

*Programming Challenges with C++
Advanced Trainning Guide for Programming Contest*

C++，挑战编程

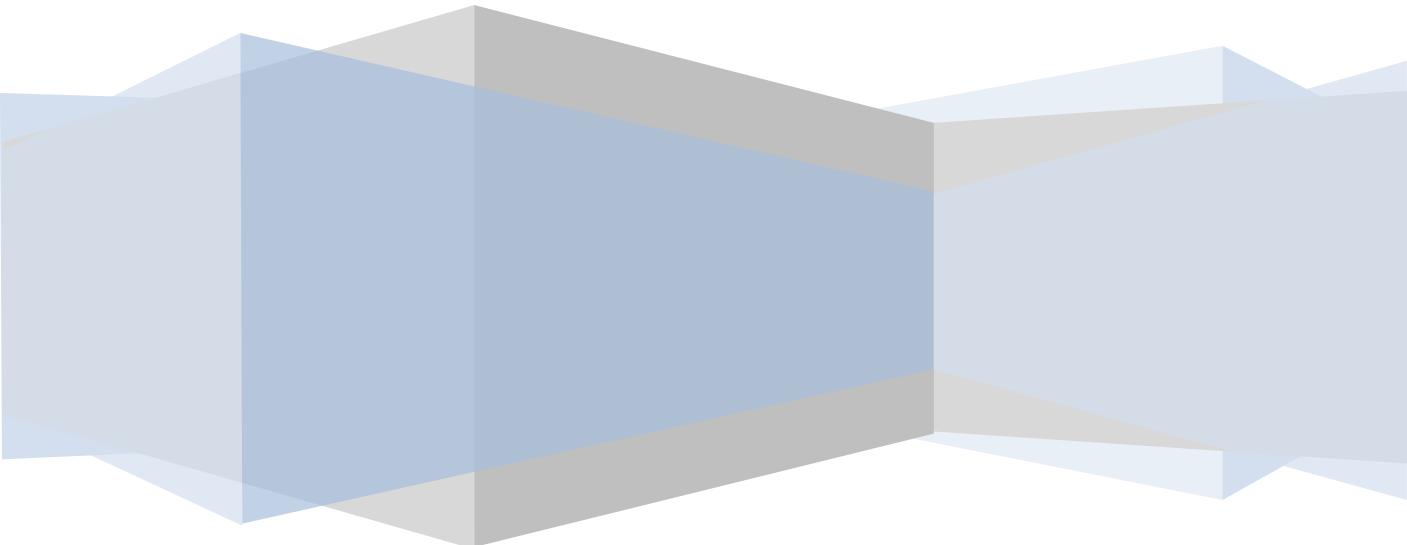
程序设计竞赛进阶训练指南

寂静山林 编著

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

using namespace std;

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



内容简介

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）是国际上公认的水平最高、规模最大、影响最深的计算机专业竞赛，目前全球有 110 多个国家约 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，截至 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，或者是期望通过学习算法和练习以获得进一步的提高大学生。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，涵盖了绝大部分的基本算法。

本书适用于参加 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）的中学生也很有指导价值。此外，作为程序设计、数据结构、算法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》^{II}

在高一的时候，自己就对计算机产生了浓厚的兴趣。那时候是 1998 年，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》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，看看是否有新书可以“免费”阅读。一次偶然的机会，看到书架上有一本机械工业出版社的《C 语言基础教程》^{III}，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^{IV} 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》

^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。通过查阅收录爱因斯坦名言的网页，亦未发现有类似“Interest is the best teacher”的英文字句出现 (<http://www.alberteinsteinsite.com/quotes/einsteinquotes.html>, 2020)。按照常理来推断，有下列几种可能：一种是某人借用爱因斯坦之名写了这段话被大家以讹传讹（或某人误引了他人作品中的文字作为爱因斯坦的名言，不过搜索万维网，发现这段话除了和爱因斯坦相关外，并未和其他名人有关联）；一种是爱因斯坦在非正式场合曾说过这段话，被某人转述并予以广为传播，但出处已不可考；一种是这段话存在于尚未出版的手稿或书信中，未见于公开出版的爱因斯坦文集。望知道确切出处的读者不吝赐教。

^{II} 《论语》由孔子弟子及再传弟子编写而成，至汉代成书，主要记录孔子及其弟子的言行，较为集中地反映了孔子的思想，是儒家学派的经典著作之一。

^{III} 此书不慎遗失，准确名称记忆中已经模糊不清，大致是这个书名。

^{IV} 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（未见官方发布的讣告，故去世原因不详，参见 Revilla 教授的网上讣告：https://www.rememori.com/1018994:miguel_angel_revilla_ramos, 2020）。

^[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)》¹, Halim 的《Competitive 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的提高大学生。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“授人以鱼不如授人以渔”, 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。

感谢父母、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!

邱秋 (寂静山林)

二〇二〇年一月一日于海南儋州

¹ 考虑到在阅读此书后很可能会影响本书的写作, 因此并没有将其作为参考资料, 计划在本书出版后再购买并加以研读, 以便本书再版时加以改进。

前 言

尽信《书》，则不如无《书》。
——《孟子·尽心上》^I

纸上得来终觉浅，绝知此事要躬行。
——陆游，《冬夜读书示子聿》^{II}

一、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），是世界上公认的规模最大、水平最高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。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，有的还经常出资赞助比赛的进行（例如 IBM、Oracle、惠普、微软等公司）。

二、在线评测网站

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），其官方网站为：<https://uva.onlinejudge.org/>。UVa OJ 的特点是题目丰富、题型多样，比较适合中等水平的 ACM 选手进行训练。

三、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地深入学习算法，同时结合习题来提高分析和解决算法问题的能力。

四、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

^I 孟子（约公元前 372 年—约公元前 289 年），名轲，字子舆，战国时期思想家、教育家，儒家学派的代表人物。此句中《书》是指《尚书》中《武成》篇的内容，原文为：孟子曰：“尽信《书》，则不如无《书》。吾于《武成》，取二三策而已矣。仁人无敌于天下，以至仁伐至不仁，而何其血之流杵也？”。意译：孟子说：“完全相信《尚书》，那还不如没有《尚书》。我对于《武成》这一篇书，就只相信其中的二三页罢了。仁人在天下没有敌人，以周武王这样极为仁道的人去讨伐商纣这样极不仁道的人，怎么会血流漂杵呢？”。其意是读书要善于独立思考，不能盲目相信书本。

^{II} 陆游（1125—1210），字务观，号放翁，南宋诗人。这是宋宁宗五年陆游写给儿子子聿的一首劝学诗。

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。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。

五、凡例

一、本书正文：中文字体使用宋体，9.5pt；英文字体使用 Palatino Linotype 字体，9.5pt。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。术语后括号内为其对应的英文名称，使用 Palatino Linotype 字体，9.5pt。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、如果读者已经具备一定的英文阅读能力，建议阅读英文原题。原则上都应该在认真思考后仍无法获得解题思路时才参考题解。每道题目先尝试独立完成，如果确实难以获得解题思路，不妨将此题目搁置，等待一段时间（例如利用一个小时的时间进行休息，或者干脆数周后）再回过来进行思考。当我们尽力去解决一个复杂的或者需要创造性思维的问题时，无论耗费多少精力都找不到正确的思路，在这种时候，暂时停止对问题的积极探索，反而可能会产生关键性的灵感。在心理学中，这种潜意识活动（subconscious work）^[9]称之为酝酿效应（brewing effect）^{III[10]}。对于部分较难（或题目描述容易产生歧义）的题目，利用脚注的形式给出了简要的解题提示（为了保证题目的挑战性，提示尽量做到“点到为止”）。

五、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2020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 ≥ 2000 ；B（偏易）：1999 \geq DACU ≥ 1000 ；C（中等难度）：999 \geq DACU ≥ 500 ；D（偏

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

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

^{III} 心理学家认为，所谓的“酝酿”过程并不是停止思维，而是将原先的整个思维过程转入潜在的意识层面，通过潜意识对储存在记忆里的相关信息进行组合，从而获得类似于“灵感”的思维状态。而这种状态的触发因子就是中途的休息过程。在放下难题之后，大脑消除了前期的心理紧张，忘记了前面不正确的、导致僵局的思路，反而有利于在潜意识层面形成具有创造性的思维状态^[10]。典型的例子如古希腊的阿基米德在浴缸中获得灵感，进而运用浮力原理解决纯金王冠是否被工匠掺入银子的故事（但是经过“考证”，阿基米德的故事很有可能是杜撰出来的，其目的是宣扬阿基米德的天才成就——参阅：曹天元，《上帝掷骰子吗？量子物理史话》，北京联合出版公司，2013年，第181—182页）。

难): $499 \geq \text{DACU} \geq 100$; E (较难): $99 \geq \text{DACU}$ 。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, 不太容易获得通过; 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。

六、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:

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

七、在代码的排版上, 有的 `if` 语句只有一个分支, 为了节省篇幅会将其书写在一行上, 对于 `for` 循环和 `while` 循环, 当其只有一条语句时, 如果该语句较短, 在不影响代码可读性的情况下, 也会将其书写在一行上。示例题目的译文会根据需要进行适当裁剪, 但输入和输出格式一般严格依照原文进行翻译。如果样例输入包含多组输入, 一般只列出第一组样例数据的输入和输出。对于本书中例题的参考代码, 当行数较少时, 会给出完整的参考代码, 否则仅给出关键代码或者省略参考代码, 省略的参考代码请读者从 GitHub 下载。

八、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

^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。

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)。

九、有些题目的标题中包含标点符号 (常见的是逗号、感叹号、问号), 为了排版的美观, 在不引起歧义的前提下, 会将标点符号予以删除。

十、所有示例代码和参考代码均可从 GitHub 网站下载 (<https://github.com/metaphysis/Code>), 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, 1.1.1.2.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/PCC1/01”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, 其后的日期为写作本书时最后一次访问该资源的时间。

目 录

第1章 入门	1
1.1 基本数据类型	1
1.1.1 整数的表示	1
1.1.2 浮点数的表示及精度	2
1.1.3 数据类型的取值范围	4
1.2 格式化输入	6
1.2.1 概述	6
1.2.2 标准输入	7
1.2.3 字符串输入	8
1.3 格式化输出	11
1.3.1 概述	11
1.3.2 输出对齐	13
1.3.3 整数输出	14
1.3.4 实数输出	15
1.3.5 缓冲区与输入输出同步	18
第2章 数据结构	21
2.1 内置数组	21
2.1.1 顺序记录	21
2.1.2 游戏模拟	23
2.1.3 矩阵变换	24
2.1.4 约瑟夫问题	25
2.2 向量	29
2.3 栈	32
2.4 队列及优先队列	37
2.4.1 队列	37
2.4.2 优先队列	40
2.5 双端队列	43
2.6 映射	45
2.7 集合	49
2.8 位集	52
2.9 链表	56
2.10 二叉树	57
2.11 范围最值查询	64
2.11.1 线段树	64
2.11.2 二维线段树	70
2.11.3 区间树	77
2.11.4 树状数组	78
2.11.5 稀疏表	81
2.12 并查集	83
2.13 算法库函数	85
2.13.1 <code>accumulate</code> 和 <code>count</code> 、 <code>count_if</code>	85
2.13.2 <code>copy</code> 和 <code>reverse_copy</code>	86
2.13.3 <code>fill</code>	86
2.13.4 <code>iota</code> ^{C++11}	87
2.13.5 <code>max</code> 和 <code>min</code>	88
2.13.6 <code>max_element</code> 和 <code>min_element</code>	88
2.13.7 <code>memcpy</code> 和 <code>memset</code>	88
第3章 字符串	91
3.1 编码	91
3.2 字符串类	92
3.2.1 声明	93
3.2.2 赋值	94
3.2.3 遍历	95
3.2.4 连接与删除	96
3.2.5 查找与替换	96
3.2.6 其他操作	98
3.3 字符串库函数	98
3.4 字符串类应用	100
3.4.1 文本解析	100
3.4.2 语法分析	104
3.4.3 KMP 匹配算法	108
3.4.4 扩展 KMP 匹配算法	114
3.4.5 Z 算法	116
3.4.6 字符串的最小表示	117
3.5 字符串数据结构及应用	117
3.5.1 Trie	117
3.5.2 Aho-Corasick 算法	119
3.5.3 后缀数组	126
3.5.4 最长公共子串	135
3.5.5 最长重复子串	138
3.5.6 Burrows-Wheeler 变换	138
3.6 正则表达式	140
3.6.1 元字符	141
3.6.2 转义字符	141
3.6.3 数量匹配符和分组	141
3.6.4 字符类和可选模式	142
3.6.5 断言	142
3.6.6 正则表达式类	142
3.7 算法库函数	143
3.7.1 <code>lexicographical_compare</code>	144
3.7.2 <code>next_permutation</code> 和 <code>prev_permutation</code>	144
3.7.3 <code>replace</code>	148
3.7.4 <code>reverse</code>	148
3.7.5 <code>transform</code>	149

第4章 排序与查找	150
4.1 交换排序	150
4.1.1 冒泡排序	150
4.1.2 快速排序	151
4.1.3 中位数	152
4.2 插入排序	154
4.2.1 直接插入排序	154
4.2.2 希尔排序	155
4.3 选择排序	155
4.3.1 直接选择排序	155
4.3.2 堆排序	156
4.4 归并排序	156
4.4.1 逆序对数	157
4.5 计数排序	159
4.6 基数排序	159
4.7 桶排序	160
4.8 查找	161
4.8.1 顺序查找	161
4.8.2 二分查找	162
4.8.3 方程求近似解	163
4.8.4 最大值最小化问题	164
4.9 算法库函数	166
4.9.1 <code>binary_search</code>	166
4.9.2 <code>find</code>	166
4.9.3 <code>lower_bound</code> 和 <code>upper_bound</code>	167
4.9.4 <code>nth_element</code>	170
4.9.5 <code>partial_sort</code>	170
4.9.6 <code>sort</code>	171
4.9.7 <code>stable_sort</code>	174
4.9.8 <code>unique</code>	175
第5章 算术与代数	177
5.1 割鸡焉用牛刀乎?	177
5.2 他山之石, 可以攻玉	183
5.3 高精度整数类的实现	186
5.4 进制及其转换	195
5.4.1 R 进制数转换为十进制数	195
5.4.2 十进制数转换为 R 进制数	195
5.4.3 任意进制数之间的相互转换	196
5.4.4 罗马计数法	199
5.5 实数	200
5.5.1 分数	200
5.5.2 连续分数	203
5.5.3 分数转换为小数	204
5.5.4 小数转换为分数	205
5.5.5 实数大小的比较	205
5.6 代数	206
5.6.1 多项式运算	206
5.6.2 高斯消元法	207
5.7 幂与对数	215
5.8 实数函数库	218
第6章 组合数学	220
6.1 计数原理	220
6.1.1 加法原理	220
6.1.2 乘法原理	221
6.2 排列与组合	223
6.2.1 康托展开和康托逆展开	224
6.2.2 方程的整数解个数	230
6.3 PÓLYA 计数定理	231
6.3.1 基本概念	231
6.3.2 Burnside 引理	236
6.3.3 Pólya 计数定理	240
6.4 鸽笼原理	245
6.4.1 拉姆齐理论	247
6.5 容斥原理	247
6.5.1 错排问题	249
6.6 初等数列	249
6.6.1 等差数列	249
6.6.2 等比数列	249
6.6.3 其他数列	250
6.7 计数序列	250
6.7.1 斐波那契数	250
6.7.2 卡特兰数	254
6.7.3 欧拉数	258
6.7.4 斯特林数	258
6.7.5 调和级数	259
6.7.6 其他序列	260
6.8 概率论	261
6.8.1 基本概念	261
6.8.2 条件概率和独立事件	265
6.8.3 全概率公式与贝叶斯公式	267
6.8.4 随机变量	271
6.8.5 期望	272
6.9 博弈论	281
6.9.1 Nim 游戏	281
6.9.2 Sprague-Grundy 定理	284
6.9.3 Nim 游戏和 Sprague-Grundy 定理扩展	286
6.9.4 PN 态分析	291
第7章 数论	296
7.1 素数	296

7.1.1 素数判定	298	9.4 图遍历的应用	396
7.1.2 Miller-Rabin 素性测试	298	9.4.1 图的连通性	396
7.1.3 高斯素数	301	9.4.2 最短路径	397
7.1.4 生成素数序列	301	9.4.3 最长简单路径	399
7.1.5 素因子分解	304	9.4.4 图的着色	400
7.2 整除性	306	9.4.5 最近公共祖先	400
7.2.1 最大公约数	306	9.4.6 割顶	409
7.2.2 扩展欧几里得算法	309	9.4.7 割边	412
7.2.3 线性同余方程	311	9.4.8 强连通分支	415
7.2.4 最小公倍数	312	9.4.9 半连通分支	422
7.2.5 欧拉函数	313	9.4.10 2-SAT	422
7.2.6 莫比乌斯函数	318	9.4.11 图的直径	428
7.3 模算术	320	9.5 拓扑排序	430
7.3.1 整数拆分	320		
7.3.2 可乐兑换	320		
7.3.3 模运算规则	321		
7.3.4 模的逆元	322		
7.3.5 中国剩余定理	323		
7.3.6 波拉德 ρ 启发式因子分解算法	324		
7.4 日期和时间转换	326		
7.4.1 日期转换	326		
7.4.2 时间转换	333		
第 8 章 回溯法	335		
8.1 八皇后问题	335		
8.2 搜索	347		
8.2.1 单向搜索	347		
8.2.2 双向搜索	353		
8.3 剪枝	354		
8.3.1 正方形剖分	365		
8.3.2 关灯问题	367		
8.4 15 数码问题	368		
第 9 章 图遍历	380		
9.1 基本概念	380		
9.1.1 图的属性	380		
9.1.2 欧拉公式	381		
9.1.3 路与连通	381		
9.2 图的表示	382		
9.2.1 邻接矩阵	382		
9.2.2 边列表和前向星	382		
9.2.3 邻接表	383		
9.2.4 链式前向星	384		
9.3 图遍历	386		
9.3.1 广度优先遍历	386		
9.3.2 深度优先遍历	392		
		第 10 章 图算法	434
		10.1 基本概念	434
		10.1.1 顶点度	434
		10.2 图的回路	436
		10.2.1 欧拉回	436
		10.2.2 中国投递员问题	452
		10.2.3 哈密顿回	454
		10.2.4 旅行商问题	455
		10.3 最小生成树	456
		10.3.1 Prim 算法	456
		10.3.2 Kruskal 算法	458
		10.3.3 最小生成树的扩展问题	460
		10.3.4 单度限制最小生成树	461
		10.3.5 次最优最小生成树	464
		10.4 最短路径问题	467
		10.4.1 Moore-Dijkstra 算法	468
		10.4.2 Bellman-Ford 算法	475
		10.4.3 Floyd-Warshall 算法	480
		10.4.4 传递闭包	482
		10.4.5 最小化的最大距离	485
		10.4.6 差分约束系统	485
		10.5 网络流问题	489
		10.5.1 基本概念	490
		10.5.2 Ford-Fulkerson 方法	491
		10.5.3 Edmonds-Karp 算法	494
		10.5.4 Dinic 算法	499
		10.5.5 ISAP 算法	503
		10.5.6 最小截问题	509
		10.5.7 最小费用最大流问题	510
		10.6 边独立集与二部图匹配	512
		10.6.1 网络流解法	514
		10.6.2 Hungarian 算法	515
		10.6.3 Hopcroft-Karp 算法	522

10.6.4 Gale-Shapley 算法	524	11.9.3 二进制优化	620
10.6.5 Edmonds 算法	526	11.9.4 单调队列优化	621
10.7 二部图加权完备匹配	531	11.9.5 斜率优化	624
10.7.1 网络流解法	531	11.9.6 四边形不等式优化	628
10.7.2 Kuhn-Munkres 算法	532	11.10 子序列和子串问题	631
10.8 点支配集、点覆盖集、点独立集	541	11.10.1 最短编辑距离	631
10.8.1 点支配集	541	11.10.2 最长公共子序列	635
10.8.2 点覆盖集	542	11.10.3 最长公共子串	636
10.8.3 点独立集与最大团	545	11.10.4 最长递增子序列	638
10.9 路径覆盖和边覆盖	546	11.10.5 最长不重复子串	642
10.9.1 最小路径覆盖	546	11.10.6 最长回文子串	644
10.9.2 最小边覆盖	546	11.10.7 最大连续子序列和（积）	647
10.10 树的相关问题求解	546	11.11 贪心算法	649
10.10.1 最小点支配	547	11.11.1 部分背包问题	651
10.10.2 最小点覆盖	547	11.11.2 纸币找零问题	652
10.10.3 最大点独立	548	11.11.3 硬币兑换问题	655
第 11 章 动态规划	550	11.11.4 霍夫曼编码	656
11.1 背包问题	550	11.11.5 最优策略选择	658
11.1.1 01 背包问题	550	第 12 章 网格	660
11.1.2 完全背包问题	555	12.1 矩形网格	660
11.1.3 多重背包问题	555	12.1.1 网格行走	660
11.1.4 背包问题扩展	555	12.1.2 Flood-Fill 算法	662
11.2 备忘	558	12.1.3 国际象棋棋盘	665
11.2.1 3n+1 问题	558	12.1.4 骑士周游问题	668
11.2.2 正交范围查询	561	12.2 三角形网格	674
11.2.3 最大正方形（长方形）	562	12.3 六边形网格	677
11.2.4 整数划分	565	12.4 经度与纬度	678
11.2.5 博弈树	566	第 13 章 几何	680
11.2.6 备忘与递推	570	13.1 点	680
11.3 松弛	579	13.2 直线	681
11.3.1 Moore-Dijkstra 算法	579	13.2.1 直线的表示	681
11.3.2 Bellman-Ford 算法	581	13.2.2 直线间关系	682
11.3.3 Floyd-Warshall 算法	582	13.2.3 相互垂直的两条直线交点	683
11.4 集合型动态规划	585	13.3 坐标和坐标系变换	683
11.5 区间型动态规划	592	13.3.1 平移	683
11.6 图论型动态规划	596	13.3.2 旋转	684
11.6.1 路径计数	600	13.3.3 缩放	686
11.6.2 树形动态规划	601	13.4 三角形	691
11.6.3 旅行商问题	605	13.4.1 勾股定理	691
11.6.4 双调欧几里得旅行商问题	606	13.4.2 三角函数	693
11.7 概率型动态规划	609	13.4.3 正弦定理	694
11.8 非典型动态规划	615	13.4.4 余弦定理	694
11.9 动态规划的优化	618	13.4.5 三角形面积	698
11.9.1 空间优化	618	13.4.6 三角函数库	699
11.9.2 状态优化	619		

13.4.7 桌球碰撞问题	700	14.4.1 最大化矩形问题	746
13.5 多边形	702	14.4.2 矩形并的面积	746
13.5.1 矩形	702	14.4.3 矩形并的周长	749
13.5.2 四边形和正多边形	705	14.5 凸包	752
13.6 圆	705	14.5.1 Graham 扫描法	752
13.6.1 圆的周长和面积	706	14.5.2 Jarvis 步进法	756
13.6.2 圆的切线	708	14.5.3 Andrew 合并法	758
13.6.3 三角形的内切圆与外接圆	710	14.5.4 Melkman 算法	759
13.6.4 圆与圆的位置关系	712	14.6 公式及定理应用	761
第 14 章 计算几何	718	14.6.1 Pick 定理	761
14.1 基本概念	718	14.6.2 多边形面积	761
14.1.1 线段	718	14.6.3 多边形重心	762
14.1.2 多边形	718	10.6.4 三维几何体的表面积和体积	764
14.2 几何对象间的关系	719	14.7 半平面交问题	765
14.2.1 向量、点积和叉积	719	14.7.1 凸多边形切分	765
14.2.2 点和直线的关系	722	14.7.2 多边形内核	771
14.2.3 确定线段转动方向	723	14.8 最近点对问题	772
14.2.4 确定线段是否相交	724	14.9 最远点对问题	776
14.2.5 点的投影	727	14.10 三维空间计算几何	781
14.2.6 点的映像	729	14.10.1 点、直线、平面	781
14.2.7 点和直线间距离	729	14.10.2 三维凸包	786
14.2.8 点和线段间距离	732		
14.2.9 线段和线段间距离	733		
14.2.10 点和多边形的关系	734		
14.2.11 直线和圆的交点	737		
14.2.12 圆和圆的交点	738		
14.2.13 圆的切点	738		
14.3 扫描线算法	739		
14.4 坐标离散化	742		
附录	792		
1 ASCII 表	792		
2 C++运算符优先级	793		
3 若干辅助工具	794		
4 名词索引	795		
5 习题索引	805		
参考资料	829		

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》^I

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

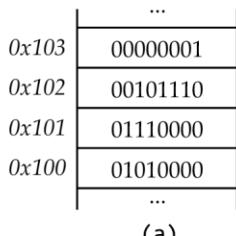
$$1100101101_2 = 1 \times 2^9 + 1 \times 2^8 + \dots + 0 \times 2^1 + 1 \times 2^0 = 813_{10}$$

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^{II}，有

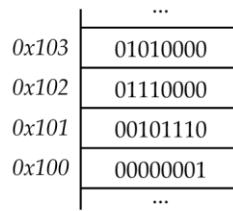
$$s = -x_{w-1} 2^{w-1} + \sum_{i=0}^{w-2} x_i 2^i$$

若 1100101101_2 表示的是一个有符号整数，它所对应的十进制数为

$$1100101101_2 = -1 \times 2^9 + 1 \times 2^8 + \dots + 0 \times 2^1 + 1 \times 2^0 = -269_{10}$$



(a)



(b)

图 1-1 小端序 (a) 和大端序 (b) 两种存储方式，各字节在内存的位置

^I 荀子 (约公元前 316 年—约公元前 237 年)，名况，受尊称为荀卿，又称孙卿，中国战国时代儒家学者和思想家，赵国猗氏 (今山西安泽县) 人，其思想由其弟子编撰为《荀子》一书。

^{II} 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。

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，整数的高位字节存储在内存地址的高位。如图 1-1 所示，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依次为

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

可以看到，存储顺序正好相反。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，只不过是在计算机出现初期时各个硬件厂商在具体实现时所做选择的不同^[11]。

利用 C++ 中的 `union` 数据结构（或者指针）可以很容易地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。
    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。

1.1.2 浮点数的表示及精度

一般的计算机软硬件实现均按照 IEEE (Institute of Electrical and Electronics Engineers, 美国电气与电子工程师协会) 754 标准来表示浮点数。该标准的全称为 IEEE 浮点数运算标准 (Standard for Floating-Point Arithmetic)，于 1975 年首次发布，最新的版本为 IEEE 754-2019^[12]。该标准使用二进制可交换浮点数格式来表示实数^[13]。



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 为偏移指数 (biased exponent)，使用实际指数加上偏移量表示； T 为尾数。默认的 d_0 已经编码到指数位中，故存储的数位从 d_1 开始。32 位浮点数的偏移指数 E 为 8 位，尾数 T 为 23 位，64 位浮点数的偏移指数 E 为 11 位，尾数 T 为 52 位

根据 IEEE 754 标准，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 (mantissa)，规定 $1 \leq m < 2$ ，尾数也称为有效数字 (significand digits)，实际上尾数是有效数字的非正式说法； b 为浮点数所使用的进制基数 (base)，一般使用二进制或十进制，对于二进制浮点数而言， $b=2$ ； e 为指数 (exponent)。例如，将浮点数 -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： $s=1$ ， $b=2$ ， $m=1.1001_2$ ， $e=3$ 。

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 $0.5 \leq m < 1$ 。例如保存 1.01101_2 的时候，可以只保存 01101 ，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， 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。

指数 e 的情况稍复杂。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量 (exponent bias) 以得到表示值 E 。对于 8 位的 e ，这个偏移量是 127；对于 11 位的 e ，这个偏移量是 1023。例如， 2^{10} 的 e 是 10，保存为 32 位浮点数时， $E=10_{10}+127_{10}=137_{10}$ ，即 100000010_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-----
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);
    cout << uis.to_string() << endl;

    return 0;
}
-----1.1.2.1.cpp-----
```

其输出为：

```
1100000101001000000000000000000000000000
```

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，则能够保存的最大尾数为

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

当使用单精度浮点数来保存实数时，如果实数整数部分的有效数字个数超过 8 位，将无法精确保存。注意有

效数字个数是指小数点前后总的数字个数，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，而最多能精确到十位，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。

```
-----1.1.2.2.cpp-----
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 << fixed << f1 << " => " << uis1.to_string() << endl;

    // 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 << fixed << f2 << " => " << uis2.to_string() << endl;

    return 0;
}
-----1.1.2.2.cpp-----
```

其输出为：

```
123456792.000000 => 01001100111010110111100110100011
1234567.000000 => 01001001100101101011010000111000
```

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，能够表示的最大尾数为

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

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 $\log_{10}(2^{24}-1)+1 \approx 8.22$ 位十进制数，`double` 数据类型为 $\log_{10}(2^{53}-1)+1 \approx 16.95$ 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>` 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。

^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。

```

//-----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)";
            }
            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¹：

```

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

```

¹ 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“./”。

最后得到的文件 third.txt 中的内容为^I:

```
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, 需要将数据按照解题的需要以指定的格式读入, 然后再进行处理, 这个过程称为格式化输入。为了示例的方便, 假设已经包含了以下头文件^{II}:

```
// <iostream> 包含了头文件<iostream>和<ostream>, 其中头文件<iostream>包含了 cin 对象,
// 头文件<ostream>包含了 cout 对象。
#include <iostream>

// 由于<iostream>头文件已经包含了<string>头文件, 可以不需再次包含。其他类型的编译器
// 可能并未有此头文件包含关系。
#include <string>

// <sstream> 包含了头文件<iostream>和<ostream>。
#include <sstream>
```

在 C 中, 一般是使用 `scanf` 函数进行读入, 在读入时指定相应的参数, 而在 C++ 中, 只要定义了变量的类型, 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。

`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);`

^I 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。

^{II} 此处是针对 GCC 编译器而言。

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 `getline`，它和 `cin` 中的 `getline` 功能类似，但是它位于头文件 `<string>` 中，并不是头文件 `<iostream>` 中的成员函数。`getline` 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：

```
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。

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样例的组数, 则可以先读入组数, 然后使用 `for` 循环逐组读入数据。

```
int cases;
cin >> cases;
for (int cs = 1; cs <= cases; cs++) {
    // 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, 12798 Handball^B, 12854 Automated Checking Machine^B, 12992 Huatuo's Medicine^C, 13012 Identifying Tea^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有空格, 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。

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;

    // 设置输入时不跳过空白字符和回车换行符以将输入原样输出，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.
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，而单词间以空白字符（空格或制表符）分隔，可使用 `istringstream` 类进行处理。

^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。

```

//-----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。
    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：12243 Flowers Flourish from France^B。

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;
}

```

而在 C++ 中，可以将 `istringstream` 类和 `getline` 函数结合使用对输入进行拆分，更为简便。

```

void parse(string line)
{
    istringstream iss(line);
    // 使用 getline 函数的另一版本，将逗号设置为默认的输入分隔符。
    string block;
    while (getline(iss, block, ',')) cout << block << endl;
}

```

```
//+++++1.2.3.3.cpp+++++//
```

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，默认为标准输出，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上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。

ios::dec	在输入和输出整数时使用十进制格式。
ios::hex	在输入和输出整数时使用十六进制格式。
ios::oct	在输入和输出整数时使用八进制格式。
ios::showbase	在输出整数时添加一个表示其进制的前缀，八进制前加 0，十进制原样输出，十六进制前加 0x。
ios::internal	两端对齐，当长度不足时，在符号位和数值的中间插入若干填充字符以使串两端对齐。
ios::left	左对齐，当长度不足时，在输出的末尾插入填充字符以使得输出左对齐。
ios::right	右对齐，当长度不足时，在输出的前面插入填充字符以使得输出右对齐。
ios::fixed	按定点小数来表示浮点数。
ios::scientific	使用科学计数法来表示浮点数。
ios::boolalpha	输出布尔值时，使用 true 表示 1，false 表示 0。
ios::showpoint	在浮点数表示的数中强制插入小数点。
ios::showpos	在正数前添加符号。
ios::skipws	忽略前导的空格。
ios::unitbuf	在每次输出操作后清空缓存。
ios::uppercase	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，若需使得输出十六进制数时字母字符从

小写字母转换为大写字母，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，可用于输出十六进制数或数制前缀の場合。对于字符串变量，想通过使用 `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

多个格式参数可使用“或”(|) 运算符进行连接，以便一次性设置。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，则可使用: `cout.setf(ios::showbase | ios::uppercase)`。为了更方便的应用，在定义时，标准库将一些参数进行了分组，称之为域(field)，即将类似的格式参数使用“或”运算事先合并在一起。

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

这样可以一次清除多个格式位，以便重设。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”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。

```
boolalpha/noboolalpha    => cout.setf/unsetf (ios::boolalpha)
```

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

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。其中 ios 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)。操纵器通过操纵子对输出的格式进行指定，以下列出了几个常用的操纵器。

setiosflags	设置输出标记。在 ios 头文件中定义了多个标记，这些标记的组合可以控制输出的格式，例如 <code>setiosflags(showbase uppercase)</code> 表示输出时显示基数，且基数以大写方式显示。
setbase	设置输出的基数，dec 表示为十进制，hex 为十六进制，oct 为八进制，例如 <code>setbase(hex)</code> 表示将基数设置为十六进制。
setfill	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 <code>setprecision(6)</code> 表示输出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 <code>setw(10)</code> ，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向左对齐 (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）。它对数值的输出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(20) << 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 `setprecision` 的参数决定保留的小数点位数, `fixed` 操纵子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 (例如 10^{-7}) 再后进行输出, 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
```

如果在输出部分, 所有的实数均需要按指定位数进行四舍五入, 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
```

在处理角度相关问题输出的时候，如果题目要求输出的角度范围在左闭右开区间[0, 360.0)，且要求对输出进行四舍五入处理，此时 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 ，表示此处旅行中的学生人数。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，並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，每个学生平摊支出所需的小总交易金额，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：各个成员所交的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。

设 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

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

$$s = a * t + r, r < t$$

将 t 个学生按花费进行划分，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。要想中间交换的钱尽可能少， 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。

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, 但是如果使用了 `endl`, 则 `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, 进行此操作后, 进行大数据量的输入输出时, 其效率和 C 使用 `scanf` 和 `printf` 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。
    // 或者使用: cin.tie(NULL); cout.tie(NULL); ios_base.sync_with_stdio(false);
    cin.tie(0); cout.tie(0); ios::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很大，可以考虑使用 `tie` 和 `sync_with_stdio`，但应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^{[14][15]}，读者可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;
    }

    // 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。

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，可以作为解题时的参考。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、并查集等，但对某些应用较少的高级数据结构以及相关算法，如二叉堆（binary heap）、左偏树（leftist tree）、平衡二叉树（treap）、伸展树（splay tree）、动态树（dynamic tree）、块状链表、树链剖分等，由于篇幅所限未能给予介绍，建议感兴趣的读者查找相关资料以获得进一步的了解。

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，且不需要频繁对数据进行增加或删除操作时，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 $A[i][j]$ 对应的一维数组元素为 $B[i \cdot n + j]$ ；反之，一维数组的元素 $B[x]$ 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级制进行描述（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) 位于最右侧的培养皿，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， $[0, 0, 0, 0, 0, 0, 0, 0, 0]$ ），而某些则可能立即导致密度爆炸（例如， $[3, 3, 3, 3, 3, 3, 3, 3, 3]$ ）。生物学家感兴趣的是

^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。

那些演化趋势看起来不那么明显的 DNA 程序是如何演变的。

编写程序模拟排成一列共 40 个培养皿中的菌群生长情况，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，按照下述格式进行输出。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的输出的前 10 行（总共应该是 50 行输出），为了可读性，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.bbbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb...bbbbbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb.xbx.bbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb.bb.bb.bbbbbbbbbbbbbbbbb
bbbbbbbbbbbbbbbbbbbb.....bbbbb...bbbbb...bb
bbbbbbbbbbbbbbbb.bb.xbbbbb...x.bbbb...bbbbb
bbbbbbbbbbbb.bb...xx...xx...bbbbb...bbbbb
bbbbbbbbbbbb.bb.WW.xbx.WW.bx.bbbb...bbbbb
bbbbbbbbbbbb.bbb.xxWb.bWxx.bbb.bbbb...bbbbb
```

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。

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，1260 Sales^A，10038 Jolly Jumper^A，10260 Soundex^A，10730 Antiarithmetic^B，11222 Only I Did It^B，11461 Square Numbers^A，11608 No Problem^A，11760 Brother Arif Please Feed Us^C，12150 Pole Position^C，12485 Perfect Choir^D，12583 Memory Overflow^B，12959 Strategy Game^D，13130 Cacho^C。

扩展练习：199 Partial Differential Equations^D，244 Train Time^E，279^I Spin^E，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，应用简单的模运算即可得到下一个位置的序号。

^I 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。

```

// 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!
...\\...\\...\\...\\O.\\O.\\O.\\O.\\O.!
...\\...\\...\\..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，可以得到以下几个约束条件：

- (1) 棋盘最多能放置 9 枚棋子，则 ‘X’ 最多为 5 枚，‘O’ 棋子最多为 4 枚。
- (2) 因为是执 ‘X’ 棋子的为先手，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。
- (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, 11520 Fill the Square^B, 11835 Formula 1^D, 12291^I Polyomino Composer^D, 12398 NumPuzz I^C, 13115 Sudoku^D。

2.1.3 矩阵变换

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。

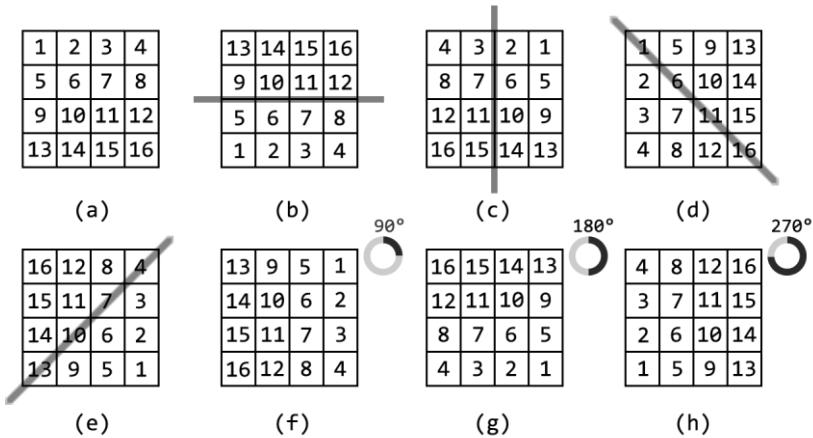


图 2-1 矩阵变换的几种常见形式。(a) 原矩阵；(b) 以水平对称轴进行变换；(c) 以垂直对称轴进行变换；(d) 以主对角线为对称轴进行变换；(e) 以副对角线为对称轴进行变换；(f) 将原矩阵顺时针旋转 90 度；(g) 将原矩阵顺时针旋转 180 度；(h) 将原矩阵顺时针旋转 270 度

矩阵的变换都具有特定的规律，根据规律可以容易的将其表示为代码，关键在于处理时要小心谨慎，特别要注意边界情况的处理。

^I 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，尽管算法逻辑是正确的。

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。

扩展练习: 250^I Pattern Matching Prelims^D, 11055^{II} 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^{III}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乔塔帕塔镇^{IV}。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杀死对方。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杀掉。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, 此人将被杀死, 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, 直到只剩下一个幸存者。

例如当 $n=5$, $k=2$, $i=1$ 时, 被杀者的编号依次为 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 的人会成为幸存者。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

^I 250 Pattern Matching Prelims。注意浮点数精度问题。

^{II} 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增加 (减少) 1, 该方阵仍然是同质的。将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。

^{III} Flavius Josephus (弗拉维斯·约瑟夫, 公元 37 年—公元 100 年), 犹太历史学家。

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

个人，将此人杀死，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 ，以一个静态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，则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;
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;
        }
        // 计数到 k，剩余参与者人数减少 1。
        circle[index] = false;
        nn--;
    }

    // 查找并输出幸存者编号。
    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，此方法能够快速得出结果。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，需要不断的遍历整个数组，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结果加一即可求得正常编号下幸存者的编号），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，不妨将编号重新排列并从 0 开始为剩余的参与者重新赋予编号，即

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

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

可以看到，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 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，利用递推可以很容易得到人数为 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，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，人数为 1 时为最简单的情形，此时剩余参与者的编号为 0。根据以上结果，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;
    for (int i = 2; i <= n; i++)
        last = (last + k) % i;
    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-----
```

¹ 使用后续介绍的 `vector` 替代静态数组，不断移除参与者使得 `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，频繁地删除导致 `vector` 不断调整存储空间，使得耗时增加。

数学方法对于 n 和 k 比较大的情况也可以很快处理¹。如果需要求逆时针方向计数的结果，根据计数过程的对称性，设从 1 开始编号顺时针进行计数得到的幸存者编号为 y ，则逆时针进行计数时，幸存者的编号为

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

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。可以使用一个静态数组，将参与者的序号作为数组元素，每当某人被杀死，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，直到剩下一人，最后根据环形对称得到结果。

参考代码

```
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知道末尾，完成一次循环，以便后续开始每次从起始位置计数。
    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;
```

¹ 参见 Graham、Knuth、Patashnik 合著的《Concrete Mathematics》^[38]，在此书中，对 $k=2$ 的情形进行了详细分析，感兴趣的读者可以进一步查阅。

```

        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*S*H^B, 440 Eeny Meeny Moo^A, 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, 当数组需要包含的元素数量不定时, 使用向量来替代 C++ 的静态数组将会给解题带来便利。由于向量是一个模板类, 需要指定数组元素的类型。数组元素类型可以是整数、浮点数、字符等基本类型, 也可以是字符串、结构体、类。以下代码声明一个元素为 int 类型的向量并压入若干数据, 然后输出其内容。

```

//-----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
----	---	---	---	---	---	---	---	---	---

使用向量需要包含头文件<vector>, 以下列出了向量的一些常用属性和方法^[16]。

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

```
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(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牌堆上。
    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第三堆牌上。
    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^I 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。

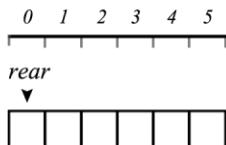
使用栈需要包含头文件<stack>，以下列出了栈的一些常用属性和方法。

empty()	返回栈是否为空。
size()	返回栈的大小。
top()	返回栈顶元素。
pop()	移除栈顶元素。
push	

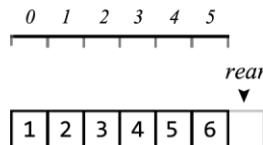
^I 10646 What is the Card。截至 2020 年 1 月 1 日，该题在 UVa OJ 上的题意描述不够清晰，容易导致误解。对于一行 52 张牌的输入数据，第一张牌对应着底部的那张牌，最后一张牌对应着顶部那张牌，起始时，第一张牌在牌堆的最下方，最后一张牌在牌堆的最上方，先将牌堆上半部分的 25 张牌移走待用，对牌堆的下半部分 25 张牌进行 3 次指定的组合操作（这些操作会移走一部分牌），然后将最初移走的 25 张牌放到组合操作后剩下的牌堆顶部，从这个牌堆中的最底下一张牌数起，确定第 Y 张牌即为所求。

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.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.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：+，-，*，/。

^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”。

(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，如加减乘除符号，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

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

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 Lukasiewicz¹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，这样中缀表达式变成

$$+/* 5 - 9 1 4 7$$

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，这样中缀表达式变成

$$5 9 1 - * 4 / 7 +$$

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。

```
// 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)  
{
```

¹ Jan Lukasiewicz 的生平介绍：<http://www.calculator.org/Lukasiewicz.aspx>，2020。

```

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，则输出运算符堆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，表明可以安全的以该运算符作为分割点，将表达式分成两个部分，这样分割不会影响表达式计算结果的正确性。

参考代码

```

// 定义运算符的优先级顺序，值越高，优先级越高。在栈中，括号的优先级最小。
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();
        }
        // 操作符堆栈不为空，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，当运算符堆栈为空，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
    // 运算符堆栈。
    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，
        // 弹出运算符堆栈栈顶元素，直到运算符堆栈为空，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堆栈。
        operators.push(c);
    }
}

// 当中缀表达式处理完毕，运算符堆栈不为空时，逐个弹出压入到操作数堆栈中。
while (!operators.empty()) {
    operands.push(operators.top());
    operators.pop();
}

// 获取操作数堆栈中保存的后缀表达式，注意栈中保存的是从左至右的顺序，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;
}

```

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。

扩展练习: 214 Code Generation^D, 514 Rails^A, 586 Instant Complexity^C, 732 Anagrams by Stack^B。

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，常被用来实现图的广度优先遍历。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>，以下列出了队列的一些常用属性和方法。

empty()	返回队列是否为空。
size()	返回队列的大小。
front()	返回队首元素。
back()	返回队尾元素。

push	将指定的元素追加到队尾。
	<code>void push(const value_type& value);</code>

pop	移除队首元素。
	<code>void pop();</code>

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。

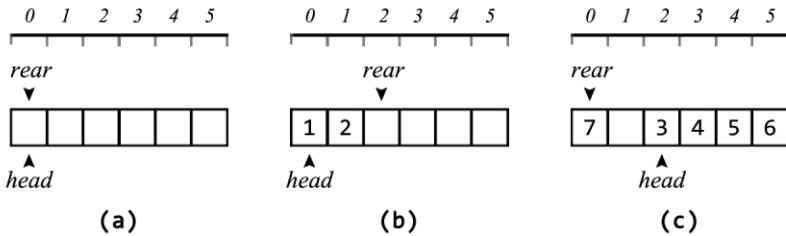


图 2-3 使用数组实现队列，队列容量为 6 个元素。队首指针为 *head*，队尾指针为 *rear*。（a）初始时，队首和队尾重合，队列为空；（b）插入两个元素后，队首位置不变，队尾向后移动；（c）插入和删除若干元素后，队尾已经在队首之前

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，末尾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，此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，只有男性才有接受更高等教育的权利）。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 $8085_{\frac{1}{2}}$ 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了每年设定的 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，表示 N 和 k ，输入以两个 0 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得以解决。具体是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(); 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。

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一些常用属性和方法。

empty()	测试队列是否为空。
size()	获取队列的大小。
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)

注意优先队列只能获取队首元素，且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是使用堆排序实现的，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，则可使用如下几种形式。需要特别注意的是使用结构模板更改优先级时，使用 **greater<T>** 得到的是最小值优先队列，使用 **less<T>** 得到的是最大值优先队列，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），故在排序过程中为了使得较小的元素位于堆顶，在重载小于运算符时要对应地更改元素大小需要满足的关系。

```

//-----2.4.2.cpp-----//
// 默认最大值优先。
priority_queue<int> greaterDefault;

// 使用结构模板 greater<T> 得到最小值优先队列。

```

^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。

```

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长，但由于输出特殊，只要求输出一个丑数，完全可以先运行程序得到丑数后再提交，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，那么可以将生成的丑数继续乘以这三个素因子得到后续丑数。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 Syncing Signals^C, 978 Lemmings Battle^B, 1064 Network^D, 1203 Argus^A, 11621 Small Factors^C, 12100 Printer Queue^B。

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一些常用属性和方法。

empty() 返回队列是否为空，为空返回 `true`，否则返回 `false`。

size() 返回队列的大小。

front() 返回队首元素。

back() 返回队尾元素。

pop_back() 移除队尾元素。

pop_front() 移除队首元素。

clear() 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数量的当选教皇数量，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 行，每一行均包含一个正整数，表示这 P 位教皇各自当选的年份，年份按照

时间顺序给出。其中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
```

(请注意，为节省篇幅，此处未按照题目所要求的格式给出数据，而是将所有数据罗列在一行上)

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。接着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 ，这样将后续年份 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。回顾双端队列的性质，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，也可以使用队列进行解题。使用此种队列方式解题类似于一个“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。

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);
        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>，以下列出了映射的一些常用属性和方法。

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()	返回映射是否为空，为空返回 true ，否则返回 false 。
clear()	清空映射。
[key]	使用下标的形式来获取键 key 对应的值 value 。

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>。

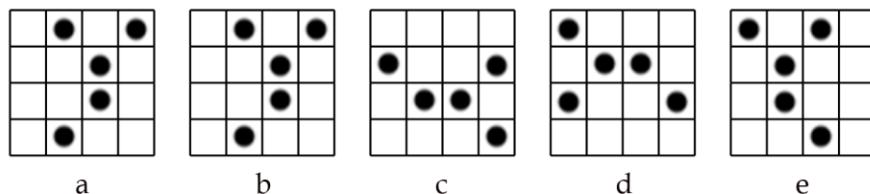
(4) map 不支持通过下标形式对容器中的元素进行访问，而只支持迭代器的访问方式，有时候会为获取指定元素带来不便，需要采取一些变通的方法。例如，需要获取容器中最后一个元素，可以使用如下方式：

```
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 在游戏的早先时候产生，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	13	9	5	1	4	8	12	16	16	15	14	13
5	6	7	8	14	10	6	2	3	7	11	15	12	11	10	9
9	10	11	12	15	11	7	3	2	6	10	14	8	7	6	5
13	14	15	16	16	12	8	4	1	5	9	13	4	3	2	1

a

b

c

d

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;
}

// 生成逆时针旋转 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, 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。

扩展练习: 335 Processing MX Records^D, 506 System Dependencies^D, 698 Index^D, 11860^I 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>`, 以下列出了集合的一些常用属性和方法。

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()	获取指向首个元素之前一个位置的逆序迭代器。
empty()	返回当前集合是否为空, 为空返回 <code>true</code> , 否则返回 <code>false</code> 。
clear()	清空集合。
 insert	
插入元素。	

^I 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>>` 代替, 第二个 map 使用 `unordered_map`, 效率会更高。

```
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

返回集合中第一个不小于指定参数的元素的位置的迭代器。

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

upper_bound

返回集合中第一个大于指定参数的元素位置的迭代器。

```
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 $O(\log n)$ 的时间内给出元素是否在集合中，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，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) 使用类似于静态数组的初始化方式，将数值直接赋予 set 实例，例如：

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

156 Ananagrams^A (非变位词)

大多数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况。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,设原始的单词为 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

```
// 将单词中的大写字母转换为小写字母。
string toLower(string line)
```

^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。

```

{
    for (int i = 0; i < line.length(); i++)
        line[i] = tolower(line[i]);
    return line;
}

int main(int argc, char *argv[])
{
    vector<string> allWords;
    multiset<string> lowerCase;

    // 读入数据。
    string line;
    while (cin >> line, line != "#") {
        allWords.push_back(line);
        line.assign(toLower(line));
        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 = toLower(allWords[i]);
        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^I SQF Problems^D, 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。

扩展练习: 215 Spreadsheet Calculator^D, 789 Indexing^D, 11348 Exhibition^C。

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)

```

^I 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。

```

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。

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。

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

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>，以下列出了位集的一些常用属性和方法。

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

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“01”字符的字符串进行初始化，如果使用包含非“01”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
0000000001100100
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^I Forests^B, 10931 Parity^A, 11926 Multitasking^B。

扩展练习: 10666^{II} 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>`, 以下列出了链表的一些常用属性和方法。

<code>begin()</code>	返回指向链表首个元素的迭代器。
<code>end()</code>	返回指向链表最末元素之后一个位置的迭代器。
<code>rbegin()</code>	返回指向链表最末元素的逆序迭代器。
<code>rend()</code>	返回指向链表首个元素之前一个位置的逆序迭代器。
<code>size()</code>	返回链表的大小。
<code>empty()</code>	返回链表是否为空。为空返回 <code>true</code> , 否则为 <code>false</code> 。
<code>front()</code>	获取链表的首个元素。
<code>back()</code>	获取链表的最末元素。
<code>clear()</code>	清空链表。
<code>pop_back()</code>	删除链表末尾元素。
<code>pop_front()</code>	删除链表首个元素。

^I 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。

^{II} 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。

```

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
移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`，相对于双向链表可以进行双向遍历，`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。

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)。

^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”。

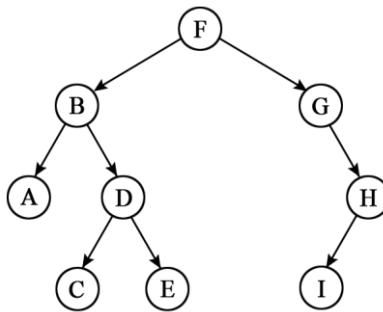


图 2-4 一棵二叉树。结点 F 为根结点, B 为 F 的左子结点, G 为 F 的右子结点, A、C、E、I 均为叶结点

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 (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。

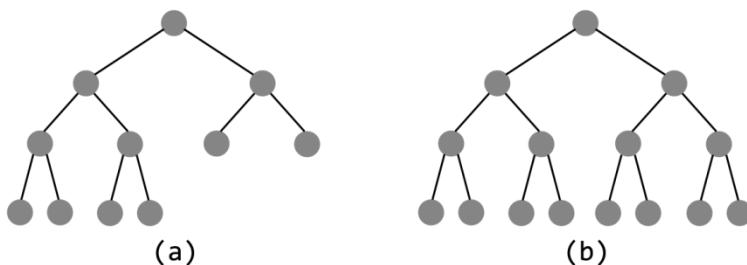


图 2-5 (a) 国际通行定义的满二叉树 (严格二叉树); (b) 国内定义的满二叉树对应国际通行定义的完美二叉树

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
{
    // 结点的权值。
    int 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。

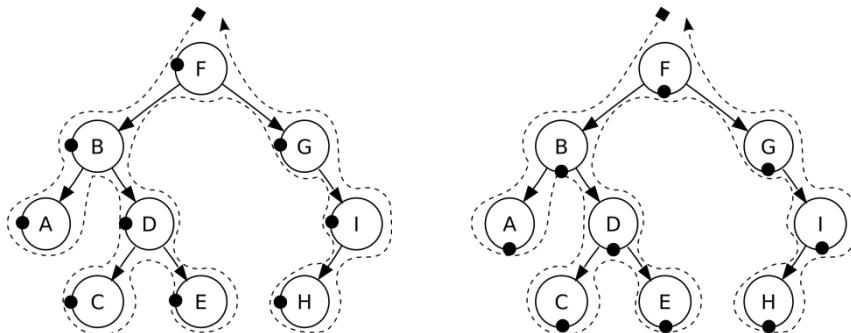


图 2-6 前序遍历 (FBADCEGIH) 和中序遍历 (ABCDEFGH)

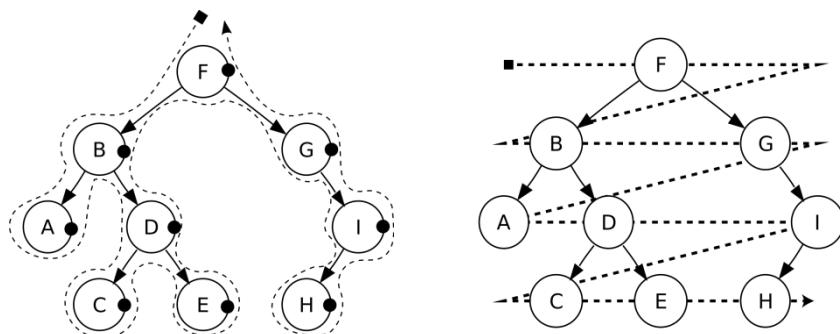


图 2-7 后序遍历 (ACEDBHIGF) 和层序遍历 (FBGADICEH)

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

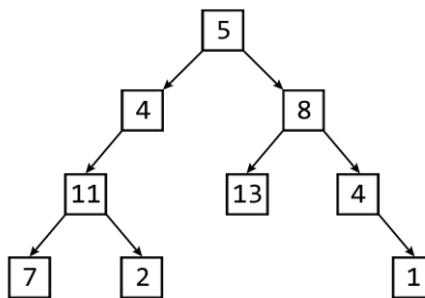
```

// 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->weight;
}
//+++++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和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

样例输出

22 (5(4(11(7())())(2())())())	(8(13())())(4()(1())())())	yes
20 (5(4(11(7())())(2())())())	(8(13())())(4()(1())())())	no
10 (3		yes
(2 (4 () ())		no
(8 () ())		
(1 (6 () ())		
(4 () ()))		
5 ()		

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”，这可以通过在遍历的同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, 11108 Tautology^C, 12347 Binary Search Tree^B。

扩展练习: 839 Not so Mobile^A, 939 Genes^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*[2*i*+1]和 *tree*[2*i*+2]作为其左右儿子结点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。

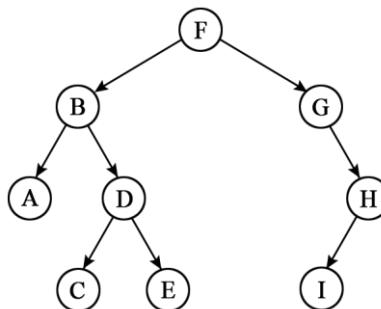


图 2-8 具有 9 个结点的二叉树

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 ABCDEFGHI, 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:

$$[F] \text{ BADCEGHI} \rightarrow (\text{ABCDE}) [F] (\text{GHI})$$

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 GIH 为右子树的遍历结果:

$$(\text{ABCDE}) [F] (\text{GHI}) \rightarrow [F] (\text{BADCE}) (\text{GIH})$$

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 $2i+1$ 。

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 的左儿子结点在遍历结果中一定是紧随 F 之后的，由遍历结果 FBADCEGIH 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……

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

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 GIH 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……

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

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。求解过程可以逻辑划分为四个步骤：(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 范围最值查询

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，显然这一做法效率不高。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]。

2.11.1 线段树

线段树（segment tree）是一种以二叉树为基础的数据结构，它主要用来进行高效的范围最大（小）值查询或者范围和查询。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，则其左右儿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儿子结点（尽管这两个儿子结点在具体应用中可能并不会使用），而区间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，所以在声明空间时，一般以查询区间长度的 4 倍来申请存储空间较为“安全”。

```

//+++++2.11.1.1.cpp+++++//
const int MAXN = 1000010, INF = 0x7f7f7f7f;

#define LCHILD(x) (((x) << 1) | 1)
#define RCHILD(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[LCHILD(p)].field, st[RCHILD(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(LCHILD(p), left, middle);
        build(RCHILD(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来, 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(LCHILD(p), left, middle, qleft, qright);
    int q2 = query(RCHILD(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, 应该根据具体应用进行设置。若查询的是“和”, 则 `INF` 应为 0; 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; 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。更为稳妥的方法是返回具有最小(大)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)
{
}

```

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儿子结点时，将此累积更新量应用到当前结点上，并向其左右儿子结点传递，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。

```

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儿子结点传递更新量。
    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;
}

// 将延迟标记向左右儿子结点传递。
void pushDown(int p)
{
    if (st[p].tag) {
        commit(LCHILD(p), st[p].tag);
        commit(RCHILD(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;
    // 在查询左右儿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(LCHILD(p), left, middle, qleft, qright);
    int q2 = query(RCHILD(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 {
        // 在更新左右儿子结点之前需要将延迟标记向下传递。
        pushDown(p);
        int middle = (left + right) >> 1;
        update(LCHILD(p), left, middle,uleft, uright, utag);
        update(RCHILD(p), middle + 1, right,uleft, uright, utag);
        pushUp(p);
    }
}

```

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。

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，还可以对其灵活改变加以巧妙地运用¹。

(1) 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 `pair<int, int>` 数据类型较为合适，`pair` 实例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，可以使用方法 `combine` 将子区间的结果进行合并。

```

//+++++2.11.1.2.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;
```

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

(2) 查询具有最大和的子区间。给定区间 $a[l, r]$, 要求在此区间中寻找一个子区间 $a[l', r']$, 其中 $l \leq l', r' \leq r$, 使得在区间 $a[l', r']$ 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。

```
//+++++++.11.1.3.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：

(1) 具有最大和的子区间位于左儿子结点；

(2) 具有最大和的子区间位于右儿子结点；

(3) 具有最大和的子区间“跨越”左右儿子结点，即最大和子区间一部分位于左儿子结点所代表的区间内，另外一部分位于右儿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 LCHILD(x) (((x) << 1) | 1)  
#define RCHILD(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[LCHILD(p)], st[RCHILD(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, LCHILD(p), left, middle);  
        build(data, RCHILD(p), middle + 1, right);  
        pushUp(p);  
    }  
}
```

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，或者更复杂地，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构体，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(r_1, c_1, r_2, c_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 轴水平向右，Y 轴垂直向下），即：

```
int mr = (r1 + r2) / 2, mc = (c1 + c2) / 2  
左上角子矩形范围: (r1, c1, mr, mc);  
右上角子矩形范围: (r1, mc + 1, mr, c2);  
左下角子矩形范围: (mr + 1, c1, r2, mc);  
右下角子矩形范围: (mr + 1, mc + 1, r2, c2)
```

在四叉树中，这四个子矩形所对应的树结点是“母矩形”对应树节点的四个儿子结点。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 r1, int c1, int r2, int c2)
{
    // 当范围无效时, 需要正确设置子节点的值, 否则在进行 pushUp 操作时会得到错误的结果。
    if (r1 > r2 || c1 > c2) {
        st[p] = -INF;
        return;
    }
    if (r1 == r2 && c1 == c2) {
        st[p] = data[r1][c1];
        return;
    }
    int mr = (r1 + r2) >> 1, mc = (c1 + c2) >> 1;
    build(4 * p + 1, r1, c1, mr, mc);
    build(4 * p + 2, r1, mc + 1, mr, c2);
    build(4 * p + 3, mr + 1, c1, r2, mc);
    build(4 * p + 4, mr + 1, mc + 1, r2, c2);
    pushUp(p);
}

int query
(int p, int r1, int c1, int r2, int c2, int qr1, int qc1, int qr2, int qc2)
{
    if (r1 > r2 || c1 > c2) return -INF;
    if (r2 < qr1 || c2 < qc1 || r1 > qr2 || c1 > qc2) return -INF;
    if (qr1 <= r1 && r2 <= qr2 && qc1 <= c1 && c2 <= qc2) return st[p];

    int mr = (r1 + r2) >> 1, mc = (c1 + c2) >> 1;
    int q1 = query(4 * p + 1, r1, c1, mr, mc, qr1, qc1, qr2, qc2);
    int q2 = query(4 * p + 2, r1, mc + 1, mr, c2, qr1, qc1, qr2, qc2);
    int q3 = query(4 * p + 3, mr + 1, c1, r2, mc, qr1, qc1, qr2, qc2);
    int q4 = query(4 * p + 4, mr + 1, mc + 1, r2, c2, qr1, qc1, qr2, qc2);
    return max(max(q1, q2), max(q3, q4));
}

void update(int p, int r1, int c1, int r2, int c2, int ur, int uc, int v)
{
    if (r1 > r2 || c1 > c2) return;
    if (r1 == r2 && c1 == c2 && r1 == ur && c1 == uc) st[p] = v;
    else {
        int mr = (r1 + r2) >> 1, mc = (c1 + c2) >> 1;
        if (ur >= r1 && ur <= mr && uc >= c1 && uc <= mc)
            update(4 * p + 1, r1, c1, mr, mc, ur, uc, v);
        else if (ur >= r1 && ur <= mr && uc >= mc + 1 && uc <= c2)
            update(4 * p + 2, r1, mc + 1, mr, c2, ur, uc, v);
        else if (ur >= mr + 1 && ur <= r2 && uc >= c1 && uc <= mc)
            update(4 * p + 3, mr + 1, c1, r2, mc, ur, uc, v);
        else if (ur >= mr + 1 && ur <= r2 && uc >= mc + 1 && uc <= c2)
            update(4 * p + 4, mr + 1, mc + 1, r2, c2, ur, uc, v);
        pushUp(p);
    }
}

```

```

    }
}

//-----2.11.2.1.cpp-----//

```

在上述实现中，使用静态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，而另一个维度较小，如果仍然使用静态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，从而尽可能的节省空间，但是由于是“随用随建”，程序的运行效率要稍低。另外一个可以进行改进的是对范围的操作，可将操作“封装”到一个表示矩形的结构体中，这样可以使得实现更为“井然有序”。

```

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

struct rectangle
{
    int r1, c1, r2, c2;

    rectangle(int r1 = 0, int c1 = 0, int r2 = 0, int c2 = 0):
        r1(r1), c1(c1), r2(r2), c2(c2) {}

    // 测试是否为有效矩形。
    bool isBad() { return r1 > r2 || c1 > c2; }

    // 测试是否已经为单位矩形。
    bool isCell() { return r1 == r2 && c1 == c2; }

    // 测试是否包含指定的单位矩形。
    bool contains(int r, int c)
    { return r1 <= r && r <= r2 && c1 <= c && c <= c2; }

    // 测试是否包含矩形 q。
    bool contains(rectangle q)
    { return r1 <= q.r1 && q.r2 <= r2 && c1 <= q.c1 && q.c2 <= c2; }

    // 测试是否与矩形 q 相交。
    bool intersects(rectangle q)
    { return !(r1 > q.r2 || c1 > q.c2 || r2 < q.r1 || c2 < q.c1); }

    // 返回位于左上角的子矩形。
    rectangle getLU()
    { return rectangle(r1, c1, (r1 + r2) >> 1, (c1 + c2) >> 1); }

    // 返回位于右上角的子矩形。
    rectangle getRU()
    { return rectangle(r1, ((c1 + c2) >> 1) + 1, (r1 + r2) >> 1, c2); }

    // 返回位于左下角的子矩形。
    rectangle getLB()
    { return rectangle(((r1 + r2) >> 1) + 1, c1, r2, (c1 + c2) >> 1); }

    // 返回位于右下角的子矩形。
    rectangle getRB()
    { return rectangle(((r1 + r2) >> 1) + 1, ((c1 + c2) >> 1) + 1, r2, c2); }
};

```

```

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.r1][r.c1];
        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));
}

```

```

}

// 更新线段树。
void update(node *nd, rectangle r, int ur, int uc, int v)
{
    if (r.isBad()) return;
    if (r.isCell() && r.contains(ur, uc)) nd->high = v;
    else {
        if (r.getLU().contains(ur, uc))
            update(nd->children[0], r.getLU(), ur, uc, v);
        else if (r.getRU().contains(ur, uc))
            update(nd->children[1], r.getRU(), ur, uc, v);
        else if (r.getLB().contains(ur, uc))
            update(nd->children[2], r.getLB(), ur, uc, v);
        else if (r.getRB().contains(ur, uc))
            update(nd->children[3], r.getRB(), ur, uc, v);
        pushUp(nd);
    }
}
//-----2.11.2.2.cpp-----

```

二维线段树的嵌套实现

使用“树套树”的实现方法，则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。

```

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

#define LCHILD(x) (((x) << 1) | 1)
#define RCHILD(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[LCHILD(px)][py], st[RCHILD(px)][py]);
    } else {
        int my = (ly + ry) >> 1;
        buildY(px, lx, rx, LCHILD(py), ly, my);
        buildY(px, lx, rx, RCHILD(py), my + 1, ry);
        st[px][py] = max(st[px][LCHILD(py)], st[px][RCHILD(py)]);
    }
}

void buildX(int px, int lx, int rx)
{
    if (lx != rx) {
        int mx = (lx + rx) >> 1;
        buildX(LCHILD(px), lx, mx);
        buildX(RCHILD(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 < qly) return -INF;
    if (qly <= ly && ry <= qry) return st[px][py];
    int my = (ly + ry) >> 1;
    int q1 = queryY(px, LCHILD(py), ly, my, qly, qry);
    int q2 = queryY(px, RCHILD(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(LCHILD(px), lx, mx, qlx, qly, qrx, qry);
    int q2 = queryX(RCHILD(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] = data[lx][ly];
        else st[px][py] = max(st[LCHILD(px)][py], st[RCHILD(px)][py]);
    } else {
        int my = (ly + ry) >> 1;
        if (y <= my)
            updateY(px, lx, rx, LCHILD(py), ly, my, x, y, value);
        else
            updateY(px, lx, rx, RCHILD(py), my + 1, ry, x, y, value);
        st[px][py] = max(st[px][LCHILD(py)], st[px][RCHILD(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(LCHILD(px), lx, mx, x, y, value);
        else
            updateX(RCHILD(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。

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, 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, 1642 **Magical GCD**^D, 11297^I Census^D, 11992^{II} Fast Matrix Operations^D。

^I 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 x1 y1 x2 y2`’ 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。

^{II} 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。

2.11.3 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。

```
-----2.11.3.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);
}
-----2.11.3.cpp-----
```

2.11.4 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 ，需要计算指定范围内元素的和，一种常见的做法是计算数组的“前缀和”—— $P[i] = A[0] + A[1] + \dots + A[i]$ ，得到“前缀和”数组 P 后，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。

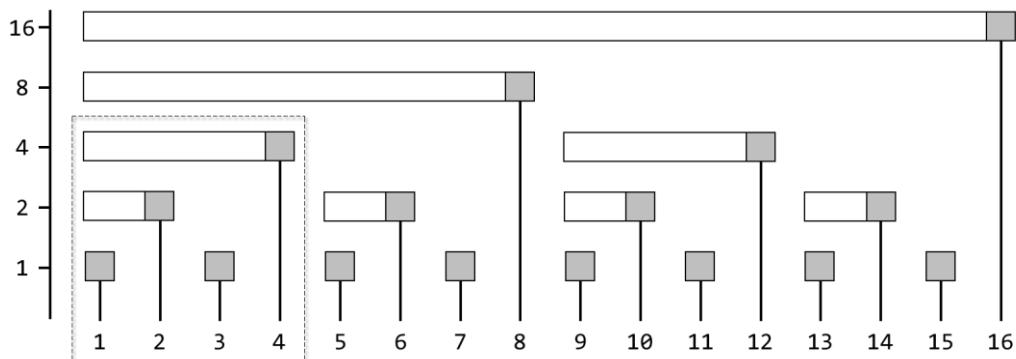


图 2-9 树状数组的简易表示。长条矩形下侧所覆盖的结点被长条矩形右端结点所累加（包括右端结点本身，即阴影方块）。例如（有虚线外框的）树状数组元素 $T[4]$ ，其长条矩形“覆盖”了表示数组元素 $A[1], A[2], A[3], A[4]$ 的结点，表示 $T[4]$ 累加了数组元素 $A[1], A[2], A[3], A[4]$

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。横坐标为数组元素的序号 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 40_{10} ，其二进制表示为 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 $\text{lowbit}(x)$ 。

```
//++++++++++++++2.11.4.cpp+++++++
inline int lowbit(int x) { return x & (-x); }
```

在图 2-9 中，长条矩形对应树状数组的数组 T 所表示的结点的求值范围，例如 $T[4]=A[1]+A[2]+A[3]+A[4]$ ， $T[6]=A[5]+A[6]$ ， $T[7]=A[7]$ 。

^I 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。

初始时构建（或者当 $A[i]$ 改变后需要更新）数组 T 的过程，可以看做从 x 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] = {};

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。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 $O(n \log n)$ 。

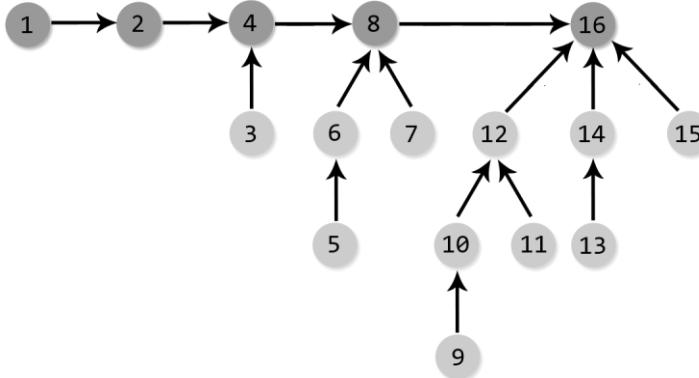


图 2-10 构建（更新）树状数组时，各个数组元素的“累加路径”。箭头所指方向表示数组元素所需要累加到的下一个树结点。例如， $A[3]$ 在累加时，经过 $T[3]$ 、 $T[4]$ 、 $T[8]$ 、 $T[16]$ 四个树结点， $A[5]$ 在累加时，经过 $T[5]$ 、 $T[6]$ 、 $T[8]$ 、 $T[16]$ 四个树结点

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 T 计算 P 的过程，可以看做从 x 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结点的值。

```
int get(int x)
{
    int sum = 0;
    for (int i = x; i; i -= lowbit(i))
        sum += T[i];
    return sum;
}
```

以 $x=11$ 为例，在求和过程中依次累加了 $T[11]+T[10]+T[8]$ ，而 $T[11]=A[11]$ ， $T[10]=A[9]+A[10]$ ， $T[8]=A[1]+A[2]+\dots+A[8]$ ，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 << get(R) - get(L - 1);
    // 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]);
}
//+++++++++++++++++2.11.4.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;
    }
};

```

扩展练习：12028^I A Gift from the Setter^D。

2.11.5 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二进制数，例如，

$$n = 25_{10} = 11001_2$$

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 $[L, R]$ ，可以将其唯一的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。

^I 12028 A Gift from the Setter。对求和计算式进行适当变换后，可以将题目转化为“范围和查询”问题。

根据区间划分的性质，稀疏表先进行预算操作，预算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。在预算过程中，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

$K \geq \text{floor}(\log_2(N)) + 1$, floor 表示向下取整函数

对于一般的应用来说，取 $N=10^7$ ，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 $A[n]$ 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

$$[L, R] = [L, R - 2^j] \cup [R - 2^j + 1, R], \quad j = \log_2(R - L + 1)$$

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。

```

//++++++++++++++++++++++++++2.11.5.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]);
}

```

类似的，使用稀疏表可以求范围和，只需在预算算时将 \min 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 i 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;  
}  
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++2.11.5.cpp++++++++++++++++++++++++++++++++++
```

强化练习：11491 Erasing and Winning^c。

2.12 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。

```
-----2.12.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;
}
//-----2.12.cpp-----//

```

强化练习: 459 Graph Connectivity^A, 599 The Forrest for the Trees^B, 793 Network Connections^A, 1197 The Suspects^B, 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^I Almost Union-Find^C。

2.13 算法库函数

2.13.1 accumulate 和 count、count_if

accumulate 返回指定范围内各元素的累积。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);

```

^I 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。

```
int sum = accumulate(numbers.begin(), numbers.end(), int(0));
```

强化练习：414 Machined Surfaces^A。

2.13.2 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);
```

2.13.3 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);
```

^I 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。

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。

```
-----2.13.3.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;
}
-----2.13.3.cpp-----
```

输出为：

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

2.13.4 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 Iverson¹于 1972 年开发的编程语言——APL (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’。`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。

```
-----2.13.4.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;
```

¹ Kenneth E. Iverson (肯尼斯·艾弗森，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。

```
    cout << '\n';
    return 0;
}
//-----2.13.4.cpp-----//
```

输出为：

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

2.13.5 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：

```
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，例如如下的代码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。

2.13.6 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，节省了时间。

强化练习：499 What's The Frequency Kenneth^A，11364 Optimal Parking^A。

2.13.7 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^D。

`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^[23]。

```
// 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。

```
-----2.13.7.1.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;
}
-----2.13.7.1.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。

```
-----2.13.7.2.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;
}
//-----2.13.7.2.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。

```

//-----2.13.7.3.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;
}
//-----2.13.7.3.cpp-----//

```

以上代码的本意是将数组 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？原因就是 memset 按字节进行填充所致。一个整数占四个字节存储空间，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分别为

$$00000001000000010000000100000001_2 = 16843009_{10}$$

所以一般情况下，通过使用 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 $111111111111111111111111111111_2$ ，在赋值时取其低位字节为 11111111_2 ，最后总的效果刚好把整数数组中元素的所有字节均置为 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。

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，有很多题目是和字符串操作有关的，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位进行互相转换，将大写字母和小写字母进行互相转换等。常用的，数字从 0 到 9 对应的码值为 48 至 57，小写字母 a 至 z 对应的码值为 97 至 122，大写字母 A 至 Z 对应的码值为 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 为 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 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
----	------	----	----

^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。

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 1 的“中日韩统一表意文字”区间 (U+4E00 至 U+9FA5)。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。

扩展练习: 12555 Baby Me^C。

3.2 字符串类

若要在 C++ 中表示一个字符串 (准确的来说是字符序列), 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, 缺点是占用空间较多, 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, 效率一般较使用 `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的编程效率。`string` 类在标准库的源文件中是这样定义的:

```
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++ 进行挑战编程时能用 `string` 类的地方尽量优

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。

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

当已经声明了一个 string 类实例，若需要更改其内容，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

逐个处理字符串中的每个字符，在对字符串的操作中非常常见，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、可维护性，为了节省输入源代码的时间，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，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>	
<code>const char& at(size_t pos) const;</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，如获取 `string` 类实例 `s` 的第一个字符和最末一个字符可以通过使用 `s[0]` 和 `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。

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。
`string& erase(size_t pos = 0, size_t len = npos);`
`iterator erase(iterator p);`

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, 11734 Big Number of Teams Will Solve This^A, 12646 Zero or One^A。

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

^I 列表中只列出了部分常用的方法, 完整的方法列表请读者进一步查阅标准库手册。

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
```

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)` 这样的方法。在 Java 中也有类似的 `String.replace(char oldChar, char newChar)` 方法。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` 类还提供了一系列的操作，方便对字符串进行处理。

<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` 类实例或字符串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，还提供了更多有用的方法。

```
#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 aChar = 'a';
cout << (char)(toupper(aChar)) << 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，但是它们是 C++11 标准的，

在编译时需要使用支持此标准的编译器（或启用支持 C++11 的编译选项）。

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;
}
```

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, 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操作，需要应用本节介绍的字符串的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。对字符串完成解析后，对得到的记录按规则统计后格式化输出。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 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, 11340 Newspaper^A, 11530 SMS Typing^A, 11577 Letter Frequency^A, 11743 Credit Check^A, 11839 Optical Reader^B, 11878 Homework Checker^A, 12543 Longest Word^B, 12626 I Love Pizza^A, 12696 Cabin Baggage^B, 12700 Banglawash^B。

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^I 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

^I 10625 GNU = GNU'sNotUnix。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”。

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，处理时只需逐个位置比对相应的字符是否相同即可判定。

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, 10197 Learning Portuguese^C, 10659 Fitting Text into Slides^D, 10800 Not That Kind of Graph^B, 11074 Draw Grid^B, 11965 Extra Spaces^B, 12155 ASCII Diamond^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^I 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，按照指定规则排序然后输出。

强化练习：169 Xenosemantics^D, 230 Borrowers^C, 642 Word Amalgamation^A, 790^{II} Head Judge Headache^D, 857 Quantiser^D, 10138 CDVII^C, 10194 Football (aka Soccer)^A, 10508 Word Morphing^B, 10698^{III} 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。

^I 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。

^{II} 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。

^{III} 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。

扩展练习: 346 Getting Chorded^D, 425^I Enigmatic Encryption^D, 433 Bank (Not Quite O.C.R.)^D, 613 Numbers That Count^D, 726 Decode^D, 795^{II} Sandorf's Cipher^D, 828 Deciphering Messages^D, 856 The Vigenère Cipher^D, 1091^{III} Barcodes^D, 11697 Playfair Cipher^D, 12134 Find the Format String^D。

(6) 指令解析。题目给定一组规则并有相应的一系列指令, 要求按照指令进行模拟操作。

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

^I 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”。

^{II} 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。

^{III} 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://www.barcodeisland.com/code11.shtml>, 2020。

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) {
            // 尽可能多的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, 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，语法定义一般以下列形式给出：

$$MOD \rightarrow ga \mid ge \mid gi \mid go \mid gu$$

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

$$S \rightarrow L, \quad L \rightarrow S$$

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

$$<predstring> \rightarrow PREDA \mid <predstring> PREDA$$

不同于循环定义，递归定义有出口，而循环定义无出口，因此递归定义是可以正确解析的。

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^[24]。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。

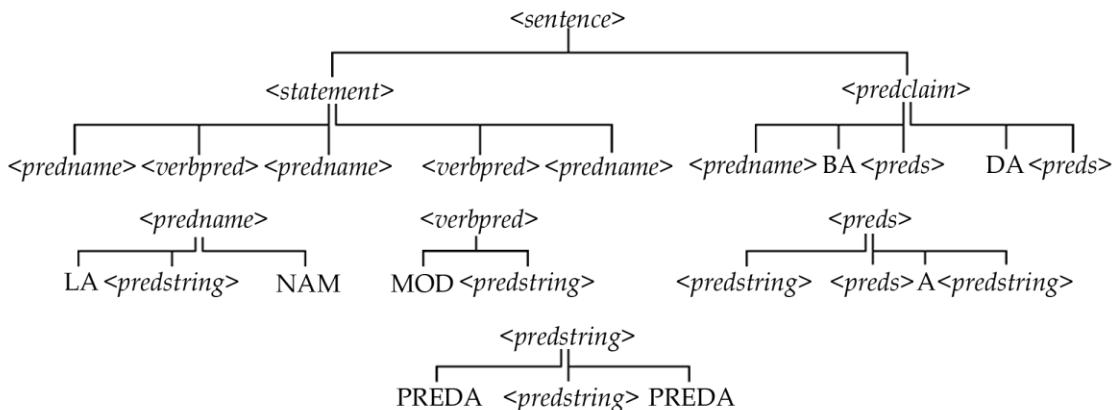


图 3-1 语法解析树（未将所有语法成分表示在一棵树中，而是表示为森林形式）

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 $\langle sentence \rangle$ 。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¹。

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—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 $\langle predname \rangle$ 及后缀 $\langle preds \rangle$ 可将其归约为 $\langle predclaim \rangle$ 。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) {
```

¹ 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>。

```

        int bitOr = 0;
        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^I 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。

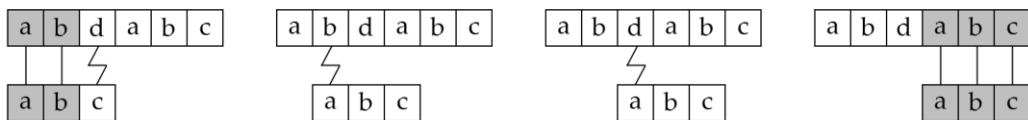


图 3-2 朴素的字符串匹配过程。 $s = "abdabc"$, $p = "abc"$ 。阴影覆盖的方格内为匹配成功的字符, 中间使用直线连接。使用折线连接的为最先匹配不成功的字符。观察朴素字符串的匹配过程, 不难发现如下规律: 模式串在匹配时每失败一次就向右“滑动”一个字符

```
-----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 bf(string &s, 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, 736¹ Lost in Space^D, 850 Crypt Kicker II^A, 886¹¹ Named

¹ 743 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”。

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^{III} Wild Thing^D, 581^{IV} 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。

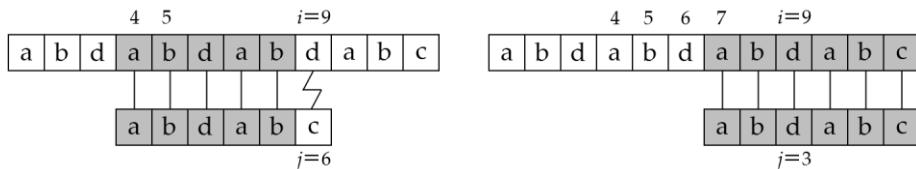


图 3-3 失配时“额外信息”的利用。主串 $s = "abdabdababc"$, 模式串 $p = "abdabc"$, 在 $i=9, j=6$ 处失配。

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^[25]。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 $\Theta(m)$ 的时间内根据模式预先计算得到, 算法总的匹配时间为 $\Theta(n)$ 。

假设主串为 $s_1s_2\dots s_n$, 模式串为 $p_1p_2\dots p_m$, 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”(即 $s_i \neq p_j$) 时, 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, 则模式串中前 $k-1$ 个字符构成的子串必须满足

$$p_1p_2 \cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)} \cdots s_{i-1} \quad (3.1)$$

^I 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。

^{II} 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。

^{III} 475 Wild Thing。给定的模式串中可能包含连续的星号通配符。

^{IV} 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。

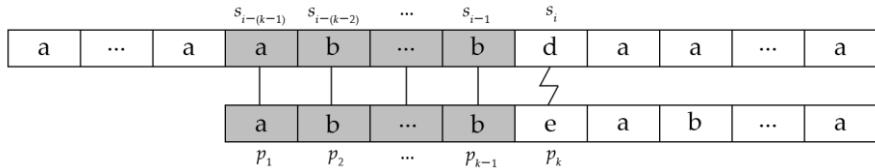
且对于任意的 k' ($k < k' < j$), k' 均不满足关系式 (3.1), 亦即 k 是满足关系式 (3.1) 的最大字符序号。而当前已经得到的“部分匹配”结果是

$$p_{j-(k-1)}p_{j-(k-2)} \cdots p_{j-1} = s_{i-(k-1)}s_{i-(k-2)} \cdots s_{i-1} \quad (3.2)$$

由式 (3.1) 和式 (3.2) 可以得到

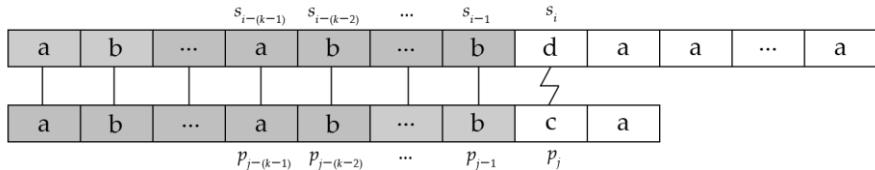
$$p_1p_2 \cdots p_{k-1} = p_{j-(k-1)}p_{j-(k-2)} \cdots p_{j-1} \quad (3.3)$$

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。

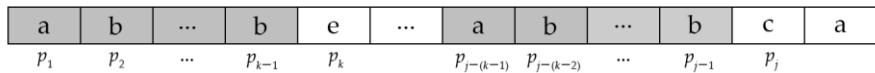


(a) 失配后, 模式串向右“滑动”, 主串的第 i 个字符与模式串的第 k 个字符开始匹配, 有:

$$p_1p_2 \cdots p_{k-1} = s_{i-(k-1)}s_{i-(k-2)} \cdots s_{i-1}$$



(b) 根据失配进行“滑动”之前部分匹配的结果, 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: $p_1p_2 \cdots p_{k-1} = p_{j-(k-1)}p_{j-(k-2)} \cdots p_{j-1}$

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

$$fail[j] = \begin{cases} 0, & \text{当 } j = 1 \text{ 时} \\ \max\{k | 1 < k < j, p_1p_2 \cdots p_{k-1} = p_{j-(k-1)}p_{j-(k-2)} \cdots p_{j-1}\}, & \text{当此集合不为空时} \\ 1, & \text{其他情况} \end{cases}$$

由上述定义可以得到模式串 $p = “abcabcdabcde”$ 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 的初值为 pos , j 的初值为 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} 起和模式串重新开始匹配。

```
//++++++3.4.3.2.cpp
const int MAXN = 1010;

int fail[MAXN] = {};

// 根据 fail 函数进行匹配。
bool kmp(string &s, string &p)
{
    int i = 1, j = 1;
    while (i < s.length() && j < p.length()) {
        if (j == 0 || s[i] == p[j]) i++, j++;
        else j = fail[j];
    }
    return j >= p.length();
}
```

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和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 $p_{j-(k-1)}=p_1$, $p_{j-(k-2)}=p_2$, \cdots , $p_{j-1}=p_{k-1}$, 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。若令 $fail[k]=k'$, 则有 $p_k=p_{j'}$, 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 $p_1 p_2 = p_3 p_4$, 易知 $fail[5] = 3$, 由于 $p_1 p_2 p_3 p_4 = p_7 p_8 p_9 p_{10}$, 易知 $fail[11] = 5$ 。当计算 $fail[12]$ 时, 由于 $p_5 \neq p_{11}$, 但 $p_1 p_2 p_3 = p_9 p_{10} p_{11}$, $p_1 p_2 p_3 = p_3 p_4$, 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_j 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。
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 $p_j=p_k$, 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¹。

```
// 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];
        }
    }
}
```

¹ 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, 依次类推, 直到出现以下两种情况之一: 令 $k' = fail[fail[\cdots fail[j]]]$, 第一种情况是 $fail$ 函数“回退”到某个值使得 $p_j \neq p_{k'}$, 此时 $fail[j] = k'$; 第二种情况是 $fail$ 函数不断“回退”且 $p_j=p_{k'}$, 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”即可达到第一种情况的状态, 使得上述“激进”的优化效果不明显。

```

        } 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++;
            if (p[i] != p[j]) fail[i] = j;
            else fail[i] = fail[j];
        } else j = fail[j];
    }
}

bool kmp(string &s, string &p)
{
    int i = 0, j = 0;
    // 注意: p.length() 的类型为 size_t (默认为 unsigned int)，而 j 可能等于 -1,
    // -1 在 unsigned int 下为 0xFFFFFFFF, 故需要将 p.length() 转换为 int。
    while (i < s.length() && j < (int)p.length()) {
        if (j == -1 || s[i] == p[j]) i++, j++;
        else j = fail[j];
    }
    return j == p.length();
}
//++++++++++++++3.4.3.2.cpp+++++++

```

强化练习：10679^I I Love Strings^A。

^I 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。

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达到 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

$$S[k, p_r] = T[k - p_l, p_r - p_l]$$

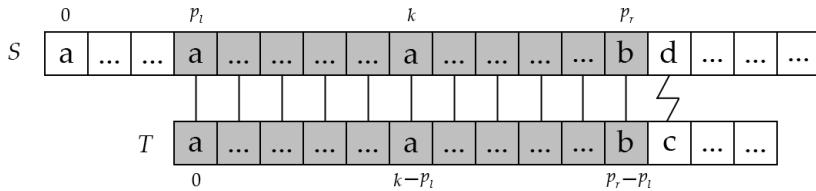


图 3-7 p_r 为之前确定 $extend[0]$ 至 $extend[k-1]$ 的值进行匹配的过程中能够到达的最右匹配位置, p_l 为与 p_r 对应的起始位置, 不难得出: $0 \leq p_l \leq k \leq p_r$

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 $extend[k] = L = next[k-p_l]$ 。

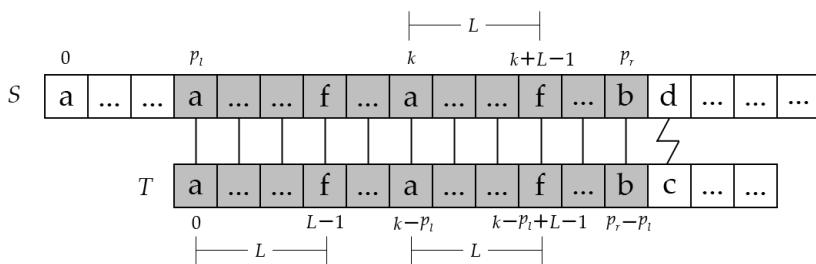


图 3-8 令 $L = next[k-p_l]$, $k+L < p_r$ 的情形, 此时 $extend[k] = L = next[k-p_l]$

(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 $k+extend[k]$ 大于原有的 p_r , 则需要更新 p_r 和 p_l 。

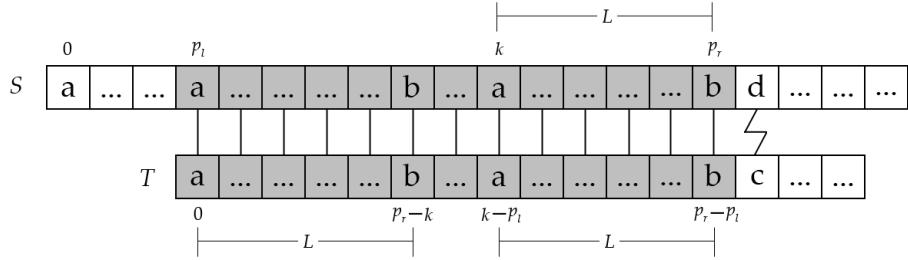


图 3-9 令 $L=next[k-p_l]$, $k+L=p_r$ 的情形, 此时可以从 $S[p_r+1]$ 和 $T[p_r-k+1]$ 开始往后匹配

(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 $extend[k]=p_r-k+1$ 。

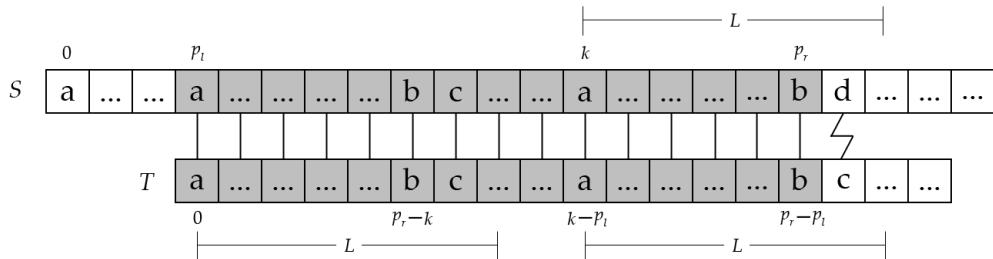


图 3-10 令 $L=next[k-p_l]$, $k+L>p_r$ 的情形, 此时 $extend[k]=p_r-k+1$

根据 $extend$ 数组和 $next$ 数组的定义, 不难看出, 计算 $next[i]$ 的过程和计算 $extend[i]$ 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 $extend[i]$ 的方式计算 $next[i]$, 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 $extend[i]$; 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 为目标字符串的长度。

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 i = 1; i < m; i++) {
        while (pl >= 0 && T[pl] != T[i]) pl = next[pl];
        if (T[pl] == T[i]) next[i] = pl + 1;
    }
}
```

```

        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 $S_iS_{i+1}\cdots S_j$ ， $suffix[i]$ 表示 S 的以 S_i 开始的后缀，即 $S_iS_{i+1}\cdots S_{L-1}$ 。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_i 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 $S_i \neq S_{j-i}$ ；
- (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^I。

强化练习: 12467 Secret Word^D。

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。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^[26]。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++;
        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 + 1;
}
-----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^{II}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 结构结构如图 3-11 所示。

^I 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。

^{II} 名称 Trie 来源于 information reTRIEval。

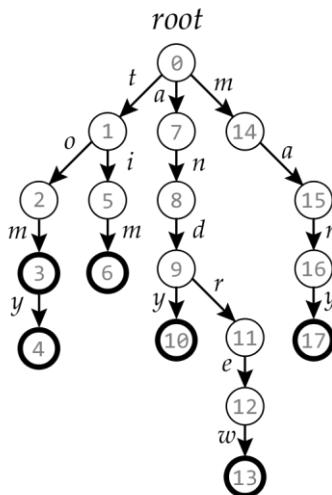


图 3-11 Trie 数据结构示意。从根结点 *root* 到某一结点，将路径上经过的字符连接起来，为该结点所对应的字符串。

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-----//
// CHILDREN 表示每个结点最多可能的儿子结点数量，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 中存于此字符串。
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），则存在较多的空间花销。

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（有向边上的字符具有较小的 ASCII 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的功能。

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^[27]。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 3-12 所示。

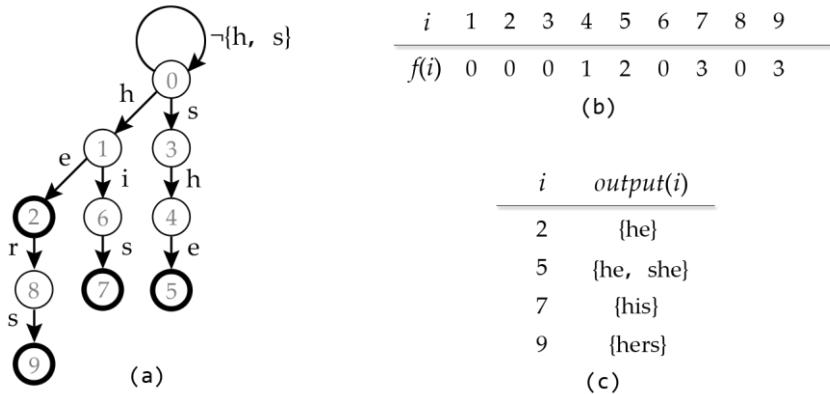


图 3-12 关键字集合 $K=\{\text{he, she, his, hers}\}$ 所对应的模式匹配机。(a) 转移函数, 使用有向有根树予以表示。结点中的数字表示状态的编号。状态 0 为初始状态, 如果当前位于状态 0 且输入字符不属于集合 {h, s}, 则后续转移到状态 0, 相当于失配函数中的 $f(0)=0$ 。(b) 失配函数, 失配函数指示当匹配失败时, 当前状态需要跳转到哪个状态。例如 $f(1)=0$, 当位于状态 1 时, 如果读入的字符不属于集合 {e, i}, 则当前状态应该跳转到状态 0。失配函数的作用是“补全”转移图, 也就是说, 当状态 1 发生失配时, 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 3-12 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, 而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, 继续沿着转移图向下匹配。

如果在转移过程中, 某个状态和输出函数 $output$ 有关联, 即 $output(s) \neq \emptyset$, 则表明在此处发现了一个匹配, 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进行描述。

```
//  $x=a_1a_2\cdots a_n$  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  
     $state \leftarrow 0$   
    for  $i \leftarrow 1$  until  $n$  do  
        begin  
            while  $g(state, a_i) = fail$  do  $state \leftarrow f(state)$   
             $state \leftarrow g(state, a_i)$   
            if  $output(state) \neq empty$  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以下述的伪代码予以表示。

```
// 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  $a_1a_2\cdots a_m$  插入到转移函数所对应的转移图中。  
Enter( $a_1a_2\cdots a_m$ )  
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的状态。
 $output(state) \leftarrow \{a_1 a_2 \dots a_m\}$ 
end

```

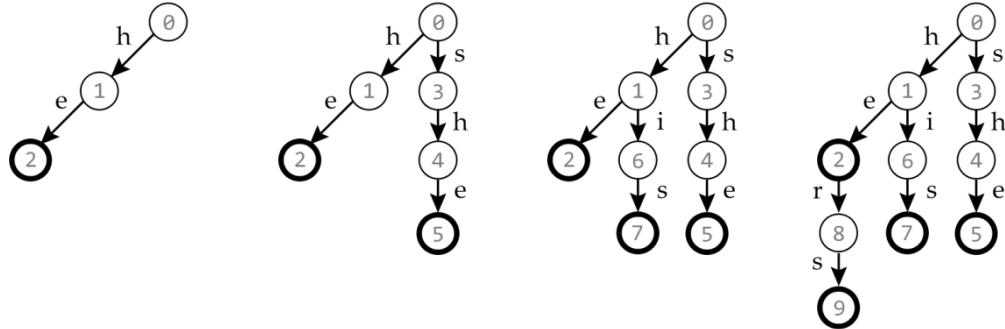


图 3-13 构建转移函数。每次将一个关键字添加到 Trie 中。第一次将关键字“he”添加到 Trie 中，新建 1 和 2 两个结点，结点 2 关联到输出函数；第二次将关键字“she”添加到 Trie 中，新建 3、4、5 三个结点，结点 5 关联到输出函数；第三次将关键字“his”添加到 Trie 中，新建 6 和 7 两个结点，结点 7 关联到输出函数；第四次将关键字“hers”添加到 Trie 中，新建 8 和 9 两个结点，结点 9 关联到输出函数。注意，在将关键字“his”添加到 Trie 中时，“his”和“he”具有最长公共前缀“h”，因此结点 1 为关键字“he”和“his”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 $g(r, a) = fail$ ，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 $g(state, a) \neq fail$ （注意，当位于初始状态 0 时，对于任意输入字符 a 来说，均有 $g(0, a) \neq fail$ ，故使得 $g(state, a) \neq fail$ 的状态总是存在）；

(2.3) 置 $f(s) = g(state, a)$ 。

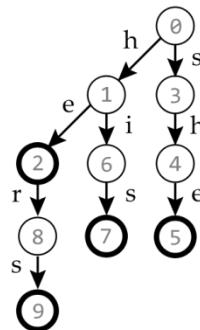


图 3-14 转移函数所对应的转移图

如图 3-14 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的算法伪代码。

```
// 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 ← empty
    for each a such that  $g(\theta, a) = s \neq \theta$  do
        begin
            queue ← queue ∪ {s}
             $f(s) \leftarrow \theta$ 
        end
    end
    while queue ≠ empty do
        begin
            let r be the next state in queue
            queue ← queue - {r}
            for each a such that  $g(r, a) = s \neq fail$  do
                begin
                    queue ← queue ∪ {s}
                    state ←  $f(r)$ 
                    while  $g(state, a) = fail$  do state ←  $f(state)$ 
                     $f(s) \leftarrow g(state, a)$ 
                    output(s) ← output(s) ∪ output(f(s))
                end
            end
        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。

```
const int MAXN = 10240, CHARSET = 128;

class AhoCorasick
{
private:
    int cnt, root;
    int go[MAXN][CHARSET], fail[MAXN];
    vector<string> keywords;
    vector<int> output[MAXN];

    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);
        }
    }

    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:
    void initialize()
    {
        root = cnt = 0;
        keywords.clear();
        memset(go[0], 0, sizeof(go[0]));
        for (int i = 0; i < MAXN; i++) output[i].clear();
    }
}
```

```

void add(string s) { keywords.push_back(s); }

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;
}

```

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^I 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^[28]。相对于后缀树，后缀数组空间消耗较少，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

^I 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。

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
<code>S</code>	e	d	c	b	a	a	b	c	d	e
<code>s</code>	101	100	99	98	97	97	98	99	100	101
<code>cnt</code>	0	0	0	0	0	0	0	0	0	.
<code>ranks</code>	0	1	2	3	4	5	6	7	8	9
<code>sa</code>	-	-	-	-	-	-	-	-	-	.

图 3-15 字符串 $S = "edcbaabcde"$ 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 3-16 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 $z = \sum_{y < x} \text{cnt}[y]$, 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 $z' = \sum_{y \leq x} \text{cnt}[y]$, 如果从 1 开始计数位置, 则次数 $z' = \sum_{y \leq x} \text{cnt}[y]$ 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 3-17 根据累加的字符频次确定各单个字符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 的元素其值为 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, 字符 ‘d’ 最后的位置必定位于第 8 位, 由于后缀数组从 0 开始计数, 对应的序号需要减 1, 即可得后缀 *s*[8, 1] 在后缀数组中的值 *sa*[‘d’] = *sa*[7] = 8, 即在后缀数组中, 位于第 7 位的是从位置 8 开始的后缀 *s*[8, 1]。其他后缀 *s*[*i*, 1] 所对应的后缀数组值可以依此类推得出

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 3-18 根据后缀数组 sa 获得名次数组 $ranks$ 。实际上是将后缀 $s[i, 1]$ 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 $s[i, 1]$ 对应的高位数字和 $s[i+1, 1]$ 所对应的低位数字。参考实现中使用下述代码确定高位数字和低位数字的具体值：

```

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字和低位数字，此处数组 $higher$ 存储的是高位数字，数组 $lower$ 存储的是低位数字。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 3-19 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 3-17 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。

	0	1	2	3	4	5	6	7	8	9
lower	4	3	2	1	1	2	3	4	5	0
sa	0	1	2	3	4	5	6	7	8	9
↓										
countSort(lower, sa, ranks, n, n);										
↓										
memset(cnt, 0, sizeof(cnt));										
cnt	0	0	0	0	0	0	0	0	0	0
for (int i = 0; i < n; i++) cnt[lower[sa[i]]]++;										
cnt	1	2	2	2	2	1	0	0	0	0
for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];										
cnt	1	3	5	7	9	10	10	10	10	10
for (int i = n - 1; i >= 0; i--) ranks[-cnt[lower[sa[i]]]] = sa[i];										
ranks	9	3	4	2	5	1	6	0	7	8
	9	5	4	2	5	1	6	0	7	8
S[i, 2]	e	b	a	c	a	d	b	e	c	d
	-	a	a	b	b	c	c	d	d	e

(a)

	0	1	2	3	4	5	6	7	8	9
higher	5	4	3	2	1	1	2	3	4	5
ranks	9	3	4	2	5	1	6	0	7	8
↓										
countSort(higher, ranks, sa, n, n);										
↓										
memset(cnt, 0, sizeof(cnt));										
cnt	0	0	0	0	0	0	0	0	0	0
for (int i = 0; i < n; i++) cnt[higher[ranks[i]]]++;										
cnt	0	2	2	2	2	2	0	0	0	0
for (int i = 1; i <= m; i++) cnt[i] += cnt[i - 1];										
cnt	0	2	4	6	8	10	10	10	10	10
for (int i = n - 1; i >= 0; i--) sa[--cnt[higher[ranks[i]]]] = ranks[i];										
sa	4	5	3	6	2	7	1	8	9	0
	4	5	5	6	2	7	1	8	9	0
S[i, 2]	a	a	b	b	c	c	d	d	e	e
	a	b	a	c	b	d	c	e	-	d

(b)

图 3-20 (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^[29], 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，因而使得这 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子串 $s[k, k]$, $s[k+1, k]$, ..., $s[n-1, k]$ ，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”，因为 sa 保存的就是 $s[0, k]$, $s[1, k]$, $s[2, k]$, ..., $s[n-1, k]$ 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 $n-k$, $n-k+1$, ..., $n-1$ ，那么剩下的 $n-k$ 个位置需要填充 0, 1, ..., $n-k-1$ ，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^[30]，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。

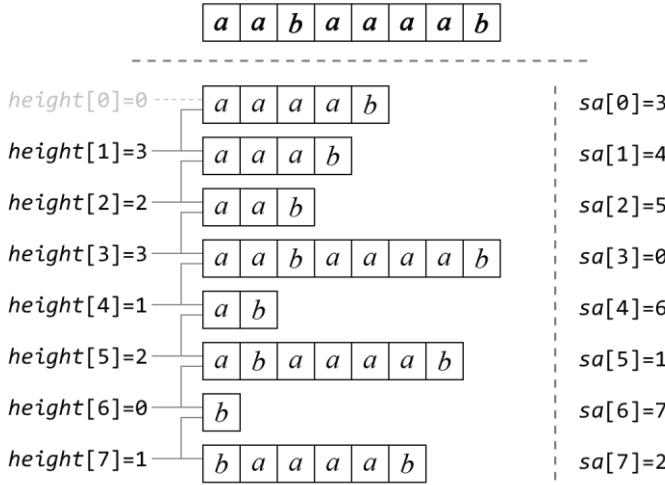


图 3-21 字符串 $s = aabaaaab$ 所对应的后缀数组 sa 和 $height$ 数组。示例: $s_{sa[0]} = aabaaaab$, $s_{sa[3]} = aabaaaab$, 则 $LCP[s_{sa[0]}, s_{sa[3]}] = \min\{height[1], height[2], height[3]\} = 2$

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^[31]。

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++++++
```

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

^I 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。

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^I Top 10^D, 10526^{II} 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

^I 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) 构建后缀数组，结合线段树进行查询。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。(3) 使用后缀 Trie 并构建 AC 自动机予以解决，运行效率相对较高，但编码难度也相对较高。

^{II} 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。

Burrows 和 David Wheeler 于 1994 年发明^[32]。

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.EUSFXDII0IIIT
```

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
^	B	^B	BA
A	N	AN	NA
A	N	AN	NA
	^	^	^B
A		A	^

附加 3： 排序 3 附加 4 排序 4

BAN	ANA	BANA	ANAN
NAN	ANA	NANA	ANA
NA	A^	NA^	A ^B
^BA	BAN	^BAN	BANA
ANA	NAN	ANAN	NANA
ANA	NA	ANA	NA^
^B	^BA	^BA	^BAN
A^	^B	A ^B	^BA

附加 5： 排序 5 附加 6 排序 6

BANAN	ANANA	BANANA	ANANA
NANA	ANA ^	NANA ^	ANA ^B
NA ^B	A ^BA	NA ^BA	A ^BAN
^BANA	BANAN	^BANAN	BANANA
ANANA	NANA	ANANA	NANA ^
ANA^	NA ^B	ANA ^B	NA ^BA
^BAN	^BANA	^BANA	^BANAN
A ^BA	^BAN	A ^BAN	^BANA

附加 7： 排序 7 附加 8 排序 8

BANANA	ANANA ^	BANANA ^	ANANA ^B
NANA ^B	ANA ^BA	NANA ^BA	ANA ^BAN
NA ^BAN	A ^BANA	NA ^BANA	A ^BANAN
^BANANA	BANANA	^BANANA	BANANA ^
ANANA ^	NANA ^B	ANANA ^B	NANA ^BA
ANA ^BA	NA ^BAN	ANA ^BAN	NA ^BANA
^BANAN	^BANANA	^BANANA	^BANANA
A ^BANA	^BANAN	A ^BANAN	^BANANA

输出： **^BANANA|**

根据原始字符串的序号 6 即可确定原始字符串为 “**^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^[33]。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’ 的意义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为指定的子串。

```
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，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况：(1) 在比较的过程中，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) 若 $x_i > y_i$ ，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 ‘a’ < ‘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（`reverse`）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。

SGI 标准库实现中的 `next_permutation` 算法很巧妙：从最尾端开始寻找两个相邻的元素，令第一个

元素是 i , 第二个元素是 ii , 且满足 $*i < *ii$, 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 i, j 元素对调, 然后将 ii 之后的所有元素逆序排列, 就得到了下一个组合。库实现源代码如下:

```
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;
        }
    }
}
```

代码很短, 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 i 与从后往前第一个大于它的元素交换, 并且颠倒 $[i+1, last]$ 的元素就可以得到下个组合。根据代码可以知道 $[first, i]$ 这部分元素与原来的序列相同, 把 i 与从后往前第一个大于它的元素 j 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元素对, 这就可以肯定 $[ii, last]$ 这部分数据是逆序的, 有 $*(j-1) \geq *j \geq *(j+1)$, 而 $*j$ 是从后往前第一个大于 $*i$ 的元素, 故 $*(j-1) \geq *j > *i \geq *(j+1)$, 所以 $*i$ 与 $*j$ 互换后 $[ii, last]$ 这部分数据仍然是逆序的, 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。

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
```

^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。

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
cbbaa
#
```

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 Bandwith^A，195 Anagram^A，234 Switching Channels^D，729 The Hamming Distance Problem^A，1209^I 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);
```

^I 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”。

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;
    // 将所有英文字母转换为大写。
    transform(s.begin(), s.end(), s.begin(), ::toupper);
    cout << s << endl;
    // 将所有英文字母转换为小写。
    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.
```

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

^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 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¹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, 需要考虑许多边界的情况。

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-----//
void quickSort(int data[], int left, int right)

```

¹ Charles Antony Richard Hoare (Tony Hoare 或 C.A.R. Hoare, 查尔斯·安东尼·理查德·霍尔, 1934—), 英国计算机学家, 1980年图灵奖获得者。

```

{
    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

$$m = \frac{a_{\lfloor n/2 \rfloor} + a_{\lfloor n/2 \rfloor + 1}}{2}$$

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^I。

强化练习：10041 Vito's Family^A。

^I 参见本章第 4.9 节“库函数”中第 4.9.4 小节“`nth_element`”的内容。

4.2 插入排序

4.2.1 直接插入排序

直接插入排序 (straight insertion sort)，其算法思想为：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，因而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--;
        }
    }
}
```

^I 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。

```

        }
        // 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^[34]。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 $\Theta(n \log^2 n) \sim \Theta(n^2)$ ，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

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])

```

^I Donald L. Shell (唐纳德·希尔，1924—2015)，美国计算机科学家。

```

        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 Football^c。

4.4.1 逆序对数

令人感到惊奇的是，归并排序竟然能够与数列通过交换相邻元素达到有序状态的最小交换次数产生关联。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。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。

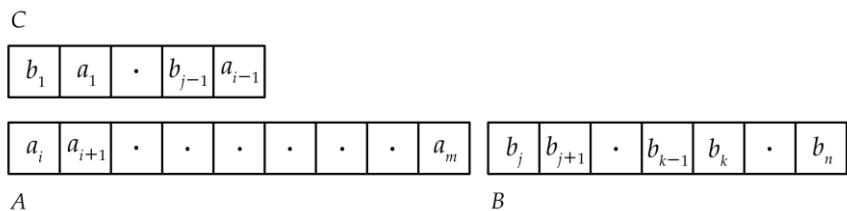


图 4-1 确定逆序对数。子序列 A 是归并前位于左侧的已排序子序列，子序列 B 是归并前位于右侧的已排序子序列，子序列 C 是归并后的结果序列，假设 $b_j < a_i$ (如果 $a_i < b_j$ 可以同理论证)，则将 b_j 附加到结果序列 C 中时， b_j 和子序列 A 中从 a_i 到 a_m 的元素构成逆序对，因此 b_j 对逆序对数的贡献为 $m - i + 1$

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^[35]。

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 tmp[MAXN];

long long mergeAndCount(int data[], 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++] =
            data[i] <= data[j] ? data[i++] : (count += middle + 1 - i, 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;
}

long long mergeSort(int data[], int left, int right)
{
    long long count = 0;
    if (left < right) {
        int middle = (left + right) >> 1;
        count += mergeSort(data, left, middle);
        count += mergeSort(data, middle + 1, right);
        count += mergeAndCount(data, left, middle, right);
    }
    return count;
}
//-----4.4.1.cpp-----//
```

强化练习: 10810 Ultra-QuickSort^A, 11495 Bubbles and Buckets^B, 11858 Frosh Week^C。

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， 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。在第二次遍历时，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，直到达到数据的最高位，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

$$(K/B) * i \leq x < (K/B) * (i + 1)$$

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（或使用其他基本排序方法），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（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）。由于是逐个元素比较，顺序查找的时间复杂度为 $O(n)$ 。

```

//-----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¹。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：

(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)

```

^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。

```
{  
    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^[36]。

```
// 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。

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^[37]。

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

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

样例输出

```
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。

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，11881 Internal Rate of Return^D，11935 Through the Desert^C，12032 The Monkey and the Oiled Bamboo^A，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)
{
    // 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。

扩展练习: 11516 WiFi^B, 12255 Underwater Snipers^E。

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, 否则返回指定范围的结束。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排序，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>
_ForwardIterator
__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排序，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;

    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 $f(1), f(2), f(3), \dots$ 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
```

样例输出

```
21
```

分析

显式的生成这个序列显然是超出内存限制也是没有必要的。有两种方法，一种是根据递推关系计算，一种是根据序列特点得到指定的 $f(n)$ 值。

递推方法：设 $G(n)$ 表示满足 $f(k) > f(k-1)$ 时的自变量值，观察序列可以知道 $G(1)=2, G(2)=4, G(3)=6, G(4)=9, \dots$ ，由序列的定义可知左闭右开区间 $[G(n-1), G(n))$ 有 $f(n)$ 个 n ，那么

$$G(n) = f(1) + f(2) + f(3) + \dots + f(n) + 1$$

即

$$G(n) = G(n-1) + f(n)$$

由于已知区间 $[G(n-1), G(n))$ 内的数其函数值都是 n ，可以得到计算 $G(n)$ 的递推关系式： $G(1)=2, G(2)=4, G(3)=6, \dots$ ，对于 $[G(k), G(k+1))$ 之间的数，其 $f(n)=k+1$ ，则有 $G(n)=G(n-1)+k+1$ 。

非递推方法：根据以下规律，从 $n=3$ 开始对于序列中的每一对 $f(n)=k$ ，必将在序列中添加 $f(n) \times k$ 个元素。可以使用一个队列来记录添加的元素，同时计算将要添加的元素总数，直到该队列将要产生的元素数量超过指定数量。根据情况判定要求的 n 值的函数值。具体方法如下：

(1) $n=1$ 或 $n=2$, 特例, 予以特殊处理。

(2) 设 Q 为队列, 用于记录序列的元素, 初始时, 队列中存储了 $n=3$ 时的函数值 $f(3)=2$ 。变量 t 根据 $f(n) \times k$ 的关系记录当前队列中存储元素所能产生的元素总量, 初始值为 0。变量 n 为当前的自变量值。

(3) 从队列中取出第一个元素, 设其值为 f_n , 可知 $f(n)=f_n$ 。则队列所能产生的元素个数为 $t+f_n \times n$ 。比较当前 n 与输入中要求函数值的数 k , 若相等则表明 $f(k)=f_n$ 。否则继续计算 t 直到上限值。

(4) 当前序列已经产生了足够的元素, 最后能生成指定上限的元素, 则继续从队列首位取出一个元素, 按前述步骤, 只不过这时不需要实际在队列的最后添加元素, 而是计算 t 的变化范围, 在 $[t, t+f_n)$ 之间数的函数值都是刚弹出的元素所对应的 n 值。同时仍应比对队列首的 n 值, 若有值对应, 则该数的函数值就是弹出的队列元素值 f_n 。

(5) 继续处理直到找到所有输入数的函数值, 然后输出结果。

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, 使得在指定位置序号 n 的元素恰好为排序好的数组中的第 n 小元素, 位置从 0 开始计算, 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。因为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 = "1354968720";

    nth_element(line.begin(), line.begin() + 5, line.end());
    cout << line << endl;

    return 0;
}
//---------------------------------4.9.4.cpp-----//
```

输出为:

```
3102456789
```

强化练习: 501 Black Box^B。

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

^I Regent，瑞金特，图书销售公司，总部位于美国新泽西州。

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，每个事件都有相对的重要性提示。

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

```
1993
A 23 12 5 Partner's birthday
A 25 12 7 Christmas
A 20 12 1 Unspecified Anniversary
D 20 12
#
```

样例输出

```
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, 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, 12210 A Match Making Problem^B, 12346 Water Gate Management^D, 12406 Help Dexter^C, 12541 Birthdates^B。

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。

第 5 章 算术与代数

他山之石，可以攻玉。
——《诗经·小雅·鹤鸣》¹

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);
```

¹ 诗经，是中国最古老的一部诗歌总集，它收录了从西周初年到春秋中年（公元前 1100 年—约公元前 600 年）的诗歌 300 余首。诗经在内容上分为《风》、《雅》、《颂》三个部分。《风》是周代各地的歌谣；《雅》是周人的正声雅乐，又分《小雅》和《大雅》；《颂》是周王庭和贵族宗庙祭祀的乐歌，又分为《周颂》、《鲁颂》和《商颂》。

```

// 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上，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， $X=421052631578947368$ 是一个双重麻烦数，因为当对 X 进行“右移”操作后会得到： $842105263157894736=2X$ 。

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 和 0101 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 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。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
```

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^A <--> smordnilaP^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^I Dear GOD^D，10669^{II} Three Powers^C。

5.2 他山之石，可以攻玉

Java 中的 BigInteger 类支持大数运算的相关功能^{III}。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。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，常用的方式是将一个十进制的字符串转换为大整数，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）。
```

^I 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。

^{II} 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。

^{III} 参阅：<https://docs.oracle.com/javase/7/docs/api/java/math/BigInteger.html>，2020。

```
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);
```

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

```
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;
    }
}
```

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, 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。

扩展练习: 10023^I Square Root^B, 10606^{II} Opening Doors^D, 12143^{III} 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^{IV}。

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, 但是这样表示实现乘法和除法的效率不是很高。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, 每四个数字作为一组, 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&);
```

^I 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 $X \times X > Y$, 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。

^{II} 10606 Opening Doors。可参阅 10110 Light More Light。

^{III} 12143 Stopping Doom's Day。计算 $n \in [1, 50]$ 时的 T 值, 观察总结规律。

^{IV} 参阅: <https://docs.oracle.com/javase/7/docs/api/java/math/BigDecimal.html>, 2020。

```

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&);

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，每四位作为一个数位组，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 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地处理。  
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，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，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的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 的结果, 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^I。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 , 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)

^I 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。

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^I Base Equality^C。

扩展练习: 11952^{II} Arithmetic^D。

^I 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 $18480_{10}/9240_{10}=2=c$ 。

^{II} 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。

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 MMMMM 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的实现判断。

```
//++++++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
};
```

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

$$\frac{0}{1}, \frac{1}{1}, \frac{1}{0}$$

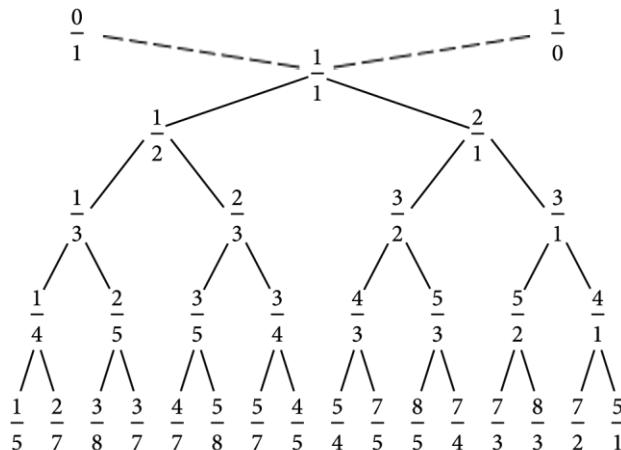
然后下一步将得到两个新分数

$$\frac{0}{1}, \frac{1}{2}, \frac{1}{1}, \frac{2}{1}, \frac{1}{0}$$

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:



构造过程是保序的,因此同一个分数不会在多个地方出现。事实上,我们可以把 Stern-Brocot 树看成一个表示有理数的代数系统,因为每个正的最简分数在这棵树中恰好出现一次。我们用字母“L”和“R”分

别表示从树根开始的一步“往左走”和“往右走”，则一个 L 和 R 组成的序列唯一地确定了树中的一个位置。例如，LRRL 表示从 $\frac{1}{1}$ 往左走到 $\frac{1}{2}$ ，然后往右走到 $\frac{2}{3}$ ，再往右走到 $\frac{3}{4}$ ，最后往左走到 $\frac{5}{7}$ 。我们可以把 LRRL 看作 $\frac{5}{7}$ 的一种表示方法。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
RRLRRRLRLRLRRR

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^[38]：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。

扩展练习: 10437 Playing with Fraction^E, 11968^I In the Airport^D, 12464^{II} Professor Lazy Ph.D.^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: 11113 Continuous Fractions^D, 10521 Continuously Growing Fractions^E。

^I 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。

^{II} 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: $Q_0=\alpha$, $Q_1=\beta$, $Q_2=(1+\beta)/\alpha$, $Q_3=(1+\alpha+\beta)/(a\beta)$, $Q_4=(1+\alpha)/\beta$, $Q_5=\alpha$, $Q_6=\beta$, 循环周期为 5。

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 存储小数位数, 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： $7/22 = 0.3181818\cdots$ ，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 f 可以表示为

$$f = \frac{10^{k+j} \times x - 10^k \times x}{10^{k+j} - 10^k}$$

可以容易地将其实现为代码。

```
-----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，可以使用类似的方法，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

由于计算机采用的是浮点小数来表示数学中的实数，存在精度误差，即有些数字不能够精确的表示，这样在比较实数的大小时，不能够简单的使用数学中的大于小于来比较，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这个阈值时，可以认为它们相等^[39]。

```
-----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。

扩展练习：697^I Jack and Jill^D，11816^{II} HST^D。

5.6 代数

5.6.1 多项式运算

多项式展开

多项式展开 (polynomial expansion) 和整数的乘法类似，将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。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。如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];
}

```

^I 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。

^{II} 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，将所有运算转换为整数进行。

```

    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

因式分解 (polynomial factorization) 和多项式相乘恰好相反, 其目的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如下形式

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0, \quad a_i \in \mathbb{Z}, \quad 0 \leq i \leq n$$

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 , 则可以将多项式表达为

$$a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x^1 + a_0 = (x - r) Q(x)$$

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

在解二元、三元一次方程组时, 常用的是加减消元法或代入消元法, 例如以下的三元一次方程组

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & (1) \\ 2x_1 + 3x_2 + x_3 = 12 & (2) \\ 7x_1 - x_2 + 2x_3 = 0 & (3) \end{cases} \quad (5.1)$$

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

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & (1) \\ -x_2 - 9x_3 = -48 & (2) \\ -15x_2 - 33x_3 = -210 & (3) \end{cases} \quad (5.2)$$

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

$$\begin{cases} x_1 + 2x_2 + 5x_3 = 30 & (1) \\ -x_2 - 9x_3 = -48 & (2) \\ 102x_3 = 510 & (3) \end{cases} \quad (5.3)$$

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, 不断的将未知元的系数化为零。为了使得消元过程书写简便, 可以将线性方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \cdots \cdots \cdots \cdots \cdots \cdots \cdots \cdots \\ a_{m1}x_1 + a_{m2}x_2 + \cdots + a_{mn}x_n = b_m \end{cases} \quad (5.4)$$

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形数表

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} & b_1 \\ a_{21} & a_{22} & \cdots & a_{2n} & b_2 \\ \vdots & \vdots & & \vdots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} & b_m \end{bmatrix} \quad (5.5)$$

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。为了更为方便的描述高斯消元法和后续代码的实现, 先给出矩阵的定义^[40]。

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, 并括以圆括号的数表

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \quad (5.6)$$

称为数域 F 上的 $m \times n$ 矩阵, 通常用大写字母记做 A 或 $A_{m \times n}$, 其中的 a_{ij} 称为矩阵 A 的第 i 行第 j 列元素, 当 $a_{ij} \in \mathbb{R}$ (实数域) 时, A 称为实矩阵; 当 $a_{ij} \in \mathbb{C}$ (复数域) 时, A 称为复矩阵。当 $m=n$ 时, 称 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 (A, b) , 其中由未知元系数排成的矩阵 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

^I 相应的还有数值稳定性更强的全主元高斯消元法。

$$\begin{cases} x_1 + x_2 + x_3 + x_4 = 10 & ① \\ 3x_1 - 2x_2 - x_3 + 6x_4 = 20 & ② \\ 4x_1 - 3x_2 - x_3 + 2x_4 = 3 & ③ \\ x_1 - x_2 + 7x_3 + 5x_4 = 40 & ④ \end{cases} \quad (5.7)$$

线性方程组 (5.7) 的增广矩阵为

$$(\mathbf{A}, \mathbf{b}) = \left[\begin{array}{cccc|c} 1 & 1 & 1 & 1 & 10 \\ 3 & -2 & -1 & 6 & 20 \\ 4 & -3 & -1 & 2 & 3 \\ 1 & -1 & 7 & 5 & 40 \end{array} \right] \quad (5.8)$$

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

$$(\mathbf{A}, \mathbf{b}) = \left[\begin{array}{cccc|c} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & \frac{3}{4} \\ 3 & -2 & -1 & 6 & 20 \\ 1 & 1 & 1 & 1 & 10 \\ 1 & -1 & 7 & 5 & 40 \end{array} \right] \quad (5.9)$$

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

$$(\mathbf{A}, \mathbf{b}) = \left[\begin{array}{cccc|c} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & \frac{3}{4} \\ 0 & \frac{1}{4} & -\frac{1}{4} & \frac{18}{4} & \frac{71}{4} \\ 0 & \frac{7}{4} & \frac{5}{4} & \frac{1}{2} & \frac{37}{4} \\ 0 & -\frac{1}{4} & \frac{29}{4} & \frac{9}{2} & \frac{157}{4} \end{array} \right] \quad (5.10)$$

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

$$(\mathbf{A}, \mathbf{b}) = \left[\begin{array}{cccc|c} 1 & -\frac{3}{4} & -\frac{1}{4} & \frac{1}{2} & \frac{3}{4} \\ 0 & 1 & \frac{5}{7} & \frac{2}{7} & \frac{37}{7} \\ 0 & \frac{1}{4} & -\frac{1}{4} & \frac{18}{4} & \frac{71}{4} \\ 0 & -\frac{1}{4} & \frac{29}{4} & \frac{9}{2} & \frac{157}{4} \end{array} \right] \quad (5.11)$$

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

$$(\mathbf{A}, \mathbf{b}) = \left[\begin{array}{cccc|c} 1 & 0 & \frac{2}{7} & \frac{5}{7} & \frac{45}{14} \\ 0 & 1 & \frac{5}{7} & \frac{2}{7} & \frac{37}{7} \\ 0 & 0 & -\frac{3}{7} & \frac{31}{7} & \frac{115}{7} \\ 0 & 0 & \frac{52}{7} & \frac{32}{7} & \frac{284}{7} \end{array} \right] \quad (5.12)$$

继续此过程, 最后矩阵可以变换为

$$(A, b) = \left[\begin{array}{cccc|c} 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 2 \\ 0 & 0 & 1 & 0 & 3 \\ 0 & 0 & 0 & 1 & 4 \end{array} \right] \quad \begin{array}{l} (1) \\ (2) \\ (3) \\ (4) \end{array} \quad (5.13)$$

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：

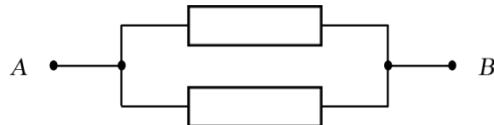


当两个电阻以串联的方式连接时：



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，它才会影晌总的等效电阻值。

(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^[41]，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

^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。

$$\sum_{i=1}^k \frac{V_i - V_j}{R_{ij}} = 0$$

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];

// 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。

扩展练习：10109^I 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` 表示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

^I 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。

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，故 $m = \log_{10}\left(\frac{2^x}{10^k}\right) = x\log_{10}2 - k$ 为无理数，而 x 和 k 可取任意正整数值，不论 m 为何值，通过调整 x 和 k 的值都能得到 m ，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。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

$$M_n = \frac{C_{n-1}^0 d_1 + C_{n-1}^1 d_2 + \dots + C_{n-1}^{n-2} d_{n-1} + C_{n-1}^{n-1} d_n}{2^{n-1}} = \frac{\sum_{i=1}^{n-1} C_{n-1}^{i-1} d_i}{2^{n-1}}$$

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 * di;
            }
        }
        cout << "Case #" << c << ": " << fixed << setprecision(3) << mean;
    }
}
```

```

        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。

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 , 设整数 $n = \sqrt{d}$, 最大步数为 n 步时能达到的距离是 $1 + 3 + \dots + (n-1) + n + (n-1) + \dots + 3 + 1 = n^2$ 。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, 880 Cantor Fractions^B, 860 Entropy Text Analyzer^C, 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^I Car^A, 11970 Lucky Numbers^B。

扩展练习: 11692^{II} Rain Fall^D, 11714 Blind Sorting^C, 11847 Cut the Silver Bar^C, 12027 Very Big Perfect Squares^D。

^I 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。

^{II} 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)。