Ice, 422
 1099 Sharing Chocolate, 699
 1103 Ancient Messages, 746
 1105 Coffee Central, 766
 1111 Trash Removal, 813
 1112 Mice and Maze, 540
 1121 Subsequence, 54
 1124 Celebrity Jeopardy, 18
 1146 Now or Later, 496
 1148 The Mysterious X Network, 462
 1151 Buy or Build, 527
 1152 4 Values Whose Sum is 0, 236
 1153 Keep the Customer Satisfied, 730
 1160 X-Plosives, 125
 1169 Robotruck, 701
 1172 The Bridges of Kölberg, 693
 1174 IP-TV, 529
 1175 Ladies' Choice, 602
 1176 A Benevolent Josephus, 38
 1180 Perfect Numbers, 370
 1184 Air Raid, 621
 1185 Big Number, 289
 1193 Radar Installation, 800
 1194 Machine Schedule, 617
 1195 Calling Extraterrestrial ..., 366
 1196 Tiling Up Blocks, 721
 1197 The Suspects, 124
 1198 The Geodetic Set Problem, 552
 1200 A DP Problem, 165
 1201 Taxi Cab Scheme, 621
 1202 Finding Nemo, 545
 1203 Argus, 52
 1205 Color a Tree, 739
 1206 Boundary Points, 839
 1207 AGTC, 715
 1208 Oreon, 527
 1209 Wordfish, 210
 1210 Sum of Consecutive Prime ..., 366
 1211 Atomic Car Race, 675
 1212 Duopoly, 582
 1213 Sum of Different Primes, 736
 1215 String Cutting, 162
 1216 The Bug Sensor Problem, 529
 1217 Route Planning, 429
 1218 Perfect Service, 684
 1219 Team Arrangement, 241
 1220 Party at Hali-Bula, 685
 1221 Against Mammoths, 595
 1222 Bribing FIPA, 681
 1223 Editor, 200
 1224 Tile Code, 326
 1225 Digit Counting, 384
 1226 Numerical Surprises, 248
 1229 Sub-Dictionary, 490
 1230 MODEX, 385
 1231 ACORN, 700
 1232 SKYLINE, 90
 1233 USHER, 545
 1234 RACING, 529
 1235 Anti Brute Force Lock, 527
 1237 Expert Enough, 233
 1238 Free Parentheses, 696
 1239 Greatest K-Palindrome ..., 724
 1240 ICPC Team Strategy, 665
 1241 Jollybee Tournament, 63
 1242 Necklace, 571
 1243 Polynomial-Time Reductions, 554
 1244 Palindromic Paths, 700
 1246 Find Terrorists, 368
 1247 Interstar Transport, 552
 1249 Euclid, 780

1250 Robot Challenge, 650	1482 Playing With Stones, 349
1251 Repeated Substitution with ..., 456	1494 Qin Shi Huang's National ..., 537
1252 Twenty Questions, 665	1513 Movie Collection, 97
1253 Infected Land, 456	1515 Pool Construction, 582
1254 Top 10, 200	1537 Picnic Planning, 532
1258 Nowhere Money, 320	1557 Calendar Game, 645
1260 Sales, 32	1558 Number Game, 645
1261 String Popping, 635	1559 Nim, 645
1262 Password, 414	1560 Extended Lights Out, 281
1263 Mines, 459	1561 Cycle Game, 645
1265 Tour Belt, 529	1566 John, 348
1266 Magic Square, 411	1567 A Simple Stone Game, 358
1280 Curvy Little Bottles, 850	1571 How I Mathematician Wonder ..., 856
1281 Bus Tour, 686	1583 Digit Generator, 384
1292 Strategic Game, 623	1584 Circular Sequence, 179
1304 Art Gallery, 856	1585 Score, 161
1309 Sudoku, 407	1586 Molar Mass, 161
1310 One-Way Traffic, 481	1587 Box, 785
1312 Cricket Field, 827	1588 Kickdown, 169
1315 Crazy Tea Party, 32	1589 Xiangqi, 748
1329 Corporative Network, 124	1593 Alignment of Code, 19
1330 City Game, 640	1594 Ducci Sequence, 61
1338 Crossing Prisms, 850	1595 Symmetry, 761
1339 Ancient Cipher, 163	1600 Patrol Robot, 677
1342 That Nice Euler Circuit, 810	1601 The Morning after Halloween, 456
1347 Tour, 689	1605 Building for UN, 162
1349 Optimal Bus Route Design, 616	1610 Party Games, 240
1368 DNA Consensus String, 161	1613 K-Graph Oddity, 464
1378 A Funny Stone Game, 354	1636 Headshot, 329
1382 Distant Galaxy, 823	1639 Candy, 284
1388 Graveyard, 788	1641 ASCII Area, 815
1391 Astronauts, 496	1642 Magical GCD, 373
1395 Slim Span, 529	1644 Prime Gap, 367
1399 Puzzle, 679	1646 Edge Case, 317
1400 Ray Pass Me the Dishes, 90	1647 Computer Transformation, 252
1411 Ants, 616	1648 Business Center, 370
1415 Gauss Prime, 364	1663 Purifying Machine, 600
1427 Parade, 704	1709 Amalgamated Artichokes, 781
1438 Asteroids, 879	1721 Window Manager, 785
1449 Dominating Patterns, 188	1723 Intervals, 558
1451 Average, 708	1727 Counting Weekend Days, 58
1453 Squares, 865	1738 Ceiling Function, 70
1456 Cellular Network, 691	1749 Airport Construction, 818
1476 Error Curves, 230	1753 Need for Speed, 226
1478 Delta Wave, 324	1757 Secret Chamber at Mount ..., 554

- 10000 Longest Path, 463
 10001 Garden of Eden, 414
 10002 Center of Masses, 849
 10003 Cutting Sticks, 712
 10004 Bicoloring, 464
 10005 Packing Polygons, 798
 10006 Carmichael Numbers, 362
 10007 Count the Trees, 324
 10008 What's Cryptanalysis, 161
 10009 All Roads Lead Where, 462
 10010 Where's Waldorf, 169
 10011 Where Can You Hide, 821
 10012 How Big Is It, 797
 10013 Super Long Sums, 245
 10014 Simple Calculations, 315
 10015 Joseph's Cousin, 366
 10016 Flip-Flop the Squarelotron, 34
 10017 The Never Ending Towers of ..., 162
 10018 Reverse and Add, 245
 10019 Funny Encryption Method, 65
 10020 Minimal Coverage, 730
 10021 Cube in the Labirint, 456
 10022 Delta-wave, 758
 10023 Square Root, 253
 10025 The ? 1 ? 2 ? ... ? n = k ..., 315
 10026 Shoemaker's Problem, 738
 10028 Demerit Points, 397
 10029 Edit Step Ladders, 463
 10032 Tug of War, 660
 10033 Interpreter, 163
 10034 Freckles, 527
 10035 Primary Arithmetic, 245
 10036 Divisibility, 632
 10037 Bridge, 739
 10038 Jolly Jumper, 32
 10039 Railroads, 454
 10040 Ouroboros Snake, 519
 10041 Vito's Family, 215
 10042 Smith Numbers, 361
 10043 Chainsaw Massacre, 827
 10044 Erdős Numbers, 454
 10047 The Monocycle, 679
 10048 Audiophobia, 555
 10049 Self-Describing Sequence, 234
 10050 Hartals, 384
 10051 Tower of Cubes, 463
 10054 The Necklace, 522
 10055 Hashmat the Brave Warrior, 17
 10056 What is the Probability, 331
 10057 A Mid-Summer Night's Dream, 220
 10058 Jimmy's Riddles, 168
 10060 A Hole to Catch a Man, 846
 10061 How Many Zero's and How Many ..., 368
 10062 Tell Me the Frequencies, 161
 10063 Knuth's Permutation, 414
 10065 Useless Tile Packers, 846
 10066 The Twin Towers, 717
 10067 Playing with Wheels, 454
 10068 The Treasure Hunt, 462
 10069 Distinct Subsequences, 715
 10070 Leap Year or Not Leap Year, 393
 10071 Back to High School Physics, 17
 10072 Bob Laptop Woolmer and Eddie ..., 629
 10073 Constrained Exchange Sort, 442
 10074 Take the Land, 640
 10075 Airlines, 759
 10077 The Stern-Brocot Number ..., 268
 10078 The Art Gallery, 856
 10079 Pizza Cutting, 315
 10080 Gopher II, 588
 10081 Tight Words, 695
 10082 WERTYU, 153
 10083 Division, 252
 10084 Hotter Colder, 856
 10085 The Most Distant State, 456
 10086 Test the Rods, 695
 10087 The Tajmahal of ++Y2k, 411
 10088 Trees on My Island, 846
 10089 Repackaging, 803
 10090 Marbles, 375
 10091 The Valentine's Day, 691
 10092 The Problem With the Problem ..., 571
 10093 An Easy Problem, 262
 10094 Place the Guards, 411
 10097 The Color Game, 456
 10098 Generating Fast Sorted ..., 210
 10099 The Tourist Guide, 555
 10100 Longest Match, 717
 10101 Bangla Numbers, 267
 10102 The Path in the Colored ..., 743

- 10104 Euclid Problem, 376
10105 Polynomial Coefficients, 273
10106 Product, 252
10107 What is the Median, 216
10109 Solving Systems of Linear ..., 281
10110 Light More Light, 360
10111 Find the Winning Move, 642
10112 Myacm Triangles, 815
10113 Exchange Rates, 462
10114 Loansome Car Buyer, 14
10115 Automatic Editing, 159
10116 Robot Motion, 743
10117 Nice Milk, 856
10118 Free Candies, 700
10120 Gift?!, 456
10122 Mysterious Mountain, 600
10123 No Tipping, 665
10125 Sumsets, 233
10126 Zipf's Law, 161
10127 Ones, 385
10128 Queue, 695
10129 Play on Words, 507
10130 SuperSale, 629
10131 Is Bigger Smarter, 721
10132 File Fragmentation, 169
10134 AutoFish, 163
10135 Herding Frosh, 838
10136 Chocolate Chip Cookies, 790
10137 The Trip, 26
10138 CDVII, 162
10139 Factovisors, 373
10140 Prime Distance, 366
10141 Request for Proposal, 18
10142 Australian Voting, 162
10145 Lock Manager, 58
10146 Dictionary, 162
10147 Highways, 529
10149 Yahtzee, 662
10150 Doublets, 462
10152 ShellSort, 213
10154 Weights and Measures, 696
10157 Expressions, 323
10158 War, 124
10159 Star, 758
10160 Servicing Stations, 616
10161 Ant on a Chessboard, 743
10162 Last Digit, 385
10163 Storage Keepers, 629
10164 Number Game, 427
10165 Stone Game, 347
10166 Travel, 654
10167 Birthday Cake, 762
10168 Summation of Four Primes, 361
10169 Urn-Ball Probabilities, 329
10170 The Hotel with Infinite ..., 225
10171 Meeting Prof. Miguel, 544
10172 The Lonesome Cargo ..., 47
10173 Smallest Bounding Rectangle, 865
10174 Couple-Bachelor-Spinster ..., 384
10176 Ocean Deep Make it Shallow, 248
10177 (2/3/4)-D ..., 784
10178 Count the Faces, 445
10179 Irreducible Basic Fractions, 378
10180 Rope Crisis in Ropeland, 790
10181 15-Puzzle Problem, 432
10182 Bee Maja, 758
10183 How Many Fibs, 252
10184 Equidistance, 876
10186 Euro Cup 2000, 414
10187 From Dusk till Dawn, 454
10188 Automated Judge Script, 169
10189 Minesweeper, 742
10190 Divide But Not Quite Conquer, 370
10191 Longest Nap, 823
10192 Vacation, 717
10193 All You Need Is Love, 252
10194 Football (aka Soccer), 162
10195 The Knights of the Round ..., 778
10196 Check the Check, 747
10197 Learning Portuguese, 162
10198 Counting, 245
10199 Tourist Guide, 478
10200 Prime Time, 361
10201 Adventures in Moving: Part ..., 675
10202 Pairsumonious Numbers, 414
10203 Snow Clearing, 509
10205 Stack'em Up, 41
10206 Stars, 771
10207 The Unreal Tournament, 691
10209 Is This Integration?, 787

- 10210 Romeo & Juliet, 813
 10212 The Last Non-Zero Digit, 384
 10213 How Many Pieces of Land?, 251
 10215 The Largest/Smallest Box, 785
 10217 A Dinner with Schwarzenegger, 329
 10218 Let's Dance, 336
 10219 Find the Ways, 290
 10220 I Love Big Numbers, 262
 10221 Satellites, 788
 10222 Decode the Mad Man, 153
 10223 How Many Nodes, 324
 10226 Hardwood Species, 58
 10227 Forests, 65
 10229 Modular Fibonacci, 318
 10233 Dermuba Triangle, 758
 10235 Simply Emirp, 367
 10236 The Fibonacci Primes, 317
 10237 Bishops, 748
 10238 Throw the Dice, 633
 10239 The Book-Shelver's Problem, 652
 10242 Fourth Point, 761
 10243 Fire Fire Fire, 623
 10245 The Closest Pair Problem, 860
 10246 Asterix and Obelix, 541
 10247 Complete Tree Labeling, 261
 10249 The Grand Dinner, 571
 10250 The Other Two Trees, 785
 10252 Common Permutation, 169
 10254 The Priest Mathematician, 262
 10255 The Knight's Tour, 755
 10256 The Great Divide, 819
 10258 Contest Scoreboard, 240
 10259 Hippity Hopscotch, 463
 10260 Soundex, 32
 10261 Ferry Loading, 650
 10263 Railway, 811
 10264 The Most Potent Corner, 63
 10267 Graphical Editor, 746
 10268 498-Bis, 274
 10269 Adventure of Super Mario, 653
 10270 Bigger Square Please..., 429
 10271 Chopsticks, 695
 10276 Hanoi Tower Troubles Again, 730
 10278 Fire Station, 550
 10279 Mine Sweeper, 743
 10281 Average Speed, 27
 10282 Babelfish, 58
 10283 The Kissing Circles, 797
 10284 Chessboard in FEN, 747
 10285 Longest Run on a Snowboard, 463
 10286 Trouble with a Pentagon, 775
 10287 Gifts in a Hexagonal Box, 788
 10288 Coupons, 344
 10290 $\{\text{Sum}+=\text{i}++\}$ to Reach N, 368
 10293 Word Length and Frequency, 161
 10294 Arif in Dhaka, 310
 10295 Hay Points, 58
 10296 Jogging Trails, 523
 10297 Beavergnaw, 788
 10298 Power Strings, 169
 10299 Relatives, 378
 10300 Ecological Premium, 17
 10301 Rings and Glue, 795
 10302 Summation of Polynomials, 315
 10303 How Many Trees, 324
 10304 Optimal Binary Search Tree, 709
 10305 Ordering Tasks, 502
 10306 e-Coins, 736
 10307 Killing Aliens in Borg Maze, 529
 10308 Roads in the North, 498
 10309 Turn the Lights Off, 432
 10310 Dog and Gopher, 761
 10311 Goldbach and Euler, 366
 10312 Expression Bracketing, 324
 10313 Pay the Price, 736
 10315 Poker Hands, 41
 10316 Airline Hub, 759
 10318 Security Panel, 432
 10319 Manhattan, 492
 10321 Polygon Intersection, 819
 10323 Factorial You Must be ..., 289
 10324 Zeros and Ones, 727
 10325 The Lottery, 313
 10326 The Polynomial Equation, 273
 10327 Flip Sort, 213
 10328 Coin Toss, 652
 10330 Power Transmission, 581
 10331 The Flyover Construction, 552
 10333 The Tower of ASCII, 162
 10334 Ray Through Glasses, 317

- 10336 Rank the Languages, 746
10337 Flight Planner, 695
10338 Mischievous Children, 290
10339 Watching Watches, 398
10340 All in All, 223
10341 Solve It, 225
10342 Always Late, 561
10344 23 Out of 5, 414
10346 Peter's Smokes, 385
10347 Medians, 778
10349 Antenna Placement, 620
10350 Liftless EME, 540
10354 Avoiding Your Boss, 545
10356 Rough Roads, 654
10357 Playball, 761
10359 Tiling, 253
10360 Rat Attack, 743
10361 Automatic Poetry, 159
10363 Tic Tac Toe, 33
10364 Square, 425
10365 Blocks, 416
10368 Euclid's Game, 373
10369 Arctic Network, 529
10370 Above Average, 27
10371 Time Zones, 399
10372 Leaps Tall Buildings, 781
10374 Election, 162
10375 Choose and Divide, 290
10377 Maze Traversal, 743
10382 Watering Grass, 797
10385 Duathlon, 230
10387 Billiard, 783
10388 Snap, 157
10389 Subway, 540
10391 Compound Words, 61
10392 Factoring Large Numbers, 392
10393 The One-Handed Typist, 61
10394 Twin Primes, 366
10397 Connect the Campus, 529
10400 Game Show Math, 632
10401 Injured Queen Problem, 679
10404 Bachet's Game, 349
10405 Longest Common Subsequence, 717
10406 Cutting Tabletops, 812
10407 Simple Division, 372
10408 Farey Sequences, 269
10409 Die Game, 300
10415 Eb Alto Saxophone Player, 61
10419 Sum-Up the Primes, 427
10420 List of Conquests, 161
10422 Knights In FEN, 462
10424 Love Calculator, 384
10426 Knights' Nightmare, 747
10427 Naughty Sleepy Boys, 236
10430 Dear GOD, 250
10432 Polygon Inside A Circle, 785
10433 Automorphic Numbers, 253
10436 Cheapest Way, 540
10437 Playing with Fraction, 269
10440 Ferry Loading II, 730
10441 Catenyms, 513
10443 Rock Scissors Paper, 34
10445 Make Polygon, 847
10446 The Marriage Interview, 635
10447 Sum-Up the Primes (II), 427
10448 Unique World, 630
10449 Traffic, 547
10450 World Cup Noise, 317
10451 Ancient Village Sports, 788
10452 Marcus, 743
10453 Make Palindrome, 667
10457 Magic Car, 529
10459 The Tree Root, 498
10460 Find the Permuted String, 295
10461 Difference, 459
10462 Is There a Second Way Left, 536
10464 Big Big Real Numbers, 253
10465 Homer Simpson, 736
10466 How Far, 764
10469 To Carry or Not to Carry, 61
10473 Simple Base Conversion, 161
10474 Where is the Marble, 226
10475 Help the Leaders, 418
10477 The Hybrid Knight, 749
10480 Sabotage, 582
10482 The Candyman Can, 700
10483 The Sum Equals the Product, 416
10484 Divisibility of Factors, 368
10487 Closest Sums, 236
10489 Boxes of Chocolates, 385

- 10490 Mr. Azad and His Son, 366
 10491 Cows and Cars, 334
 10493 Cats With or Without Hats, 370
 10494 If We Were a Child Again, 253
 10496 Collecting beepers, 525
 10497 Sweet Child Makes Trouble, 315
 10499 The Land of Justice, 850
 10500 Robot Maps, 743
 10501 Simplified Shisen-Sho, 416
 10502 Counting Rectangles, 637
 10503 The Dominoes Solitaire, 412
 10505 Montesco vs Capuleto, 464
 10506 The Ouroboros Problem, 522
 10507 Waking up Brain, 450
 10508 Word Morphing, 162
 10509 R U Kidding Mr. Feynman, 282
 10511 Councilling, 571
 10513 Bangladesh Sequences, 407
 10514 River Crossing, 814
 10515 Powers Et Al., 385
 10518 How Many Calls, 318
 10519 Really Strange, 253
 10520 Determine It, 635
 10521 Continuously Growing ..., 270
 10522 Height to Area, 780
 10523 Very Easy, 253
 10525 New to Bangladesh, 545
 10526 Intellectual Property, 200
 10527 Persistent Numbers, 247
 10528 Major Scales, 232
 10529 Dumb Bones, 693
 10530 Guessing Game, 18
 10532 Combination Once Again, 652
 10533 Digit Primes, 367
 10534 Wavio Sequence, 721
 10536 Game of Euler, 645
 10537 The Toll Revisited, 544
 10539 Almost Prime Numbers, 366
 10541 Stripe, 296
 10543 Traveling Politician, 658
 10544 Numbering the Paths, 679
 10550 Combination Lock, 24
 10551 Basic Remains, 253
 10554 Calories from Fat, 161
 10555 Dead Fraction, 272
 10557 XYZZY, 547
 10559 Blocks, 673
 10561 Treblecross, 354
 10562 Undraw the Trees, 459
 10564 Paths Through the Hourglass, 679
 10566 Crossed Ladders, 763
 10567 Helping Fill Bases, 236
 10570 Meeting with Aliens, 416
 10573 Geometry Paradox, 797
 10576 Y2K Accounting Bug, 407
 10577 Bounding Box, 791
 10578 The Game of 31, 645
 10579 Fibonacci Numbers, 317
 10582 ASCII Labyrinth, 414
 10583 Ubiquitous Religions, 125
 10585 Center of Symmetry, 761
 10586 Polynomial Remains, 273
 10589 Area, 788
 10591 Happy Number, 384
 10592 Freedom Fighter, 746
 10594 Data Flow, 584
 10596 Morning Walk, 509
 10600 ACM Contest and Blackout, 536
 10601 Cube, 307
 10602 Editor Nottoobad, 165
 10603 Fill, 654
 10604 Chemical Reaction, 652
 10606 Opening Doors, 253
 10608 Friends, 125
 10610 Gopher And Hawks, 462
 10611 The Playboy Chimp, 236
 10615 Rooks, 591
 10616 Divisible Group Sums, 632
 10617 Again Palindromes, 727
 10620 A Flea on a Chessboard, 312
 10622 Perfect P-th Powers, 368
 10624 Super Number, 414
 10625 GNU = GNU'sNotUnix, 162
 10626 Buying Coke, 700
 10633 Rare Easy Problem, 370
 10635 Prince and Princess, 717
 10637 Coprimes, 414
 10642 Can You Solve It, 743
 10643 Facing Problem with Trees, 326
 10645 Menu, 630

- 10646 What is the Card, 41
10648 Chocolate Box, 691
10650 Determinate Prime, 361
10651 Pebble Solitaire, 635
10652 Board Wrapping, 844
10653 Bombs NO They Are Mines, 462
10655 Contemplation Algebra, 318
10656 Maximum Sum (II), 730
10659 Fitting Text into Slides, 162
10660 Citizen Attention Offices, 416
10662 The Wedding, 416
10664 Luggage, 632
10666 The Eurocup is Here, 65
10667 Largest Block, 640
10668 Expanding Rods, 788
10669 Three Powers, 250
10670 Work Reduction, 735
10672 Marbles on a Tree, 502
10673 Play with Floor and Ceil, 376
10674 Tangents, 821
10677 Base Equality, 265
10678 The Grazing Cow, 788
10679 I Love Strings, 175
10680 LCM, 376
10681 Teobaldo's Trip, 658
10682 Forró Party, 456
10683 The Decadary Watch, 399
10684 The Jackpots, 728
10685 Nature, 125
10686 SQF Problems, 61
10687 Monitoring the Amazon, 761
10688 The Poor Giant, 675
10689 Yet Another Number Sequence, 318
10690 Expression Again, 632
10693 Traffic Volume, 285
10696 f91, 635
10698 Football Sort, 162
10699 Count the factors, 361
10700 Camel Trading, 41
10701 Pre In and Post, 72
10702 Travelling Salesman, 679
10703 Free Spots, 34
10706 Number Sequence, 233
10707 2D-Nim, 746
10710 Chinese Shuffle, 385
10714 Ants, 730
10717 Mint, 377
10718 Bit Mask, 64
10719 Quotient Polynomial, 273
10720 Graph Construction, 505
10721 Bar Codes, 635
10722 Super Lucky Numbers, 695
10724 Road Construction, 552
10730 Antiarithmetic, 32
10731 Test, 490
10733 The Colored Cubes, 310
10734 Triangle Partitioning, 778
10735 Euler Circuit, 571
10738 Riemann vs Mertens, 384
10739 String to Palindrome, 667
10740 Not the Best, 561
10741 Magic Cube, 411
10742 The New Rule in Euphomia, 368
10746 Crime Wave The Sequel, 608
10747 Maximum Subsequence, 739
10748 Knight Roaming, 749
10751 Chessboard, 749
10755 Garbage Heap, 729
10759 Dice Throwing, 633
10761 Broken Keyboard, 162
10763 Foreign Exchange, 240
10765 Doves and Bombs, 478
10771 Barbarian Tribes, 38
10773 Back to Intermediate Math, 774
10774 Repeated Josephus, 38
10776 Determine The Combination, 418
10777 God! Save Me, 344
10779 Collector's Problem, 571
10780 Again Prime No Time, 368
10783 Odd Sum, 17
10784 Diagonal, 285
10785 The Mad Numerologist, 730
10789 Prime Frequency, 161
10790 How Many Points of ..., 315
10791 Minimum Sum LCM, 377
10792 The Laurel-Hardy Story, 781
10793 The Orc Attack, 552
10800 Not That Kind of Graph, 162
10801 Lift Hopping, 654
10802 Lex Smallest Drive, 459

- 10803 Thunder Mountain, 552
 10804 Gopher Strategy, 595
 10805 Cockroach Escape Networks, 529
 10806 Dijkstra Dijkstra, 582
 10807 Prim Prim, 529
 10810 Ultra-QuickSort, 220
 10812 Beat the Spread, 17
 10813 Traditional BINGO, 34
 10814 Simplifying Fractions, 247
 10815 Andy's First Dictionary, 161
 10816 Travel in Desert, 540
 10817 Headmaster's Headache, 665
 10819 Trouble of 13-Dots, 629
 10820 Send a Table, 378
 10821 Constructing BST, 730
 10823 Of Circles and Squares, 815
 10827 Maximum Sum on a Torus, 637
 10828 Back to Kernighan-Ritchie, 343
 10832 Yoyodyne Propulsion Systems, 804
 10842 Traffic Flow, 529
 10843 Anne's Game, 326
 10848 Make Palindrome Checker, 165
 10849 Move the Bishop, 747
 10850 The Gossipy Gossipers Gossip ..., 461
 10851 2D Hieroglyphs Decoder, 153
 10852 Less Prime, 361
 10854 Number of Paths, 168
 10855 Rotated Square, 34
 10856 Recover Factorial, 367
 10858 Unique Factorization, 418
 10859 Placing Lampposts, 684
 10862 Connect the Cable Wires, 317
 10865 Brownie Points I, 805
 10870 Recurrences, 318
 10871 Primed Subsequence, 367
 10874 Segments, 544
 10875 Big Math, 162
 10876 Factory Robot, 461
 10878 Decode the Tape, 153
 10879 Code Refactoring, 384
 10880 Colin and Ryan, 368
 10881 Piotr's Ants, 783
 10882 Koerner's Pub, 314
 10883 Supermean, 283
 10887 Concatenation of Languages, 61
 10888 Warehouse, 608
 10890 Maze, 429
 10891 Game of Sum, 672
 10892 LCM Cardinality, 377
 10894 Save Hridoy, 162
 10895 Matrix Transpose, 34
 10896 Known Plaintext Attack, 163
 10897 Travelling Distance, 759
 10898 Combo Deal, 665
 10900 So You Want to Be a 2n-aire?, 339
 10901 Ferry Loading III, 49
 10902 Pick-Up Sticks, 810
 10903 Rock-Paper-Scissors ..., 165
 10905 Children's Game, 240
 10906 Strange Integration, 168
 10907 Art Gallery, 856
 10908 Largest Square, 637
 10909 Lucky Number, 98
 10910 Marks Distribution, 633
 10911 Forming Quiz Teams, 665
 10912 Simple Minded Hashing, 635
 10913 Walking on a Grid, 679
 10916 Factstone Benchmark, 285
 10917 A Walk Through the Forest, 679
 10918 Tri Tiling, 695
 10919 Prerequisites, 61
 10920 Spiral Tap, 34
 10921 Find the Telephone, 163
 10922 2 the 9s, 384
 10923 Seven Seas, 414
 10924 Prime Words, 361
 10925 Krakovia, 253
 10926 How Many Dependencies, 459
 10927 Bright Lights, 781
 10928 My Dear Neighbours, 504
 10929 You Can Say 11, 248
 10930 A-Sequence, 632
 10931 Parity, 65
 10934 Dropping Water Balloons, 700
 10935 Throwing Cards Away I, 47
 10937 Blackbeard the Pirate, 686
 10938 Flea Circus, 475
 10940 Throwing Cards Away II, 38
 10942 Can of Beans, 393
 10943 How Do You Add, 633

- 10944 Nuts for Nuts, 686
10945 Mother Bear, 723
10946 You Want What Filled, 746
10947 Bear With Me Again, 797
10948 The Primary Problem, 366
10950 Bad Code, 414
10951 Polynomial GCD, 373
10954 Add All, 675
10957 So Doku Checker, 407
10958 How Many Solutions, 368
10959 The Party Part I, 462
10961 Chasing After Don Giovanni, 743
10963 The Swallowing Ground, 743
10964 Strange Planet, 743
10967 The Great Escape, 545
10970 Big Chocolate, 17
10973 Triangle Counting, 448
10976 Fractions Again, 269
10977 Enchanted Forest, 462
10978 Let's Play Magic, 33
10980 Lowest Price in Town, 630
10982 Troublemakers, 582
10983 Buy One Get the Rest Free, 571
10986 Sending Email, 544
10987 Antifloyd, 552
10990 Another New Function, 381
10991 Region, 788
10992 The Ghost of Programmers, 250
10993 Ignoring Digits, 456
10994 Simple Addition, 384
10997 Medals, 416
11000 Bee, 317
11001 Necklace, 273
11002 Towards Zero, 696
11003 Boxes, 721
11005 Cheapest Base, 263
11008 Antimatter Ray Clearcutting, 806
11012 Cosmic Cabbages, 867
11013 Get Straight, 343
11015 05-2 Rendezvous, 552
11019 Matrix Matcher, 188
11021 Tribles, 331
11022 String Factoring, 667
11026 A Grouping Problem, 695
11028 Sum of Product, 326
11029 Leading and Trailing, 284
11032 Function Overloading, 98
11034 Ferry Loading IV, 49
11036 Eventually Periodic Sequence, 385
11038 How Many O's, 289
11039 Building Designing, 240
11040 Add Bricks in the Wall, 34
11042 Complex Difficult and ..., 273
11044 Searching for Nessy, 24
11045 My T-Shirt Suits Me, 588
11047 The Scrooge Co Problem, 552
11048 Automatic Correction of ..., 169
11049 Basic Wall Maze, 462
11052 Economic Phone Calls, 416
11053 Flavius Josephus Reloaded, 38
11054 Wine Trading in Gergovia, 739
11055 Homogeneous Squares, 34
11056 Formula 1, 162
11057 Exact Sum, 231
11059 Maximum Product, 728
11060 Beverages, 502
11062 Andy's Second Dictionary, 161
11063 B2-Sequence, 61
11064 Number Theory, 378
11065 A Gentlemen's Agreement, 620
11067 Little Red Riding Hood, 679
11068 An Easy Task, 763
11069 A Graph Problem, 620
11070 The Good Old Times, 168
11072 Points, 844
11074 Draw Grid, 162
11076 Add Again, 290
11078 Open Credit System, 729
11080 Place the Guards, 464
11084 Anagram Division, 650
11085 Back to the 8-Queens, 407
11086 Composite Prime, 366
11088 End up with More Teams, 665
11089 Fi-Binary Number, 319
11090 Going in Cycle, 550
11092 IIUC HexWorld, 758
11093 Just Finish It Up, 33
11094 Continents, 746
11096 Nails, 839
11097 Poor My Problem, 550

- 11099 Next Same-Factored, 368
 11100 The Trip 2007, 730
 11101 Mail Mania, 462
 11103 WFF 'N PROOF, 163
 11105 Semi-Prime H-Numbers, 367
 11107 Life Forms, 200
 11108 Tautology, 70
 11110 Equidivisions, 746
 11111 Generalized Matrioshkas, 46
 11113 Continuous Fractions, 270
 11115 Uncle Jack, 289
 11121 Base -2, 263
 11122 Tri Tri, 856
 11125 Arrange Some Marbles, 679
 11127 Triple-Free Binary Strings, 429
 11130 Billiard Bounces, 783
 11131 Close Relatives, 459
 11133 Eigensequence, 679
 11136 Hoax or What, 61
 11137 Ingenuous Cubrency, 735
 11138 Nuts and Bolts, 600
 11140 Little Ali's Little Brother, 34
 11147 KuPellaKeS BST, 70
 11148 Moliu Fractions, 205
 11150 Cola, 385
 11151 Longest Palindrome, 717
 11152 Colourful Flowers, 791
 11157 Dynamic Frog, 730
 11159 Factors and Multiples, 595
 11160 Going Together, 456
 11161 Help My Brother (II), 317
 11163 Jaguar King, 442
 11164 Kingdom Division, 780
 11167 Monkeys in the Emei Mountain, 575
 11170 Cos(NA), 774
 11172 Relational Operator, 17
 11173 Grey Codes, 63
 11176 Winning Streak, 690
 11181 Probability|Given, 329
 11185 Ternary, 253
 11192 Group Reverse, 211
 11195 Another n-Queen Problem, 407
 11196 Birthday Cake, 427
 11198 Dancing Digits, 456
 11199 Equations in Disguise, 429
 11201 The Problem of the Crazy ..., 414
 11202 The Least Possible Effort, 749
 11203 Can You Decide It for ME, 168
 11204 Musical Instruments, 289
 11205 The Broken Pedometer, 659
 11207 The Easiest Way, 785
 11212 Editing a Book, 418
 11218 KTV, 665
 11219 How Old Are You, 397
 11220 Decoding the Message, 163
 11221 Magic Square Palindromes, 34
 11222 Only I Did It, 32
 11223 O: Dah Dah Dah, 163
 11225 Tarot Scores, 161
 11226 Reaching the Fix-Point, 635
 11227 The Silver Bullet, 805
 11228 Transportation System, 529
 11230 Annoying Painting Tool, 730
 11231 Black and White Painting, 747
 11232 Cylinder, 850
 11233 Deli Deli, 159
 11234 Expressions, 70
 11235 Frequent Values, 90
 11236 Grocery Store, 414
 11237 Halloween Treats, 311
 11239 Open Source, 58
 11240 Antimonotonicity, 721
 11241 Humidex, 284
 11242 Tour de France, 241
 11244 Counting Stars, 746
 11246 K-Multiple Free Set, 314
 11247 Income Tax Hazard, 384
 11249 Game, 358
 11254 Consecutive Integers, 315
 11255 Necklace, 310
 11258 String Partition, 695
 11259 Coin Changing Again, 736
 11262 Weird Fence, 595
 11264 Coin Collector, 736
 11265 The Sultan's Problem, 856
 11267 The Hire-a-Coder Business ..., 529
 11269 Setting Problems, 739
 11270 Tiling Dominoes, 326
 11278 One-Handed Typist, 163
 11279 Keyboard Comparison, 161

- 11280 Flying to Frederiction, 550
11281 Triangular Pegs in Round ..., 792
11282 Mixing Invitations, 315
11283 Playing Boggle, 416
11284 Shopping Trip, 665
11285 Exchange Rates, 695
11286 Conformity, 58
11287 Pseudoprime Numbers, 363
11291 Smeech, 343
11292 Dragon of Loowater, 730
11294 Wedding, 496
11296 Counting Solutions to an ..., 370
11297 Census, 90
11298 Dissecting a Hexagon, 370
11300 Spreading the Wealth, 215
11301 Great Wall of China, 584
11307 Alternative Arborescence, 681
11308 Bankrupt Baker, 58
11309 Counting Chaos, 723
11310 Delivery Debacle, 289
11311 Exclusively Edible, 645
11313 Gourmet Games, 384
11314 Hardly Hard, 764
11317 GCD + LCM, 381
11319 Stupid Sequence, 281
11321 Sort Sort and Sort, 241
11324 The Largest Clique, 490
11326 Laser Pointer, 781
11327 Enumerating Rational Numbers, 381
11329 Curious Fleas, 456
11330 Andy's Shoes, 730
11331 The Joys of Farming, 632
11332 Summing Digits, 384
11335 Discrete Pursuit, 285
11338 Minefield, 545
11340 Newspaper, 161
11341 Term Strategy, 629
11342 Three-Square, 285
11343 Isolated Segments, 810
11344 The Huge One, 248
11345 Rectangles, 785
11346 Probability, 329
11347 Multifactorials, 368
11348 Exhibition, 61
11349 Symmetric Matrix, 34
11350 Stern-Brocot Tree, 269
11351 Last Man Standing, 38
11352 Crazy King, 749
11353 A Different Kind of Sorting, 368
11354 Bond, 536
11356 Dates, 397
11357 Ensuring Truth, 165
11358 Faster Processing ..., 571
11360 Have Fun with Matrices, 34
11362 Phone List, 169
11364 Optimal Parking, 148
11367 Full Tank?, 544
11368 Nested Dolls, 730
11369 Shopaholic, 241
11371 Number Theory for Newbies, 384
11374 Airport Express, 544
11375 Matches, 287
11377 Airport Setup, 544
11378 Bey Battle, 860
11380 Down Went The Titanic, 575
11383 Golden Tiger Claw, 616
11384 Help is Needed for Dexter, 284
11385 Da Vinci Code, 317
11387 The 3-Regular Graph, 505
11388 GCD LCM, 377
11389 The Bus Driver Problem, 730
11391 Blobs in the Board, 665
11393 Tri-Isomorphism, 504
11395 Sigma Function, 368
11396 Claw Decomposition, 464
11398 The Base-1 Number System, 262
11401 Triangle Counting, 286
11402 Ahoy Pirates, 90
11403 Binary Multiplication, 162
11405 Can U Win, 686
11406 Best Trap, 792
11407 Squares, 734
11408 Count DePrimes, 366
11412 Dig the Holes, 417
11413 Fill the Containers, 227
11414 Dream, 505
11415 Count the Factorials, 367
11417 GCD, 380
11418 Clever Naming Patterns, 588
11419 SAM I AM, 619

- 11420 Chest of Drawers, 695
 11423 Cache Simulator, 98
 11424 GCD Extreme (I), 381
 11426 GCD Extreme (II), 381
 11427 Expect the Expected, 692
 11428 Cubes, 58
 11432 Busy Programmer, 679
 11437 Triangle Fun, 781
 11439 Maximizing the ICPC, 607
 11447 Reservoir Logs, 846
 11448 Who Said Crisis, 253
 11450 Wedding Shopping, 633
 11451 Water Restrictions, 414
 11452 Dancing the Cheeky-Cheeky, 169
 11455 Behold My Quadrangle, 785
 11456 Trainsorting, 721
 11459 Snakes and Ladders, 34
 11461 Square Numbers, 32
 11462 Age Sort, 220
 11463 Commandos, 543
 11464 Even Parity, 432
 11466 Largest Prime Divisor, 368
 11468 Substring, 691
 11470 Square Sums, 34
 11471 Arrange the Tiles, 414
 11472 Beautiful Numbers, 695
 11473 Campus Roads, 804
 11474 Dying Tree, 461
 11475 Extend to Palindromes, 727
 11476 Factorizing Larget Integers, 368
 11478 Halum, 558
 11479 Is This the Easiest Problem, 771
 11480 Jimmy's Balls, 287
 11482 Building a Triangular Museum, 162
 11483 Code Creator, 153
 11486 Finding Paths in Grid, 681
 11487 Gathering Food, 679
 11489 Integer Game, 358
 11490 Just Another Problem, 416
 11491 Erasing and Winning, 100
 11492 Babel, 654
 11494 Queen, 747
 11495 Bubbles and Buckets, 220
 11496 Musical Loop, 33
 11498 Division of Nlogonia, 764
 11500 Vampires, 335
 11503 Virtual Friends, 125
 11504 Dominos, 458
 11505 Logo, 766
 11506 Angry Programmer, 582
 11507 Bender B. Rodriguez Problem, 766
 11510 Erdös Unit Fractions, 366
 11511 Frieze Patterns, 32
 11512 GATTACA, 200
 11513 9 Puzzle, 456
 11514 Batman, 650
 11515 Cranes, 795
 11516 WiFi, 227
 11517 Exact Change, 736
 11518 Dominos 2, 459
 11519 Logo 2, 761
 11520 Fill the Square, 34
 11524 In-Circle, 790
 11525 Permutation, 293
 11526 H(n), 370
 11530 SMS Typing, 161
 11532 Simple Adjacency ..., 65
 11534 Say Goodbye to Tic-Tac-Toe, 351
 11536 Smallest Sub-Array, 54
 11538 Chess Queen, 289
 11540 Sultan's Chandelier, 310
 11541 Decoding, 163
 11542 Square, 281
 11545 Avoiding Jungle in the Dark, 679
 11547 Automatic Answer, 17
 11548 Blackboard Bonanza, 157
 11549 Calculator Conundrum, 61
 11550 Demanding Dilemma, 445
 11552 Fewest Flops, 695
 11553 Grid Game, 210
 11554 Hapless Hedonism, 287
 11555 Aspen Avenue, 695
 11556 Best Compression Ever, 284
 11559 Event Planning, 17
 11561 Getting Gold, 746
 11565 Simple Equations, 285
 11566 Let's Yum Cha, 630
 11567 Moliu Number Generator, 63
 11569 Lovely Hint, 679
 11572 Unique Snowflakes, 58

- 11573 Ocean Currents, 456
11574 Colliding Traffic, 781
11576 Scrolling Sign, 169
11577 Letter Frequency, 161
11579 Triangle Trouble, 780
11581 Grid Successors, 34
11582 Colossal Fibonacci Numbers, 317
11583 Alien DNA, 730
11584 Partitioning by Palindromes, 724
11585 Nurikabe, 746
11586 Train Tracks, 509
11587 Brick Game, 645
11588 Image Coding, 241
11597 Spanning Subtrees, 526
11601 Avoiding Overlaps, 94
11605 Lights Inside a 3D Grid, 343
11608 No Problem, 32
11609 Teams, 386
11610 Reverse Prime, 369
11614 Etruscan Warriors Never Play ..., 285
11615 Family Tree, 71
11616 Roman Numerals, 267
11621 Small Factors, 52
11624 Fire, 462
11626 Convex Hull, 844
11627 Slalom, 226
11628 Another Lottery, 329
11629 Ballot Evaluation, 58
11631 Dark Roads, 529
11634 Generate Random Numbers, 61
11635 Hotel Booking, 541
11636 Hello World, 282
11638 Temperature Monitoring, 384
11639 Guard the Land, 785
11643 Knight Tour, 755
11646 Athletics Track, 788
11648 Divide the Land, 785
11650 Mirror Clock, 399
11655 Water Land, 680
11658 Best Coalitions, 632
11659 Informants, 418
11660 Look-and-Say Sequences, 326
11661 Burger Time, 823
11664 Langton's Ant, 743
11666 Logarithms, 284
11667 Income Tax Hazard (II), 343
11677 Alarm Clock, 399
11678 Exchanging Cards, 231
11679 Sub-Prime, 17
11683 Laser Sculpture, 823
11686 Pick up Sticks, 490
11687 Digits, 18
11689 Soda Surpler, 385
11690 Money Matters, 124
11692 Rain Fall, 285
11693 Speedy Escape, 552
11695 Flight Planning, 498
11697 Playfair Cipher, 163
11701 Cantor, 263
11703 sqrt log sin, 635
11709 Trust Groups, 490
11710 Expensive Subway, 529
11713 Abstract Names, 157
11714 Blind Sorting, 285
11715 Car, 285
11716 Digital Fortress, 163
11717 Energy Saving ..., 29
11718 Fantasy of a Summation, 386
11719 Gridland Airports, 326
11721 Instant View of Big Bang, 548
11722 Joining with Friend, 329
11723 Numbering Roads, 384
11727 Cost Cutting, 17
11728 Alternate Task, 368
11729 Commando War, 739
11730 Number Transformation, 462
11733 Airports, 529
11734 Big Number of Teams Will ..., 157
11736 Debugging RAM, 161
11742 Social Constraints, 210
11743 Credit Check, 161
11744 Parallel Carry Adder, 65
11747 Heavy Cycle Edges, 529
11749 Poor Trade Advisor, 459
11752 The Super Powers, 282
11753 Creating Palindrome, 414
11754 Code Feat, 390
11755 Table Tennis, 691
11760 Brother Arif Please Feed Us, 32
11761 Super Heronian Triangle, 792

- 11762 Race to 1, 693
 11764 Jumping Mario, 17
 11769 All Souls Night, 876
 11770 Lighting Away, 490
 11774 Doom's Day, 300
 11777 Automate the Grades, 17
 11780 Miles 2 Km, 632
 11782 Optimal Cut, 681
 11783 Nails, 810
 11786 Global Raining at Bididibus, 161
 11787 Numeral Hieroglyphs, 163
 11790 Murcia's Skyline, 721
 11792 Krochanska is Here, 462
 11795 Mega Mans Missions, 686
 11797 Drutojan Express, 49
 11799 Horror Dash, 17
 11800 Determine the Shape, 804
 11804 Argentina, 241
 11805 Bafana Bafana, 384
 11806 Cheerleaders, 314
 11809 Floating-Point Numbers, 12
 11813 Shopping, 686
 11816 HST, 273
 11817 Tunnelling the Earth, 759
 11821 High-Precision Number, 253
 11824 A Minimum Land Price, 241
 11827 Maximum GCD, 373
 11830 Contract Revision, 157
 11831 Sticker Collector Robot, 743
 11832 Account Book, 696
 11833 Route Change, 545
 11834 Elevator, 797
 11835 Formula 1, 34
 11838 Come and Go, 490
 11839 Optical Reader, 162
 11840 Tic-Tac-Toe, 354
 11841 Y-Game, 461
 11847 Cut the Silver Bar, 285
 11849 CD, 61
 11850 Alaska, 220
 11854 Egypt, 774
 11857 Driving Range, 529
 11858 Frosh Week, 220
 11859 Division Game, 347
 11860 Document Analyzer, 58
 11875 Brick Game, 217
 11876 N + NOD (N), 369
 11877 The Coco-Cola Store, 385
 11878 Homework Checker, 162
 11879 Multiple of 17, 248
 11881 Internal Rate of Return, 226
 11888 Abnormal 89's, 724
 11889 Benefit, 377
 11890 Calculus Simplified, 730
 11894 Genius MJ, 766
 11898 Killer Problem, 312
 11900 Boiled Eggs, 730
 11902 Dominator, 476
 11906 Knight in a War Grid, 459
 11908 Skyscraper, 695
 11909 Soya Milk, 774
 11916 Emoogle Grid, 389
 11917 Do Your Own Homework, 58
 11926 Multitasking, 65
 11930 Rectangles, 496
 11933 Splitting Numbers, 63
 11934 Magic Formula, 384
 11935 Through the Desert, 226
 11936 The Lazy Lumberjacks, 771
 11942 Lumberjack Sequencing, 17
 11945 Financial Management, 27
 11946 Code Number, 163
 11947 Cancer or Scorpio, 397
 11951 Area, 637
 11952 Arithmetic, 265
 11953 Battleships, 746
 11955 Binomial Theorem, 273
 11956 Brainfuck, 24
 11957 Checkers, 680
 11958 Comming Home, 399
 11959 Dice, 300
 11960 Divisor Game, 368
 11961 DNA, 414
 11962 DNA II, 289
 11965 Extra Spaces, 162
 11966 Galactic Bonding, 461
 11967 Hic-Hac-Hoe, 461
 11968 In the Airport, 269
 11970 Lucky Numbers, 285
 11971 Polygon, 329

- 11974 Switch The Lights, 456
 11975 Tele-Loto, 743
 11983 Wired Advertisement, 830
 11984 A Change in Thermal Unit, 27
 11986 Save from Radiation, 284
 11987 Almost Union-Find, 125
 11988 Broken Keyboard (a.k.a. ..., 66
 11990 Dynamic Inversion, 114
 11991 Easy Problem from RuJia Liu, 58
 11992 Fast Matrix Operations, 90
 11995 I Can Guess the Data ..., 52
 11997 K Smallest Sums, 631
 12001 UVa Panel Discussion, 290
 12002 Happy Birthday, 721
 12003 Array Transformer, 112
 12004 Bubble Sort, 315
 12005 Find Solutions, 369
 12015 Google is Feeling Lucky, 241
 12016 Herbicide, 819
 12019 Doom's Day Algorithm, 397
 12022 Ordering T-Shirts, 326
 12024 Hats, 315
 12027 Very Big Perfect Squares, 285
 12028 A Gift from the Setter, 241
 12030 Help the Winner, 679
 12032 The Monkey and the Oiled ..., 226
 12036 Stable Grid, 312
 12043 Divisors, 368
 12047 Highest Paid Toll, 544
 12049 Just Prune The List, 61
 12060 All Integer Average, 269
 12063 Zeros and Ones, 633
 12068 Harmonic Mean, 373
 12070 Invite Your Friends, 743
 12071 Understanding Recursion, 220
 12074 The Ultimate Bamboo Eater, 766
 12083 Guardian of Decency, 620
 12085 Mobile Casanova, 161
 12086 Potentiometers, 89
 12090 Counting Zeroes, 368
 12096 The SetStack Computer, 61
 12097 Pie, 227
 12100 Printer Queue, 52
 12101 Prime Path, 462
 12108 Extraordinarily Tired ..., 66
 12114 Bachelor Arithmetic, 328
 12118 Inspector's Dilemma, 509
 12124 Assemble, 226
 12125 March of the Penguins, 575
 12134 Find the Format String, 163
 12135 Switch Bulbs, 456
 12136 Schedule of a Married Man, 399
 12137 Puzzles of Triangles, 369
 12143 Stopping Doom's Day, 253
 12144 Almost Shortest Path, 545
 12148 Electricity, 396
 12149 Feynman, 315
 12150 Pole Position, 32
 12155 ASCII Diamond, 162
 12157 Tariff Plan, 17
 12159 Gun Fight, 595
 12160 Unlock the Lock, 462
 12165 Triangle Hazard, 804
 12167 Proving Equivalences, 487
 12168 Cat vs. Dog, 620
 12169 Disgruntled Judge, 416
 12170 Easy Climb, 704
 12171 Sculpture, 827
 12186 Another Crisis, 681
 12187 Brothers, 149
 12190 Electric Bill, 226
 12192 Grapevine, 233
 12195 Jingle Composing, 20
 12205 Happy Telephones, 416
 12207 That is Your Queue, 52
 12210 A Match Making Problem, 241
 12218 An Industrial Spy, 367
 12230 Crossing Rivers, 343
 12238 Ants Colony, 475
 12239 Bingo, 231
 12243 Flowers Flourish from France, 19
 12247 Jollo, 210
 12249 Overlapping Scenes, 210
 12250 Language Detection, 20
 12255 Underwater Snipers, 227
 12256 Making Quadrilaterals, 785
 12269 Lawn Mower, 241
 12279 Emoogle Balance, 17
 12280 A Digital Satire of Digital ..., 162
 12281 Hyper Box, 319

- 12289 One-Two-Three, 18
 12290 Counting Game, 384
 12291 Polyomino Composer, 34
 12293 Box Game, 358
 12299 RMQ with Shifts, 90
 12300 Smallest Regular Polygon, 785
 12301 An Angular Puzzle, 778
 12302 Nine-Point Circle, 792
 12307 Smallest Enclosing Rectangle, 865
 12308 Smallest Enclosing Box, 879
 12311 All-Pair Farthest Points, 865
 12318 Digital Roulette, 386
 12319 Edgetown's Traffic Jams, 552
 12321 Gas Stations, 800
 12324 Philip J. Fry Problem, 700
 12335 Lexicographic Order, 293
 12337 Bob's Beautiful Balls, 416
 12342 Tax Calculator, 24
 12346 Water Gate Management, 241
 12347 Binary Search Tree, 70
 12348 Fun Coloring, 658
 12356 Army Buddies, 29
 12363 Hedge Maze, 481
 12364 In Braille, 162
 12366 King's Poker, 34
 12372 Packing for Holiday, 148
 12376 As Long as I Learn I Live, 459
 12379 Central Post Office, 498
 12390 Distributing Ballot Boxes, 227
 12392 Guess the Numbers, 46
 12394 Peer Review, 61
 12397 Roman Numerals, 267
 12398 NumPuzz I, 34
 12403 Save Setu, 20
 12405 Scarecrow, 730
 12406 Help Dexter, 241
 12412 A Typical Homework, 162
 12414 Calculating Yuan Fen, 165
 12416 Excessive Space Remover, 284
 12428 Enemy at the Gates, 504
 12439 February 29, 399
 12442 Forwarding Emails, 459
 12455 Bars, 632
 12457 Tennis Contest, 693
 12459 Bees' Ancestors, 317
 12460 Careful Teacher, 461
 12461 Airplane, 331
 12463 Little Nephew, 289
 12464 Professor Lazy Ph.D., 269
 12467 Secret Word, 178
 12468 Zapping, 17
 12469 Stones, 358
 12470 Tribonacci, 318
 12478 Hardest Problem Ever (Easy), 169
 12482 Short Story Competition, 162
 12485 Perfect Choir, 32
 12488 Start Grid, 216
 12498 Ant's Shopping Mall, 743
 12499 I am Dumb 3, 354
 12502 Three Families, 269
 12503 Robot Instructions, 163
 12504 Updating a Dictionary, 58
 12515 Movie Police, 163
 12516 Cinema-Cola, 730
 12527 Different Digits, 384
 12531 Hours and Minutes, 399
 12532 Interval Product, 90
 12538 Version Controlled IDE, 136
 12541 Birthdates, 241
 12542 Prime Substring, 367
 12543 Longest Word, 162
 12545 Bits Equalizer, 18
 12549 Sentry Robots, 619
 12554 A Special Happy Birthday ..., 384
 12555 Baby Me, 153
 12563 Jin Ge Jin Qu Hao, 632
 12569 Planning Mobile Robot on ..., 456
 12571 Brother & Sisters!, 61
 12577 Hajj-e-Akbar, 20
 12578 10:6:2, 788
 12582 Wedding of Sultan, 459
 12583 Memory Overflow, 32
 12585 Poker End Games, 693
 12592 Slogan Learning of Princess, 58
 12602 Nice Licence Plates, 262
 12608 Garbage Collection, 29
 12611 Beautiful Flag, 786
 12614 Earn For Future, 61
 12620 Fibonacci Sum, 317
 12621 On a Diet, 632

- 12626 I Love Pizza, 162
 12640 Largest Sum Game, 728
 12641 Reodrnreig Lteetrs in Wrods, 61
 12643 Tennis Rounds, 61
 12644 Vocabulary, 595
 12646 Zero or One, 157
 12648 Boss, 459
 12650 Dangerous Dive, 17
 12654 Patches, 695
 12657 Boxes in a Line, 66
 12658 Character Recognition, 32
 12662 Good Teacher, 32
 12665 Joking with Fermat's Last ..., 416
 12667 Last Blood, 32
 12668 Attacking Rooks, 596
 12673 Football, 218
 12694 Meeting Room Arrangement, 418
 12696 Cabin Baggage, 162
 12700 Banglawash, 162
 12703 Little Rakin, 368
 12704 Little Masters, 786
 12705 Breaking Board, 241
 12708 GCD The Largest, 373
 12709 Falling Ants, 17
 12712 Pattern Locker, 289
 12720 Algorithm of Phil, 11
 12723 Dudu the Possum, 693
 12725 Fat and Orial, 27
 12730 Skyrk's Bar, 343
 12748 WiFi Access, 786
 12750 Keep Rafa at Chelsea, 17
 12751 An Interesting Game, 315
 12770 Palinagram, 61
 12783 Weak Links, 481
 12786 Friendship Networks, 505
 12791 Lap, 225
 12792 Shuffled Deck, 389
 12796 Teletransport, 680
 12797 Letters, 456
 12798 Handball, 17
 12801 Grandpa Pepe's Pizza, 416
 12802 Gift From the Gods, 361
 12805 Raiders of the Lost Sign, 368
 12808 Banning Balconing, 284
 12820 Cool Word, 58
 12821 Double Shortest Paths, 584
 12822 Extraordinarily Large LED, 399
 12826 Incomplete Chessboard, 456
 12834 Extreme Terror, 730
 12840 The Archery Puzzle, 422
 12841 In Puzzleland (III), 525
 12842 Courier Problem, 285
 12843 Disputed Claims, 774
 12844 Outwitting the Weighing ..., 416
 12845 The Jolly Friar's Puzzle, 660
 12846 A Daisy Puzzle Game, 645
 12848 In Puzzleland (IV), 269
 12849 Mother's Jam Puzzle, 281
 12850 Skating Puzzle, 281
 12851 The Tinker's Puzzle, 850
 12852 The Miser's Puzzle, 377
 12853 The Pony Cart Problem, 788
 12854 Automated Checking Machine, 17
 12861 Help Cupid, 730
 12862 Intrepid Climber, 681
 12869 Zeroes, 289
 12873 The Programmers, 575
 12878 Flowery Trails, 544
 12893 Count It, 17
 12894 Perfect Flag, 788
 12895 Armstrong Number, 384
 12896 Mobile SMS, 154
 12901 Refraction, 774
 12904 Load Balancing, 269
 12908 The Book Thief, 315
 12909 Numeric Center, 225
 12910 Snakes and Ladders, 341
 12911 Subset Sum, 632
 12917 Prop Hunt, 17
 12918 Lucky Thief, 315
 12924 Immortal Rabbits, 262
 12930 Bigger or Smaller, 273
 12931 Common Area, 819
 12934 Factorial Division, 289
 12938 Just Another Easy Problem, 416
 12950 Even Obsession, 545
 12951 Stock Market, 695
 12952 Tri-Du, 17
 12955 Factorial, 731
 12959 Strategy Game, 32

- 12965 Angry Bids, 225
12967 Spray Graphs, 679
12970 Alcoholic Pilots, 269
12981 Secret Master Plan, 34
12992 Huatuo's Medicine, 17
12995 Farey Sequence, 379
12996 Ultimate Mango Challenge, 17
13004 At Most Twice, 414
13007 D as in Daedalus, 17
13012 Identifying Tea, 17
13015 Promotions, 459
13018 Dice Cup, 416
13025 Back to the Past, 397
13026 Search the Khoj, 157
13030 Brain Fry, 693
13031 Geek Power Inc., 730
13034 Solve Everything, 17
13037 Chocolate, 61
13038 Directed Forest, 463
13047 Arrows, 162
13048 Burger Stand, 157
13049 Combination Lock, 153
13054 Hippo Circus, 730
13055 Inception, 46
13057 Prove Them All, 490
13059 Tennis Championship, 416
13067 Prime Kebab Menu, 368
13071 Double Decker, 269
13082 High School Assembly, 730
13091 No Ball, 162
13093 Acronyms, 19
13095 Tobby and Query, 98
13096 Standard Deviation, 315
13103 Tobby and Seven, 416
13107 Royale With Cheese, 163
13108 Juanma and the Drinking ..., 253
13109 Elephants, 217
13113 Presidential Election, 241
13115 Sudoku, 34
13117 ACIS A Contagious vIruS, 814
13127 Bank Robbery, 541
13130 Cacho, 32
13131 Divisors, 368
13132 Laser Mirrors, 379
13135 Homework, 296
13140 Squares, Lists and Digital ..., 384
13141 Growing Trees, 695
13142 Destroy the Moon to Save the ..., 285
13145 Wuymul Wixcha, 163
13148 A Giveaway, 61
13151 Rational Grading, 162
13161 Candle Box, 315
13172 The Music Teacher, 545
13177 Orchestral Scores, 227
13180 The Countess' Pearls, 242
13181 Sleeping in Hostels, 157
13185 DPA Numbers I, 370
13190 RockabYE Tobby, 52
13194 DPA Numbers II, 370
13209 My Password is a Palindromic ..., 271
13212 How Many Inversions, 220
13215 Polygonal Park, 785
13216 Problem With A Ridiculously ..., 385
13217 Amazing Function, 285
13249 A Contest to Meet, 552
13252 Rotating Drum, 522
13275 Leap Birthdays, 393
13293 All-Star Three-Point Contest, 162
13295 Carroll's Scrabble, 463

参考资料

- [1] Skiena S S, Revilla M A. *Programming Challenges, The Programming Contest Training Manual* [M]. London: Springer, 2003(中译本: 斯基纳 S S, 雷维拉 M A. 挑战编程: 程序设计竞赛训练手册. 刘汝佳, 译. 北京: 清华大学出版社, 2009).
- [2] Skiena S S. *The Algorithm Design Manual* [M]. London: Springer, 2008.
- [3] Sedgewick R, Wayne K. *Algorithms* [M]. London: Pearson Education, 2011.
- [4] Halim S, Halim F. *Competitive Programming* [M]. Singapore: Lulu, 2013.
- [5] 吴永辉, 王建德, 等. *ACM-ICPC 世界总决赛试题解析(2004-2011 年)* [M]. 北京: 机械工业出版社, 2012.
- [6] 李学军主编, 常雷等编. *英语姓名译名手册* [M]. 北京: 商务印书馆, 2018.
- [7] Knuth D E. *The Art of Computer Programming* [M]. Boston: Addison-Wesley, 2011.
- [8] Cormen T H, Leiserson C E, Rivest R L, Stein C. *Introduction to Algorithms* [M]. New York: McGraw-Hill Press, 2001(中译本: 科曼 T H, 雷瑟尔森 C E, 李维斯特 R L, 斯坦 C. 算法导论. 潘金贵, 等, 译. 北京: 机械工业出版社, 2006).
- [9] Bryant R E, O'Hallaron D R. *Computer System: A Programmer's Perspective* [M]. London: Pearson Education, 2003(中译本: 布莱恩特 R E, 奥哈拉伦 D R. 深入理解计算机系统. 龚奕利, 雷迎春, 译. 北京: 机械工业出版社, 2011).
- [10] IEEE Computer Society. 754-2019 - IEEE Standard for Floating-Point Arithmetic [S]. New York: IEEE. 2019.
- [11] Goldberg D. What every computer scientist should know about floating-point arithmetic [J]. *ACM Computing Surveys*. 1991, 23(1): 5-48.
- [12] <https://github.com/morris821028/UVa> [OL], 2020.
- [13] https://github.com/forthright48/code-library/tree/master/code_templates [OL], 2020.
- [14] Conway J H, Coxeter H S M. Triangulated polygons and frieze patterns [J]. *The Mathematical Gazette*, 1973, 57(400): 87-94.
- [15] Conway J H, Coxeter H S M. Triangulated polygons and frieze patterns (continued) [J]. *The Mathematical Gazette*, 1973, 57(401): 175-183.
- [16] <http://www.cplusplus.com/reference/> [OL], 2020.
- [17] codingtmd. 编程谜题 [M]. 北京: 人民邮电出版社, 2016.
- [18] Weiss M A. *Data Struct and Algorithm Analysis in C* [M]. London: Pearson Education, 1997.
- [19] 严蔚敏, 夏伟民. 数据结构, C 语言版 [M]. 北京: 清华大学出版社, 2007.
- [20] <https://cp-algorithms.com/> [OL], 2020.
- [21] Fenwick P M. A new data structure for cumulative frequency tables [J]. *Software: Practice and Experience*, 1994, 24(3): 327-336.
- [22] https://github.com/Morass/e-maxx-eng/blob/master/src/data_structures/fenwick.md [OL], 2018.
- [23] J. Boehm H, Atkinson R, Plass M. Ropes: An alternative to strings [J]. *Software Practice and Experience*, 1995, 25(12): 1315-1330.
- [24] 侯捷. *STL 源码剖析* [M]. 武汉: 华中科技大学出版社, 2002.
- [25] Aho A V, Lam M S, Sethi R, Ullman J D. *Compilers: Principles, Techniques, and Tools* [M]. London: Pearson Education, 2007.
- [26] Knuth D E, Morris J H, Pratt V R. Fast pattern matching in strings [J]. *SIAM Journal on Computing*, 1977, 6(2): 323-350.
- [27] 周源. 浅析“最小表示法”思想在字符串循环同构问题中的应用 [C]. IOI 国家集训队论文, 2003.
- [28] Aho A V, Corasick M J. Efficient string matching: An aid to bibliographic search [J]. *Communications of the ACM*, 1975, 18(6): 333-340.
- [29] Manber U, Myers G. Suffix arrays: a new method for on-line string searches [C]. *First Annual ACM-SIAM Symposium on Discrete Algorithms*, 1990: 319-327.
- [30] 罗穗骞. 后缀数组——处理字符串的有力工具 [C]. IOI 国家集训队论文, 2003.

- [31] Kärkkäinen J, Sanders P. Simple linear work suffix array construction [C]. International Colloquium on Automata, Languages and Programming, 2003: 943-955.
- [32] 许智磊. 后缀数组 [C]. IOI 国家集训队论文, 2004.
- [33] Burrows M, Wheeler D J. A block sorting lossless data compression algorithm [R]. Research Report 124, 1994, Palo Alto, California, Digital System Research Center.
- [34] Friedl E F J. Mastering Regular Expressions [M]. Sebastopol, California: O'Reilly Media, 2002.
- [35] Shell D L. A high-speed sorting procedure [J]. Communications of the ACM, 1959, 2(7): 30-32.
- [36] Kleinberg J, Tardos É. Algorithm Design [M]. London: Pearson Education, 2006.
- [37] Bentley J. Programming Pearls [M]. Boston: Addison-Wesley, 2000.
- [38] Giordano F R, Fox W P, Horton, S B. A First Course in Mathematical Modeling [M]. Boston: Cengage Learning, 2014(中译本: 吉奥丹诺 F R 等著, 叶其孝等译. 数学建模, 第 5 版. 北京: 机械工业出版社, 2014).
- [39] Graham R L, Knuth D E, Patashnik O. Concrete Mathematics: A Foundation for Computer Science [M]. London: Pearson Education, 1994.
- [40] Horvath G, Verhoeff T T. Numerical difficulties in pre-university informatics education and competitions [J]. Informatics in Education, 2003, 2(1): 21-38.
- [41] 居余马, 等. 线性代数, 第 2 版 [M]. 北京: 清华大学出版社, 2002.
- [42] Wilson R J. Introduction to Graph Theory [M]. London: Pearson Education, 2010.
- [43] O'Shea O. Sam Loyd's Courier Problem with Diophantus, Pythagoras, and Martin Gardner. Martin Gardner in the Twenty-First Century [M]. Mathematical Association of America, 2012: 201-206.
- [44] Brualdi R A. Introductory Combinatorics [M]. London: Pearson Education, 2009.
- [45] Galante J. Generalized Cantor Expansions [J]. Rose-Hulman Undergraduate Mathematics Journal, 2004, 5(1).
- [46] 赵春来, 徐明曜. 抽象数学 I [M]. 北京: 北京大学出版社, 2008.
- [47] Hungerford T W. Algebra [M]. Newyork: Springer, 2003.
- [48] 周治国. 组合数学及应用 [M]. 哈尔滨: 哈尔滨工业大学, 2012.
- [49] 许蔓苓. 离散数学 [M]. 北京: 北京航空航天大学出版社, 2004.
- [50] Vajda S. Fibonacci and Lucas Numbers and the Golden Section: Theory and Applications [M]. Ellis Horwood, Chichester, England, 1989.
- [51] Hilton P, Pedersen J. Catalan numbers, their generalization, and their uses [J]. The Mathematical Intelligencer, 1991, 13(2): 64-75.
- [52] Davis T. Catalan numbers [OL]. <http://www.geometer.org/mathcircles>, 2011.
- [53] Stanley R P. Enumerative Combinatorics, Volume 2 [M]. London: Cambridge Studies in Advanced Mathematics, Cambridge University Press, 1999.
- [54] Ross S M. A First Course in Probability [M]. London: Pearson Education, 2014(中译本: 罗斯 S M. 概率论基础教程, 第 9 版. 童行伟, 梁宝生, 译. 北京: 机械工业出版社, 2014).
- [55] 同济大学概率统计教研组. 概率统计, 第 4 版. 上海: 同济大学出版社, 2009.
- [56] Gorroochurn P. Classic Problems of Probability [M]. New York: Wiley, 2012.
- [57] 平冈和幸, 堀玄著, 陈筱烟译. 程序员的数学——概率统计 [M]. 北京: 人民邮电出版社, 2015.
- [58] 梅诗珂. 信息学竞赛中概率问题求解初探 [C]. IOI 国家集训队论文, 2009.
- [59] Schwalbe U, Walker P. Zermelo and the Early History of Game Theory [J]. Games and Economic Behavior, 2001, 34(1): 123-137.
- [60] Gardner M. 选编, 陈为蓬译. 萨姆·劳埃德的数学趣题 [M]. 上海: 上海科技教育出版社, 1999.
- [61] Bouton C L. Nim, a game with a complete mathematical theory [J]. Annals of Mathematics, 3(14): 35-39, 1901-1902.
- [62] 秋叶拓哉, 等著, 巫泽俊, 等译. 挑战程序设计竞赛 [M]. 北京: 人民邮电出版社, 2013.
- [63] Sprague R P. Über mathematische Kampfspiele [J]. Tohoku Mathematical Journal, 1935, 41: 438-444.
- [64] Grundy P M. Mathematics and games [J]. Eureka, 1939, 2: 6-8.
- [65] 编程之美小组. 编程之美, 微软技术面试心得 [M]. 北京: 电子工业出版社, 2008.
- [66] Lenhardt S. Composite Mathematical Games [D]. Bachelor Thesis, Bratislava, 2007.

- [67] Berlekamp E R, Conway J H, Guy R K. *Winning Ways For Your Mathematical Plays(Volumes I-IV, Second Edition)* [M], A K Peters, Ltd. Wellesley, Massachusetts, 2001.
- [68] 王晓珂. 解析一类组合游戏 [C]. IOI 国家集训队论文, 2007.
- [69] Moore E H. A generalization of the game called Nim [J]. *The Annals of Mathematics*, 1910, 11(3): 93-94.
- [70] Whinhan M J. Fibonacci Nim [J]. *Fibonacci Quarterly*, 1963, 1(4): 9-13.
- [71] Silverman J H. *A Friendly Introduction to Number Theory* [M]. London: Pearson Education, 2013(中译本: 西尔弗曼 J H. 数论概论, 第 4 版. 孙智伟等译. 北京: 机械工业出版社, 2016).
- [72] De Koninck J M, Mercier A. *1001 Problems in Classical Number Theory* [M]. Providence, Rhode Island: American Mathematical Society, 2007.
- [73] Rosen K H. *Elementary Number Theory and Its Applications* [M]. London: Pearson Education, 2011(中译本: 罗森 K H. 初等数论及其应用, 第 6 版. 夏鸿刚译. 北京: 机械工业出版社, 2015).
- [74] 潘承洞, 潘承彪. 初等数论, 第 3 版. 北京: 北京大学出版社, 2013.
- [75] Dunham W. *Journey Through Genius: The Great Theorems of Mathematics* [M]. New York: John Wiley & Sons(中译本: Dunham W 著, 李繁荣, 李莉萍译. 天才引导的历程: 数学中的伟大定理. 北京: 机械工业出版社, 2013).
- [76] Pollard J M. A Monte Carlo method for factorization [J]. *BIT Numerical Mathematics*, 1975, 15(3): 331-334.
- [77] Riesel H. *Prime Numbers and Computer Methods for Factorization* [M]. Boston: Birkhäuser, 2012.
- [78] 熊金平, 唐郑熠. 基于位运算的 N 皇后问题的解法 [J]. 计算机与数字工程, 2011, 39(1): 42-44.
- [79] Knuth D E. *Dancing links* [OL]. <https://arxiv.org/pdf/cs/0011047.pdf>, 2020.
- [80] 俞勇. ACM 国际大学生程序设计竞赛: 算法与实现 [M]. 北京: 清华大学出版社, 2013.
- [81] Loughry J, van Hemert J I, Schoofs L. Efficiently enumerating the subsets of a set [OL]. <http://applied-math.org/>, 2020.
- [82] Heineman G T, Pollice G, Selkow S. *Algorithms in a Nutshell* [M]. Sebastopol, California: O'Reilly Media, 2016.
- [83] Korf R E. Recent Progress in the Design and Analysis of Admissible Heuristic Functions, Proceeding, Abstraction, Reformulation, and Approximation: 4th International Symposium(SARA) [C]. Lecture notes in Computer Science #1864, 2000: 45-51.
- [84] Bauer B. The Manhattan pair distance heuristic for the 15-Puzzle [R]. Technical Report Paderborn Center for Parallel Computing, TR-001-94, 1994.
- [85] 徐俊明. 图论及其应用, 第 3 版 [M]. 安徽合肥: 中国科学技术大学出版社, 2010.
- [86] Tarjan R. Depth-first search and linear graph algorithms [J]. *SIAM Journal on Computing*, 1972, 1(2): 146-160.
- [87] Gabow H N. Path-based depth-first search for strong and biconnected components [J]. *Information Processing Letters*, 2000, 74: 107-114.
- [88] Aspvall B, Plass M F, Tarjan R E. A linear-time algorithm for testing the truth of certain quantified boolean formulas [J]. *Information Processing Letters*, 1979, 8(3): 121-123.
- [89] 赵爽. 2-SAT 解法浅析 [C]. IOI 论文, 华中师范大学第一附属中学.
- [90] Kahn A B. Topological sorting of large networks [J]. *Communications of the ACM*, 1962, 5(11): 558-562.
- [91] 程钊. 图论中若干著名问题的历史注记 [J]. 数学的实践与认知, 2009, 39(24): 73-81.
- [92] Hakimi S L. On realizability of a set of integers as degrees of the vertices of a linear graph. I [J]. *Journal of the Society for Industrial and Applied Mathematics*, 1962, 10: 496-506.
- [93] 管梅谷. 奇偶点图上作业法 [J]. 数学学报, 1960, 10: 263-266.
- [94] 管梅谷. 中国投递员问题综述 [J]. 数学研究与评论, 1984, 4(1): 113-119.
- [95] Edmonds J, Johnson E L. Matching, Euler tours and the Chinese postman [J]. *Mathematical Programming*, 1973, 5(1): 88-124.
- [96] Ore O. Note on Hamilton circuits [J]. *The American Mathematical Monthly*, 1960, 67(1): 55.
- [97] Dirac G A. Some theorems on abstract graphs [J]. *Proceedings of the London Mathematical Society*,

- 1952, 2: 69-81.
- [98] Palmer E M. The hidden algorithm of Ore's theorem on Hamiltonian cycles [J]. *Computers & Mathematics with Applications*, 1997, 34(11): 113-119.
- [99] Prim R C. Shortest connection networks and some generalizations [J]. *Bell System Technical Journal*, 1957, 36(6): 1389-1401.
- [100] Kruskal J B. On the shortest spanning subtree of a graph and the traveling salesman problem [J]. *Proceedings of the American Mathematical Society*, 1956, 7(1): 48-50.
- [101] Hassin R, Tamir A. On the minimum diameter of spanning tree problem [J]. *Information Processing Letters*, 1995, 53(2): 109-111.
- [102] Kariv O, Hakimi S L. An Algorithmic Approach to Network Location Problems. I: The p-Centers [J]. *Siam Journal on Applied Mathematics*, 1979, 37(3): 513-538.
- [103] Goemans M X. Minimum Bounded Degree Spanning Trees [C]. *Foundations of Computer Science*, 2006: 273-282.
- [104] 汪汀. 最小生成树问题的拓展 [C]. IOI 国家集训队论文, 2004.
- [105] Dijkstra E W. A note on two problems in connexion with graphs [J]. *Numerische Mathematik*, 1959, 1(1): 269-271.
- [106] Bellman R. On a routing problem [M]. Santa Monica, California: RAND Corporation. 1956.
- [107] 段凡丁. 关于最短路径的 SPFA 快速算法 [J]. 西南交通大学学报, 1994, 29(2): 207-212.
- [108] Karp R M. A Characterization of the minimum cycle mean in a digraph [J]. *Discrete Mathematics*, 1978, 23(3): 309-311.
- [109] Floyd R W. Algorithm 97: shortest path [J]. *Communications of The ACM*, 1962, 5(6).
- [110] 王桂平, 等. 图论算法理论、实现及应用 [M]. 北京: 北京大学出版社, 2011.
- [111] Harris T E, Ross F S. Fundamentals of a method for evaluating rail net capacities. *Research Memorandum* [M]. Santa Monica, California: Rand Corporation. 1955.
- [112] Goldberg A V, Tarjan R E. A new approach to the maximum-flow problem [J]. *Journal of the ACM*. 1988, 35(4): 921.
- [113] Ford L R Jr, Fulkerson D R. Maximal flow through a network [J]. *Canadian Journal of Mathematics*, 1956, 8: 399-404.
- [114] Ford L R Jr, Fulkerson D R. *Flows in Networks* [M]. Princeton, NJ: Princeton University Press, 1962.
- [115] Edmonds J, Karp R M. Theoretical improvements in algorithmic efficiency for network flow problems [J]. *Journal of the ACM*, 1972, 19(2): 248-264.
- [116] Dinitz E A. Algorithm for solution of a problem of maximum flow in a network with power estimation [J]. *Soviet Math. Doklady*, 1970, 11(5): 1277-1280.
- [117] Dinitz Y. Dinitz' algorithm: the original version and even's version [C]. *Theoretical Computer Science: Essays in Memory of Shimon Even*. Springer, 2006: 218-240.
- [118] Ahuja R K, Orlin J B. Distance-directed augmenting path algorithms for maximum flow and parametric maximum flow problems [J]. *Naval Research Logistics*, 1991, 38(3): 413-430.
- [119] Ahuja R K, Magnanti T L, Orlin J B. *Network Flows: Theory, Algorithms, and Applications* [M]. Englewood Cliffs, NJ: Prentice Hall, 1993.
- [120] Kuhn H W. The Hungarian method for the assignment problem [J]. *Naval Research Logistics Quarterly*, 1955, 2: 83-97.
- [121] Edmonds J. Paths, trees, and flowers [J]. *Canadian Journal of Mathematics*, 1965, 17: 449-467.
- [122] Berge C. Two theorems in graph theory [J]. *Proceedings of the National Academy of Sciences of the United States of America*, 1957, 43(9): 842-844.
- [123] Hopcroft J E, Karp R M. An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs [J]. *SIAM Journal on Computing*, 1973, 2(4): 225-231.
- [124] Even S, Tarjan R E. Network flow and testing graph connectivity [J]. *SIAM Journal on Computing*, 1975, 4(4): 507-518.
- [125] Gale D, Shapley L S. College admissions and the stability of marriage [J]. *The American Mathematical Monthly*, 1962, 69(1): 9-15.

- [126] Galil Z. Efficient algorithms for finding maximum matching in graphs [J]. *Computing Surveys*, 1986, 18(1): 23-38.
- [127] Gabow H N. An efficient implementation of Edmonds' algorithm for maximum matching on graphs [J]. *Journal of the ACM*, 1976, 23(2): 221-234.
- [128] Mucha M, Sankowski P. Maximum matchings via Gaussian elimination [C]. *Proceedings of 45th Annual IEEE Symposium on Foundations of Computer Science*, 2004: 248-255.
- [129] 杨家齐. 基于线性代数的一般图最大匹配 [C]. IOI 国家集训队论文, 2017.
- [130] Munkres J. Algorithms for the assignment and transportation problems [J]. *Journal of The Society for Industrial and Applied Mathematics*, 1957, 5(1): 32-38.
- [131] Fulkerson D R. Note on Dilworth's decomposition theorem for partially ordered sets [J]. *Proceedings of the American Mathematical Society*, 1956, 7(4):701-702.
- [132] Bron C, Kerbosch J. Algorithm 457: finding all cliques of an undirected graph [J]. *Communications of The ACM*, 1973, 16(9): 575-577.
- [133] Norman R Z, Rabin M O. An algorithm for a minimum cover of a graph [J]. *Proceedings of the American Mathematical Society*, 1959, 10(2): 315-319.
- [134] 冯林, 金博, 于瑞云, 等. 图论及应用 [M]. 哈尔滨: 哈尔滨工业大学出版社, 2012.
- [135] Dreyfus S E. Richard Bellman on the Birth of Dynamic Programming [J]. *Operations Research*, 2002, 50(1): 48-51.
- [136] 方奇. 动态规划 [C]. IOI 国家集训队论文, 2000.
- [137] 崔添翼. 背包问题九讲 [OL]. <https://github.com/tianyicui/pack>, 2012.
- [138] 渡部有隆著, 支鹏浩译. 挑战程序设计竞赛——算法和数据结构 [M]. 北京: 人民邮电出版社, 2016.
- [139] Christian B, Griffiths T. *Algorithms to Live By: The Computer Science of Human Decisions* [M]. New York: Henry Holt and Company, 2016(中译本: 克里斯汀 B, 格里菲思 T. 算法之美. 万慧, 胡小锐, 译. 北京: 中信出版集团, 2018).
- [140] Warren H S. The quest for an accelerated population count. *Beautiful Code* [M]. Sebastopol, California: O'Reilly Media, 2011.
- [141] Kubicka E. The Chromatic Sum and Efficient Tree Algorithms [D]. *Philosophy Doctor Thesis*, Western Michigan University, 1989.
- [142] Guerreiro P. The Canadian Airline Problem and the Bitonic Tour: Is This Dynamic Programming [C]? Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2003.
- [143] 高融. 有关概率和期望问题的研究 [C]. IOI 国家集训队论文, 2004.
- [144] 俞勇. ACM 国际大学生程序设计竞赛: 知识与入门 [M]. 北京: 清华大学出版社, 2013.
- [145] Galil Z, Park K. A linear-time algorithm for concave one-dimensional dynamic programming[J]. *Information Processing Letters*, 1990, 33(6): 309-311.
- [146] 周源. 浅谈数形结合思想在信息学竞赛中的应用 [C]. IOI 国家集训队论文, 2004.
- [147] Knuth D E. Optimum binary search trees [J]. *Acta Informatica*, 1971, 1(1): 14-25.
- [148] Yao F F. Efficient dynamic programming using quadrangle inequalities [C]. *Symposium on the Theory of Computing*, 1980: 429-435.
- [149] Bein W W, Golin M J, Larmore L L, et al. The Knuth-Yao quadrangle-inequality speedup is a consequence of total-monotonicity [J]. *Symposium on Discrete Algorithms*, 2006, 6(1): 31-40.
- [150] Manacher G. A new linear-time 'on-line' algorithm for finding the smallest initial palindrome of a string [J]. *Journal of the ACM*, 1975, 22(3): 346-351.
- [151] Apostolico A, Breslauer D, Galil Z. Parallel detection of all palindrome in a string [J]. *Theoretical Computer Science*, 1995, 141(1-2): 163-173.
- [152] Horn W A. Single-machine job sequencing with treelike precedence ordering and linear delay penalties [J]. *SIAM Journal on Applied Mathematics*, 1972, 23(2): 189-202.
- [153] Galil Z. Applications of efficient mergeable heaps for optimization problems on trees [J]. *Acta Informatica*, 1980, 13: 53-58.
- [154] Schwenk A J. Which rectangular chessboards have a knight's tour? [J]. *Mathematics Magazine*, 1991,

- 64: 325-332.
- [155] Cull P, De Curtins J. Knight's tour revisited. *Fibonacci Quarterly*, 1978, 16: 276-285.
- [156] Conrad A, Hindrichs T, Morsy H, Wegener I. Solution of the knight's Hamiltonian path problem on chessboards [J]. *Discrete Applied Mathematics*, 1994, 50(2): 125-134.
- [157] Squirrel D, Cull P. A Warnsdorff-rule algorithm for knight's tours on square chessboards [R]. Oregon State Research Experience for Undergraduates Program, 1996.
- [158] Ganzfried S. A simple algorithm for knight's tours [R]. Oregon State Research Experience for Undergraduates Program, 2004.
- [159] Hearn D, Baker M P. *Computer Graphics with OpenGL* [M]. London: Pearson Education, 2004.
- [160] de Berg M, Cheong O, van Kreveld M, Overmars M. *Computational Geometry: Algorithms and Applications* [M]. London: Springer, 2008.
- [161] Graham R L. An efficient algorithm for determining the convex hull of a finite planar set [J]. *Information Processing Letters*, 1972, 1(4): 132-133.
- [162] Jarvis R A. On the identification of the convex hull of a finite set of points in the plane [J]. *Information Processing Letters*, 1973, 2(1): 18-21.
- [163] Andrew A M. Another efficient algorithm for convex hulls in two dimensions [J]. *Information Processing Letters*, 1979, 9(5): 216-219.
- [164] Melkman A A. On-line construction of the convex hull of a simple polyline [J]. *Information Processing Letters*, 1987, 25(1): 11-12.
- [165] Pullman H W. An elementary proof of Pick's theorem [J]. *School Science and Mathematics*, 1979, 79(1): 7-12.
- [166] 周培德. 计算几何: 算法设计、分析及应用 [M]. 北京: 清华大学出版社, 2016.
- [167] 朱泽园. 半平面交的新算法及其实用价值 [C]. IOI 国家集训队论文, 2006.
- [168] Shamos M I, Hoey D. Closest-point problems [C]. *Proceedings 16th Annual Symposium on Foundations of Computer Science*, New York, USA: IEEE Computer Society, 1975: 151-162.
- [169] Hocking J G, Young G S. *Topology* [M]. Boston: Addison-Wesley, 1961.
- [170] 刘凯, 夏苗, 杨晓梅. 一种平面点集的高效凸包算法 [J]. 工程科学与技术, 2017, 49(5): 109-116.
- [171] Shamos M I. Computational Geometry [D]. Philosophy Doctor Thesis, Yale University, 1978.
- [172] Toussaint G T. The rotating calipers: an efficient, multipurpose, computational tool [C]. *Proceedings of the International Conference on Computing Technology and Information Management*, Dubai, UAE, 2014.
- [173] Pirzadeh H. Computational geometry with the rotating calipers [D]. Master of Science Thesis, School of Computer Science, McGill University, 1999.
- [174] Freeman H, Shapira R. Determining the minimum-area encasing rectangle for an arbitrary closed curve [J]. *Communications of The ACM*, 1975, 18(7): 409-413.
- [175] Arnon D S, Gieselmann J P. A linear time algorithm for the minimum area rectangle enclosing a convex polygon [R]. Department of Computer Science Technical Reports, 1983: 382.
- [176] 丘维声. 解析几何 [M]. 北京: 北京大学出版社, 2015.
- [177] 金博, 郭立, 于瑞云. 计算几何及应用 [M]. 哈尔滨: 哈尔滨工业大学出版社, 2012.
- [178] O'Rourke J. Finding minimal enclosing boxes [J]. *International Journal of Parallel Programming*, 1985, 14(3): 183-199.